Decimal Engine for Energy-Efficient Multicore Processors

Alberto Nannarelli
DTU Compute, Technical University of Denmark
Kongens Lyngby, Denmark
E-mail: alna@dtu.dk

***Abstract*—Prior work demonstrated the use of specialized processors, or accelerators, be energy-efficient for binary floating-point (BFP) division and square root, and for decimal floating-point (DFP) operations. In the dark silicon era, where not all the circuits on the die can be powered simultaneously, we propose a hybrid BFP/DFP engine to perform BFP division and DFP addition, multiplication and division. The main purpose of this engine is to offload the binary floating-point units for this type of operations and reduce the latency for decimal operations, and power and temperature for the whole die.**

I. Introduction

Contemporary and future multicore System-on-Chips (SoCs) containing billions of transistors might face the *"Dark Silicon"* challenge. With technology scaling the number of cores, or functional units, increases at each generation, but the power necessary to power the whole chip has increased to a point that it is no longer possible to power all the circuits on the die simultaneously.

In [1] the author indicates several design scenarios in the dark silicon era. One of these, called *"specialized"* consists in replacing general-purpose cores with specialized, yet re-configurable, cores that, when running special applications, are more power efficient than general-purpose CPUs.

Decimal arithmetic operations are necessary for accounting and financial applications since the binary floating-point (BFP) representation of decimal fractions not always rounds correctly [2], and in finance every cent counts. For these rounding problems, decimal fractions are treated by software functions that are executed in hundreds of machine cycles. IBM was the first company to include a decimal floating-point (DFP) hardware unit in the chips of their modern servers [3].

Recently, Fujitsu announced the new Sparc64 X processor which includes an accelerator, called *"software on chip (SWoC)"*, to speed up operations in cryptography and decimal calculations [4].

In this work, we propose a re-configurable processor for DFP operations (addition, multiplication, and division) for the *decimal64* IEEE 754 format [5]. Additionally, the processor shares part of the DFP hardware specialized for division by digit-recurrence [6] with the hardware necessary to implement BFP division.

978-1-4799-6016-3/14/$31.00

BFP division is normally executed in hardware by implementing two classes of algorithms:

1) digit-recurrence: [7], [8], [9].
2) multiplicative algorithms (e.g., Newton-Raphson): [10], [11], [12].

In [13], the authors show that a radix-16 digit-recurrence unit is much more energy-efficient than division by multiplication executed on Fused Multiply-Add (FMA) units.

The processor datapath handles both decimal numbers (represented in binary-coded decimal, or BCD) and binary numbers. In summary, the processor can execute DFP addition, DFP multiplication, DFP division and BFP division.

Our motivation is to have energy-efficient execution of FP operations for decimal calculations and for BFP division.

The main contributions of this work are:

- We provide an efficient implementation of the three most common decimal operations; addition (and subtraction), multiplication, and division, by reusing common blocks necessary to process decimal operands, encoded as BCD.
- Because digit-recurrence radix-10 (decimal) division is quite similar to radix-16 (binary) division, we share part of the division hardware to obtain a dual-radix digit-recurrence division unit. In this way, we can avoid implementing a BFP digit-recurrence division unit in the SoC, or avoid using the BFP multipliers, or the FMAs, for division.
- We show that by off-loading the BFP-units, we get better performance for DFP operations and save power for both DFP operations and BFP division.

With respect to the DFP-unit of [3], our processor implements a different division algorithm which allow using part of the datapath to implement binary division.

The paper is organized as follows. In Sec. II, we describe the algorithms implemented for the DFP operations and BFP. In Sec. III, we illustrate the architecture of the processor and how the different operations are executed on it. In Sec. IV, we show the results of the implementation of the processor and we compare its performance with that of a BFP-FMA executing the same operations. In Sec. V, we draw some conclusions and point to future work.

II. Operations and Algorithms

In this section, we describe the algorithms chosen to implement the processor's operations. Division algorithms for pre-

cisions of more than 16–20 bits are iterative and implemented by a recurrence. BFP multiplication is implemented in parallel multipliers, pipelined to guarantee the necessary throughput.

