

Contributions to computer arithmetic and applications to embedded systems

Nicolas Brunie

▶ **To cite this version:**

Nicolas Brunie. Contributions to computer arithmetic and applications to embedded systems. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2014. English. NNT : 2014ENSL0894 . tel-01078204

HAL Id: tel-01078204
https://theses.hal.science/tel-01078204

Submitted on 28 Oct 2014

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

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

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Discipline : Informatique

Laboratoire de l'Informatique du Parallélisme

École Doctorale InfoMaths (ED 512)

présentée et soutenue publiquement le 16 mai 2014

par Monsieur Nicolas BRUNIE

Contribution à l'arithmétique des ordinateurs et applications aux systèmes embarqués

---

Directeur de thèse : M. Florent de DINECHIN

Devant la commission d'examen formée de :

M. Renaud AYRIGNAC, Kalray, Examinateur

M. Laurent-Stéphane DIDIER, Université du Sud Toulon Var, Examinateur

M. Florent de DINECHIN, INSA Lyon, Directeur

M. David LUTZ, ARM, Rapporteur

M. Arnaud TISSERAND, Laboratoire IRISA, Rapporteur

M. Lionel TORRES, Université Montpellier 2, Examinateur

Contributions to computer arithmetic in embedded systems

a hardware/software co-design story

Thanks

À mes parents et mes frères, à ma famille, à mes amis, à mes collègues,

Ce travail n'aurait jamais pu être mené à bien sans le suivi et les conseils éclairés de mon directeur de thèse : Florent de Dinechin, et de mes encadrants à Kalray : Renaud Ayrignac et Benoît de Dinechin.

Je souhaite aussi particulièrement remercier les deux relecteurs de cette thèse : Arnaud Tisserand et David Lutz, ainsi que les autres membres de mon jury : Laurent-Stéphane Didier et Lionel Torres.

Il me serait difficile de remercier séparément toutes les personnes ayant contribué à cette thèse. J'ai eu la chance de rencontrer tant de gens passionnés et passionnants, je leur serai toujours reconnaissant. Merci à Sylvain Collange et Marius Cornea ainsi qu'à mes collègues de l'équipe Arénaire/Aric, de Kalray, du LIP et d'Intel.

Contents

1 Introduction **19**
- 1.1 Embedded architectures . . . 19
- 1.2 Exploiting arithmetic parallelism . . . 21
- 1.3 Versatility versus efficiency in arithmetic . . . 22
- 1.4 Outline of this thesis . . . 23
- 1.5 Kalray's MPPA . . . 24
  - 1.5.1 Architecture of the MPPA cluster . . . 25
  - 1.5.2 Architecture of the K1 core . . . 25

I Contributions to Floating-Point Arithmetic **29**

2 Context and state of the art **31**
- 2.1 The IEEE-754 floating-point standard . . . 31
  - 2.1.1 Floating-point formats . . . 31
  - 2.1.2 Rounding modes and exception . . . 32
- 2.2 Unit in the last place . . . 33
- 2.3 State of the art of floating-point support in embedded processors . . . 33
  - 2.3.1 ARM architecture . . . 33
  - 2.3.2 ST231 . . . 33
  - 2.3.3 IBM's Power architecture . . . 34
  - 2.3.4 Texas Instrument TMS320 . . . 34
  - 2.3.5 Conclusion . . . 34

3 Hardware floating-point design **37**
- 3.1 Introduction . . . 37
- 3.2 Fused-Multiply and Add state of the art . . . 38
  - 3.2.1 Single-path FMA . . . 39
  - 3.2.2 Multiple-path FMA . . . 43
  - 3.2.3 Exotic FMA designs . . . 46
- 3.3 Building K1's FPU: architectural study . . . 48
  - 3.3.1 Mixed-precision FMA . . . 48
  - 3.3.2 2D dot-product . . . 56
  - 3.3.3 Triple-operand add . . . 60
  - 3.3.4 Conclusion on K1's first FPU structure . . . 64
- 3.4 Building K1's FPU: miscellaneous contributions . . . 64
  - 3.4.1 Automatic generation of bit-pattern detector . . . 64
  - 3.4.2 Management of subnormal numbers . . . 66
  - 3.4.3 Multiplier tiling . . . 67

3.4.4 Iterative process of extended multiplication . . . 68
3.5 Testing, performance and perspectives . . . 70
3.5.1 Operator testing . . . 70
3.5.2 Summary and conclusion . . . 71

4 Software Floating-Point stack **73**
4.1 Implementation error . . . 73
4.2 Optimized low-level floating-point primitives: division and square root . . . 76
4.2.1 Newton-Raphson iteration . . . 77
4.2.2 Binary32 iteration implementation . . . 78
4.2.3 Code example and error study . . . 81
4.2.4 ISA extension to improve multiple rounding modes and correct exception support . . . 83
4.2.5 Performance result . . . 84
4.3 Introduction to the development of mathematical functions . . . 85
4.3.1 Exponential evaluation scheme . . . 86
4.3.2 Implementation and error analysis . . . 88
4.3.3 Performance results and conclusion . . . 90

5 Automated code generation of mathematical functions **93**
5.1 Introduction . . . 94
5.1.1 State of the art . . . 94
5.1.2 Objectives and philosophy of our project . . . 95
5.2 Arithmetic support: PythonSollya . . . 96
5.3 Description language . . . 97
5.3.1 Annotation system . . . 99
5.3.2 Constants, tables . . . 101
5.3.3 Polynomials . . . 102
5.3.4 Abstract type . . . 104
5.4 Metalibm support libraries: miscellaneous and multi-precision . . . 104
5.5 Metalibm core: internal representation and optimization . . . 105
5.5.1 Format determination . . . 106
5.5.2 Operand-flavour selection and interval determination . . . 107
5.5.3 Scalar vectorizer . . . 107
5.6 Code and proof generation . . . 108
5.6.1 Gappa script generation . . . 108
5.6.2 Code generation and target specific processing . . . 109
5.7 Vectorization support . . . 110
5.7.1 Result blending . . . 112
5.7.2 Callout extraction . . . 113
5.7.3 Static vectorization implementation . . . 113
5.8 Application: vectorized correctly rounded exponential and logarithm . . . 115
5.9 Conclusion and perspectives . . . 117

II Architecture and application of a deeply integrated reconfigurable fabric **119**

6 Reconfigurable taxonomy and state of the art **121**
6.1 FPGAs . . . 122
6.1.1 Reconfigurable Logic Cell . . . 122

6.1.2 Routing network . . . 122
6.1.3 Fast arithmetic support . . . 123
6.1.4 Memory block . . . 123
6.1.5 Conclusion: FPGA versus ASIC as computing systems . . . 123
6.2 PiCoGA and the DREAM Digital Signal Processor . . . 124
6.3 TCE Project . . . 126
6.3.1 Transport-triggered architecture . . . 126
6.3.2 TCE configurability . . . 127

7 Deeply Integrated Reconfigurable Fabric 129
7.1 Overall view . . . 129
7.2 Basic cell . . . 131
7.2.1 Cell interfaces . . . 131
7.2.2 Cell micro-architecture after a first round of design exploration . . . 131
7.3 Interconnect network . . . 133
7.3.1 Generalized carry-chain . . . 134
7.3.2 Elementary interconnect node: β-network . . . 135
7.3.3 Crossbar-based architecture . . . 135
7.3.4 Permutation-based networks: introduction and advantages . . . 136
7.4 Contribution to permutation-based Benes networks . . . 139
7.4.1 Using the interconnect network to perform runtime-dynamic shifts and rotations . . . 139
7.4.2 Benes network and rotation: state of the art . . . 141
7.4.3 Realizing rotation . . . 141
7.4.4 Configuration unfolding . . . 143
7.4.5 Hardware implementation and comparison . . . 144
7.4.6 Generalization . . . 145
7.5 Conclusion . . . 148

8 Transport Triggered Co-Processor 149
8.1 Single master coprocessor . . . 149
8.1.1 Coprocessor integration . . . 149
8.1.2 Coprocessor architecture . . . 149
8.1.3 Instruction Set Architecture . . . 151
8.2 Coprocessor integration and shared TTA . . . 151
8.2.1 Shared TTA . . . 152
8.2.2 Input buffer . . . 152
8.2.3 Input arbiter . . . 154
8.2.4 Configurable master id pipeline . . . 154
8.2.5 Output FIFO . . . 155
8.2.6 System sizing . . . 156
8.3 Conclusion . . . 157

9 Reconfigurable Kernel Toolchain 159
9.1 DIRF RTL generation and architectural exploration . . . 160
9.1.1 Reconfigurable matrix parametrization . . . 160
9.1.2 Basic cell parametrization . . . 161
9.2 RKC front-end . . . 162
9.2.1 Intermediate representation . . . 163
9.2.2 Decomposition: technology mapping . . . 166

9.3 RKC Backend: place and route . . . 166
9.3.1 Single level placement . . . 167
9.3.2 Multiple-level placement . . . 168
9.3.3 Routing . . . 169
9.4 DIRFsim: software emulation . . . 169
9.5 Conclusion . . . 169

10 Application case studies **171**
10.1 Advanced Encryption Standard . . . 171
10.1.1 SubBytes . . . 172
10.1.2 ShiftRows . . . 173
10.1.3 MixColumns . . . 173
10.1.4 AddRoundKey . . . 173
10.1.5 Conclusion . . . 173
10.2 Hash function: SHA-1 . . . 173
10.3 Design exploration and results . . . 174
10.4 Conclusion and future work . . . 175

III NIMT Architecture **177**

11 SIMT improvements for divergent control flow **179**
11.1 SIMT architecture . . . 180
11.1.1 Management of multiple branches: mask stack . . . 181
11.1.2 Dynamic Warp Formation . . . 183
11.1.3 Thread frontier . . . 184
11.2 Towards NIMT architecture . . . 184
11.2.1 Multiple Fetch units . . . 185
11.2.2 Enforcing earliest possible reconvergence: min-PC scheduling policy . . . 186
11.2.3 Simultaneous Branch Interweaving . . . 187
11.2.4 Simultaneous Warp Interweaving . . . 188
11.2.5 Hot and cold context tables: providing minPC list with easy access . . . 188
11.3 Evaluation . . . 190
11.4 Conclusion and perspectives . . . 191

IV Conclusion and perspectives **193**

12 Conclusion and future works **195**

V Appendix **197**

13 Introduction to correctly rounded implementation **199**

14 Useful basic blocks for FPU design **201**
14.1 Synthesis results . . . 201
14.1.1 Integer adder . . . 201
14.1.2 Leading zero counter . . . 202
14.1.3 Logic shifter . . . 203
14.1.4 Multiplier . . . 204

14.2 Analysis and conclusion 205

List of Figures

1.1 Taxonomy of current architectures and markets according to power consumption and performance . . . 20
1.2 Comparison of architectures according to their number of cores . . . 23
1.3 MPPA architecture [82] . . . 25
1.4 MPPA's cluster architecture . . . 26
1.5 Kalray's K1 core architecture . . . 27

3.1 FMA simplified micro-architecture. The internal critical path is indicated by boxed arrows . . . 39
3.2 Comparison leading zero count mechanisms on addition: LZC versus LZA . . . 40
3.3 Last-level carry-select adder structure (*Left*). Compound adder structure (*Right*) . . . 43
3.4 FMA multi-path architecture . . . 44
3.5 Comparison of latencies: FMA versus paired FADD/FMUL . . . 46
3.6 IBM High and Low part FMA overview . . . 47
3.7 Mixed precision FMA overview . . . 50
3.8 MPFMA simplified architecture . . . 51
3.9 MPFMA datapath layout, and possible leading digit positions, for product-anchored cases, $e_{product} - e_{addend} > 2p - 1$ . . . 52
3.10 MPFMA datapath layout, and possible leading digit positions, for addend-anchored cases, $e_{product} - e_{addend} < -2$ . . . 52
3.11 MPFMA datapath layout, and leading digit possible positions, for cancellation cases, $-2 \leq e_{product} - e_{addend} \leq 2p - 1$ . . . 53
3.12 Two solutions for product pre-normalization in the MPFMA . . . 54
3.13 2D dot-product operation scheme . . . 58
3.14 2D dot-product micro-architecture (main differences with FMA32 are highlighted in red) . . . 59
3.15 Triple-operand addition operation scheme . . . 60
3.16 Triple-operand micro-architecture . . . 62
3.17 Alignment scheme for triple-operand add . . . 63
3.18 Tiling of 16×32 and 11×27 multipliers used to implement K1 operations . . . 68
3.19 Iterative binary64 multiplication effect on K1's pipeline . . . 69

4.1 Correlation between function implementation speed and numerical quality . . . 74
4.2 Newton-Raphson method . . . 77

5.1 Operation graph of MDL description reproduced in Listing 5.2 . . . 98
5.2 Metalibm's process overview . . . 106
5.3 Example of Super Word Level Parallelism extraction . . . 107
5.4 Metalibm architecture-description hierarchy . . . 109

5.5 Scalar operation scheme (left) comparison to vector operation scheme (right) 111
5.6 Example of blending result reduction effect on an operation graph 112
5.7 Example of core, validity flag and callout extraction 114

6.1 ASIC, Reconfigurable and Software ordered according to flexibility, design effort and performance 121
6.2 LUT and registered/direct output 122
6.3 General architecture of an FPGA matrix 124
6.4 The DREAM architecture (figure extracted from [23]) 125
6.5 Example of transport-triggered architecture 126

7.1 Reconfigurable matrix overview 130
7.2 Reconfigurable Logic Cell interfaces and micro-architecture 131
7.3 The two operation modes of the i-th 8×2 LUTs of a RLC 133
7.4 RLC generalized carry chain 134
7.5 Interconnect junction point 135
7.6 An example of crossbar-based interconnect architecture 136
7.7 Example of crossbar-based routing configuration 136
7.8 Two types of half 16-input Benes networks: **butterfly**(left) and **unshuffle**(right). . . . 137
7.9 Full butterfly-based 8-input Benes network 138
7.10 Full shuffle-based 8-input Benes network 139
7.11 Differences between runtime-static and runtime-dynamic rotation modes 140
7.12 Example of configuration unfolding for rotation in shuffle-based Benes network . . 143
7.13 Extension of shuffle-based Benes network for odd sizes 145

8.1 Reconfigurable coprocessor architecture 150
8.2 Architecture for TTA operator virtual sharing 153
8.3 Structure of an input register 154
8.4 Example of multiple masters accessing the coprocessor with input buffering and arbitration towards the DIRF (master i: formatting register of the i-th master, IB: input buffer ready register, C i: DIRF i-th pipeline stage). 155
8.5 Example of multiple masters accessing the coprocessor with output FIFO buffering and shared pipeline steps (M i: data originating from Master i). 156

9.1 Steps of the reconfigurable kernel toolchain 159

11.1 Single-Instruction Multiple-Thread architecture 180
11.2 Example program execution on SIMT architecture with mask-stack 182
11.3 Example of kernel program, *tid* is the thread identifier (unique for each thread in a warp). 182
11.4 Dynamic Warp Formation with thread migration 183
11.5 *Lane-aware* Dynamic Warp Formation 184
11.6 Example of thread frontier 185
11.7 N-Instruction Multiple-Thread architecture for N=2 186
11.8 Reconvergence instruction and min-PC policy for the example program 187
11.9 Example program execution with SBI 188
11.10 Slowdown of various set-associativity with respect to fully associative friendly warp lookup tables 189
11.11 Structure and relations of the context tables 189
11.12 Performance of SBI, SWI, and combination of SBI and SWI, with a thread-frontier based 64-wide warp implementation for reference. 190

13.1 Illustration of hard-to-round case condition 199

14.1 Adder area with respect to latency and width 202
14.2 Leading Zero Count area with respect to latency and width 203
14.3 Logic shift area with respect to latency and width 204
14.4 Multiplier area with respect to latency and width 205

List of Tables

2.1 IEEE floating-point format . . . 31

3.1 FLIP: binary32 performance results [79] . . . 37
3.2 ARM Cortex-A9: binary32 performance results . . . 38
3.3 Multiple FMA designs synthesis results for 28nm high density technology. . . . 55
3.4 Comparison of mantissa multiplier area depending on mantissa format . . . 55
3.5 Latency/Throughput for low dimension scalar products with and without FDMA (2D dot product instruction) . . . 57
3.6 Shift area according to width and latency constraint (28nm process) . . . 58
3.7 Summary of K1 FPU performance . . . 72

4.1 Binary32 implementation of $\frac{1}{b}$ based on Newton-Raphson algorithm . . . 78
4.2 Binary32 implementation of $\frac{1}{\sqrt{a}}$ based on Newton-Raphson algorithm . . . 79
4.3 Code size for K1 binary32 approximation table and division function without table . 80
4.4 Latencies for Newton-Raphson initial approximation loading from memory, in K1 core . . . 80
4.5 Synthesis result for hard-coded approximation tables and seed operators . . . 81
4.6 Comparison of K1 division implementation with and without RN instructions . . . 84
4.7 Performance of several implementations of binary32 reciprocal, division and square root . . . 85
4.8 Performance comparison between Horner's and Estrin's evaluation scheme for the polynomial p of degree 6 (L_{FMA} is the latency of an FMA operation) . . . 90
4.9 Performance results for binary32 exponential implementations . . . 90

5.1 Accuracy requirement for correctly-rounded implementation . . . 104
5.2 Performance results of scalar function generated by our metalibm and compared to manual implementations . . . 116
5.3 Performance results for vectorized correctly rounded exponential and logarithm on several Intel's architectures . . . 116

7.1 Synthesis results for shuffle-based and butterfly-bases Benes networks, with and without dynamic rotation support . . . 144

9.1 Synthesis result comparison various multiplication by constant algorithms implemented in RKC . . . 165

10.1 Comparison of interconnect efficiency on AES . . . 175
10.2 Comparison of RLC structure effects on DIRF size requirement for the AES kernel . 175
10.3 DIRF/ASIC area comparison for the same function . . . 175

11.1 Area overhead SBI, SWI and SBI+SWI components 191

14.1 Integer adder synthesis results 201
14.2 Leading Zero Counter synthesis results 202
14.3 Logic Shifter synthesis results 203
14.4 Multiplier synthesis results 204

List of Code Listings

3.1 binary64 accumulation of binary32 operand products . . . 48
3.2 Sum-of-product of binary32 operands in binary64 precision . . . 56
4.1 K1's software implementation of binary32 reciprocal core based on Newton-Raphson method . . . 81
4.2 Gappa proof for the error of our implementation of Newton-Raphson method for the binary32 reciprocal . . . 82
4.3 PythonSollya script used to determine p^* and ϵ_{approx}. . . . 87
4.4 Gappa script for the error evaluation of exponential argument reduction . . . 88
5.1 Some examples of PythonSollya calls into the Python interpreter . . . 96
5.2 Example of function implementation using Metalibm's description language . . . 97
5.3 Generation of a bi-partite approximation to log(2) with PythonSollya, for use in exponential argument reduction . . . 101
5.4 Example of Table creation, initialization and use . . . 102
5.5 Example of Gappa script generated by metalibm for the computation of the evaluation error of a polynomial in fixed-point . . . 103
5.6 Using backend pass to instantiated undetermined precision . . . 106
5.7 Example of architecture description, deriving from **GenericProcessor** . . . 110
9.1 Example of array configuration file used for an AES kernel . . . 160
9.2 Example of basic cell configuration file . . . 161
9.3 Example of VHDL-like source file for RKC . . . 163
9.4 Example of C-like source file for RKC . . . 163
9.5 Example of program containing a multiplication by a constant . . . 165
10.1 SHA1 code kernel . . . 174

1 Chapter 1

Introduction

Embedded computing used to be the computing that does not show itself, hidden inside systems (appliances, automotive, aerospace) where computation was only one component among many others. The main asset of embedded architecture was to be easily forgotten while providing sufficient computation capabilities. They have extended far beyond this initial outlet and reached markets where they are much more visible, conquering a wider audience. For example, every handheld device (mobile phones, smartphones, MP3 players) relies on a embedded processor for its computation needs. A more accurate and up-to-date definition of embedded computing could be computing with constraints: low power, low cost, integrated into a more complex system and sometime invisible. This evolution is still on-going, but already embedded systems success goes far beyond the embedded market: embedded architectures are almost everywhere, from the no-contact radio-frequency tags to low-power servers and laptops. This success came along growing application requests: embedded architectures are expected to provide increasing computation capabilities. Arithmetic, the science of calculus, is at the center of this challenge. Efficient and versatile arithmetic support is both a reason of this success and a requirement for future designs.

1.1 Embedded architectures

Embedded systems start with micro-controller: low power/ low capabilities processors embedded into systems where energy consumption is at least as important as peak performance. This market, very low power, represents the low left corner of the power / performance taxonomy presented in Figure 1.1. A good representative of that market is the TI MSP430, a very low power 16-bit micro-controller. It is very successful in application markets such as RFID tags and sensors which benefit from its capability to perform low intensity computations on very limited power resources. Such energy efficiency contributed to the development of more complex embedded systems such as handheld devices (eg: mobile phones, mp3 players, portable gaming consoles and more recently smartphones and tablets). Those products have known a large success in the general public and have become the main outlet for embedded processors. Those systems rely on stronger energy supplies but also require higher computation capabilities. They created a demand for more powerful embedded architectures. The middle and high-end embedded systems have evolved to supply this demand: they have become more capable and more versatile. Energy efficiency is still key, but peak performance comes close second. ARM architectures have conquered this market. Designed by Acorn (which became the fabless ARM Holdings), this RISC instruction-set architecture has known many extensions (VFP, Thumb, NEON, ...) since its initial design in 1985. It is implemented in low-power processors built by numerous partners (Apple,

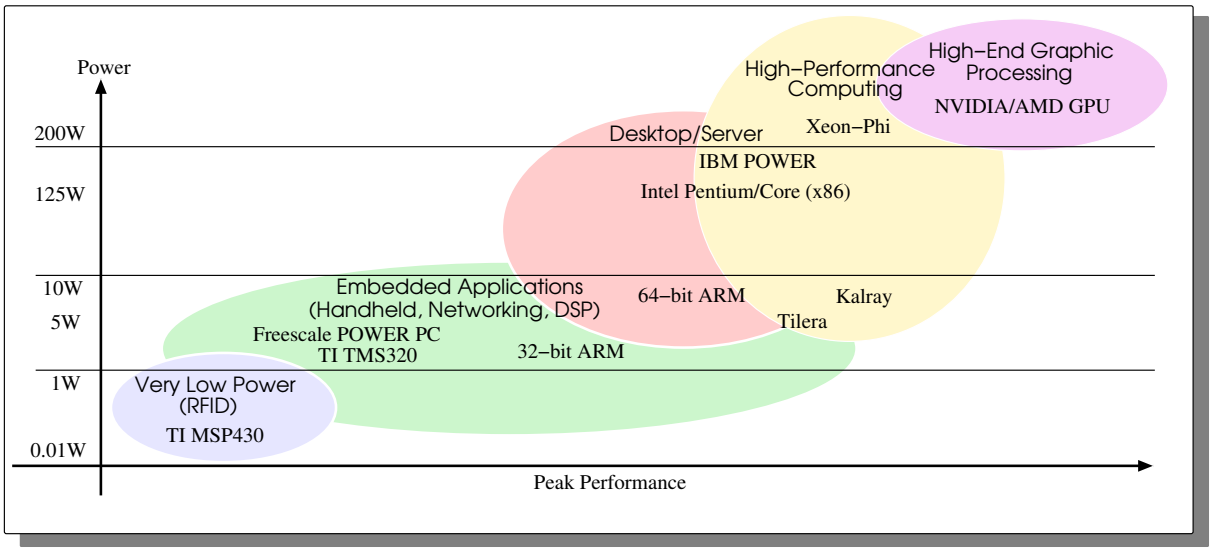


Figure 1.1: Taxonomy of current architectures and markets according to power consumption and performance

Samsung, Qualcomm, Nvidia, ...) and is nowadays part of almost every smartphone and tablet. Other successful architectures include embedded versions of IBM's Power architecture such as Freescale PowerPC e6500 or custom VLIW architectures such as Texas Instrument TMS 320.

As low-power processors gained in performance, the other end of the computer market (desktop computers, data-centers and High Performance Computing servers [HPC]), located on the top right-hand corner of Figure 1.1, moved towards lower power consumption. For long, these markets relied on very powerful systems, providing much more peak performance that any embedded architecture could provide. Recent examples of such systems include Intel's Pentium, Core and Xeon processors, IBM POWER, HP PA-Risc then Itanium, SPARC (Sun then Fujitsu). But those systems were not very power efficient: silicon area and power were used to reduce latency. Finally this market hit the power wall. Power concern became critical and high-end architecture started to evolve towards more energy efficiency. In the meantime, performance of embedded architectures kept increasing, without impacting their energy efficiency, making them already interesting for these markets. In the last years, architectures originating from embedded system designs have started to conquer the low-end part of the server/data-center and laptop markets. The new ARM-v8A 64-bit architecture is an example of embedded architecture evolution which interests the low-end server market. Embedded architecture are also considered serious candidates for the HPC challenge of the decade: exaflops capabilities under 20MW [141]. Nowadays, embedded systems are everywhere, from RFID tags to cars. Embedded architectures are spreading well passed their native market.

Because of this, embedded architectures have been expected to support a growing set of applications. This set, first limited to low-power computation-light tasks, has been extended to intensive tasks such as cryptography, digital signal processing, graphics processing, advanced operating systems. Many of those applications are computation intensive: they require a large number of operations per time unit (and per energy unit). The G(Fl)ops.s^{-1}.W^{-1} expected from an embedded architecture has been increasing continuously as the market requires more multitasking, smoother interfaces, nicer graphics and overall more capabilities inside a still limited power budget. Those operations are also very diverse. Cryptography algorithms, such as encryption/decryption or hashing, rely on intensive arithmetic (integer, finite fields, modulo arithmetic,

bit-wise logic). Digital Signal Processing (DSP) and graphic processing are intensive in fixed-point and floating-point computations (linear algebra, mathematical functions). To meet these application needs, embedded architectures have to provide efficient and versatile arithmetic. It is one of the keys to market success.

1.2 Exploiting arithmetic parallelism

When the targeted applications hold enough parallelism there is a straightforward way to improve arithmetic efficiency: increase the number of arithmetic units per embedded chip. Adding more units is facilitated by exploiting Moore's law [109]: the number of transistor on a chip doubles approximately every two years. With each new generation of fabrication process it becomes clearer that this law is coming to an end, but it stayed unchallenged until now. What is certain is that the power dissipation is caped and leakage is increasing as transistors become smaller. In any case, newly available transistors can be used to implement more arithmetic units. We can distinguish three ways to do so: SIMD architectures, superscalar architectures and multicores. Both SIMD and superscalar architectures consist in increasing the number of units per computing core. Superscalar architecture adds more independent units to the core. Independents units are more flexible, they do not necessitate compiler support (eg: hardware-managed parallelism in superscalar out-of-order architectures). SIMD architectures add vector units to the core. These architectures trade some of the superscalar flexibility for better energy efficiency, since part of the control is shared across a vector.

The third way is to increase the number of cores in a circuit, automatically increasing the number of arithmetic units. This trend was illustrated by the apparition of multi-core CPUs in general public products (eg: Intel's Pentium D, IBM's Power 4, AMD's Opteron). Those general purpose processors (GPP) are built with high-complexity cores: CISC architecture, overabundant instruction set, superscalar out-of-order, high frequency. As the power consumption is directly related to a core complexity and frequency, those multi-cores were never interesting for the embedded market. There, this multicore trend echoed slightly differently. To improve a system efficiency, increasing the number of cores while keeping them simple appears more efficient than using fewer, faster, and more complex cores. Following this idea, embedded systems were the first to welcome manycore processors. Manycore processors pack several tens of rather simple cores on a single silicon die. Some foresee that we should reach several thousands of independent core in a single die before the end of the decade. Current GPUs already reached that number with processing elements paired through the Single Instruction Multiple Threads architecture (eg: NVIDIA's Maxwell architecture). As illustrated by Figure 1.2, manycores represent a better balance between power efficiency, flexibility and peak performance than GPUs. They outperform single and multicore CPUs in both power efficiency and peak performance. This makes manycores more interesting to meet the versatility requirements of the embedded market.

Several manycore architectures have already been released. Tilera, a company founded in 2004, was among the first to offer a manycore: the TILE architecture. This architecture offered up to 64 cores embedded in a mesh network. Each core integrates two integer ALUs, a Load/Store unit, two levels of cache plus a router to access the network on chip (NoC) but no floating-point support. In 2009, the new Tile-GX architecture made up for this absence. Each TILE-Gx core integrates a 32/64-bit floating-point unit. The architecture has been implemented in several chip models, containing between 9 and 72 cores.

Adapteva, a small start-up founded in 2008, is also about to launch its Parallela board using its own Epipahny chip [5]: a scalable manycore implementing a RISC proprietary Instruction Set Architecture (ISA).

Small companies are not the only one to consider manycore architectures. Intel, better known

for its desktop/laptop and server processors (Pentium, Core, Xeon), also designed its own manycore: the Xeon-Phi coprocessor [74]. It implements Intel's Knight architecture (declined as Knight Landing, Knight Corner ...) with 61 improved x86 cores (modified version of the original Pentium core). Each of them contains a 512-bit wide vector unit able to process up to 16 binary32 operations in parallel. As a result, considering each vector lane as a processing element, the Xeon-Phi can also be seen as containing 976 processing elements (PEs). In June 2013, Intel took back the head of the Top 500 (500 more efficient supercomputers) with the Tianhe-2 supercomputer built around Ivy-Bridge processors and Xeon-Phi coprocessors. A new iteration of the product, called Knight Landing, was announced, also in June 2013, for a future release. It will use a 14nm process and will contain 72 cores derived from the Atom architecture.

Kalray, a French start-up, develops its own architecture: the Multi-Purpose Processor Array (MPPA) [40], a clustered manycore designed, among other markets, for embedded computing. Kalray's MPPA will be described in more details in Section 1.5. This thesis describes work which was made during and towards the development of MPPA's processing element: the K1 core.

SIMD, superscalar and manycore architectures can be mixed and matched to better suit silicon/power budget and application requirements. Intel's Xeon-Phi is a good example: a manycore with vector units in each core.

The other side of the arithmetic challenge is versatility. Managing versatility while providing efficiency can be difficult: different application domains often require distinct arithmetic supports.

Embedded systems often provide adaptable support: a core architecture can be augmented with instruction-set extensions to improve specific arithmetic support. For example, ARM extended its ISA with floating-point extensions: VFP and NEON. Those extensions provide floating-point capabilities (plus vector capabilities for NEON) to the ARM architecture. They are mandatory in the most recent architecture iteration (ARMv8-A) which illustrates the need for efficient floating-point support. ARM also introduced a cryptography ISA extension to better support AES and SHA algorithms. Texas Instrument offers floating-point oriented version of its TMS320: the C67x series. This architecture implements several floating-point units in each core and transforms the TMS320 from an integer DSP into a very capable floating-point DSP. Nowadays, embedded systems, especially the high-end, have to support simultaneously a very wide range of application domains with as many diverse arithmetic requirements. It is not a good fit for limited hardware customization, which is more suited when the set of application to support is limited. Indeed, supporting every complicated operation with a dedicated hardware unit is too costly both in area and development time. Other ways are used to provide a versatile support: for example highly-tuned software libraries. Those libraries try to exploit to its maximum the available hardware and extend the architecture capabilities: providing mathematical functions, multi-precision arithmetic, cryptography. But software can only bring limited performance, it is always limited in some way by the underlying hardware. Reconfigurable circuits, such as field-programmable gate arrays (FPGAs), present a possible way to circumvent that difficulty and mix hardware efficiency and software flexibility. For now, they did not break through in the general market, but are already being successful in some niche markets.

Finding the most efficient balance between, hardware, software and reconfigurable computing remains a challenge.

1.3 Versatility versus efficiency in arithmetic

The challenge we chose to address in this work is to provide very efficient arithmetic implementations, using hardware, software and reconfigurable circuit. Our goal is to allow embedded systems in general and the MPPA in particular to offer sufficient computing capabilities to address every single need they may encounter.

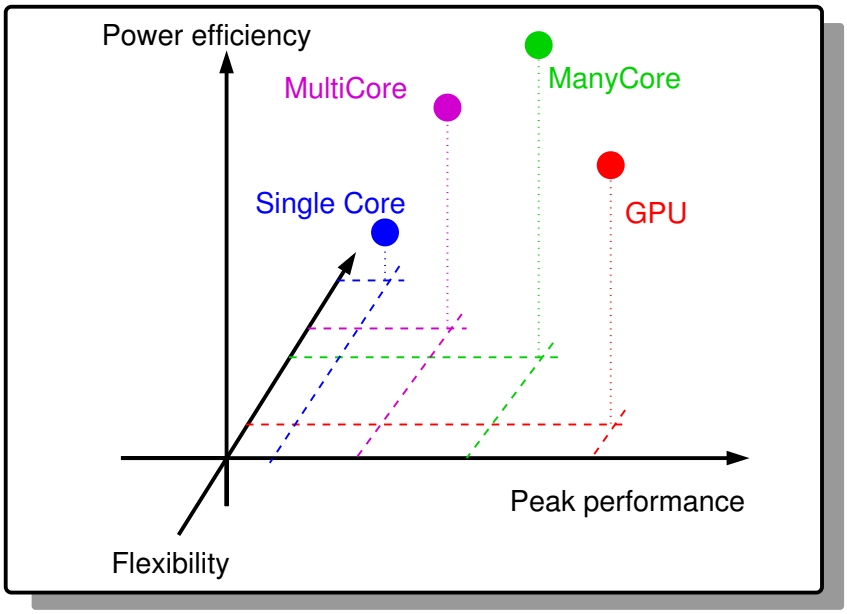


Figure 1.2: Comparison of architectures according to their number of cores

In our context, efficient means: with the required level of performance (operations per second) using the smallest possible amount of resources. In an embedded system, the two most restricted resources are die size (silicon area) and power. However, other considerations have to be taken into account. The first one is time predictability: embedded systems often have to do real-time processing. This implies that operation processing time should be limited and statically determined to assert operational deadlines. We will study this aspect mostly in relation with floating-point computations. The IEEE standard (IEEE-754 rev 2008) that defines floating-point formats and computations introduces many complex cases to be managed. Managing those cases in an acceptable delay appears to be of the highest importance for the system to be certified for correct real-time processing.

This is linked to the second consideration: numerical predictability. Most often, understanding finely the accuracy requirements of an application can lead to more efficient implementations. This is nicely summed-up by the motto "computing just right", which we try to follow when developing floating-point software.

We tackle the challenge of efficient embedded arithmetic through the optic of the hardware/-software co-design: determining which arithmetic elements require a hardware support and to which extent, and at the same time which part can be implemented in software with good enough performance.

1.4 Outline of this thesis

The first part of this thesis studies the implementation of floating-point computing in embedded systems. For the most common floating-point operations, a full hardware implementation is often required. Basic floating-point operations are used so extensively in a wide range of applications that they justify the integration of a dedicated unit in an embedded system CPU (including the development cost). This constitutes the focus of the first chapter of this work: design and implementation of an efficient Floating-Point Unit (FPU) for embedded systems.

Less common floating-point operations cannot justify extensive hardware development in embedded systems. Thus we study the manual development of highly optimized software kernels to support two floating-point primitives: division and square root. We extend this development flow to an even less common primitive: floating-point exponential, chosen as an example of mathematical functions.

Such manual development appears too troublesome when considering the various flavours of the numerous mathematical functions which should be provided to fit various needs (architecture, latency, throughput, accuracy). To address this problem, we suggest a domain-specific framework for implementation automation and code generation of mathematical kernels. The project, called *Metalibm*, provides a specific generator for elementary functions, which processes a common implementation description and generates source code and proof code from an unified intermediate representation. Metalibm tries to use as efficiently as possible the generic floating-point operations provided by the FPU integrated into the processor. This project tries to address the two extra considerations listed previously: time and numerical predictability. On the one hand, we offer the generation of correctly rounded functions which represent the ultimate goal for floating-point predictability at a given precision. On the other hand, we also provide several parametric implementations to decrease the computation accuracy while staying in known bounds and thus we are able to save precious computation time.

The next part of this thesis starts from the following analysis: the hardware/software codesign, that proved successful for floating-point, can not be applied to every single operation that needs to be supported by embedded systems. Some operations (e.g. cryptography primitives) require a performance level close to what hardware can bring without being able to justify specific hardware, except for very dedicated systems. As software implementations lack efficiency, another solution must be studied. We suggest the design of a new reconfigurable fabric to be integrated as a CPU functional unit. The hardware reconfigurability brings close to hardware performance for operations only required intensively by a few applications. We focus our design on latency and area efficiency, studying original architectural possibilities. This project is supported by an architectural exploration framework that includes a parametric RTL description, a configurable compiler and the corresponding simulator.

Finally we extend our horizon and tackle a more architectural problem: improving the use of arithmetic processing elements in GPU-like Single Instruction Multiple Thread (SIMT) architecture. This part focuses on the following challenge: making sure that the system arithmetic resources are used as much as possible. This can prove especially complicated on a system managing hundred of threads at the same time. We suggest new mechanisms to improve GPU efficiency on heterogeneous and branching workloads.

Let us know present Kalray's MPPA architecture, which was used as integration target for most of our contributions.

1.5 Kalray's MPPA

The Multi Purpose Processor Array (MPPA) is a manycore architecture developed by Kalray [2]. The first available product derived from this architecture is the MPPA-256. The MPPA-256 was commercially available by the end of this thesis. At the beginning of this work most of the architecture features were already fixed but no floating-point support was implemented. The floating-point stack (FPU and software primitives) developed during this thesis was integrated into the MPPA-256 and is currently used in production with the product. The reconfigurable fabric was also studied for possible integration in the MPPA architecture but was not selected.

Let us now describe the MPPA-256 architecture, illustrated by Figure 1.3. It contains 256 processing elements (K1 cores) which are organized in 16 clusters of 16 cores. The clusters are con-

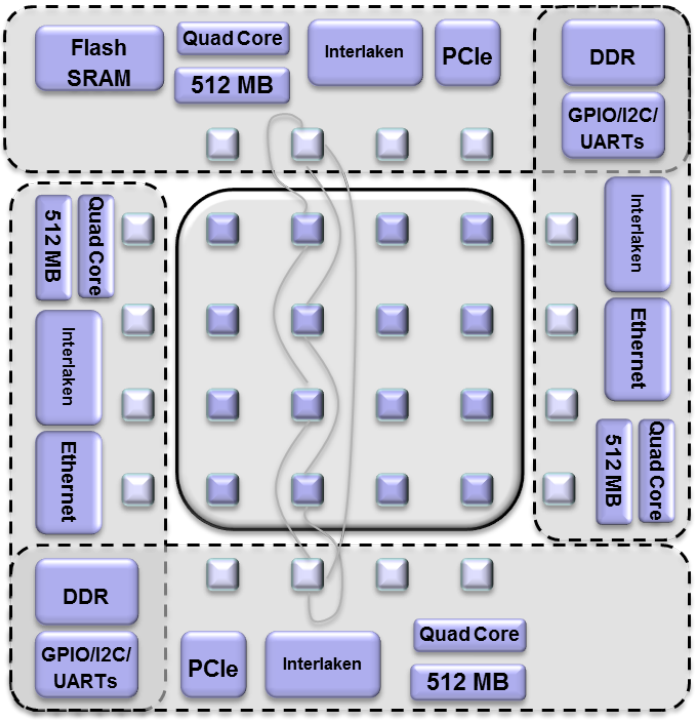


Figure 1.3: MPPA architecture [82]

nected by a Network on Chip (NoC) which also connects them to I/O clusters. I/O clusters are in charge of interfacing the clusters with the exterior through high speed I/Os (Ethernet, PCI-Express, Interlaken) and running an operating system (Linux).

1.5.1 Architecture of the MPPA cluster

The cluster architecture is illustrated by Figure 1.4. Each cluster contains 16 K1 cores plus an extra K1 core called the Resource Manager (RM) which is used to manage the control part of an application. Each cluster has several Mbytes of local shared memory, accessible by the 17 cores and by a Direct Memory Access (DMA) micro-core. The DMA manages accesses to the NoC from and to the cluster; it can also execute local memory moves.

1.5.2 Architecture of the K1 core

Each core has an instruction and a data cache to alleviate memory access latency. It implements a 5-issue VLIW architecture built around:

- a double arithmetic and logic unit (ALU), which manages up to two 32-bit ALU operations, or a single 64-bit ALU operation each cycle;
- a multiply-accumulate unit, which manages the integer MAC operation (which includes simpler multiplication) or a floating-point operation;
- a load-and-store unit, which manages exchanges with the memory system;
- a branch and control unit, which manages branches and control operations (modifying/accessing status registers ...).

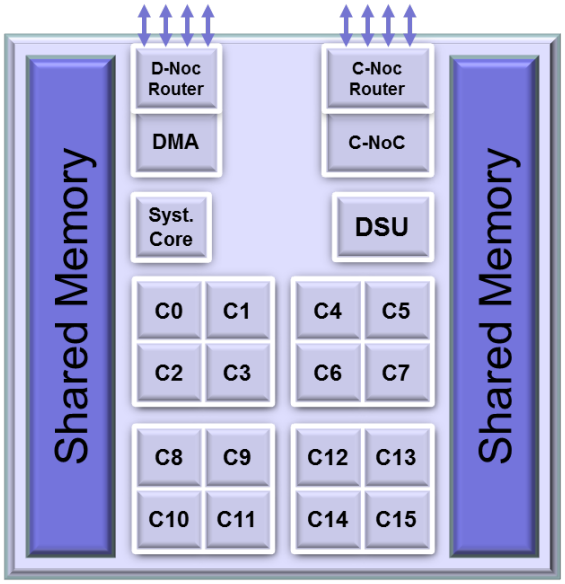


Figure 1.4: MPPA's cluster architecture

Figure 1.5 illustrates the structure of a Kalray K1 core pipeline. PFB stands for prefetch buffer, it is the connection between the instruction cache and the core instruction pipeline. ID is the instruction decode stage. RR is the register read stage. The K1 instruction set architecture (ISA) has been designed for embedded systems efficiency. It contains 397 instructions including some advanced bitwise operations, fixed-point and floating-point arithmetic and extended integer arithmetic.

This architecture is predictable: under certain conditions timings of instruction execution and memory accesses are deterministic. This feature allows some strong real-time constraints to be enforced on the MPPA. It is provided by determinism enforced at every level of memory and operation hierarchy. Implication for the floating-point unit will be detailed on chapter 3.

This feature is also enforced by the fact that the K1 core executes in-order: the order of instructions in the program is not modified by the execution. Thus it is easy to pre-compute an execution latency, and an Instruction Set Simulator (ISS) can be used very accurately, as soon as the program does not make heavy use of the higher levels of the memory hierarchy whose timing behavior, while being easily bounded, remains difficult to predict accurately.

In order to save power, the MPPA-256 is built using a low-power standard library of cells. This implies that it runs at a low frequency (400 to 600 Mhz). This allows us to relax some synthesis constraints and to build a shorter floating-point pipeline than usual. In nominal conditions the whole chip consumes around 5W of power.

Nonetheless, it offers interesting peak performance: the 5-issue VLIW is able to deliver peak performance of around 2 Giga integer operations per second (Gops) at 400 Mhz. With the contributions of Chapter 3, it is also able to sustain 1.2 Giga floating-point operations per second (GFlops). However the K1 core is simple enough to make an interesting brick to build a manycore.

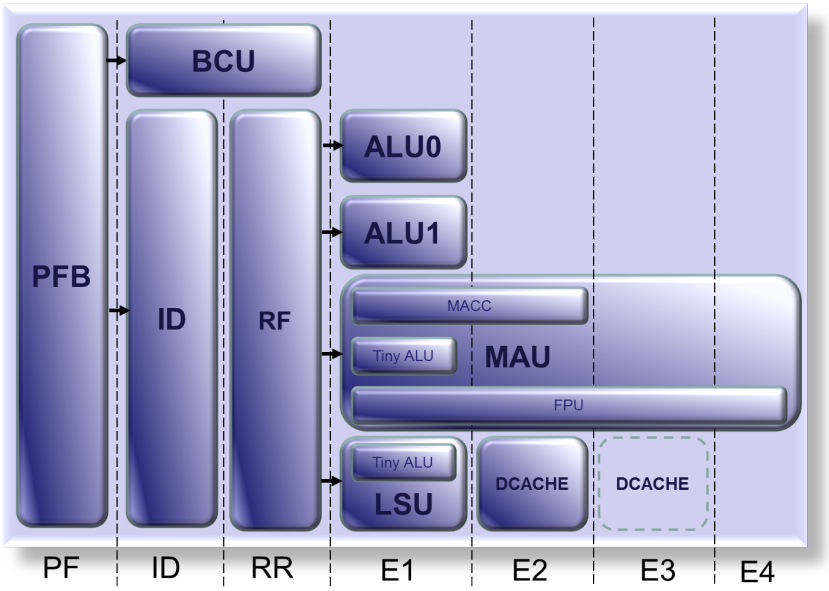


Figure 1.5: Kalray's K1 core architecture

Part I

Contributions to Floating-Point Arithmetic

CHAPTER 2
Context and state of the art

2.1 The IEEE-754 floating-point standard

Integer are not well-suited for computation with a wide dynamic range. Floating-point numbers were designed to improve computer accuracy in such calculus. A floating-point number f is defined with respect to a base b, and encoded using a mantissa m and an exponent e such as: $f = m \times b^e$. Zuse's Z3 computer was the first commercial system to provide, in 1942, floating-point capabilities [154]. It started a trend among manufacturers, and until the mid 1980s many companies integrated their own proprietary floating-point formats into their architectures. Those formats were generally not compatible between manufacturers and sometimes within the products of a same manufacturer (eg: IBM). Size, encoding and behaviour were machine-dependant which made the development of portable floating-point programs and architecture comparison very difficult. In 1985, under the impulse of Intel proposal, following the development of its i8087 numerical co-processor, a IEEE standard was accepted: [10]. It normalized the formats (e.g. single, double precision) and the behaviours (required operations, rounding modes). This standard was recently revised (2008, [73]) to take into account the latest evolutions (e.g. decimal floating-point [11], binary16 format, Fused Multiply and Add).

Let us now review three points of the standard which are relevant to this work: the binary floating-point formats and roundings and exceptions.

2.1.1 Floating-point formats

Table 2.1 lists the most common formats defined by the IEEE standard. Each format is built on the same basis: a number is expressed as $s.m \times 2^{e-p-bias}$ with s the sign, m the mantissa and e the exponent encoded in the bit string. p is the format precision (length of the mantissa) and *bias* is an exponent offset defined for each format.

format	bit width	mantissa size $(p-1)$	exponent size	$bias$	common name
binary16 (fp16)	16	10	5	15	half float
binary32 (fp32)	32	23	8	127	float
binary64 (fp64)	64	52	11	1023	double
binary80 (fp80)	80	64	15	16383	extended

Table 2.1: IEEE floating-point format

Mantissas are encoded in sign-magnitude format. According to value of e, three cases are distinguished:

- e **is the max value**: the number is a specific value, non numerical value, among Not a Number (NaN) or signed infinities.
- e **non zero**: the number is said to be normal, mantissa leading bit is a one. The leading one is not encoded into the mantissa bitfield which saves 1 bit. It is called the **implicit one**.
- e **is zero**: the number is said to be subnormal or denormalized (deprecated name used in [10]). The leading bit is an implicit zero and the mantissa leading one can be located anywhere in the remaining $p - 1$ bits of mantissa.

Because of the space occupied by sign and exponent fields, the encoded mantissa width is less than the word length: 23 vs 32 bits in binary32 and 52 vs 64 bits in binary64. We add the implicit bit to the encoded with, to get the format precision p: 24 for binary32 and 53 for binary64. Specific values (encoded by an exponent equals to the max value) are distinguished through mantissa values (0 for infinities, non-zero for NaN).

2.1.2 Rounding modes and exception

For binary floating-point arithmetic, the IEEE standard [10] requires the exact implementation of several floating-point operations including addition/subtraction, multiplication, square root, division, conversion from and to integers. The 2008 revision [73] added the fused-multiply and add operation (FMA) to that list. Here, **exact** means that the operation result should, first, be computed as if in infinite precision and unbounded exponent range, and eventually rounded toward the output format according to the current rounding mode. In binary floating-point arithmetic, this rounding mode has to be chosen among the four modes defined by the standard:

- roundTiesToEven: the exact result is rounded to the nearest floating-point value, ties are rounded to the nearest floating-point number with an even mantissa.
- roundTowardPositive:: the result is rounded to the nearest greater or equal floating-point value.
- roundTowardNegative: the result is rounded to the nearest less or equal floating-point value.
- roundTowardZero: positive results are rounded as if rounding towards minus infinity was in place, negative results are rounded as if rounding towards plus infinity was in place.

To detect and manage specific computation events, the standard defines 5 exceptions:

- division by zero
- inexact result
- invalid operation
- underflow
- overflow

A precise definition for these exceptions can be found on section 7 of [73]. A compliant implementation of the standard, should have them raised by every operations when the conditions are met.

2.2 Unit in the last place

The unit in the last place or **ulp** is an important concept used to measure floating-point errors. However this concept has many definitions that were summarized and studied in [114]. We are not going to go into details about the specificities of each ulp definition and for the simple examples illustrating this work the following [114]'s definition should suffice:

Definition 2.2.1 *If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then $ulp(x) = |b - a|$, otherwise $ulp(x)$ is the distance between the two finite floating-point numbers nearest x. Moreover, $ulp(NaN)$ is NaN.*

In a coarse approximation, for a floating-point number X, $ulp(X)$ can be considered as the weight of the least significant bit of X's mantissa.

2.3 State of the art of floating-point support in embedded processors

Floating-point constitutes the core of numerous embedded applications. It is key in digital signal processing, graphics processing. Thus many embedded architectures, especially if they target floating-point intensive applications, provide hardware support. Let us review in more details the floating-point support provided by some architectures representatives of the embedded taxonomy.

2.3.1 ARM architecture

VFP ARM architecture offers a floating-point coprocessor extension since the ARMv5 architecture. This extension called Vector Floating Point (VFP) [12] provides support for single and double precision formats. It is implemented as a coprocessor and features a separate floating-point register file containing between 16 and 32 64-bit floating-point registers. Later versions were extended with support for a fused-multiply accumulate. The extension is compliant with IEEE 754 [73], and also provides a *Run-Fast* mode which degrades compliance and improves performance. Contrary to what its appellation could suggest, this extension does not provides SIMD acceleration, as vectors are executed sequentially. This feature (vector sequential execution) was removed in last version of VFP.

NEON To provide efficient SIMD acceleration, ARM introduced an other extension called NEON [1]. The NEON extension features a truly SIMD engine containing 128-bit vector registers. It provides both integer and floating-point vector instructions. Floating-point support is limited to single precision, compliant with IEEE 754[73], with up to 4 elements simultaneously processed by each instruction. NEON provides conversions and standard arithmetic (addition, subtraction, multiply and fused multiply-add).

ARM architecture can feature both VFP and NEON, for example the ARM cortex A8 implements a VFPv3 coprocessor and a NEON extension. It is one of the advantages of ARM architecture: it can be customized to adapt to application requirements. In the most recent architectures (e.g. ARMv8-A), both extensions are mandatory.

2.3.2 ST231

The ST231 is an embedded processor from the ST200 family by ST Microelectronics. It is a digital signal processor targeting the embedded media processing market. It implements a Very Long

Instruction Word (VLIW) architecture able to execute two integer multiplies and two integer adds each cycle. It constitutes an interesting example of floating-point support because it does not integrate a hardware floating-point unit. Floating-point capabilities are provided by a highly optimized software library: FLIP [79]. FLIP is a state of the art software library implementing the basic floating-point operations in compliance with the IEEE standard [73]. A more detailled presentation of FLIP is given in Section 3.1, p. 37.

2.3.3 IBM's Power architecture

The Power architecture was first introduced in 1990, in the RISC System/6000 computers. Initially designed for mainframe computers, it has since known several iterations and has been extended to the embedded market.

The Power ISA v2.06, last iteration of IBM's Power architecture, provides both 32-bit and 64-bit binary floating-point support, compliant with the IEEE 754 standard. This support includes most current operations (including Fused Multiply-Add) and exists in vector format with the Altivec ISA extension. Altivec extension works on 128-bit vector registers and supports up to 4 32-bit floating-point operations in parallel. It does not support 64-bit floating-point. IBM's Power is among the very few architectures which provide hardware support for decimal floating-point (32, 64 and 128-bit Decimal Floating-Point (DFP) numbers are supported). It provides common operations (conversions, comparisons) and four basic arithmetic primitives: addition, subtraction, multiplication and division. It does not provides a decimal Fused Multiply-Add. The DFP unit is not part of the embedded implementation of the Power architecture.

The Power architecture is implemented by several manufacturers beyond IBM. Among them we can cite Freescale, that produces the PowerPC e6500, an embedded processor which implements a Power architecture. A e6500 core contains five integer units, two load-store units and a 128-bit Altivec unit providing vectorized floating-point support for binary32 computation.

2.3.4 Texas Instrument TMS320

In 1983, Texas Instruments introduced the first representative of the TMS320 family: the TMS32010. This DSP was rather successful in the embedded market and has since known several variations. It still appears, as a DSP coprocessor connected to ARM cores, in the OMAP System on Chip (SoC), processor from TI used in handheld devices. It implements a VLIW architecture. The floating-point unit is not part of the main ISA and appears as a hardware option. Some specialized DSP, for instance the TMS320C6731B of the C67x series, have been designed to provide a heavy 32-bit floating-point support. This version features four floating-point ALUs (addition) and two floating-point multiply units: it is able to compute up to two floating-point multiply-accumulate each cycle. It does not provides a Fused Multiply-and-Add and thus is not fully compliant with the revised standard [73].

2.3.5 Conclusion

Three of those architectures share a common trend of recent embedded processors: hardware floating-point support. This support, if not provided by default, is always provided as an option. The floating-point capabilities of embedded processors are increasing with each generation. Recent needs in multimedia acceleration have made this support a must-have for any new architecture that wishes to target those markets. This little sample of architecture illustrates the variety of floating-point support in embedded processor. The most recent ones support at least binary32 and implement vector extensions to improve throughput. All implementations are compliant with

the IEEE 754-1985 floating-point standard [10] and many are already compliant with the revised version [73]. Binary64 support is partial and limited to scalar operations.

CHAPTER 3
Hardware floating-point design

3.1 Introduction

Among the operations required by the IEEE standard, four are used more frequently than the others: addition, subtraction, multiplication and fused-multiply and add. Those operations are used so often that providing a fast implementation is a must-have for any embedded system that wants to target floating-point oriented applications. Many solutions exist to implement these operations. They can be decomposed in two categories: software implementations and hardware implementations, namely a floating-point unit (FPU).

Software implementation relies on an other arithmetic (e.g. integer) to implement these four floating-point operations. It is most flexible and easier to implement than hardware, thus its time to market is shorter. It is less efficient, exposing longer latencies and lower throughputs. FLIP [79] constitutes a good example of software implementation. It is a highly optimized library providing floating-point arithmetic support to an integer processor: the ST231. Performance result of FLIP are listed in Table 3.1.

These results can be compared to Table 3.2 which lists the performance of ARM Cortex-A9 hardware unit on the same operations. The real difference is even larger, ARM Cortex-A9 runs between 0.8 and 2 GHz while the ST231 is limited to 400MHz. This comparison illustrates a general fact: even optimized software implementations do not match hardware performance. Hardware provides the faster solution with the more dynamic power efficiency. Every FPU transistor is dedicated to floating-point computation. It is also more complex to develop and its integration impacts the archicture area.

Kalray's MPPA was designed for application making intensive use of floating-point arithmetic (scientific computing, digital signal processing). To target these markets, it was decided to integrate a hardware FPU into the K1 core. This chapter describes the architectural study and the micro-architectural developments of K1 FPU design.

It is divided into 3 sections. Section 3.2 describes the state of the art concerning floating-point

operation	latency (cycles)	throughput (cycles by result)
add/sub	23-26	23-26
mul	18-21	18-21

Table 3.1: FLIP: binary32 performance results [79]

operation	latency (cycles)	throughtput (cycle[s] by result)
add/sub	4	1
mul	5	1

Table 3.2: ARM Cortex-A9: binary32 performance results

unit implementation, focusing on the Fused Multiply-and-Add (FMA) operators. It first presents some generic designs and common improvements and continues by giving a short list of exotic designs such as ARM iterative FMA and IBM's multi-operand accumulator.

Section 3.3 presents three new operators:

- the mixed precision FMA which accepts a higher precision addend operand for better accumulation accuracy
- the 2D dot-product with full subnormal support
- the 3-operand floating-point adder

Section 3.4 presents micro-architectural contributions that can be used across several operators:

- bit-pattern recognition generator for leading zero anticipator development
- multiplication tiling: how to integrate a FPU inside an integer multiply-accumulate unit while maximizing multiplier uses.
- cookbook for hardware support of subnormal numbers.
- iterative support of double precision multiplication with half size multipliers focusing on the low latency support of subnormal numbers

To the best of our knowledge each of these contributions brings something new to the literature. Subnormal support is not a novelty per say as it has already been addressed for example by Schwarz et al. in [140] which was used during the development of the IBM mainframe eServer z990 CPU [61]. We provide an extension of such a support to our new operators, and to the best of our knowledge we are among the first that provides such operators with subnormal support in a commercially available embedded CPU.

3.2 Fused-Multiply and Add state of the art

Recent floating-point units often rely on a fused multiply and add (FMA). This floating-point operator first appeared in 1991 in IBM RS/6000 mainframe processor ([108]). It has since been generalized in most general public CPUs (For x86 extension, Intel integrated i, very recently, in its last instance of the Core architecture: Haswell, in 2013).

The FMA is a 3-input (A, B, C) operation which computes $A \times B \pm C$. It has several advantages. The first one is that it integrates both a floating-point adder and a multiplier which can be used separately or in combination. Together they constitute the basis operation for linear algebra. It also has some interesting numerical properties: the result is rounded only once, meaning that the multiplication result is injected without intermediate rounding into the addition. This can be used for efficient software implementation of a lot of floating-point primitives (eg: division, square root, elementary functions) which helps move the hardware/software co-design cursor towards software for the implementation of those primitives and thus avoid some costly specialized

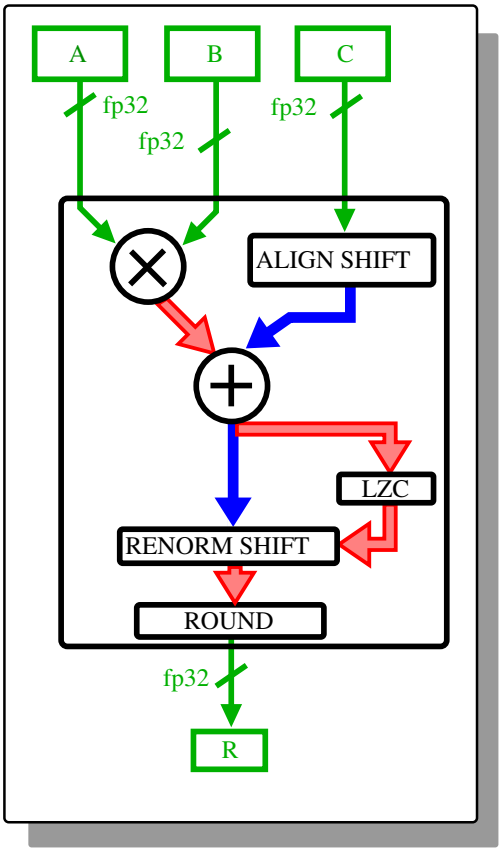


Figure 3.1: FMA simplified micro-architecture. The internal critical path is indicated by boxed arrows

hardware. This technique, which was used for Kalray's MPPA, was already applied by Intel for the Itanium and IBM on its Power architecture. It will be described in more details in Chapter 4 which addresses the implementation of division and square root on K1.

This section describes the basis of FMA micro-architecture required to understand the contributions of our designs. The reader interested in a broader taxonomy of FMA designs can refer to [131].

3.2.1 Single-path FMA

The main difficulty behind FMA implementation is that the operator gathers the cost of a floating-point multiplier, which is the area of the big mantissa multiplier, and the cost of a floating-point adder, which is mostly the complexity of the alignment/add/normalization path. Those costs are even worse for an FMA since one of the addition operand is twice as large as what can be expected for a standard floating-point adder. An FMA has a higher latency than a floating-point add or multiply.

FMA architecture contains the following elements:

- a mantissa multiplier, multiplying A and B's mantissas
- in parallel the addend mantissa (C) is aligned with the product.
- according to the instruction ($A \times B + C$ or $A \times B - C$) and the operand signs, an effective addition or subtraction is realized by the adder.

