 DOI: 10.1109/TVLSI.2017.2784807

Vector Processing-Aware Advanced Clock-Gating Techniques for Low Power Fused Multiply-Add

Ivan Ratković, Oscar Palomar, Milan Stanić, Osman Unsal, Adrian Cristal, and Mateo Valero

Abstract—The need for power-efficiency is driving a rethink of design decisions in processor architectures. While vector processors succeeded in the high-performance market in the past, they need a re-tailoring for the mobile market that they are entering now. Floating point fused multiply-add, being a functional unit with high power consumption, deserves special attention. Although clock-gating is a well-known method to reduce switching power in synchronous designs, there are unexplored opportunities for its application to vector processors, especially when considering active operating mode. In this research, we comprehensively identify, propose, and evaluate the most suitable clock-gating techniques for vector fused multiply-add units (VFU). These techniques ensure power savings without jeopardizing the timing. We evaluate the proposed techniques using both synthetic and "real world" application-based benchmarking. Using vector masking and vector multi-lane-aware clock-gating, we report power reductions of up to 52%, assuming active VFU operating at the peak performance. Among other findings, we observe that vector instruction-based clock-gating techniques achieve power savings for all vector floating-point instructions. Finally, when evaluating all techniques together, using "real world" benchmarking, the power reductions are up to 80%. Additionally, in accordance with processor design trends, we perform this research in a fully parameterizable and automated fashion.

Index Terms—Digital Arithmetic, Clock-Gating, Low Power, Vector Processors, Fused Multiply-Add, Methodologies.

1 Introduction

Power- and energy-efficiency have become the dominant limiting factor to processor performance and have increased significantly processor design complexity, especially when considering the mobile market. Being able to exploit high degrees of data-level parallelism (DLP) at low cost in a power- and energy-efficient way [2, 3, 4], vector processors are an attractive architectural-level solution. Undoubtedly, the design goals for mobile vector processors clearly differ from the performance-driven designs of traditional vector machines [5]. Therefore, mobile vector processors require a redesign of their functional units (FU) in a power-efficient manner.

Clock-gating is a common method to reduce switching power in synchronous pipelines [6, 7, 8, 9, 10]. It is practically a standard in low-power design. The goal is to "gate" the clock of any component whenever it does not perform useful work. In that way, the power spent in the associated clock tree, registers and the logic between the registers is reduced. It is the most efficient power reduction technique for active operating mode[1]. Therefore, the conditions under which clock-gating can be applied should be extensively studied and identified. A widely used approach is to clock-gate a whole FU when it is idle [6, 7]. A complementary and more challenging approach is clock-gating the FU or its sub-blocks when it is active, i.e. operating at peak performance [8]. Furthermore there are characteristics of vector processors that provide additional clock-gating opportunities (that we discuss in Section 4).

Since fused multiply-add (FMA) units usually dissipate the most power of all FUs, their design requires special attention. Abundant floating-point (FP) FMA is typically found in vector workloads such as multimedia, computer graphics or deep learning workloads [11]. Although in the past FMA have been used for high-performance, it recently have been included in mobile processors as well [4, 12]. In contrast to high-performance vector processors (e.g. NEC SX-series [13] and Tarantula [14]) that have separated units for each FP operation, mobile vector processors' resources are limited, thus, we typically have a single unit per vector lane capable of performing multiple FP operations rather than separate FP units [4]. Apart from that, additional advantages of using FMA over separate FP adder and multiplier are: (1) computation localization inside the same unit reduces the number of interconnections (power- and energy-efficiency), (2) higher accuracy (single, instead of two round/normalize steps), and (3) improved performance (shorter latency).

In this paper, we investigate the design of a low power fully pipelined double precision IEEE 754-2008 compliant FMA unit for vector processors (VFU). In our main contribution, we comprehensively identify, propose, and evaluate (using both synthetic and real world workloads) the most suitable clock-gating techniques for VFU running at peak performance periods without jeopardizing performance. We present three kinds of techniques: (1) novel ideas to exploit unique characteristics of vector architectures for clock-gating during active periods of execution (e.g. vector instructions with a scalar operand or vector masking), (2) novel

- *Ivan Ratković is with Esperanto Technologies and Semidynamics. Oscar Palomar is with University of Manchester. Milan Stanić is with ASML. Osman Unsal, Adrian Cristal, and Mateo Valero are with Barcelona Supercomputing Center (BSC).*
  E-mail: ivan-srb@live.com, oscar.palomar@gmail.com, stanic.milan@gmail.com, osman.unsal@bsc.es, adrian.cristal@bssc.es, mateo.valero@bsc.es