For DFP multiplication (DFP-mul) the hardware resources needed to implement a parallel multiplier are quite significant, and the frequency/importance of decimal multiplication does not justify the use of so much area on the die. Therefore, we opted for an iterative implementation of DFP-mul: computing one partial-product (PP) per clock cycle. We use the same recurrence used for division.

The IEEE standard 754 describes three types of decimal formats: significands with precision of 7, 16 and 34 decimal digits and base-10 exponents of 8, 10 and 14 bits (formats *decimal32*, *decimal64*, and *decimal128* respectively) [5]. Differently from BFP, DFP operands are not normalized. The significand of DFP numbers can be represented in two different encodings: Densely Packed Decimal (DPD) format or Binary Integer Decimal (BID) format. In the DPD format, a word representing the significand of a decimal number is divided into 10 bit fields, called *"declets"*, representing 3 decimal digits each. The conversion from declet to BCD, and vice versa, can be implemented with simple logic [14].

Our processor supports the *decimal64* format (not fully compliant) and the *binary64* (double-precision) format for BFP division (BFP-div). Moreover, we assume the conversion DPD to BCD be done outside our processor. We plan to lift this restriction in future work.

We assume, in the following, the X DFP/BFP number be represented by sign, exponent and signicand (or magnitude):
$(S_X,\ E_X,\ M_X) \quad : \quad X = (-1)^{S_X} \times M_X \times B^{E_X}$
where $B = 2$ for BFP and $B = 10$ for DFP.

A. DFP Addition

The addition of two floating-point numbers requires the alignment of the decimal point of the significands. In BFP, this can be achieved by looking at the exponents of the two BFP numbers and by shifting to the right the significand of the smallest number by as many positions as the exponent difference. In DFP, as the significands are not normalized, in addition to the exponent comparison, it is also necessary to detect the position of the first non-zero digit [15].

Once the significands are aligned, and one is complemented if the effective operation is subtraction, the significand addition is performed. The result (sum) is eventually shifted, rounded and packed (conversion BCD to DPD).

B. DFP Multiplication

For DFP-mul the handling of sign and exponent is trivial. The significands can be directly multiplied:

$$M_Z = M_X \times M_Y \ .$$

Both multiplicand (M_X) and multiplier (M_Y) are not normalized (may contain leading 0s).

The iterative algorithm to compute the product is:

$$\begin{aligned} &W[0] = 0 \\ &W[j+1] = 10^{-1}(W(j) + M_X \cdot 10^n M_{Y\,j}) \\ &M_Z = W[n] \end{aligned}$$

for $j = 0, 1, \ldots, n-1$ ($n = 16$ for *decimal64*).

To avoid complicated multiples of M_X, the digits of M_Y are recoded in a way that only the multiples x, $2x$ and $5x$ (and the respective negative multiples) are necessary [16].

To speed-up the iteration, the partial product is accumulated carry-save: 1 BCD plus 1 carry bit for each digit.

If the product is represented by more than 16 digits, the number has to be normalized and rounded.

C. DFP/BFP Division

The algorithm for digit-recurrence division converges if the following condition on the dividend x and divisor d holds

$$x < \rho d \tag{1}$$

where ρ is the redundancy factor and depends on the radix and the quotient-digit set (see [6] for more detail). While for BFP, the significands are normalized, for DFP-div, we need to shift the operands to satisfy (1) before starting the recurrence. The expressions for the digit-recurrence iteration for the radix-10 case are

$$\begin{aligned} v[j] &= 10w[j-1] - q_{Hj}(5d) \\ w[j] &= v[j] - q_{Lj}d \end{aligned}$$

with $q_{Hj} \in \{-1, 0, 1\}$ and $q_{Lj} \in \{-2, -1, 0, 1, 2\}$ for a redundancy factor $\rho_{10} = 7/9$.

For the radix-16 (BFP) case the digit-recurrence iteration are

$$\begin{aligned} v[j] &= 16w[j-1] - q_{Hj}(4d) \\ w[j] &= v[j] - q_{Lj}d \end{aligned}$$

with $q_{Hj} \in \{-2, -1, 0, 1, 2\}$ and $q_{Lj} \in \{-2, -1, 0, 1, 2\}$ for a redundancy factor $\rho_{16} = 10/16$.