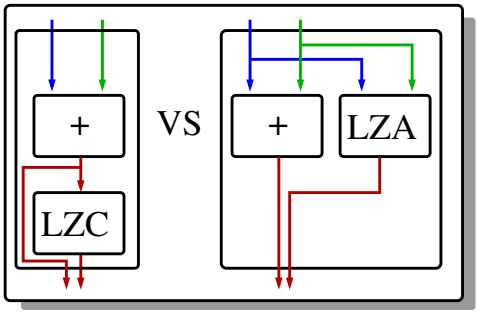


Figure 3.2: Comparison leading zero count mechanisms on addition: LZC versus LZA

- the result leading-one position is computed by a Leading Zero Counter (LZC).
- if the result is normal:
  - the result is normalized, i.e shifted so that the MSB of its mantissa is a 1.
  - the normalized result is rounded to fit IEEE-754 format
- if the result is subnormal
  - the result is denormalized, i.e shifted so that the MSB exponent is e_{min}.
  - the denormalized result is rounded to fit IEEE-754 format.

The overall architecture of an FMA is illustrated by Figure 3.1. The expected critical path has been highlighted in dark red. It goes through the multiplier, the main adder and the renormalization and rounding circuit. Let us now details three famous significant improvements to FMA, two micro-architectural: leading-zero anticipation and compound addition, and a more architectural one: the multiple-path design.

Leading Zero Anticipator

The Leading Zero Count (LZC) is defined as the numbers of zeros before the first one, starting from the most significant bit. A leading zero anticipator is a circuit built to compute a leading zero count on an addition. Rather than considring the addition result, it inputs the addition operands. It is also called leading one prediction (LOP) in the literature. Contrary to the LZC which is sequentialized with the addition, the LZA can be computed simultaneously as illustrated by Figure 3.2. Therefore, using a Leading Zero Anticipator rather than a Leading Zero Counter (LZC) saves several logic levels on the FPU critical path, while increasing the FPU silicon area.

The basic principle to compute the position of the result leading one is somewhat simple:

1. computing a bit string, from the two addition operands, that encodes the position of the leading one
2. performing a leading zero count on that bit string to get the final leading zero count.

The sum of bit string computation and LZC latencies is shorter than the addition latency. A drawback of leading zero anticipation is that this methods does not allow for a precise leading zero count. Indeed because of the carry propagation phenomenon in the actual addition, the LZA determines two consecutive possible positions for the leading one. A single one of these positions is correct.

We are now going to detail a possible mechanism for leading zero anticipation. A more detailed and exhaustive description of LZA mechanisms can be found in [130]. Let us consider the addition of two n-bit numbers, a and b, encoded in 2's complement by $a = a_{n-1}...a_0$ and $b = b_{n-1}...b_0$; b_{n-1} and a_{n-1} are the most significant bits while a_0 and b_0 are the least significant bits. To built the bit string, we use the following parallel prefix signals:

- p_i is true if $a_i + b_i$ propagates a carry ($p_i = a_i \; xor \; b_i$)
- g_i is true if $a_i + b_i$ generates a carry ($g_i = a_i \; and \; b_i$)
- z_i is true if $a_i + b_i$ kills a carry ($z_i = \neg(a_i \; or \; b_i)$)

The computation of p_i, g_i and z_i for $0 \leq i \leq n-1$ can be performed simultaneously, there is no dependency between two values of i. The hardware implementation only requires a few logic levels to the LZA critical path.

Let us now introduce some formalism to explain leading zero anticipation. This formalism is common with Section 3.4.1, p.64 about the bit pattern generator. We express bit string patterns using regular expressions. Our set of symbols is P, G, Z, $-$, $^+$ and * defined as follows:

- P matches a true p_i
- G matches a true g_i
- Z matches a true z_i
- $-$ corresponds to a wildcard or "don't care" (it can match without distinction any of the previous symbols)
- $^+$ is a rule symbol, the previous character is matched at least once
- * is a rule symbol, the previous character is matched an arbitrary number of times (including zero)

Let us consider an example: P^+GK^* matches the bit string derived from a and b. This pattern is the concatenation of P^+, G and Z^*. P^+ matches one or more P symbols: the input string starts with a propagating pair ($a_{n-1} \oplus b_{n-1}$=1). This initial pair is followed by an arbitrary number of propagating pair: $\exists j$, $a_i \oplus b_i = 1, \forall n-1 \geq i \geq j$. G matches a single G symbol: $a_{j-1}.b_{j-1} = 1$, which implies $j \geq 1$. Z^* matches zero or more K symbol: $\forall i, 0 \leq i < j-1, \neg(a_i \; or \; b_i) = 1$. Each regular sub-expression is supposed to be of maximal length, for example P^+- is equivalent to P^+G or P^+Z: the P^+ sub-pattern being of maximal possible length, $-$ can not match the P symbol.

Let us now describe the LZA using this formalism. We want to list the patterns which describe the possible addition results and determine the position of the result leading one in those patterns. Without loss of generality we can restrain ourselves to positive results: the following reasoning can then be easily extended to negative results. Once positive and negative leading one position are determined, the correct LOP is selected by the sign of the addition result.

We are going to express every possible input pattern which lead to a positive addition result. The position or possible positions of the leading one are indicated in **bold**.

We exclude overflow: inputs and result lengths are assumed to be wide enough so that no addition will overflow. A positive result can be obtained by a chain starting with Z^+ or P^+G, every other start leads to a one in the MSB position and thus a negative results. We exhaustively decompose the set of chain starting by Z by $Z^*\mathbf{Z}G-^*$, $Z^*\mathbf{Z}P^+G-^*$, $Z^+\mathbf{P}P^*Z-^*$. It is easy to see that this set covers every chain starting with Z. We consider $P^+GZ^*\mathbf{Z}G-^*$, $P^+GZ^*\mathbf{ZP}-^*$ and $P^+\mathbf{G(P\,|\,G)}-^*$ for chains starting with P^+G.

If we study those patterns closely we can determine that our detection bit string should be 0 for patterns PP, ZZ, PG, GZ. Indeed those patterns can not create a leading one in either of their two indexes. On the contrary ZG or GG or PZ creates a possible leading one in their left position, ZP or GP creates a possible leading one in either of their positions.

From this decomposition we can deduce an expression to indicate leading one bit candidates. By taking the negation of the non leading pattern we find:

$$\neg((\neg(p_i).z_{i-1}) + (p_i.\neg(z_{i-1})))$$

which can be simplified into the general expression for the leading one prediction bit-string:

$$lop_i = \neg(p_i \oplus z_{i-1})$$

This expression is thus null for every pattern that does not create a leading one and, non-zero for patterns creating a one, which could possibly be the leading one if no other one precedes it. To determine the possible leading one position, a leading zero count is then computed on the bit string lop_i.

The final step is to select this positive leading one position or the negative leading one position according to the sign of the result. This selection can be performed on the addition result is available or before, by adding a lightweight sign determination circuit as suggested in [91].

LZA have proven to be faster than LZC plus adder pair [130], thus justifying their integration into low-latency FPUs.

Compound adder

Let us consider the implementation of a floating-point addition (equivalently subtraction) of A and B. As the floating point number encoding is sign magnitude, this implementation must compute the absolute value of the results, and its sign. As A and B can have opposite sign, $|A - B|$ is equal to $A - B$ or $B - A$. The easy way to compute the absolute value is to compute one of them, let us say $A - B$, and take its opposite if the results is negative. As the operation is performed in 2's complement format, this result in possibly two additions: one to compute $R = A - B$ and one to compute $-R = \overline{R} + 1$. This design sequencialized two adders (more specifically one adder and one incrementer). A more latency-efficient design is to compute $A - B$ and $B - A$ in parallel, and then to select the positive on. The previous incrementer is swapped for a multiplexer, much more latency efficient. It can appear expensive in silicon since two adders have to be implemented, hopefully a simple trick alleviates this difficulty: using a compound adder. In 2's complement encoding the following stands:

$$B - A = -(A - B - 1) - 1 = \overline{A - B - 1} + 1 - 1 + 1 = \overline{A - B - 1}$$

This means that by computing $R = A - B - 1 = A + \overline{B}$ and $R + 1 = A - B = A + \overline{B} + 1$ we can obtain easily both $A - B$ and $B - A$ from which we can select the absolute result value. From the operands A and B, a compound adder computes simultaneously two results: $R = A + B$ and $R' = A + B + 1$. Thus it can be implemented into a floating-point adder to compute simultaneously $A - B$ and $B - A$.

Implementing a compound adder can be done easily by modifying a fast adder scheme such as the carry select adder. The left part of Figure 3.3 presents the scheme of a carry select adder: the input is divided in high and low parts. Each corresponding pair is added twice, considering both cases of carry input, finally the result is built by selecting one of those pre-computed results thanks to the computed carries. The right part of Figure 3.3 shows a compound adder scheme obtained by routing some intermediary results of the carry select adder to the new $R + 1$ output: no extra

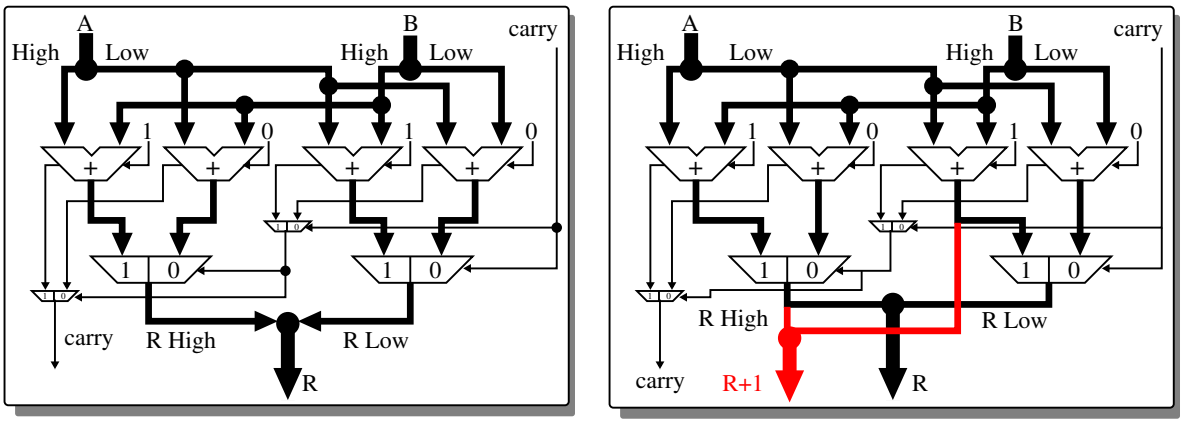


Figure 3.3: Last-level carry-select adder structure (*Left*). Compound adder structure (*Right*)

logic is needed. This solution is faster than using a fast adder sequentialized with an incrementer / multiplexer. For the interested reader, more information about adder design can be found in [51] and an example of compound adder can be found in [87]. This solution was implemented in the second version of Kalray's K1 FPU. Due to time and resource limitations, the first version used a generic fast adder macro provided by the RTL synthesizer. Studies on those hard-macro is available in appendix Section 14, p.201.

Other uses of compound adder have been suggested. The critical path of floating-point addition contains two addition operations. The first one computes the infinitely precise result and the second one increments the normalized result for rounding. By extension this is also true for the floating-point FMA. Merging them into a single operator using a compound adder, as suggested in [21], reduces the floating-point adder latency. Overall designing a compound adder is an interesting way to try improving FPU latency and/or area.

3.2.2 Multiple-path FMA

As stated previously FMA operation has a very long latency.

Many solutions have been suggested to reduce the operator timing, including the two optimizations presented previously. Among those works, the FMA path decomposition has been extensively studied.

FMA path decomposition is linked to the cancellation phenomenon.

Cancellation is defined as loss of significance that can happen in a floating-point operation. For example, subtracting two very close numbers may lead to a catastrophic cancellation. The most significant digits of both operands can cancel each other extensively and the result leading one has an index very inferior to the operands' exponent. Let us go through an example: the floating-point addition of $A = 1.000001 \times 2^{42}$ and $B = -1.111100 \times 2^{41}$. After alignment on the same exponent we get A and $B_a = -0.111110 \times 2^{42}$, and after adding them together we get $R = 0.000011 \times 2^{42}$ which can be normalized into $R_n = 1.100000 \times 2^{36}$. We are not interested in the relative error increase by such phenomenon but rather by the fact that it necessitates specific hardware management. This phenomenon justifies the integration of a LZC/LZA, in FPUs, to compute R exponent and perform the correct normalization.

In the following, "*effective operation*" stands for the practical operation performed by the main adder of the floating-point unit: adding two operands of opposite signs will result in an effective subtraction while subtracting two operands of opposite signs results in an effective addition (the sign is managed separately in the sign-magnitude floating-point encoding).

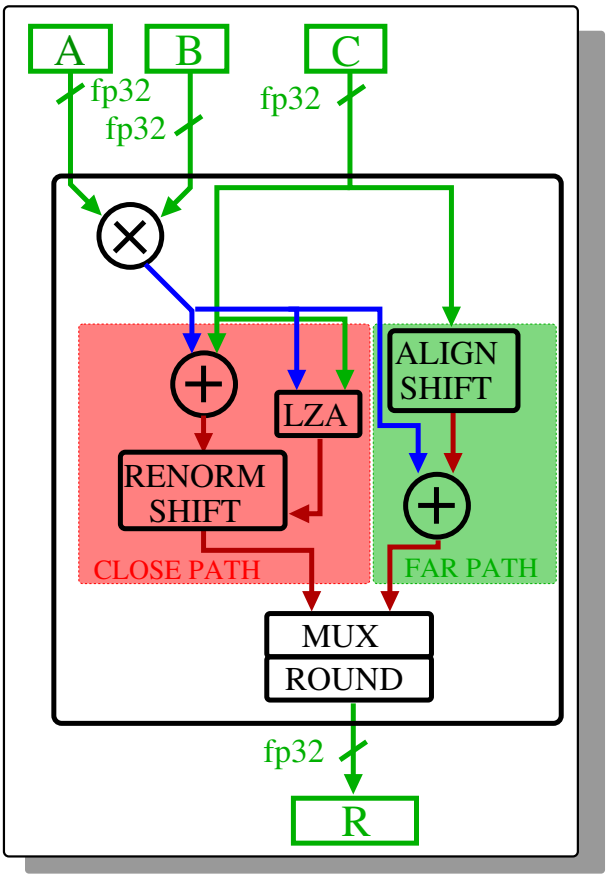


Figure 3.4: FMA multi-path architecture

When looking at a floating-point add architecture one can notice that there are at two, mutually exclusive, cases:

- if the operand exponents are far enough from each other, the addition of the two operands can not lead to a large cancellation. The exponent of the effective addition result is at most the exponent of the highest operand increased by one. In case of effective subtraction, there may only be a unit decrease of the highest exponent. Those considerations do not consider overflows and underflows. The effective operation datapath must only contain one extra bit to the left of the highest operand, to account for carry overflow. The result leading bit has one of three positions: same weight w as the leading digit of the highest operand, $w+1$ or $w-1$. After the effective operation, a wide renormalization is not required, a simple 3-input multiplexer can be used to manage such a case. In addition to the adder the main hardware cost of induced by this case comes from the alignment shifter which must manage every possible alignment cases. Such a case is called the **far case**.

- if the operand exponents are very close, meaning strictly less than 2 units apart, the consequences depend on the effective operation. Effective addition requires no expensive alignment since there are only three possible result alignment cases. A large cancellation may occur during an effective subtraction. In such case, the leading one position can not be limited to a small number of static positions. A leading zero counter is required after the effective operation, this LZC drives a large normalization shifter. Such a case is called the **close case**.

In the single path architecture, presented in Figure 3.1, these independent cases are managed by the same single path. The critical path of each case is added, resulting in a poor overall latency.

The independence of this cases can be exploited. A path for each case can be implemented, by duplicating the main adder, and partionning the other main components. One single path is active per input. The multi-path architecture was introduced in [121] for the floating-point addition. The architecture is illustrated by Figure 3.4 where the **far path** managing the far case can be distinguished from the **close path** managing the close case. Dividing the main path has a beneficial effect on latency since the alignment shifter and LZC/normalization shifter can now be considered as independent. The area increase which results from the adder duplication is often limited because of the latency decrease. Indeed, the decomposition is lowering the timing pressure on the critical path thus allowing the RTL synthesis tools to select smaller gates with less power drives. Thus resulting in an overall smaller design. This is illustrated in Section 14, p.201: as can be noted in Figure 14.3, p.204, decreasing the size of the shifter allows to synthesize at lower latencies without reaching the exponential part of the area increase.

The multi-path architecture, introduced for floating-point addition, can be generalized to FMA. A presentation of a such a generalization is available in [88]. This idea has been carried out further and some designs with more than 2 paths exist: [132]. The principle is the same: dividing a path between possible case to diminish its critical path latency. [132] subdivides the far path between multiplier anchored and addend anchored cases. This translates into far cases where multiplier is greater than addend being distinguished from addend greater than multiplier with separate circuits. This modification also improves the operator latency by further reducing the critical path length. However more and more hardware has to be duplicated, in this case at least the alignment shifter for far path appears twice. [131] lists a 38.6 % area increase between classic FMA and 3-path FMA with a 12.0% latency decrease on a low V_T technology.

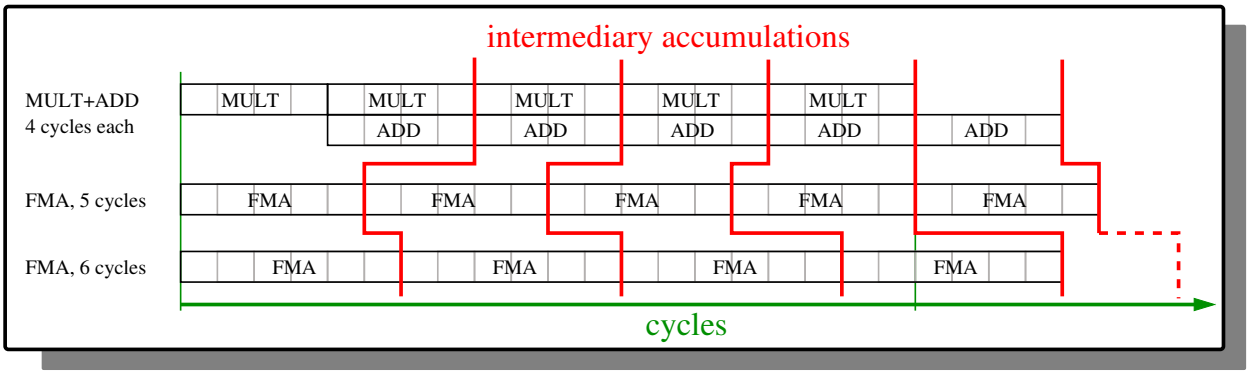


Figure 3.5: Comparison of latencies: FMA versus paired FADD/FMUL

3.2.3 Exotic FMA designs

In the previous subsections we focused on describing the more generic Fused-Multiply and Add, generally as they are implemented in common CPUs. But the research on FMA's design is still on going and new designs are suggested every year. We chose two of them that constitutes a link between the generic FMA and the original designs we contributed.

ARM iterative designs

One of the FMA disadvantages in front of the adder/multiplier pair is its pipeline latency. With equal design optimizations, an FMA requires more pipeline stages than an FP adder or multiplier. This means that in a series of linked FMAs corresponding for example to a large dimension scalar product the FMA is less effective than a adder/multiplier pair. This is illustrated by Figure 3.5 which compares the accumulation latency of a scalar product: solution using separate adder/-multiplier versus solution using an FMA. The comparison considers the most favorable case, that is when the FMA has only one extra pipeline stage compared to the adder/multiplier pair.

The reader can easily deduce the same conclusion with other latencies than those chosen on Figure 3.5 as long as they respect the hypothesis: an FMA is slower than an adder or a multiplier considered on its own.

Starting at the 4th accumulation, FMA-based scalar product becomes slower than ADD-MUL based.

David R. Lutz of ARM corp. offered a solution to that problems in [101]. This solution takes the shape of an FMA built with two separate operators: a FPADD and a FPMUL. Of course the FPADD had to be modified to accept a larger second operand which can contain the exact result of the floating-point multiply.

This architecture has several advantages:

- it provides two distinct operators that can be used at the same time to benefit from instruction level parallelism (ILP).
- each operator has a better latency that an FMA could provide.
- the operators used together provide a true FMA, meaning an operation equivalent to a multiply and add with a single rounding.
- in the case of products accumulations, the use of this exotic FMA proves faster than a single block FMA because of the latency gain of decoupling the operators.

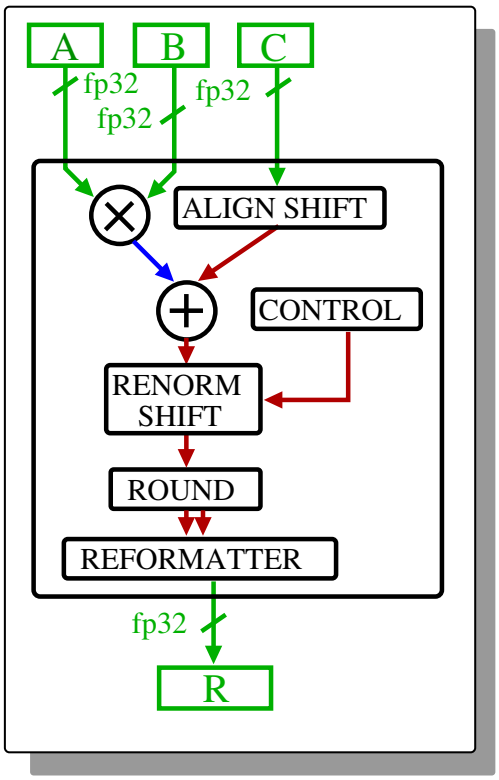


Figure 3.6: IBM High and Low part FMA overview

However, one could argue that the cost of such a design is greater that a simple FMA because of the duplication of the rounding circuit and operand field extraction circuit for example. It can also be noticed that this architecture only improves latency for chain of FMAs addend-dependent. For FMAs depending on one of the multiplication operands this design does not bring any latency improvement and for vectorized workload the throughput is not improved.

IBM high and low parts FMA accumulation

IBM introduced a modified FMA [60] to support extended accumulation. This FMA has been designed to improve the support of fixed-point multiprecision accumulation. It outputs its result on two separate outputs, called High (H) and Low (L) and those added together contain the $A \times B + C$ results in extended precision. An overview of the FMA's architecture is illustrated in Figure 3.6. The output has the same width as the operands and an extra reformatting circuit is in charge of forwarding the high or low result parts when required.

This operator is quite useful for exact fixed-point operation. The product of two n-bit mantissas fits onto a $2n$-bit mantissa which can then be partitioned between a high and a low part of n bits.

The accumulation of multiple products may involve some rounding errors, but those errors are limited in front of the single-output FMA which does not provide the extended output accuracy.

The operator can only output a single n-bit result each cycle. This means that two cycles are required to output the complete high and low part results, thus reducing the throughput of the operator by a factor 2.

IBM also claims that an extra fourth operand could be included before the effective addition with very low cost. However this does not take into account the need for possible alignment of

this operand.

In fact the whole operator has been built considering fixed-point support. It contains a specific module to bypass the normalization and keep unnormalized output for future accumulation in fixed-point. The support for standardized floating-point format is unclear and IBM seems to focus mostly on its own proprietary HFP format.

3.3 Building K1's FPU: architectural study

The previous section focused on FPU/FMA state of the art. We started by describing the more generic single and multiple paths FMA architectures. Finally we introduced some of the more recent designs. These designs have in common that they target in some way or others multi-precision. [101] adds an extended output to the floating-point multiplier and an extended operand to the floating-point adder in order to process exact FMA operation with two separate operators. [60] provides an extended result using two numbers rather than one to express the FMA output. It greatly improves the accuracy of the accumulation.

In this section we study several operators to build Kalray's floating-point unit: a mixed-precision FMA (MPFMA) extended with support for binary64 addition support, a 2D-dot product and a 3-operand addition. Those operators are variations of a standard 32-bit FMA: they share interfaces (number of inputs) or part of the internal structure (product accumulation). Eventually the first two operators were merged into Kalray's FPU and the last one was discarded due to integration difficulties.

3.3.1 Mixed-precision FMA

In the following, *N-format* refers to a floating-point format with N-bit mantissa.

The vast majority of FMAs available in the literature only supports the same precision for inputs and output for each mode of operation. Some designs support multiple precisions: [71] or [63] or [7]. But precision can not be mixed, each operation mode supports a single uniform operation precision for both inputs and outputs.

We suggest a new design called Mixed-Precision Fused Multiply-and-Add or **MPFMA** which offers such an interface with asymmetrical precisions. This operators expects two *P-format* operands for the multiplication and a *Q-format* addend, the result is returned in *Q-format*. *P-format* and *Q-format* can be chosen among many formats (e.g. $binary64 \times binary64 + binary128 \rightarrow binary128$). We chose to implement the version shown on Figure 3.7: $binary32 \times binary32 + binary64 \rightarrow binary64$, which we will use to illustrate this section. Like in a standard FMA, there is only one rounding at the end of the multiply-and-add, which means that the multiplication is computed exactly internally.

Let us first details in what this operator could be useful.

```
1  /*
2   * float inputA_vector[1024];
3   * float inputB_vector[1024];
4   */
5  double acc = 0.0;
6  int i;
7  for (int i = 0; i < 1024; i++) {
8      acc = acc + inputA_vector[i] * inputB_vector[i];
9  };
10 return acc;
```

Listing 3.1: binary64 accumulation of binary32 operand products

Indeed, the idea of mixing precisions in a single operation is interesting because it covers real application cases. Let us consider a large dimension scalar product on low-precision (fp32) inputs, example code can be found in Listing 3.1.

Even if the precision of the inputs is low because of measurement limitations or memory constraints, one could want to perform an accurate accumulation of the coefficients multiplications, if only to reduce the weight of the accumulated rounding errors. A standard FMA offers two choices: converting every inputs into a higher precision and processing the accumulation in the higher precision, or using the lower precision during the accumulation while keeping input precision unmodified. The first solution is costly because it requires a higher precision FMA which can be quite expensive. Details can be found in Table 3.3, p.55. The much higher cost of an FMA64 is the consequence of several factors. First the multiplier size which represents a large part in an FMA design has more than quadrupled, since an FMA32 requires a 24×24-bit multiplier and an FMA64 requires a 53×53-bit multiplier. Multiplier complexity is proportional to the square of its input size (because it drives the number of partial product generated and the compression tree size). By referring to the results presented in Table 14.4, p.204, we can determine that in our technology this ratio is of 4.5 in favour of the 24×24 multiplier.

The second reason is that each datapath of the FMA64 needs to be more than twice as large as the one in the FMA32, since it is manipulating 53-bit mantissas rather than 24-bit mantissas for the FMA32. This implies more timing pressure during the synthesis steps (for the same latency constraint). Thus the RTL synthesis tools tends to use bigger gates (with lower latencies), increasing even more the FMA64 cost with respect to an FMA32. Finally this solution is inefficient, because more than three quarters of the multiplier remain unused, still leaking power.

The second solution, keeping a low precision during the computation, does not meet the goal of providing an extended precision accumulation. Indeed the accumulation is rounded in low precision at every step and a substantial number of bits are lost during this operation. This can be tragic if the accumulation contains close products with opposite signs: big cancellations can happen and completely ruin the accumulation accuracy. This phenomenon also happens with the MPFMA, but thanks to its extended accumulation precision, it can absorb larger cancellation than the reduced precision accumulation.

In conclusion standard FMAs (both low and high precision) do not fit, as nicely as the MPFMA, the requirements for low-cost high precision accumulation of low-precision inputs.

C11 compatibility

An other justification for the creation of the MPFMA is that it is fully compatible with the C-standard (C99 [76] and more recently C11 [77]) when adding a product of binary32 to a binary64 addend.

The paragraph 8 of section 6.5, p.77 of the standard [77] (p.65 for C99 [76]) allows for multiply and add fusion into atomic FMA (with single rounding). Input type conversions from **float** to **double** are allowed by the standard (paragraph 6.3.1.8 p.52), as there is no change of type domain. After that conversion, a double precision FMA can be computed to output a double precision result. The MPFMA merges those conversions and FMA operations into a single operator. Moreover it is better because smaller (in both area and latency) than a double precision FMA while providing bit to bit identical (isomorphic) results.

Our design interface is presented by Figure 3.7, it accepts a large precision addend and output a large precision accumulator. The architecture detailed below respects the FMA principle: a single rounding is done after the exact multiply-add has been performed. The asymmetrical precisions allow us to spare a lot of resource but unveil some challenges. In particular, subnormal numbers in the low precision input format does not constitute subnormal numbers for the

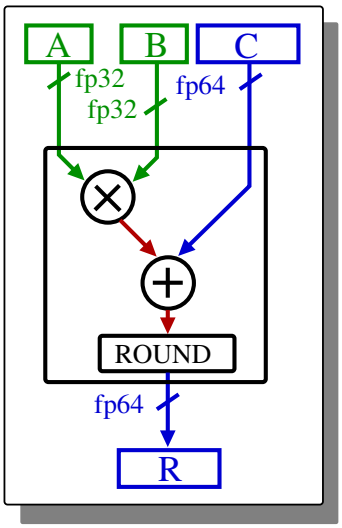


Figure 3.7: Mixed precision FMA overview

extended precision accumulation format. It increases the normalization to be performed on certain multiplication results. This challenge and more micro-architectural features are discussed in details in the remaining of this section.

MPFMA micro-architecture

In the following p stands for the mantissa width of the P-format; q stands for the mantissa width of Q-format. In our implementation P-format is instantiated by binary32 (p=24) and Q-format is instantiated by binary64 (q=53). All this would apply to binary64/binary128 precision (p=53, q=113)

The first MPFMA architecture is illustrated by Figure 3.8. This first version was designed for the first prototype of MPPA implementation. There was no strong frequency constraint during this design. This lead to a few area optimizations, for example not-using a LZA.

The main differences with a standard single precision FMA are:

- wider addend operand C and result
- the presence of two LZCs, one for each multiplication operand
- the post-renormalization of the multiplication result
- a $2q + 5$ wide adder (rather than a $3p + 5$ for FMA32)

Let us now details those differences. The first difference originates directly from MPFMA's interface: the addend operand and the result are twice as wide as the multiplication operand, binary64 vs binary32.

First consequence concerns register file interfaces:

- an FMA32 requires three 32-bit inputs and one 32-bit output (96/32).
- an FMA64 requires three 64-bit inputs and one 64-bit output (196/64).
- a MPFMA requires two 32-bit and one 64-bit inputs, and one 64-bit output (128/64).

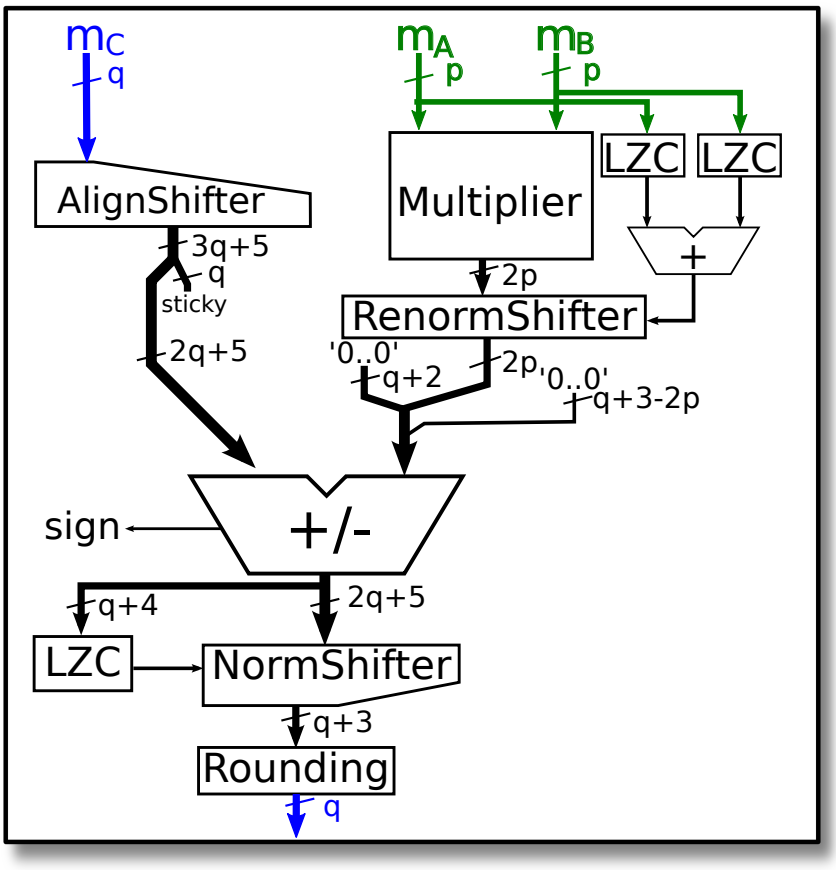


Figure 3.8: MPFMA simplified architecture

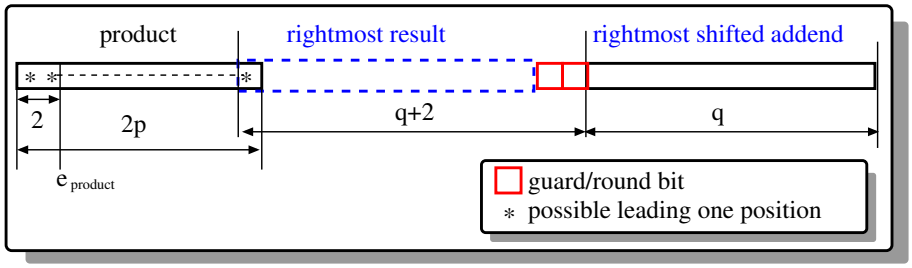


Figure 3.9: MPFMA datapath layout, and possible leading digit positions, for product-anchored cases, $e_{product} - e_{addend} > 2p - 1$

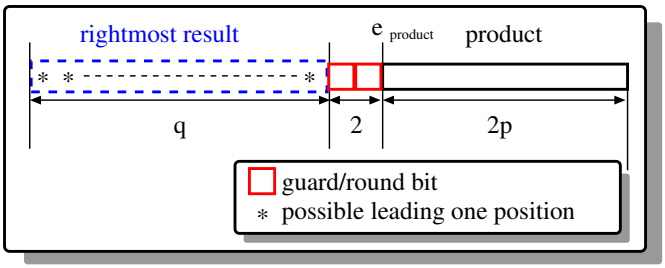


Figure 3.10: MPFMA datapath layout, and possible leading digit positions, for addend-anchored cases, $e_{product} - e_{addend} < -2$

The three operators have the same number of interfaces but the FMA32 interface totals to 128-bit wide, the MPFMA to 196-bit wide and the FMA64 to 256-bit. Register file is an important factor in processor energy consumption. Register power is linked to the number of read/write ports, the size of the ports, and the number of registers. The MPFMA reduces the size of the ports with respect to FMA64.

The second difference concerns the support for subnormal numbers. In a standard FMA when both multiplication operands are subnormal, the multiplication result lies way behind the lowest representable number. Indeed, binary32 subnormal operands have exponents lower than -126. Moreover, their mantissas lie in the range [0, 1[. The multiplication result exponent is at most -252. This exponent is very far from the lowest subnormal extended exponent of the binary32 format which is $-126 - (p - 1) = -149$. As a consequence, the multiplication result will only modify the rounding direction but not the pre-rounding result itself.

In such a case, the multiplication can be thrown out into a sticky flag to determine rounding directions but does not need to be injected into the effective adder. Thus such a case may be ignored when building the FMA32 datapath for correct rounding, even when considering support for subnormal inputs.

This does not applies to the MPFMA. Even the product of the two lowest representable binary32 numbers exponent (-298) is still greater than the minimal normal exponent of binary64: -1023. Thus such a case needs to be considered when building the MPFMA datapath and determining normalization requirements.

Our architecture is single path and product-anchored which means that the product is statically positioned in the datapath and is never shifted before reaching the main adder. Product length is $2p$-bit.

Let us first consider the case where the addend exponent is lower than the product without extended cancellation. This case is illustrated by Figure 3.9. If the product is zero we can output

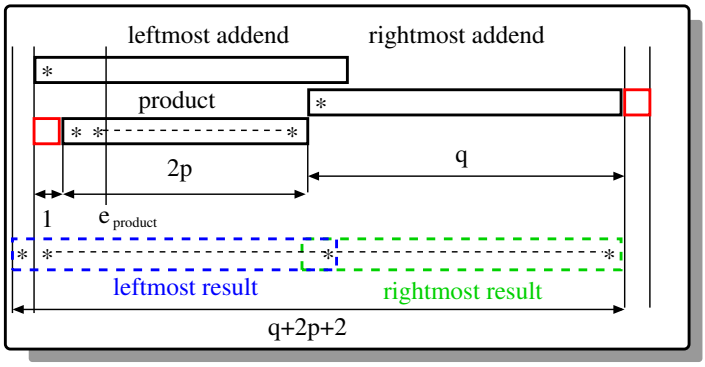


Figure 3.11: MPFMA datapath layout, and leading digit possible positions, for cancellation cases, $-2 \leq e_{product} - e_{addend} \leq 2p - 1$

directly the addend. However if the product is non-zero, its leading one can be located everywhere inside its $2p$-bit size, because of the normal behavior of subnormal binary32 in binary64 format.

We are trying to determine the rightmost shift required to the addend to obtain a correctly rounded result. This shift drives the datapath width. The extreme cases is located $q + 2$ bits right of the product rightmost leading one position. q bits for the result mantissa and 1 bit for the rounding bit and 1 extra bit as guard bit in case the product mantissa is 10^+ and the addend is of opposite sign which means that the round bit is shifted one bit to the right, because the leading one position will decrease by one bit. As intermediary conclusion we need to keep $q + 1$ bits right of the statically positioned product. The extra right q bits can be reversed into the sticky flag because we do not need their exact values for correct rounding, the fact of knowing that at least one of them is non-zero is enough.

Let us now consider the case where the product exponent is lower than the addend exponent and no extended cancellation is happening. This case is illustrated by Figure 3.10. In such a case if the product is non-zero, the leading one of the addend is at its leftmost position. Indeed because of binary64 to binary32 emin differences, a non-zero binary32 $\times$ binary32 product can not be lower than a subnormal binary64 addend. Because of this leading one position we just need to insert a round and a guard bit between the product and the leftmost position of the shifted addend.

From the two previous cases we can deduce some of our datapath dimensions:

- $q + 2 + 2p + q + 1 + q = 3q + 2p + 3$ bits alignment shift for the addend
- rightmost q bits are discarded into a sticky after alignment
- this leaves a $2q + 2p + 3$ bit wide main adder

Once we have determined the alignment shift and adder size, we need to determine the leading zero count required to support the cancellation cases. Cancellation cases are illustrated by Figure 3.11. A cancellation can happen when subtracting two operands whose leading one position is no more than one index apart.

The leftmost addend alignment, fitting this definition, is when the addend is one bit to the left of the product. The rightmost alignment is when the addend is 2p bits to the right of the product. Because the product leading one can be located on its rightmost position, and because a exponent offset of 1 in either direction can still trigger cancellation. This means that our LZC input needs to be $q + 2p + 2$-bit wide. This LZC also detect carry overflow when the effective operation is an addition. In the current implementation of the MPFMA, we call cancellation case the condition

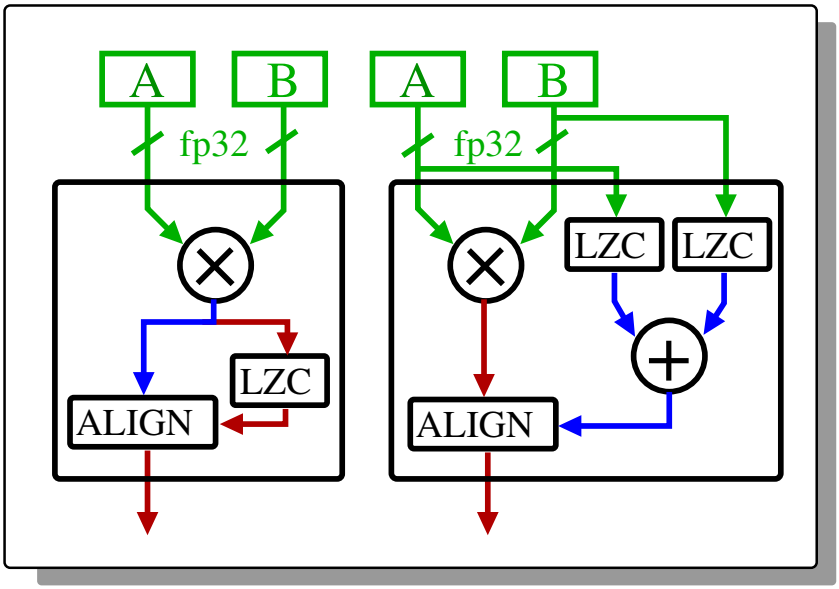


Figure 3.12: Two solutions for product pre-normalization in the MPFMA

$-2 \leq e_{product} - e_{addend} \leq 2p - 1$, without considering the effective operation. Thus we also have to consider effective additions.

We have studied some ways of reducing the MPFMA datapath width. Our suggestion is to pre-normalize the product before it is used as operand in the main adder.

This pre-normalization can be done two ways:

- determine an exact leading zero count on the computed product and use it to pre-normalize it.
- compute a smaller LZC on each of the product operands, add those counts and use the sum to shift the product.

The first solution is the easier one, because it is exact and does not trigger an extra post-normalization. But the second solution is more efficient, since the leading zero count (including the summation) can be computed in parallel with the multiplication. However it is less accurate, for two operands of size n and m the product size is $n + m$ or $n + m - 1$. Thus using leading zero counts on the operands results in an inaccurate leading zero count on the product by at most one bit.This inaccuracy can be handled later in the pipeline. For example, by taking into account the inaccuracy while determining the datapath (normalization LZC and shifter) dimensions.

The advantages of the pre-normalization are:

- reducing the size of the leading zero counter from $q + 2p + 2$ to $q + 4$ bits.
- reducing the size of the overall datapath: the right align case for the addend only requires $q + q - 2p$ extra bits, which means shift and adder sizes are reduced by approximately $2p$ bits.

Evaluation of the MPFMA The suggested MPFMA architecture was implemented in VHDL. Its silicon are was evaluated by performing a RTL synthesis. Results are listed in Table 3.3 which compares the MPFMA to a 32-bit and a 64-bit FMA. As expected the MPFMA area is between FMA32 and FMA64. It is 33% more expansive than the FMA32 and 43% less expansive than the FMA64.

Operator	latency (ns)	area μm^2
FMA32	3.5	10600
FMA64	3.5	24500
MPFMA32	3.5	14000

Table 3.3: Multiple FMA designs synthesis results for 28nm high density technology.

description	latency (ns)	area (cells)
24x24	3.0	1707
54x27	3.0	3771
54x54	3.0	7689

Table 3.4: Comparison of mantissa multiplier area depending on mantissa format

Conclusion on the MPFMA micro-architecture The MPFMA is a new operator, fully compliant with the C standard. Designing the MPFMA micro-architecture presents some challenges, especially when considering a complete and efficient support of subnormal numbers. By using product pre-normalization, and by carefully dimensioning the datapath we were able to implement such support. The MPFMA was implemented into the K1 first FPU.

The architecture, described in this section, remains basic. It has been upgraded to a single path with LZA in the second version of Kalray K1's Floating-Point Unit. In our context, area constraint are too predominant over timing to consider multiple-path architectures. If, in future versions, latency should become critical, we will consider such evolution.

Double precision support in MPFMA

As described in the previous section, the MPFMA is a combined operator. It provides partial support for binary64 accumulation on an FMA 32 structure. An interesting aspect of its feature is that its interfaces have the same size as binary64 addition or multiplication (larger than FMA32 and smaller than FMA64). Let us now study the integration of these binary64 operations in the MPFMA architecture.

Integrating the multiplication while keeping the fully pipelined structure is a challenge. Because a binary64 multiplication requires a 53×53 multiplier which is much more costly than its binary32 24×24 counterpart. Relevant results on multiplier area from Table 14.4 have been reproduced in Table 3.4. A 54x54 multiplier which is very close to the 53x53 multiplier required by binary64 multiplication is 4.5 times more costly that the 24x24 multiplier required by binary32 multiplication. This cost was too high for the first K1 area budget. So we discarded the implementation of a complete binary64 multiplier with the MPFMA as a basis. Instead we suggested to divided the full multiplication and to support it by an iterative process which will be described in details in Section 3.4.4.

Integrating the addition was easier. Because of the datapath widening implied by the MPFMA, the operator already supports the addition between a $2p$-bit and a q-bit wide numbers. Double precision addition is the sum of two q-wide numbers. The width differences between q and $2p$ is very narrow: only 5 bits. The MPFMA already contains the necessary circuit to round a result towards binary64, so no modification were required at this level. The exponent circuit is a little different since the addition operand that replaces the product is a new binary64 operand.

The cancellation cases are very similar between MPFMA and binary64 addition. Let us con-

sider that the new operand is statically positioned with the same left bound as the product in the MPFMA. It is 5 bits wider to the right. The cancellation can happen for exponent differences of +/- 1 which already fit into the cases considered for the MPFMA. In the case of binary64 addition, cancellation could also happen when operating on two subnormal operands. But, since these two operands share the same format, two subnormal numbers are positioned exactly at the same index in the datapath and fit within the pre-existing adder and LZC circuit. No extra modification are required here neither.

The MPFMA subnormal management issue, occuring with binary32 subnormal operands, does not occur for the binary64 addition. Indeed a binary64 subnormal number is necessarily lower than a binary64 normal number. Overall, supporting double precision addition does not implies a large widening of the MPFMA datapath. It constitutes an interesting feature which extends the FPU capabilities towards a wider support for binary64 operations.

K1's first FPU integrate binary64 addition support through a modification of the MPFMA structure. The performance of this operation is uniform with the binary32 operations provided by the MPFMA: 4-cycle latency and throughput of 1 cycle.

3.3.2 2D dot-product

During K1 FPU design we studied an other operator: the two-dimensional dot product. From four operands A, B, C, D, this operator computes $A \times B + C \times D$. We consider an implementation of this scheme with binary32 inputs, extended to support any combination of product addition/ subtraction . This operator already exists in the literature. For example, an implementation was suggested in [138]. We propose a novel implementation of this scheme with some specificities. Our design fully supports subnormal numbers. It also provides both a standard (binary32) and an extended precision output (binary64).

The general scheme of the binary64 output version of this operator is illustrated by Figure 3.13. This operator returns a result equals to the rounding of a dot product in infinite precision. This operator fully supports subnormal numbers with the same performance level as normal numbers. It is compliant with the IEEE-754 standard: it supports all 4 rounding modes and implements exception detection according to standard definition. It can be considered as a hardware implementation for a specific case ($n = 2$) of the **dot reduction operation** described in paragraph 9.4 of the standard [73].

Using dot-product

```
1    /* float inputs[4];
2    double acc = inputs[0] * inputs[1] + inputs[2] * inputs[3];
```

Listing 3.2: Sum-of-product of binary32 operands in binary64 precision

Listing 3.2 presents a program example which benefits from the dot-product operator. Moreover, using the dot-product to compute such a sum of binary32 products in binary64 precision, is C11 compliant.

Let us now focus on the possible benefit of the dot-product for generic scalar product evaluation. In large dimensions, dot-product offers no performance advantage over FMA (or over MPFMA for accumulation in binary64). Indeed, as the operator does not accept an accumulator as input, it has to be complemented by a floating-point add to perform the scalar product accumulation. With these two operations, two products are accumulated into a temporary accumulator: on average, one product is accumulated per operation. The FMA offers the same level of performance.

Description	Dimension	# Op.	Latency expression	Latency (cycles)	Throughput (cycle per result)
FMA	2	2	L_{FMA}+L_{FMUL}	8	2
+ 2D Dot-Product	2	1	L_{FDMA}	4	1
FMA	3	3	$2L_{FMA}$+L_{FMUL}	12	3
+ 2D Dot-Product	3	2	L_{FMA}+ L_{FDMA}	8	2
FMA *throughput*	4	2	$3L_{FMA}$+L_{FMUL}	16	4
FMA *latency*	4	4	L_{FMA}+L_{FMUL}+L_{FADD}	12	5
+ 2D Dot-Product	4	3	L_{FDMA}+ L_{FADD}	8	3
FMA *throughput*	5	5	$4L_{FMA}$+L_{FMUL}	20	5
FMA *latency*	5	6	$2L_{FMA}$+L_{FMUL}+L_{FADD}	16	6
+ 2D Dot-Product	5	4	L_{FDMA}+ L_{FMA}+ L_{FADD}	12	4

Table 3.5: Latency/Throughput for low dimension scalar products with and without FDMA (2D dot product instruction)

However, the dot-product operator becomes interesting in low dimensions. Latency and throughput results for various low dimension dot products are presented in Table 3.5. Those implementations are not numerically equivalent. To expose more instruction parallelism, the order of operations is not uniform over the multiple implementations. As floating-point arithmetic is not associative, numerical evaluation may differ across implementations. Both for throughput and latency the dot-product operator exposes better results than FMA.

Finally, 2D dot-product constitutes the basis for complex arithmetic implementation. The multiplication of two complex numbers, $a + i \times b$ and $c + i \times d$, can be implemented with two dot-product instructions, computing $a \times c - b \times d$ and $a \times d + b \times c$. In this case, the dot-product offers a straight advantage over the FMA, complex multiplication is reduced from 4 operations to only 2.

Micro-architecture

Let us now focus on the implementation of the dot-product operator.

As for the MPFMA, to enforce time predictability, we chose to design an operator with full subnormal support. In the binary32 output version, both intermediate products can have an exponent lower than the minimal result exponent. Thus the sum-of-product may be lower than the lowest representable numbers. However, contrary to the MPFMA, it can not be discarded into a sticky bit. The exact computation of this sum is still required to determine precisely the rounding direction.

Figure 3.14 presents one of the possible micro-architecture for the 2D dot product. The structure is very similar to a standard FMA. Thus we are only going to focus on the differences.

The main difference is the integration of a second multiplier. As for the MPFMA our design performs a pre-normalization on both multiplier outputs. Then our architecture performs a product ordering, before operating an alignment shift on the lowest product. In FMA design, ordering constitutes a solution to reduce datapath size. However, as FMA addend operand is available during the other operands multiplication, it is more latency-efficient to perform the alignment in parallel with the multiplication. Ordering will require to insert logic levels (for ordering and alignment shift) in the FMA critical path.

For the dot product, two multiplications are performed simultaneously, ordering has to be sequentialized with them. Without ordering, the alignment shift would be:

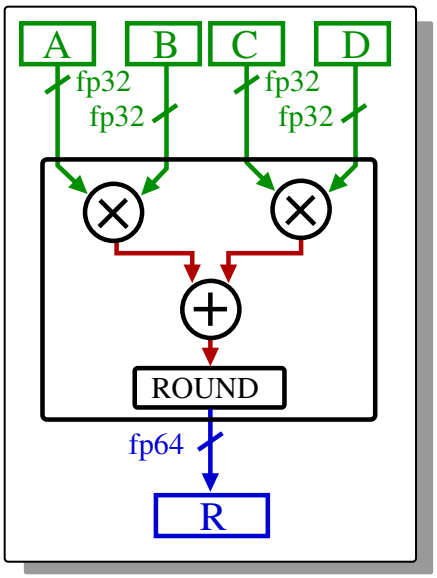


Figure 3.13: 2D dot-product operation scheme

Shift width (bits)	area 2ns latency	area 1.8ns latency
98	586	970
127	923	1797
148	1124	2058
206	1859	4463

Table 3.6: Shift area according to width and latency constraint (28nm process)

- $6p + 4$-wide without pre-normalization for binary32 output
- $2q + 4p + 4$-wide without pre-normalization for binary64 output

By ordering the operands, datapath width can be reduced to: Ordering reduces the alignment shift to:

- $4p + 2$ bits without pre-normalization for binary32 output
- $3p + q + 2$ bits without pre-normalization for binary64 output

We evaluated the possible gain of operand-ordering on the alignment shift, by synthesizing the corresponding shifters. Synthesis were made at two latencies: 2ns and 1.8ns. The first one represents low-constraint environment (non-exponential part of Figure 14.3, p 204) and the second one represents higher-frequency constraints. Assuming low constraints, ordering reduces alignment area by 48% for binary32 output and by 50% for binary64 output. In high constraints, the reduction is even larger: 53% for binary32 output and 60% for binary64 output.

Operands ordering introduces two e-bit wide exponent adder, one e-bit wide comparator and a single level of $2p$-bit multiplexers. The exponent logic can be parallelized with the multiplication: it has no effect on the critical path. Multiplexers have to be inserted in the critical path, but these are not expensive blocks. Operands ordering appears as a good solution when targeting low area. Its effect on latency should be studied further.

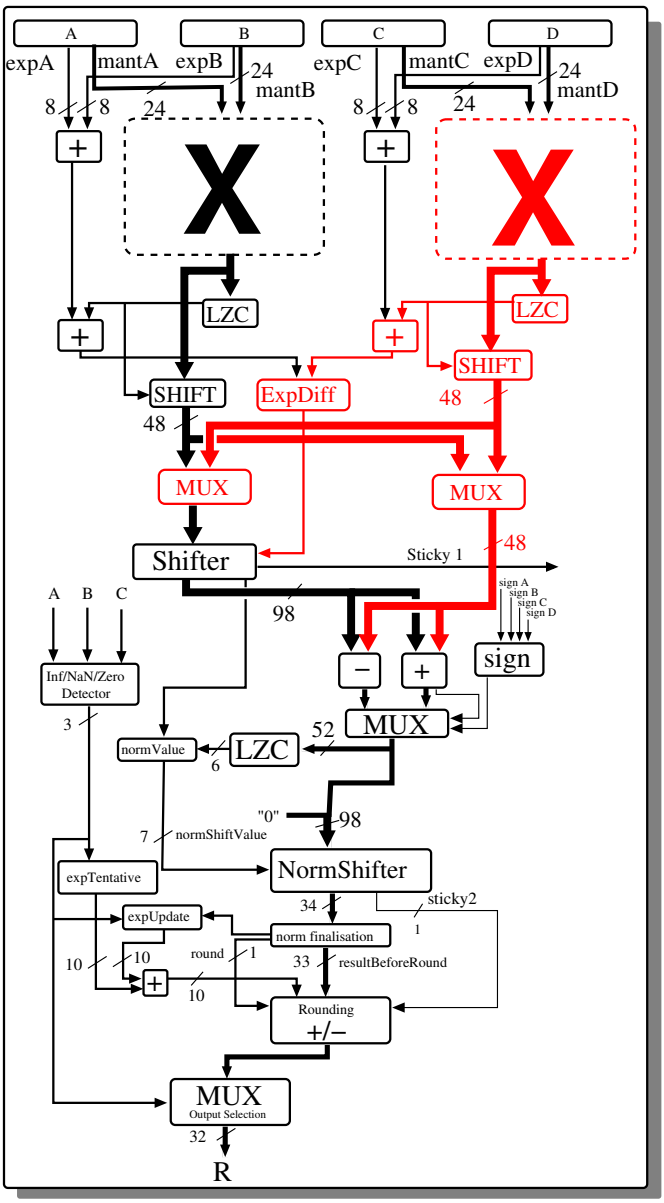


Figure 3.14: 2D dot-product micro-architecture (main differences with FMA32 are highlighted in red)

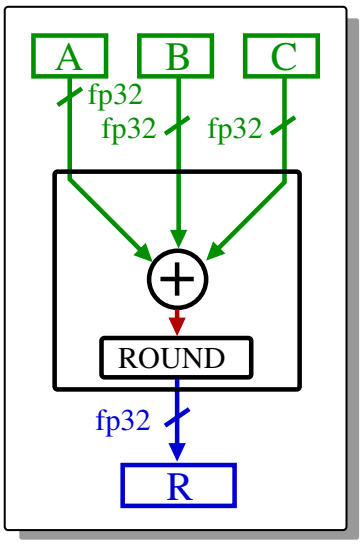


Figure 3.15: Triple-operand addition operation scheme

Conclusion

The dot-product provides performance improvements for low dimension scalar product or complex arithmetic. However, it does not outperform the FMA for large dimension dot-products: they both expose the same asymptotic accumulation capability, one product per operation.

We used the FMA architecture as a basis to build a dot-product operator. The fact that the addend is replaced by a product, makes alignment and multiplication parallelization impossible. Alignment has to be sequentialized with multiplication, which increases the latency of the operator with respect to the FMA. To support subnormal numbers, while limiting the datapath width, we introduced product ordering. Performed before the alignment, the ordering saves area.

The dot-product operator was integrated into K1 ISA. It was implemented as a 3-operand instruction: computing a result (binary32 or binary64) from two 64-bit registers, each containing a pair of binary32 numbers. To facilitate the compiler task, mostly register allocation, several variations of this instruction exist. They implement the required combination of operand swapping to implement complex multiplication without explicitly swapping the high and low part of the 64-bit input registers. (Assuming a 64-bit register contains both real and imaginary part of a complex number, respectively in its high and low part).

There exists a way to accelerate the large dimension dot-product while using a dot-product operator: adding a 3-operand add operator. Indeed such an operator, associated with the dot-product, provides the capability to accumulate four products using three operations, thus outperforming by 33% the FMA. Therefore, in the next section, we study the implementation of a 3-operand floating-point addition.

3.3.3 Triple-operand add

The triple-operand floating-point adder scheme is summarized by Figure 3.15. It performs the floating-point addition of 3 operands, with a single final rounding. Such an operator was already suggested in [145]. We contributed a new scheme, with subnormal support.

Using the triple-operand adder

A benefit of the triple-operand addition is to extend the dot-product use to large dimension accumulations. Indeed by using two dot-products and one triple add operations, four products can be accumulated with an intermediary accumulator. This scheme requires three operations, providing an accumulation of $\frac{4}{3}$ products for every operation. It outperforms both FMA-based and FADD/dot-product - based schemes.

A other use case of the triple-operand adder is the multi-addition. To add n inputs, a 2-operand add scheme requires $n-1$ operations. A 3-operand add scheme requires $\lfloor \frac{n}{2} \rfloor$ operations, outperforming by 50% the 2-operand add. In such accumulation, the triple-operand add could also help to improve the accuracy of the summation. Indeed, it performs less intermediate rounding than a 2-operand add scheme. However as precision is very dependent on cancellation phenomenon linked to operand summation order and dynamic, it is complicated to evaluate the possible benefit of the operator in such cases.

Architecture of the triple-operand adder

Let us now focus on some of the specificities of the datapath structure of the triple-operand adder. We suggest a different architecture from [145]. Nonetheless, both proposals rely on the same basic scheme:

- ordering the operands
- taking into account catastrophic cancellation that could occur when adding the two highest operands
- outputting a correctly rounded result for the overall operation

The ordering is performed by a mechanism similar to the one used in the dot-product operator. Exponent comparators, rather than mantissa subtractions, are used to provide a coarse order. We now distinguish three ordered operands $highest \geq middle \geq lowest$. The ordering is approximate, $\geq$ stands for *exponent greater than exponent of*. This order is sufficient for alignment purposes.

The alignment scheme of our design is illustrated by Figure 3.17. In our design, contrary to [145], the highest operand is statically positioned at index $p_{highest}$, two bits to the right of the highest weight of the datapath. Two extra bits are required to account for possible carry propagation. Since we have 3 operands, which could share the same exponent, the result could have up to two-unit exponent increases with respect to inputs' exponent. The alignment is performed as follows:

- The middle operand is shifted right from $p_{highest}$. The shift amount, δ_m, is determined by $\delta_m = min(e_{highest} - e_{middle}, p+2)$
- Then, the lowest operand is shifted right, also from the $p_{highest}$ position. The shift amount, δ_l, is determined as by $\delta_l = min(\delta_m + (e_{middle} - e_{lowest}), 2p+4)$.

The operation between the highest and the middle operands could result in a catastrophic cancellation. The architecture needs to provide at least p guard bits between the extreme alignment position of the highest and lowest operands. The shift amount computation has to be sequentialized since δ_l depends on δ_m.

As pointed out in [145], one of the difficulty of a correctly-rounded multi-operand addition is the computation of the exact sticky flag. All the operations studied previously (FMA, MPFMA, FDMA) performs an effective addition of two operands. The sticky is always computed by a linear logical-OR of the lowest operand bits discarded during alignment or normalization shifts.