- *An earlier 8-page conference version of this paper appeared in ISLPED [1]. This article extends the prior work with more background, more explanations, upgraded methodology and new evaluations.*

1. Active operating mode assumes a busy functioal unit.

ideas for clock-gating during active periods of execution that are also applicable to scalar architectures but especially beneficial to vector processors (e.g. gating internal blocks depending on the values of input data), and (3) ideas that are already used in other architectures and that we present as its application is beneficial to vector processors, and for the sake of completeness (e.g. idle VFU). Regarding the second and the third group of ideas, an advantage of vector processing that extends the applicability of clock-gating is that vector instructions last many cycles, so the state of the clock-gating and bypassing logic remains the same during the whole instruction. As a result, power savings typically overcome the switching overhead of the added hardware (which is often not a case in scalar processors).

To fulfill current trends in digital design that promote building generators rather than instances [15, 16] we perform this research in a fully parameterizable, scalable and automated manner. We developed an integrated architecture-circuit framework, that consists of several generators, simulators and other tools, in order to join architectural-level information (e.g. vector length or benchmark configuration) with circuit-level outputs (e.g. VFU power and timing measurements). We implement our clock-gating techniques and generate hardware VFU models using a fully parameterizable Chisel-based [17] FMA generator (FMAgen) and a 40nm low power technology.

We discuss the related work individually for each of our clock-gating techniques together with the description of the technique in Section 4. Besides, in the context of alternative low power techniques for FP units, interesting approaches have been proposed: memoing (caching results that can be reused) and byte encoding (computation performed over significant bytes). However, detailed and accurate evaluation reveals that the actual savings are often low and with an unaffordable area overhead [18].

In summary, the main contributions of the paper are:

- The first proposal of active clock-gating techniques for VFU. (Section 4).
- An in-depth evaluation of the proposed techniques:
  - Detailed power savings evaluation of limits of each proposed technique separately using synthetic benchmarking (Section 6.2).
  - Realistic, combined, scenario evaluation using real application-based benchmarking. We find the techniques significantly reduce power with no performance loss (Section 6.3).
- A fully automated, parameterizable, and scalable exploration framework including our hardware and software (benchmark) generators. (Section 5).

2 Vector Processors Background

Vector processors operate on vectors of data within the same instruction[2]. Vector instruction set architecture (ISA) provides an efficient organization for controlling a large amount of computation resources. Furthermore, vector ISAs emphasize local communication and provide excellent computation/area ratios. Vector instructions express DLP in a very compact form, thus removing much redundant work (e.g. instruction fetch, decode, and issue). For example, a vector FP FMA instruction (`FPFMAV`) indicates the operation (FMA), three source vector registers and one destination vector register. Thus, tuples of three elements, one from each source register, are the inputs for the VFU, and the result is written to the destination. All tuples can be processed independently, and multiple elements could be accommodated in a vector register.

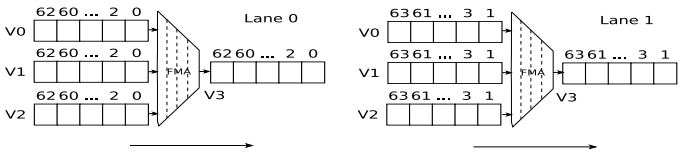


Fig. 1: A 2-lane, 4-stage VFU (MV_L=EV_L=64) executing `FPFMAV V3<-V0,V1,V2`.

The register file is designed so that a single named register holds a number of elements. The entire architecture is designed to take advantage of the vector style in organizing data. Additionally, the memory system of vector processors allows efficient strided and indexed memory access. The number of elements of a vector register is denoted by the maximum vector length (MV_L). Occasionally fewer elements than the MV_L are used, which reduces the effective vector length (EV_L).

The vector execution model streamlines one vector register element per cycle to a fully pipelined vector FU. As a result, the execution time of a vector instruction is the start-up latency (number of stages) of the vector FU plus the EV_L. A common technique to reduce this time is to implement multiple vector lanes through replicated lock-stepped vector FU. Each lane accesses its own "slice" of the vector register file, which reduces the need for increasing the number of ports typically associated with a larger number of FUs. Lock-stepping the lanes simplifies the control logic and is power-efficient. These concepts are illustrated in Figure 1. Although lanes were proposed for increasing performance, using multiple lanes can increase the energy-efficiency of a vector architecture [3, 9, 10].