Therefore, the two recurrences can be combined into

$$v[j] = rw[j-1] - q_{Hj}(kd) \tag{2}$$
$$w[j] = v[j] - q_{Lj}d \tag{3}$$

with quotient digit selection functions

$$q_H = SEL_H(\widehat{rw}, \widehat{d}) \tag{4}$$
$$q_L = SEL_L(\widehat{v}, \widehat{d}) \tag{5}$$

and quotient digit $q_j = kq_{Hj} + q_{Lj}$. The switch between the two radices is performed by setting a bit R such that

when $R = 0 \quad \rightarrow \quad r = 16$ and $k = 4 \quad$ (radix-16)
when $R = 1 \quad \rightarrow \quad r = 10$ and $k = 5 \quad$ (radix-10)

To ensure convergence, the recurrence is initialized as

$$w[0] = x/r^2 \ .$$

Detail on this dual-radix division algorithm is given in [17].

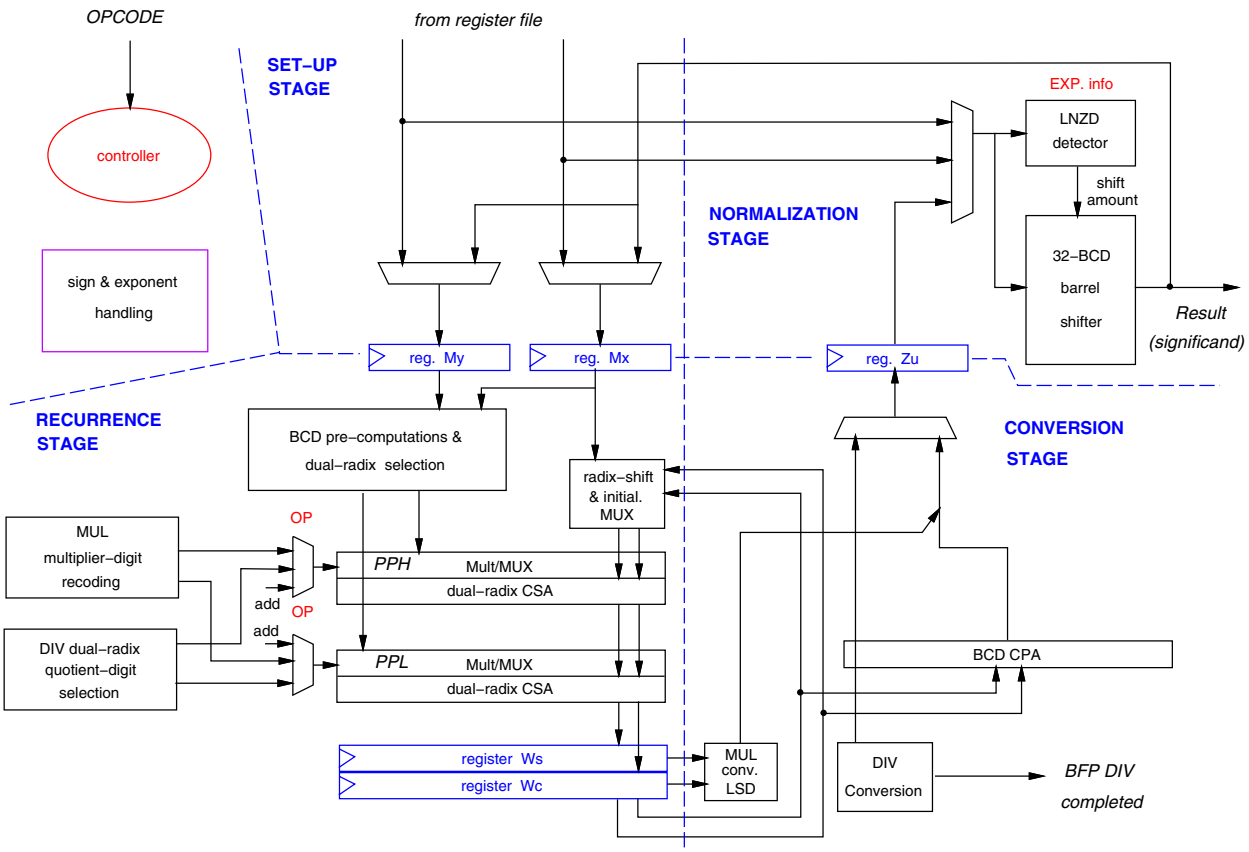

Fig. 1. Block diagram of the processor.