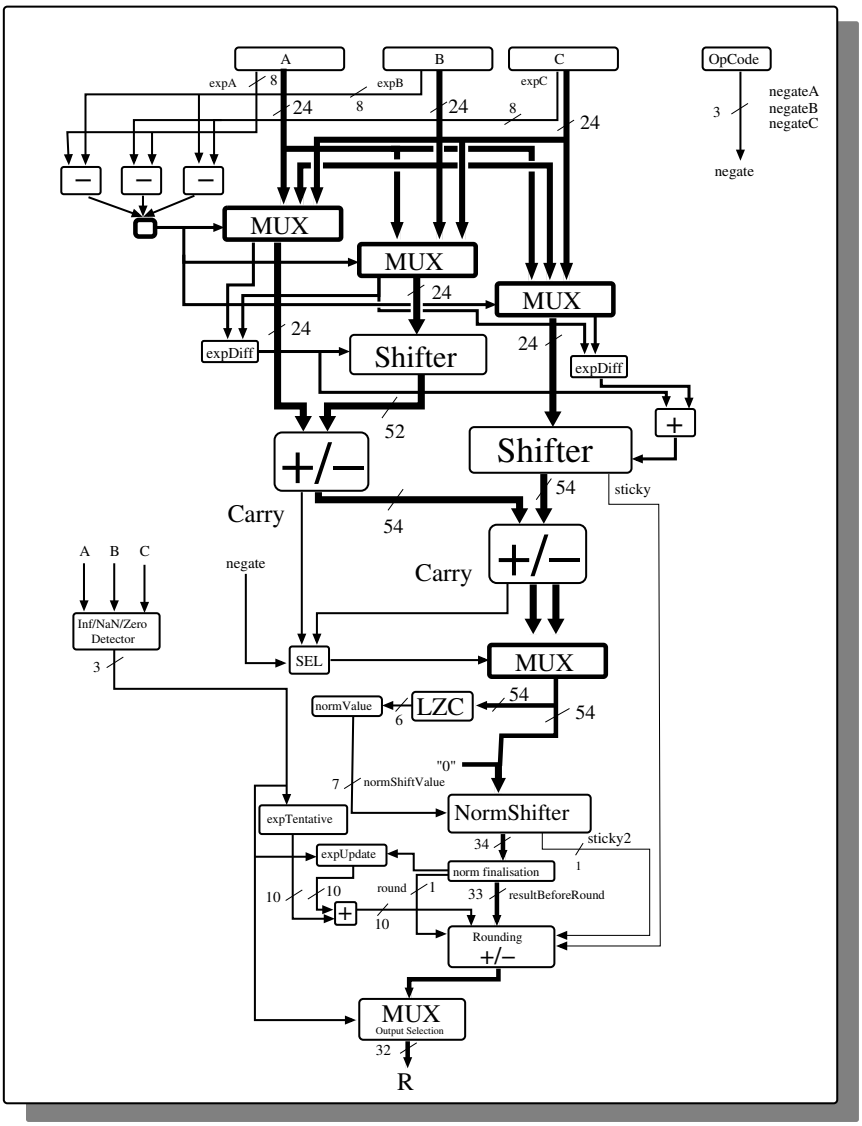


Figure 3.16: Triple-operand micro-architecture

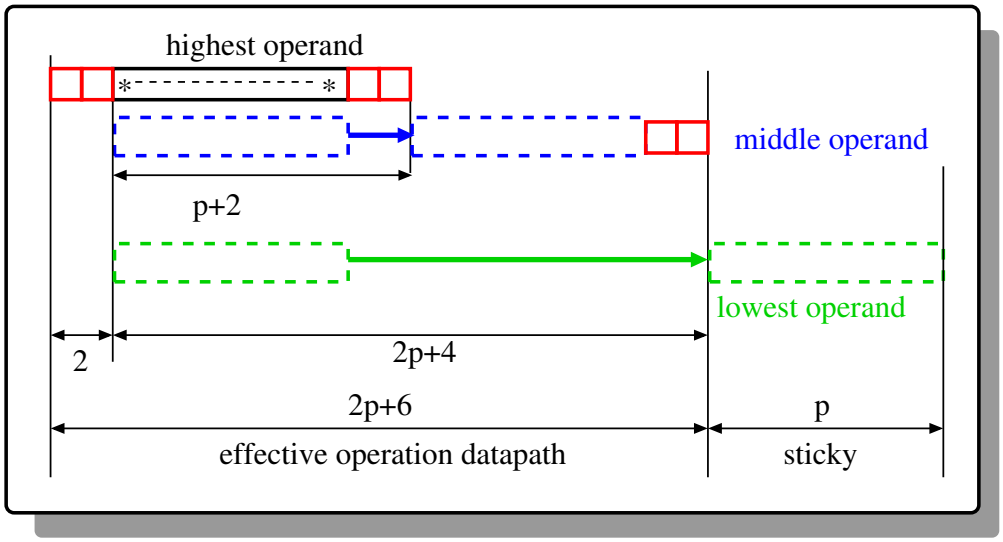


Figure 3.17: Alignment scheme for triple-operand add

In triple operand addition, $middle + lowest$ could be equal to zero. This cancellation could happen whatever $e_{highest}$ value. This means that, even in extreme case where $e_{highest} \gg e_{middle}$, the sum $middle+lowest$ must be computed exactly to determine the rounding direction. However, bits from the lowest operands, which are farther than index $2p + 6$ can be discarded into a sticky flag. The sum $middle + lower$ does not need to be computed separately, thanks to the guard bit inserted in the alignment paths, it can be part of a fused $highest + middle + lowest$. The triple-operand alignment datapath is $3p + 4$-bit wide for the lowest operand and $2p + 2$-bit wide for the middle operand. The effective operation path is $2p + 6$-bit wide.

When designing this datapath, there is one extra case that needs to be managed carefully: the complete cancellation between the highest and middle operands. This case is special because it implies that the lowest operands will be returned. In two-operand cases (FADD, FADD, FDMA), the result exponent can always determined with respect to the highest operand exponent. This it not the case when returning the lowest operands. The result exponent basis can not be determine before knowing the result of $highest+middle$. As this case only occurs with a perfect cancellation, this sum does not need to be computed separately. We suggest two ways to manage this $highest+middle$ complete cancellation case:

- using the normalization LZC to determine if the result could correspond to the lowest operand which would have been alignment shifted of $2p + 4$ bits.
- by detecting $|highest| = |middle| \wedge sign_{highest} \neq sign_{middle}$.

Let us justify the first solution. We exclude case with infinities and NaNs. We know that if the lowest operand is not shifted of $2p + 4$ bits during the alignment, then its position depends on e_{middle}. If δ_m has been saturated, then it is easy to see that $lzc \leq 3$ and $e_{result} = e_{highest} + 2 - lzc_{renorm}$. If δ_m has not been saturated then $\delta_l = e_{highest} - e_{lowest}$, the position of the lowest and the middle operands are directly related to unsaturated difference between their exponents and $e_{highest}$. We can safely use $e_{result} = e_{highest} + 2 - lzc_{renorm}$. In all cases, e_{result} is determined by $e_{highest}$. However if $\delta_l = 2p + 4$ and $lzc_{renorm} = 2p + 4$, this could hide a complete cancellation of $highest + middle$ and you must use $e_{result} = e_{lowest}$. Those considerations also apply when $highest = 0$, since all operands share the same format, this implies $e_{highest} = e_{middle} = e_{lowest}$ and the result exponent still depends on $e_{highest}$.

Let us now quickly evaluate the cost of those solutions. This first solution inserts a comparator in the critical path. Indeed, a few logic levels must be inserted between the normalization LZC and the result formatting circuit (which insert the exponent in the final result). The second solution is less invasive. The equality operator is easily parallelized with the alignment circuit.

The overall architecture of the triple-operand adder is illustrated by Figure 3.16. This scheme distinguishes $s = highest + middle$ and $s + lowest$, but in our implementation those are fused into a carry-save compression level followed by a 2-operand fast adder.

3.3.4 Conclusion on K1's first FPU structure

Triple-operand adder is difficult to integrate in an FMA-based FPU. An FMA is essential for modern FPU design, it is difficult to integrate a new operator if it does not fit into the general FMA scheme. The MPFMA fits nicely in such a structure, with very little architectural modifications. Triple-operand adder is too different from this structure. It requires operand-ordering and multiple alignments, making it not well-suited for FPU integration. For those reasons, the triple-operand adder was not integrated into K1 FPU. Dot-Product is also costly because it introduces an extra multiplier and it requires the alignment to be performed sequentially with the multiplication stages. For Kalray's MPPA it was decided to integrate both MPFMA and Dot-Product for efficient support of standard and complex floating-point arithmetic. Finally this first FPU supported: 32-bit FMA, MPFMA, 32-bit dot-product with binary32 and binary64 output, and binary64 addition. Section 3.5.2 introduces the new version of K1 FPU, scheduled for future products.

3.4 Building K1's FPU: miscellaneous contributions

The previous section was focused on describing new architectures and their specific implementation. Let us now focus on describing some on the more minor contributions and cooking rules used during the FPU design.

3.4.1 Automatic generation of bit-pattern detector

In Section 3.2.1 we studied the design of a Leading Zero Anticipator. This module provides significant latency improvements. Several designs of LZA have been suggested, [139] contains an extended taxonomy of the available methods. Among the most original, we could point out [156] which suggests a 3-input Leading Zero Anticipator. This LZA directly works on the carry-save outputs of the multiplier and the addend. It aims at reducing the critical path by bypassing the last compression stage before the main adder.

The common default of most LZA designs is the inaccuracy of their anticipation. LZAs output a leading zero count which could be off by one position. This inaccuracy is easily solved by inserting an extra multiplexer level after the normalization shift. This solution increases the critical path, which increases the operator latency. [157] addresses this problem by suggesting an exact LZA. This LZA computes both an approximate LZC and its error, so it can be corrected. It is an interesting response to the LZA inaccuracy problem. We wished to evaluate such a solution in our design, but such a circuit is complex to design and implement manually. This design detects input pattern which leads to an erroneous LZC. There are two main patterns to detected, one for error on positive results, one for error on negative results. These patterns are detected by dividing the input in two halves and detecting sub-patterns on each part. Each pattern and sub-pattern requires an independent detection tree. The complete system requires the detection of 11 sub-patterns. Implementing such a system manually is error prone and debugging is time consuming.

Our proposal is to automate the generation of such a circuit and more generally the detection of bit patterns. We suggest to develop a generator for recursive bit pattern detector. The idea is to abstract the bit pattern detection scheme from the RTL generation. The bit pattern detection scheme is described through regular expressions and rewriting rules.

Pattern expressions and systems rules

The patterns are described using the formalism described in 3.2.1, p40. This description is extended with a set of rewriting rules. These rules described the relation between the patterns, needed for the recursive detection generation. A rewriting rule is of the form $left+right \rightarrow result$: the pattern *result* is decomposed between a MSB pattern *left* and a LSB pattern *right*. The detection of *result* can be decomposed at the detection of *left* in the input higher half and *right* in the input lower half.

Our future perspective is to develop a system that generates the rules from the pattern set. This system should also be able to verify the coherency and the consistency of the system. For now you do not have an automated system to generate neither patterns nor rewriting rules. The developer has to make sure it described every rules and verify manually the system coherency.

To illustrate our mechanism, let us consider a simple example. We wish to determine the sign of an addition. Let us list the patterns that lead to a positive result:

- P^+G-^* : addition of a negative with a positive numbers with overflow
- $Z-^*$: addition of two positive numbers

The second expression is easy to detect, it only requires the sign bits of the two operands. However, the first pattern is a little more complex. It can be decomposed into the following sub-patterns: P^+G-^*, $G-^*$, P^+. Let us describe the rewriting rules used to build the recursive detection:

- $P^+ + P^+ \rightarrow P^+$
- $P^+ + G-^* \rightarrow P^+G-^*$
- $P^+ + P^+G-^* \rightarrow P^+G-^*$
- $P^+G\ -^* +-^* \rightarrow P^+G-^*$

The final pattern is detected by:

- dividing the input in two halves: H_{high} and H_{low}
- recursively detecting each sub-pattern which appears in the left part of a rewriting rule in H_{high}
- recursively detecting each sub-pattern which appears in the right part of a rewriting rule in H_{low}
- testing if the detected left and right sub-patterns, concatenated, match the left side of one of the four rules

The input matches the main pattern if, and only if, at least one the rewriting rules is matched at the last steps.

RTL generation

The framework we developed generates the final pattern and the sub-pattern detections. For each sub-pattern it is going to build a detection tree, based on the rewriting rules which match the sub-pattern. It starts by ordering patterns according to their minimal length which is the length of the minimal bit string matched by the pattern.

Then it considers the input divided in 1-bit chunks, that constitutes the first level. For this level, the framework considers all the patterns that could match a 1-bit string to build the leaves of the detection trees. Then the recursive process starts: bits are grouped by pair. The framework lists the rewriting rules that apply to 2-bit results and build the 2-bit pattern detection for each pair. Then pair are gathered two by two, and the process continues until the complete input string is built from its two halves.

Conclusion on bit-pattern detector generation

This simple process disconnects the RTL development from the system building. The developer can focus on describing a complete and coherent detection system, the framework takes care of the hardware generation. For now the system has to be manually verified. The proof of coherency and completeness are left to the developer responsibility? We are studying the possibility of automating this proof generation. A formal proof checker could be used to verify the regular expressions and rewriting rule system.

3.4.2 Management of subnormal numbers

Subnormal numbers are at the lower extremity of the range of representable numbers, defined by the IEEE-754 standard [73]. Let us lists the specificities of subnormal numbers, compared to normal numbers:

- subnormal's mantissa is not normalized
- subnormal's encoded mantissa is extended by an implicit zero rather than an implicit one
- subnormal's exponent is not $e_{encoded} - bias$, but is set to e_{min} instead

Moreover, an operation resulting in a subnormal result must raise the underflow exception, except if the operation result, assuming an unbounded exponent range, can be exactly represented by a subnormal number. Contrary to normal numbers, their mantissa is not normalized, and their exponent has the minimal possible value e_{min}. Let us list some of the difficulties inherent to subnormal support:

- The mantissa implicit bit must be determined dynamically. This has several consequences. For example, for floating-point multiplication, it increases the critical path on the mantissa multiplier.
- Contrary to normal numbers, the normalization shift applied to a subnormal result is not its leading zero count. It depends on the difference between its unbound exponent and e_{min}.
- The uncertainty on operand leading one position increases the uncertainty on result leading one position. Datapath width has to be adapted.

Most floating-point applications do not rely on sufficient accuracy to care about subnormal numbers. CPU manufacturers exploit that fact to provide a downgraded support for subnormal numbers, either by flushing subnormals to zero or by trapping to a software routine each time a

subnormal number is encountered. Both those solutions save hardware. The first one does not preserve the numerical behavior of the application. The second one swaps hardware complexity for software latency. This has a severe drawback: timing of a floating-point application may become hard to predict. Indeed, software management exposes the operation to memory dependencies. Memory timing are hard to predict and very dependent on the hierarchy level which will be addressed. Thus it is impossible to bound closely the execution time of any floating-point application that may encounter some subnormal numbers. Real-time constraints can not be enforced with such management.

For an embedded architecture, targeting real-time application, it is interesting to consider extending hardware support to subnormal numbers. The subject of efficient hardware support for subnormal was already studied. For example, [140] suggests some implementation tips to support subnormal numbers. Let us now evaluate the cost of extending the MPFMA to support subnormal numbers. From Section 3.3.1, we gathered the main changes required:

- extending the leading zero count from $q + 2$ to $2p + q + cte$ bits
- extending the main adder path from $2q + cte$ to $3q + cte$ bits
- extending the normalization shift from $q + cte$ to $2p + q + cte$ bits

In conclusion, subnormal support does not come cheap. It requires datapath enlargement, more complex case managements. It impacts latency and area. In the case of a general purpose processor there is no advantage to support subnormals. But for a real-time system where timing is critical, the hardware cost is balanced by the predictability.

3.4.3 Multiplier tiling

We are now going to present scheme of integer multipliers used to build the first version of K1 FPU. Due to an architectural choice, this first version had to reuse two 16×32 multipliers implemented for the integer multiply accumulate unit. These multipliers were built as monolithic blocks with single output. They fit in a single pipeline stage. This section presents the scheme of multiplier tiling used to support both floating-point and integer operations.

Let us first list the various multiplication schemes required by the floating-point units:

- Floating-point FMA and MPFMA, one 24×24 multiplier
- Floating-point dot product, two 24×24 multipliers
- Floating-point half binary64 multiplication, 27×53 multiplier

The functionality of the last item is discussed in more details in Section 3.4.4.

To implement all the required schemes, we integrated two 11×27 multipliers. The four available multipliers can be tiled to implement the required scheme, as illustrated by Figure 3.18. The 32×32 multiplier is required by integer instructions. The combinations are implemented through multiplexer for input selection and several adders to accumulate the intermediary tile results.

The integration of the new multipliers was a necessary step to support the dot-product operation. FMA and MPFMA could have worked with only the two 16×32 blocks. Moreover, the integration of these blocks provides support for a 27×53 multiplication. It corresponds to the half-size of a binary64 mantissa multiplication. Next section will study the integration of such operation.

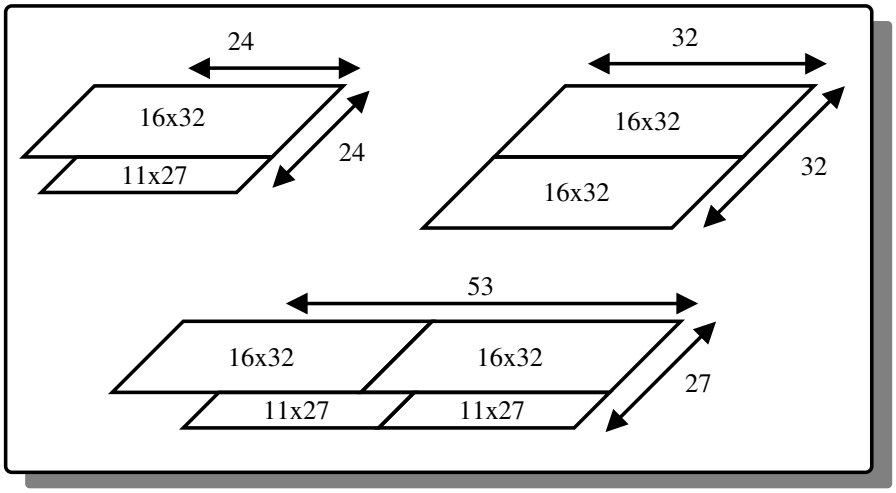


Figure 3.18: Tiling of 16×32 and 11×27 multipliers used to implement K1 operations

3.4.4 Iterative process of extended multiplication

In the first generation of Kalray's MPPA, the choice was made to focus on a fast implementation of binary32. Support for binary64 remained limited. The MPFMA provided partial support for binary64 accumulation, and it was extended to support binary64 addition. This section studies the implementation of binary64 multiplication in the first version of K1 FPU. The first part focuses on the support of subnormal numbers, which constitutes our contribution to the multiplication design. The second part studies the behaviour of a binary64 operation in K1 pipeline.

To multiply two operands A and B, our architecture implements the following mechanism:

- the mantissa of A is divided in high and low part: A_H and A_L
- $high = A_H \times mantissa_B$ is computed
- $low = A_L \times mantissa_B$ is computed
- $high + low$ is rounded to the result mantissa, exponent is computed from $e_A + e_B$

Thus the mantissa multiplication is implemented by iterating over the 27×53 tiling presented in Section 3.4.3.

Lets us now describe in more details our mechanism to support subnormal numbers. This mechanism computes a normalized mantissa result. If the result is subnormal, the multiplication result denormalization is not performed by the mechanism. The interest of this decomposition is to simplify the subnormal management. Let us now describe this mechanism.

Both inputs share the same format: binary64. The result of the multiplication between two subnormal numbers is lower than $2 \times e_{minfp64} = 2 \times -1023 = -2046 \ll e_{minfp64}$. The only relevant information when multiplying two subnormal numbers is the result sign and absolute comparison with respect to zero. Such a result can be entirely discarded into a sticky flag. It is only used to determine rounding: should the final result be 0 or the lowest possible subnormal number. No actual mantissa multiplication is required. Our mechanism discards this cases.

We can focus on cases with a single subnormal operand. In such cases, the result may be subnormal or may be normal. Our mechanism is designed to provide a post-mantissa multiplication normalized intermediary result. It proceeds as follows:

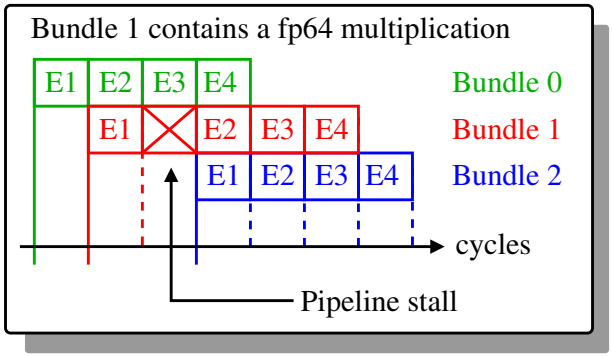


Figure 3.19: Iterative binary64 multiplication effect on K1's pipeline

1. The 1^{st} **cycle** is used to normalize the subnormal operand and to process the first multiply half
   (a) according to the input exponents, the subnormal operand D is selected, the other normal operand is tagged M
   (b) the first half of the multiplication is processed $R_H = D \times M_{high}$
   (b) in parallel a leading zero count l_D is performed on D
   (c) D is shifted left of l_D to get a renormalized D_r
2. The 2^{nd} **cycle** is used for the second multiplication half
   (a) the second half of the multiplication is processed $R_L = D_r \times M_{low}$
   (a) in parallel the first half R_H is shifted left of l_D bits to obtain R_{Hr}
3. At the beginning of the 3^{rd} **cycle**, the final result is obtained by accumulating $R_{Hr} + R_L$ in carry-save format

At the beginning of the third cycle a normalized result is provided and can be inserted into the more standard normalization/subnormalization, rounding datapath. Moreover the shifters used to renormalize D into D_r and shift R_H to R_{Hr} can be fused into one shifter, time-shared. The alignment shifter of the MPFMA can be reused for that purpose.

Through this mechanism our design is able to a full and transparent support for subnormal inputs:

- subnormal inputs are normalized during the multiplication
- the process has no effect on pipeline-latency: subnormal and normalized operands are processed with exactly the same latency and throughput
- subnormal outputs are managed by a more standard subnormalization/rounding circuit

Let us now consider the pipeline execution of a binary64 multiplication. As the operation iterates over the 27×53 multiplier, it forces the upstream pipeline to stall for one cycle. Indeed the multiplier is unavailable for other operations, since there is no dependency checking on the K1, the MAU pipeline has to be stalled to avoid resource conflict. Kalray K1 implements a VLIW architecture which executes instruction bundles to exploit instruction level parallelism. It does not support bundle desynchronization. Thus when a bundle contains a binary64 instruction, the

whole bundle has to be stalled one cycle. The next instruction bundle has to wait for two cycles (and not only one) before starting: this includes any other binary64 multiply. A binary64 multiplication can only be launched every two cycles. Its throughput represents 50% of the performance of the fully pipelined instructions (eg: FMA, MPFMA).

This is illustrated by Figure 3.19. 3 bundles are executed, **bundle 1** contains a binary64 multiplication, each square represents an execution stage of the bundles (from left to right E1 to E4). The result is committed into the register file at the end of E4. It can be started as could be expected one cycle after **bundle 0**. However it occupies the first pipeline stage during 2 cycles as the binary64 multiplication is stalling the pipeline for one cycle. **Bundle 2** can only be started two cycles after **bundle 1** beginning. As the stall occurs in the first pipeline stage and upstream, it only affects **bundle 1**. **Bundle 0** continues its execution without stalling and finishes two cycles before **bundle 1**. As E4 is performed at different cycle for each bundle, register write conflict is impossible. Similarly, no register conflict can happen between **bundle 1** and **bundle 2**. Moreover, as **bundle 2** start is stalled for one cycle, **bundle 1** and **bundle 2** do not occupy simultaneously the same execution stage. There is no pipeline conflict between these two bundles.

This operator was integrated into K1 FPU, and provides hardware binary64 multiplication support to K1 architecture. It required some pipeline modifications to implement the iteration on the half mantissa multiplier. It respects the time-predictability of the FPU: computation with subnormals is as fast as with normal numbers.

3.5 Testing, performance and perspectives

This section details three different subjects. The first part describes some of the tests designed to validate the floating-point design previously described. The second part summarizes the performance results of Kalray's K1 FPU. The third part concludes this chapter and present our perspective.

3.5.1 Operator testing

When our design was scheduled for integration into the MPPA, it became critical to develop an intensive test-suite.

The first part of operator testing was done before the integration into the core at a block-level. A Test-Bench was developed to wrap the FPU RTL description. This Test-Bench relies on a RTL simulator to simulate the FPU behaviour on test vectors. It compares the FPU output to expected values (result and exceptions). The test vectors are generated by a MPFR implementation. MPFR [110] is a C library that implements correctly-rounded arbitrary-precision arithmetic for floating-point numbers. It is a very valuable and reliable library used in large scale projects such as the Gnu Compiler Collection (GCC). In our case, it provides the emulation for the extended precision MPFMA and dot-product, which are non-standard operations.

The interface size of the FPU (between 96-bit and 128-bit input and 32 to 64-bit output) does not allow for exhaustive testing. The testing is performed by a CPU, which performs around 10^9 operations per second. Even if one test could be performed with a single operation it will necessitate 10^{30} seconds to test exhaustively the FPU. Even with 10^{11} CPUs, exceeding the number currently available on earth, it will still necessitate 10^{19} seconds which is over 10^{11} years (much more than the universe age estimated to 13.798×10^9 years [6]. Thus exhaustive testing is not realistic. The generation of test vector implements a directed-random. A uniform random generation is modified to target both arbitrary cases and error-prone cases such as large cancellation, subnormal/normal frontier, overflow and underflow values.

Once the FPU was integrated into the K1 core, a custom software implementation was developed. It was integrated into the Instruction Set Simulator (ISS) of Kalray's K1. This software implementation is faster than the MPFR-based emulation. It is similar to what provides a software floating-point library like FLIP [79]. This implementation is used to verify the FPU integration into the core, by comparing the RTL behavior with the expected behavior defined by the ISS.

Brute-force random-oriented testing can be extended by pairing it with code-coverage and error-injection RTL validation. Code-coverage is a feature of most recent RTL simulation tools.It checks if the totality of the RTL description is covered by the tests. It works by simulating the RTL behavior on the test suite and verifying that each description element is triggered at least by one test. If an element is not triggered by any test, the code coverage lists it as unverified, indicating that the test-suite should be extended to cover this case. The test-suite executed on the FPU allowed us to detect dead-part of the RTL description. Those parts were superfluous, and were simplified.

This system is very useful to detect both redundant parts of the description and holes in the test coverage, but it has its limits. Code-coverage does not assert complex conditions, defined over several elements of the RTL description. Fault-injection can be used to extend test coverage. Its mechanism is simple: it modifies the RTL description by inserting faulty components and verify that the test-suite raises an error. Possible modifications include changing the operators of a logical equation or the order of a multiplexer selections. The test-suite is executed on the modified RTL. A failure is expected. However there exist several reason the alteration could be undetected:

- if the test cover is not large enough;
- if the fault-created errors have not been propagated: the implementation is error-safe to this fault, the error has been detected and corrected inside the circuit;
- if the modifications do not create a faulty behaviour.

Thus, it is up to the designer (and the verification engineer) to study every undetected fault.

The best possible verification is formal proof: the FPU behavior is proven against formal assertions describing the IEEE-754 expected properties. It is also the more time consuming since there are few tools to provide such functionality. Some works have already tackled the problem [124].

K1 FPU was verified using random-directed test generation and test-coverage. We intend to extend the testing to fault-injection and formal verification in the future. Some efforts have already started to certify the behaviour of internal blocks such as the custom multiplier.

3.5.2 Summary and conclusion

The work presented in this chapter was used to design K1 FPU. This FPU mixes standard and new operators, implementing:

- all the standard binary32 operations (addition/subtraction, multiplication, FMA)
- the extended binary32/binary64 Mixed-Precision FMA
- a 2D binary32 dot-product with binary32 or binary64 outputs
- standard binary64 addition
- standard binary64 multiplication

Our design effort focused on supporting equally normal and subnormal numbers. The FPU design is pipelined over 4 stages (E1 to E4). A new floating-point operation can be launched every cycle,

operation	latency (cycle)		throughput (cycle per result)	
	normals	subnormals	normals	subnormals
add (fp32 and fp64)	4	4	1	1
mul fp32	4	4	1	1
mul fp64	5	5	2	2
fma, fdma, mpfma fp32	4	4	1	1

Table 3.7: Summary of K1 FPU performance

and finishes in 4 cycles. The only exception is the binary64 multiplication, implemented with a two-cycle iteration, it stalls the pipeline for one cycle.

Table 3.7 presents the latency and throughput results of the main operation supported in K1's FPU. It distinguishes performance result with and without subnormal numbers. As described in the previous sections, our implementation does not expose any difference between the two cases: subnormal numbers are supported at full-speed and with no impact on the throughput. The FPU runs successfully at 400 MHz in Kalray's MPPA-256, taped out in August 2012. A matrix multiply test case has been run successfully, demonstrating a performance by power ratio of 3.17 Gflops/W in binary64 precision and 8.05 Gflops/W in binary32 precision.

Our perspective is to continue improving the floating-point support of Kalray's MPPA. A second version of K1 FPU has already been implemented. It received both architectural and micro-architectural improvements. The main architectural modification was the support of a binary64 FMA. FPU interfaces were extended to provide 3 64-bit inputs, the 64-bit output was left unmodified. This new FPU provides 2-way binary32 vector operations, thank to an extra FMA32 integrated to exploit the triple 64-bit interface. The FPU still supports FMA32, MPFMA and 2D dot product. Let us now review some of the most significant micro-architectural modifications. We dropped the hard-macro multiplier, provided by the RTL synthesize, for a custom design. The synthesizer hard-macro does not provide multiplier with carry-save outputs. Thus the first FPU design had to sequentialized a single-output multiplier with the fast adder used for FMA effective operation. By implementing a custom multiplier design, we were also able to extract the carry-save outputs. We could then use a level of compression between the multiplier carry-save outputs and the addend, saving some logic levels on the critical path. Two implementation scheme were studied for the multiplier, both based on a Dadda scheme [148]: one based on Booth-recoding, a second relying on 4:2 counters. Our study showed similar results to [151], the 4:2 counters scheme provide the most efficient implementation, thus this scheme was selected for product integration. The second micro-architectural improvement was to integrate a Leading Zero Anticipator. We used the bit pattern generator, described in Section 3.4.1, to design part of this LZA. This new FPU has been scheduled for integration in a future version of Kalray's architecture.

CHAPTER 4
Software Floating-Point stack

The Floating-Point Unit introduced in chapter 3 does not implement the complete set of primitive operations required by the IEEE standard. Missing operations include division and square root and the mathematical functions listed in paragraph 9.2 of [73] such as the derivatives of exponential (exp, log, cos, sin). This list can be extended with many other interesting functions, including the error function family (erf, inverf), the gamma function. These functions are not listed in [73] but appears in other standards such as OpenCL [117]. For the division and square root, both hardware implementations [9, 13, 37, 120], and software implementations [35, 105] have been suggested. For the mathematical functions, some hardware implementations have been suggested, for example in [122]. But the literature is mostly extensive about software implementation, for instance [115], [52], [129]. For embedded systems, without heavy focus on complicated floating-point computations, the cheaper and still efficient software implementation makes more sense for both primitives and functions. This is especially true when software flexibility can be used to tuned up implementations to fit multiple special needs (precision, latency constraint, throughput-oriented ...).

This chapter starts by introducing several error concepts relevant to our work in Section 4.1. It goes on by studying several aspects of the software implementation of floating-point arithmetic for embedded systems. The first aspect is the optimized implementation of low-level floating-point primitives. This problem is tackled by Section 4.2 which describes the implementation of binary32 division and square root. The second aspect is the implementation of standard functions of the mathematical libraries. This problem is addressed in Section 4.3 through an example: the development of a floating-point exponential function based on a fixed-point core.

4.1 Implementation error

The functions studied in this chapter and the next one can not be computed exactly by a floating-point algorithm: they must be approximated. Implementing a function consists in implementing an approximation evaluation.

The inherent characteristic of an approximation is that it differs from the exact function. Both the approximation and its evaluation are sources of errors: the **approximation error** and the **evaluation error**. The implementation error or overall error is the sum of these two errors.

We can categorize an implementation according to the absolute value of its implementation errors:

- **correctly rounded**: a correctly rounded implementation returns the best possible floating-point result. For the standard [73] "best" is translated into the floating-point number that

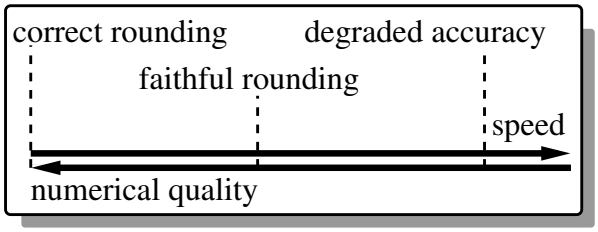


Figure 4.1: Correlation between function implementation speed and numerical quality

corresponds to the rounding according to the current rounding mode of the infinitely precise result. The interested reader can refer to Appendix 13, p.199 for more details on correctly rounded implementation.

- **faithfully rounded**: a faithful implementation returns one of the two floating-point numbers closest to the exact result. If the result is an exact floating-point number, it shall be returned. This generally corresponds to an error strictly less than one ulp. This is the implementation most often provided by mathematical libraries such as GNU LibC's libm [62].
- **degraded accuracy**: a degraded accuracy implementation is an implementation which does not fit in any of the previous categories. Such implementations are the most efficient, latency-wise and throughput-wise. They are not subject to accuracy constraints as strong as the other categories. Thus, they need to be used with caution.

Only correct rounding is uniquely defined, faithful implementations may differ. It makes comparisons and application portability more difficult.

As stated in the introduction of this section, the implementation error can be separated between the approximation error and the evaluation error. Let us define those errors and introduce possible methods to compute them.

Approximation error: ϵ_{approx}

The **approximation error** is the difference between the exact elementary function and the mathematical object used to approximate it. Let us consider an example to illustrate this error: we approximate a function f by p^*. The approximation error ϵ_{approx} is defined as:

$$\epsilon_{approx} = p^* - f \tag{4.1}$$

There are many different approximation methods. In this work we limit ourselves to the study of multiplication-based iterations and polynomial approximations. The interested reader may refer to [115] or [116] for a broader taxonomy of approximation methods. For some primitives, it exists an iterative method such as Newton-Raphson. This method will be described along the iteration algorithm in Section 4.2. For the other mathematical functions, we are going to use polynomial approximations.

As the elementary function can never be known with infinite precision except for specific inputs, it is hard to determine precisely the gap between an approximation and the exact function. The solution is to compute a bound of that error as tight as possible so it can be used to certify the expected implementation error. We rely on external software to compute such an error bound. Our choice has been to use Sollya ([27]). Sollya is a state of the art tool for the floating-point developer. It is described by its developers has "both a tool environment and a library for safe floating-point code development". Several PhD works have contributed to its development

[90, 25, 81]. When implementing a polynomial approximation, we use Sollya's **supnorm** function. This function, presented in [81, 26], is the state of the art to determine precise error bounds for polynomial approximations.

Rounding error: ϵ_{round}

Rounding error is also called evaluation error. It can be defined as the difference between the mathematical approximation and its evaluation in machine operations. Let illustrate the rounding error on our previous example: p^*, the approximation of f, is a polynomial with floating-point coefficients:

$$p^* = a_0 + a_1 \times x + a_2 \times x^2$$

We call p the binary32 evaluation of p^*.

$$\epsilon_{round} = p - p^*$$

If the only operations at our disposal are additions and multiplications which are described as $\circ(x + y)$ and $\circ(x * y)$, with $\circ$ the rounding operation from an infinitely precise result to binary32, then the evaluation p becomes $\circ(a_0 + \circ(x \times \circ(a_1 + \circ(x \times a_2))))$. If an FMA $\circ(x + y * z)$ is available then p could be $\circ(a_0 + x \times \circ(a_1 + x \times a_2))$, which contains fewer rounding operations. FMA can often be used advantageously to reduce the evaluation error.

An important concept to express rounding error is the **unit in the last place** introduced in Section 2.2. For the standard correctly-rounded floating-point operations (addition, subtraction, multiplication, FMA), the rounding error is bounded by the half ulp of the result.

As for the approximation error, some tools have been developed to compute the evaluation error. We choose to use Gappa [38] (*Génération Automatique de Preuves de Propriétés Arithmétiques*) which according to its description is "a tool intended to help verifying and formally proving properties on numerical program". Gappa can be used to verify or to determine evaluation errors in fixed-point or floating-point arithmetic. Gappa produces proofs of the results it returned. Those proofs can be verified by two proof checkers: HOL [67] or COQ [14], independently from Gappa.

Overall error: ϵ

The overall error or implementation error is the inaccuracy of the approximation evaluation. It is obtained by adding the approximation error to the evaluation error.

$$\begin{aligned} \epsilon &= p - f && (4.2)\\ &= (p - p^*) + (p^* - f) && (4.3)\\ &= \epsilon_{round} + \epsilon_{approx} && (4.4)\\ & && (4.5) \end{aligned}$$

By applying the triangle inequality to the previous equation, we get:

$$\begin{aligned} |\epsilon| &= |\epsilon_{round} + \epsilon_{approx}| && (4.6)\\ &\leq |\epsilon_{round}| + |\epsilon_{approx}| && (4.7)\\ & && (4.8) \end{aligned}$$

Thus, the overall error can be bounded by adding bounds on the approximation and evaluation error.

Carefully bounding the errors is important. Obviously an erroneous bound nullifies the implementation certification. But a bound too large often implies computing more precisely than required. There is few advantages in computing too precisely. Improving an implementation accuracy requires a more complex approximation (eg: higher degree polynomial with more operations) or a higher intermediary precision. Both those mechanisms have the same effect: increasing the latency and/or reducing the throughput of the implementation.

4.2 Optimized low-level floating-point primitives: division and square root

The IEEE standard requires the correctly-rounded implementation of 7 operations. Those operations, listed in section 5.4.1 of [73], are:

- addition
- subtraction
- multiplication
- conversion from integer
- fused-multiply add
- division
- square root

When designing a new ISA, the architect selects the implementation mean for each operation: hardware or software. A hardware implementation implies a specific encoding in the instruction set and the operator implementation in the CPU's functional units. It is costly in development time but brings the best performance (latency, throughput and power consumption). It represents a good choice for the most used primitives such as addition, multiplication and FMA. In embedded systems, for cost reasons, it is often limited to those simple operations. As described in Chapter 3, Kalray's MPPA implements in hardware the first five operations including an FMA.

Complex operations, such as division and square root, are less frequent. Moreover, they are most costly to implement in hardware. Software implementation is easier, less development-time consuming. Software provides more flexibility to optimize implementation for specific needs (latency/throughput oriented, accuracy). Thus for those operations, embedded system designers most often chose the software implementation.

The literature distinguishes two main families of division and square root software implementations:

- addition-based iteration
- multiplication-based iteration

The most well-known example of addition-based iteration is the SRT algorithm ([136, 147, 65]) that computes the division by a digit recurrence. Those digit recurrence iterations are widely used for integer arithmetic ([149]) and hardware implementations ([120]). These iterations converge linearly and require some conversions between integer and floating-point. Newton-Raphson (a.k.a Newton's method), a multiplication-based method, is more advantageous. It converges quadratically and can be implemented in pure floating-point arithmetic. Moreover it can exploit the FMA of K1's FPU ([34, 98]). For those reasons, we focus our study on Newton-Raphson method. The remaining of this section describes the development of binary32 division and square root. Division, specifically the reciprocal computation, is addressed in more details.

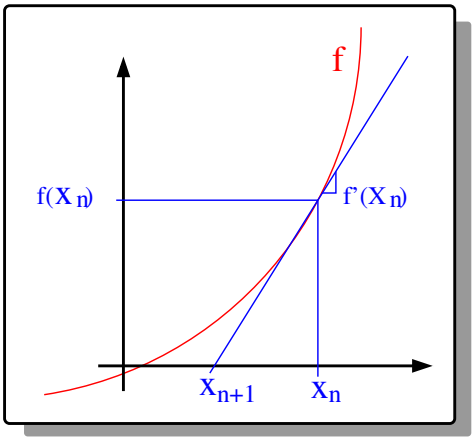


Figure 4.2: Newton-Raphson method

4.2.1 Newton-Raphson iteration

The IEEE standard [73] only recommends a correctly-rounded implementation for most mathematical functions. This is not the case for division and square root. They are considered as standard floating-point operations and must be implemented with correct rounding in each of the 4 rounding modes. Previous works [34], have shown that correctly-rounded implementation of both division and square root could be obtained using an FMA. For example, it was used to implement division and square root on Intel's Itanium [35].

For a more complete study on Newton-Raphson, the interested reader can refer to [106, 35, 34, 84, 116]. Let us now describe the method basis before going into implementation details. Let f be a differentiable function. The method goal is to determine a zero of f: x such that $f(x) = 0$. The method is an iteration, it starts with a coarse approximation x_0 of x determined using an external methods (oracle). Each iteration improves the accuracy of x approximation. The iteration are based on the following identity, illustrated by Figure 4.2:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.9}$$

To implement the division $\frac{b}{a}$, we start by computing $\frac{1}{a}$ To do so, we apply Equation 4.9 to $f(x) = a - \frac{1}{x}$ which is equal to 0 when x is the reciprocal of a. f is differentiable and defined everywhere except in 0, $f'(x) = \frac{1}{x^2}$. By applying Equation 4.9, we get the following iteration for the reciprocal:

$$\begin{aligned} x_{n+1} &= x_n - \frac{a - \frac{1}{x_n}}{\frac{1}{x^2}} && (4.10)\\ &= x_n - a \times x_n^2 + x_n && (4.11)\\ &= 2 \times x_n - a \times x_n^2 && (4.12)\\ &= x_n + x_n \times (1 - a \times x_n) && (4.13) \end{aligned}$$

Equation 4.13 contains the two expressions used for the Newton-Raphson method implementation. First an approximation error computation $e = (1 - a \times x)$ where x is the previous value of the reciprocal approximation and then a correcting expression: $x_{new} = x + e \times x$. We can notice that both expressions can be implemented by an FMA. In fact, an FMA is necessary to ensure correct rounding [104, 34, 116].

As stated in the introduction, Newton-Raphson iteration is said to be of quadratic convergence. This fact can be intuited by deducing the following equality from Equation 4.13:

$$(\frac{1}{a} - x_{n+1}) = a \times (\frac{1}{a} - x_n)^2 \tag{4.14}$$

Equation 4.14 demonstrates that the error of an iteration is proportional to the square of the previous approximation error, which is the definition of the quadratic convergence. From a more practical point of view, it means that each execution of the iteration doubles the number of correct digits of the current approximation. More details can be found in the previously cited references: [106, 104, 35, 34].

Let us now consider the implementation of square root. Several methods, based on the Newton-Raphson algorithm, exist to compute the square root of a. We will describe two of them. The first method uses $f(x) = x^2 - a$ which by application of the equation 4.9 gives us the following iteration:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

The drawback of this iteration is that it uses a division.The software implementation of division makes it a slow operation. we want to favor iterations with only FMA-based (addition, subtraction, multiplication) operations.

The second method first computes $\frac{1}{\sqrt{a}}$ before multiplying it by a to obtain $\sqrt{a}$. This second method uses $f(x) = \frac{1}{x^2} - a$ which injected into equation 4.9 gives the following iteration:

$$x_{n+1} = \frac{x_n}{2}(3 - ax_n^2) \tag{4.15}$$

Division by 2 can be exactly translated into a floating-point multiplication by 0.5. This iteration only relies on additions and multiplications which makes it pertinent for implementation on the K1 core.

4.2.2 Binary32 iteration implementation

The two Newton-Raphson iterations described in equation 4.13 and 4.15 constitute the basis for our implementation of the floating-point division and square root.

	computation of $\frac{1}{b}$
1.	$rcp = recapprox(b)$
2.	$e = \circ(1 - b \times rcp)$
3.	$R = \circ(rcp + rcp \times e)$
4.	$e = \circ(1 - b \times R)$
5.	$R = \circ(R + e \times R)$
6.	$e = \circ(1 - b \times R)$
7.	$R = \circ(R + e \times R)$

Table 4.1: Binary32 implementation of $\frac{1}{b}$ based on Newton-Raphson algorithm

Let us consider the implementation of the division of two binary32 numbers: a and b. The operation $\frac{a}{b}$ is implemented in two steps:

1. $R = \frac{1}{b}$ is computed with correct-rounding accuracy

2. R is multiplied by a using self-correcting iterations to get the correctly rounded $\frac{a}{b}$ ([106, 104, 35, 34])

The second step is not part of the Newton-Raphson method. We focus our study on the reciprocal $\frac{1}{b}$ computation. The implementation of this reciprocal is presented in Table 4.1.The initial approximation is provided by a specific instruction *recapprox* which is described in details in Section 4.2.2. The approximation is refined 3 times. We will demonstrate in Section 4.2.3 that this is sufficient for correct rounding. Each iteration implements with an FMA the two steps of Equation 4.13.

	computation of $\frac{1}{\sqrt{a}}$
1.	$rcp = recsqrtapprox(a)$
2.	$h = \circ(0.5 \times rcp)$
3.	$s = \circ(a \times rcp)$
4.	$d = \circ(0.5 - s \times h)$
5.	$h = \circ(h + d \times h)$
6.	$s = \circ(a \times rcp)$
7.	$d = \circ(0.5 - s \times h)$
8.	$h = \circ(h + d \times h)$
9.	$R = \circ(2 \times h)$

Table 4.2: Binary32 implementation of $\frac{1}{\sqrt{a}}$ based on Newton-Raphson algorithm

Let us now consider the implementation of the reciprocal square root. The implementation has been reproduced in Table 4.2. It contains 2 iterations. Each iterations uses 3 FMAs to implement Equation 4.15, as follows:

- $h = \frac{x_n}{2}$
- $s = a \times x_n$
- $d = \frac{1}{2} - \frac{a \times x_b^2}{2}$

Let us now go into more details about the implementation of the oracle to determine the initial approximation value.

Initial approximation implementation: hardware and software co-design

Both implementations start (at **line 1.**) by retrieving an initial approximation value of $\frac{1}{a}$ for division and $\frac{1}{\sqrt{a}}$ for square root. These values constitute the starting point of the first Newton-Raphson iteration **line 2. and 3.** for reciprocal and **line 2. to 4.** for square root.

There are three problems related to the initial approximation:

- select an accuracy for the initial approximation values
- chose those values
- implement their retrieval

In our implementation the initial value has 9 bits of accuracy and is accessed by a 7-bit tabulation index. With such accuracy three Newton-Raphson iterations are required to converge to a

description	code size (# bytes)
7-bit index table	256
full division (w/o table)	880
division core (extracted from full)	56

Table 4.3: Code size for K1 binary32 approximation table and division function without table

description	latency (#cycles)	
index extraction	1	
table load	*hit*	*miss*
	2	9
full operation	4	11

Table 4.4: Latencies for Newton-Raphson initial approximation loading from memory, in K1 core

correctly rounded reciprocal. We will not go into detail about the choice of starting values. The interested reader can refer to [84] for more information.

The second challenge: retrieval of the initial approximation, is of more interest to us. There are several solutions to implement initial value retrieval. The lowest development cost solution is tabulation in the main memory. The highest 7 bits of b's mantissa are used to address a table containing the initial approximation values.

This is the easiest solution since it does not require any specific hardware support. It simply relies on the CPU's load from memory instruction and some bit level manipulation to build the index from b. However it can prove very inefficient: it increases both code size and the latency. Indeed integrating a n-bit indexed table, storing 9-bit values, increases the function code size by $2^n \times 2$ bytes. In our implementation such table would cost 256 extra bytes. Table 4.3 presents this result along the code size of the current division program without memory table. Using memory table would result in a 29% code size increase with respect to a purely computabional implementation. This does not take into account the extra instruction required to prepare the index and read from memory. In embedded systems where memory is limited, especially low-level cache, storing a table in memory is detrimental to the implementation performance.

The second inefficiency is the latency impact. Computing the index is relatively easy in K1 architecture. There is no specific floating-point registers: general purpose registers are used to stored floating-point values. Thus bitfield manipulation are available on those register to extract the index bit field. This is not the case for some architectures (for example x87) which require conversion through memory before bit field processing can be performed on floating-point numbers. Reading a value from memory is always a costly operation. If the value is not present in a low level cache, it will required to be fetched from a higher level of the memory hierarchy, inducing at least a few tens of cycle of extra latency. For example in Kalray's K1, the latency of a memory access, shown in Table 4.4, is 11 cycles. This rather short latency, due to the use of local shared memory, still impacts performance. The memory access latency can be reduced if the data has been loaded in a low-level cache. This is for example the case when repeating, at close intervals, the execution of the primitive. The first execution misses and loads the data into low-level cache which can then benefit future executions, as long as the data is not discarded from the cache. As the table is larger than a typical cache line (32 bytes for K1 architecture), it takes several misses in different part of the table to load it completely into low-level cache. The cache mechanism does not improve the worst execution time which is computed with the highest possible latency of a memory access.

description	area (# cells)
K1 cpu core	153379
K1 FPU	33919
reciprocal seed	408
reciprocal table	187
rec. square root seed	233
rec. square root table	195

Table 4.5: Synthesis result for hard-coded approximation tables and seed operators

On embedded systems were latency in general, and real time constraint in particular, is relevant, memory storing is detrimental to implementation performance.

An other solution, to store and load data from the initial value table, is the hardware implementation. Indeed, such a table (with a limited index and data size) can stored into a hard-coded table. This hard-coded requires the implementation of a hardware operator, which we called a seed operator. This seed operator can be advantageously extended to provide more functionalities. For example, it can manage specific values (NaN, infinity) or detect and flag numerical conditions (eg: inputs leading to an overflow or underflow in the Newton-Raphson iterations). This operator is accessed by a new instruction in the ISA. This solution has been implemented in Intel's Itanium [35] and IBM's Power architecture [106].

To avoid memory dependency, the K1 architect selected the hardware solution. We implemented it by two seeds operators: one for division and for the reciprocal square root. Each seed implements its own hard-coded table of initial approximation values and is accessed by a specific instruction to K1 ISA. To ensure convergence of the Newton-Raphson method, the least significant bit of the initial approximation mantissa must be set to 1 [34]. Our seed operator set this bit to 0 to flag an invalid seed and we use the other payload bits to distinguish between cases. Initially, we implemented a 2-operand seed instruction for the division and a 1-operand seed instruction for the reciprocal square root. The division seed contains a reciprocal initial approximation table. It inputs the two operands of the division to perform specific and numerical condition detections. The division seed was, in a later version of the architecture, extended to support the unary reciprocal operation. The reciprocal square root seed is used to start both reciprocal square root and square root iterations.

Table 4.5 presents some area results for the seed operators implemented in the K1. This area is compared to the sizes of a complete processor core and the floating-point unit. The reciprocal square root seed is smaller than the reciprocal seed since it is a unary operation while the reciprocal seed is a two-operand operations Those operators are small compared to the core. Integrating them did not impact K1 area.

4.2.3 Code example and error study

```
1 __inline__ float raphson_rec(float a) {,
2     // ire contains the initial low-accuracy approximation
3     // of 1/b obtained through the seed instruction
4     float ire;
5     float err, re; // relative error and reciprocal approximation
6
7     ire = __builtin_k1_fsinv(a);
8
9     // if ire's LSB is one, the standard path is executed
```

```
10     if (fix32(ire) & 1 == 1) {
11         // main iteration
12         // 1st iteration
13         err     = __builtin_k1_ffmsrn(1.0f, b, ire); // err = 1.0f - b * ire;
14         re  = __builtin_k1_ffmarn(ire, err, ire);    // re = ire + err * ire;
15         // 2nd iteration
16         err     = __builtin_k1_ffmsrn(1.0f, b, re); // err = 1.0f - b * re;
17         re  = __builtin_k1_ffmarn(re, err, re);     // re = re + err * re;
18         // 3rd iteration
19         err     = __builtin_k1_ffmsrn(1.0f, b, re); // 1.0f - b * re;
20         re  = __builtin_k1_ffmarn(re, err, re);     // re = re + err * re;
21
22         return re;
23     } else {
24         // specific cases management
25         (...)
26     }
27 }
```

Listing 4.1: K1's software implementation of binary32 reciprocal core based on Newton-Raphson method

Listing 4.1 shows the code for the core part of the binary32 reciprocal as implemented for Kalray's K1. **__builtin_k1_ffmarn** and **__builtin_k1_ffmsrn** are specific flavours of the FMA instruction. They will be described in details in Section 4.2.4. For now we can just consider them equivalent to **__builtin_k1_ffma** and **__builtin_k1_ffms** which are the GCC bindings to the floating-point instruction Fused-Multiply and Add and Fused-Multiply and Subtract in binary32 precision. The parameter **ire** is an initial approximation to the reciprocal of b provided by the seed instruction introduced in 4.2.2. It is exact up to the 7th bit of mantissa.

```
1     @rnd = float<ieee_32, ne>;
2
3     R = 1 / x;
4     # first iteration
5     err1 = rnd(1 - x * r0);
6     r1 = rnd(r0 + err1 * r0);
7
8     # second iteration
9     err2 = rnd(1 - x * r1);
10    r2 = rnd(r1 + err2 * r1);
11
12    # third iteration
13    err3 = 1 - x * r2;
14    r3 = r2 + err3 * r2;
15
16    err3_eval = rnd(err3);
17    r3_eval = rnd(r2 + err3_eval * r2);
18
19    # goal
20    {x in [1, 16777215b-23] /\ r0 - R in [-1b-7,1b-7]-> (r3_eval - R) / R in ? }
21
22    # hints
23    1 - x * r0 -> 1 - x * (r0 - R) - 1;
24
25    r1 - R -> (r1 - r0 * (2 - x * r0)) - (r0 - R) * (r0 - R) * x;
26    r0 * (2 - x * r0) -> r0 + (1 - x * r0) * r0;
27
```

```
28    1 - x * r1 -> 1 - x * (r1 - R + R);
29    1 - x * r1 -> 1 - x * (r1 - R) - 1;
30
31    r2 - R -> (r2 - r1 * (2 - x * r1)) - (r1 - R) * (r1 - R) * x;
32    r1 * (2 - x * r1) -> r1 + (1 - x * r1) * r1;
33
34    1 - x * r2 -> 1 - x * (r2 - R + R);
35    1 - x * r2 -> 1 - x * (r2 - R) - 1;
36
37    r3 - R -> (r3 - r2 * (2 - x * r2)) - (r2 - R) * (r2 - R) * x;
38    r2 * (2 - x * r2) -> r2 + (1 - x * r2) * r2;
```

Listing 4.2: Gappa proof for the error of our implementation of Newton-Raphson method for the binary32 reciprocal

Let us now provide an error certification of our implementation of the binary32 reciprocal function. Listing 4.2 presents the Gappa script used to prove the evaluation error for the reciprocal code. Here, Gappa is used to determine the complete implementation error, sum of the approximation and the evaluation error. This is possible because the division is a function known to Gappa which allows us to ask it directly for the difference between R and r where R is the exact division result and r is the result of our implementation method.

The Gappa script is divided in three sections:

- definitions, from line 1 to 18: line 1 defines an alias to the rounding mode and the following lines define the iteration as performed by our implementation
- goal, line 19 to 21: a goal defines the objective numerical property. It assumes two hypotheses: $x \in [1, 2]$ and the initial approximation error is less than 2^{-7}. The unique goal is to determine a bound for the difference between r3 (result of the third iteration) and R (exact division).
- hints, from line 21 to 38: hints are indications to the Gappa rewriting engine. They show possible transformations of the expression to ease the goal determination. In our case, we unroll 4.14 to help Gappa identifying the Newton-Raphson quadratic convergence property.

Executing that script through Gappa provides the following range for the error $\frac{|r-R|}{R} \in [-2^{-24}, 2^{-24}]$. It corresponds to the expected half ulp accuracy. This proof can be generalized to the whole division routine and to the 4 rounding modes required by the standard [73]. For a manual, and more in depth, version of such a proof, the interested reader can refer to [98] or [34].

4.2.4 ISA extension to improve multiple rounding modes and correct exception support

Let us now present some modifications contributed to K1 ISA to improve the support of rounding modes and exceptions. The IEEE standard requires that division and square root be implemented correctly rounded for each one of the 4 rounding modes. The implementation must also set the floating-point exceptions listed in Section 2.1.2. Newton-Raphson core implementation must use rounding to nearest to converge to the correctly rounded result. Directed rounding mode is supported by changing twice the current rounding mode during program execution. Once at the beginning of the iteration: the current rounding mode is saved, and rounding to nearest is set. Once at the end of the iteration, before the last operation: the saved rounding mode is restored. Thus most of the computation is done using rounding to nearest, and eventually (if needed) the rounding mode is modified towards the current rounding mode to provide a correctly-rounded result. A similar mechanism has to be design for exceptions. Exceptions, specifically inexact, set during

Description	latency (cycles)	speedup	throughput cycles by result	throughput improvement
division without RN	70	0%	25	$\times 1.0$
division with RN	45	+55%	13	$\times 1.92$

Table 4.6: Comparison of K1 division implementation with and without RN instructions

the core computation are meaningless and must be discarded. [34] has shown that the inexact exception is correctly set by the last FMA operation. Moreover, [34] considers that intermediary steps do not raised any exception flags. In a first version of our implementation, exception status was saved before execution of the primitive core and restored just before the final operation.

Changing rounding mode or saving/restoring exception status, during a computation, impacts the implementation performance. Moreover, for rounding mode change, compilers have difficulties identifying the correct dependencies and may generate a faulty instruction scheduling [107]. Rounding mode and exception flags are generally stored in a status register. In K1 architecture, this status register is called the compute status (CS). Status registers (SR) differ from general purpose registers (GPR) used to stored computation data.

In Kalray's K1, status registers are accessed through the Branch and Control Unit (BCU). There is no hardware check to ensure the behavior after modification on a status register. To ensure execution coherency, a barrier instruction must be inserted after each status register modification. The barrier flushes, indifferently, both the execution pipeline and the memory system. This delay of 4 to 5 cycles only applies to the most favorable cases. It the most favorable case, it stalls the execution for 4 to 5 cycles, depending on the instruction previously executed. But if the BCU was waiting on other events (e.g. memory access after a cache miss), the pipeline is stalled until all current pending actions are terminated (up to hundreds of cycles).

Division and square root must modify a status register (CS) twice: once to save the rounding mode/exceptions and set round to nearest, and once to restore rounding mode and exceptions. These modifications must be followed by two barrier instructions which stall the pipeline. As our implementation do not execute any memory access, only the 4 to 5 cycles stall applies. This flush is short enough not to impact performance in most cases. However the division/square-root routines are very optimized and short-latency primitives: their latencies are inferior to a hundred cycles. The two stalls implies an overall 8% increase in latency.

To circumvent this latency increase, K1 ISA has been extended with "RN" flavoured Fused Multiply-Add instructions. Those instructions do not set any exception flag and do not depend on the CS rounding mode. RN instructions always apply rounding to nearest. Thus these instructions can be used for the core of the Newton-Raphson algorithm implementation: alleviating the requirement for rounding mode and exceptions saving and restoring. They reduce the latency of the division and square root routines. This reduction is illustrated, for the division, by Table 4.2.4. We measure a 55% latency improvement and a 92% throughput improvement. The implementation cost of those instructions is insignificant: some instruction decoder extensions and a mux for FPU rounding mode input. These instructions are also useful for elementary function implementations in directed rounding modes, and for supporting efficiently interval arithmetic.

4.2.5 Performance result

Based on the core iterations described in Section 4.2.2, the seed operator described in Section 4.2.2 and the ISA extension introduced in Section 4.2.4, we implemented the complete routine for reciprocal, division and square root. Those routines support all four rounding modes, specific

description	latency (cycles)	throughput (cycles per result)
K1 reciprocal	25	8
K1 division	45	13
K1 square root	34	14
FLIP division (ST 231)	35	35
FLIP square root (ST 231)	21	21
ARM Cortex-A9 fdiv	15	10
ARM Cortex-A9 fsqrt	17	13

Table 4.7: Performance of several implementations of binary32 reciprocal, division and square root

cases and set exception flags. Table 4.7 summarizes the evaluation of those implementations, compared to another software implementation FLIP and to a hardware implementation (ARM Cortex-A9). Let us remind that FLIP is an independent software implementation, made on a different architecture (ST231) with no floating-point support. FLIP reaches a lower latency by using the ST 231 integer pipeline which is shorter than K1 floating-point pipeline. It exposes a worst throughput because the ST231 ISA does not contain a multiply accumulate: K1's FMA is more throughput-efficient that the two instructions multiply and add used by FLIP. The hardware implementation is better on all aspects. ARM Cortex A9 is 3 times faster for the division, and 2 times faster for the square root. Our implementations stay more competitive for throughput: ARM Cortex-A9 is more efficient by 29% for the division and 7% for the square root.

To conclude, we implemented division and square root by: selecting an algorithm which could benefit from the FPU design, improving its implementation with low cost hard-coded seed instructions and ISA extensions. We are currently studying an integer-based implementation for binary32 latency-oriented division. Simultaneously, we are developing a binary64 division that uses fixed-point arithmetic, since K1 binary64 support is too limited to efficiently implement the complete iteration. This analysis may evolve with newer MPPA versions since they should provide a binary64 FMA, but integer-based schemes may still be relevant for latency-oriented implementations.

4.3 Introduction to the development of mathematical functions

As stated in the introduction, the IEEE standard [73] recommends more than the implementation of generic floating-point primitives. It also recommends the implementation of correctly-rounded mathematical functions. Very few implementations ([3, 72]) effectively provide a correctly rounded implementation. The vast majority (eg: the libm integrated into GNU's libc [62]) provides faithful implementations at best. The cost of correct rounding may seem inappropriate for embedded systems, but for critical applications, reproducible results can balance the implementation cost.