Additionally, an interesting feature that vector processors typically offer is a vector mask control. Masked operations are used to vectorize loops that include conditional statements. Masked operation uses an MV_L-bit vector mask register (VMR) for indicating which operations of the vector instruction are actually performed. In other words, masked vector instructions operate only on the vector elements whose corresponding entries in the VMR are '1'.

Conventional vector processors should not be confused with single instruction multiple data (SIMD) multimedia extensions such as AVX-512 [19] that are an alternative way to exploit DLP and indicate operations to perform on multiple elements[3]. The main difference of these extensions with a conventional vector processor is that they exploit subword-SIMD parallelism and are typically implemented with multiple vector FUs that operate on all independent elements in parallel. Having a vector FU per element to operate on all them in parallel would be inefficient for vector processors because they operate on much longer vectors. Instead, a vector FU is fully pipelined, and the elements of the vector register are streamlined to the unit, one per cycle, possibly using a small number of vector lanes.

3 Floating-Point Fused Multiply-Add

This section briefly describes FP representation and FP FMA. Additional details about floating-point arithmetic are available

2. In contrast, a "traditional", non-vector, processor can be defined as a processor that operates on scalar values, hence known as scalar processors.

3. Vector processors are SIMD architectures in Flynn's taxonomy [20], although by SIMD we refer to such type of multimedia extensions.

TABLE 1: IEEE754 single and double precision formats. The number of bits for each field is shown (bit ranges are in square brackets, 0 = least-significant bit).

	Sign	Exponent	Fraction
Single Precision (32 bits)	1 [31]	8 [30-23]	23 [22-0]
Double Precision (64 bits)	1 [63]	11 [62-52]	52 [51-0]

in [21, 22, 23].

3.1 Floating-Point Representation

Floating-point (FP) representation is the most common way to represent real numbers in computers. It is based on the scientific notation to encode numbers, $M * 10^E$, where M and E are the mantissa and the exponent respectively. For example, 123.4 could be represented as $1.234 * 10^2$. In the same way the binary number 10100.1_2 could be represented as $1.01001_2 * 2^4$.

IEEE754 floating-point numbers have three basic components: the sign (S), the exponent (E), and the fraction (F). IEEE754 double and single precision floating-point formats are shown in Table 1. The sign bit '1' indicates negative, while '0' indicates positive numbers. The mantissa is composed of the fraction and an implicit (hidden) leading '1'[4]. The exponent base (2) is implicit and needs not be stored. The exponent field contains the sum of bias (B) and true exponent (E_T). The bias is 127 for single and 1023 for double precision numbers. Therefore the value represented by an IEEE754 FP number is: $(1-2S) * M * 2^{E_T} = (1-2S) * (1+F) * 2^{E-B}$.

Special value *NaN* is used for representing undefined values. This happens when one (or more) operand is *NaN* or when the operation is: (1) $0 * \infty$, (2) $\infty - \infty$, (3) $0/0$, ∞/∞, (4) x *mod* 0, ∞ *mod* y, or (5) $\sqrt{x}, x < 0$. Another important special value is infinity ($\pm\infty$). This happens when either input is ∞ or in case of division by zero. *NaN* and ∞ handling are explained in [21].

3.2 Fused Multiply-Add (FMA)

The FMA unit executes the FMA instruction (`FMA R <- A, B, C`) that implements $R = A * B + C$. In contrast to a multiplication followed by an addition, the FMA instruction assumes all three operands at the same time. It was introduced for the first time in IBM's RS/600 in 1990 [24]. IEEE754-2008 standard defines the FMA instruction to be computed initially with unbounded range and precision, rounding only once to the destination format. For this reason, FMA is faster and more precise than a multiplication followed by an addition. The FMA unit performs operand alignment in parallel with the multiplication. This leads to shorter latency (n_S) compared to a multiplication followed by an addition. Additionally, the FMA operation reduces the number of interconnections between floating-point units and the number of adders and normalizers. The FMA instructions help compilers to produce more efficient code. Potential drawbacks are increased latency of `FPADD` and `FPMUL` instructions (if executed on the FMA) and a complex normalizer. A simplified list of steps of the computations the FMA flow are:

1) Mantissas multiplication ($M_A * M_B$), exponents addition ($E_A + E_B$), alignment of the addend's mantissa (M_C), and calculation of the intermediate result exponent $E_R = \max(E_A + E_B, E_C)$.
2) Addition of the product ($M_A * M_B$) and aligned M_C.
3) Normalization of the addition result and exponent update.
4) Rounding.
5) Determination of the exception flags and special values.