III. Processor Architecture

The block diagram on the proposed processor is depicted in Fig. 1. The figure does not display all the connections and control signals to be easily readable.

The processor consists in five different parts:

- The operands **set-up** stage handles the significands to make them ready for the operations.
- The **recurrence** stage is the core of the iterative operations (DFP-mul and BFP/DFP-div).
- The **conversion** stage performs the digit conversion necessary for division and executes carry-propagate addition necessary for DFP-add and DFP-mul.
- The **normalization** stage normalizes BCD significands, if necessary.
- The controller provides the control signals to the datapath depending on the type of operations (specified by an opcode) and the flags generated by the *sign & exponent handling* unit. The latter, computes the sign and the exponent of the BFP/DFP results as well.

We now describe how the four operations are implemented in the processor. For simplicity, we start with DFP-mul, then we describe BFP-div, DFP-div, and DFP-add.

A. Implementation of DFP-mul

The multiplication of the significands can be started as the operands enter the processor. Therefore stage **set-up** is bypassed while sign and exponent are computed.

In stage **recurrence**, the operation requires 16 iterations (one for each BCD digit of the multiplier M_y). Each digit of M_y is recoded in two parts $y_i = y_{Hi} + y_{Li}$ with $y_H \in \{0, 5, 10\}$ and $y_L \in \{-2, 1, 0, 1, 2\}$ according to Table I. With this recoding, only the carry-save multiples $5x$ and $2x$ need to be computed (in block *BCD pre-computations*).

The two parts of the partial-product at iteration j (PPH_j and PPL_j) are selected among the pre-computed values through multiplexers and then added by the two dual-radix carry-save adders (CSA) and stored carry-save in the registers Ws and Wc.

The dual-radix CSA must handle both BCD carry-save addition and radix-16 (blocks of 4 bits) carry-save addition (necessary for BFP-div). A scheme of the dual-radix CSA is shown in Fig. 2 for one digit, we indicate with $x_{(i)}$ the digit of weight r^{-i}. A signal R selects the radix.

y_i	y_H	y_L
0	0	0
1	0	1
2	0	2
3	5	-2
4	5	-1

y_i	y_H	y_L
5	5	0
6	5	1
7	5	2
8	10	-2
9	10	-1

TABLE I
Recoding of digits for DFP-mul.

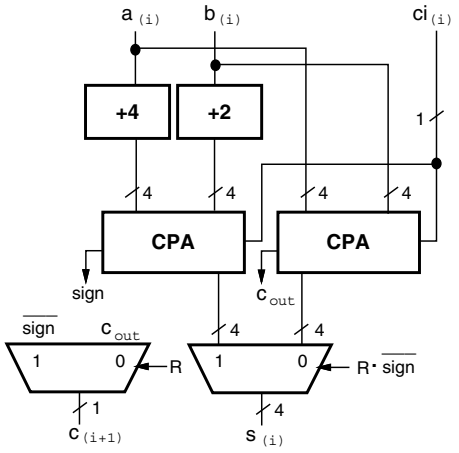

Fig. 2. Scheme of radix-10/radix-16 CSA (one digit).

To avoid the recurrence datapath increasing in size, the digit in the least-significant position (LSD) is shifted out of the recurrence and converted from carry-save to BCD. Consequently, the recurrence datapath necessary for DFP-mul is 16 digit. However, to accommodate DFP-div, 18 digits are implemented.

Once the 16 PPs have been accumulated in Ws/Wc, the upper part of the product in BCD carry-save format is converted in BCD format by the BCD carry-propagate adder (CPA) of **conversion** stage and combined to the least-significant part extracted during the iterations.

If the product exceeds the 16 digits of *decimal64* it has to be normalized and rounded in the **normalization** stage.

A total number of 20 cycles are required to implement DFP-mult in the processor.