For now we are going to focus on not correctly-rounded implementations. A huge number of such implementation of those functions can be found in the literature. Among them, a few consist in hardware implementations (eg: [122]). There are example of commercial use of these work: for example the specialized function unit used in NVIDIA's GPU [94]. For embedded systems not focusing only on floating-point applications, such as Kalray K1, such a hardware implementation is too expensive.

Most embedded architectures rely on generic software implementations such as the mathematical library provided with Newlib [152]. The Newlib is a set of standard C libraries implemented specifically for embedded systems. It is available as a package of generic C implementations. It has not been optimized for any specific architecture and generally shows poor performance when built on a new platform, compared to what can be obtained by a developer with a little knowledge of the architecture. It provides a good solution to rapidly support mathematical functions but lacks efficiency. MPPA's developer toolkit provides the Newlib built for the K1 core. Our intention is to replace some of the most used functions in Newlib's libm by our own optimized implementations.

This section describes the development of a floating-point mathematical function: a binary32 exponential. We develop a simple implementation which focus on low-memory dependency (no table use) and simplicity. It is build as an example to illustrate the different stages of function development.

4.3.1 Exponential evaluation scheme

Exponential is a transcendental function. As a consequence it can not be expressed exactly as a finite polynomial. In floating-point arithmetic, this translates to the following: no series of addition/subtraction/multiplication (or FMA) can compute exactly the result of an elementary function for every possible floating-point input. Thus the exponential must be implemented by **evaluation of an approximation**. Many techniques have been suggested to find the best approximation and implement its evaluation. Describing them is not the purpose of this work and we refer the interested reader to [115] or [116] for entry pointers to the extensive literature on this subject. A specific, state of the art, algorithm for the exponential is suggested in [144]. This algorithm, with a table-based argument reduction, has been implemented in [35] and [104] on Intel's Itanium architecture (IA64).

Our implementation focuses on not using memory beyond program instructions and constants stored into program memory. Table-based algorithms do not fit in those requirements. Rather, we implement a simple, computational, argument reduction described in [115] and a polynomial approximation. This implementation follows the following scheme:

1. Test for specific cases and inputs leading to foreseeable underflows or overflows.
2. Decompose the input to obtain a reduced argument $\in \left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$.
3. Evaluate a polynomial approximation of exponential at the reduced argument.
4. Build the final result from the approximation evaluation and the decomposition.

Let us describe in more details those steps. We implement the evaluation of exponential at x. We distinguish e^x which is the exact evaluation of exponential at x and $exp(x)$ which is the evaluation of our approximation at x.

We first start by reducing the input range to the valid input domain, by excluding trivial underflows and overflows. For that purpose we consider the greatest possible binary32 number: $\Omega = \texttt{0x1.fffffe}.2^{127}$ and the lowest possible normal binary32 number: $\Gamma = \texttt{0x1.0}.2^{-126}$

$$89 > log(\Omega),\ log(\Gamma) > -88$$

If x is outside $Ir = [-88, 89]$ then e^x is outside the range of representable binary32 numbers. Our approximation does not have to be valid outside this interval: $exp(x)$ will underflow or overflow outside this interval. These obvious cases of underflow/overflow will be excluded early on, before

sending valid input to the approximation evaluation. They will be managed by a specific code branch whose execution remains unlikely if the input interval is sensible.

For the following, we can assume $x \in Ir$. The next step is to reduced the input range once more approximating the function with a polynomial. We use the following decomposition of x, simplified version of the argument reduction described in [144, 115, 116]:

$$x = j \cdot log(2) + r,\ j \in \mathbb{N},\ r \in \left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right] \tag{4.16}$$

j and r are computed as follows:

$$j = \left\lfloor \frac{x}{log(2)} \right\rceil \tag{4.17}$$

$$r = x - j \cdot log(2) \tag{4.18}$$

By applying equation 4.16, the computation of e^x can be rewritten as:

$$e^x = e^{j.log(2)+r} = 2^j.e^r$$

r is the new (reduced) variable for the exponential function, $r \in i\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$. The input interval has reduced to $\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$. We can restrain the validity of our approximation of exponential to $\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$.

Let us make a quick digression and justify why argument reduction is important. The polynomial approximation degree depends, among other things, on the interval in which the approximation is valid. For example, according to Sollya, approximating $exp(x)$ on $[-88, 89]$ with an absolute error of 2^{-24} requires a polynomial with degree higher than 129. The same approximation on $[\frac{-log(2)}{2}, \frac{log(2)}{2}]$ only requires a degree 6 polynomial. The degree of the polynomial approximation directly impacts the implementation performance. Indeed higher degree implies more operations. It may degrade the latency but it always decreases the throughput. It also increases the evaluation error. Reducing the approximation interval, reduces the degree and improves performance. However reducing the interval has its own cost. It can require complex computations on the input variable and/or the use of tables stored in memory. One of the challenge of function implementation is to determine the optimal configuration of range reduction and final approximation for an implementation. We believe that this balance is tightly related to the objective of the implementation (latency, throughput, accuracy). This section focuses on a single implementation configuration but chapter 5 will suggest a solution to address the challenge of adapting function implementation to goals.

Let us go back to our implementation of exponential. Finally e^x is approximated by a polynomial p^* on $[\frac{-log(2)}{2}, \frac{log(2)}{2}]$ with ϵ_a a upper bound to its absolute approximation error. e^x can be rewritten as:

$$e^x = 2^j \cdot (p^*(r) + \epsilon_a)$$

The approximation being $exp(x) = 2^j \cdot p(r)$, where p is the evaluation of p^*. Let us notice that there is no approximation error for 2^j since the computation of j is exact. To complete this approximation we have to determine the polynomial p^*.

```
1    I = Interval(-log(2)/2, log(2)/2)
2    degree = guessdegree(exp(x), I, S2**-24)
3    p = fpminimax(exp(x), degree, [binary32] * (degree+1), I, absolute)
4    epsilon_approx = sup(abs(supnorm(p, exp(x), I, absolute, S2**-50)))
```

Listing 4.3: PythonSollya script used to determine p^* and ϵ_{approx}.

To determine such polynomial, we execute the script reproduced in Listing 4.3, using Sollya (more precisely PythonSollya described in Section 5.2, p.96). First we use Sollya's **guessdegree** to return the minimal degree of an approximation of e^x whose approximation error is lower than 2^{-24} Sollya's guessdegree estimates that a degree 6 is sufficient to ensure such approximation error. Then, the call to the **fpminimax** method returns a polynomial of degree d that best approximates the function e^x on the interval $I = \left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$, optimizing the absolute error value. Focusing on the absolute error, rather than the relative error, is justified by the fact that $exp \approx 1$ on $\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$. A function converging towards 0 (eg: $log(1+x)$ around 0) would have necessitated the use of relative error. This call returns the following polynomial:

$p = \texttt{0x5.b560fp-12} \times x^6 + \texttt{0x2.247a04p-8} \times x^5 + \texttt{0xa.aaa6fp-8} \times x^4 + \texttt{0x2.aaa918p-4} \times x^3 + \texttt{0x8.p-4} \times x^2 + x + 1$

Finally Sollya's **supnorm** gives us the approximation error:

$$|\epsilon_a| = |p - e^x| =< 2^{-28}$$

We now have the required elements to compute $exp(x)$.

4.3.2 Implementation and error analysis

Once the approximation $exp(x)$ for the exponential function has been determined, the next step is to translate it into an implementation. The implementation is the mapping between mathematical operations and operations available on the target architecture (i.e translatable into machine instructions). As stated in Section 4.1, this transformation induces an evaluation (or rounding) error.

Range reduction Let us first focus on the implementation of the argument reduction, described by Equation (4.17) and (4.18). Equation (4.17) is implemented as follows:

$$\texttt{j_eval} = int(\circ_{fp32}(x \times \texttt{invlog2}))$$

where $\texttt{invlog2} = \circ_{fp32}(\frac{1}{log(2)}) = \texttt{0x1.715476p0}$, int is the round towards the nearest integer.

We call r_p the evaluation of r, it is implemented as follows:

$$\begin{aligned} \texttt{log2h} &= \texttt{0xb.174p} - 4 && (4.19)\\ \texttt{log2l} &= -\texttt{0x1.e8082ep} - 16 && (4.20)\\ \texttt{pre_r} &= \circ(x - \texttt{j_eval} \times \texttt{log2h}) && (4.21)\\ r_p &= \circ(\texttt{pre_r} - \texttt{j_eval} \times \texttt{log2l}) && (4.22) \end{aligned}$$

To provide enough accuracy, we use Cody and Waite's method ([28, 29, 83]) to compute r_p. This method simulates a higher accuracy by partitioning the approximation of $log(2)$ into a high (`log2h`) and a low part (`log2l`), such that `log2h` + `log2l` approximates log(2) with more accuracy than any single binary32 value. Moreover, `log2h` is built such that `log2h` $\times$ `j_eval` is exact, by forcing its mantissa to contains enough trailing zeros. As $\frac{1}{2} \times x \leq \texttt{j_eval} \times \texttt{log2h} \leq 2 \times x$, by application of Sterbenz's lemma, we demonstrate that `pre_r` computation is exact, that is:

$$x - \texttt{j_eval} \times \texttt{log2h} = \circ(x - \texttt{j_eval} \times \texttt{log2h}) = \texttt{pre_r}$$

The only inexact part is the subtraction `pre_r` $-$ (`j_eval` $\times$ `log2l`). Indeed `j_eval` $\times$ `log2l` is also exact, because computed internally by an FMA.

```
1 @rnd = float< ieee_32, ne >;
2
3 # constants
4 invlog2 = 0x1.715476p0;
5 log2h = 0xb.174p-4;
6 log2l = -0x1.e8082ep-16;
7
8 #evaluation
9 j_eval = fixed<0, ne>(x * invlog2);
10 pre_r = rnd(x - log2h * j_eval);
11 rp = rnd(pre_r - log2l * j_eval);
12
13 # exact values
14 diff = (log2h + log2l) - log2;
15 invlog2_exact = 1 / log2;
16 rp_exact = (x - log2h * j_eval) - log2l * j_eval;
17 r = x - log2 * j_eval;
18
19 {x in [-88, 89] /\ diff in[-1b-42, 1b-42] -> r - rp in ? /\ j_eval in ? /\ r in ?}
20
21 x - log2 * j_eval -> x - log2 * x * invlog2_exact + log2 * (x * invlog2_exact - j_eval)
22 log2 * x * invlog2_exact -> x;
23
24 r ~ rp_exact;
25
26 r - rp_exact -> ((log2h + log2l) - log2) * j_eval;
```

Listing 4.4: Gappa script for the error evaluation of exponential argument reduction

Listing 4.4 reproduces the Gappa script used to determine the evaluation error of the argument reduction: Its execution returns the following results:

$$\texttt{j_eval} \in [-127, 128]$$

$$r_p \in [-0.346582, 0.346582] = Ir_p$$
$$r - r_p \in [-2^{-24.9986}, 2^{-24.9986}]$$

It can be noticed that r_p lies slightly outside the range $\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$. This is due to the evaluation errors. In regard to this error, we have to reconsider the polynomial p determined in Section 4.3.1. Indeed this polynomial was build as an approximation of e^x on $\left[-\frac{log(2)}{2}, \frac{log(2)}{2}\right]$. It may not expose the same approximation error on Ir_p. In practice, it is not the case: Sollya's **supnorm** still return a 2^{-28} approximation error for $p - e^x$ on Ir_p.

Polynomial evaluation We now need to determine a polynomial evaluation scheme. Such a scheme is defined as the translation of the exact polynomial into a series of machine operations and machine numbers. Sollya's **fpminimax** provides polynomial approximations whose coefficient are machine numbers: there is no need to round those coefficients when building the polynomial evaluation. Thus, we can focus on the polynomial translation into machine operations. Horner's [50] and Estrin's [54] schemes are two typical schedulings used to implement a polynomial. Those schemes have very interesting properties: Horner is optimal in the number of operations and Estrin is optimal is the length of the longest operation dependency chain. Those are only two among many possible schemes. An interesting idea, implemented by CGPE [135], is to perform an exploration of the polynomial scheme space to determine the best scheme to suit specific goals

(latency, accuracy, throughput). We intend to focus on Horner's and Estrin's schemes only. We point the interested reader towards [50, 54, 135, 115] for more in-depth studies of polynomial schemes.

Let us now us now propose two schemes, respectively based on Horner's and Estrin's scheme, to implement p. To describe these schemes, we write p as:

$$p = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

Using K1 FMA, we suggest the following implementations for p:

$$p_{horner} = \circ(c_0 + x \times \circ(c_1 + x \times \circ(c_2 + x \times \circ(c_3 + x \times \circ(c_4 + x \times \circ(c_5 + x \times c_6))))))$$

$$p_{estrin} = \circ(\circ(\circ(c_0 + x \times c_1) + \circ(x^2) \times \circ(c_2 + x \times c_3)) + \circ(\circ(x^2) \times \circ(x^2)) \times \circ(\circ(c_4 + x \times c_5) + \circ(x^2) \times c_6))$$

Let us notice that, without FMA, a rounding operator is added to each unrounded multiplication.

Description	Horner's (with FMA)	Estrin's (with FMA)	Horner's (without FMA)	Estrin's (without FMA)
Number of operations	6	8	12	15
Latency	$6.L_{FMA}$	$3L_{FMA}$	$6.L_{MUL} + 6.L_{ADD}$	$3.L_{MUL} + 3.L_{ADD}$
K1 Latency	24	12	48	24
Evaluation errors				
Polynomial	$2^{-24.5944}$	$2^{-24.2398}$	$2^{-24.0649}$	$2^{-23.7814}$
Overall	$2^{-23.5457}$	$2^{-23.3634}$	$2^{-23.259}$	$2^{-23.0959}$

Table 4.8: Performance comparison between Horner's and Estrin's evaluation scheme for the polynomial p of degree 6 (L_{FMA} is the latency of an FMA operation)

We implemented both schemes and evaluated their errors, and their performance on K1 architecture. A comparison of these evaluation results is presented in Table 4.8. Evaluation errors were determined using Gappa. The polynomial error represents the error of the polynomial evaluation alone. The overall evaluation error considers the use of these polynomial in exponential implementation: it integrates the evaluation error on the polynomial input. It has to be noticed that Estrin's scheme is twice as fast as Horner's scheme, but contains 33% more operations and is less accurate. To illustrate the benefice of the Fused Multiply-Add, we implemented both schemes with and without an FMA. The FMA reduces the latency and the number of operations, it also increases the accuracy. In practice, for K1 architecture, the FMA provided a 50% latency reduction.

4.3.3 Performance results and conclusion

Using the argument reduction and polynomial evaluations described previously, we made several implementations of the binary32 exponential. We also made an implementation based on fixed-point arithmetic. Table 4.9 presents the performance result of our implementations plus Newlib's. As expected, the Estrin-based implementation is faster than Horner-based implementation. Both schemes outperform Newlib implementation by a significant factor.

Our approach: building a specific approximation for K1, computational only and exploiting the FMA, was beneficial. But this manual implementation process has its limits. Even for a function as simple as the exponential, It is a time-consuming and error-prone process. It requires

Description	latency (cycles)	speedup (wrt Newlib)
K1 expf (Estrin-based)	45	4.04
K1 expf (Horner-based)	53	3.43
K1 fixed expf (Horner-based)	62	2.93
Newlib's expf	182	1.00

Table 4.9: Performance results for binary32 exponential implementations

an extended knowledge of the architecture. Several implementations scheme, with a multitude of parameters, should be evaluated (polynomial degree, polynomial scheme, range reduction algorithm). The next chapter suggests a possible solution to help increase the function developer productivity in this domain.

Chapter 5

Automated code generation of mathematical functions

Division, square root and exponential implementation, studied in Chapter 4, were developed manually in non-portable C (containing K1-specific intrinsics). This process is very time consuming and error prone. Division and square root, are frequently used by a large range of applications, which justifies manual optimization and long development times.

The number of mathematical functions to support makes this process difficult to generalize. Moreover, the recommendation of the IEEE standard: a single, correctly-rounded, implementation per function, is not followed in practice. There exist very different application requirements (eg: accuracy, latency, throughput). A few applications effectively rely on correctly-rounded implementation, but most would rather trade lower accuracy for better performance. Several response to this variety of requirements have been studied. [48] suggests to provide at least 3 accuracy levels for each function and each precision. Intel's MKL [75] offers 3 accuracy levels (although not compliant with [48]) OpenCL [143] introduced some relaxed requirements (2 ulps accuracy). The paradigm of the standard mathematical libraries, one single implementation for each precision and each function, appears deprecated. Small companies, such as Kalray, do not have the time or the resources to extend the manual development process to the variety of implementations required for mathematical functions.

[43] stands that FPGAs are better at floating-point than CPU. Its argument is that FPGA implementation can be optimized to suit the various application requirements, beating CPU static floating-point pipeline peak performance with flexibility. This analysis lead to the development of FloPoCo [46], a framework for the design of arithmetic operators on FPGAs. FloPoCo philosophy can be summarized as follows: assisting the developer designing an arithmetic operator, while optimizing the use of available resources to implement the most efficient operator for a specific FPGA and specific application requirements.

The project described in this chapter intends to apply this approach to the software implementation of mathematical functions on generic processors. Several other projects follow the same goal. There are commonly named **metalibms**. The term should belong to C. Lauter who started the earliest project [86]. But we feel it is the best suited phrase to grasp the concept behind all the different projects including our own. That is why we are going to use the phrase *our Metalibm* to describe our project.

Our contribution comes as a continuation of two previous PhD theses [90] and [25] which shared the same project: the automation of elementary function code generation. Our Metalibm uses extensively the results and software tools developed during those PhDs. To develop the first application of our project, described in Section 5.8, we used results from [47, 90].

The first objective of this work is to automate the development of optimized replacement for

the mathematical library functions (eg: functions of Newlib's libm for the K1). Its second objective is to automatically provide implementations better suited to application requirements. To implement the first objective we share [46]'s philosophy: the developer knows best. The first version of our Metalibm is intended as a framework to help an experienced developer implementing mathematical functions with architecture specific support. Eventually, to meet the second objective, you intend to transform into a tool for non-experienced user interested in certified and efficient code generation.

5.1 Introduction

Before introducing our approach, let us describe some examples of state of the art mathematical function generators and introduce the challenges related to our problem: the automation of elementary function implementation.

5.1.1 State of the art

Our problem has already been addressed by several works. Christoph Lauter [90] and Sylvain Chevillard [25] both devoted part of their PhD to this problem. They put their efforts in common to develop Sollya ([27]). Sollya was introduced in Section 4.1, p.74. Sollya addresses a first important challenge of our problem: computing function approximations and approximation errors. Many mathematical function developers use general computer algebra systems such as Mathematica [155] or Maple [103] to build their approximations. Sollya is a state of the art alternative to those algebra systems. It provides some original features such as floating-point oriented function approximations, and as stated in Section 4.1, p.74, Sollya's **supnorm** function is a state of the art solution to compute approximation errors.

Efficiently implementing a function is not enough. An implementation should be delivered with a verifiable certificate which demonstrates its error and correctness. Gappa [38], introduced in Section 4.1, p.75, addresses this second challenge: automating the implementation certification.

Apart from Sollya, C. Lauter also started the development of his own Metalibm, during his PhD thesis [86]. The project is still under development. C. Lauter's Metalibm (CLM) implements an original approach which does not share FloPoCo's philosophy. It does not rely on the developer knowledge to guide the implementation generation, rather it applies a systematic and generic process. Implementations are generated from the following inputs:

- a function expressed using Sollya primitives
- an input interval
- some approximations constraints

The generation process proceeds as follows:

1. It infers some numerical properties (e.g. parity: is it an even or an odd function ?), directly from the function expression, to obtain a simplified description.
2. On this simplified description, it systematically applies a catalog of possible argument reduction and polynomial approximation techniques.
3. By using performance profiling, it selects the best approximations and generates a final implementation.
4. It generates a Gappa script to certify the final implementation error.

The function is manipulated as a black box. For example, when generating e^x implementation, the tool does not try to apply exponential-specific algorithms, such as Tang's table-based method [144]. This approach is very efficient to rapidly generate implementations of non standard functions. However it does not optimize the implementation for a specific architecture, nor does it provide means to the developer to use its knowledge of the function properties.

5.1.2 Objectives and philosophy of our project

Contrary to CLM [86], which provides a generic tool, we intend to develop a framework to help the developer. Our objectives differ: C. Lauter's Metalibm develops a generic and systematic flow to generate implementations. Our approach is to use the developer knowledge to define an implementation specific to the function and optimized for a specific architecture. The two approaches can be merged. It is the purpose of the ANR project Metalibm [41], which started during the redaction of this manuscript.

Our work is designed as an extra layer built on top of Sollya and Gappa. It intends to provide an automated framework to help the libm developer in his implementation of mathematical function evaluations. In this work, our intention is not to provide newer implementations but to provide means to easily customize and optimize known implementations for different architectures and contexts using an automated framework: our **Metalibm**. Let us first introduce the goal of our Metalibm project:

- provides way to automate code generation from function-specific implementation descriptions
- disconnect implementation description from optimizations and code/proof generation
- provides ways to the developer to transmit his knowledge to Metalibm core engine

To fulfill those, our project is designed to provide several key features:

- a tool that inputs a unified implementation description annotated by the developer
- a tool to automate the optimization of an intermediate representation built from the description
- a code generator to take care of the troublesome step of generating source code
- a proof generator that relies on proof-checker like Gappa to demonstrate the correctness of the implementation
- an independent target description that is accepted by both the optimization pass on the intermediate representation and code generator

Our Metalibm is a framework written in Python which provides implementation description tools, IR optimization passes, support libraries and code generators. It is still in a very early stage of development. The remaining of this chapter is divided as follows: Section 5.2 describes the arithmetic support system developed for our Metalibm. Section 5.3 introduces Metalibm's Description Language (MDL), an annotated language used to describe an implementation. This description is the input processed by the core of our Metalibm to generate its intermediate representation (IR). Several optimizations can be performed on this IR. Finally it is translated into source codes and certificates. Section 5.4 describes Metalibm's support libraries which provide builtin support for utility functions and extended precision arithmetic. Section 5.5 presents Metalibm's core: type determination pass and optimization passes. Section 5.6 describes the output

generation part of Metalibm: code generation and proof generation. It also presents how Metalibm performs target specific code generation. Finally Section 5.8 present one of the first use case of our Metalibm: the development and evaluation of vectorized correctly rounded binary64 exponential and logarithm functions.

5.2 Arithmetic support: PythonSollya

As previously stated, our framework has been developed using the well-known Python programming language ([57]). Python has been chosen for its portability, easiness of use and easiness of learning. It offers an easy interface to more efficient low-level C-written primitives when required. However Python has a big drawback: its arithmetic support.

Function implementations (eg: during argument reduction) require constants with a larger precision than the machine largest format to ensure final implementation accuracy. Our Metalibm requires a arbitrary-precision floating-point arithmetic to provide this level of accuracy. Python does not provides such support. It provides arbitrary-precision integer arithmetic but limited precision floating-point capabilities. The Python interpreter provides the largest format supported by the architecture, typically binary64. For mathematical function implementations, Python provides the **math** module [58] which depends on the interpreter implementation. For the most common interpreter, CPython, this implementation is, according to Python's documentation, a "thin wrappers around the platform C math library functions". Therefore, it inherits the accuracy of the platform C math library (faithful in most cases at best).

To address this problem we introduce PythonSollya: a binding between Sollya's library and Python. Sollya's library provides an API containing Sollya's core functions which bypasses Sollya script language interpreter. This binding allows a developer to use Sollya as the arithmetic system of Python: every number and arithmetic expressions are interpreted by Sollya library. Sollya library provides support for both arbitrary-precision floating-point arithmetic through MPFR, and interval arithmetic through MPFI. This support covers both the standard arithmetic and the common mathematical functions (eg: exp, log, trigonometric function). By using PythonSollya module, the Python developer is able to compute arbitrary-precision evaluations of those functions. The developer has the possibility to mix Python object and Sollya object in an arithmetic expression. The result is always evaluated through Sollya's library and expressed as a Sollya number. Sollya takes care of the rounding and manages the intermediary precision. One of Sollya greatest feature: flagging when an evaluation induces a rounding error, can be enabled in PythonSollya. PythonSollya binds most of Sollya methods for function implementation (eg: fpminimax, supnorm, guessdegree). It allows the generation and manipulation of polynomial approximations directly into the Python language. Every Sollya object is embedded into a Python object which is an instance of the class **SollyaObject**.To extend PythonSollya support to not bound function, **SollyaObject** constructor can also be used to execute Sollya script commands.

```
1  >>> from pythonsollya import *
2  >>> round(log(S2), 100, RN)
3  0.693147180559945309417232121457981863566262059365 09
4  >>> display(dyadic)
5  >>> fpminimax(exp(x), 5, [binary32]*6, Interval(0, 1), absolute)
6  1866159b-27 * _x_^5 + 9341529b-28 * _x_^4 + 1429441b-23 * _x_^3
7  + 4186719b-23 * _x_^2 + 8389275b-23 * _x_ + 16777197b-24
8  >>> p = fpminimax(exp(x), 5, [binary32]*6, Interval(0, 1), absolute)
9  >>> coeff(p, 3)
10 1429441b-23
11 >>> supnorm(p, exp(x), Interval(0, 1), absolute, 2**-50)
```

```
12    [700976274800962961407b-89;2525533191991342331540082417815938659 3b-144]
```

Listing 5.1: Some examples of PythonSollya calls into the Python interpreter

Listing 5.1 provides some examples of use of the module PythonSollya. It presents a Python interpreter log. Command syntax should be very familiar to Sollya's script developers, with some differences (eg: display is now a function rather than a global variable).

Let us summarize the contribution of PythonSollya module to Python:

- support for arbitrary-precision floating-point (through MPFR)
- support for arbitrary-precision interval arithmetic (through MPFI)
- arbitrary precision correct rounding of common elementary functions (through MPFR)
- constrained polynomial approximation
- accurate approximation error bounding

5.3 Description language

One of the objective of our Metalibm is to simplify the function developer work. The function developer should not have to chose between output formats (eg: C, assembly, Fortan, Compiler IR). A Metalibm should provide tools independently of the final target. Implementation performance should not rely on low-level description. Our objective is to allow the function developer to focus on the implementation algorithm. We suggest to disconnect the implementation description from the generation of its implementation. This chapter introduces the implementation description solution of our Metalibm: the Metalibm Description Language (MDL). The MDL is, a Python-Based language, used by the function developer to describe a parametric function implementation.

MDL is built around a set of Python object constructors. The function developer uses those constructors to describe the implementation operation graph. MDL provides basic control flow constructors with limited capabilities (eg: no loop description). MDL's constructors can be divided in four categories:

- **Leaf nodes**: such as VARIABLE, Constant and Table, which are the operation graph entry points.
- **Arithmetic operators**: such as Addition and Multiplication, which are used to describe arithmetic expressions.
- **Control flow graph operators**: such as ConditionBlock, which are used to describe logical links and dependencies between basic blocks.
- **Return operator**: Return, which is used to declare which node(s) constitutes the final result(s) and function output(s) (operation graph outputs).

```
1   vx = VARIABLE("x", precision = ML_Binary64)
2   c1 = Constant(1 / log(2), precision = ML_Binary64)
3   c2 = Constant(log(2), precision = ML_Binary64)
4   n = NearestInt(vx * c1)
5   r = vx - n * c2
6   s = 1 + r * (1 + 0.5 * r)
7   result = Return(s)
```

Listing 5.2: Example of function implementation using Metalibm's description language

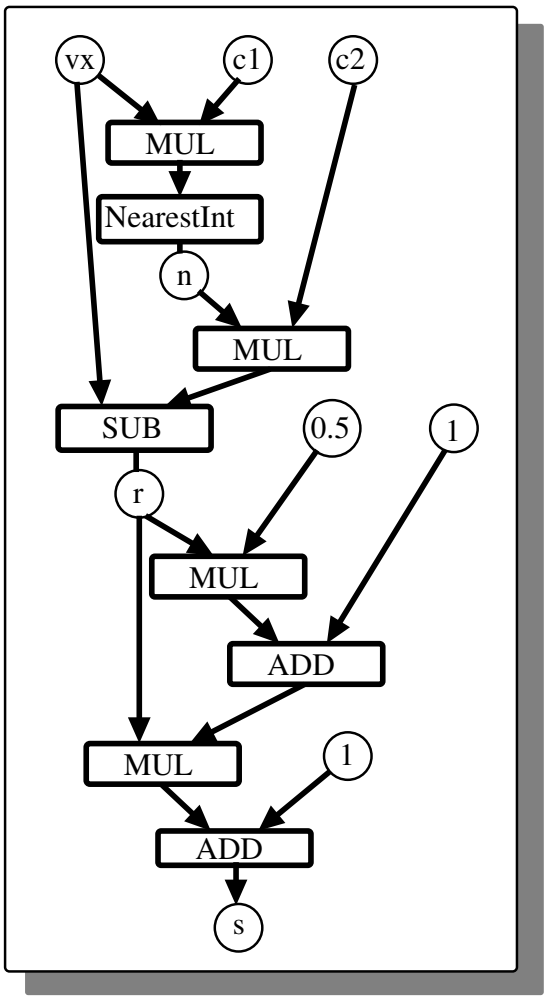


Figure 5.1: Operation graph of MDL description reproduced in Listing 5.2

Let us illustrate MDL with an example. Listing 5.2 presents a MDL implementation description. Figure 5.1 shows the corresponding operation graph. The description starts, **line 1. to line 3.**, by declaring the VARIABLE node corresponding to the function input and two constants. Those leaf nodes are assembled together into an arithmetic expression using both explicit operation constructor **line 4.** (NearestInt) and implicit arithmetic **line 5. and 6.**: $+, -, *$. Finally the implementation result is encapsulated into a Return operator **line 7.**.

Our Metalibm provides some tools to generate parts of this description. Metalibm's generation of polynomial evaluation will be described in more details in Section 5.3.3. In the current MDL version, description parametrization is limited to:

- undetermined node precision
- the use of abstract operation

It is our intention to expand that parametrization in the future. A MDL description directly builds an intermediate representation (IR). The IR corresponds to the operation graph inferred from MDL description. Before considering this IR, we are going to describe in details some of the features of MDL which impact IR manipulation.

5.3.1 Annotation system

An experienced function developer knows more about the numerical properties and specificities of the implementation than what the early version of our Metalibm could infer. To benefit from this knowledge, MDL gives the developer the means to express it in the implementation description. This mean is MDL's annotation system.

Annotations can be used by the developer to force specific operation behaviours or as tools during development and debug. Each annotation is associated to a node. MDL provides a wide range of annotations, which can be sorted in four categories:

- Application development and debug: annotations used by the developer to facilitate the application debug (eg: dynamic value display, source code variable naming).
- Numerical properties: annotations used to express some numerical or arithmetic properties (eg: exact operation, variable's value interval).
- Control flow hints: annotations used to give hints about execution behavior (eg: most likely scenario in a if-then-else branch).
- Code generation hints: annotations used to express code generation properties (eg: should the node be generated as an external function).

An annotation already appeared in Listing 5.2: the **precision** annotation. This annotation is part of the numerical property category. Let us now describe in more details the annotation categories.

Application development and debug

This set of annotations contains two principal annotations: **tag** and **debug**. The **tag** annotation constitutes a name indication to Metalibm core engine. When generating source code Metalibm uses this tag as a prefix for the name of the variable which contains the node result. It is very useful annotation to maintain readable source code. Metalibm engine enforces tag uniqueness: if two nodes have the same tag Metalibm modifies one of them to ensure that the two variables do not conflict. However for very specific cases, a tag can be more than an indication and can be a strong constraint that the developer want the engine to respect (2 nodes describing the same underlying variable for example). To that purpose the annotation **force_tag** can be used, it bypasses the uniqueness check.

If the **debug** annotation is associated with a node, then Metalibm code generator will integrate a debug message in the source code to display the node value for each execution. It allows the developer to check dynamically the node behaviour.

Numerical properties

Metalibm core relies on knowing node's numerical properties to optimize the operation graph. Those properties can come from two sources:

- determined by inference during Metalibm core processing
- indicated by the developer through a numerical property annotation

MDL provides the following numerical properties annotation:

- precision

- interval
- exact

The **precision** annotation may be one of the most important of the Metalibm system. It indicates the node's format. This annotation is restrictive: Metalibm core introduces the necessary conversions in the operation graph so that the node format matches its precision annotation. Function developers can use it to constrain a specific family of format at certain positions in the dataflow graph.

The following formats can be used as argument to the precision annotation:

- floating-point: **ML_Binary32, ML_Binary64, ML_Binary80**
- fixed-point: **uint32, int32, uint64 and int64**

Metalibm also provides support for arbitrary fixed-point format through the **FixedPoint** constructors. It also provides extended precision support through multi-binary64 formats (double double and triple double). This feature is discussed in details in Section 5.4. All these formats are part of the Metalibm core engine. They are used, outside the annotation system, by the IR. Before code generation, Metalibm engine associates a format to every node of the operation graph, following the precision annotation when defined.

The **interval** annotation is used to indicate the node value known bounds. By default, Metalibm automatically determines the interval of a Constant node as equal to an interval of radius 0 centered on the constant value. Moreover when manipulating arithmetic operations, if both operands have a known interval, Metalibm determines automatically the result interval by applying the arithmetic operation to the operand intervals.

The **exact** annotation indicates that no rounding error are expected for the annotated operation node. For example, it applies in Cody's and Waite's argument reduction, when multiplying floating-point numbers with enough trailing zeros, or under Sterbenz's lemma condition to the addition/subtraction of close-enough numbers. This indication is used by the Metalibm backend which generates error certification.

Control flow hints

MDL provides control flow description node. Those nodes are used to organize the evaluation flow: partitioning it in conditional branches. These control flow nodes can be annotated by the **likely** annotation. This annotation has several possible values:

- non defined: no information is available
- True: the condition directing the control flow is true for a majority of inputs
- False: the condition directing the control flow is false for a majority of inputs
- Likely_Possible: the condition directing the control flow is as likely to be true as to be false.

When no information is available Metalibm can not infer any likely behavior and generates generic branch code. If **likely** is defined with a Boolean value, Metalibm can extract the control flow most likely behavior. It can transmit this piece of information to the compiler (eg: through GCC's built-in __builtin_expect). Consecutively, the compiler may be able to perform specific code optimizations, based on this knowledge.

If **likely** is set to Likely_Possible, it also constitutes a valuable piece of information. Metalibm can assume that none of the branch directions can be tagged as most likely. If enough instruction

level parallelism is available it can generates both branches for parallel evaluation, and finally insert a select operation. This behaviour alleviates branch-prediction error and can be very efficient on predicated instruction set or when vectorizing functions. Indeed it flattens the control flow and avoid divergent branches that do not fit into vector instruction.

Code generation hints

Some operators are provided in several flavours. For example extended precision addition (double double or triple double precision) is available in both **fast** and **slow** variants, Those variants can be chosen according to some numerical properties of the operand nodes. For example when adding two binary64 numbers into a double double precision number, a faster algorithm, fast2sum, can be used rather than the slower method, 2sum, if the operand order is known during code generation. If the developer is sure that this property is true in his implementation, he can use the **fast** annotation. If a fast variant is indeed available for the node operation, Metalibm uses that variant during code generation.

Conclusion on the annotation system

The annotation system allows the developer to transmit a lot of information to the Metalibm engine. Some of them are indications or hints and others are restrictive constraints. They give the developer the capability to integrate his knowledge to the implementation description.

We are now going to describes tools provided by our Metalibm to build a function implementation. More specifically, these tools are used to build part of the MDL description.

5.3.2 Constants, tables

Constants and tables are a important part of a function implementation. The MDL offers various ways to express constant values. As metalibm is based on Python script, the developer can use Python number system. As stated in Section 5.2, this system lacks many of the key elements required for function development, that is why we introduced PythonSollya. PythonSollya can be used to accurately generate constants and tables required by the elementary function implementation.

Constant generation

```
1    # importing the PythonSollya wrapper
2    from pythonsollya import *
3
4    # MSB of log(2) with 43-bit precision (binary64 with 10 trailing zeros),
5    # rounding up
6    log2_hi = round(log(2), 43, RU)
7    # LSB of log(2) in binary64 precision, rounding to the nearest
8    log2_lo = round(log(2) - log2_hi, binary64, RN)
```

Listing 5.3: Generation of a bi-partite approximation to log(2) with PythonSollya, for use in exponential argument reduction

Listing 5.3 illustrates an example PythonSollya use, the generation of an approximation of log(2) with 96 bits of accuracy.

Table generation

Table is an extension to constant: a table is built from several constants stored in an ordered layout. Metalibm offers a helper for table generation: the **Table** object. Tables can later on be

addressed by two of MDL operations: **TableLoad** or **TableLoad_HL**. The first operation loads a single value, while the second loads two consecutive aligned values (useful to make sure that loads are vectorized according to the developer wishes).

```
1     # Table object initialization
2     new_table = Table(storage_format = ML_Binary64, dimensions = [128, 2])
3
4     # Table value initialization
5     for i in xrange(128):
6         index_value = SollyaObject(i)/128
7
8         # computing table content values
9         value_hi = round(log(index_value), binary64, RN)
10        value_lo = round(log(index_value) - value_hi, binary64, RN)
11
12        # storing table content values
13        new_table[i][0] = value_hi
14        new_table[i][1] = value_lo
15
16    vi = VARIABLE("i", precision = int32)
17    # index building
18    v_index = vi & 127
19    v_index_hi = v_index << 1
20    v_index_lo = v_index_hi + 1
21
22    # reading value in the table
23    value_hi = TableLoad(new_table, v_index_hi)
24    value_lo = TableLoad(new_table, v_index_lo)
25
26   # returning result
27    result = ReturnValue(value_hi + value_lo)
```

Listing 5.4: Example of Table creation, initialization and use

Listing 5.4 shows an example of **Table** object creation, value initializations and table use inside an implementation description. Multi-dimension tables can be built as multidimensional arrays by defining multiple numbers in the **dimensions** list parameter of the **Table** constructor. Multi-dimension tables are linearized by metalibm during code generation. Thus, they are accessed through a single linear index in **TableLoad** operations. This is the reason why, in Listing 5.4, the variable **v_index** is multiplied by 2 to get the high value index before being incremented to get the low value index.

5.3.3 Polynomials

When considering the automation of mathematical function implementation, polynomial approximation techniques are particularly interesting. First of all, because a polynomial is easily implemented: it is easily mapped into basic machine operations (addition, multiplication, FMA). Then thanks to Remes' algorithm [134] and its derivatives [19, 20], computing a polynomial approximation for a specific function is a very systematic process. Sollya implements a state-of-the-art version of this algorithm, specifically designed to generate machine coefficients. Both part of the implementation error of a polynomial can be determined automatically. Sollya [81] determines the approximation error, and Gappa [38] the evaluation error. Such a process fits nicely into the automation of implementation generation.

Our metalibm uses Sollya's capabilities, through PythonSollya, for the generation of the polynomial approximation and the evaluation of approximation errors. Metalibm's polynomial gen-

eration is provided by the **PolynomialGenerator**, which relies on Sollya's **fpminimax** function. Once generated, the approximation is encapsulated into Metalibm's **Polynomial** object which can be used for additional processing. For example, its approximation error can be computed through the Sollya's **supnorm** function. A sub-polynomial can also be extracted if the developer wishes to build a custom evaluation scheme without using the tool described next. To implement the translation between polynomial and evaluation scheme, Metalibm provides the **PolynomialGenerator** module. From a **Polynomial** object, this module can generate several operation graph corresponding to as many evaluation schemes. In the current version, two scheme models are available: Horner and Estrin.

```
1   @rnd = fixed<-30, dn>;
2   c_1 = 1073741865b-30;
3   c_2 = 134217731b-28;
4   c_3 = 44738565b-28;
5   c_4 = 1398085b-25;
6   c_5 = 1124101b-27;
7   c_6 = 1498023b-30;
8
9   # polynomial exact expression
10  pol_exact = (1 + (x * (c_1 + (x * (c_2 + (x * (c_3 + (x * \
11  (c_4 + (x * (c_5 + (x * c_6))))))))))));
12
13
14  # polynomial rounded evaluation scheme
15  pol_eval = rnd(1 + rnd(x * rnd(c_1 + rnd(x * rnd(c_2 + \
16  rnd(x * rnd(c_3 + rnd(x * rnd(c_4 + rnd(x * rnd(c_5 + \
17  rnd(x * c_5))))))))))));
18
19
20  # diff declaration
21  diff = pol_exact - pol_eval;
22
23
24  # statement to prove: polynomial eval. error
25  { x in [-0.346574, 0.346574] -> diff in ? }
```

Listing 5.5: Example of Gappa script generated by metalibm for the computation of the evaluation error of a polynomial in fixed-point

The PolynomialGenerator also features evaluation error determination. To compute the evaluation error, the module generates a Gappa script. The script goal is to determine the interval in which the difference between the exact and the evaluated polynomial lies. The module executes the script with Gappa and extract Gappa's response. Listing 5.5 provides an example of Gappa script generated by Metalibm for the determination of polynomial evaluation error. The exact polynomial is described line 10 and 11. The evaluation scheme is described from line 15 to line 17. To avoid infinite loops, Gappa engine does not use commutative property when rewriting an expression, to affine interval computation. To facilitate Gappa process, and obtain the finest possible response, it is crucial to express the exact scheme and the evaluation with the same operand order and parenthesis. Ideally the evaluation should be a copy of the exact scheme, extended with rounding operators. Metalibm Gappa script generators enforces that rule. The goal is defined line 21: Gappa is asked to determine the difference between the evaluation and the exact object assuming a specific interval for the input variable.

Function	required accuracy (bits)	
	round to nearest	directed rounding
exp	113	159
ln	115	119
sin	114	127
cos	113	143

Table 5.1: Accuracy requirement for correctly-rounded implementation

5.3.4 Abstract type

One of the parametrization possibilities, offered by MDL, is to use abstract types when annotating node with a precision. This gives some freedom to Metalibm engine when optimizing the IR precisions. In the current version of MDL, this feature is limited to integer types. Rather than using a **int32** or **int64** precision tag, the developer can specify the **ML_Integer** precision value. Metalibm core tries to infer the interval of the node operands to determine which format fits best the operation result. It also tries to limit the number of conversions between 32-bit and 64-bit values, favouring the use of a single integer format among dependant operations. This process has been implemented because, on most processors, casting a value between two integer formats requires an instruction. This instruction occupies an execution unit and impacts implementation latency. In a future release, this process will become configurable and adapted to the target architecture.

5.4 Metalibm support libraries: miscellaneous and multi-precision

Our metalibm framework provides a set of support libraries. They implement some useful primitives for function implementation: type conversions, arithmetic without hardware support (eg: fixed-point emulation, extended floating-point arithmetic).

They are written in portable C code and are not part of metalibm core written in Python. A call to the support library is translated (during code generation) into a call to a C function. The extended floating-point arithmetic library also exists as embedded into metalibm core, it is dynamically generated during code source generation. If the embedded library is used, a call to the support library is translated into inlined code in the source output. The first choice lets the inlining to the compiler discretion while the dynamic generation forces it.

One of the first application cases, of our Metalibm, was the development of vectorized correctly rounded exponential and logarithm functions in binary64 precision. Implementation details and results are described in Section 5.8. We are going to describe a part of metalibm support libraries developed for this application case. Describing all the complexity of correctly rounded implementation is out of the scope of this manuscript. For the reader, unfamiliar with the subject, we added Appendix 13, p.199. It introduces the basic concepts required to grasp the difficulty of correctly rounded implementation. For the benefit of clarity, let us simply state that we use the method described in [90] to implement correctly-rounded functions. This method implements two steps:

- a fast step, implementing the evaluation of the function approximately to machine-precision accuracy
- an accurate step, implementing the evaluation of the function to extended accuracy

Table 5.1 lists the required accuracies of the accurate step for some of the most common functions ([116]).

These accuracy requirement exceed the capability of binary64 precision (53 bits of accuracy). Binary64 is the largest precision with hardware support available on most CPUs. Thus we need to provide a support for higher precisions.

A solution could be to use an arbitrary precision library such as [110]. The problem of such solution is the performance impact: MPFR uses dynamic memory allocation and management. Our application case exposes very limited requirement: a fixed precision format of 160 bits is enough to manage all cases listed in Table 5.1. Memory management latency largely overcomes arithmetic latency, making this solution ill-suited for our case.

A second solution to this problem was suggested in [42]. The solution is to use the extended double precision available in x87 instruction-set to provide some of the extra bits required by correct rounding. This precision uses 80-bit registers and offers a 64-bit mantissa. This accuracy is not enough to fit our requirements. Moreover the extended precision is only available in x87 and Itanium architectures. The newest extension to x86 ISA, including the vector oriented SSE, AVX, AVX2, do not implement support for the 80-bit format. Thus this approach can not be used inside vectorized implementations relying on vector instructions.

An other solution was provided later in [89]. This solution is to use multi-binary64 formats: double double and triple double. Under specific conditions, those formats offer increased accuracy over binary64, around 100-bit for double double and 150-bit for triple double. [89] describes the basic blocks required to implement double double and triple double operations. All those operations can be implemented using binary64 operations. Thus it can be ported to every architecture which supports binary64 operations. It scales to vector instruction set extensions implementing binary64 operations (x86's SSE, AVX, AVX2, but not ARM's NEON).

The extended precision support library implements the algorithms described in [89]. Our own implementation is just a rewriting of part of CRlibm's code [3]. It has been extended with an implementation of those algorithms in Metalibm's IR format for dynamic generation (inlining). This support library provides most combinations of binary64, double double and triple double basic operations (multiplication, addition, subtraction). It brings support for two extra formats in our Metalibm:

- ML_DoubleDouble
- ML_TripleDouble

Those formats can be used as argument to the **precision** annotation as any generic format available in Metalibm. The selection between the two versions of the library is done by configuring a Metalibm's IR optimization pass. Thanks to this support, the developer can use double double or triple double arithmetic in his implementation description. Metalibm target specific code generation, described in Section 5.6.2, performs some optimizations on the multi-precision implementation. For example it merge some multiply and add operations, used in the double double multiplication, into FMAs, if this operator is provided by the targeted architecture. (eg: using an FMA when available for extended precision multiplication ...).

5.5 Metalibm core: internal representation and optimization

The global scheme of metalibm process is illustrated by Figure 5.2. First, MDL implementation description, extended with annotations, is translated into an internal representation (IR). The IR is the common representation, used by the optimization passes and input by code generators. Then, Metalibm's core applies several operation graph optimization passes to the IR. Finally the IR is

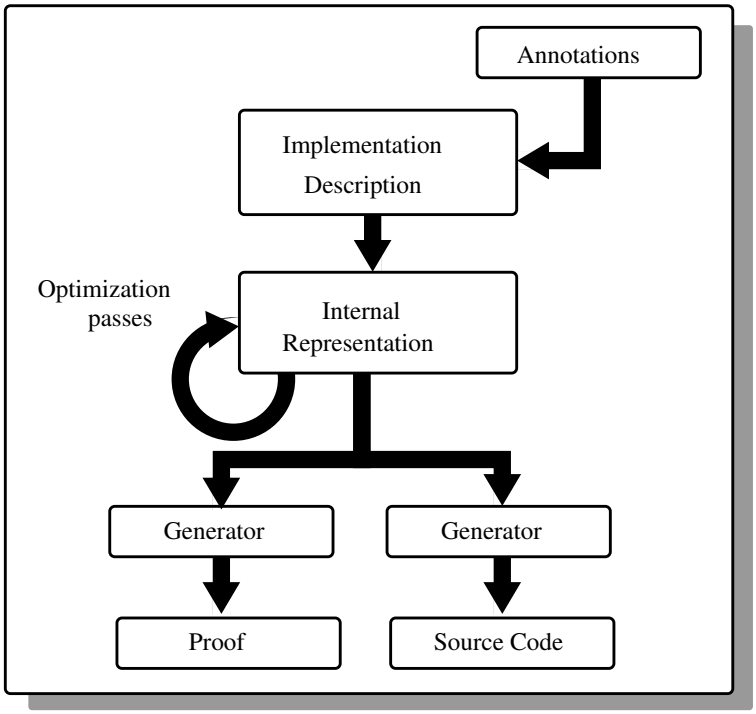


Figure 5.2: Metalibm's process overview

transmitted to generators which performs source code or evaluation error proof generations. In our Metalibm, the operation graph optimization are separated from the code-level optimization. The operation graph optimizations directly manipulate the intermediate representation (IR). Code optimizations are performed by the code generators, during the translation of the IR into source code. Let us now study in more details, some of the processes performed by Metalibm core. The first process, described in Section 5.5.1 is mandatory, it must be performed before code and proof generation. The three others: operation-flavour selection, interval determination and scalar vectorizer, are optional. They are intended as illustration of Metalibm core capability.

5.5.1 Format determination

```
1         def generate_source_code(eval_format):
2                 # eval_format is a parameter to the description (ML_Binary32|ML_Binary64)
3                 ri = Interval(-log(S2)/2, log(S2)/2)
4                 vx = VARIABLE("x", precision = eval_format)
5
6                 # backend object determination
7                 backend = Backend()
8
9                 k = NearestInt(vx * (1 / log(2)), precision = ML_Integer)
10                dag = (vx - log2 * k)
11
12                # use <eval_format> to arbitrarily instantiate every
13                # undetermined format
14                # backend.backend_process performs both the format determination
15                # and the operation graph optimizations
16                instantiated_dag = backend.backend_process(dag, eval_format)
```

```
17
18                  # generating C source code
19                  source_code = C_CodeGenerator().generate_expr(instantiated_dag)
20                  return source_code
```

Listing 5.6: Using backend pass to instantiated undetermined precision

The first process is the format determination. The current version of MDL authorizes description with undetermined precisions. The first manipulation performed, on the IR, by metalibm core is to instantiate every undetermined precision to a static computation precision given as argument to the core call. This behaviour can appear somewhat limited since a single precision replaces every undefined precision. However it gives the capability of describing a format agnostic implementation description. Listing 5.6 gives an example of use of this feature. By modifying `eval_format`, the description can be instantiated to binary32 or binary64 precision. Once every node's format has been determined, Metalibm can perform some higher level optimizations.

5.5.2 Operand-flavour selection and interval determination

The second process, performed by Metalibm core, is operand-flavour selection. Operand-flavour selection is a process which annotates the IR with the **fast** annotation when it applies. Let us take the example of the addition of two binary64 nodes into a double double result. If one operand is greater than the second, then the fast2sum algorithm can be used to implement the operation, rather than the slower, and more generic, 2sum algorithm. To determine operand order, Metalibm core compares their intervals. If they are determined, and do not intersect, then a static order exists between these operands. Metalibm core annotates the operation as **fast**. Using this annotation, code generation generates a fast variant of the addition (an implementation of fast2sum). Operand-flavour selection relies on an other of Metalibm processes: interval determination. Metalibm core propagates known intervals (eg: variable's interval, constant node values) to arithmetic operation nodes and try to determine the node values interval.

5.5.3 Scalar vectorizer

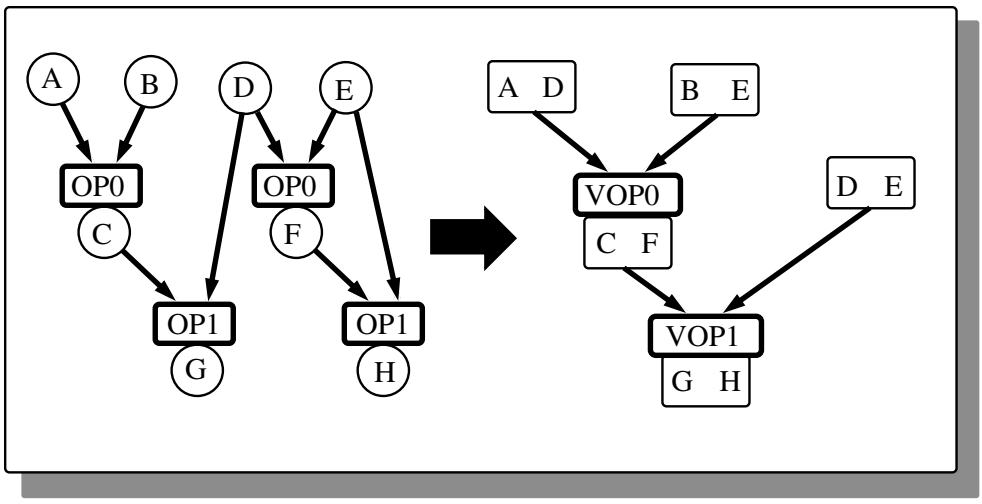


Figure 5.3: Example of Super Word Level Parallelism extraction

The scalar vectorizer was designed as a try-out for future vectorization efforts. This module implements the algorithm described in [95]. This algorithm exposes Super Word Level Parallelism

(SWLP) in scalar code. SWLP is the parallelism exploited by single instruction multiple data (SIMD) operations. The corresponding operands of two isomorphic and independent operations can be merged into superwords. If the target architecture implements a vector unit, then those superword can then be mapped into a vector register, and the independent isomorphic operations are performed by the vector unit. This improves implementation efficiency, by exploiting vector registers and unit to parallelize some operations. Figure 5.3 illustrates an example of superword parallelism extraction. Unitary data chunks are merged in superword (eg: $A + D \rightarrow AD$ and $B + E \rightarrow BE$) so that operations can be vectorized A **(OP0)** $B = C + B$ **(OP0)** $E = F \rightarrow AD$ **(VOP0)** $BE = CF$). [95]'s algorithm starts by listing all the possible superword and then selecting the most reused ones. This heuristic intends to alleviate the cost of forming and breaking vector registers from scalar registers, by ensuring that the elected superwords are the most reused. Metalibm's module applies the algorithm to the intermediate representation and exhibits possible candidates for vectorization. In its current version, Metalibm core discards this information. We are currently modifying Metalibm core to add vector operations to the IR. This modification will allow to use the scalar vectorizer.

5.6 Code and proof generation

The IR is used as common source for both the code and the certificate (or proof) generation. The fact that both generations share the same input can help ensuring the certificate validity, but is not enough. The certificate is only valid if it applies to the final implementation. Both code and proof generation have to generate two versions of the implementation with identical numerical properties. This constraint has to be enforced when developing metalibm code and proof generation. Section 5.6.1 describes the proof generator. Section 5.6.2 describes the source code generation process.

5.6.1 Gappa script generation

One of Metalibm code generator is in fact a certificate generator. It has been distinguished on Figure 5.2, under the name proof generation. This proof generator interfaces Metalibm with Gappa, described in Section 4.1, p.75. It generates Gappa scripts, used to determine the implementation evaluation error. From the IR, it generates two versions of the intermediate representation:

- an exact expression, which does not contain any rounding operator and corresponds to the exact mathematical expressions.
- an implementation expression, which contains rounding operators to encapsulate operators provided by the target architecture. It corresponds to the function as evaluated by the implementation.

As stated previously it is very important that both the Gappa script generation and the code generation agree on the intermediary representation semantic. For example, the rounding operators introduced in the implementation expression must correspond one by one to the effectively rounded operations generated by the code generation. When this assumption is verified, the implementation evaluation error is computed by subtracting the exact expression from the evaluation expression. For simple expressions, like polynomial schemes, Gappa can generally determine an acceptable bounds on its own. Gappa's engine can be helped by making sure that suitable parentheses are located on corresponding position on both expression versions. However for more complicated formulas, it can be hard for Gappa to find the best bound without help. To solve such problems, Gappa relies on hints. Hints are expression transformation hints. Those indications

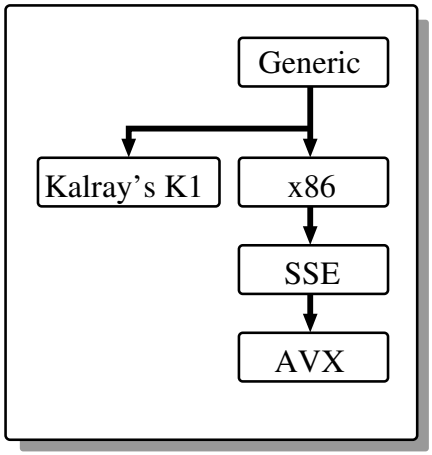


Figure 5.4: Metalibm architecture-description hierarchy

are used by Gappa rewriting engine to simplify expressions. Those simplifications are performed to bound more accurately the expression. Hints example can be found in Listing 4.2, p.82; they were used to exhibit the Newton-Raphson convergence property. Metalibm provides two ways to indicate hints to Gappa:

- an indirect way: the **exact** annotation, described in Section 5.3.1. When encountered by the proof generator, this annotation is translated into a hint describing the equality between the exact and evaluation sub-expressions.
- a direct way: the Gappa script generator provides a method called **add_hint** which allows the developer to directly write a hint, as a piece of Gappa script. This hint will be copied into the generated Gappa script.

A hook mechanism is provided to link a generated variable to the original node in the implementation description. These two ways are other examples of our metalibm philosophy: give the implementation developer the capability to transmit his knowledge into the implementation representation.

5.6.2 Code generation and target specific processing

Code generation generation constitutes the backend of our metalibm. It is the process of translating the IR to source code. The generated source code implements the function described in MDL. Two code generators are available in the current version of Metalibm. The first one generates Intel specific code, in an Intel proprietary language. It targets statically vectorized implementations and provided some of the results described in Section 5.8. It was developed during an internship, and is the property of Intel. It is not integrated into Metalibm trunk (which should be open-sourced). This section focuses on the other: a C code generator. It is provided with a generic C implementation description. It can be extended with target descriptions for architecture specific generation.

The code generation process of metalibm was design to overcome one of mathematical libraries drawbacks: their genericity. For example, the mathematical library provided in the GNU's libc [62] is a generic software implementation developed in portable-C. Its first advantage is its portability, since it only requires a compliant C-compiler to bring the mathematical library support to a new platform. However such genericity as an efficiency cost. Indeed, the library performance relies exclusively on the compiler abilities. Architecture-specific tuned-up implementations

are offered by manufacturers: Intel's Math Kernel Library (MKL) [75], AMD Core Math Library (ACML) [8]. But their development is very time consuming. It requires several full time engineers to optimize and maintain those libraries. Each new architecture modification necessitate new highly-optimized implementations.

```
1    # defining the interface of the MPFMA operation
2    MPFMA_SIGNATURE = (ML_Binary64, ML_Binary32, ML_Binary32, ML_Binary64)
3
4    # defining a new architecture
5    class Kalray_K1(GenericProcessor):
6        implementation_table = {
7            FMA: {(ML_Binary32,)*4: OpFunction("__builtin_k1_ffma"),
8            FMA: {MPFMA_SIGNATURE: OpFunction("__builtin_k1_ffmawd"),
9        }
```

Listing 5.7: Example of architecture description, deriving from **GenericProcessor**

Our philosophy is to disconnect implementation description from code specialization. To do so, metalibm uses the Metalibm Description Language (MDL), addressed in Section 5.3, and a code generation process which can be parametrized by an architecture description. The new architecture description is independent from the implementation description. Target-optimized implementation of MDL's operations are selected during code generation. A code generator inputs both the IR and an architecture description. Listing 5.7 gives an example of architecture description. The architecture description is a derivative of the base class **GenericProcessor** which corresponds to a generic C implementation. This target supports every operation described in MDL. A custom architecture description can be derived from this class by overloading all or part of the operation allocation table **implementation_table**. To implement an operation *op*, the code generator, first, searches for an implementation in the processor given as parameter to the generator. If this architecture description implements this operation, the generator generates this architecture-specific implementation in the source code. If not, the generator goes up into the processor parent hierarchy, until it finds a description that implements *op*. As every processor is a derivative (possibly after multiple levels) of the class **GenericProcessor**, an implementation of *op* is find at least at the **GenericProcessor** level. The hierarchy of architecture description is illustrated by Figure 5.4. This processor overloads the generator for the FMA operation and replace the generic C code by a call to a specific built-in intrinsic.

Metalibm provides various utilities to define new architecture descriptions, such as the **OpFunction** method showed in Listing 5.7. Metalibm generation can be extended to other languages, by implementing the corresponding code generators.