A simplified implementation block diagram of the FMA unit used in our research is shown on Figure 2. As we assume double precision we need a 162-bit adder and a 53x53 multiplier. For the adder and the multiplier, we choose Brent-Kung and Wallace algorithms, respectively, as it is aligned with our findings in [9, 10]. The aligner performs shifting of the addend based on the exponents difference in order to align it with the product ($M_A * M_B$).

Floating point addition using the FMA unit is implemented by setting the first operand to 1 ($A = 1.0$), while floating point multiplication is implemented by setting the third operand to 0 ($C = 0.0$).

4. An exception are subnormal numbers where the implicit bit is '0'.

TABLE 2: A classification of the proposed techniques using two criteria: (1) Vector Processing-*Specific* or -*Beneficial* and (2) operating mode (*Active* or *Idle*).

	VP-specific	VP-beneficial
Active	*MaskCG*, *ScalarCG*, *ImplCG*	*InputCG*
Idle	n/a	*IdleCG*

4 Proposed Clock-Gating Techniques

This section presents the proposed clock-gating techniques for VFU. The classification is presented in Table 2.

4.1 Scalar Operand Clock-Gating (ScalarCG)

We propose this technique to tackle the cases in which one or two operands do not change during the vector instruction. Table 3 lists the types of instructions during which *ScalarCG* is active. As only one of all the supported vector instructions has all three vector operands, often at least one operand is scalar. Only the `FPFMAV` instruction, in which all operands are vectors, does not benefit from this technique.

During these instructions the corresponding input register(s) of scalar operand(s) should latch a new value only on the first clock edge of the execution of the instruction, while during the rest of the instruction, they can be clock-gated. To implement this, we introduce the signals *VS*[2..0] (Figures 2), where *VS*[i] = 0 means that the i-th operand is gated after the mentioned first cycle. *VS* signals are derived from the instruction *OPCODE*. Deriving VS signals from the *OPCODE* is done before the first pipeline stage (as indicated in Figure 2). This generation (decoding) requires regular comparators and they are not on the critical path as the *OPCODE* is available at least one cycle in advance. Table 3 shows corresponding *VS* signals for all the instructions.

4.2 Implicit Scalar Operand Clock-Gating (ImplCG)

This technique is an additional optimization of *ScalarCG* and aims to exploit further the information given through the instruction *OPCODE* for clock-gating, operand isolation, and computation bypassing. In the case of addition and subtraction instructions, such as `FPADDV` and `FPSUBV`, the 53x53 mantissa multiplier is

5. Without loss of generality, we assume that the compiler puts the scalar multiplicand and addend always as the second and third operand in these instructions, respectively.

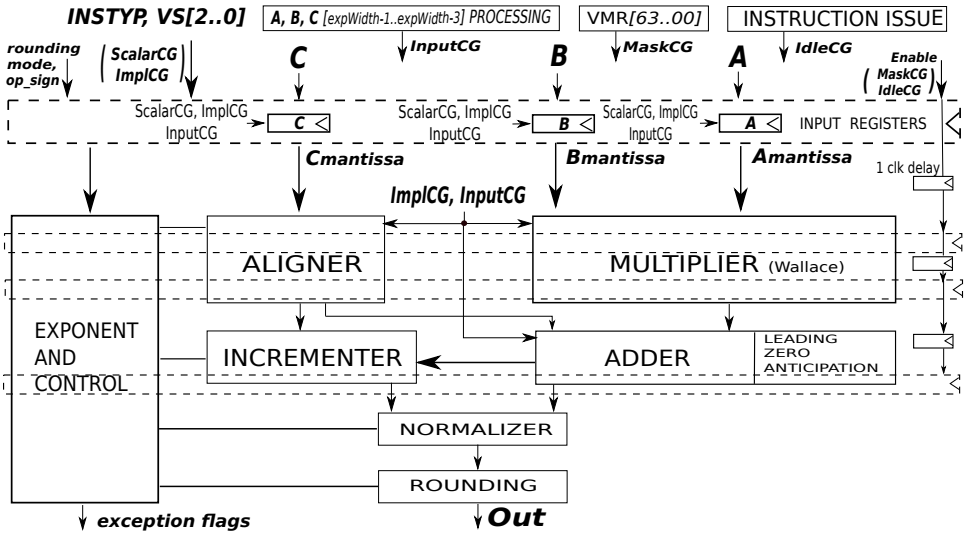


Fig. 2: A simplified block diagram of a 1-lane, 4-stage, VFU with all clock-gating techniques applied (*AllCG* technique). Input signals for the baseline without clock-gating are multiplicands (*A, B*), addend (*C*), *rounding mode*, and operation sign (*op_sign*) while output signals are result (*Out*) and *exception flags*. Details regarding applied clock-gating techniques are explained in Section 4.