B. Implementation of BFP-div

The algorithm for BFP-div is implemented as described in [17]. Because BFP operands are normalized[1], in the processor of Fig. 1, the stage **set-up** is bypassed. Moreover, the result is normalized and rounded in the **conversion** stage, and therefore, the **normalization** stage is not used for this operation. The significand divisor is stored in register M_y.

In the recurrence, 4 bits of the quotient are computed each iteration by the *quotient-digit selection* block. Then, the partial reminder, or residual, is computed in the recurrence datapath. In parallel, on-the-fly, the quotient-digits are converted from signed-digit to conventional binary representation in the *DIV conversion* unit.

For set-up, initialization and rounding 3 cycles are required, plus 15 cycles to compute the 15 digits necessary for *binary64* operands, the total number of cycles is 18 cycles.

C. Implementation of DFP-div

Differently from BFP-div, for DFP division, before starting the operation on significands, dividend and multiplicand have to be shifted to ensure convergence. Therefore, the significand M_y (divisor d) is shifted to the left-most position in the **normalization** stage and the significand M_x (dividend x) is shifted such that $w[0] = M_x/100 < M_y$ (two digits to the right of M_y). These normalizations are performed in the **normalization** stage in two iterations. The 16+2 BCD digits required for the recurrence, determine the width of the datapath: 18 BCDs = 72 bits.

The Leading Non-Zero (digit) Detector (LNZD) detects the position of the leading non-zero digit and its output controls the BCD barrel-shifter which shifts the operand. Clearly, in parallel, the exponent of the quotient is adjusted accordingly. To compute the 16 digits plus the rounding digit, 18 iterations are required. As for BFP-div, the quotient-digits are determined in the *quotient-digit selection* block and the on-the-fly conversion is done in the *DIV conversion* unit.

The **normalization** stage is used in case the first quotient-digit is zero due to the initial shifting of M_x.

The total number of cycles needed for DFP-div is 23 (the longest latency).

D. Implementation of DFP-add

The DFP-add is implemented without iterations in the processor. However, the **normalization** stage is used twice at the beginning of the operation. The barrel-shifter only shifts to the left. Therefore, to align the significands, the one with the largest exponent is shifted to the left-most position, while the other is left-shifted to the left-most minus the difference of the exponents position. In this way, we can implement right-shifting of the operand with the smallest exponent.

Once the initial shifting is done, and the operands are stored in registers Mx and My in Fig. 1, the addition is performed by adding to the largest operand (stored in Mx) either M_y or $-M_y$ (depending on the effective operation) which is routed to the CSA through the Mult/Mux used for PPL in DFP-mult. In the next cycle, the carry-save sum is converted to BCD in the *BCD CPA*. The result is not fully compliant with the IEEE standard 754, because for exact (not rounded) results, the representation must be done with the *preferred exponent* [5].

The total latency for DFP-add is 5 cycles.

IV. Implementation and Results

The processor is implemented in a 40 nm commercial library of standard cells designed for low power. We synthesized and laid-out the circuit with a target frequency of 25 FO4 delay, which is typical of aggressive FP-unit design, corresponding to a clock period of $1.2\ ns$ in our library.

We compare our hybrid DFP/BFP processor, called **UniPro** in the following, with a FMA unit typical of contemporary processors and GPUs. The FMA is the one designed in [13] and supports BFP-div (by the Newton-Raphson method). The FMA was re-synthesized in the new library with the same constraints as for UniPro.

The results of the implementation of the FMA and UniPro are displayed in Table II. In addition to the well knows metrics,

[1]The unit does not handle *sub-normals*.

Unit	Area NAND2	T_{clock} $[ns]$	Operation	Cycles	Latency $[ns]$	P_{ave} $[mW]$	E_{op} $[pJ]$
UniPro	38,000	1.20	DFP-add	5	6.0	13.58	81.5
			DFP-mul	20	24.0	48.92	1174.0
			DFP-div	25	30.0	21.42	642.5
			BFP-div	18	21.6	17.92	387.0
FMA	81,500	1.20	BFP-fused	4	4.80	34.92	167.6
			BFP-mul			33.42	160.4
			BFP-add			19.92	95.6
			BFP-div	26	31.20	51.83	1617.2

P_{ave} is average power measured at 833 MHz ($f = 1/T_C$).