5.7 Vectorization support

To illustrate the capability of metalibm we developed statically vectorized functions (with correct rounding). This application case study will be described in Section 5.8. Let us first introduce the vectorization challenge and focus on Metalibm vectorization support

A vectorized function implementation inputs and outputs a vector. It evaluates the function on each element of the input vector and stores each results in the corresponding index of the output vector. This process is highly parallel, each element computation is independent. To exploit this parallelism, CPUs have been extended with vector computation capabilities. Vector computation is provided by vector functional units (VFU) and vector register files (VRF). Each register of a VRF contains multiple data: typically a vector of 2 to 8 chunks. A VFU inputs and outputs vector registers, it is built from multiple scalar units, called lanes. Each lane computes the same operation on a different chunk of the input data and returns a different chunk of output. This mechanism is

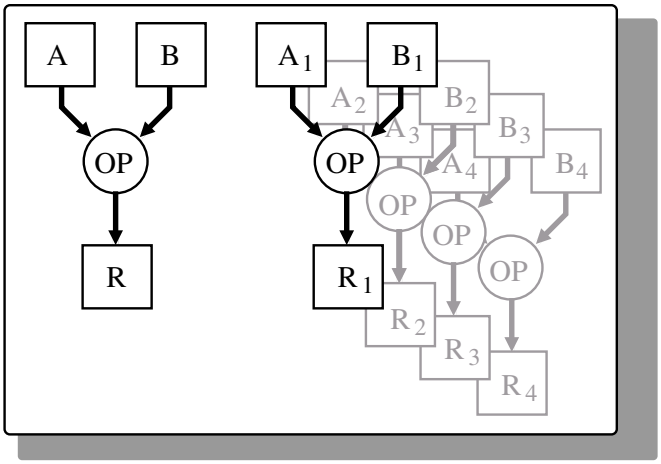


Figure 5.5: Scalar operation scheme (left) comparison to vector operation scheme (right)

illustrated by Figure 5.5. The operation **Op** is the same for all 4 lanes, inputs and outputs are lane-independent and stored in a vector register file. Vector units implements the Single Instruction Multiple Data (SIMD) architecture.

Increasing vector performance, has been a common trend in the general public market. In 1997, Intel's MMX extension introduced SIMD to x86-based processors. It uses 64-bit registers. MMX offers the ability to execute a single operation on up to 8 lanes, each processing a byte-level operation. For example, a single MMX instruction can perform 8 additions $byte + byte$ in parallel. In 1999, x86 was extended with SSE (Streaming SIMD Extensions). SSE implements 128-bit registers and a vector unit which can process up to four binary32 operations parallel. In 2008, x86 was extended with yet an other vector extension: AVX (256-bit registers and up to 8 binary32 or 4 binary64 operations processed in parallel). Those are only a few example among many vector extensions (ARM's NEON, AMD's 3DNow!, x86 AVX2...). Vectorization is the process of transforming a scalar implementation into a vectorized implementation, more suited to process vectors. To exploit vector instruction set, vectorization support has been implemented in most compilers: GCC [118, 119] ICC [80], LLVM [24], open64 ... However, to this day, this support remains limited to well-formed loops and easy to vectorize cases.

There are two solutions to build a vectorized implementation:

- generate a branch-less scalar implementation and wrap it in a well-formed loop. The compiler is in charge of the vectorization.
- statically vectorize the source code before compiling. In this case the developer, or code generator, performs the vectorization.

To simplify the compiler task, the first approach requires a control flow as uniform as possible. There should not be any branch in the scalar implementation. Branches could trigger divergent behavior among vector elements. As most vector unit do not support divergent lane behaviour, the vector processing would be sequentialized, loosing the vector unit benefit.

The second approach does not rely on the compiler capability to vectorize. It consists in performing the vectorization beforehand, for example by using intrinsics to force the use of vector instruction. Intrinsics are functions which directly map to machine instructions. It forces the instruction selection, gives the compiler less possibilities to optimize the code. This approach relies on a strong assertion: the developer knows its implementation behavior. He knows which part could benefit from vectorization and which part must be modified to better suit vectorization.

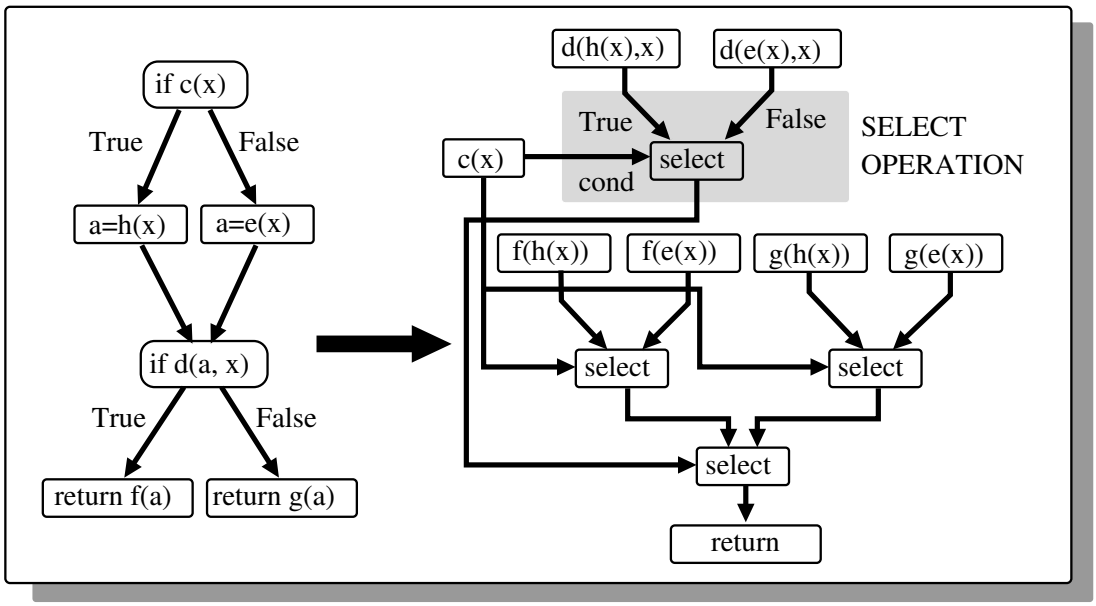


Figure 5.6: Example of blending result reduction effect on an operation graph

The first approach is easier to integrate: as a scalar implementation is built, it can easily replace existing implementations. The second approach builds a vectorized-implementation which does not fit standard function interfaces. Some language such as OpenCL already contain vector types (eg: *float**n***, *double**n***). Vectorized functions with vector length matching those of the language definitions can be integrated without difficulty. In other cases, vectorized implementations require the definition of a new API. To benefit from these implementations, the application developer has to use this specific API. Generating a branch-less scalar implementation is the easiest, but it provides less control over the final vectorization. The second approach requires more involvement from both function and application developers but it brings more control over the vectorization. To give the developer maximum flexibility, both approaches were implemented in Metalibm. To support both approaches, metalibm can:

- generate inline code without branch or callout, to simplify the compiler vectorization process.
- directly generate vector code (using processor intrinsics to force the use of vector instructions)

The remaining of this section describes some of Metalibm mechanisms developed for this support.

5.7.1 Result blending

The first mechanism implemented in Metalibm to support vectorization is result blending. This mechanism consists in extracting the various paths of the operation graph which return a result. Then, those paths are flatten so each a result is generated separately for each path. Finally, the function results selection is generated. Figure 5.6 gives an example of such operation graph transformation reduction. The operation has to be flattened so every branch operation is transformed into a select operation. A select instruction is a ternary operator, defined as $select(a, b)_{cond} = a$ `if` $cond$ `is true else` b. Contrary to a control flow branch, where only one side of the branch is executed, both a and b are evaluated before the select operation is performed.

There are several ways a graph can be flattened. Figure 5.6 illustrates a case where every possible execution path has its own new node in the flattened graph. Sub-expressions could be factorized. An other solution is to compute the value in order: $c(x)$, $a = select(h(x), e(x))_{c(x)}$ and $f(a)$, $g(a)$, $d(a, x)$ before finally performing the result selection $result = select(f(a), g(a))_{d(a,x)}$. In this case the execution is sequentialized and not all the available parallelism is expressed. But f and g are only evaluated once, contrary to twice in Figure 5.6 example. Metalibm version of this mechanism lists every possible execution path before generating a different operation sub-graph for each path, as illustrated by Figure 5.6. A future development perspective is to explore the possibility of choosing between the various possibilities according to their respective efficiency (eg: number of operations, latency).

This mechanism support both vectorization approaches. For scalar implementation, it deletes every branch operation and generates a branch-less operation graph. For statically vectorized implementation, it is used to implement branch operations with hard to predict behaviour.

5.7.2 Callout extraction

A second mechanism, used only for static vectorization, is the callout extraction. A callout is a function which evaluate a sub-part of the implementation. It is implemented in a separate compile-unit from the main function which implements MDL description. Callout extraction mechanism is part of the code generation. It is triggered by MDL **callout** annotation. For annotated nodes, the code generator extracts the sub-graph constituted by the annotated operation and it dependency graph. The dependency graph is restricted to operation which have not been generated yet. The resulting sub-graph is generated as a callout. When the depth-search, building the sub-graph, encounters a node already generated, it lists it as an argument of the callout function. The code generator automatically determines the result format according to the node **precision** field, and use it along the argument list to define the callout interfaces. The callout function is generated separately in a separate compile-unit from the function core. Inside the core, the node operation is implemented as a function call to the callout. This feature allows metalibm to extract low-probability branches from the core part of the operation and externalize their evaluation into callouts.

5.7.3 Static vectorization implementation

To provide static vectorization, Metalibm implements a technique developed by Intel MKL team, for the manual development of Intel Vector Math Library (VML). VML supports arbitrary-length vector and is intended for large vector (typically more than a hundred elements). We consider the same large vector target for our vectorized function development. Intel's technique is based on a partition of the vectorized function in two parts: a **core part** and a **callout part**. It is illustrated by Figure 5.7. The most likely execution path is extracted from the function Control Flow Graph (CFG), it forms the base of the **core part**. This core part is extended with a validity flag computation. This validity flag indicate if the input matches the core path condition or not. The part of the control flow graph not covered by core part forms the callout part. Typically, this part manages special cases. From this partition a vectorized implementation is constructed as follows:

- an unrolling factor (UF) is selected. It indicates how many function evaluation are processed in parallel
- UF instances of the core vector are instantiated by implementing UF-wide vector operation for each operation of the core part operation graph. This includes the computation of a UF-wide vector of validity flag, this vector is called the validity mask

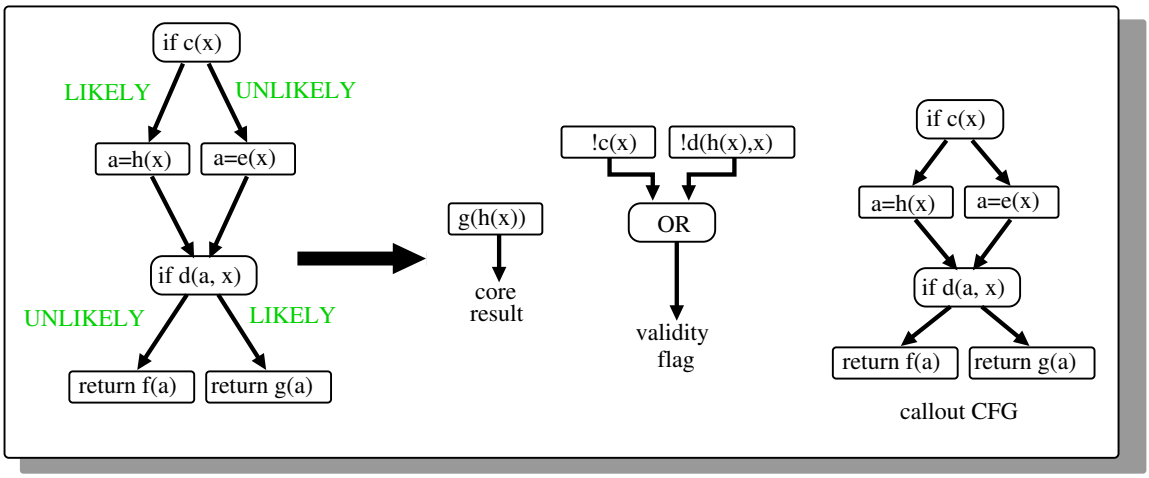


Figure 5.7: Example of core, validity flag and callout extraction

- Once the UF core instances have returned the result vector, the validity mask is checked. If a single flag is invalid, the vector execution is stalled. A sequential loop iterates on the validity mask, it executes the callout function for each input flagged as invalid. The corresponding element of the result vector is replaced by the callout result.

This implementation computes UF results simultaneously, it can easily be extended to an arbitrary-length vector by managing vector start and end separately. Since the core part corresponds to a single execution path, it is branchless and a good fit for the easy vectorization using UF-wide vector operation. Generally many UF values are tested to determine the best value to implement large vector computation. A too low UF value (eg: 2 for a binary64 computation on a AVX-capable architecture) does not allow to use the full potential of available vector units. A too large UF value requires too many simultaneous computations overloading the architecture resources (eg: registers, functional units) and eventually slow down the computation. Moreover, a larger UF value increases the probability of a callout happening during the vector computation.

Intel technique requires manual development, the core part and the callout part are partitioned by hand, as is the validity flag implementation. Our metalibm provides a vectorization backend which automatically splits an operation graph between a **core part** and a **callout part**. This backend relies on the **likely** annotation to determine the most likely execution path. If this annotation has the **Likely_Possible** value, then the backend calls the blending result process to generate both branch sub-graphs and insert a select instruction. During the core graph generation, this backend automatically extracts every control flow test it encounters. It gathers them to generate the validity flag computation. The callout generation is performed by the callout process which automatically determine the function interfaces. In this cases, the callout process does not limit the sub-graph to not encountered nodes. As the callout will only receive the input as parameter, it implements the complete graph. This backend simplifies vectorization. A vectorized implementation is generated from the same MDL description as a scalar implementation. It implements our philosophy: generating multiple implementations from a single description.

5.8 Application: vectorized correctly rounded exponential and logarithm

The metalibm framework has been used to develop several examples. These examples include two original contributions: a vectorized correctly-rounded exponential and a vectorized correctly-rounded logarithm. To our knowledge, it is the first example of attempt at vectorizing efficiently correctly rounded functions. These applications do not target embedded systems. They were developed for Intel's desktop/server architectures. However we think it is relevant to integrate them in this work because they illustrate the capability of Metalibm to generate codes for several platforms and constraints.

Every evaluation is made by executing the implementation on a large vector of inputs, even for scalar implementation. All performance results are given in Clock Per Element (CPE). CPE is a throughput measure. It is defined as the ratio of the function execution time, on the whole vector, divided by the vector size.

This work started by the development of scalar version of binary64 exponential and logarithm on Intel's SSE2 architecture. Metalibm generated the source code from an implementation described in MDL and an architecture description describing some x86-SSE2 implementations. Let us use these examples to illustrate the benefit of the architecture specific generation provided by our metalibm. In standard C99 the conversion from a floating-point number to the nearest integer is managed by the function **lrintf** for binary32 and **lrint** for binary64 precision. Since SSE2, Intel's processor implements an instruction that performs this specific operation for binary64: ***CVTSD2SI***. This instruction is accessible through a specific C intrinsic: ***_mm_cvtsd_si32***. Unfortunately, a call to **lrint**, even in the highest level of optimization supported by GCC, is not directly to the ***CVTSD2SI*** instruction. Rather, it is mapped to an effective function call instruction to lrint. This is detrimental to performance, because of the overhead of the function call and return. The compiler can not inline lrint call because the libm, which provides **lrint**, is built separately from program and the compiler.

Metalibm does not have this problem: if the SSE2 architecture description is specified to the code generator then the NearestInt Operation, assuming suitable format, is translated into ***_mm_cvtsd_si32***. Writing code with such intrinsics is detrimental to code portability. However this drawback does not concern metalibm, since it aims at generating architecture-specific code. Moreover, architecture description and implementation description are separated, it is easy to generate both a generic, portable implementation and an architecture-specific one. If a specific implementation like ***_mm_cvtsd_si32*** is available then our metalibm inserts it in the generated code, if not it falls back on the generic **lrint**. This is a big advantage of target specific code generation: it can tune the implementation to the target architecture without requiring implementation description modification. This tuning is transparent for the developer: the automatic code generation process takes care of such decisions. To be honest, when designing the implementation algorithm, the function developer should be aware of the underlying architecture capabilities. It may lead to chose some implementation over others, for example to avoid slow operation on the architecture. The current version of metalibm relies on the function developer to make the good choice when describing its implementation.

Let us now present the evaluation result for the scalar implementations. Table 5.2 presents performance results for scalar exponential and logarithm, in both binary32 and binary64 (except logarithm which was not implemented in binary32 in our framework). The error is the implementation error provided by IMLTS, a proprietary test system developed at Intel. It should not be considered as a strict relative error measurement: $0.5ulp$ does not imply correct-rounding. **libimf_32e** is Intel's proprietary mathematics library, used in their compiler ICC. It has been carefully and manually optimized for each architecture and should represents the best performance.

Function	expf		exp		logf		log	
Description	CPE cycles	error ulp(s)	CPE cycles	error ulp(s)	CPE cycle	error ulp(s)	CPE cycle	error ulp(s)
libimf_32e	28.99	0.5066	46.36	0.512	33.5	0.503	45.89	0.503
glibc_32e	72.16	0.5005	80.38	0.5	57.75	0.828	130.86	0.5
early metalibm	46.56	0.5065	54.73	1.1	43.09	0.503	89.36	1.71
our metalibm	84.49	0.51	63.12	0.52			79.62	0.5009

Table 5.2: Performance results of scalar function generated by our metalibm and compared to manual implementations

Architecture	Description	CPE (best UF)
SSE3	fp64 VCR log	35.34 (4)
AVX	fp64 VCR log	21,81 (4)
Xeon-Phi™	fp64 VCR log	45.03 (32)
AVX2	fp64 VCR log	17.98 (8)
SSE3	fp64 VCR exp	29.98 (8)
AVX	fp64 VCR exp	20.99 (8)
Xeon-Phi™	fp64 VCR exp	63.1 (32)
AVX2	fp64 VCR exp	17.84 (8)
SSE3	fp64 vectorized ML's exp	19.39 (8)
AVX	fp64 vectorized ML's exp	11.59 (8)
SSE3	fp64 vectorized Intel MKL's exp	10.81 (16)
AVX	fp64 vectorized Intel MKL's exp	7.59 (16)

Table 5.3: Performance results for vectorized correctly rounded exponential and logarithm on several Intel's architectures

glic_32e is GNU's libc, its code is always slower than Intel library. This could be partially explained by its very good accuracy. For example, for binary64 exponential, the test system did not find any not correctly rounded result. **Early metalibm** represents an early stage of metalibm development, before MDL and the core engine were introduced. As illustrated by the table, it still needed tuning and generated some inaccurate implementation. Nonetheless, it already delivered promising accuracy-result, for the generated expf and logf implementations. **Our metalibm** presents the result obtained with the framework described in this chapter. It also was at an early stage of development. Compared to Intel's libimf, our expf is 2.91 times slower, our exp is 1.36 times slower and our log is 1.74 times slower. There are still a lot of work to be done before we compare favorably to manual implementations.

The scalar description was then used to generate vectorized implementations. This was performed by using the vectorization backend to separate the description in two parts (core and call-out). Our metalibm was extended with an extra target description to directly generate VLANG. VLANG is a C-based language, used internally at Intel. VLANG can be post-processed to generate vector implementation for several architecture and several unrolling factors. Table 5.3 presents the performance results for our vectorized correctly rounded function. The error was measured to be less than the half ulp by Intel test system, extended with some hardest to round cases listed in [116]. For each table entry we evaluate several UF. The table only reproduces the best result, indicating which UF was used to obtain it. The final 4 lines list the result for vectorized non

correctly-rounded exponential generated by our metalibm compared to Intel's MKL state of the art implementation. It shows that our implementation is 79% slower on SSE3 architecture and 53% slower on AVX architecture. Part of the performance gap is explained by the fact that our implementation is a little more accurate than Intel's one: error of 0.503 ulp(s) versus 0.508 ulp(s). But it does not explain such large difference. As for the scalar implementations, there is still a lot of room for improvement before our implementations can compare favorably to manually written functions.

5.9 Conclusion and perspectives

This chapter has presented our vision of metalibm: a framework to help libm developers rapidly and efficiently target new architectures and multiple requirements. Our framework is divided between several key elements. The first one, the input, is an implementation description using a domain specific language: MDL. The second, Metalibm's core, is formed by an IR, derived from MDL, and an optimizer. The optimizer manipulates the intermediate representation to optimize its future translation into source code. For example, Metalibm provides some IR manipulation to prepare for vectorization (eg: automatic core and callout partition or result blending). Finally metalibm backend is constituted by several code and proof generators along with some target descriptions. The target descriptions allow the backend to generate architecture-specific code (eg: by using intrinsics). An advantage of our framework is that both the proof and code generations input the same source,the intermediate representation. As long as each backend makes the same assumption on the IR, the proof validity is certified. Our metalibm also provides some helper tools: such as a scalar vectorizer which exposes ILP in scalar code.

In spite of still being at an early development stage, our Metalibm, has been used to generate new implementations, such as vectorized correctly-rounded functions. It has already known several forks (Florent's, Intel's, Kalray's). They should be merged in the near future. Our project and C. Lauter's [86], are now part of the ANR Metalibm project [41] which started during the redaction of this manuscript. [86] is much more advanced on automatic function generations. We think our approach is more promising than Lauter's concerning architecture specific generation. The next challenge for our metalibm is to generate optimized libm functions, tuned for Kalray's K1, to replace generic newlib's implementations. We are also studying the development of data layout optimization. This is part of a wider effort, with the scalar-vectorizer enabling, to support vectorization.

Part II

Architecture and application of a deeply integrated reconfigurable fabric

CHAPTER 6

Reconfigurable taxonomy and state of the art

In the previous chapters we have studied the efficient implementation of floating-point arithmetic. Our point was to demonstrate that hardware and software design should coexist to balance performance and cost. Implementation method had to be selected depending on operation frequency, resources and development time available.

We now suggest the study of an other solution to improve arithmetic support in embedded system: the deep integration of reconfigurable fabric into an embedded core. This chapter is dedicated to a small taxonomy of reconfigurable computing solutions. Chapter 7 presents a new reconfigurable matrix architecture and micro-architecture. Chapter 8 presents coprocessor architectures to integrate the reconfigurable matrix into single-master and multi-master systems. Finally, Chapter 9 describes the software framework developed to supports the new reconfigurable architecture.

Reconfigurable computing can be defined as using a hardware still modifiable after fabrication (tape-out). Hardware configuration can be changed statically, at boot time, or dynamically, during runtime. Here configuration covers both the hardware functionality and its connectivity. Figure 6.1 illustrates the position of reconfigurable computing between Application Specific Integrated Circuit (ASIC) and software. Reconfigurable hardware is more efficient than software and more flexible / less development-time consuming than ASIC.

Our objective is to provide reconfigurable computation capabilities to a CPU core, the K1 processor. We design a reconfigurable matrix architecture, its coprocessor and develop a software-support framework. We try to focus on providing architectural exploration capabilities to determine the best architecture for our target.

The remaining of this chapter is dedicated to a study of some of the existing reconfigurable fabrics. A very large number of reconfigurable architectures have been suggested over the years. We selected three representative works. We start with the most famous reconfigurable architecture:

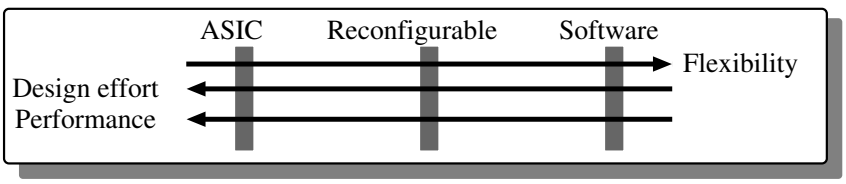


Figure 6.1: ASIC, Reconfigurable and Software ordered according to flexibility, design effort and performance

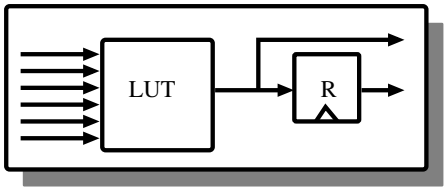


Figure 6.2: LUT and registered/direct output

Field Programmable Gate Arrays (FPGAs). We continue with two other representative projects. We starts with Pico-GA which is a medium-grain reconfigurable functional unit integrated into a RISC pipeline. Finally we describe the TCE project (a.k.a MOVE) which implements a Transport-Triggered Architecture (TTA). It provides architecture exploration capability and static instruction-set extension.

6.1 FPGAs

Field Programmable Gate Arrays (FPGAs) are the most common reconfigurable hardware used today. The main manufacturers are Xilinx (eg: Virtex family) and Altera (eg: Stratix family). FPGAs were derived from Programmable Logic Devices (PLD) more than 30 years ago and constituted an evolution in configurability. FPGAs are now offered as stand-alone chip or integrated into more advanced System on Chips (SoCs) were they are connected to General purpose CPU cores. Nowadays, FPGA chips contains hundred of thousands of reconfigurable gates connected by a reconfigurable network. A complete computing systems can fit (eg: MPSoC [100]) in a single FPGA. The configuration is generally loaded at start-up, but dynamic reconfiguration, during run-time, has become possible on some of the more advanced chips [17].

6.1.1 Reconfigurable Logic Cell

An FPGA is set of reconfigurable logic cells (RLCs), hierarchically organized, and integrated into a configurable interconnect network. In nowadays FPGA, RLCs are implemented by Look-Up Tables (LUTs). A LUT is a memory, whose content is part of the FPGA configuration. FPGA LUTs have a few inputs and outputs, typically between 4 and 6 inputs and between 1 and 2 outputs. It is addressed by the RLC inputs and its outputs constitutes the RLC output. As illustrated by Figure 6.2, RLCs also contain a register after the LUTs output to offer pipelining and delay insertion capabilities. LUT-based RLCs give to FPGA a very fine-grain configurability. Each output bit functionality is defined independently.

6.1.2 Routing network

The FPGA effectiveness, apart from bit level functional configurability, comes from the configurable interconnect which surrounds the RLCs. Connections between RLCs are not determined statically when the chip is melted but are determined by the FPGA's configuration. To be efficient the routing is divided into several hierarchical levels. The fine grain routing is done inside the RLC. The medium and coarse grain routing are made between clusters of RLCs using longer connections.

The place and route (P&R) is an important step of FPGA configuration compiling. It corresponds to:

- the mapping of a cell configuration into a physical cell positioned in the FPGA's RLC matrix;
- the computation of the routing network configuration.

There are so many physical cells, connections and routing possibilities inside an FPGA that, in practice, the P&R steps can not be solved optimally. Many solutions ([16]) corresponding to as many heuristics have been suggested. Derivatives of a method called **simulated annealing**, seem to be the most efficient techniques currently in use. As a result, FPGAs can not be fully utilized (100% RLCs used) to allow P&R to succeed.

6.1.3 Fast arithmetic support

The first FPGAs were LUT-based only. It was discovered that this was not a good fit for some, intensively-used, primitives such as integer arithmetic operations (additions, multiplications).

Let us consider the example of the addition. We can divide adder implementation in two categories: linear scheme adder and more advanced scheme. Both are not a good fit for FPGAs. A linear adder scheme requires a carry-chain. Implementing such a scheme in FPGAs requires to a linear dependency chain connecting a large number of RLCs, each RLC computing a few bits of the addition result. The long dependency chain puts a lot of pressure on the routing network, resulting in a high latency. The more complex schemes (eg: carry look-ahead, carry skip) proved too expensive in area. To provide fast adder implementation, FPGA designers integrated fast carry chains in the FPGA fabric. Such a chain goes through a line of multiple RLCs without using the general purpose routing network. It allows for fast implementation of linear dependant logic in particular adders. The chain is so efficient that, in practice, adders up to tens of bit wide can be implemented using the linear adder scheme with very low latency.

For multipliers the situation is a little different. A multiplier is inherently an expensive block, even in Application Specific Integrated Circuit (ASIC). It is built from two expensive sub-blocks: the partial product array and the compressor tree. Its area is proportional to the multiplication of its input dimensions. Nonetheless, one of FPGA bigger markets, the small series signal processing chip, relies on multiplication. Thus DSP blocks containing configurable multipliers were integrated into FPGAs. These pipelined blocks implement a MAC operation. It can be configured into several smaller or one bigger multipliers/accumulators. The level of configurability of such blocks is very restrained compared to the RLC capabilities but they implement several schemes of multiplication very efficiently. Despite of this lack of flexibility, confirmed FPGA developers know how to use such blocks to efficiently implement any multiplier sizes [45].

6.1.4 Memory block

Additionally to the RLCs and DSP blocks, FPGAs also contain memory blocks. Those large RAM blocks are implemented using a much more cost-effective technologies than the flip-flop implementing RLC registers. They are used to implement memory blocks up to tens of KB without wasting FPGA's functional logic. The integration of such blocks facilitate the use of FPGAs as prototyping platform or to implement softcore processors.

6.1.5 Conclusion: FPGA versus ASIC as computing systems

Figure 6.3 describes the structure of the reconfigurable matrix of an FPGA. One can notice the multiple levels of routing: inside the cluster of RLCs, locally between a group of CRLC [1] and at a more coarse grain between multiple groups of CRLC. It should be noticed that this structure is

[1] Cluster of Reconfigurable Logic Cells

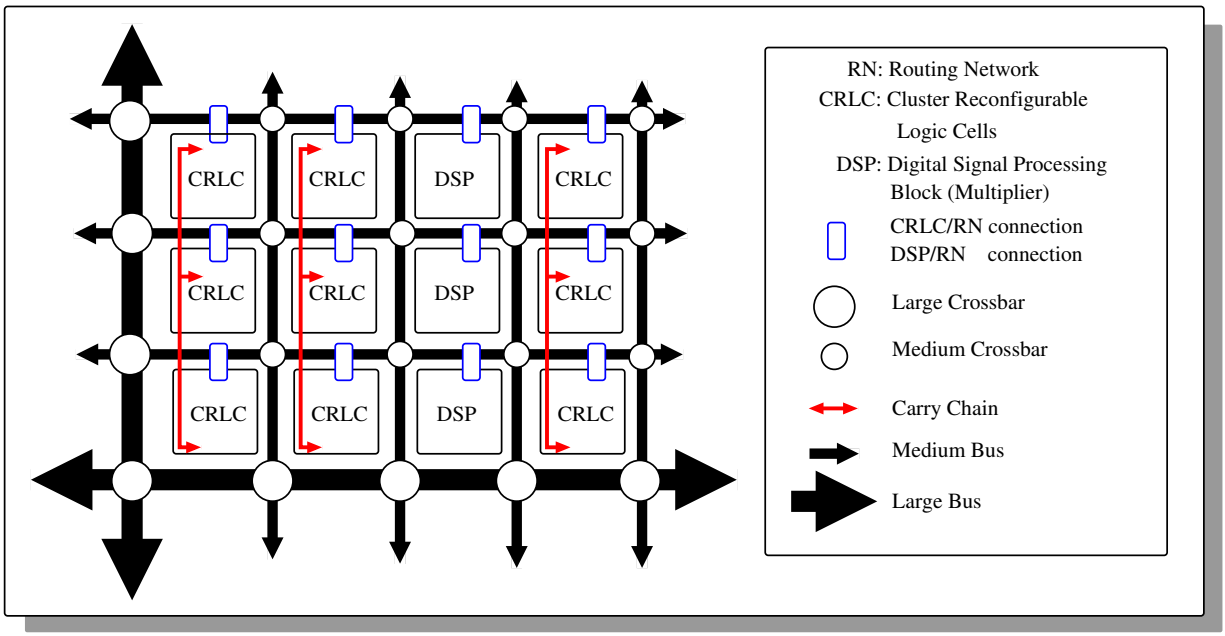


Figure 6.3: General architecture of an FPGA matrix

very hierarchical. Most connections take place locally. Expensive long wires are still required to implement the very long connection, to connect multiple groups of CRLC together.

The first market of FPGA is the prototyping of numerical circuits (ASIC) and small series circuits which can not justify the development cost of an ASIC solution. They did not replace ASIC in many domain, the main reason may be that FPGA flexibility comes at a cost. FPGAs run slower and are less power efficient than a dedicated hardware implementation (ASIC). The performance gap between FPGA and ASIC have been measured, for example in [85]. They estimate than on average an application mapped onto an FPGA is 40 times larger, 3.2 times slower and consumes 12 times the amount of dynamic power a ASIC synthesized operator would. That can be sumed up abruptly: if you have fewer than 40 applications to run and enough design workforce, then reconfigurable fabric is not the best fit. To counter this disadvantage, more efficient reconfigurable structures were studied. Those structures lose some flexibility in favour of better performance. Their frequency should be higher and they should be more efficient on a reduced set of applications. Our work is part of that philosophy. Before going into the details of our project, let us describe two other interesting reconfigurable solutions that inspired our contribution.

6.2 PiCoGA and the DREAM Digital Signal Processor

The DREAM project [113] designed a medium grain reconfigurable fabric. Medium-grain implies that RLC I/O and routing network are not configurable at the fine-grain or bit-level. In the case of the DREAM project, the routing network uses 4-bit wide buses, the RLCs have 4-bit wide I/Os. LUTs are organized with 2-bit wide inputs. This project is interesting in our perspective for several reasons. The first one is that we both implemented a medium-grain fabric. The second one is that it is one of the few project which developed a toolchain to support development on the hardware. This toolchain is detailed in [112]. The third one is we share one of the application cases: AES encryption.

The idea behind the DREAM project is to integrate reconfigurable capabilities into an hetero-

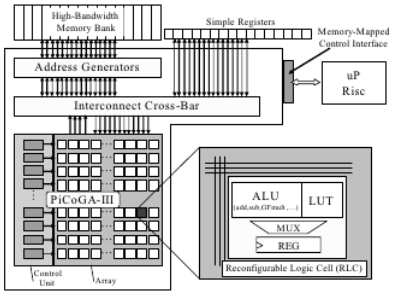


Figure 6.4: The DREAM architecture (figure extracted from [23])

geneous system on chip (SoC). The description of the DREAM processor is available in [23] and [113]. The module should be able to interact with the whole system through memory exchanges. For efficiency purpose the control is performed by a small area RISC processor rather than dedicating part of the reconfigurable hardware to the control. The system provides 4 configuration contexts with only one cycle swap latency. The use of several contexts with such a small swapping time helps masking the configuration latency: a context can be loaded while an other is used to process data. The DREAM project architecture contains:

- a RISC processor
- a configurable datapath (The Pico-GA)
- a configurable control unit
- a configurable address generator
- exchange buffers, high bandwidth memory banks

The architecture of the DREAM processor is illustrated by Figure 6.4.

The reconfigurable datapath is called PiCoGA [96]. It is built around a medium grain reconfigurable cell working on two 4-bit inputs and one 4-bit output. The datapath is organized along one dimension, as a pipeline: multiple RLC rows are connected together and correspond to as many pipeline stages. The RLC contains an ALU and a 64-bit LUT (4-bit address and 4-bit data). The RLC's ALU is built around standard operators such as adder, subtractor, multiplier slice. It also incorporates a more exotic Galois Field multiplier on $GF(2^4)$. The Galois Field multiplier has been incorporated to support specific cryptography kernels such as the Advanced Encryption Standard (AES [142]). AES is one of the key application targeted by the DREAM processor. We are going to make similar choices in our architecture: integrate effective hard-macro for key applications rather than mapping them inefficiently on not well-suited reconfigurable hardware. This choice is intended to keep the RLC somewhat simple while increasing performance for key kernels.

The routing has three components:

- global vertical routing: wires that can bring input to any rows and route output from any row
- global horizontal routing: wires that connect cells in the same row (eg: shift implementation without logic occupation).
- local segmented lines (3 RLCs by segment) and local direct connection to connect neighbour cells in the same column.

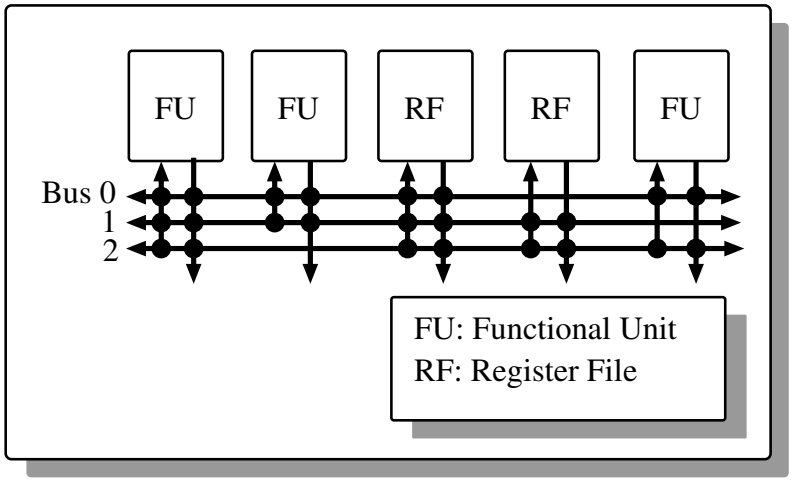


Figure 6.5: Example of transport-triggered architecture

The software toolchain is detailed in [111]. It provides a compiler of a subset of C called Griffy-C. Griffy-C provides similar expressiveness to an Hardware Description Language (HDL).

6.3 TCE Project

The TTA-Base Co-design Environment (TCE, [53]) offers a statically reconfigurable processor. TCE provides a softcore processor implementing a generic Instruction Set Architecture (ISA). The ISA can be extended and the environment generates automatically the corresponding architecture and toolchain. This generation is static, once the processor has been mapped onto an FPGA it can not be modified. Even with that constraint, the TCE project is very interesting because it implements an architecture that supports ISA extension. This architecture is based on the transport triggered architecture model. We are going to study that solution for the reconfigurable fabric integration into a multi-processor system. This will be the focus of Chapter 8. Let us now introduce the transport-triggered architecture (TTA).

6.3.1 Transport-triggered architecture

A TTA is an architecture where data transport triggers computation. It exposes register files (RF) as any other functional units to the program control. There is no functional instruction. Data move is the only operation provided. This operation moves a data from a functional unit (FU) to an other through an available transport bus. Some specific moves to FU trigger computation: the FU pipeline goes forward one step when data is sent to those specific interfaces. Thus the program has complete control over the hardware. It controls when a data is output by a FU since this event is triggered by a fixed number of data moves to the same FU. Figure 6.5 illustrates an example of such architecture with 3 functional units and 2 register files, interconnected by 3 independent buses.

The architecture can exploit more Instruction Level Parallelism by adding more buses and more units. Each new buses allows an extra data move and can then trigger a new computation. One of the big advantages of TTA is that the architecture ignores the complexity of its functional units. It does not need to implement bypasses or multi-port register files to support multiple operators. It can easily support operators with multiples latency. Virtually an arbitrary number of FUs, RFs and buses can be supported.

6.3.2 TCE configurability

The power of TCE is to provide both a generator for the RTL description and a retargetable C-compiler to support the generated architecture. TCE base architecture can be extended with extra pre-defined register files and functional units. Custom operators can also be described and integrated into the architecture. The framework automatically generates a simulator and C-compiler for the modified architecture. All these features make TCE a remarkable architecture exploration tools. These tools can be used to evaluate the performance consequences of architectural modifications. This is the subject of [53] which introduces the framework and compares it with known softcore processor on established benchmarks. Its conclusion is that the TCE architecture remains more costly than most softcores (eg: Altera's Nios or Xilinx's Blaze) but the flexibility of the framework and the higher configurability makes it a great tool for architectural space exploration.

The TCE project is still on going. Nowadays the community is trying to bring OpenCL [143] support to the architecture. This is done through the POCL (Portable Computing Language [4], [78]) project which contains TTA platform produced by TCE as one of its targets.

CHAPTER 7
Deeply Integrated Reconfigurable Fabric

Our objective is to design a new reconfigurable fabric architecture that diminishes the gap between ASIC and FPGA. Our project mixes some of the features of the Pico-GA project described in Section 6.2 with the TTA architecture of the TCE project described in Section 6.3. In a simplified approach, our integration follows the same path as the PicoGA: the reconfigurable matrix is integrated as a functional unit to a CPU core. The CPU integration provides the necessary memory accesses, commands, and data inputs/outputs. The CPU is also in charge of managing DIRF configuration. To reach a better efficiency the main ideas developed in this work are: restrain the reconfigurable matrix structure, simplify its integration into an embedded system and provide a support software framework. The restriction to the matrix structure aims at providing more performance (eg: higher frequency). To balance this restriction, and maintain matrix efficiency, we introduce some new micro-architectural features such as a generalized carry chain. We also contribute a permutation-based routing network, designed to support dynamic rotations.

Our reconfigurable matrix architecture is called the Deeply Integrated Reconfigurable Fabric (DIRF). It proposes yet an other response to the challenge of designing a reconfigurable architecture suited for integration into embedded systems. This chapter focuses on the matrix architecture and micro-architecture. The DIRF is embedded into a coprocessor which manages the interaction with the surrounding system. The architecture of this coprocessor is the subject of Chapter 8. The DIRF is supported by a software framework. The software framework is intended, first to provide architectural exploration possibilities and then to support the development on the reconfigurable matrix. Chapter 9 addresses this software framework.

7.1 Overall view

The DIRF is a unidirectional medium-grain reconfigurable matrix designed as a straightforward pipeline. Its structure is illustrated by Figure 7.1. The inputs always originate from the same matrix side: the input registers located at the top. They are processed in the same direction and always output at the same location: the output registers. The matrix starts with input registers and ends with the output registers. In between, it contains many RLCs, typically between one hundred and a thousand, organized as rows of cells. Each row is connected only to the previous and the next rows, by two separate interconnect networks. The first top row is connected to the input registers with a similar interconnect, as is the last bottom row to the output registers. There is no direct connection between cells of a same row except a generalized carry chain which will be detailed in Section 7.3.1.

The DIRF architecture is different from the general FPGA structure, presented in Figure 6.3:

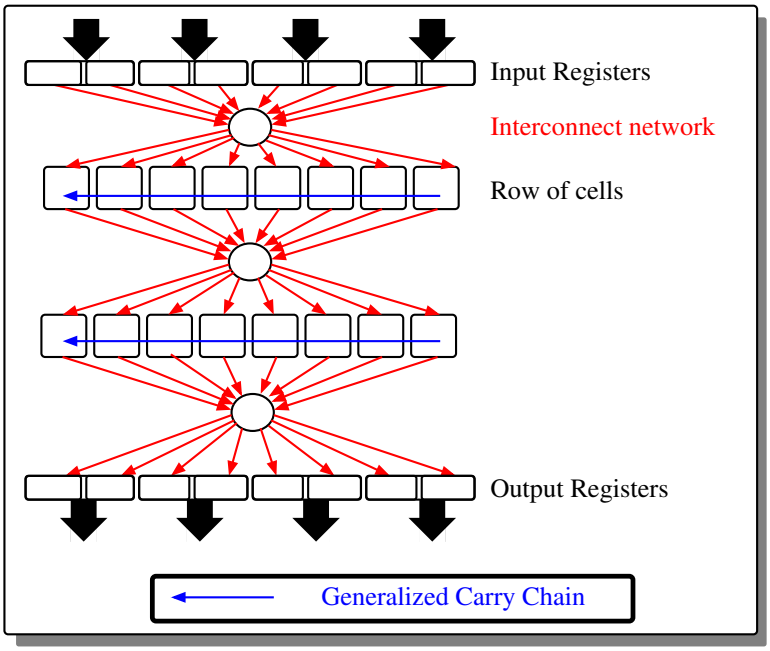


Figure 7.1: Reconfigurable matrix overview

the DIRF structure is not hierarchical. There exists a big asymmetry between connections within a row of RLCs (i.e within a pipeline stage) and between pipeline stages themselves. The DIRF is designed to implement pipelined straightforward operators, not complete system with heavy control and feed-back loops. Thus the pipeline structure, saving a full dimension of routing, makes sense.

An other specificity of the DIRF is to be medium grain. The bus size is configurable to support architectural space exploration. We mainly focus on a fixed size of 4 bits, dictated by the applications we first targeted (detailed in Chapter 10. This parameter is instantiated at every level of the architecture: it is the RLC I/O port size, it is the interconnect network bus size and it is the matrix I/O port sizes (between the first and last row and the I/O registers). The only elements which are not directly 4-bit wide are the I/O ports (between the I/O registers and the coprocessor interface) and the generalized carry chain.

Due to this architecture-exploration support, we have to distinguish two levels of configuration. The first level (often called *configuration file*) is static, it is the configuration used to generate the DIRF HDL description. This level is used for architectural exploration. Once a good static configuration is found, the DIRF can be synthesised and melted into a chip, the static configuration will no longer be modifiable. The second configuration level is dynamic. It is the configuration of the reconfigurable matrix. It can be modified dynamically during runtime and drives the basic cells functionality and the interconnect configuration. Some elements belonging to both levels are discussed in this chapter. Both the configuration file and the dynamic matrix configuration generation are addressed by Chapter 9.

The remaining of this chapter is organized as follows: Section 7.2 describes the architecture of the DIRF Reconfigurable Logic Cell, Section 7.3 describes the interconnection network between rows of cells and the generalized carry chain.

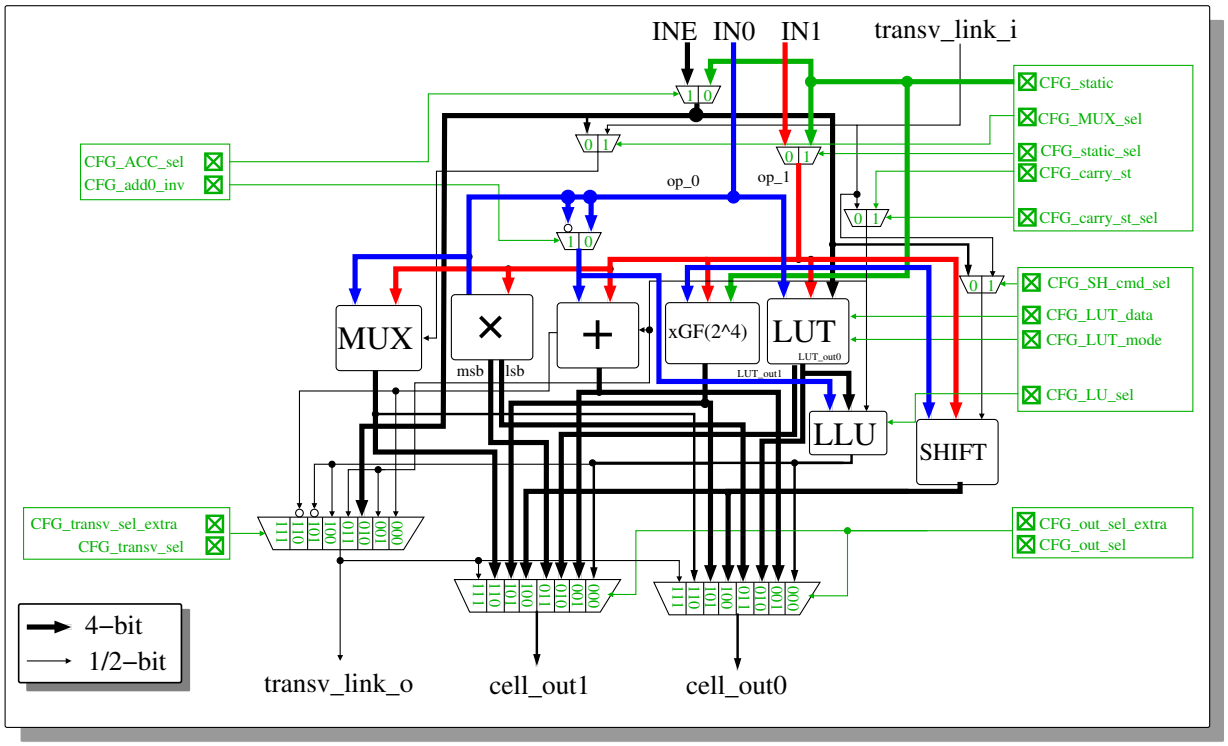


Figure 7.2: Reconfigurable Logic Cell interfaces and micro-architecture

7.2 Basic cell

The DIRF is built around a basic cell reproduced in rows and columns to build the matrix. This basic cell is described in synthesizable parametric VHDL. Part of this description is generated from a cell configuration file, which will be the focus of Section 9.1.2. This cell configuration is part of the DIRF static configuration, to support architectural exploration. This section describes the basic cell structure.

7.2.1 Cell interfaces

Figure 7.2 illustrates both the micro-architecture and interfaces of the RLC used in our current developments. As stated previously the size of the cell interfaces is the grain of the DIRF, typically 4-bit in our applications. In its current state the cell exposes 3 inputs (**INE**, **IN0** and **IN1** on Figure 7.2) and between 2 and 3 outputs (**cell_out0** [possibly duplicated] and **cell_out1** on Figure 7.2). The number of outputs is part of the cell configuration file.

7.2.2 Cell micro-architecture after a first round of design exploration

The cell micro-architecture was determined after some try-outs with our development tools and the applications we target. The cell is built in 3 levels: a middle level of arithmetic and logic macro-block surrounded by two levels of multiplexers. The first multiplexer level is designed for input selection: it provides the choice between static inputs stored locally and dynamic inputs coming from the interconnect network. The last multiplexer level is designed for output selection: it selects output of the macro-block corresponding to the functionality configured in the RLC. The

basic cell functionality is provided by the operator implemented in the macro-block level. In the version presented in Figure 7.2, the cell implements:

- a multiplexer for direct input forwarding
- a 4×4 multiplier slice for multiplication
- a 4-bit adder for addition and subtraction
- a $GF(2^4)$ finite field multiplier for cryptography
- a 64-bit LUT for unary and bitwise operation
- a logical unit for linear logic operation
- a 4-bit shifter for shift and rotation

The macro-blocks integrated in the RLC current form represent a good illustration of what type of functional capabilities can be integrated into the basic cell. A part from the multiplexer, which is required by the DIRF architecture to transmit data along the pipeline, all the other operators are optional. Let us now focus on some of these operators.

Adder

The adder is a simple 4-bit adder. Support of 2's complement subtraction is provided by of a multiplexed inverter and a carry-in signal. The inverter implements the bitwise inversion of one of the adder inputs. The carry-in signal can be connected to the previous cell of the row to propagate a carry, or to a configuration register to statically inject a carry (eg: +1 is required in the least significant bit to perform the 2's complement subtraction).

Moreover, one of the adder inputs can be connected to a configuration register which allows the simplified implementation of addition/subtraction with constants. This configuration register is also accessible to other macro blocks. The fact that this register is routed to the macro block inputs, rather than the RLC output multiplexer, save a row of cells when constants need to be generated (it also hurts RLC latency).

Multiplier

The basic cell contains a 4×4 unsigned multiplier to implement multiplication. This macro-block is not well-suited for implementation of large multiplication. Implementing a simple 32-bit multiplication requires the use of 64 basic cells for the partial product computation and about 142 cells for compression into the final 64-bit result. In total 32×32 unsigned multiplication requires about 196 cells.

We are currently investigating other solutions to provide more efficient multiplication support. We already implemented, multiplier-free, multiplication by constant using binary reduction techniques on the constant operands. Results of this study are presented in Section 9.2.1.

Look-Up Table

One of the goals of medium grain reconfigurable is to be more area efficient than fine grain FPGA. FPGA are almost exclusively LUT-based, and the medium grain tries to differentiate itself by integrating more DSP or other specified macro-blocks (eg: Galois Field multiplier). However some functionalities can not be easily implemented without LUT support.

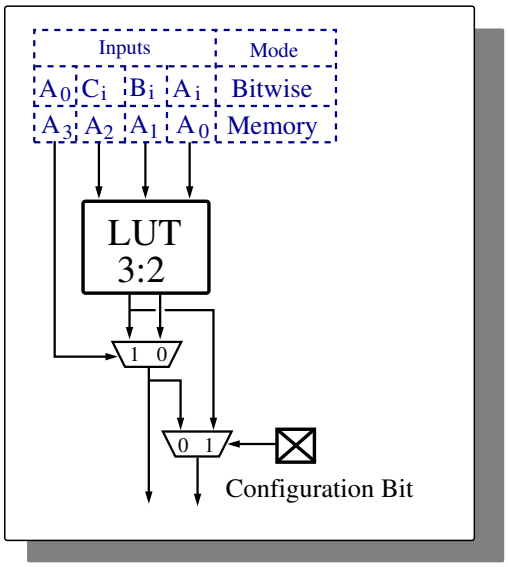


Figure 7.3: The two operation modes of the i-th 8×2 LUTs of a RLC

In its current implementation the basic cell integrate a Look-Up Table which contains four elementary 8x2-bit LUTs, one of which is presented on Figure 7.2.2. The LUT addresses can come from the three cell inputs in one of the two configurations depicted on Figure 7.2.2: **bitwise** or **memory**. The bitwise mode applies an arbitrary 3-input Boolean function to the 3 inputs in a bitwise manner. The memory mode implements an arbitrary 4-input, 4-output function of the A input. Inside the basic cell the input **A** can be connected to cell input **INE** or **IN0**.

Shifter

The shifter module performs shifts whose amount is between 0 and 3 bits. Its command can be one of the cell inputs, a static value stored in a configuration register, or be directly routed from the transverse input (generalized carry). It is combined with the interconnect network to perform static and dynamic arbitrary shifts. Section 7.4.1 will detail how a dynamic shift is performed by the DIRF.

Linear Logic/Carry Propagation Unit

Linear logic describes a set of logic operation with single bit output: every input bit is OR-ed (resp. ANd-ed ...) to compute the result. The Linear Logic/Carry Propagation Unit (LLCPU) is a two-input operator: the first input is one of the 4-bit output of the LUT, the other input is an incoming carry. It performs one of the four following linear logic operations: xor, or, and, sign. **Sign** is simply a linear xor of the ALU operands and the ALU carry-out bit. The fact that one of the input can be the carry-in and that the LLCPU output can be connected to the cell's carry-out makes it possible to implement large linear logic operator across a row of cells.

7.3 Interconnect network

Two kinds of interconnect network structures were studied: a crossbar inspired from classical FPGAs, and a permutation network. The underlying goal is to provide the maximal routing capabilities with:

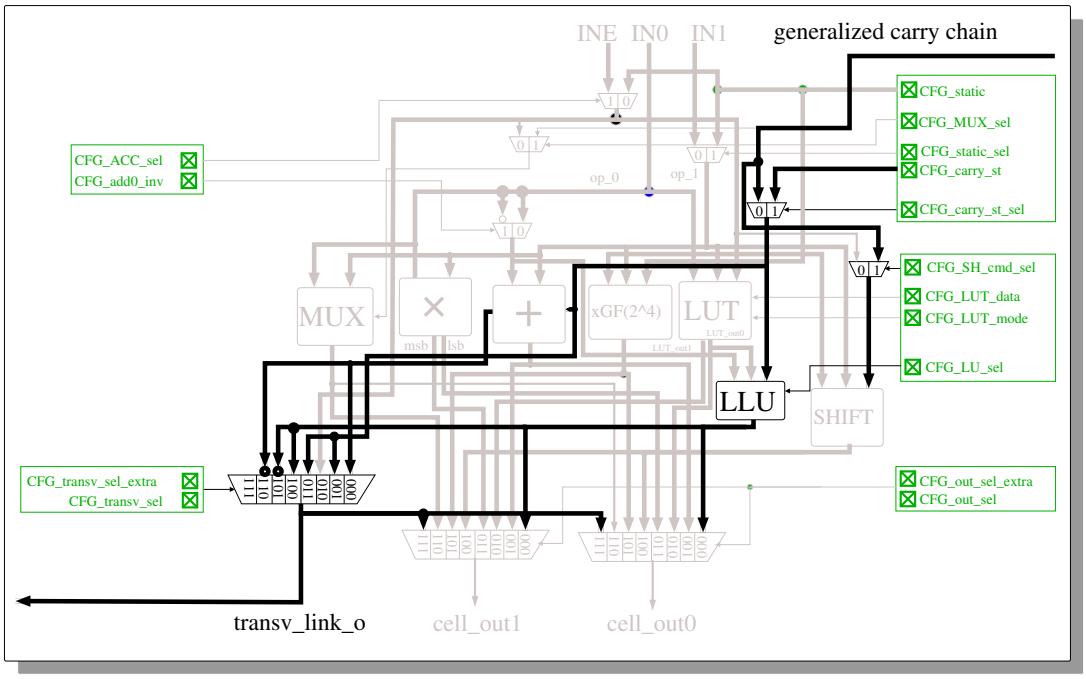


Figure 7.4: RLC generalized carry chain

- as few configuration bits as possible.
- the lowest latency

To minimize the contribution of routing configuration to the overall dynamic configuration, interconnect networks are built with 4-bit wires. The network does not have the capability to mix data within this granularity. All the fine grain bit manipulations are implemented by the RLCs.

This section is divided as follows: sub-section 7.3.1 introduces the generalized carry-chain, sub-section 7.3.2 describes the basic node we used to construct both our interconnect proposals; sub-section 7.3.3 describes a first interconnect structure based on a crossbar scheme; subSection 7.3.4 introduces permutation-based interconnect networks.

7.3.1 Generalized carry-chain

As stated in Section 6.1.3, FPGAs were enhanced with a fast carry chain to improve adder implementation. This chain can be used for more than carry propagation in linear adder designs. For example [127] uses the carry chain to implement fast compression tree (multi-operand adders). Some modifications of this carry chain structure have been suggested. For example, [128] suggested to reduce the pressure on routing resources by implementing carry chain supporting more functionalities; [68] suggests a faster carry-chain design, improving addition efficiency, in FPGAs. This subsection describes our own proposal for a generalized carry chain.

The generalized carry-chain is implemented in each row of RLCs as a 2-bit wide transverse connection. It is designed to improve the application of three operations: addition, shift and linear logic. Carry chain width, 2 bits, has been chosen to support those three operations. The shift implemented in the basic cell supports shift amount between 0 and 3. This shift amount is encoded using 2 bits. Adder carry-chain is encoded using 1 bit, as is the linear logic. Figure 7.4 highlights the structure of this generalized carry-chain, implemented in the current version of the basic cell. Inside the basic cell, the carry-chain is connected to:

- the adder as a standard carry input and output
- the linear logic unit (LU) as a single bit input and output
- the shift command, as a possible 2-bit input

The common benefit of those connections is to support the implementation of $4p$-bit operators (adder, shifter and linear logic module) using p cells, in a single row and without using the interconnect network.

7.3.2 Elementary interconnect node: β-network

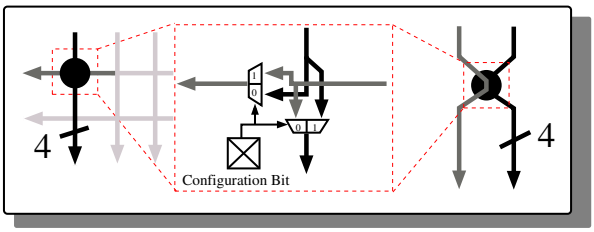


Figure 7.5: Interconnect junction point

We implemented two versions of the interconnect network, respectively based on a crossbar and on a permutation network. Both implementations use on the same elementary junction, shown on Figure 7.5. In the literature, this junction is called β-element or β-network . A junction has two inputs and two outputs plus a configuration bit. The I/O sizes are defined by the DIRF grain: 4 bits in the current experiments. The junction has two configurations:

- the **straight configuration**: input 0 is routed toward output 0 and input 1 toward output 1.
- the **cross configuration**: input 0 is routed toward output 1 and input 1 toward output 0

The β-network implementation uses two muxes with swapped inputs, with the configuration bit as their common command signal. This junction requires a single bit of configuration. It can not implement input duplication. In the current version, signal duplication can not be performed by the interconnect and is provided through basic cell output duplication capability. As the basic cell has a limited number of outputs, generating more than this number of duplicated signal requires the use of several rows of cells. One of our perspective is to study other junctions and interconnect architectures to lift those restrictions.

7.3.3 Crossbar-based architecture

The general structure of the crossbar-based interconnect is illustrated by Figure 7.6. This structure is divided between multi-segmented lines and columns. Lines are unidirectional, there exist two possibilities: lines going right and lines going left. Columns are also unidirectional but with a single possibility, implied by the DIRF pipeline structure: top to bottom. Such interconnect connects a row of cell A to another row B. Cell output from the row A corresponds to column inputs. Column outputs are connected to inputs of the row B cells.

Junctions are implemented at certain line/column crossings. The configuration of these junctions defines the interconnect routing. An example of routing configuration, in such interconnect, is presented by Figure 7.7. This configuration implements two connections from outputs of row A

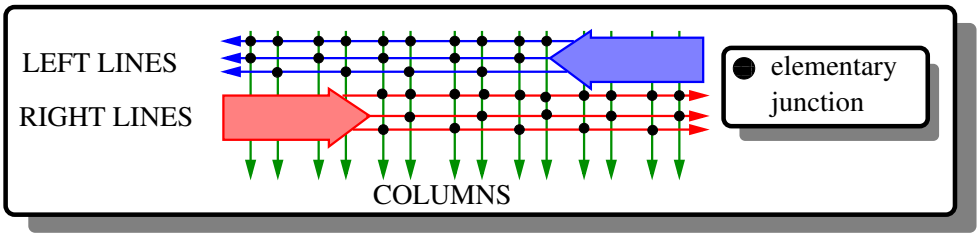


Figure 7.6: An example of crossbar-based interconnect architecture

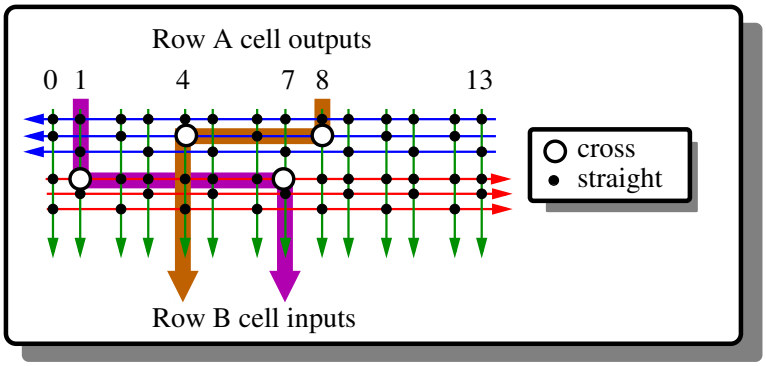


Figure 7.7: Example of crossbar-based routing configuration

to inputs of row B: 1 to 7 and 8 to 4. Each line is defined by a *step* and an *offset* parameters. The first one indicates the number of columns crossed between two junctions, and the second indicates the column index of the first junction. In practice, lines with step $= k$ are implemented in group of k lines, covering all the possible offsets. Let us illustrate this on Figure 7.6. From top to bottom, the left lines have the following configuration: offset $= (0, 0, 1)$ and step $= (1, 2, 2)$.