TABLE 3: Types of instructions where *ScalarCG* and *ImplCG* is applicable. Capital letters indicate vector operands. Third and fourth column shows the corresponding *VS* and instruction type signals, respectively.

Operation	Vector Instruction	*VS[0..2]*	*INSTYP*
$A*b+C$	FPFMAVSV - a multiplicand is a scalar	101	0
$A*B+c$	FPFMAVVS - the addend is a scalar	110	0
$A*b+c$	FPFMAVSS - a multiplicand and the addend are scalars	100	0
$A*B+0$	FPMULV - the addend c is a scalar (0)	110	1
$A*1+C$	FPADDV - the multiplicand b is a scalar (1)	101	1
$A*/+\ b/c$	FPMULVS/FPADDVS - 2 out of 3 operands are scalars[5]	100	1

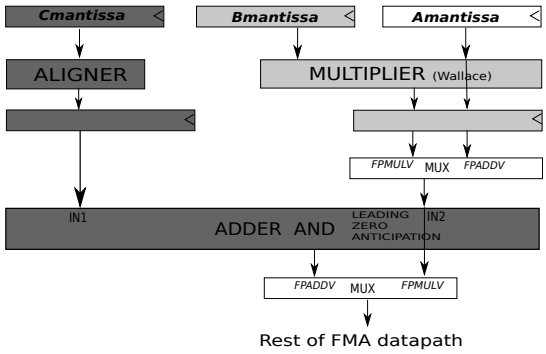


Fig. 3: Gated FMA submodules during FPMULV (dark grey) and FPADDV (light grey) instructions in case of *ImplCG*.

not needed as it is known that one of the multiplicands is '1', thus, we can bypass, isolate, and clock-gate it providing the value of the other multiplicand directly to the adder. There is an analogous situation for FPMULV, since the the addend is known to be '0'. In this case, the 162-bit wide adder, leading zero anticipation and the aligning part are not needed.

To control bypassing, isolation, and clock-gating of the mentioned submodules, we introduce signal *INSTYP* (Figure 2 and Table 3), generated from the instruction *OPCODE*, which indicates whether an FPFMAV or an FPADDV/FPSUBV/FPMULV instruction is executed. *INSTYP* together with VS signals provide information of the instruction type. For example, *INSTYP*=1 and *VS*[0]=1 and *VS*[2]=0 indicate that we have an FPMULV instruction while *INSTYP*=1, and *VS*[1]=0 indicate that we have an FPADDV/FPSUBV instruction. Figure 3 shows the simplified block diagram of gated FMA submodules when the aforementioned instructions are executed. Circuitry added for implementing *ImplCG* mostly consists of clock gating cells and MUXs.

In the context of instruction-dependent techniques, there is interesting research done in the past for scalar processors [8]. The main advantages of our *ImplCG* proposal over the mentioned research are: (1) we apply the technique for a variable number of pipeline stages, (2) we evaluate power, timing, and area, and (3) we propose the technique for vector processors.

The advantage of applying this technique on a vector processor over other models (e.g. scalar) is that vector instructions last many cycles, so the state of the related hardware (clock-gating logic and MUXs) maintains the same state during the whole instruction. Thus, there will be less switching overhead than in the scalar case.

4.3 Vector Masking and Vector Multi-Lane-Aware Clock-Gating (MaskCG)

Here we target cases in which there are idle cycles during the vector mask instructions (e.g. FPFMAV_MASK). Common cases in which vector mask control is used are: (1) sparse matrix operations and (2) conditional statements inside a vectorized loop. Additionally, we assume the same mechanism is also used to reduce the EV_L to less than the MV_L. We assume that the control logic will detect and optimize this case, skiping the last elements of the vector corresponding to the trailing 0s of the mask. However, in vector designs with n_L lanes, there will still be mod(EV_L, n_L) idle lanes in the last cycle of the operation.

TABLE 4: *InputCG* - conditions under which a hardware block of mantissa arithmetic computations and corresponding input registers can be bypassed, isolated and clock-gated. The fourth column shows three highest bits of the exponent that we use for the detection.