TABLE II
RESULTS OF IMPLEMENTATIONS OF UNIPRO AND BFP-FMA.

Function	Max.	Median
add64	133	71
div64	266	171
mul64	132	69

TABLE III
CLOCK CYCLE COUNTS FROM A SUBSET OF *decimal64* ARITHMETIC FUNCTIONS (SOURCE [18]).

Operation	n. cycles	Latency $[ns]$	n.cycles in FMA	E_{op} $[pJ]$
DFP-mul	69	82.8	35	1383
DFP-add	71	85.2	36	1423
DFP-div	171	205.2	86	3428

TABLE IV
ESTIMATE OF ENERGY NECESSARY TO PERFORM DFP OPERATIONS ON THE FMA.

we define the energy consumption to perform an operation (in n cycles) as:

$$E_{op} = P_{ave} \cdot n \cdot T_C \quad [J] \ .$$

Table II shows the latency, average power dissipation and E_{op} for the different operation performed on the two units. UniPro is half the area the FMA. For BFP-div, the only operation supported by both units, the average power dissipation in UniPro is about one third and the energy necessary to complete the operation is less than one quarter than the one necessary in the FMA.

For the DFP operations we cannot make a direct comparison, but according to the results of [18], partially reported in Table III for the operations implemented in UniPro, a hardware implementation is much more efficient than software functions run on the FMA. For example, for DFP-mul the average number of clock cycles to implement the software function is 69. Assuming the same clock frequency, UniPro is about 3.5 times faster (20 vs. 69 cycles).

Assuming that only fused multiply-add operations are used to implement decimal operations on the FMA ($P_{ave} \simeq 35\ mW$) and that with respect to the median values of Table III only half the cycles are actually spent in the FMA, we can estimate the energy consumption as shown in Table IV.

By considering an average activity of the two units, we estimated the temperature rise in the different parts of the die. Fig. 3 (at left) shows the floorplan of one FMA and one UniPro placed next to each other. The right part of Fig. 3 shows how the temperature is distributed in the die. For both the FMA and UniPro we considered the average power dissipation among the implemented operations.

Moreover, for UniPro we assumed it is active half the time the FMA is active. This is a quite reasonable assumption considering that in most applications the frequency of DFP operations is much lower than the frequency of BFP operations.

The maximum temperature estimated is $51^o\ C$ over the ambient temperature. The gradient (maximum temperature difference in the die) is about $6^o\ C$.

V. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a re-configurable processor implementing DFP operations (*decimal64*) and BFP division (radix-16). The processor is intended to improve the performance of decimal operations both in terms of latency and power efficiency with respect to their software implementation.

Moreover, because the hardware for DFP division is very similar to that of BFP division, we can combine BFP-div in the UniPro as well, and further reduce latency and power with respect to the FMA implementation.

The results show that UniPro can be used as an accelerator of DFP operations at a much lower latency, power and energy consumption than software solutions. The extra area on the die required by UniPro can be an acceptable compromise in the dark silicon era.

In future work, we plan to lift some of the current limitations of UniPro. In particular, we plan to increase the compliancy with the IEEE standard 754 by implementing the preferred exponent feature in DFP and by handling subnormals in BFP.

Moreover, we plan to profile the execution of decimal arithmetic functions on BFP-units and to compare the UniPro to the software solution for actual workloads.

Finally, we plan to optimize the processor by disabling parts of the datapath not used for some operations (e.g., by power-gating).