As part of the architectural exploration objective, the implementation of this crossbar is statically generated. The number of columns is defined by the number of cells implemented in a row. The number of lines, their respective offset and step are parameters of the DIRF static configuration. Each line has a number of configuration bits equal to the number of junctions on the line. Lines with step greater than 1 are generated in groups, each group contains one junction per column. As a result, each group of line requires one configuration bit per column. While implementing application test cases, we tried to minimize the interconnect size, that is minimizing the number of lines.

During those first experiments, we found that the crossbar-based network was always less efficient than the second interconnect option. Results, which will be presented in Section 10.3, illustrate this analysis. As a result, we focused on the second option. This network is no longer actively supported by our software framework.

7.3.4 Permutation-based networks: introduction and advantages

Benes networks are an instance of Clos networks defined as non-blocking re-arrangeable permutation network. For every permutation of the input vector, there exists at least a network configuration to obtain this permutation on the data output. Such configuration can not always be obtained by building one by one, in an arbitrary order, the input to output paths in the network. This process may result in a blocking situation, requiring the modification of paths defined ear-

lier, to complete the routing. [125] proposed a blocking free routing algorithm for shuffle based Benes network called the *looping algorithm*. Benes networks were first used in telecommunication routers. When multiprocessing systems appeared, they became useful for routing communication between processors, since they allow every permutation with n concurrent connections (towards n distinct destinations). Recently they became interesting at a lower level, as CPU functional units, for accelerating multimedia and cryptographic application. Indeed, general purpose CPU do not support efficiently the complex bit manipulations (eg: permutation, extract/deposit) required by these applications.

The purpose of this work is to study the use Benes Network as reconfigurable interconnect for designing reconfigurable hardware. Using a permutation network as interconnect can appear as an expensive solution. Indeed, reconfigurable fabrics, such as FPGA, rely on limited, but efficient, routing capabilities to limit the configuration size. We studied Benes network because they provide interesting row to row routing possibilities and because they can be extended to support operations on which other reconfigurable fabrics lack efficiency. They implement any permutation with a limited number of configuration bits. Benes networks are close to optimal to perform permutation: they can realized every single permutation of their inputs with close to the least possible amount of configuration bits. Benes network requires $\#BNC = \frac{n}{2} \times (2 \times log_2(n) - 1)$ configuration bits. For n inputs, there are $n!$ possible permutations. The configuration of a permutation network requires at least one unique configuration per permutation, thus an optimal permutation network requires $\lceil log_2(n!) \rceil$ bits of configuration. The following stands: $\#P = log_2(n!) \geq n \times log_2(\frac{n}{e})$, from which we deduce:

$$\#BNC - \#P \leq n \times log_2(e) - \frac{n}{2} = n \times (log_2(e) - \frac{1}{2}) \approx 0.94 \times n$$

Benes network can be modified to support dynamic rotations and shifts. Those operations, required by some applications (e.g. floating-point normalization) lack support in other reconfigurable architectures. We suggest to integrate support for these operations directly in the interconnect network, rather than using RLCs. Implementation of those operations will be the subject of Section 7.4.

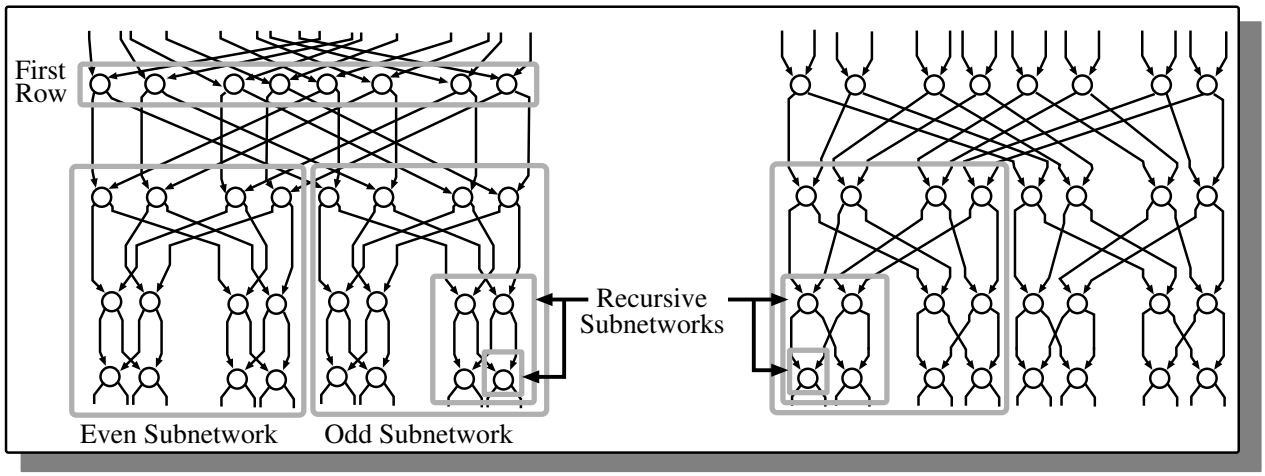


Figure 7.8: Two types of half 16-input Benes networks: **butterfly**(left) and **unshuffle**(right).

The litterature does not make any distinction between the several possible type of Benes network. We will distinguish butterfly-based and shuffle-based Benes networks. The differences between the two types of Benes network is illustrated in Figure 7.8 with 16-input networks. Let us now detail these two types.

Butterfly-based Benes network

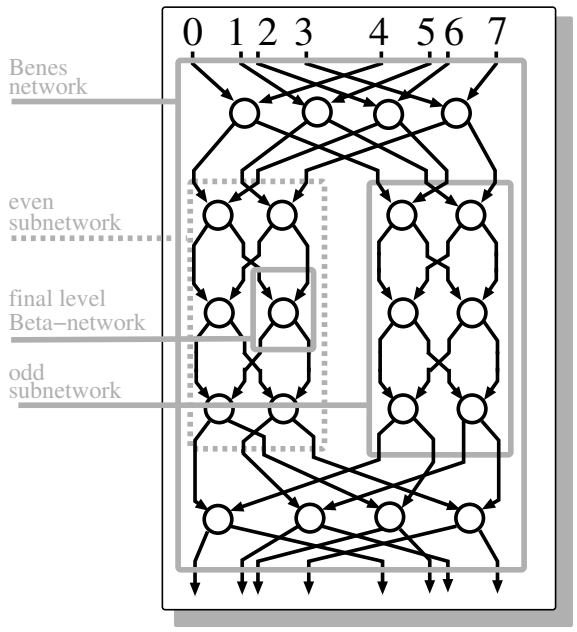


Figure 7.9: Full butterfly-based 8-input Benes network

Let us begin by describing the butterfly-based structure for Benes network. It is illustrated by Figure 7.9 which represents a butterfly with input size 8. An empty disk represents a β-network(an elementary junction). Let $n = 2^k$ be the number of inputs to this butterfly network. The first row of β-networks is connected as follow: β-network indexed i is connected to the inputs with index i and $i + \frac{n}{2}[k]$. The β-networks indexed from 0 to $\frac{n}{4} - 1$ are the inputs of the left subnetwork which is a butterfly of dimension $\frac{n}{2}$, this subnetwork will be called the **even subnetwork**. The last $\frac{n}{4}$ β-network are the inputs of the right subnetwork which is also a $\frac{n}{2}$-wide butterfly which we will call **odd subnetwork**. It is important to notice that there is no internal connection between the two subnetworks. They remain completely separated until the outputs. The rest of the structure is built recursively until reaching subnetwork of width 2 that corresponds to the basic β-network element. The complete Benes network is built with a butterfly and an inverse butterfly, assembled as illustrated by Figure 7.9. The even and odd subnetwork notation is generalized to the complete network by fusing subnetworks from the butterfly and its inverse. The last recursion level is common to both half.

Shuffle-based Benes network

The second type of Benes network is based on an bit manipulation operation called shuffle. This structure is illustrated by Figure 7.10 The basic shuffle operation is:

$$abcd\ ABCD \rightarrow aAbB\ cCdD$$

and the basic unshuffle is:

$$aAbB\ cCdD \rightarrow abcd\ ABCD$$

In the first row of this network, the input $2k$ and $2k + 1$ are the inputs of the k^{th} β-network which outputs toward the k^{th} input of the left subnetwork and the k^{th} input of the right subnetwork.

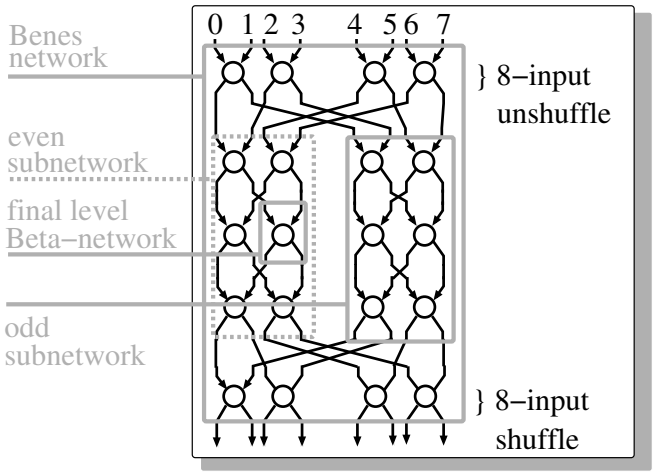


Figure 7.10: Full shuffle-based 8-input Benes network

Each subnetwork of the upper half, is a half unshuffle network with $\frac{n}{2}$ inputs. As for butterfly, the half network is built recursively, and the permutation network is build by fusing an unshuffle and an shuffle half.

We can notice that the first row of our shuffle-based Benes network is a input-by-input configurable shuffle. The β-network configuration indicates whether it will be an inner unshuffle (left input to left subnetwork) or outer unshuffle (left input to right subnetwork). As in the butterfly network, it is important to notice that there is no internal connection between the subnetworks. By construction, inputs $2k$ and $2k+1$ can not be routed to the same subnetwork, since they are transmitted through the same β-network which has one connection to each subnetwork.

7.4 Contribution to permutation-based Benes networks

Not all operations can be mapped with similar efficiency on reconfigurable fabrics. For example, bitwise functions, where uniform operations are performed on independent sets of bits, are efficiently mapped into FPGAs. This is due to the hierarchical LUT structure of FPGAs. Operations involving long linear dependency, such as additions, or complex reduction, such as multiplications, do not benefit from the LUT structure as much and have required architectural improvements: fast carry-chain and DSP blocks.

There is a family of operations which is still mapped quite inefficiently on FPGA: rotations and shifts. Those operations are not bit-level independent: one output bit must be linked towards every input bits for the correct selection to happens. However, rotations and shifts have an interesting specificites: they are connection operators performing a permutation on their inputs. Thus they fit well into permutation networks such as the Benes network. This section describes one of our contribution: using Benes network to perform dynamic shift/rotation in reconfigurable fabrics. It describes both the modification of Benes network required and the mechanism to map a dynamic shift/rotation.

7.4.1 Using the interconnect network to perform runtime-dynamic shifts and rotations

Two types of rotation/shift need to be distinguished: runtime-static and runtime-dynamic.
Runtime-static is a static operation: the shift/rotation amount is known at the dynamic configu-

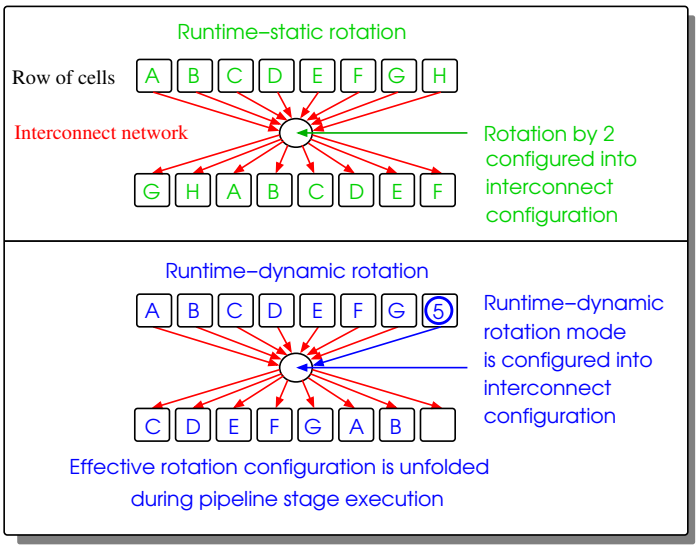


Figure 7.11: Differences between runtime-static and runtime-dynamic rotation modes

ration compile-time, that is when the program that will run on the reconfigurable is transformed into a configuration for the reconfigurable matrix. Runtime-dynamic is a dynamic operation: the shift/rotation amount is determined only during execution and depends on the reconfigurable matrix inputs. Realizing a static rotation/shift is quite easy since our Benes interconnect network is able to perform any permutation. In this *runtime static* working mode, $O(log(n) \times n)$ configuration bits are used to set any permutation of the 4-bit interconnect inputs determined at compile time. We suggest to use the Benes interconnect network to perform dynamic shift/rotation as well. This second new mode is called *runtime dynamic*, configuration bits are no longer used to configured the interconnect. The interconnect routing is changed dynamically, possibly every cycle, according to some specific inputs: the shift amount or shift command. The main differences between runtime-static and runtime-dynamic are illustrated by Figure 7.11. The routing functionality of the runtime-dynamic mode is degraded, the interconnect can only perform rotations or shifts on its inputs. More precisely the interconnect between two rows is divided into sub-areas. Each area can be configured separately, at compile-time, to one of these two modes.

Our mechanism performs a bit-level shift/rotation of s in two steps:

- the **multiple shift**: a shift/rotation of $s_{multiple} = \lfloor \frac{s}{4} \rfloor$ is performed inside the interconnect network at the level of the interconnect wire (4 bits), the bit-level shift/rotation amount is thus of $4 \times s_{multiple}$.
- the **modulo shift**: a shift/rotation of $s_{modulo} = s \mod 4$ is performed inside a row of RLCs.

Those steps are commutative and could be swapped. For the runtime-static shifts/rotations, both steps are embedded into the dynamic configuration: the first step is encoded into the interconnect configuration and the second step is encoded into the RLCs configuration. For the runtime-dynamic shifts/rotations, both steps are performed dynamically. The lowest two bits of the shift amount are sent as shift amount operands to the RLCs performing the modulo shift. The remaining bits are used to dynamically configure the interconnect so that it performs the expected multiple shift. Section 7.3.1 described the generalized carry chain used to route the modulo shift amount outside the interconnect network, directly inside the RLCs. We are now going to describe modifications made to the Benes network to support runtime-dynamic rotation/shift, on top of its usual permutation functionality.

7.4.2 Benes network and rotation: state of the art

Benes network have been studied to implement many operations including other subset of permutation like the group operation [70]. In [69], Hilewitz and Lee have shown that it is possible to dynamically configure a butterfly-based Benes network for rotation-based operations, which includes rotation, shift and more complex bit manipulations (extract, deposit, ...). Those operations are very useful for many applications, especially in the multimedia and the cryptographic domains, as they are not well supported in current General Purpose Processors (GPP). The objective of [69] was to demonstrate that permutation operations could advantageously replace the generic shifter of CPUs. Our perspective is a little different: we want to design a mechanism that allow us to use an interconnect Benes network as a runtime-dynamic shifter or rotater. We now focus on the runtime dynamic configuration (RDC) and its construction.

As we have seen, a n-input Benes network requires $\frac{n}{2} \times (2 \times log(n) - 1)$ configuration bits. This runtime-static configuration is too big to be loaded every cycle: it will not be practical to determine statically the configuration for every possible dynamic shift and select which one to load during each execution cycle. A better solution is to dynamically compute the RDC to configure the interconnect to perform a specific rotation. It can be considered as a configuration expansion, from very few bits ($\approx log(n)$) we design a small logic module that expands the full $\frac{n}{2} \times (2 \times log(n) - 1)$ configuration bits in the same cycle they are used to configure the interconnect network. This requires a very low latency and if possible small area. This is the subject of this section. It extends [69], which presented such a solution for butterfly-based Benes network, to shuffle-based networks. These networks are more generic and allow non power of 2 sized interconnect, which are useful for some of the DIRF dimension we study later on.

This rest of this section is divided as follows. Section 7.4.3 presents a simple algorithm configuring a full shuffle network for rotation. Section 7.4.4 provides a hardware design implementing this algorithm. Finally Section 7.4.6 generalizes the algorithm to shifts and odd-sized networks.

In the rest of this section:

- n is the size of the rotation input and output.
- r is the rotation to the left amount to be performed.

We will only consider (except in the last section) Benes networks which have I/O length equal to a power of 2.

7.4.3 Realizing rotation

As the Benes network is a permutation network ([125]) and rotations are a subset of permutations, a Benes network can be configured to realize a rotation. We are now going to describe a mechanism to determine the rotation configurations. We are considering the full network (an unshuffle half followed by a shuffle half). We distinguish:

- **even rotation**: rotation to the left with an even amount
- **odd rotation**: rotation to the left with an odd amount

We distinguish input β-network and output β-network : an input β-network is the first crossed by an input, before entering a subnetwork. A output β-network is the last crossed before exiting the network, and after exiting a subnetwork. A column is the position of a β-network within its row. A β-network have two inputs and two outputs: even and odd. The even (respectively odd) inputs of the input β-networks are connected to even (respectively odd) inputs of the complete network. The even (respectively odd) outputs of the output β-networks are connected to even (respectively

odd) outputs of the complete network, On an even rotation of $r = 2k$, two consecutive inputs, indexed $2t$ and $2t + 1$ enter and exit the network through the same input β-network and the same output β-network. The output β-network column corresponds to the input β-network column, rotated left by k columns. On an odd rotation $r = 2k + 1$, two consecutive inputs ($2t$ and $2t + 1$) enter through the same column but exit by different columns. $2t$ is gathered with $2t - 1 \mod n$ and $2t + 1$ is gathered in the same output β-network as $2t + 2 \mod n$.

We decompose the rotation in two steps :

- the rotation in terms of columns of β-networks
- the inner output β-network swapping which is a rotation by $0, +1$ for even indexed input and $-1, +0$ for odd indexed input.

For an odd rotation, an even indexed input will be rotated k columns to the left and then will swap its position in the output β-network to become an odd-indexed output. An odd indexed output will be rotated $k + 1$ columns to the left and then swap its position in an output β-network to became an even indexed output.

Let us consider the first β-network row configuration. For the odd rotation, two contiguous β-network must have the same configuration. Indeed, as the rotation is an odd number, the inputs $2 \times i + 1$ and $2 \times i + 2$, located in input column i and $i + 1$, exit the network through the same output β-network. If they are routed toward the same subnetwork, by construction, they can not come out in the same output β-network. Those inputs will be routed toward the same subnetwork if input column i and $i + 1$ β-networks have distinct configuration (eg: cross and straight). This demonstrates that there must a single uniform configuration for all input β-networks. We will consider that for odd or even rotation, the first β-network row is always configured straight, which is compatible with the odd rotation contiguous constraint.

Let us now consider the configuration of the last β-network row, output β-networks , located after the two subnetworks. With the first row configuration, the even indexed inputs $0, 2, 4, ...$ are routed toward the even subnetwork, and come out to the even input of the output β-networks. For an even rotation, these even inputs must exit through even outputs of the output β-networks , the configuration shall be **straight**. For an odd rotation, even inputs must exit through odd outputs of the output β-networks , the configuration shall be **cross**. this output β-network configuration, cross for odd rotation and straight for even rotation, also works for odd inputs. Thus, the last β-network row configuration is fixed by the parity of the rotation.

We shall now consider the subnetwork configuration. Let us first consider an even rotation, $r = 2k$. The i^{th} output of left subnetwork is linked to the output β-network i^{th} column. This β-network is linked to network output $2i$ and $2i + 1$. The subnetwork input j, which corresponds to the network input $2j$, must be linked to output $2j + 2k \mod n$. Thus, inside the subnetwork, this input j must be linked to the output $j + k \mod \frac{n}{2}$. This output is connected to the $j + k \mod n_{half}$ column in the network last row which corresponds to the right output β-network to reach output $2j + 2k \mod n$. Thus the left subnetwork has to realize a k-bit rotation to the left, $k = \frac{r}{2}$. The reader would easily convince himself that the same reasoning applies for odd inputs (in the even subnetwork).

For an odd rotation, the same reasoning applies for even-indexed inputs. Indeed the $r = 2k+1$ rotation is made by a rotation of k columns to the left and a internal output β-network swapping. So the even subnetwork must perform a k rotation. For odd indexed input, the odd subnetwork must realize a $k+1$ left rotation. The overall rotation is done by a $k+1$ columns rotation and a last internal swapping in the output β-network which that corresponds to a left rotation of -1, since the signal is swapped from odd to even position in the output β-network .

Thus the remaining question is: does this process finishes ? Is the inner subnetwork given a feasible rotation ? A Benes network has $\lceil log(n) \rceil$ levels of β-networks rows (or level of subnet-

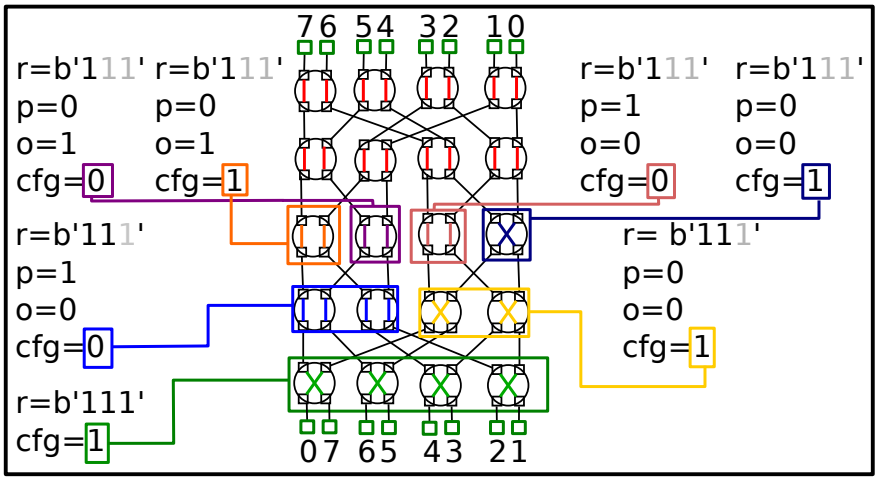


Figure 7.12: Example of configuration unfolding for rotation in shuffle-based Benes network

works). If a level rotation is r, the next level will have to compute at most a $\lceil \frac{r}{2} \rceil$ left rotation. It is easy to see that the k^{th} level (starting with $k = 0$) will have to process at most a $\lceil \frac{n}{2^k} \rceil$ rotation, which corresponds to the level size (and so is feasible).

7.4.4 Configuration unfolding

The previous section has presented a recursive process to build a rotation configuration for a shuffle based Benes network. Let us now describe the implementation of this process. As as Lee et al. suggested in [69], our objective is to offer an efficient hardware design capable of unfolding the whole configuration from the binary encoded rotation amount. We do not use the same technique as [69] because we use shuffle-based network rather than butterfly-based network. The reason to use such networks is two-folded: determine if they are more efficient than butterfly-based networks, and generalized the mechanism to odd-sized network (which we built following [125] description, using shuffle-based networks). Figure 7.12 shows an example of the mechanism described below: a rotation left by 7 on a 8-bit wide shuffle-based Benes network. Our design is built recursively, each level inputs the following signals:

- the rotation parameter r, in binary $r_m ... r_1 r_0$
- a parameter p giving the parity of the previous level
- an overflow parameter o

This design determines the configuration bits for every β-network of the network. We build that configuration recursively by determining the configuration bit for the input and output β-networks and the bits to configure the subnetwork rotation. In case of odd rotation, the odd subnetwork must perform a $\frac{r}{2} + 1$ rotation. The contribution of our design is to avoid the computation of the increment by using the o signal, so that latency is not impacted by a carry propagation chain at each level. For the overall network, r is initialized with the given rotation, p is set to zero (previously even) and o is set to 0 (no overflow).

At each level the configuration of the first row is always straight, which correponds to a configuration bit set to 0, while 1 corresponds to a crossed configuration. The configuration of the last row is given by cfg $= r[0] \oplus o$ when $p = 0$ (even) and cfg $= (\neg r[0]) \oplus o$ when $p = 1$ (odd). The rotation for each of the subnetworks is always equal to r right-shifted by one bit, which corresponds

Description	latency (ns)	cell area	ratio wrt butterfly
butterfly	5.0	1880.3	1.0
	2.5	2329.6	1.0
	2.0	5108.5	1.0
butterfly with rotation from [69]	5.0	2916.2	1.00
	2.5	3956.4	1.00
	2.0	7354.7	1.00
shuffle	5.0	1880.3	1.0
	2.5	2329.6	1.0
	2.0	5108.5	1.0
proposed shuffle with rotation	5.0	2652.1	0.91
	2.5	3927.9	0.99
	2.0	11751.5	1.60

Table 7.1: Synthesis results for shuffle-based and butterfly-bases Benes networks, with and without dynamic rotation support

to $\frac{r}{2}$. The overflow is $(o \wedge r[0])$ when $p = 0$ and $(r[0] \vee o)$ when $p = 1$. Finally left subnetwork parity is always 0 and right subnetwork parity is cfg.

In details :

- the parity p is only used for right subnetworks, it is set to 1 when the over-network realizes an odd rotation and indicates that the transmitted rotation $r_m, ..., r_1$ should be incremented by 1 since the subnetwork needs to perform a rotation of $\frac{r}{2} + 1$
- the overflow o indicates that an input overflow or parity could not be consumed by the least significant bit and has been transmitted up to this level. It is equivalent to a carry, propagated level to level rather than integrated at each level.
- the rotation parameter r is the local rotation to be realized, (that should be incremented by overflow and parity).
- cfg is the real parity bit of the current rotation, and the configuration of the level output β-networks .

Recursively each subnetwork will process either $\frac{r}{2}$ or $\frac{r}{2} + 1$ according to its parity.

7.4.5 Hardware implementation and comparison

We have synthesized our designs along with our own implementation of the design described in [69]. All designs were made with an equal level of optimization. We did the synthesis at 3 frequencies: 200, 400 and 666 MHz which correspond to not constrained, lightly-constrained and strongly constrained design, to study the constraint effect on the frequency/area. The Table 7.1 summarizes the synthesis results.

We can first notice that area for both flavours of Benes without rotation module are identical. This can be explained by the fact that a shuffle-based network can be constructed from a butterfly-based network by applying a specific permutation to the inputs and the outputs. From a synthesis point of view, these two networks are strictly equivalent. This synthesis does not perform place and route, this step may have impact this comparison. However the rotation implementation differs. Our implementation seems to be less area intensive for lightly constrained designs, but its

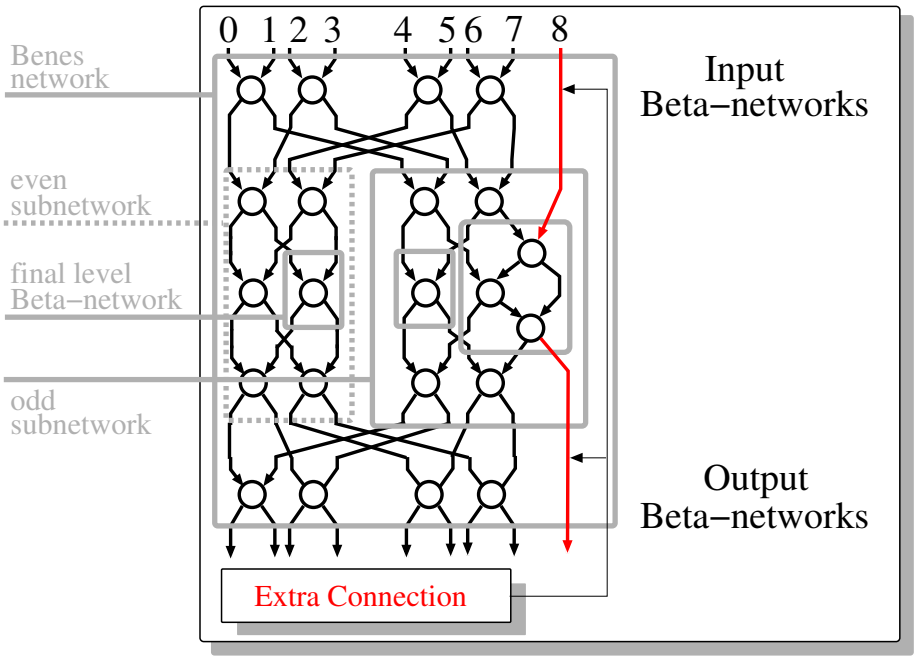


Figure 7.13: Extension of shuffle-based Benes network for odd sizes

area increases faster than the butterfly with rotation suggested in [69]. To explain this differences we consider the critical path of both designs, determined by the synthesis tool. Our implementation exposes critical paths much more constrained than [69] (14 logic levels against 8) and more cells (2912 against 2699). Those cells are smaller when less constrained. Our design is less suited for heavily constrained integration but still brings some area improvement for lower frequency architecture.

7.4.6 Generalization

In the previous sections we limited ourselves to rotation on a power of 2 data input and outputs sizes. It is quite easy to extend this support to right rotations and shifts, this is the focus of the first paragraph of this section. We contribute a mechanism to extend rotation to a shuffle-based Benes network with odd-sized I/O. The second paragraph studies the problem of unfolding rotation configuration in such networks.

Right rotation and shifts

Right rotation of r is simply performed by doing a left rotation of $n-r$. From rotations, it is easy to deduce a mechanism for shifts. As a shift by s is equivalent to a rotation by s in the same direction and a replacement of the s most significant bits (right shift) or s least significant bits (left shifts) by zeros. We will advise for the mechanism used by [69]: multiplex each network output with a zero and generate a s-bit long chain of '1' bit to configure this muxes to introduce the correct amount of zeros. Our mechanism does not provide for free this chain, contrary to [69]. So our solution will be more expensive for such operation.

Generalized Benes network

For odd-sized network, we implement the design described in [125]. A $2k+1$-size Benes network is built by gathering a k and a $k+1$ subnetwork and by directly connecting the $2k+1^{th}$ input to the odd subnetwork. An example is shown on Figure 7.13. Unfolding a rotation configuration is more complex for those odd-sized networks. In this section, we suggest a mechanism to generate those configurations.

We describe the mechanism for a rotation of r ($r < 2k+1$). Our mechanism applies a similar approach to the one developed for power of 2 sized network. The even subnetwork is going to perform a rotation of $\lfloor \frac{r}{2} \rfloor$ and the odd subnetwork (the one with an extra input) is going to perform a rotation of $\lceil \frac{r}{2} \rceil$. But contrary to power of 2 size networks, the input β-networks configuration is no longer statically set to straight. If r is even, then the first row is configured cross. If r is odd, the first row is configured straight. Finally the last row is going to be configured as follows: the first $\lfloor \frac{r}{2} \rfloor$ β-networks are configured straight and the remaining are configured cross. Let us now prove that the suggested configuration effectively performs a rotation.

Let us first justify the first row configurationThis configuration is dictated by the static extra last line added for odd size network (indexed 8 in Figure 7.13. The last input do not cross any input β-network before entering a subnetwork. The same analysis applies to the last output, which directly exits from a subnetwork. Both are connect to the odd subnetwork. the input index $2k-r$ must be connected to output index $2k$. For even rotation $2k-r$ is an even number. If the first row configuration was straight, this index would end up on the even subnetwork and could never by routed to the index $2k$ output. For odd rotation $2k-r$ is an odd index, which means that if the input β-networks were statically configured to cross, an odd index would end up in the even subnetwork thus missing the connection to the output index $2k$.

Even rotations We first consider an even rotation $r = 2p$. We considered the input even-indexed $2i$. As an even-indexed input, it ends up on the odd subnetwork which performs a rotation by p on it, moving it from index i to index $o = i + p \mod (k+1)$. It is then sent to the β-network indexed o on the last row. By construction $p < k$ since $r = 2p < 2k+1 \Rightarrow p < k + 0.5 \Rightarrow p \leq k$ as p is an integer. If $o < p$, $o = i + p - (k+1)$ and the last β-network is configured straight, which means the output index is $2o+1$ since it comes from the odd subnetwork.

$$2o + 1 = 2i + 2p - 2 \times (k+1) + 1 = 2i + 2p - (2k+1) \equiv 2i + r \mod (2k+1)$$

This proves that the input index $2i$ underwent a left rotation by r. If $o \geq p$, $o = i + p$ and the last β-network is configured cross, which means the output index is $2o$

$$2o = 2i + 2p = 2i + r, \text{ QED}$$

Let us now consider an odd index input $2i+1$, it is routed towards the even subnetwork, locally undergoes a rotation of p to the left and exits the subnetwork with index o.

$$o = i + p \mod k$$

We first consider the subcase, $o < p$

$$o = i + p - k$$

The last row β-network encountered is configured straight which implies that the output index is $2o$ (because it originates from the even subnetwork).

$$2o = 2i + 2p - 2k = 2i + 1 + 2p - (2k+1) \equiv 2i + 1 + r \mod (2k+1), \text{ QED}$$

Finally we consider the subcase $o \geq p$,

$$o = i + p$$

The last row β-network is configured cross which implies that the output index is $2o + 1$.

$$2o + 1 = 2i + 2p + 1 = 2i + 1 + r,\ QED$$

Odd rotations We now consider an odd rotation by $r = 2p + 1$. Let us first consider the input even-indexed $2i$, $i \leq \lfloor \frac{n}{2} \rfloor$. It is routed by the first β-networks row to the even subnetwork which performs a rotation by p, moving it from index i to index $o = i + p \mod k$ sending it the the β-network indexed o on the last output row. By applying the same reasoning as for even rotations we get that $o < p \Rightarrow o = i + p - k$, the last row β-network is configured straight and the output index is $2o$ since it comes from the even-subnetwork. When $o < p$, we have:

$$2o = 2i + 2p - 2k - 1 + 1 = 2i + 2p - (2k + 1) + 1 = 2i + r \mod (2k + 1),\ \text{QED}$$

We now consider the case $o \geq p$, $o = i + p$ and the last row β-network is configured cross, which means the output index is $2o + 1$.

$$2o + 1 = 2i + 2p + 1 = 2i + r,\ \text{QED}$$

This proves that the input indexed $2i$ was rotated left by r while going through the Benes network.

Let us now consider the odd-indexed input $2i + 1$. The first β-network row is configured straight, so this input is routed toward the odd subnetwork. It enters the odd subnetwork on the input indexed i and its output index is $o = i + (p + 1) \mod (k + 1)$. We consider the case $o < p$, the subnetwork output is routed toward the last row β-network indexed o which is configured straight. Thus, the Benes output index for the input indexed $2i + 1$ is $2o + 1$.

$$\begin{array}{rcrcl} o < p & \Rightarrow & o & = & i + p + 1 - (k + 1) = i + p - k \\ & \Rightarrow & 2o + 1 & = & 2i + 2p - 2k + 1 \\ & & & = & (2i + 1) + (2p + 1) - (2k + 1) \\ & & & \equiv & 2i + 1 + r \mod (2k + 1),\ \text{QED} \end{array}$$

The case $o = p$ corresponds to the input indexed k of the odd subnetwork which is in fact the highest input (indexed $2k$) of the overall Benes network. This input is routed towards a last row β-network configured cross and ends up at the Benes output indexed $2o$.

$$2o = 2p \equiv 2k + 2p + 1 \mod (2k + 1),\ \text{QED}$$

We now consider the remaining case $o > p$:

$$o > p \Rightarrow o \geq p + 1 \Rightarrow o = i + p + 1 \Rightarrow 2o = (2i + 1) + (2p + 1) = 2i + 1 + r,\ \text{QED}$$

We have shown that our configuration realizes a left rotation by r. This proofs can easily be generalized to right rotation (at least by considering a right rotation of r, as a left rotation of $2k + 1 - r$). We did not build the hardware support module for this algorithm. But let us make some remarks about the way to design such a module. Each first row needs to be configured with the parity bit of the rotation (even: cross, odd: straight) this is done by simply configuring the i-th row of β-network with the negation of the i-th bit of the rotation binary encoding. The last row configuration is more difficult to generate as for each subnetwork performing a rotation by r we have to generate a $\lfloor \frac{r}{2} \rfloor$-wide bit string of '1's. Those strings can not be obtained easily. Building the rotation module for generalized Benes network is more difficult that for power of 2 size networks.

7.5 Conclusion

The pipelined architecture we selected for our reconfigurable fabric allowed us to explore some specific row to row interconnect structures. One of them is original and could not be integrated easily into a fine-grain, hierarchical FPGA: the permutation network. Apart from their permutation capabilities, we have shown that we could extend the network architecture to support runtime-dynamic rotations and shifts, at least for power of 2 sized networks. FPGAs implementation of such operations lack efficiency, and our improvements should allow us to consider the mapping of floating-point operations into the DIRF. Indeed floating-point functionality relies on fixed-point/integer arithmetic extended with normalization operators. We are now going to consider the coprocessor architecture itself.

CHAPTER 8
Transport Triggered Co-Processor

The previous chapter has described the architecture and micro-architecture of the DIRF reconfigurable matrix. This matrix is integrated into a hardware unit called **the coprocessor** which manages configuration, input transmissions and output retrieving. This coprocessor is designed to be connected to one or multiple K1 cores, called masters. This chapter focuses on the structure of this coprocessor.

To simplify the access to the reconfigurable operator it was decided that the coprocessor will implement a transport-triggered architecture (TTA). This architecture has the advantage of exposing only a move instruction to the coprocessor master, thus simplifying the access to the reconfigurable fabric. A move operation is used to prepare the input data, and an other move operation is used to retrieve the output. These operations exist in two flavours: triggering and non triggering. The first flavour launches a pipeline step when decoded, the second one does not.

8.1 Single master coprocessor

8.1.1 Coprocessor integration

As described in Section 1.5.1, Kalray's MPPA is made of clusters of 16 K1 cores sharing access to a common memory. Within these clusters the cores are organized in groups of 4 elements which share direct access to their respective register files through a 64-bit interface. In the current MPPA version, this interface can address 4 destinations, from each core. In group, cores have only three neighbours: there is room for an extra fourth neighbour. Our objective is to design a coprocessor architecture so it can be integrated as that fourth neighbour. We studied an implemented a single-master design. It is the focus of this section. Section 8.2 will present the architecture of a multi-master design, not yet implemented.

8.1.2 Coprocessor architecture

Let us now detail the single-master coprocessor we implemented. The master is connected through a 64-bit data input, 64-bit data output and 16-bit command input to the coprocessor. The 16-bit command input is used to transmit instructions to the coprocessor. This architecture is illustrated by Figure 8.1. The configuration is partitioned into two parts: a part for the interconnect networks and one for the reconfigurable cells. Each part is a stored into its own addressable memory. Those memories are not accessible externally: to simplify control, configuration is made through two bitstream loadings.

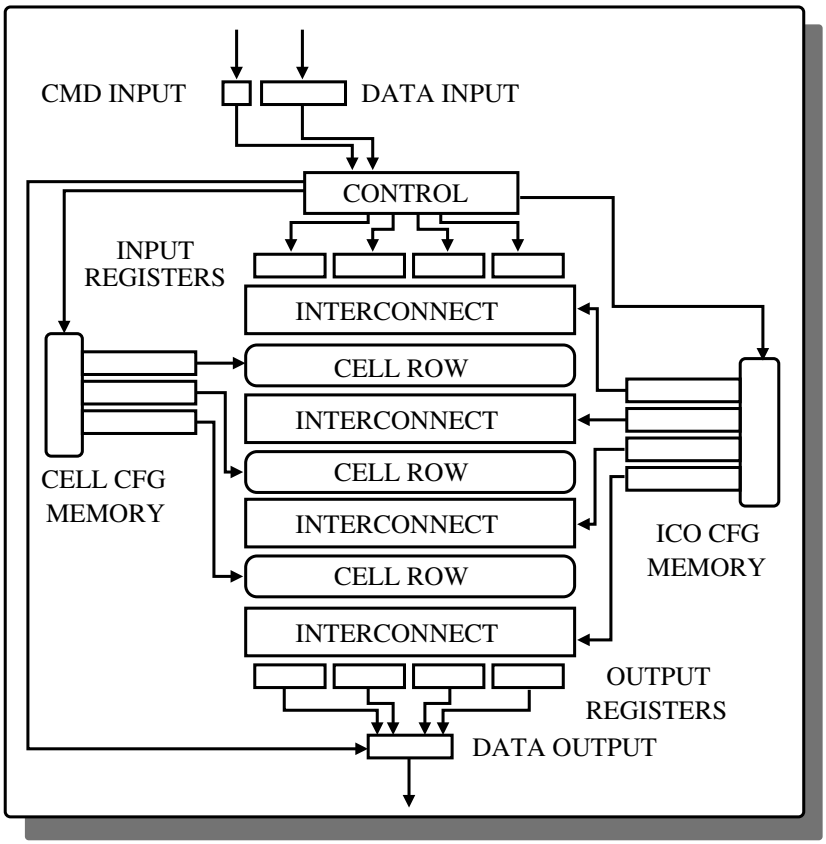


Figure 8.1: Reconfigurable coprocessor architecture

To allow architectural exploration, the coprocessor HDL description is parametrized. For example, the size and the number of input and output registers can be changed easily.

8.1.3 Instruction Set Architecture

Let us now list the supported instructions and detail their roles:

- **INIT**: initializes the coprocessor, resets the configuration automaton.
- **CFG_ICO_BEGIN**: starts the interconnection configuration, send the first 64 bits of configuration and starts the interconnect configuration automaton.
- **CFG_ICO_STREAM**: sends the next 64 bits of configuration for the interconnect network.
- **CFG_CELL_BEGIN**: starts the basic cells configuration, sends the first 64 bits of configuration and starts the cell configuration automaton.
- **READ_FLAGS**: reads the coprocessor status flags.
- **PUSH_INPUT[X]**: loads 64-bit data in input register **X**.
- **READ_OUTPUT[Y]**: read 64-bit data from output register **Y**.
- **PUSH_INPUT_TRIGGER[X]**: loads 64-bit data in input register **X** and triggers a computation cycle in the reconfigurable pipeline.
- **READ_OUTPUT_TRIGGER[Y]**: read 64-bit data from output register **Y** and triggers a computation cycle in the reconfigurable pipeline.

As the coprocessor implements a transport-triggered architecture, the pipeline is stalled until a triggering operation such as **PUSH_INPUT_TRIGGER** or **READ_OUTPUT_TRIGGER** is sent. These instructions are the triggering MOVEs which enable a pipeline stepping. **PUSH_INPUT** and **READ_OUTPUT** are the non-triggering MOVEs.

The coprocessor contains a control unit which decodes the command instruction and routes the data towards the relevant unit. It also manages the pipeline stepping. The coprocessor implements two configuration automaton: the interconnect configuration automaton (ICA) and the cell configuration automaton (CCA). The ICA manages the configuration of the interconnect. Each interconnect network has its own configuration memory. The ICA cuts and assembles each memory value from the input stream. It also stores the configuration memory index and keeps track of which is the next interconnect to be configured. The CCA performs a similar task for the cell configuration.

8.2 Coprocessor integration and shared TTA

As our experiment of Chapter 10 shows DIRF is a large block. Adding a DIRF-based functional unit to each MPPA core is too expensive. Moreover, for some applications (e.g. AES) the data interface to the DIRF is not wide enough to trigger a computation each cycle: the DIRF is underused if only one master can access it. Sharing the coprocessor among multiple cores alleviates the cost and the throughput limitation. Let us now describe our multi-master architecture.

8.2.1 Shared TTA

Transport-triggered architecture does not support more than one master. Indeed because the master has a direct control over the pipeline it can not share it with any other controller without risk of losing sight of the data it inserted into it. In Kalray's 4-core cluster each core can run independently from the others, thus making TTA synchronization impossible. Thus, it was necessary to design a mechanism for sharing the physical operator among several masters.

We suggest the following mechanism: each master is given complete control over a virtual TTA operator. A master does not have to take into account actions from other masters. The virtual operators are implemented using a single physical operator implementing the new architecture we suggest. This multi-master coprocessor architecture is illustrated by Figure 8.2. It contains the following elements:

- input buffers (one per master);
- an arbiter which manages access to the reconfigurable matrix in a round-robin fashion;
- an output FIFOs (one per master).

This architecture introduces some restriction on the DIRF configuration. The multi-master mode can only be used when fully pipelined operators, with no heterogeneous bypasses, are mapped on the DIRF. If a register is bypassed in a row, every registers of that row need to be bypassed. Every operators inputs must enter the DIRF matrix at the same cycle.

This architecture assumes each master is using the same DIRF configuration. Let us now describe in more details each element of the multi-master architecture.

8.2.2 Input buffer

Multiple masters may now access the reconfigurable matrix, but the matrix can only (at most) consume one input, produce one output and perform a pipeline step per cycle. In the multi-master context, multiple inputs or multiple pipeline triggerings from various masters could be received at the same time, therefore the masters' commands must not be directly connected to the DIRF input and pipeline management. An input buffer (IB) is inserted between the masters and the DIRF access (the arbiter). There is a configurable IB for each master. It is in charge of building the input vector out of one or multiple MOVE instructions, since The DIRF can accept inputs larger than each master interface. The size of the input vector is part of the runtime-static configuration and is loaded in the input buffer as part of the DIRF configuration bistreams. The maximal input buffer size is part of the coprocessor static configuration and dictates restrictions on the configuration that can be deployed on the DIRF.

When an input buffer contains a full input vector, it notifies the arbiter. As other input buffers may send notifications at the same time, the arbiter may not be available to transmit all ready input vectors to the DIRF. To manage such cases, the input buffer structure, illustrated by Figure 8.3, is divided between a formatting register (FR) and a ready register (RR). Each 64-bit data coming with a **PUSH_INPUT[X]** instruction is routed towards the 64-bit chunk of the formatting register addressed by **X**. The same process is performed for **PUSH_INPUT_TRIGGER[X]**. Once the FR is full, its content is transferred into the ready register and the formatting register is cleared, simultaneously the arbiter is notified.

Figure 8.4 gives an example of coprocessor execution: 4 masters are using the coprocessor, the input vector size is 4 times the master interface size and the 4-stage pipeline is mapped on the DIRF. Let us describes two key uses of the input buffer. At the end of cycle 3, master 3 has filled

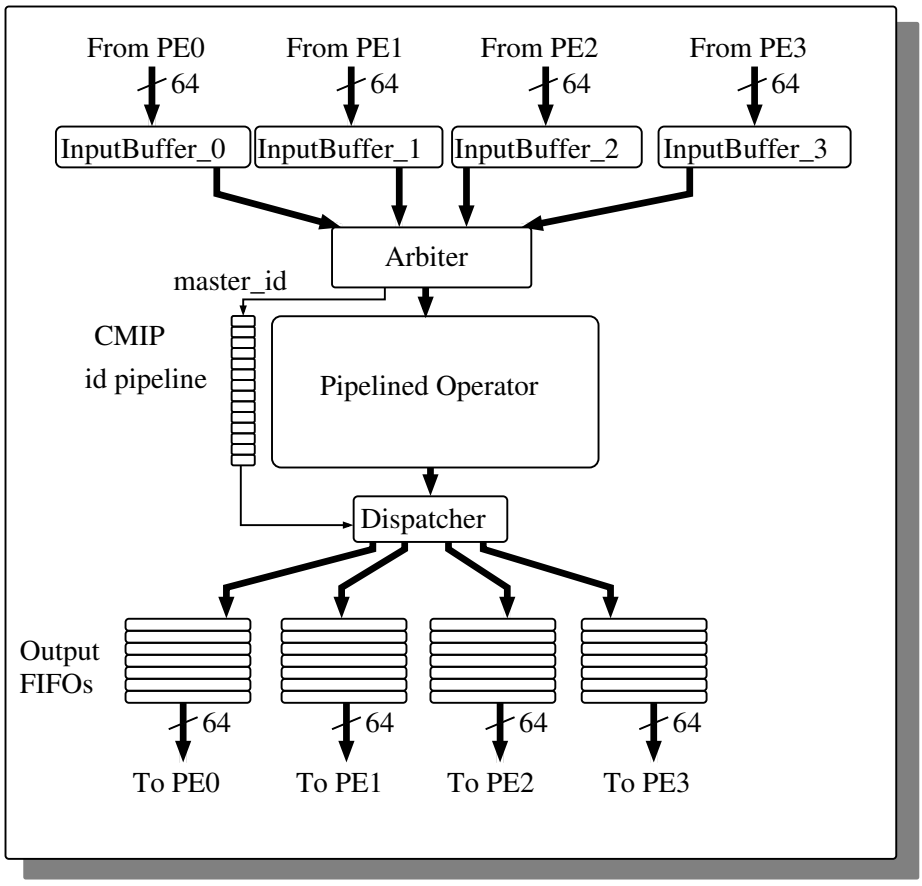


Figure 8.2: Architecture for TTA operator virtual sharing

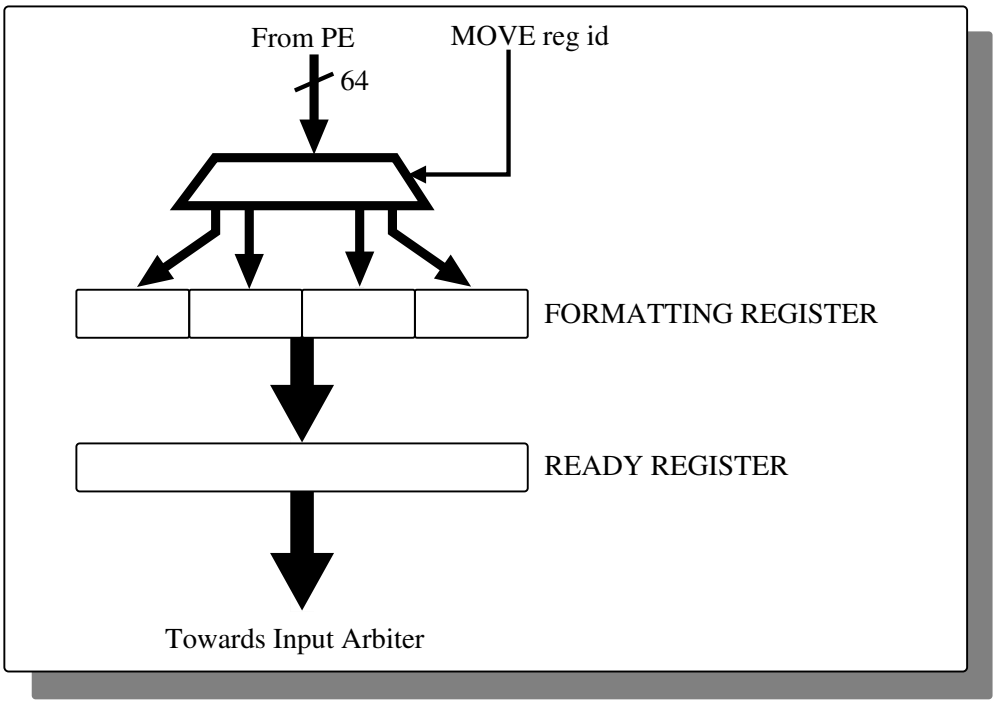


Figure 8.3: Structure of an input register

up its formatting register which is transferred into the IB ready register in cycle 4. As of cycle 5, master 3 RR is the only one that had notified the input arbiter and so the data is transferred into the DIRF first cycle. In cycle 5, the RR of master 0, 1 and 2 are ready, and 3 simultaneous notifications are sent to the input arbiter. Starting at cycle 6, the input arbiter serves those requests in order: first it transfers IB0 to the DIRF, then at cycle 7 it transfers IB1 and finally IB2 at cycle 8.

The IB provides limited input data buffering: there are some conditions to respect so that the IB does not overflow. More details are provided in Section 8.2.6.

8.2.3 Input arbiter

The Input Arbiter (IA) is in charge of transmitting input vectors from the IBs to the DIRF input. The IA also manages the DIRF pipeline stepping. Each time an input vector is ready, the IA triggers a pipeline step and move the input vector to DIRF first stage. In some way, DIRF is still transport-triggered, but rather than being triggered directly by each master, the IA acts as the single pipeline trigger.

8.2.4 Configurable master id pipeline

When an input vector is finally sent to the DIRF, the coprocessor must track the id of the master which sent it. This id is required to route the DIRF result toward the correct master. We design the Configurable Master Id Pipeline (CMIP) to track these ids.

The CMIP is a configurable pipeline. It contains master ids and is configured to mimic the DIRF pipeline: it implements as many cycle as the DIRF configuration. When an input vector is transmitted to the DIRF input, its master id is entered into the CMIP. Thanks to the CMIP configuration, the master id follows the input vector route in the pipeline, cycle by cycle. They

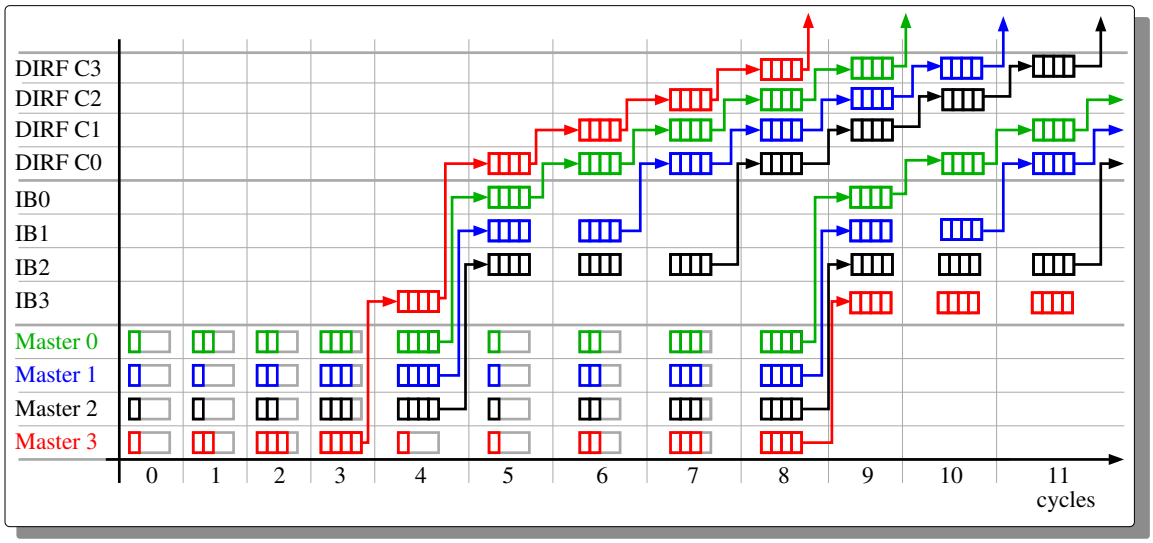


Figure 8.4: Example of multiple masters accessing the coprocessor with input buffering and arbitration towards the DIRF (master i: formatting register of the i-th master, IB: input buffer ready register, C i: DIRF i-th pipeline stage).

are output simultaneously. Finally, this id is used by the output arbiter to route the DIRF result towards the correct output FIFO, described in the next paragraph.

8.2.5 Output FIFO

In the multi-master architecture, the DIRF receives input from multiple masters, as a consequence the pipeline stepping has to be disconnected from each master command and driven by the input arbiter. For each master, it becomes impossible to predict when a data result is going to exit the pipeline. We will show in the next section that the maximal time required by the coprocessor to output a result can be predicted. However, because other masters trigger pipeline steps, through the input arbiter, the result may be output at an earlier, unpredictable, time. We inserted output FIFOs in our design, to receive and store the result while waiting for the master to retrieve them. Those FIFOs can not be bypassed, the master can not retrieve a result directly from the DIRF, it has to request it from its output FIFO. The input/output ordering provided by FIFOs is required so the master can retrieve results in the expected order. Those FIFOs are configurable: the size of each entry is part of the DIRF dynamic configuration since an entry should be able to store a complete output of the DIRF. Entry maximal size is part of the DIRF static configuration.

Figure 8.5 illustrates the function of output FIFOs. In this simplified example masters are assuming a 4-stage DIRF pipeline: they expect a result after 4 triggering operations following the first input vector transmission. At the end of cycle 3 when a computation originating from master 3 is finished, it is not directly sent to master 3 but moved to its output FIFOs. Master 3 is not expecting this data at cycle 3, it has sent only one of the five triggering operations required to push the result out of the 4-stage DIRF At cycle 9, master 3 sends its fifth triggering operations, and thus retrieves the first result. As pipeline stepping has been triggered indirectly by other masters, the result data is no longer in the DIRF. It has been stored into the master 3 output FIFOs, and is made available to this master as if output by the DIRF.

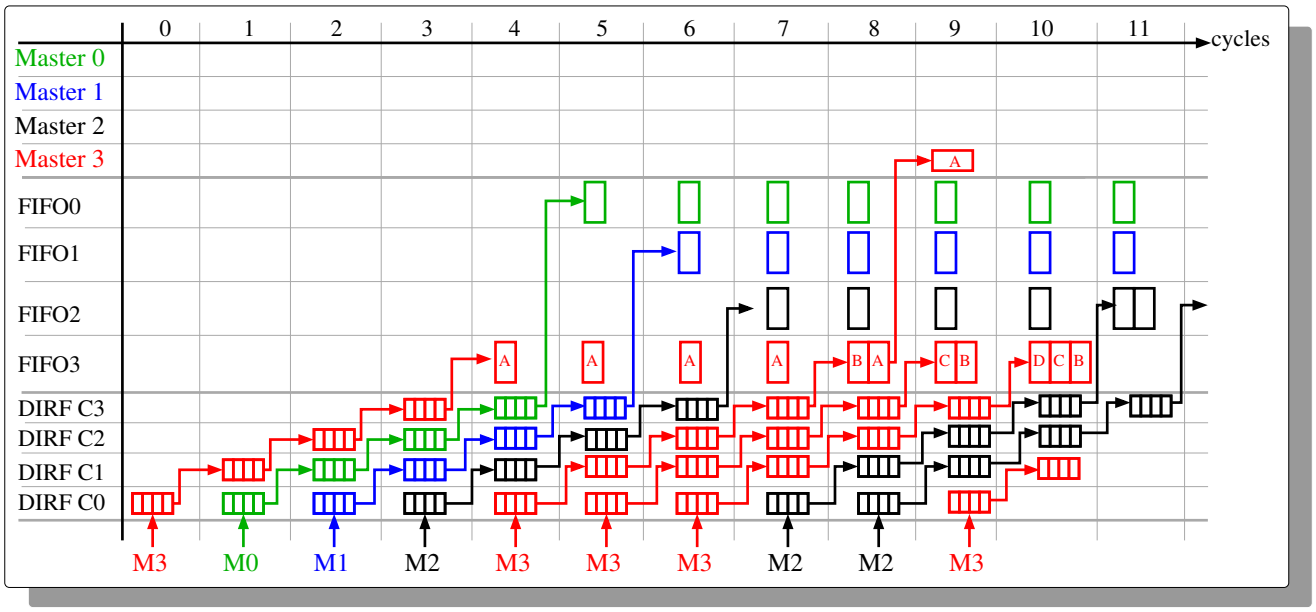


Figure 8.5: Example of multiple masters accessing the coprocessor with output FIFO buffering and shared pipeline steps (M i: data originating from Master i).

8.2.6 System sizing

The multi-master architecture does not support every execution cases: a restriction on input rate has to be enforced to guarantee functioning. This section describes both this condition and the sizing of IB and output FIFOs to ensure the system does not overflow.

Let assume a system with n masters, each of them is interconnected with an m-wide data interface (I/O) to the coprocessor. The DIRF is configured to a p-stage pipeline, expecting a o-wide input and returning a o-wide output. This constitutes a simplified symmetric cases. We assume o is a multiple of m: $o = k \times m$, thus $m \leq o$. First, let us focus on the input buffer size. If and only if $n.m > o$ then the DIRF is not able to consume the input vector fast enough. Each cycle the coprocessor receives more data than the DIRF can accept, there exists no scheduling policy that the input arbiter can apply to accommodate this input rate. On the contrary if $n.m \leq o$ the system is able to manage the input rate. Let us demonstrate this by considering the worst case: each master sends a data each cycle and all masters are synchronized, they start to send data at the same time (cycle 0). They require $\frac{o}{m}$ cycles to fill their formatting register and thus their first input vector is ready at the end of cycle $\frac{o}{m} - 1$. This first input is transferred into the ready register so the formatting register can be fed again. The last input to be consumed by the reconfigurable matrix stays in the input buffer for n cycles. During those n cycles, the master sends at most $n \times m$ bits of data. We assumed $n \times m \leq o$, so the formatting register does not overflow. In the worst case scenario, the formatting register is filled one cycle before the input arbiter empties the ready register. During the next cycle, the ready register is transferred to the DIRF input, the formatting register content is transferred into the newly empty ready register, and the formatting register is available to receive the first chunk of the new input vector. By authorizing only configurations which respect $n \times m \leq o$, we ensure that no input buffer will overflow.