Hardware block	*Condition*	*Subtechnique*	*Exp[11..9]*
Full computation	The result is *NaN*.	$InputCG_{NaN}$	111
Full computation	The result is ∞.	$InputCG_{inf}$	110
Multiplier	Multiplicand is '0'.	$InputCG_{mul0}$	000
Adder and aligner	Addend is '0'.	$InputCG_{add0}$	000

The VMR directly controls the clock-gating of the whole arithmetic unit during these idle cycles (Figure 2). Regarding the internal implementation, we perform clock-gating at pipeline stage granularity [25], so we prevent useless cycles inside the unit i.e. the data is latched in subsequent stages only if necessary. Once the *Enable* signal of the first pipeline stage's register gets the value '1', this *Enable* signal propagates to the end of the pipeline, one stage per cycle (Figure 2). In other words, the *Enable* signal of n-th stage is actually the first stage's *Enable* signal delayed by $n-1$ cycles. This is implemented by adding a 1-bit wide, $n_S - 1$ long shift register that drives clock-gating cells.

To the best of our knowledge, there is no related work that aims to exploit vector conditional execution with VMR to lower the power of vector processors.

4.4 Input Data Aware Clock-Gating (InputCG)

Here we identify the scenarios in which, depending on the input data, a part of mantissa processing is not needed for the correct result and thus, can be bypassed. We use a recoded format for internal representation [26], that allows us to detect special cases (explained in Section 3.1) and zeros with an negligible hardware overhead: it requires inspection of only three most significant bits of the exponent (fourth column in Table 4). Table 4 presents the identified scenarios (conditions) in which a hardware block of mantissa arithmetic computations and the corresponding input registers can be bypassed, isolated and clock-gated.

The recoded format allows detection of relevant scenarios by using simple 3-bit comparators. They are located at the inputs of VFU (A, B, C Processing block on Figure 2)[6]. In that way we assure the mentioned detection comparators are not on the VFU's critical path, i.e. gating information is available in time.

The added internal hardware is similar as for *ImplCG*. Having zero addend is analogue to `FPMULV` instruction case (Figure 3). Zero multiplicand allows gating and bypassing all the modules from Figure 3 except the registers that holds operand *C* value as in that case the final result is operand *C*. In case of *NaN* and infinity there is no need for any computation as the result which has to be at VFU output is already known (explained in Section 3.1) so we can gate/bypass/isolate vast majority of FMA submodules.

There are many workloads whose data contain a lot of zero values [27, 28], thus can fairly benefit from the last two subtechniques presented in Table 4. Although these techniques are applicable to other architectures as well, their application to vector processors is more efficient since the recurrent values are common within the vector data, thus lowering the switching overhead in added hardware (clock-gating logic and MUXs).

6. Formally they are located one stage before, in the instruction decode stage.

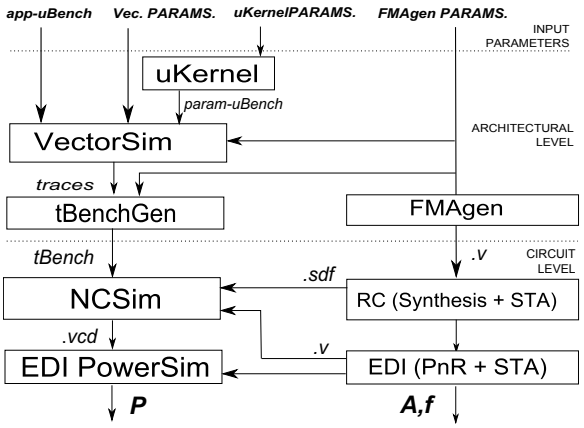


Fig. 4: A simplified block diagram of the framework.

While both *ImplCG* and *InputCG* techniques aim to exploit cases when the addend is '0', in this case, there is no external information of '0' existence via *VS* signals, but it has to be detected, and the gating has to be done on time.

As in the case of *ImplCG*, the research done in [8] presents a related data-driven technique for scalar processors. The main advantages (that enable additional savings) of our *InputCG* technique over the mentioned research are (1) detection of zero operands, (2) distinction between ∞ and *NaN*, and (3) gating the mantissa multiplier when processing *NaN*s.