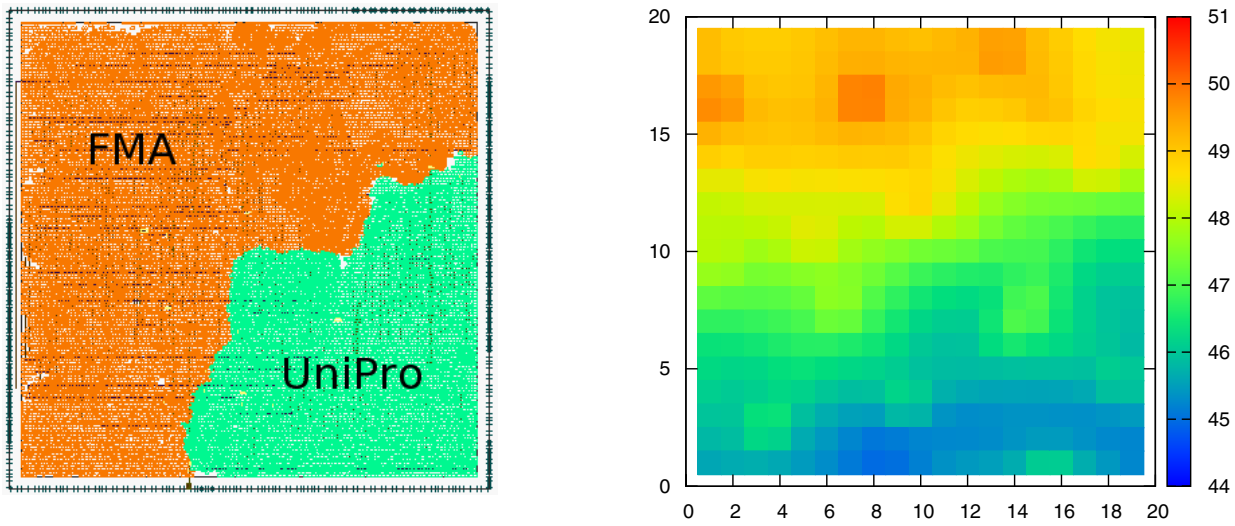

Fig. 3. Floorplan of FMA and UniPro (left). Thermal maps of the two units (right).

REFERENCES

[1] M. B. Taylor, "A Landscape of the New Dark Silicon Design Regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, Sept.-Oct. 2013.

[2] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proc. of 16th Symposium on Computer Arithmetic*, June 2003, pp. 104–111.

[3] S. Carlough, S. Mueller, A. Collura, and M. Kroener, "The IBM zEnterprise-196 Decimal Floating Point Accelerator ," in *Proc. of the 20th Symposium on Computer Arithmetic*, July 2011.

[4] T. Yoshida *et al.*, "Sparc64 X: Fujitsu's New-Generation 16-Core Processor for Unix Servers," *IEEE Micro*, vol. 33, no. 6, pp. 16–24, Nov.-Dic. 2013.

[5] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std. 754, 2008.

[6] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publisher, 1994.

[7] N. Burgess and C. N. Hinds, "Design of the ARM VFP11 Divide and Square Root Synthesisable Macrocell," *Proc. of 18th IEEE Symposium on Computer Arithmetic*, pp. 87–96, July 2007.

[8] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess, "High performance floating-point unit with 116 bit wide divider," *Proc. of 16th Symposium on Computer Arithmetic*, pp. 87–94, 2003.

[9] H. Baliga, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles, "Improvements in the Intel Core2 Penryn Processor Family Architecture and Microarchitecture," *Intel Technology Journal*, pp. 179–192, Oct. 2008, http://www.intel.com/technology/itj/2008/v12i3/3-paper/1-abstract.htm.

[10] S. F. Oberman, "Floating-point division and square root algorithms and implementation in the AMD-K7 microprocessor," *Proc. of 14th Symposium on Computer Arithmetic*, pp. 106–115, 1999.

[11] NVIDIA. "Fermi. NVIDIA's Next Generation CUDA Compute Architecture". Whitepaper. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[12] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, Sep/Oct 2000.

[13] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1059–1070, 2012.

[14] M. F. Cowlishaw, "Densely packed decimal encodings," *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 3, pp. 102–104, May 2002.

[15] L.-K. Wang, M. J. Shulte, J. D. Thompson, and N. Jairan, "Hardware Design for Decimal Floating-Point Addition and Related Operations," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 322–335, Mar. 2009.

[16] T. Lang and A. Nannarelli, "A Radix-10 Combinational Multiplier," *Proc. of 40th Asilomar Conference on Signals, Systems, and Computers*, pp. 313–317, Nov. 2006.

[17] ——, "Combined Radix-10 and Radix-16 Division Unit," in *Proc. of 41st Asilomar Conference on Signals, Systems, and Computers*, Nov. 2007, pp. 967–971.

[18] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," in *Proc. of the 18th Symposium on Computer Arithmetic*, July 2007.