Let us now consider the sizing of output FIFOs. To size the FIFOs, we first have to determine the virtual latency l the coprocessor exposes to each master. To determine this latency we start by the input buffer. In the worst case scenario, an input vector is going to stay n cycles in the IB

ready register before being sent to the DIRF by the input arbiter. Thanks to the restriction on input rate, $n \leq k$, which means that the waiting time in the IB is lower that the triggering rate of the master, if we assume the master only triggers the pipeline once its input vector is full. The master can assume that the IB only requires one extra triggering to be crossed. The DIRF can contain p intermediary state, one in each of its pipeline stage. Thus $p+1$ triggerings are required before the DIRF output the corresponding result. Overall the master should assume a $p+2$ depth for the coprocessor. This depth drive the size of the output FIFOs: in the worst case one master, M, is going to fill the DIRF pipeline and its ready register with $p+2$ input vectors and then stop for a while. In the meantime, the other masters can send a sufficient amount of triggering operations to push each of M results out of the pipeline (including the RR stage). Thus the required depth for the output FIFOs is $p+3$. One extra stage is added so that the master has time to retrieve the result, even once the extra $p+3$-th input vector, which has been inserted to push the result, is brought to the DIRF exit by other master triggers.

If we consider a system with:

- n masters
- p_{max} the maximal depth of a DIRF configuration
- o_{max} the maximal width of a DIRF configuration interfaces.

then the implementation cost of the transport-triggered sharing can be evaluated to:

- n FIFOs of depth $p_{max}+3$ and entry-size o_{max}
- $2n$ o_{max} registers used for input buffering
- $log(n)$-wide p-deep configurable pipeline
- some arbitration logic

8.3 Conclusion

In this chapter, we described a single-master coprocessor architecture, implemented to connect the DIRF to a master. We introduced a mechanism to share a transport-triggered operator. Such sharing requires the insertion of buffering, on both inputs and outputs. It also implies a restriction on the input rate, to ensure functioning. We finally studied the sizing of the output buffering to ensure no data will be lost in the transfer.

CHAPTER 9
Reconfigurable Kernel Toolchain

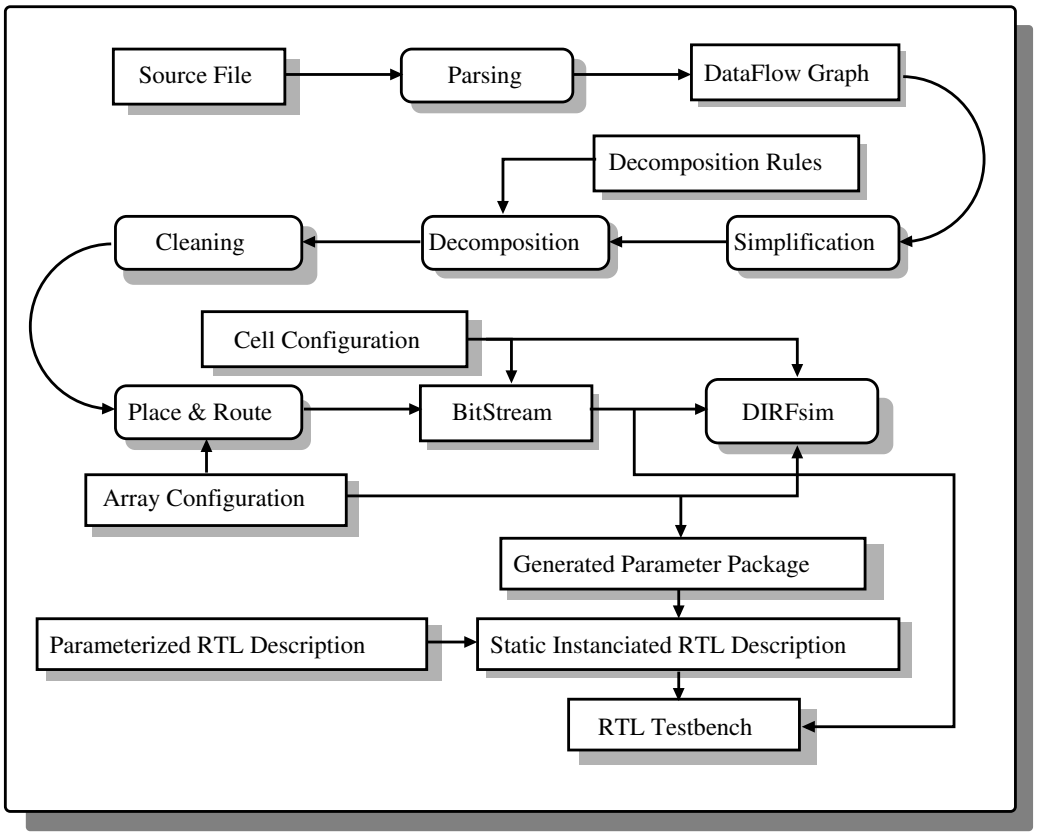


Figure 9.1: Steps of the reconfigurable kernel toolchain

The DIRF reconfigurable matrix presented in Chapter 7 is a complex operator, containing a lot of diverse resources (RLS, interconnect networks, carry chain). The practical static configurations, used in our experiments, contains about 1000 RLCs and between 10 and 15 pipeline stages. It makes building a configuration for the DIRF a complex task. A software toolchain, implementing a compiler, is a good solution to support this task. The compiler is used to automatically generate a configuration from a source program. It takes care of mapping the operator on the available resources and generates the required routing configuration. Developing such a software toolchain

represents a time consuming effort: reconfigurable fabrics are often too different from CPU to be easily targeted by compilers such as GCC or LLVM. Many small reconfigurable operator projects lacked resources to provide a toolchain and had to restrain to manual development (eg: [39]). Some larger projects, such as PicoGA [112], were able to develop a toolchain. In our case, the need for a toolchain was also justified by architectural exploration. We intend to explore several architectural choices for the DIRF, a toolchain that can be parametrized with the architecture makes the design space exploration much easier.

Thus we designed a toolchain, built around a parametrized RTL description of the DIRF, the Reconfigurable Kernel Compiler (RKC) and DIRFsim. The parametrized description is an implementation of the DIRF written in VHDL with easy to modify parameters. It allows to synthesize DIRF architectures and evaluate design choices. The RTL description is addressed by Section 9.1. RKC is a proof of concept compiler which builds a configuration for the DIRF from a source program, sometimes called kernel. Sections 9.2 and 9.3 focuses on RKC. DIRFsim is a software simulator which emulates the behavior of the DIRF single-master coprocessor. It inputs configurations produced by RKC. DIRFsim is the subject of Section 9.4. The components of our toolchain and their dependencies are illustrated by Figure 9.1.

9.1 DIRF RTL generation and architectural exploration

The DIRF RTL description is partially generated using two static configuration files. The first file, described by Section 9.1.1, configures the DIRF matrix dimensions. It has been designed to offer architectural exploration capabilities. The second file, described by Section 9.1.2, configures the RLC configuration mapping. In its current implementation it does not provides design choice capabilities. Both files are used by several other components of the toolchain.

9.1.1 Reconfigurable matrix parametrization

The parametrization of the DIRF reconfigurable matrix is provided through a file called **Array Configuration** in Figure 9.1.

```
interfaces:
  inputs:
    - size: 64
      number: 4
  outputs:
    - size: 64
      number: 4
cellArray:
  rowNum: 14
  cellByRow: 192
  colByCell: 3
  cells:
    grain: 4
    inputs: 3
    outputs: 3
    pin_mapping:
        inputs:
            None: [0, 1, 2]
            Pin_0: [0]
            Pin_1: [1]
            Pin_2: [2]
        outputs:
```

```
23              None: [0,1, 2]
24              Pin_0: [0]
25              Pin_1: [1]
26              Pin_2: [2]
27 network:
28   timingModel:
29     node: 10
30     jump: "lambda size: size * 1.0"
31   colNumByCell: 3
32   block_size: 4 # number of cell by dynamic block
33   type: benes # standard | benes
34   subtype: complete
```

Listing 9.1: Example of array configuration file used for an AES kernel

Listing 9.1 gives an example: the array configuration file used for the AES application described in Section 10.1. This array configuration file is divided in several sections:

- the **interfaces** section configures the number and size of the DIRF input and output registers
- the **cellArray** section configures the RLC matrix dimension, and RLC I/O. For example, the number of rows is configured by **rowNum** and the number of cell in each row by **cellByRow**. The subsection **cells** configures the basic cell interfaces and grain.
- the section **network** configures the interconnect network.

As illustrated by Figure 9.1, the array configuration file is used to generate a RTL parameter package which provides the corresponding DIRF HDL description.

9.1.2 Basic cell parametrization

```
1     config:
2        CFG_ACC_sel: 0
3        CFG_LUT_in_sel: 5
4        CFG_add0_inv:  1
5        CFG_carry_st: 3
6        CFG_LUT_data: [73,10]
7        (...)
8     lut_modes:
9         dataLength: 64
10        memory:
11            mapping: "lambda truthTable, i: \
12                truthTable[digitTuple(i % 16, 4)][i / 16]"
13            reverse: "lambda inNum: (lambda key, data:  \
14                getDataBit(data, key + 3* 16) << 3 | \
15                getDataBit(data, key+ 2* 16) << 2 | \
16                getDataBit(data, key + 16) << 1 | \
17                getDataBit(data, key)) "
18            keygen: "lambda arg: (0,2**4)"
19        bitwise:
20            (...)
21    parameter:
22        length: 86
23    interfaces:
24        input: 3
25        output: 2
```

```
26    suboperation:
27        lut_4x4:
28            constant_set:
29                CFG_out_sel: "10"
30                CFG_LUT_mode: "01"
31            specific:
32                CFG_LUT_data: "lambda self, subnode: \
33                    self.getMemoryLutValue(subnode.id)"
34            operation: "lambda self, ops, carryIn: \
35                (self.LUT_Memory(ops[0]), 0)"
36            simConf: "lambda self, data: \
37                self.extractMemoryLutConfig(data)"
38        constant:
39            (...)
```

Listing 9.2: Example of basic cell configuration file

The basic cell has its own configuration file, called **Cell Configuration** in Figure 9.1. This configuration file does not offer design choice modifications. It contains the mapping between configuration bits and configuration signal (presented by Figure 7.2, p.131). It is used by:

- the HDL generator to generate the cell configuration mapping;
- the the compiler to generate RLC configurations;
- the simulator to decode DIRF configuration.

It must be manually modified each time a modification is made to the cell structure. Listing 9.2 shows an example of basic cell configuration file. The first section **config** contains a mapping between configuration signal and a bit index (or range) in the RLC configuration bitstream. For example **CFG_acc_sel**, which is used to select one of the hard-macro operands, is driven by the first bit (index 0) of the configuration bitstream. The second section **lut_modes** configures the look-up table modes of the RLC: it dictates how to generate the LUT bitstream according to the LUT configuration chosen (memory or bitwise, single or multiple output, 1,2 or more inputs). The **parameter** section lists the total length of the configuration. The **interfaces** section lists the configuration of the cell interfaces. Finally the **suboperation** section lists the mapping between operations and cell configuration. This section contains one subsection per operation. This subsection contains the bitstream configuration to implement the operation (both static and data dependant configuration). It also contains a behavioral description of the operation used by the DIRF simulator described in Section 9.4.

9.2 RKC front-end

Let us now focus on RKC. RKC is a compiler, it inputs source code and generate a configuration for the DIRF. RKC's input program can be written in one of two dialects: one C-like and one VHDL-like. They will be described in more details in Section 9.2.1. RKC outputs a bitstream which configures both the basic cells and the interconnect networks of the DIRF to perform the operation described in the input source program. RKC can also wrap the bitstream in command instructions to generate a binary program executable by the coprocessor.

The main steps of RKC's process are listed on Figure 9.1. Our compiler offers several points of parametrization:

- **array configuration file**
- **cell configuration file**

- **decomposition rules**

The first two are commons with the other components of the toolchain and were introduced in Section 9.1. Decomposition rules are specific to the compiler. They are used during the technology mapping step.

9.2.1 Intermediate representation

The central element manipulated by RKC is an intermediate representation (IR). This IR is build from the source code by a front-end. There is one separate frontend for each of the dialects accepted by RKC. Once created, the IR can then be manipulated by internal optimization passes, Finally it is translated into a configuration for the DIRF.

Frontend: IR building

```
1         port(
2             x : in std_logic_vector(7 downto 0);
3             r : out std_logic_vector(7 downto 0);
4         );
5         architecture rtl of aes_module is
6         begin
7                         --bmvm is a bit matrix vector multiplication,
8             -- (one of RKC's builtins)
9             r <= bmvm(0x854a22580248a14, x);
10        end;
```

Listing 9.3: Example of VHDL-like source file for RKC

```
1         in int32 a, b, c, d, e, k, w;
2         out int32 r_a, r_b, r_c, r_d, r_e;
3         {
4             // <] is a rotation operator
5             r_a = (a <] 5) + ((b & c) ^ ((!b) & d)) + e + k + w;
6             r_e = d;
7             r_d = c;
8             r_c = b <] 30;
9             r_b = a;
10        }
```

Listing 9.4: Example of C-like source file for RKC

RKC input is a source file written in one of two dialects:

- a very small subset of C, containing a subpart of C expression grammar
- a very small subset of VHDL, containing part of the combinatorial part of VHDL (include generate for loop).

Both those sub-languages are in single assignment form (a variable can only be set once). They have been extended with builtins to provide access to specific functionalities of the DIRF (eg: delay register insertion or cryptographic primitives). RKC entry point is a lexer/parser which translates the source file into RKC's intermediate representation (IR). This IR is an annotated DataFlow Graphs (DFG) containing some, very limited control flow element, such as static loops. Every elementary operation inferred during the source file parsing is translated into a DFG node. Edges between those nodes indicates dependencies. RKC performs two types of transformations on the IR:

- preparation: as the DIRF is limited in functionality, some changes must be applied to the IR so it can be effectively mapped onto the DIRF. Those passes are mandatory, the mapping will fail if they are not performed.
- optimization: those passes aim at reducing the number of operations required to perform the functionality described in the source code so it can be mapped more efficiently onto the DIRF. Those passes are optional.

Preparation passes

CFG unrolling The first preparation pass performed by RKC is the CFG unrolling. As the DIRF does not support loop back or conditional execution, the static loops contained in the CFG must be unrolled. This first RKC pass unfolds all the loops and returns a fully unrolled DFG.

Shift and rotation splitting The second preparation pass performs a split of shift and rotation operations. As described in Section 7.2.2 and 7.4.1 the DIRF support shifts and rotations through a 2-step mechanism: the basic cell performs the operation modulo the bus size and the interconnect network realizes the operation multiple of the bus size. RKC divides each rotation/shift operations into a modulo bus size operation and a multiple bus size operation so it can be mapped into the relevant DIRF element during the technology mapping step.

Optimization passes

Once the DFG has been prepared, RKC tries to optimize it. RKC structure has been designed to provide easy integration of optimization passes. The DFG manipulation is divided into several steps. Each step can be extended with new optimization passes. Those passes have access to the IR and can manipulate it to make it more efficient (eg: by reducing the overall latency or the number of RLCs required to map the design). While working on our target application we developed two passes, a LUT building pass and a multiplication by constant pass.

LUT building and gathering DIRF's RLCs contain a bitwise LUT which can be used to implement any 2 or 3-input logic operation. Thus every elementary logic operation (eg: and, or, ...) could be mapped into a separate LUT. This is the default behavior of RKC. This solution works but lack efficiency. Multiple inputs / multiple elementary logical operations can be mapped efficiently into a single multiple-output LUT.

RKC provides a LUT building optimization pass. This pass applies a modified version of Cong and Ding 's algorithm [33] to gather a cone of logical operations into a single-output LUT operation.

After this pass, a second pass performs LUT gathering operations: it tries to fit into a single multiple-input/multiple-output LUT several single-output LUTs that shares inputs. It optimizes the DFG by reducing the number of LUTs (and thus RLCs) required to implement it.

Multiplication by a constant Dynamic multiplication is expensive to support. We are studying solutions to avoid using the hard-macro multiplier integrated into our current basic cell design (described in Section 7.2.2). Eventually, our goal is to replace it with a more effective solution, that has yet to be determined.

However most of the algorithm we consider as application expose multiplications by constants rather than dynamic multiplication. Multiplication by constant is easier to support than a multiplication between two dynamic inputs because a lot of pre-processing can be performed on the

description	# cells	max width	# levels	DIRF area
shift-and-add	400	63	13	819
Booth recoding	191	33	8	264
shift-and-add + patterns	177	25	9	225
Booth recoding + patterns	155	26	7	182

Table 9.1: Synthesis result comparison various multiplication by constant algorithms implemented in RKC

constant to simplify the graph of operations required to perform the multiplication. For example, considering a size n multiplier, a straightforward Booth recoding ([51]) can be used to expand the multiplication into a shift-and-add sequence. This implementation requires fewer than $\frac{n}{2}$ elementary operations. Indeed a multiplication by a n-bit wide constant can be decomposed into less than n shift-and-add operations (one operation by non-zero digit). Modified-Booth recoding ensures that at least one of two consecutive digits is a zero. We implemented a pattern recognition/sharing algorithm derived from Lefèvre algorithm [92] and [18]. This algorithm looks for patterns appearing multiple times in the operand bit string. It uses those to factorize the multiplication computation.

We implemented and evaluated several variants of those techniques for multiplication by constant expansion into RKC: Those algorithms are:

- **shift-and-add**: basic decomposition in shift and add of the binary coding of the constant
- **Booth recoding**: prior to shift-and-add decomposition, the constant is encoded using Booth algorithm to decrease the number of non-zero digit
- **patterns**: a pattern recognition search and factorization can be added to both previous algorithms

The reconfigurable fabric presents several specificites:

- shift of a multiple of bus width can be considered as free (because they are performed in the interconnect and do not occupy RLCs).
- shift, non multiple of bus width, are expensive since they require a full row of cells to be performed

We extended the pattern-based algorithm to exploit those specificities. When encountering several recurring patterns providing similar reductions, the algorithms favor the pattern which could be implemented with multiples of bus width shifts.

```
1       in int32 x;
2       out int32 s;
3       {
4               s = 0xbe79 * x;
5       }
```

Listing 9.5: Example of program containing a multiplication by a constant

We evaluated those algorithms by implementing them in RKC and compiling the program reproduced in Listing 9.5. It is multiplication of a 32-bit input by a 16-bit constant into a 32-bit result. The Table 9.1 presents the result of this evaluation. The results show:

- the max width of a cell level and the number of levels: these determines the minimum required size of the DIRF (DIRF area).

- the number of cells used by the application. This is a good metric for the reconfigurable array occupancy.

The pattern search improves both algorithm result, on DIRF size and DIRF occupancy. Booth recoding provides better results than shift-and-add. Pattern search has a bigger impact on shift-and-add than on Booth-recoding. On DIRF occupancy, pattern search diminishes the gap from a 109% difference to 14%. On DIRF area, it diminishes the difference from 210 % to 24%.

9.2.2 Decomposition: technology mapping

Once the DFG has been prepared and optimized, RKC performs the technology mapping: it translates abstract operation into **subnode operations**. Abstract operation are the node of the DFG, for example a 32-bit addition. A **subnode operation** is an operation that can be performed by a a single RLC, for example a 4-bit addition with carry input and output.

During this process, RKC uses a decomposition rule table (DRT). A decomposition rule describes how to generate the required number of subnodes to implement an abstract operation. When performing the decomposition, RKC searches into the DRT for an implementation of the corresponding operation. If one is found, RKC applies the decomposition rule to this DFG node and builds the mapped representation of the DFG. If none is found, it means that RKC does know how to implement the operation and it raises an error.

The decomposition rules describe both the generation and the connection of the subnodes required to implement abstract operations. For example when implementing a $4n$-bit addition, it generates a first subnode which performs a 4-bit addition without carry and then $n-2$ subnode for addition with carry input and carry output and then a final subnode with only a carry input. It connects sequentially the carry chain so the addition can be performed across the n RLCs.

9.3 RKC Backend: place and route

DFG decomposition has mapped the DFG into connected **subnode operations**. Each subnode operation can be carried by a DIRF RLC. Those RLCs must be placed and the connections routed. During this place and route step, a position on the DIRF matrix is determined for each subnode and the configuration of the interconnect networks is computed.

The place and route problem (P&R) is often a challenge for hardware synthesis tools. During the technology mapping step, the synthesis tool has generated a large number of cells, typically between several hundreds for the DIRF to several millions for an FPGA (even a few billion for non reconfigurable MPSoC such as the MPPA). The P&R tool needs to position each of them on a predefined reconfigurable matrix, while ensuring that routing can be carried out successful. P&R can be seen as a combinatorial optimization problem. There is a finite number of gates to be placed on a finite number of positions while ensuring that routing can be performed over the finite number of available connections. Connection is often the limiting factor. Indeed if you have enough position for each gate, it is quite easy to arbitrary allocate a position to a gate. However making sure that those positions allow for every connection to be made is more complicated since long connections are available in very limited number (because of the cost).

On a large FPGA, an application can now contain up to several million RLCs with even more connections. This size makes finding an optimal solution too computation intensive to be realised in practice.Heuristic approaches, such as simulated annealing, have been developed [150]. They are used on top of a hierarchical division called floor planning: the sea of gates is first divided into clusters corresponding to functionalities (and closely connected RLCs), then RLCs are placed and routed within each cluster, finally each cluster are placed and connected together [146]. Those

heuristics provide good enough solution in good enough time. They are used in industrial system such as Altera and Xilinx synthesis tools, or in academic tools like VPR [16] for P&R over FPGAs.

The P&R problem on the DIRF is similar to the one on FPGA, but at a much smaller scale. Because the DIRF is intended to implement functional operators (and not complete systems) and because of its medium grain, there are much fewer unitary gates to consider during P&R. The DIRF P&R problem, expressed as a series of combinatorial optimization problem, can be expressed and solved optimally, using integer linear programming. The practical solving is performed by ILP solvers such GPLK [102], COINOR [97] or CPLEX [36]. We are now going to describe two placement algorithm developed for the DIRF. They are two possible versions of the combinatorial optimization problem, to express the placement over the whole matrix.

9.3.1 Single level placement

Our first placement algorithm is a row-by-row positioning division. Once the positions of the DIRF inputs are determined, the position of the first row of cells is expressed as a linear problem and solved. With the solution of the first row, a problem expressing the positioning of the second row is built and solved. This process is repeated until the outputs are placed.

There is one ILP problem per row. Throughout this section, the boolean true is equivalent to the integer 1 and false is equivalent to 0. The problem is described using variables and linear constraints. Let us details those problem for a row containing n cells, numbered from 1 to n, and m positions, numbered from 1 to m. The variables are $p_{i,j}$, defined for every cell i allocated to the row and every position j of the row; $p_{i,j}$ is true if cell i is set to position j and false otherwise. Once the ILP solver has extracted a solution, the placement is uniquely defined by the values of $p_{i,j}$ in that solution.

Using those variables we define two sets of constraints. The first set of constraints u_i is used to assert place uniqueness: no more than one cell should occupy a position. The second set of constraint e_j is used to assert existence: every cell needs to occupy one and only one position. When carry-chain ares used inside the row, an extra set of constraints must be added to ensure the carry-chain coherency: that is that two cells connected by the carry chain are positioned next to each other and in the right order. This set of constraints defines a system of linear inequations:

- $u_j = \sum_{i=1}^{n} p_{i,j},\ u_j \leq 1$, defined for every position j
- $e_i = \sum_{j=1}^{m} p_{i,j},\ e_i = 1$, defined for every cell i

Let us now add to that system a energy function to optimize.

Let us now described the heuristic we developed. Our objective was to minimize the size of the connections, to avoid congesting the crossbar-based routing network when it was still supported. This can be done by using the sum of connection length as the goal to minimize.

$$\text{energy} = \sum_{a \text{ linked to b}} \sum_{j=1}^{n} \sum_{k=1}^{n} |j-k| \times p_{a,j} \times p_{b,j}$$

We can also use a non-linear goal, the sum of the squares of the connection lengths. It associates an even higher cost to long connections.

$$\text{energy} = \sum_{a \text{ linked to b}} \sum_{j=1}^{n} \sum_{k=1}^{n} |j-k|^2 \times p_{a,j} \times p_{b,j}$$

With the linear energy function, those could be hidden in the average. This energy function is still linear in the problem variables ($p_{i,j}$). Only the static coefficients are non-linear. The problem dimensions are not modified by this change (number of variables, number of constraints).

Let us now study the problem complexity. For a line with n cells and m possible positions, the P&R problems contains:

- $n \times m$ variables, one for each possible pair (node, position)
- $n + m$ inequations, n to assert the existence of a position for each node and m to assert that there is no more than one node at each position

The overall system contains then $n.m$ variables for $n + m$ inequations, plus the energy function to optimize.

9.3.2 Multiple-level placement

The previous single line problem is easily solved even for line of hundreds of cells. But the solution optimality is limited to one dimension. To obtain better placement on the matrix we designed an algorithm that generalizes the single line approach to multiple levels.

The set of variables is the set of the single-level problem for each of the rows, extended with new variables. This new variables are defined for cells i and j, connected together. Cell i is in one row and j in the next row, that is, i is an input a cell j. For every such pair (i, j) and for every pair of possible positions (r, s), r for i and s for j, we introduce the variable $q_{i,j,r,s}$ which is true when i is located on r and j is located on s. As the nodes are connected, they are not located on the same row and thus r could be equal to s, since it only encodes the horizontal position. The variables are constrained by the same set of rules as the single-level problem, defined for each row covered by the multi-level, extended with the following constraints:

- **position uniqueness**: for each input i of j, each position r of i, each position s of j, the value of variable q must be compatible with p_i and p_j:

$$q_{i,j,r,s} = p_{i,r} = q_{j,s}$$

- **link uniqueness**: for each link (i, j), there is one and only one $q_{i,j,r,s}$ set to true:

$$\sum_{r=1,s=1}^{m,m} q_{i,j,r,s} = 1$$

Those constraints are sufficient to ensure the uniqueness of both position and connection.

Let us now study the problem complexity. In a problem with row of size m, with n cells linked by o links and a multi-level of l levels, we introduced:

- one variable for each quadruplet (i, j, r, s): $n^2 \times m^2$ variables.
- one inequation for each possible position of each pair of connected node (link): $o \times m^2$ inequations.
- one equation for each link: o equations.

We can notice that the number of variables is the square of the single-level problem and the number of equations grew by approximately $o \times m$. The multi-level problem is much more complex: it takes longer for an ILP solver to find a solution, if one exists. In practice we went from a few second for the single-level placement to several minutes for two levels.

9.3.3 Routing

The DIRF offers two types of interconnects: crossbar-based and permutation-based. Each of them is supported by RKC's backend.

Crossbar-based routing The routing in the crossbar-based network has been implemented using a straightforward iterative depth search. Connections in the interconnect are considered one by one and routed using the shortest path available. If no path is available, the routing fails and RKC stops and reports an error.

Permutation-based routing To route inside the permutation-based Benes network we implemented the looping algorithm described in [125]. As Benes are permutation network, a configuration always exists and the routing can never fail.

9.4 DIRFsim: software emulation

A HDL simulator can be used to simulate the behavior of the DIRF RTL description. However for application design and validation, HDL simulation is not fast enough. To this purpose, our toolchain was extended with a software simulator: DIRFsim.

As illustrated by Figure 9.1, DIRFSsim execution is configured by the same configuration files as compiling and HDL generation. DIRFsim extracts the DIRF matrix dimensions from the array configuration file. The cell configuration and behavioral description are extracted from the cell configuration file.

DIRFsim simulates the single-master coprocessor architecture. It executes programs produced RKC to configure the emulated DIRF matrix. Those programs are manually extended with PUSH_INPUT and READ_OUTPUT instructions, plus their triggering counterparts, to send inputs to the DIRF, retrieve outputs and trigger pipeline stepping.

9.5 Conclusion

We have developed a functional toolchain which:

- generates DIRF HDL description from a parametrized template;
- produces configurations from application kernels;
- emulates the behavior of the DIRF on those configuration.

This toolchain is parametrized at several levels (array and cell configurations plus decomposition rules for RKC). This constitutes the first step to develop application for the DIRF and try out different architectural possibilities. We were able to introduce an original ILP-based P&R. The ILP P&R problems can be solved in practice because of the DIRF limited size. The next chapter will focus on applications case studies developed to evaluate our solution, and demonstrate capabilities and limitations of our framework.

CHAPTER 10
Application case studies

To evaluate the integration of a DIRF and develop/validate its toolchain we implemented two applications:

- an AES cryptography kernel
- a SHA1 hash function

AES implementation was the one that received the most attention, thus its description is more detailed. Section 10.1 details the implementation of AES. Section 10.2 focuses on the hash function SHA1. Section 10.3 introduces several architectural exploration attempts and their evaluations. Finally Section 10.4 presents the comparison of our operator with ASIC solutions, alongside a conclusion and perspectives for future work.

10.1 Advanced Encryption Standard

The Advanced Encryption Standard [142] is a widely used encryption/decryption standard. It is based on the Rijndael algorithm. This algorithm uses finite field arithmetic on the Galois Field $GF(2^8)$. It manages 128-bit, 192-bit or 256-bit keys while working on 128-bit data blocks. Encryption and decryption are very similar. They are based on multiple rounds : 10, 12 or 14 according to key length. The key is expanded to create as many 128-bit round keys as there are round. Each round consists of 4 operations :

- **SubBytes** : each byte is inverted in $GF(2^8)$ and submitted to an affine transformation.
- **ShiftRows** : each row (32-bit vector in the 128-bit block) is rotated from an amount which is row-dependant.
- **MixColumn** : the 4×4 byte matrix is multiplied by a constant matrix
- **AddRoundKey** : the round key is added considering each byte as an element of $GF(2^8)$.

The first and last round are slightly different.

A lot of AES implementation have been developed over the years, some in software, such as [15], others in hardware, such as [137]. Intel has chosen to implement AES by an ISA extension (AES-NI) supported by a specific hardware unit [64]. This solution is very efficient ($700MBs^{-1}thread^{-1}$ on a Core architecture [153]) but very area expensive. Our context (power aware embedded DSP) makes it costly to add an extra functional unit to the K1 core for such a specific application. AES is

only part of some of the applications targeted by the MPPA. However those applications (eg: network encryption) often requires a very high throughput on AES computation, higher that what a software implementation could provide. Thus we chose to study the implementation of AES into the DIRF.

In the DIRF perspective, the implementation described in [137] is very interesting, because it decomposes the AES into 4-bit operations. 4-bit operands fit perfectly into the DIRF grain, and a lot of 4-bit operations can be implemented into the RLC. This implementation relies on composite fields decomposition of $GF(2^8)$. This arithmetic technique has been widely studied, for example in [126]. The composite field arithmetic transforms computations in $GF(2^8)$ in computations in $GF((2^4)^2)$, called a composite field. It can be implemented as computation in $GF(2^4)$, with some transformation overhead. We implemented AES using [137]'s algorithm. Let us now details the mapping of the algorithm operations onto the DIRF. Sections 10.1.1, 10.1.2, 10.1.3 and 10.1.4 present the implementation of each step of AES on the DIRF. Those sections will detail how the basic cell described in Section 7.2, p.131 is used to perform $GF(2^4)$ arithmetic.

10.1.1 SubBytes

For the SubBytes step, the 128-bit input block is cut in 16 8-bit/byte chunks. The SubBytes operation is applied independently to each byte. This operation is a two-fold operation:

- The multiplicative inverse i in $GF(2^8)$ of the input byte is computed.
- An affine transformation, formalized as a bit matrix vector multiplication, is applied to i considered as a 8-bit vector.

This results in a non-linear transformation.

The first software implementation of SubBytes used Looked-Up tables (LUT) stored in memory [15]. It was discovered that this implementation was very weak with respect to cache-timing attack, due to the storage in main memory of the Look-Up Table. To circumvent this drawbacks, many ideas were proposed. Among them, the bitslice implementation of AES, suggested in [133], uses the N-bit processor datapath as X 1-bit operators to process X blocks in parallel. This implementation is purely computational: it computes each sub-function without using Look-Up Tables. As there is no memory dependency, the computation timing is independent of the keys and memory layout. Thus, two keys can not be distinguished by cache timing or cache collision means. This approach has been previously implemented on the K1, reaching approximately 40 cycles/byte for a complete AES encryption. A fully pipeline DIRF operator could provide around 1 cycles/byte performance.

We implemented the SubBytes operation as described in [137]. The first step, in this composite field implementation of AES, is to change the representation base. This operation is performed by a bit matrix multiply (BMM) which transforms, in parallel, each byte of the 128-bit input vector into its composite field representation. Kalray's K1 core implements a 8×8 BMM which performs the base change on the 128-bit in 2 cycles (and 2 operations). The next step in the SubByte operation is to compute the multiplicative inverse. [126] implemented this operation with 1 inversion, 3 general multiplications, 2 additions and 1 constant multiplication, all in $GF(2^4)$. Let us now study the feasibility of such decomposition in our architecture.

The **addition** over $GF(2^4)$ is a simple bitwise XOR which can be covered by a bitwise LUT. Then, the **constant multiplication** and **inversion** are simple 4-bit to 4-bit vector substitution and can be implemented by a 4-bit input, 4-bit output LUT. Finally, the **general multiplication** is a more complex operation, which consist of a carry-less multiplication followed by modulo reduction. We evaluated two solutions: integrating a $GF(2^4)$ hardware multiplier in the basic cell architecture or decomposing the multiplication into elementary binary operations. The results of this

evaluation, listed in Table 10.2, show that the hardware multiplier solution is more efficient and less costly. Therefore, as Pico-GA [113], we decided to integrate a $GF(2^4)$ parametrized multiplier in our design.

10.1.2 ShiftRows

The ShiftRows operation is a byte-level permutation of the data-block. The DIRF implementation exploit the interconnect network between rows of cells to perform this operation. This operation does not require computation resources (RLCs).

10.1.3 MixColumns

The composite field implementation of MixColumn is the most expensive part of this algorithm. It is decomposed in two parts :

- first compute the equivalent in the composite fields of the weighted bytes, this is implemented by 5 additions and 3 constants multiplications in $GF(2^4)$ for each input byte
- then accumulate the partial product. This is implemented by 3 additions in $GF(2^8)$ per input byte

Again, these operations are easily decomposed in our 4-bit wide RLC datapath. The addition in $GF(2^8)$ is a 8-bit wide XOR. It is easily implemented by two bitwise LUTs, one for the higher 4-bit and one for the lower 4-bit. We can even reduce the number of operation by combining multiple XOR/addition in a multiple-input LUT, or by implementing the 8-bit addition using a single 2-output LUT.

10.1.4 AddRoundKey

This last step is a bitwise XOR, which implemented in parallel on each 4-bit chunk. The only difficulty of this step, is that it adds a 128-bit input to the dataflow to provide the round key.

10.1.5 Conclusion

By using composite fields, we were able to decompose an AES round into 4-bit centric operations that could be mapped onto the DIRF architecture. For each of those operations we suggested at least a valid implementation. The $GF(2^4)$ multiplication, could be implemented two ways: we evaluated both solutions and made an architectural design choice to integrate the best. The best available implementation of the AES round on the DIRF was evaluated against an ASIC solution. Results are reported and discussed in Section 10.3 with the other application evaluations.

10.2 Hash function: SHA-1

The Standard Hash Algorithm 1 (SHA-1) [55] is a simple hash function built around rotations, additions, and bitwise operations. It starts by dividing the input message into 512-bit chunks. Then each chunk is used to build 80 32-bit words: $w_i,\ i \in [0, 79]$. Finally a main loop iterates once over each word to build a 160-bit hash. We implemented the main loop kernel of SHA-1 for any of the 80 possible indexes (**variable t**).

```
1 in int32 a, b, c, d, e, t, w;
2 out int32 r_a, r_b, r_c, r_d, r_e;
3 {
4     // k selection
5     int32 k_0   = 0x5A827999;
6     int32 k_20  = 0x6ED9EBA1;
7     int32 k_40  = 0x8F1BBCDC;
8     int32 k_60  = 0xCA62C1D6;
9
10    int32 xor_bcd = (b ^ c ^ d );
11    int32 a_rot_5 = a <] 5;
12
13    int32 ew = e + w;
14
15    // for t in 0 to 19
16    int32 r_a_0 = a_rot_5 + ((b & c) ^ ((~b) & d)) + k_0 + ew;
17    // 20 to 39
18    int32 r_a_20 = a_rot_5 + xor_bcd + k_20 + ew;
19    // 40 to 59
20    int32 r_a_40 = a_rot_5 + ((b & c) ^ (c & d) ^ (b & d)) + k_40 + ew;
21    // 60 to 79
22    int32 r_a_60 = a_rot_5 + xor_bcd + k_60 + ew;
23
24    int32 r_a_low = t >= 20 ? r_a_20 : r_a_0;
25    int32 r_a_high = t >= 60 ? r_a_60 : r_a_40;
26
27    r_e = d;
28    r_d = c;
29    r_c = b <] 30;
30    r_b = a;
31    r_a = t >= 40 ? r_a_high : r_a_low;
32 }
```

Listing 10.1: SHA1 code kernel

The SHA-1 code kernel, compiled by RKC to produce the DIRF configuration, has been reproduced in Listing 10.1. Let us lists the elementary mappings used to implement SHA-1 components. The rotations are implemented as the runtime-static mode described in Section 7.4, p139. Additions are mapped into the RLC adders, using the generalized carry chain to build the required 32-bit adders from the 4-bit RLC adders. Bitwise logical operations are mapped into RLC LUTs. Comparisons are implemented using RLC adders to perform the subtraction and the LL-CPU, described in Section 7.2.2, p.133, to extract the result sign. This sign drives a subrow of RLCs configured as multiplexers which implement the select part of the ternary operator. SHA-1 implementation was simpler than AES: all operations could be mapped on the default RLC structure, no architecture modification was studied. Implementation evaluation and comparison to ASIC are reported in Section 10.4.

10.3 Design exploration and results

Table 10.1 compares the efficiency of our two types of interconnection network on the AES application. A relevant data is the minimum size (width and depth) of a DIRF able to implement a given functionality, depending on the architectural parameters. For the crossbar-based interconnect, the use of the 3 cell outputs, for duplication purpose, had to be disabled, because the routing pressure was too high. Our placer and router were not able to find a working configuration within

network	depth	# cell used	max. level width	interco. area (wrt Benes)
Benes	11	760	100	1.0
Crossbar	12	764	96	3.6

Table 10.1: Comparison of interconnect efficiency on AES

application	depth	# cell used	max. level width	cell area (#gates)
AES with 3 cell outputs	-	-	-	
with GF multiplier	10	615	84	702
without GF multiplier	17	1024	128	670
AES with 2 cell outputs	13	818	95	670

Table 10.2: Comparison of RLC structure effects on DIRF size requirement for the AES kernel

an acceptable number of horizontal lines. This is not a default of the crossbar-based interconnect itself, but rather a limitation of our framework. RKC router for the crossbar-based interconnect is not very advanced. In the current framework/DIRF version, Benes interconnect is much more efficient than its crossbar-based counterpart because we are able to route more efficiently, resulting in an permutation-based interconnect 3.6 times smaller than the crossbar-based interconnect.

Table 10.2 compares several cell structure choices. For example, reducing the number of duplicated output of the basic cell provides a small width reduction of the DIRF, but it implies a larger increase in pipeline depth (and overall cell number). It does not have any effect on the basic cell area, since the duplicated output is obtained by adding a simple wire, and does not require logic gates. The other choice tested concerns the integration of a $GF(2^4)$ multiplier in the basic cell; removing it decreases by 5% the cell area but increases by a factor 2 the size of the DIRF needed for AES, nullifying the gain of RLC area reduction.

10.4 Conclusion and future work

Table 10.3 compares the area of the smallest DIRF able to implement a given function to the area of the same function, compiled directly to ASIC. The best application, SHA1, shows an overhead factor of 120: implementing DIRF to compute SHA1 will result in a solution 120 times more expansive than an equivalent ASIC implementation.

This work is still in its early stages. Nonetheless we could design a new reconfigurable architecture and provide a functional toolchain to support it. The DIRF and toolchain design was driven by the development of a few applications, we considered representative (mostly AES which

application	depth	max. level width	(# cell used)	area ratio wrt ASIC
AES	10	84	615	490
SHA1	8	88	580	120

Table 10.3: DIRF/ASIC area comparison for the same function

was our first concern). These developments lead to architectural modifications and toolchain improvements. We are still far from providing a satisfactory answer to the question raised in introduction: can we reduce the overhead of reconfigurable computing to a factor lower than 40?

There are many more optimization opportunities to explore:

- macro-block based, medium grain structure implies very few distinct configurations. For example a 32-bit addition which requires 8 cells, only implements two distinct configurations: one 4-bit add without carry for the LSB and 7 add with carry. Using a configuration cache/scratch-pad filled at the beginning of reconfiguration and streaming configuration IDs rather than lengthy full cell configuration should reduce configuration size in program memory, and maybe reduce configuration time.
- explore configurable control capabilities similar to PicoGA's, and drive inputs towards inner rows, or outputs from inner rows, rather than relying on top inputs and bottom outputs.
- investigate dynamic compilation, so that specific configuration parts (eg AES keys) can be swapped directly in the configuration bitstream;
- explore multi-context configuration to improve overall efficiency ratio;
- investigate full custom design of the DIRF instead of its current design based on standard cell. It should bring a decrease of RLC area by a factor close to 2.

The last point could be extended further to floor planning: the current HDL description does not allow us to benefit from forbidding long distance connection in the interconnect, the synthesizer is not aware of that fact and will not exploit it. This has not been done yet and require some competencies outside the scope of this work.

Part III

NIMT Architecture

Chapter 11

SIMT improvements for divergent control flow

Graphic Processing Units (GPU) are architectures specifically designed for rendering graphics. To sustain the heavy computation needs induced by frame rendering, GPUs rely on efficient floating-point arithmetic. They are throughput-oriented architectures: maximizing GPU's efficiency is related to maximizing the occupancy of their operators. The optimal occupancy of an operator is defined by the operator repeat latency and the application parallelism. Maximal efficiency is reached when an operator can be fed new inputs every cycle.

GPUs provide a simple programming model called Single Program Multiple Data (SPMD): a common code kernel is executed many times on independent input data gathered in blocks. This model is well-suited to the regularity of graphic processing: graphic kernels exhibit very regular control flow (few branches, no loop) as well as regular memory access patterns. Moreover those kernels are executed on numerous independent data (geometry primitives or pixels). Graphics processing is often embarrassingly parallel. Thus GPUs have been designed to take benefit of this data parallelism, they are built around hundreds (or even thousands) execution units. To save area, those execution units are often simple, pipelined, floating-point unit, gathered by group of tens with a single instruction fetch/decode, implementing a Single Instruction Multiple Data (SIMD) scheme called Single Instruction Multiple Thread (SIMT). This simplified architecture allows to pack more functional units and to meet the throughput requirement of graphic processing applications.

GPUs are now used beyond graphic rendering. Some successful experiments have demonstrated the relevance of GPU in scientific computing domains such as physic simulation or biochemistry. This trend, called General Purpose GPU (GPGPU), has been increasingly successful [99, 66]. It has even dictated some evolution in GPU architecture. For example the binary64 support for scientific computing which is not required by graphic processing. GPGPU applications still expose a very high degree of parallelism but some shows a more irregular control flow. Control flow is less structured and branch behaviour diverges among threads.

The SIMT architecture of GPU was not designed for such divergent control flow and tend to be inefficient on such applications. We suggest architectural modification to improve SIMT architectures for divergent control flow. We call this architecture N-IMT (N-Instruction Multiple Thread), because it executes N instructions simultaneously over multiple threads. The work described in this chapter describes results of a collaboration with Sylvain Collange [22]. It constitutes an evolution of some of its earlier studies.

The remaining of this chapter is organized as follows: Section 11.1 describes a state of the art SIMT architecture and the mechanism used to support branching code, it also introduces other studies about improvement for branch support. Section 11.2 introduces our contribution towards a N-IMT architecture and the new mechanisms designed to improvement divergent branch sup-

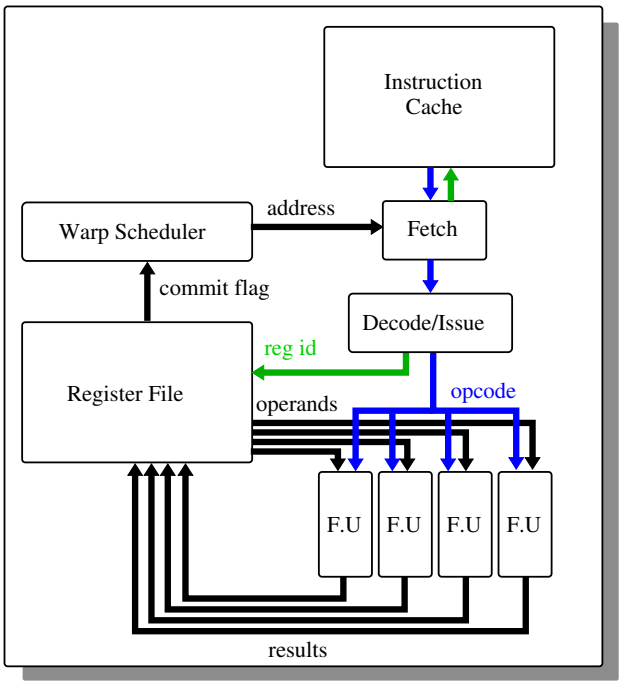


Figure 11.1: Single-Instruction Multiple-Thread architecture

port. Finally Section 11.4 concludes this chapter and opens up on future work perspectives.

11.1 SIMT architecture

Graphic processing applications require a very high throughput: thousands of polygons have to be processed and millions of pixels have to be determined to render a single frame. Multiple frames are rendered every second. Both polygon and pixel processing exhibit a large uniformity.

A single program with different input parameters is used to compute the polygon projection. Once projected, the polygons are rasterized into pixels. Each pixel coming from the same polygon will once again be rendered using the same program (shader) with variable input parameters (attributes such as position, textures, shadows ...). GPU architectures have been designed to exploit the large uniformity exhibited by polygon or pixel processing. They implement the Single Program Multiple Data scheme: a single program is executed on multiple independent data. The execution of the program on one data is called a thread. Thousands of threads are started together and executed by each single streaming multiprocessor (SM) present on a GPU core.

Threads are gathered into convoy called warp (NVIDIA's appellation), containing between 16 and 64 threads. Each thread is allocated a static position in its warp.This position dictates the register bank where the thread's registers are allocated and the functional unit index which executes the thread. A warp progresses in the program in lock-steps, each thread executing the same instruction and having the same program counter (PC). This constraint allows the SM executing the Warp to implement a SIMD architecture: a single instruction is executed across several processing units (one for each thread in the warp). SM architecture is illustrated by Figure 11.1. A SM contains multiple functional units (generally proportional to the warp size), instruction and data caches, a single instruction fetch/decode and a banked register file.

The decoded instruction is broadcast to all functional units (FU), each of whom executes a different thread from the warp. The register file is addressed through a single lane bus since the

SIMD architecture ensures that every thread is going to access the same register at the same time. Each functional unit (F.U) reads and write its own chunk of the vector register.

In addition, to hide operators and memory latencies, a single SM executes multiple warps. When a warp is stalled on a cache miss or on another dependency, a different warp is scheduled for execution. Each thread has its own registers in the register file. The register file is organized in banks, one per lane/functional units. Multiple threads positioned at the same index in a warp shares a register bank but as the can not be scheduled together, no bank conflict can happen.

11.1.1 Management of multiple branches: mask stack

When the program executed by a SM contains a if-then-else branch, it is possible that not all thread in a warp follow the same branch direction. Contrary to pure Single Instruction Multiple Data (SIMD) architecture, SIMT is designed to support those divergences in the control flow. The thread execution can be desynchronized temporarily: let us assume a group of threads come to a divergent if-else branch, let us call the if-group the threads that execute the if-branch and else-group the threads which execute the else-branch. Once the branch instruction has been executed the warp does not exhibit a single common PC: the if-group PC is at the start of the if-branch while the else-group points towards the start of the else-branch. The architecture does not know how to execute the two branches simultaneously so the execution is sequentialized. First, the else-group is stalled and the if-group executes the totality of the if-branch. Then the if-group is stalled while the else-group executes the totality of the else-branch. Finally both groups are merged together and resume standard execution after the if-then-else block.

This mechanism is implemented through a mask stack and convergence instructions in current GPUs. It was already the case in the CHAP [93]. A memory stack of masks is implemented for each warp, each entry corresponds to a group of threads executing the same instruction; it contains a mask and a PC. A mask contains a validity bit for each of the threads within the warp. The bit, corresponding to the thread, is set if the thread belongs to the group.

Initially the stack contains a single mask with all bits set. When a branch is encountered the current mask (top of the stack) is split in two: a mask which indicates the thread following the else-branch, which is pushed onto the stack first, and a mask for the if-branch threads which is pushed second and becomes the active mask. When the if-branch execution comes to an end, its masks is popped from the stack and the else-branch mask becomes the current mask, triggering the execution of the else-branch. Once the else-branch is executed its mask is popped and the merged mask which was present on the stack before the branch becomes the current execution mask.

We are going to illustrate the mask stack mechanism on a small example. The example program is reproduced in Figure 11.3. The execution of this program by a mask stack architecture is illustrated by Figure 11.2. Our example considers a Warp containing 4 threads (thread id [TID] 0 to 3). CPC is the current PC being fetch and executed.

A branch end is detected by the execution of a specific instruction **endif** or **endelse** which must be statically inserted by the compiler. It is the execution of that instructions which triggers the mask stack mechanism.

This mechanism allows a SIMT architecture to support divergent control flows. However this support lacks efficiency. As illustrated by Figure 11.2, the program execution is partially sequentialized. If not all threads follow the same branch direction, both directions have to be executed sequentially thus increasing the program latency and decreasing the throughput. Indeed once the program diverges part of the functional units becomes idle (all those corresponding to inactive thread-ids).

The mask stack mechanism also has the disadvantages of requiring a stack per warp which is often implemented as a separate hardware stack for each warp.

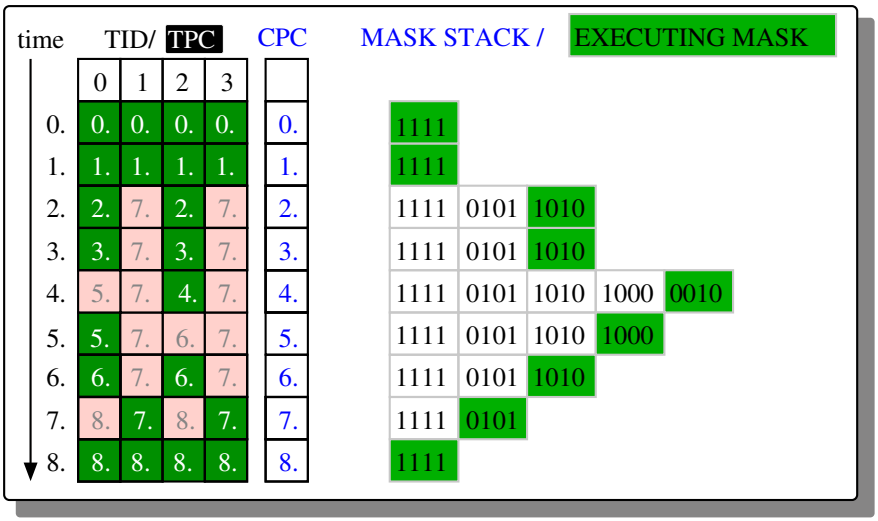


Figure 11.2: Example program execution on SIMT architecture with mask-stack

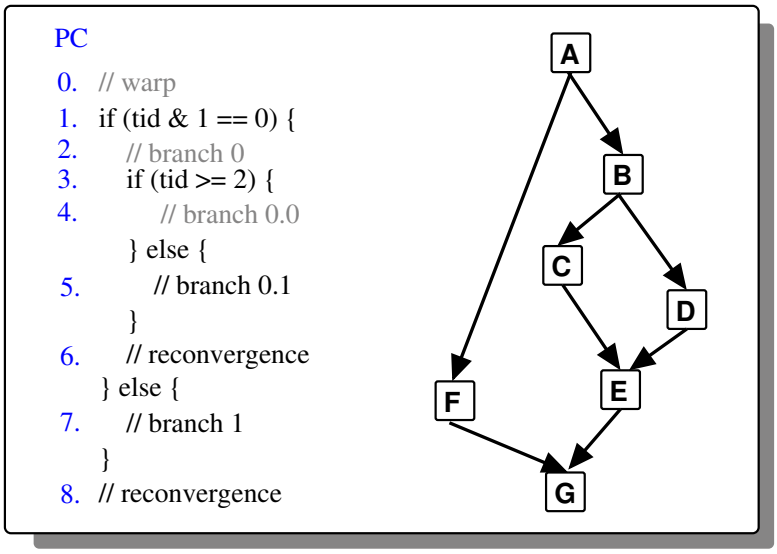


Figure 11.3: Example of kernel program, *tid* is the thread identifier (unique for each thread in a warp).

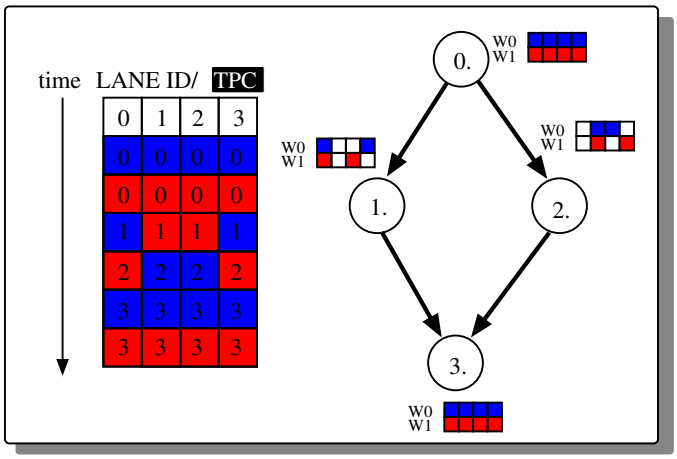


Figure 11.4: Dynamic Warp Formation with thread migration

Let us now review two recent proposals to mitigate efficiency loss due to thread divergence in GPU.

11.1.2 Dynamic Warp Formation

The efficiency decrease due to branch divergence is related to the static formation of warps (thread convoys). Threads from other warps might share a common PC with active threads in the current warp and thus might be available to fill-in the inactive lanes. Since warps can not be modified once created those inactive lanes remain idle.

To lessen this inefficiency, [59] suggests to allow for Dynamic Warp Formation (DWF) by modifying the GPU base architecture. Each time a divergent branch is executed, the current warp is split into two sub-warps each corresponding to a branch direction. Before being added to the warp pool, the scheduler tries to merge those new sub-warp with existing warps which share the same next PC. Once two warps with identical PC are found, the scheduler fills the empty lanes of the first warp with threads from the second warp. Several policies can be applied to perform that process. The first one illustrated by Figure 11.4 is to move the thread of the second warp to available lane in the first warp. Two warps W0 and W1 execute the program whose control flow is represented on the right part of the figure. When the divergence is executed threads from W1 are used to fill idle lanes of W0, this sometime require threads to migrate from a lane to another. Standard GPU architectures are not built to support thread lane change: the register file is heavily banked and each lane as access to a specific banks. This policy requires the insertion of a full crossbar between the register file and the execution lanes. This can prove quite expensive. [59] offers another policy called *lane-aware*. This policy only adds a thread to a warp if the thread lane is inactive in the warp. It is illustrated by Figure 11.5. It does not require any crossbar between the register file and the functional units. The main difference with a standard GPU is that each register bank must be addressed independently. In static warp formation, as all thread executed simultaneously were issued in the same initial warp they share a common address in the register file: each thread accesses the same register id in different banks, a thread always accesses the same bank during its execution. This mechanism is only dynamic in part, since the warp formation is made on divergence branch and not at each execution cycle. This policy requires fewer modifications of the GPU architecture. Those modifications are estimated to increase the area by less than 5%.

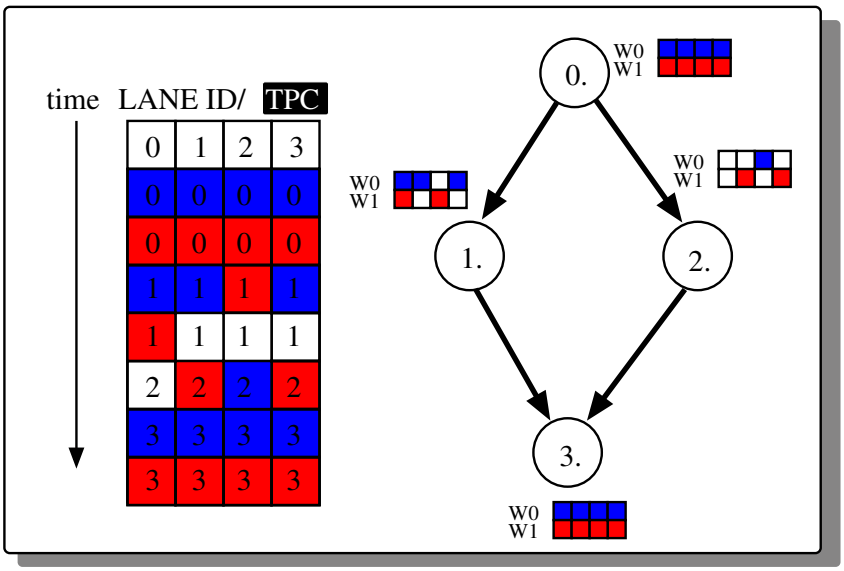


Figure 11.5: *Lane-aware* Dynamic Warp Formation

11.1.3 Thread frontier

Apart from the warp fragmentation on the branch, another reason SIMT architecture can loose efficiency on divergent control flow is when it fails to enforce early reconvergence. This is the case on unstructured control flow. An example of such control-flow (extracted from [49]) is given in Figure 11.6. This control-flow can be obtained after compiler optimization on a more generic if-then-else structure. Mask-stack architecture uses post-dominator (PDOM) reconvergence. When generating control flow, the reconvergence instruction for a divergent branch d is inserted at the immediate post-dominator of d. For Figure 11.6 it means that the reconvergence for A is enforced by an instruction inserted at the beginning of the block G. The right side of Figure 11.6 gives an example of execution of 5 threads ($T0 \rightarrow T4$) on the left side control flow graph. As the reconvergence is only enforced at G, threads are divided into subgroup and never merged before G. The SIMT architecture executes several times the same basic block (for example C is executed once by T0,T4 and once by T1, T3) when it could have been merge to avoid code expansion and increase unit occupancy.

[49] introduces a new concept to improve thread reconvergence: thread frontier. It intends to avoid code expansion and multiple executions of basic blocks by facilitating convergence as soon as possible and sooner that what is permitted by PDOM. For each possible convergence point, this is done by building a set of basic block called the *thread frontier* which gathers all basic blocks where threads converging on that point could be located. When the convergence point C is reached by a thread T, if any thread is still in the thread frontier, T should be stalled and wait for possible convergence. Once the thread frontier is empty, all threads that were supposed to reach C did it already and T execution can resume.

11.2 Towards NIMT architecture

To alleviate this difficulty we suggest to modify the standard SIMT architecture to a N-IMT architecture, by adding a multiple-fetch unit. Designed to improve the processing elements occupancy, this modification is associated with a specific scheduling policy called min-PC and two mech-

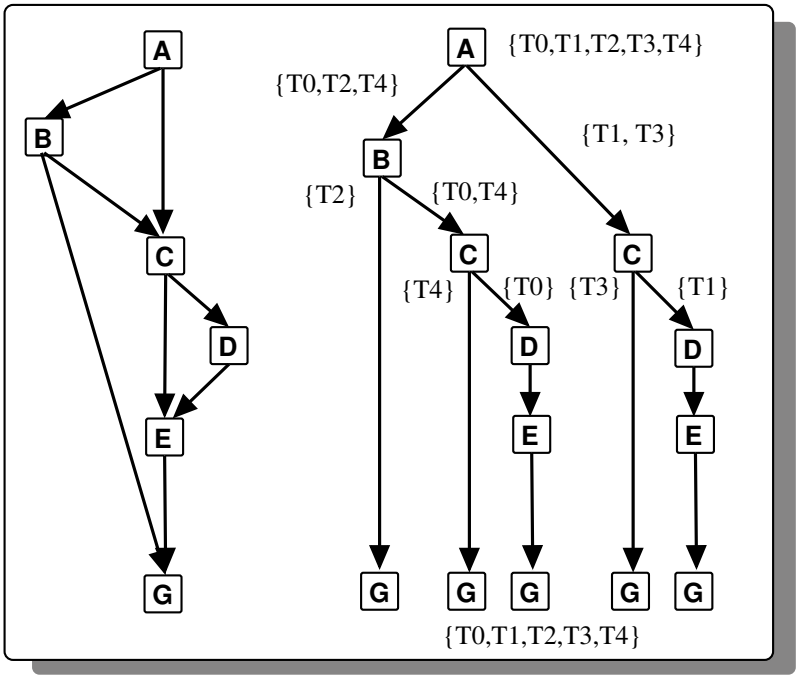


Figure 11.6: Example of thread frontier

anisms: simultaneous branch interweaving (SBI) and simultaneous branch interweaving (SWI). To implement this architecture, we designed two original hardware modules: a pair of hot/cold context tables and a double scheduler.