4.5 Idle Unit Clock-Gating (IdleCG)

This technique clock-gates the VFU when no data is supplied to it. The clock (un)gate decision is made in the instruction issue pipeline stage, where it is known if an instruction will be sent to the VFU in the next cycle (Figure 2). As indicated on Figure 2, this technique uses the same internal clock-gating circuitry as *MaskCG*. A similar approach is widely used in scalar processors and is known as *Deterministic Clock-Gating* [6, 7]. Nonetheless, this technique has more potential for power savings than its scalar equivalent as it can benefit from the following vector specific advantages:

- Vector FUs are used in burst fashion (with idle periods between bursts), since a single `FMA`/`ADD`/`SUB`/`MUL` instruction processes all vector elements in consecutive cycles. This makes clock-gating more efficient as the overhead of its buffers is minimized.
- For high frequency designs, the issue stage may need an additional cycle to determine if a unit will be used in the next cycle. In a scalar processor, we would need to waste that cycle once per each scalar `FMA`/`ADD`/`SUB`/`MUL` instruction. In a vector processor, we waste this cycle $EV_L - 1$ times less.

Although here we focus lowering power of VFU when it is active, we present this technique for the sake of completeness.

5 Methodology

A simplified block diagram of the framework is depicted in Figure 4. It includes architectural- (*uKernel*, *VectorSim*, *FMAgen*) and circuit-level (*RC, EDI, NCsim*) simulators and tools, as well as an interfacing tool (*tBenchGen*). For various parameters we obtain power (*P*) and area (*A*) of the VFU.

The first step is feeding *VectorSim* (described in Section 5.1) with vectorized microbenchmarks (*uBench*) and vector parameters

(MV_L and number of vector lanes (n_L)). Using these inputs *VectorSim* generates data and timing traces for the vector floating point operations. We use two kinds of *uBench*s (both explained in Section 5.2):

- ***param-uBenchs*** are generated by feeding the parameterizable microkernel *uKernel* with its parameters.
- ***app-uBenchs*** are manually vectorized kernels extracted from applications.

The synthesizable Verilog netlists are generated using *FMAgen* (described in Section 5.3). The output of architectural-level simulations together with *FMAgen* parameters are transformed into Verilog test benchmarks (*tBench*s) using *tBenchGen*, a tool that we developed. The most important inputs are: data and time traces from *VectorSim*, vector and *FMAgen* (explained in Section 5.3) parameters, clock cycle, and *tBench* length expressed in the number of Verilog test vectors. As output, it provides *tBench* for each lane separately, and its profiling report.

Afterwards, we use Cadence RTL Compiler (*RC*) to obtain synthesized mapped netlists and to perform static timing analysis (STA) of the VFU. All the designs are synthesized for the minimum clock period that provides a safe slack on the critical path. We optimize non-critical path logic for leakage using high-V_{TH} cells. We provide synthesized Verilog netlists together with the physical layout information to the Cadence Encounter Digital Implementation System (*EDI*) to get placed and routed designs and to perform again STA. Additionally, the most critical paths are verified with Cadence Spectre.

In order to verify the designs and extract resulting switching activity information (written into Value Change Dump (vcd) files) we simulate each VFU in Cadence *NCSim* for each matching *tBench* with back-annotated delays using standard delay format (sdf) files. Afterwards, we perform precise power estimation using *EDI PowerSim*.

All designs are implemented using the mentioned low power and low leakage TSMC40LP library for *typical* operating conditions ($V_{dd} = 1.1V, temp = 25C$). Since practically all existing vector processors are developed using standard cells [29], we selected this approach in our research. We use latch-based integrated clock gating cells from the cell library. We set the tools to meet timing constraints while prioritizing power over the area. All the optimizations in all the tools are applied using *high effort*.

An initial target core density of 70% is selected as a sensible balance between timing improvement and shrinking area for the wide set of designs parameters that we use. We experimentally found that, in place and route (PnR) stage in general, density below 70% sometimes provides negligible faster timing for a non-negligible area overhead while densities higher than 70% can spoil timings noticeably as the tool suffers from the lack of free space for optimizations and routing. Additionally, the initial densities below 70% sometimes even cause DRC errors and density (congestion) violations.

5.1 VectorSim

We built VectorSim based on the vector architecture library (*VALib*) and the *SimpleVector* simulator [11], developed in our group. VALib is a library that implements vector instructions and allows rapid manual vectorization and characterization of applications. SimpleVector is a simple and very fast trace-based simulator which helps to estimate the performance of a vector processor. We took advantage of the fact that both tools have been designed to be easily extended with new instructions or implementation alternatives. Therefore, we modified them to satisfy our research goals and to enable its integration in our exploration frameworks. Among other upgrades, we added a set of vector floating-point FMA instructions to *VectorSim*. High-level *VectorSim* configuration is presented in Table 5.

TABLE 5: High-level *VectorSim* configuration.