11.2.1 Multiple Fetch units

The first idea we studied was to exploit the branch level parallelism: executing simultaneously the two sides of a branch. This is more efficient that the sequentialization forced by mask stack architectures. Unfortunately standard SM are built to fetch a single instruction per warp and are not able to provide the two instructions per cycle required to execute both branch sides. An easy solution would be to add an extra fetch unit to each SM. The two fetch units share an instruction cache and through a comprehensive scheduling policy are able to provide two instructions for decode and execution each cycle. This solution changes the Single Instruction Multiple Threads (SIMT) to a N-Instruction Multiple Threads (NIMT). N is 2 in our case, but this scheme can easily be generalized to an arbitrary number. If N is equal to the number of thread in a warp, NIMT becomes a perfect Multiple Instruction Multiple Threads (MIMT) with possibly a different instruction fetched for each thread. GPUs chose to discard the MIMT architecture because of its cost: there is no more factorization of fetch/decode unit. The instruction fetcher/dispatcher required to supply the execution units is very complex and expensive. NIMT is making a step in this direction and to counterbalance the induced cost increase, we decide to double the warp sizes.By doing so, we basically fuse two standard streaming multiprocessors into one, building NIMT with little hardware overhead. Modifying the warp length could have consequences on application behaviours and performance. Our evaluations, presented in Section 11.3, shows that these consequences are limited and do not impact the performance improvement of our architecture.

The first consequence is we shift the grain from warp to warp-split (or sub-warp). Once a warp encounters a divergent branch (not all threads going in the same direction) it is split in two warp-splits. Each warp-split has its own PC and activity mask. The activity mask as the same

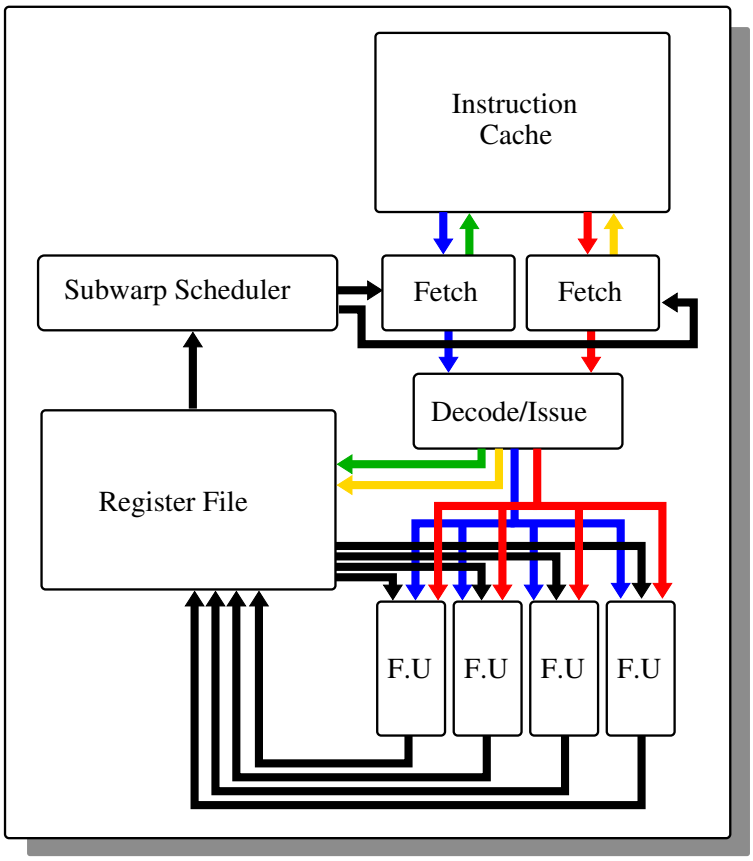


Figure 11.7: N-Instruction Multiple-Thread architecture for N=2

length as the initial (fully populated) warp and the bit sets indicate which thread is part of the warp-split. At a divergence point the activity mask is split into two complementary split masks whose unions is equal to the pre-divergence activity mask. A bit set in the activity masks is set in one and only one of the split masks. A bit null in the activity mask is also null in both split masks.

At a convergence point, two warp-splits are merged into one, as their activity masks.

Figure 11.7 illustrates the NIMT architecture. The two fetch units are connected to the same instruction cache. The decode/issue units provides two sets of register indexes to the register file, one for each decoded instruction. Each instruction is broadcast to every functional units. Each functional unit receives a single sets of registers, executes a single instruction and commit in a single register index. The selection between the two sets is done through a way-selection mask, transmitted by the subwarp scheduler to each functional unit and to the register file. For each bank, this mask indicates which of the two register ids to use. For each lane, the mask indicates which of the two instruction to execute. This mask contains a bit per thread, if the bit is null the thread executes instruction from way 0 (fetch unit 0) and if set it executes the instruction from way 1 (fetch unit 1).

11.2.2 Enforcing earliest possible reconvergence: min-PC scheduling policy

When allowing parallel execution of divergent control flows, a problem arises: the fragmentation of the initial warp. If each time a branch is encountered a warp is split again and again, it could end in single-thread warp-split. Such case nullifies SIMT/NIMT efficiency, the N fetch units can not supply enough instructions to fill even half the functional units. To limit this problem the

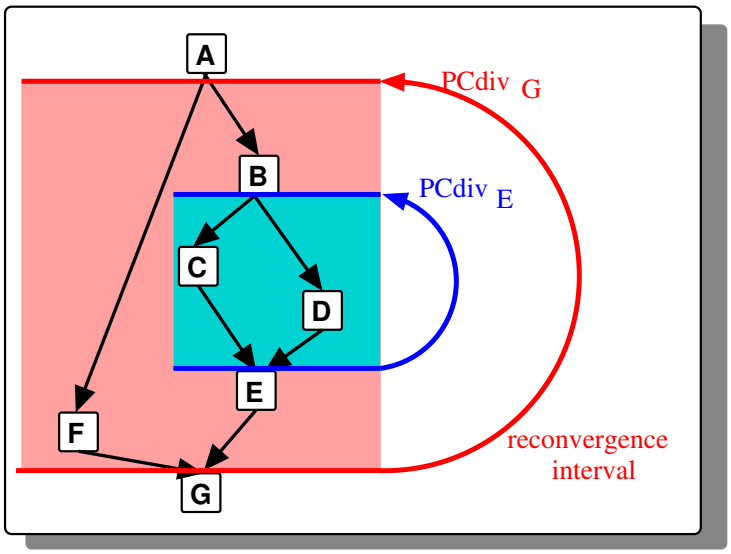


Figure 11.8: Reconvergence instruction and min-PC policy for the example program

best solution is to facilitate convergence: two warp-splits should be merged into one as soon as possible. This was already the policy of the mask stack architecture with the **endif** and **endelse** instructions: when the control flows reconverges those instructions made sure the mask stack was popped accordingly and that a common execution could continue. The problem with such instructions is that we require as many of them as they are levels of overlapping if-then-else. Note that in classical sequential processor, there is no need for such instruction at all.

We suggest a solution based on convergence instructions, helped with intelligent instruction ordering. When generating the source code, the compiler enforces the following rules: basic blocks are partially ordered so that if a basic block B is in dependency of a basic block A it is located after in the source code. This rule has to be broken when backward jumps are required (such as loops), but is fairly easy to enforce otherwise.

A convergence instruction includes a payload: the divergence PC PC_{div}, computed by the compiler. PC_{div}indicates where all the threads that should converge at the convergence point started diverging. This define a reconvergence interval as illustrated by Figure 11.8

The main rules of our scheduling policy is to execute the minimal active PC for each warp. Thanks to the rule enforced by the compiler we are making sure to execute the thread least advanced in program execution (at least in non looping control flows). When we have to provide a second instruction to fetch we chose the second minimal active PC.

11.2.3 Simultaneous Branch Interweaving

To exploit the newly designed architecture we suggest two different policies. The Simultaneous Branch Interweaving (SBI) described in this section and the Simultaneous Warp Interweaving which is described in Section 11.2.4. Both those policies rely on the two fetch units and require some extra specific architectural modifications.

SBI focuses on exploiting the branch-level parallelism: it consists in executing simultaneously both the first minPC (MPC1) and the second minPC (MPC2) instructions for each Warp. The two PCs are provided by a system of context tables described in Section 11.2.5. They are fetched and decoded, the activity mask is transmitted to the register file and the functional units which execute at most one instruction each.

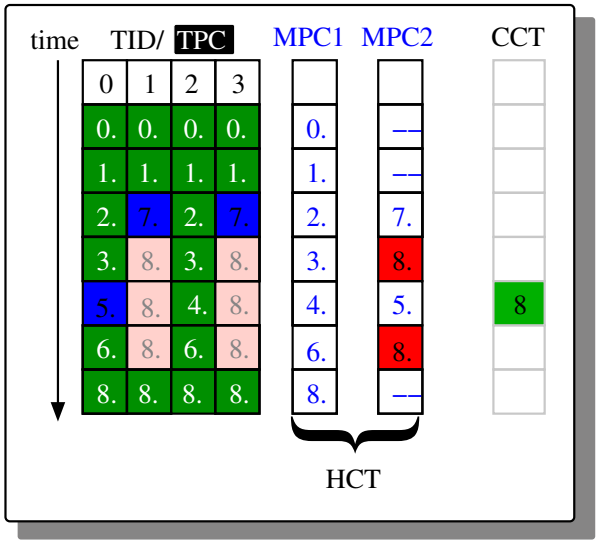


Figure 11.9: Example program execution with SBI

Figure 11.9 illustrates the execution of the example kernel on an SBI-enabled architecture. It can be noticed that the overall latency is shorter with respect to the execution on a mask-stack architecture, presented by Figure 11.2. On average, more functional units are active each cycle.

11.2.4 Simultaneous Warp Interweaving

SBI is limited to intra-warp parallelism (BLP). The benefit of the second fetch unit can be extended to inter-warp parallelism.

The main idea is to extract warp-splits from different warps for simultaneous scheduling. This technique is called the Simultaneous Warp Interweaving (SWI). A first warp-split is selected from an active warp and scheduled for execution. While it is waiting for execution, a second warp-split from another warp is selected and also scheduled for execution. To allow simultaneous execution the second warp-split (selected later in the instruction cycle) must be compatible with the first one. It must not occupy the same functional units or require access to the same register banks. As those resources can not be shared, it would imply sequentializing their use and thus decreasing the architecture efficiency.

To solve the challenge of finding a compatible second warp-split we designed a friendly-warp lookup table. This table is implemented using a modified content addressable memory (CAM). The address decoder has been modified so that the CAM returns the first entry whose activity mask is compatible with the mask used to address it.

A fully associative CAM containing each sub-warp is a very expensive hardware module. We made several experiments with various level of associativity, restraining the research of friendly sub-warps to a more limited set rather than the complete sub-warp pool. Results are reproduced in Figure 11.10. It can be noticed that impact of using a 3-way versus a fully associative CAM is only a slowdown of less than 7% for all applications and even less than 4% for most applications.

11.2.5 Hot and cold context tables: providing minPC list with easy access

The introduction of warp-split is not well-suited for the mask stacks. The architecture has to maintain accessible several warp-split for each initial warps. The number of sub-warp/warp-splits

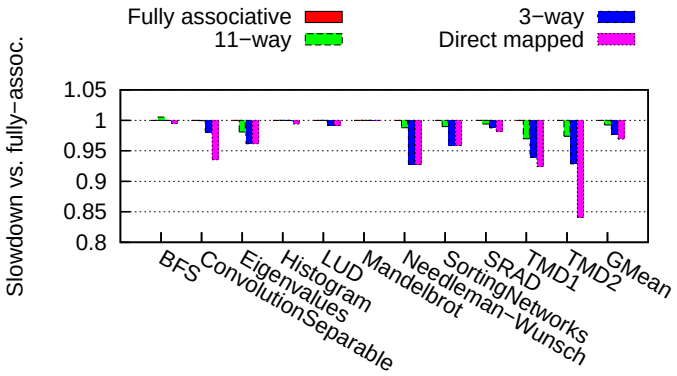


Figure 11.10: Slowdown of various set-associativity with respect to fully associative friendly warp lookup tables

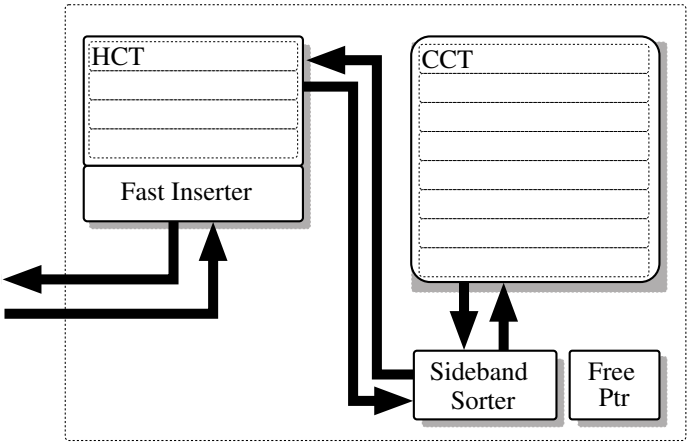


Figure 11.11: Structure and relations of the context tables

may be superior to the depth of the mask stack.Several branches can now be executed in parallel, each of them may diverge once again, increasing the number of simultaneous PCs. We suggest a new mechanism called context tables. It is based on two tables: the hot context tables (HCT) and the cold context tables (CCT). The structure and relations of the context tables is illustrated by Figure 11.11. The union of those tables stored the complete set of created (and not yet merged) warp-splits. The HCT stores active warp-splits that may be executed any moments and that may diverge or merge. The CCT stores inactive warp-splits that were recently created but do not fit in the scheduling policy (low priority, waiting on an event ...).

While the HCT needs to be latency efficient (quickly responsive), it only needs to contains N entry per warp which is enough to fill the fetch units. The CCT on the other ends may contains many more entries per warp but it does not need to be accessible as fast as the HCT.

The context tables also manage ordering of warp-splits. As our policy is to provide the two minimal PCs for each warp we need to make sure those are available at any given time. To do so the HCT stores both MPC1 and MPC2 and the third min-PC. More precisely the HCT stores an entry for each warp. Each entry contains the activity mask, and a SP/PC for each of the first three warp-split of the warp. The entry also contains a pointer towards an index in the CCT where the list of ordered warp-split (activity mask, SP/PC) for that warp continues.

A module called the **sideband sorter** ensures that the HCT entry contains MPC1, MPC2 and MPC3 at any given time. When a warp-split execute a divergent branch the sideband sorter splits the warp-split again, updates the masks, MPC1, MPC2 and MPC3 and sends to the CCT any remaining warp-split. When warp-splits are merged, it extracts an entry from the CCT to complete MPC1, MPC2 and MPC3.

The sideband sorter has to be use each time an entry of the HCT is used for execution to make sure the MPC1, MPC2, MPC3 order is enforced. To enforce that rule the CCT contains a sorted list of inactive warp-splits for each warp. Contrary to the HCT were a single entry of the table contains 3 warp-splits, the CCT is organized as a memory where each entry contains a warp-split and a pointer towards the next entry for the same warp. Unused entry are chained together in a **free list** whose head index is made easily accessible.

When a warp-split is ejected by the HCT it is sent to the CCT. The CCT inserts the new entry into the sorted list of the corresponding warp. This process is sequential: the CCT reads the list entry one by one in order until it finds one whose SP/PC is equal of greater that the entry to insert. If SP/PCs are equal both entries are merged: the activity masks are OR-ed. If the SP/PCs is greater, the CCT entry is replaced by the new entry and moved into a free location. The new entry points toward the newly occupied location. The old entry is moved to avoid a read-modify-write of next pointer of the previous list entry which is not available to the CCT sorter anymore.

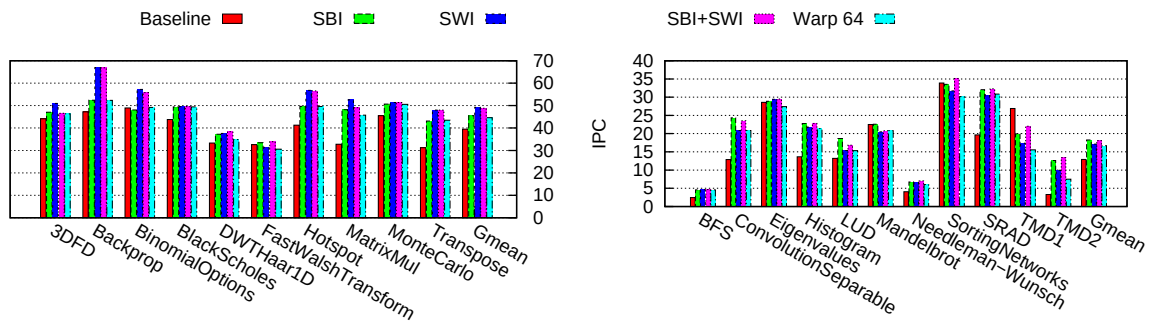
Let us go back to the SBI mechanism described in Section 11.2.3 and illustrated by Figure 11.9. We can note that less memory is used to store the HCT/CCT entry that what is necessary to store the mask stack (Figure 11.2): the HCT/CCT contains at most 3 entries when the mask-stack peaks at 5 entries.

11.3 Evaluation

The evaluation of our contributions was made using Barra [30]. Barra is a GPU simulator which was modified to get a cycle-accurate simulator of the streaming-multiprocessor pipeline. The benchmarks considered are from Rodinia Cuda benchmark and from NVIDIA Cuda SDK. Two implementations of the table maker dilemma are added to that pool of tests. They are of particular interest because they exhibit a very irregular control flow [56]. Results are separated between regular applications, which exhibit an average IPC greater than 30 on 64-wide warps, and irregular applications.

Regular applications sees a performance increase of 15% with SBI and 25% with SWI.

For irregular applications, the performance is increased by 33% with SBI and by 41% with SWI.



(a) Regular applications

(b) Irregular applications

Figure 11.12: Performance of SBI, SWI, and combination of SBI and SWI, with a thread-frontier based 64-wide warp implementation for reference.

	Area ($\times 1000 \mu m^2$)			
Component	Baseline	SBI	SWI	SBI+SWI
Register File	-	+570	+570	+570
HCT	66.8	88.8	43.8	88.8
CCT	584.4	480.8	480.8	480.8
misc.	141.4	118.4	148.4	226.0
Total	791.6	1258.0	1243.0	1365.6
Overhead	-	466.4	451.4	574.0

Table 11.1: Area overhead SBI, SWI and SBI+SWI components

Hardware overhead We synthesized the main components required by the architectural modifications we suggested. Results, normalized to 40nm technology, are reported in Table 11.1. **Misc.** represent other components not described previously such as an instruction buffer, a scoreboard for dependency checking and a warp scheduler which were modified in our new architecture. We compared these overheads to the area of a complete Streaming Multiprocessor from NVIDIA's Fermi architecture (40nm technology), estimated to $15.6mm^2$ from a publicly available die photograph. The respective overhead for SBI, SWI and SBI+SWI correspond to 3.0%, 2.9% and 3.7% of the total SM area.

11.4 Conclusion and perspectives

We have shown that with little modifications SIMT-like architectures can be improved to support divergent control flows. These modifications had to stay light, the idea was not to evolve into an expensive MIMD architecture but rather to alleviate some of the defaults of pure SIMD architectures. The evaluation of our solutions show promising results, in term of performance improvements and complexity. The HCT/CCT bundle is considerably lighter than the mask-stack for each warp. Overall SWI/SBI requires a small increase in silicon but the performance gain seems sufficient to justify these modifications.

Extending affine cache system with SWI We intend to pursue the development of our architecture by merging it with another previous work of S. Collange: the affine cache. S. Collange has shown in [31] that a large amount of computations performed by a GPU is in fact affine: most SIMD lanes computes exactly the same value (scalar) or values that are linearly dependant (given by an affine function on the lane id). In [32], S. Collange used this analysis to suggest an affine unit for GPU: affine computation is pulled out of the SIMD units to be processed by a separate unit. In the meantime the SIMD lanes can be switched off (clock gating, recirculation ...), thus saving power.

It would be possible to mix the Simultaneous Warp Interweaving technique with the Affine Vector Cache. Indeed in our system an affine computation can be issued simultaneously with a vector computation, since two fetch units are available and since those instructions do not occupy the same units. The current difficulty we are studying is the register lane sharing. For now the affine vector cache is embedded into the standard SIMD registers/cache and thus shared with the vector computations. If we wish to simultaneously execute an affine and a vector instruction we may need to access simultaneously the same register bank. This is not allowed on GPU architecture and implies a sequentialization of the computation. We are currently investigating a lane-aware scheduling similar to the technique described by Fung in [59]. By ensuring that the affine and vector warp-split do not share a lane we can avoid register bank conflict. This reduces

the possibility of issuing simultaneously those two types of instruction but simplify the register conflict management. Indeed no crossbar between the register file and the SIMD lanes is required, since each lane accesses a different register bank.

There is still a lot of room for improvements for SWI and SBI. Those techniques can be extended and improved. GPU architecture are evolving towards better divergence support. It seems NIMT already is already being implemented. The Most recent architecture from NVIDIA: Maxwell, implement multiple warp schedulers per SM. It does not seem to support warp interweaving, but rather uses the scheduler to improve occupancy of heterogeneous clusters of functional units (binary64 units, binary32 unit, memory units, integer units) by simultaneously executing multiple warps. The regular control flows of early graphical applications have given way to more complex and expressive divergent control flows. The architecture needs to adapt to this change and the NIMT scheme appears as an interesting response to this challenge.

Part IV

Conclusion and perspectives

12 Chapter 12

Conclusion and future works

Versatile arithmetic efficiency is challenging embedded processor designers. They need to balance the cost of operators with their benefits, taking into account an ever growing range of computation intensive application needs.

Floating-point is the first challenge we addressed in this work. We mixed software and hardware design to provide a good support without requesting too much silicon. The basic operators are implemented in a hardware floating-point unit. This unit integrates original operators which provide a balance between the capabilities and cost of binary32 and binary64 implementations. This work allowed the MPPA to target new markets such as high performance computing and digital signal processing which rely on intensive floating-point computations. Our contribution was concluded with the development of the new version of K1's FPU, built around a binary64 FMA and integrating a binary32 SIMD unit, the MPFMA and 2D dot-product.

In constrained embedded systems, hardware implementations are too costly to be generalized to every operations. Thus we studied the development of highly optimized software primitives to complement the hardware. By adding some low-cost hardware seed operators we were able to provide time-predictable and efficient implementations of division and square root. To address the challenge of developing such software, for more functions and optimized for numerous different contexts, we propose a code generator dedicated to function development. This generator is still in an early stage of development. However it is already able to generate useful implementations (eg: some vectorized correctly rounded functions) and to certify part of those implementations. This project will soon be integrated into the development effort of the ANR Metalibm project (starting at the time of redaction). Our future efforts include a more advanced support for architecture-specific generation, a better integration of certification and the implementation of more functions and more flavours for each function.

We also tried another path to answer the versatility versus efficiency challenge: using a reconfigurable matrix to implement small computing kernels. This idea required an heavy development: parametric HDL description, compiler and simulator. More architectural exploration is needed before such reconfigurable matrix is scheduled for production. Reconfigurable circuit are already used successfully here and there, but not at a very large scale. We feel their capacities are underused and the right design, assisted by the right framework will offer interesting responses to the efficiency versus versatility challenge. This design still has to be found.

Finally, we tried to broaden our vision by considering arithmetic efficiency from a different point of view. We took an existing architecture, the SIMT-based GPUs, and studied possible architectural evolutions to make a better use of already available processing elements. The problem of such architectures initially designed for uniform, embarrassingly parallel workloads, is that those workloads are diversifying quickly and getting more and more branching and heterogeneous. We

proposed, designed and evaluated a solution, called NIMT, based on multiple fetch units, to improve SIMT architecture efficiency on divergent workloads. The promising results we obtain, and some recent architectural evolutions in commercially available GPUs (NVIDIA's Maxwell architecture) show that hard SIMT must evolve towards more versatile architectures, which, while still relying on embarrassing parallelism to be efficient, offer a better support for non-uniform and divergent parallel workloads.

This thesis gave me the chance to explore a lot of paths and multiple responses to the efficiency versus versatility challenge. Some of those solutions gave quick answers that I was able to integrate into a fantastic project which I am very proud to be a part of. Others raised more questions than answers and require further investigation. Finally some gave early promising results and launched me towards the next step which I hope to be as enlightening and attractive.

Part V

Appendix

Chapter 13

Introduction to correctly rounded implementation

As defined previously, a correctly rounded implementation returns the closest floating-point number to the exact (mathematical) evaluation of the function. In rounding to nearest, this is equivalent to an overall error less than the half ulp measure for most cases.

Every standardized correctly rounded floating-point operation (addition, subtraction, multiplication, FMA) has an error bounded by the half ulp. This means that if the last operation of a function implementation is performed using the same precision as the output format, the last rounding error bound already reaches the overall error budget for the function implementation. There is no more room for the approximation error or any previous rounding errors.

This implies that the implementation needs to evaluate the function with a much higher precision that the output format. And the last rounding of that evaluation shall provide a correctly rounded result.

Let us now focus on precision requirement for correct rounding.

Figure 13.1 shows a case where the exact result f(x) is contained between two consecutive floating-point numbers (n and $n+1(ulp)$). The evaluation result $F(x)$ is also between these two floating-point numbers and d is the difference between $F(x)$ and $f(x)$. The distance between $F(x)$ and the mid-point between n and $n+1$ is e.

If $d > e$ (which is the case represented here), it is possible that $f(x)$ and $F(x)$ lies in two different sides of the mid-point, which means that rounding $F(x)$ to the nearest floating point number returns a wrong result: $n+1$ when n was expected.

To return the correctly rounded results d requires to be less than e for every possible floating-point input.

A method, suggested by Ziv [158], exists to ensure that this constraint is verified. It consists,

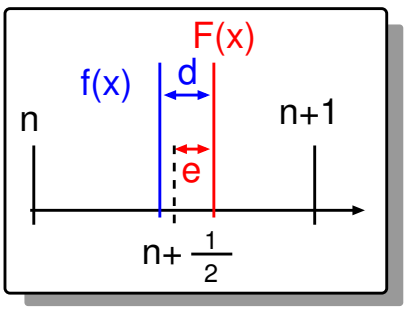


Figure 13.1: Illustration of hard-to-round case condition

for each input, in iterating over the intermediary precision, refining the approximation evaluated until correct rounding can be reaced/. For each input, a standard precision evaluation is first computed along with its overall error.

If the error is less that the distance between the result and a mid-point, the result can be returned accurately. This test is called Ziv's test. If not, the intermediary precision needs to be increased and the computation done again (along with the error). A new Ziv's test is performed and if successful the execution can stop, if not the precision needs to be increased once again and the computation started again.

An extended study on how to perform Ziv's test efficiently can be found in [44].

The finite number of possible floating-point inputs implies that it exists a precision with which every function evaluation can return the correctly rounded result. However this precision can be very large which implies a large use of memory and a long latency to reach the required precision. The precision could even overcome the capability (memory) of the machine on which the evaluation is computed.

To suppress these difficulties the project CRLIBM was launched by the PhD thesis of D. Defour [47] and continued by C. Lauter's Phd [90]. The goal was to used the hardest to round cases listed in [48, 116] to determine the largest precision required to evaluate a function correctly.

Once this precision is known the function evaluation is divided in at least two steps:

- a fast step that evaluates the approximation and the overall error at a precision close to the output format.
- an accurate step that evaluates the function up to the precision required by the hardest to round cases.

By construction the accurate step allows for correctly rounded evaluation, but its slowness makes the fast step useful in most cases. If the fast step is sufficient, meaning its Ziv's test is successful, the computation can end there, if not the code needs to branch into the slow and accurate branch.

The average execution time of the function is $T = p \times T_f + (1 - p) \times T_a$ with p the probability for the Ziv's test to be successful, T_f the latency of the fast execution path and T_a the latency of the accurate execution path (which includes a first execution of the fast path).

Even if T_a is several order of magnitude slower than T_f (which can be the case in practice), providing a fast step with a very small (1-p) reduces the effect of the accurate path latency. The fast branch is often constructed so the accurate step consequences are way under 10% time increases over the only fast branch. More details and numeric result can be found in [90].

In conclusion, an efficient way to implement a correctly rounded function is to divide between a fast branch and an accurate branch. The fast branch brings the performance and returns a result for most inputs. The accurate branch collects the ziv test failure and returns a result for the remaining inputs. It is the method we are going to use when implementing correctly rounded function in Section 5.8. It is also this method that justifies the development of a support library for multi-double precision in Section 5.4.

CHAPTER 14
Useful basic blocks for FPU design

During the design of the first version of Kalray's K1 FPU, we did not have time or resources to implement the basic integer arithmetic operators ourselves. We used the hard-macros provided by the synthesis tool. To evaluate the consequences of some of the design choices, we evaluated the area of the following operators: integer adder, leading zero counter, shifter and integer multiplier. Each evaluation considered several pairs of operator width and latency constraint. The objective was to evaluate the effect of operand sizes and timing constraint on the operator area. This section presents the result obtained and try to draw some conclusions.

All the syntheses have been performed using a 28-nm Low-Power gate library, using Cadence's RC synthesis tool.

14.1 Synthesis results

14.1.1 Integer adder

Width	Latency (ns)	Area (# gates)	Ratio ($\frac{area}{size.ratio24}$)
24-bit	2.0	116	1.0
27-bit	2.0	134	1.03
32-bit	2.0	165	1.07
54-bit	2.0	314	1.20
72-bit	2.0	460	1.32

Table 14.1: Integer adder synthesis results

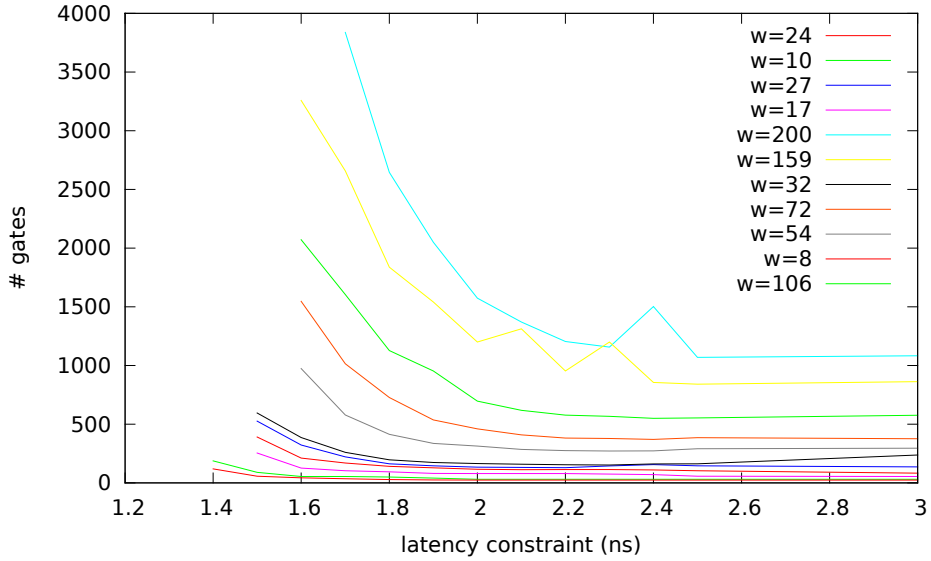


Figure 14.1: Adder area with respect to latency and width

14.1.2 Leading zero counter

Width	Latency (ns)	Area (# gates)	Ratio ($\frac{area}{size.ratio24}$)
24-bit	2.0	36	1.0
32-bit	2.0	50	1.04
48-bit	2.0	76	1.06
53-bit	2.0	94	1.18
64-bit	2.0	122	1.27
72-bit	2.0	151	1.40
96-bit	2.0	223	1.55
110-bit	2.0	240	1.45

Table 14.2: Leading Zero Counter synthesis results

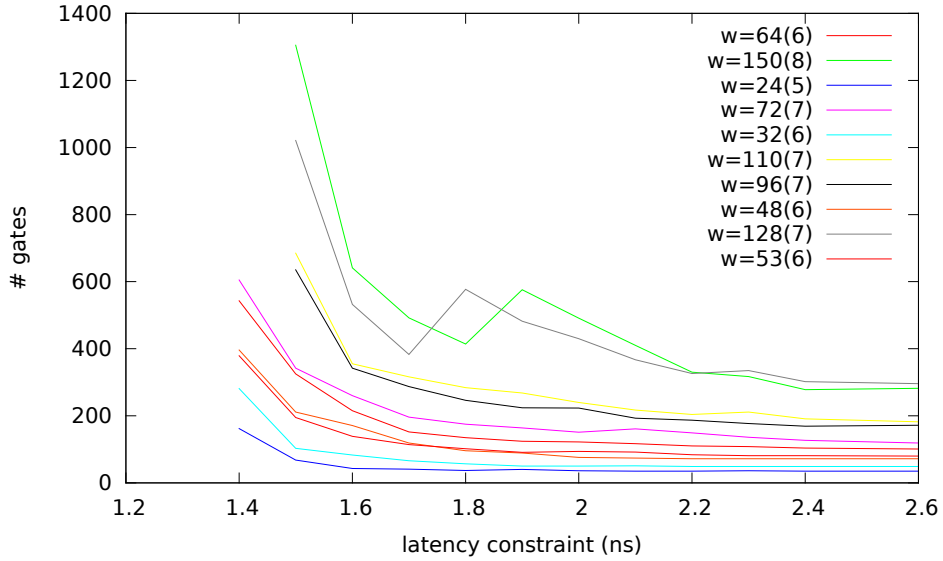


Figure 14.2: Leading Zero Count area with respect to latency and width

14.1.3 Logic shifter

Width	Latency (ns)	Area (# gates)	Ratio ($\frac{area}{size.ratio24}$)
24-bit	2.0	104	1.0
32-bit	2.0	148	1.07
48-bit	2.0	247	1.19
64-bit	2.0	362	1.31
72-bit	2.0	403	1.29
96-bit	2.0	582	1.40
128-bit	2.0	922	1.66
150-bit	2.0	1131	1.74
196-bit	2.0	1642	1.93
224-bit	2.0	2003	2.06

Table 14.3: Logic Shifter synthesis results

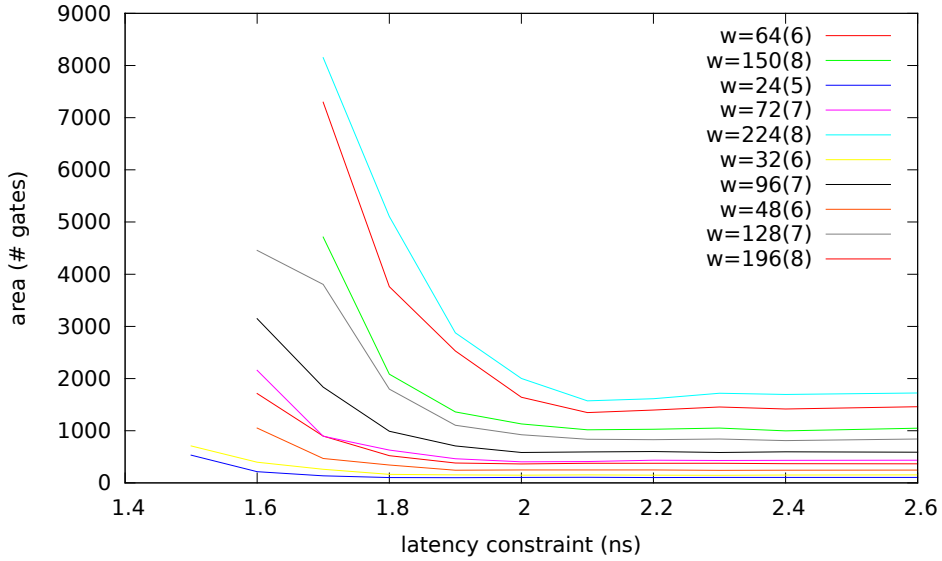


Figure 14.3: Logic shift area with respect to latency and width

14.1.4 Multiplier

Width	Latency (ns)	Area (# gates)	Ratio ($\frac{area}{width.ratio24}$)
24×24	3.0	1707	1.0
27×27	3.0	2094	0.97
32×16	3.0	1495	0.985
32×32	3.0	2851	0.94
54×27	3.0	3771	0.87
54×54	3.0	7689	0.89
64×64	3.0	10817	0.89

Table 14.4: Multiplier synthesis results

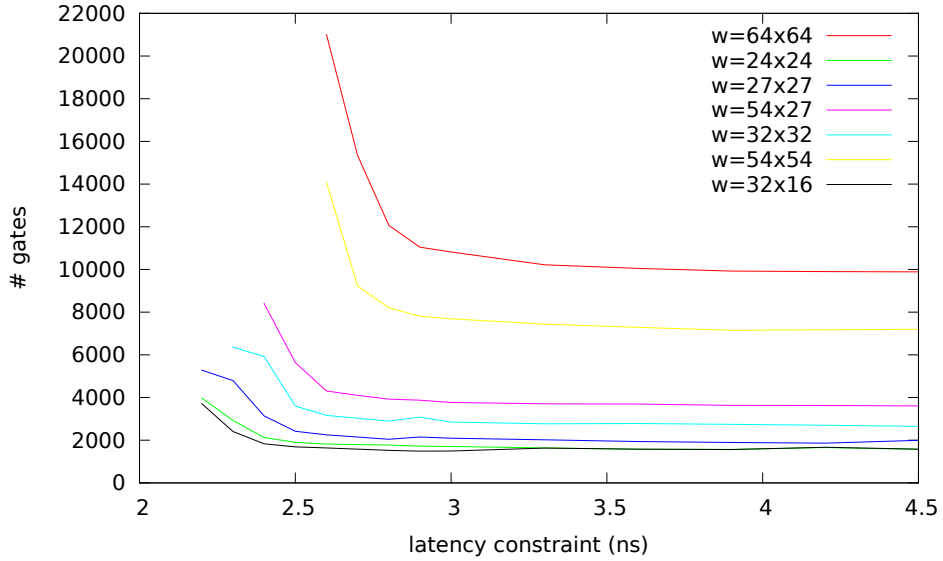


Figure 14.4: Multiplier area with respect to latency and width

14.2 Analysis and conclusion

Our first set of measures focuses on area evolution depending on the operand width. To determine this evolution, we synthesize at a constant latency: 2.0ns (adder, LZC and shifter) and 3.0ns (multiplier), for several values of the operand width parameter. The latencies have been chosen to be representative of what part of a 3.33 ns cycle the operator may occupy in a real floating-point pipeline.

The synthesis results are reported in Table 14.1 for integer adder, Table 14.2 for leading-zero counter, Table 14.3 for shifter and Table 14.4 for integer multiplier. It can be noticed that the adder area is not linear with respect to operand width: doubling the operand width of the adder results in more than doubling its area (eg: between 27 and 54 bits, the area is multiplied by 2.34). The same analysis can be said for leading zero counter and shifter.

Multiplier results exposes different properties. One important characteristic of the multipliers is that their area is approximately a linear function of the product of operand widths. Doubling the two operands widths, results in quadrupling the multiplier area. This is illustrated by Table 14.4: the multiplier area is multiplied by 3.69 between the 27×27 instance and the 54×54 instance.

An other interesting trend in multiplication results is that the relative size (ratio) decreases when the operand widths increase. With respect to unary bit multiplication, a large multiplication is less expensive than a smaller one. Not by a large factor, but this is still noticeable. The main reasons is that a bigger multiplier allows the use of more efficient compression scheme [123, 148].

Our second set of measures concerns the area evolution depending on latency constraint. Figures 14.1, 14.2, 14.3 and 14.4 respectively presents results of synthesis for integer adder, leading zero counter, shifter and multiplier. Each operator is synthesized with several latency constraints and several operand widths. The various plots are exponentially shaped: synthesizing at a lower in latency results in an exponential increase in silicon area. Some artifacts are noticeable on the plots, for example LZC width of 128-bit and 150-bit around the 1.8 ns constraint. Those can be explained by the hard-macro designs used by the synthesis tools: according to constraint and fitting measurement, the tool selects an operator implementation between several known schemes. For example, linear carry ripple adder, carry look-ahead adder, carry-select for the integer adder. It

seems that this selection, threshold-based, exposes some irregularities.

In conclusion: over constraining an operator can quickly become costly in terms of area if the exponential part of the plot is reached. For adder, LZC and shifter, doubling the operand width will more than double its area. In other word, dividing a datapath to use two halved paths rather than one full path can be help reduce the operator area. The multiplier is an exception to this rule since doubling its operand widths results in quadrupling its area. Dividing a multiplier in two half multipliers is less efficient than using a full multiplier: under light to medium latency constraints, the sum of the areas can overcome the area of the full multiplier. However smaller multipliers are less subject to important area increase when latency constraint becomes stronger.

Bibliography

[1] ARM NEON. http://www.arm.com/products/processors/technologies/neon.php.

[2] Kalray. http://www.kalray.eu/.

[3] CR-Libm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/www/crlibm/, 2010.

[4] POCL - portable computing language. http://pocl.sourceforge.net, 2013.

[5] Adapteva. Epiphany architecture reference manual. http://www.adapteva.com/uncategorized/e3-reference-manual/, 2012.

[6] European Space Agency. Planck (spacecraft). http://en.wikipedia.org/wiki/Planck_(spacecraft)#2013_data_release, 2013.

[7] Ahmet Akkas and Michael J. Schulte. A quadruple precision and dual double precision floating-point multiplier. *Proceedings of the Euromicro Symposium on Digital System Design*, 2003.

[8] AMD. Core math library. http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

[9] S.F. Anderson, J.G. Earle, R. Et Goldschmidt, and D.M. Powers. The IBM system/360 model 91: floating-point execution unit. *IBM Journal of research and development*, 11(1):34–53, 1967.

[10] ANSI/IEEE. *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559)*. 1985.

[11] ANSI/IEEE. *Standard 854-1987 Standard for radix-independent floating-point arithmetic*. 1987.

[12] ARM. ARM Floaintg Point. http://www.arm.com/products/processors/technologies/vector-floating-point.php.

[13] Tom Asprey, Gregory S Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance features of the PA7100 microprocessor. *Micro, IEEE*, 13(3):22–35, 1993.

[14] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[15] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of AES on 32-bit platforms. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[16] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, pages 213–222, 1997.

[17] Brandon Blodget, Christophe Bobda, Michael Hübner, and Adronis Niyonkuru. Partial and dynamically reconfiguration of Xilinx Virtex-II FPGAs. In *Field Programmable Logic and Application*, pages 801–810. 2004.

[18] Nicolas Boullis and Armaud Tisserand. Some optimizations of hardware multiplication by constant matrices. In *Proceedings of the 16th Symposium on Computer Arithmetic (ARITH)*, pages 20–27, 2003.

[19] Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial L-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 169–176. IEEE, 2007.

[20] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):236–256, 2006.

[21] Javier D. Bruguera and Tómas Lang. Rounding in floating-point addition using a compound adder. *University of Santiago de Compostela, Spain Internal Report*, 2000.

[22] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[23] Fabio Campi, Antonio Deledda, Matteo Pizzotti, Luca Ciccarelli, Pierluigi Rolandi, Claudio Mucci, Andrea Lodi, Arseni Vitkovski, and Luca Vanzolini. A dynamically adaptive DSP for heterogeneous reconfigurable platforms. In *Design, Automation & Test in Europe (DATE'07)*, pages 1–6, 2007.

[24] Kuan-Hsu Chen, Bor-Yeh Shen, and Wuu Yang. An automatic superword vectorization in llvm. In *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, pages 19–27, 2010.

[25] Sylvain Chevillard. *Évaluation efficace de fonctions numériques – Outils et exemples*. PhD thesis, École normale supérieure de Lyon – Univeristé de Lyon, 2009.

[26] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th IEEE Symposium in Computer Arithmetic (ARITH)*, pages 169–176, 2009.

[27] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In *Mathematical Software - ICMS*, pages 28–31. 2010.

[28] William J. Cody. *Software Manual for the Elementary Functions*. Prentice-Hall series in Computational Mathematics. 1980.

[29] William J Cody. Implementation and testing of function software. In *Problems and Methodologies in Mathematical Software Production*, pages 24–47. 1982.

[30] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Barra: A parallel functional simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.

[31] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Euro-Par 2009 - Parallel Processing Workshops*, pages 46–55. 2010.

[32] Sylvain Collange and Alexandre Kouyoumdjian. Affine Vector Cache for memory bandwidth savings. Technical report, INRIA, December 2011.

[33] Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems*, 1:145–204, 1996.

[34] Marius Cornea. Proving the IEEE correctness of iterative floating-point square-root, divide and remainder algorithms. *Intel Technology Journal*, 1998.

[35] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific computing on Itanium-based systems*. 2002.

[36] ILOG CPLEX. 11.0 user's manual. *ILOG SA, Gentilly, France*, 2007.

[37] Merrick Darley, Bill Kronlage, David Bural, Bob Churchill, David Pulling, Paul Wang, Rick Iwamoto, and Larry Yang. The TMS390C602A floating-point coprocessor for sparc systems. *Micro, IEEE*, 10(3):36–47, 1990.

[38] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transaction on Mathematical Software*, 37(1):2:1–2:20, January 2010.

[39] Jean-Pierre David. *Architecture synchronisée par les données pour système reconfigurable*. PhD thesis, Université Catholique de Louvain, June 2002.

[40] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. 2013.

[41] Florent De Dinechin. ANR Metalibm. http://www.metalibm.org.

[42] Florent De Dinechin, David Defour, and Christoph Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research Report RR-5137, INRIA, 2004.

[43] Florent De Dinechin, Jérémie Detrey, Octavian Cret, and Radu Tudoran. When FPGAs are better at floating-point than microprocessors. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 260–260, 2008.

[44] Florent De Dinechin, Christoph Lauter, Jean-Michel Muller, Serge Torres, et al. On ziv's rounding test. 2012.

[45] Florent de Dinechin and Bogdan Pasca. Large multipliers with fewer DSP blocks. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 250–255, 2009.

[46] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *Design & Test of Computers, IEEE*, 28(4):18–27, 2011.

[47] David Defour. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, 2003.

[48] David Defour, Guillaume Hanrot, Vincent Lefèvre, Jean-Michel Muller, Nathalie Revol, and Paul Zimmermann. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical algorithms*, 37(1-4):367–375, 2004.

[49] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488, 2011.

[50] W. S. Dorn. Generalizations of Horner's rule for polynomial evaluation. *IBM Journal of Research and Development*, 6(2):239–245, 1962.

[51] Miloš D. Ercegovac and Tomás Lang. *Digital arithmetic*. 2003.

[52] Miloš D. Ercegovac, Tomás Lang, Jean-Michel Muller, and Arnaud Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7):628–637, 2000.

[53] Otto Esko, Pekka Jääskeläinen, Pablo Huerta, Carlos S. de La Lama, Jarmo Takala, and Jose Ignacio Martinez. Customized exposed datapath soft-core design flow with compiler support. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*, pages 217–222, 2010.

[54] Gerald Estrin. Organization of computer systems: The fixed plus variable structure computer. In *Western Joint IRE-AIEE-ACM Computer Conference*, pages 33–40, 1960.

[55] FIPS. Publication 180-4, Secure Hash Standard (SHS). 2012.

[56] Pierre Fortin, Mourad Gouicem, and Stef Graillat. Towards solving the table maker's dilemma on GPU. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 407–415, 2012.

[57] Python Software Foundation. Python language reference, version 2.7. http://www.python.org.

[58] Python Software Foundation. Python language reference, version 2.7. http://docs.python.org/2/library/math.html.

[59] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.

[60] Guenter Gerwig, Eric M. Schwarz, and Ronald M. Smith Sr. Fused multiply add split for multiple precision arithmetic, 2007.

[61] Guenter Gerwig, Holger Wetter, Eric M Schwarz, Juergen Haess, Christopher A Krygowski, Bruce M Fleischer, and Michael Kroener. The IBM eServer z990 floating-point unit. *IBM Journal of Research and Development*, 48(3.4):311–322, 2004.

[62] GNU. The GNU C library (glibc). http://www.gnu.org/software/libc.

[63] Mustafa Gök and Metin Mete Özbilen. Multi-functional floating-point MAF designs with dot product support. *Microelectronics journal*, pages 30–43, 2008.

[64] Shay Gueron. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption*, Lecture Notes in Computer Science, pages 51–66. 2009.

[65] David L Harris, Stuart F Oberman, and Mark A Horowitz. SRT division architectures and implementations. In *13th IEEE Symposium on Computer Arithmetic*, pages 18–25, 1997.

[66] Mark Harris. GPGPU: General-purpose computation on GPUs. *SIGGRAPH, GPGPU Course*, 2005.

[67] John Harrison. HOL light: An overview. In *Theorem Proving in Higher Order Logics*, pages 60–66. 2009.

[68] Scott Hauck, Matthew M Hosler, and Thomas W Fry. High-performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):138–147, 2000.

[69] Yedidya Hilewitz and Ruby B Lee. A new basis for shifters in general-purpose processors for existing and advanced bit manipulations. *IEEE Transactions on Computers*, 58(8):1035–1048, 2009.

[70] Yedidya Hilewitz, Zhije Jerry Shi, and Ruby B. Lee. Comparing fast implementations of bit permutation instructions. In *Conference on Signals, Systems and Computers (Asilomar 38th)*, pages 1856–1863, 2004.

[71] Libo Huang, Li Shen, Kui Dai, and Zhiying Wang. A new architecture for multiple-precision floating-point multiply-add fused unit design. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 69–76, 2007.

[72] IBM. Accurate portable mathlib. http://oss.software.ibm.com/mathlib.

[73] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008.

[74] Intel. The Intel(R) Xeon Phi(TM) product family. http://intel.com/xeonphi, 2013.

[75] Intel. Math kernel library. http://software.intel.com/en-us/intel-mkl, 2014.

[76] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[77] ISO. Programming languages-C. Technical Report ISO/IEC 9899:201x, ISO/IEC, 2011.

[78] Pekka Jääskeläinen, Carlos S. de La Lama, Pablo Huerta, and Jarmo Takala. OpenCL-based design methodology for application-specific processors. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 223–230, 2010.

[79] Claure-Pierre Jeannerod and Guillaume Revy. FLIP: Floating-point library for integer processors. http://flip.gforge.inria.fr, 2012.

[80] Weihua Jiang, Chao Mei, Bo Huang, Jianhui Li, Jiahua Zhu, Binyu Zang, and Chuanqi Zhu. Boosting the performance of multimedia applications using SIMD instructions. In *Compiler Construction*, pages 59–75, 2005.

[81] Mioara Maria Joldes. *Approximations polynomiales rigoureuses et applications*. PhD thesis, École Normale Supérieure de Lyon, September 2011.

[82] Kalray. MPPA manycore. http://www.kalray.eu/IMG/pdf/FLYER_MPPA_MANYCORE.pdf.

[83] Peter Kornerup and Jean-Michel Muller. Extending the range of the Cody and Waite range reduction method. 2005.

[84] Peter Kornerup and Jean-Michel Muller. Choosing starting values for certain Newton–Raphson iterations. *Theoretical computer science*, 351(1):101–110, 2006.

[85] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 21–30, 2006.

[86] Olga Kupriianova and Christoph Lauter. Metalibm. http://lipforge.ens-lyon.fr/www/metalibm, 2014.

[87] Ohsang Kwon, Earl E Swartzlander Jr, and Kevin Nowka. A fast hybrid carry-lookahead/carry-select adder design. In *Proceedings of the 11th Great Lakes symposium on VLSI*, pages 149–152, 2001.

[88] Tomás Lang and Javier D. Bruguera. Floating-point fused multiply-add with reduced latency. *Proceedings of the IEEE International Conference on Computer Design : VLSI in Computers and Processors*, 2002.

[89] Christoph Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR-5702, INRIA, September 2005.

[90] Christoph Q. Lauter. *Arrondi correct de fonctions mathématiques: fonctions univariées et bivariées, certification et automatisation*. PhD thesis, 2008.

[91] Kyung T. Lee and Kevin .J. Nowka. 1 GHz leading zero anticipator using independent sign-bit determination logic. In *Symposium on VLSI Circuits*, pages 194–195, 2000.

[92] Vincent Lefèvre. Multiplication by an integer constant. *Rapport de recherche INRIA*, 2001.

[93] Adam Levinthal and Thomas Porter. Chap - a SIMD graphics processor. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 77–82, 1984.

[94] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[95] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 347–358, 2012.

[96] Andrea Lodi, Claudio Mucci, Massimo Bocchi, Andrea Cappelli, Mario De Dominicis, and Luca Ciccarelli. A multi-context pipelined array for embedded systems. In *International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 1–8, 2006.

[97] Robin Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, January 2003.

[98] Nicolas Louvet, Jean-Michel Muller, and Adrien Panhaleux. Newton-raphson algorithms for floating-point division using an FMA. In *Proceedings of the IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 200–207, 2010.

[99] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: general-purpose computation on graphics hardware. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 208, 2006.

[100] Slobodan Lukovic and Leandro Fiorin. An automated design flow for NoC-based MP-SoCs on FPGA. In *19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'08)*, pages 58–64, 2008.

[101] David R. Lutz. Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines. In *IEEE Symposium on Computer Arithmetic (ARITH 20)*, pages 123–128, 2011.

[102] Andrew Makhorin. GLPK (GNU Linear Programming Kit), 2006.

[103] Maplesoft. Maple 17, 2013.

[104] Peter Markstein. *IA-64 and elementary functions: speed and precision*, volume 3. 2000.

[105] Peter Markstein. Software division and square root using Goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6)*, volume 123, pages 146–157, 2004.

[106] Peter W. Markstein. Computation of elementary functions on the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, 1990.

[107] Merkert. GCC bug 34678: Optimization generates incorrect code with -frounding-math option.
http://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678.

[108] Robert K. Montoye, Erdem Hokenek, and Stephen L. Runyon. Design of the IBM Risc System/6000 floating-point execution unit. *IBM Journal of Research and Development*, January 1990.

[109] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[110] MPFR. http://www.mpfr.org/.

[111] Claudi Mucci. *Software Tools for Embedded Reconfigurable Processors*. PhD thesis, Universitá degli Studi di Bologna, 2006.

[112] Claudio Mucci, Carlo Chiesa, Andrea Lodi, Mario Toma, and Fabio Campi. A C-based algorithm development flow for a reconfigurable processor architecture. In *International Symposium on System-on-Chip*, pages 69–73, 2003.

[113] Claudio Mucci, Davide Rossi, Fabio Campi, Matteo Pizzotti, Luca Perugini, Luca Vanzolini, Tommaso De Marco, and Massimiliano Innocenti. The Dream digital signal processor. In *Dynamic System Reconfiguration in Heterogeneous Platforms*. 2009.

[114] Jean-Michel Muller. On the definition of ulp(x). Technical Report RR-5504, INRIA, February 2005.

[115] Jean-Michel Muller. *Elementary functions: algorithms and implementation*. 2006.

[116] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Nathalie Revol, Guillaume Melquiond, Damien Stehlé, and Serge Torres. *Handbook of floating-point arithmetic*. Springer, 2010.

[117] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. 2011.

[118] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the GCC Developers Summit*, pages 105–118, 2004.

[119] Dorit Nuzman and Ayal Zaks. Autovectorization in GCC–two years later. In *Proceedings of the GCC Developers Summit*, pages 145–158, 2006.

[120] Stuart F. Oberman and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, 1997.

[121] Stuart F. Oberman and Michael J. Flynn. Reducing the mean latency of floating-point addition. *Theoretical Computer Science*, 1998.

[122] Stuart F. Oberman and Michael Y. Siu. A high-performance area-efficient multifunction interpolator. In *17th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 272–279. IEEE, 2005.

[123] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, 45(3):294–306, 1996.

[124] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 3(1):1–14, 1999.

[125] D.C Opferman and N. T. Tsao-Wu. On a class of rearrangeable switching networks, part 1 : Control algorithm. *The Bell System Technical Journal*, 50, May-June 1971.

[126] Christof Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, University of Essen, 1994.

[127] Hadi Parandeh-Afshar, Arkosnato Neogy, Philip Brisk, and Paolo Ienne. Compressor tree synthesis on commercial high-performance FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 2011.

[128] Hadi Parandeh-Afshar, Grace Zgheib, Philip Brisk, and Paolo Ienne. Reducing the pressure on routing ressources of FPGAs with generic logic chains. FPGA'11, 2011.

[129] Jose-Alejandro Pineiro, Stuart F. Oberman, Jean-Michel Muller, and Javier D. Bruguera. High-speed function approximation using a minimax quadratic interpolator. *IEEE Transactions on Computers*, 54(3):304–318, 2005.

[130] Nhon T Quach and Michael J Flynn. *Leading one prediction–Implementation, generalization, and application*. Computer Systems Laboratory, Stanford University, 1991.

[131] Eric C. Quinell. *Floating-Point Fused Multiply-Add Architectures*. PhD thesis, The University of Texas at Austin, May 2007.

[132] Eric C. Quinell, Earl E. Swartzlander Jr., and Carl Lemonds. Three-path fused multiply-adder circuit, 2011. US Patent 8037118B2.

[133] Chester Rebeiro, David Selvakumar, and A.S.L. Devi. Bitslice implementation of AES. In *Cryptology and Network Security*. 2006.

[134] Eugene Remez. Sur le calcul effectif des polynomes d'approximation de Tchebichef. *CR Académie des Sciences, Paris*, 199:337–340, 1934.

[135] Guillaume Revy. *Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, December 2009.

[136] James E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7(3):218–222, 1958.

[137] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient rijndael encryption implementation with composite field arithmetic. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 171–184, 2001.

[138] Hani H. Saleh and Earl E. Swartzlander Jr. A floating-point fused dot-product unit. In *IEEE International Conference on Computer Design (ICCD'08)*, pages 427–431, 2008.

[139] Martin S. Schmookler and Kevin J. Nowka. Leading zero anticipation and detection - a comparison of methods. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 7–12, 2001.

[140] Eric M. Schwarz, Martin M. Schmookler, and Son Dao Trong. FPU implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7):825–836, 2005.

[141] Avinash Sodani. Race to exascale: Opportunities and challenges. In *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011.

[142] Federal Information Processing Standards. FIPS 197: Announcing the advanced encryption standard (AES), 2001.

[143] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[144] Ping-Tak Peter Tang. Table-driven implementation of the exponential function in ieee floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):144–157, 1989.

[145] Alexandre F Tenca. Multi-operand floating-point addition. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 161–168. IEEE, 2009.

[146] Russell G Tessier. *Fast place and route approaches for FPGAs*. PhD thesis, Massachusetts Institute of Technology, 1998.

[147] Keith D Tocher. Techniques of multiplication and division for automatic binary computers. *The Quarterly Journal of Mechanics and Applied Mathematics*, 11(3):364–384, 1958.

[148] Whitney J. Townsend, Earl E. Swartzlander Jr, and Jacob A. Abraham. A comparison of Dadda and Wallace multiplier delays. In *Optical Science and Technology, SPIE's 48th Annual Meeting*, pages 552–560, 2003.

[149] Kishor S. Trivedi and Miloš D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Transactions on Computers*, 100(7):681–687, 1977.

[150] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated annealing*. 1987.

[151] David Villeger and Vojin G Oklobdzija. Evaluation of booth encoding techniques for parallel multiplier implementation. *Electronics Letters*, 29(23):2016–2017, 1993.

[152] Corinna Vinschen and Jeff Johnston. Newlib. https://sourceware.org/newlib/.

[153] Wikipedia. Advanced Encryption Standard. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[154] Wikipedia. Floating-point. http://en.wikipedia.org/wiki/Floating_point#History.

[155] Inc. Wolfram Research. Mathematica edition: Version 8.0, 2010.

[156] Mei Xiao-Lu. Leading zero anticipation for latency improvement in floating-point fused multiply-add units. In *6th International Conference On ASIC (ASICON'05)*, volume 1, pages 53–56, 2005.

[157] Ge Zhang, Zichu Qi, and Weiwu Hu. A novel design of leading zero anticipation circuit with parallel error detection. In *IEEE International Symposium on Circuits and Systems (ISCAS'05)*, pages 676–679, 2005.

[158] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software (TOMS)*, 17(3):410–423, 1991.