Execution	32-bit in-order vector core; decoupled vector and scalar core; vector chaining.
Vector Register File	MV_L: 16, 64, or 128 elements; Number of registers: 8.
Vector FU	n_L: 1, 2, or 4; 1 32-bit arithmetic logic unit (ALU), 1 64-bit VFU; VFU latency (n_S): 1-4.
Memory System	L2 (direct access to L2, shared with scalar core): 1MB, 4-way, 128b cache line, hit latency: 7ns, miss latency: 70ns; 1 load unit, 1 store unit.

We setup *VectorSim* to model a decoupled 32-bit vector machine with support for 64-bit floating point. The decoupled execution model assumes separated in-order vector and scalar execution units [14, 30, 31, 32]. They share instruction fetch and decode, and they separate issue logic and functional units, allowing in that way independent scalar execution. In-order execution is common in low power processors due to its simplicity (e.g. some of ARM Cortex-A architectures: cortexA7, cortexA8, cortexA32, cortexA35, cortexA53 [33]). It is more efficient in vector than in scalar processing as in vector architectures the drawbacks of in-order execution are diminished, especially if the vectors are long. Additionally, we model chaining (vector equivalent of data forwarding) and dead time elimination (allowing to reuse the ALU immediately after the current instruction).

The vector execution engine is organized as n_L identical vector lanes. Possible values of the number of lanes (n_L) are 1, 2, and 4. In our experiments, we do not examine more lanes as it would not satisfy well a low power core budget[7]. Moreover, values that we choose are typical in vector processor design [29]. Each lane has a slice of the vector register file, a slice of the vector mask file, 1 vector integer ALU, 1 VFU, and a private TLB. There is no communication across lanes, except for gather/scatter, reduction, and compress instructions. In addition to the vector ALU, each lane also includes 1 logic unit that handles logic operations, shifting and rotating. We assume the division is done in software, since it is rare in vectorizable applications and the hardware support is costly. Additionally, a control unit is needed to detect hazards, both from conflicts for the functional units (structural hazards) and from conflicts for register accesses (data hazards).

5.2 Benchmarking

This section explains two benchmarking methods that we employ for an in-depth evaluation of the proposed techniques. The first method has as goal to stress each of the techniques separately, while the second tests all the techniques simultaneously and provides the results for "real world" applications.

5.2.1 *Fully Parameterizable Kernel* - uKernel

We generate different *param-uBenchs* using the same *uKernel*. It is a variant of the *DAXPY* loop: $D = A * B + C$. The inputs are random values unless specified otherwise. *uKernel* parameters (Table 6)

7. The total number of functional units per core is in accordance with many other low power processors [33, 34, 35].

TABLE 6: *uKernel* parameters.

Parameter	Description
INSTYP	It indicates whether we have `FPFMAV` (0) or `FPADDV` and `FPMULV` (1) types of vector instructions.
ADD/MUL	It indicates whether the instruction is addition (0) or multiplication (1) if *INSTYP*=1.
MULS, ***ADDS***	They indicate whether one of the multiplicands and the addend are scalar values, respectively.
p_m	It indicates the probability that a bit in the VMR during a vector operation is '0'. The probability that one lane is idle in the last cycle is included in p_M.
p_{inf}, p_{NaN}, p_0	These indicate the probabilities that an operand is ∞, not a number (*NaN*), and 0, respectively.[8].
IR	It indicates the Idleness Ratio, $IR = \frac{T_{IR}}{T_{EX}+T_{IR}}$, where T_{IR} is the average pause length between two subsequent vector instructions and T_{EX} is the average execution time of vector instructions.

are used to determine the characteristics of the generated *param-uBench*. There are parameters that modify the code (*INSTYP*, *ADD/MUL*, *MULS*, *ADDS*), execution (*IR*, p_m), and data (p_{inf}, p_{NaN}, p_0). Listing 1 shows an example of uBench pseudocode generated with the uKernel. We iterate *param-uBench* until we reach 10000 test vectors to assure a representative sample, using a uniform distribution. The aforementioned *VS* signals are derived from *INSTYP*, *ADD/MUL*, *MULS* and *ADDS* parameters.

Listing 1: A simplified *param-uBench* pseudocode generated with *INSTYP*=0, *MULVS*=0, *ADDS*=1, and *pm>0*.

```
for (i=0; i< length; i+=MVL) {
  LDV V0 <- A[i+0..63]
  LDV V1 <- B[i+0..63]
  LD R2 <- c
  VMR <- MASK[i+0..63]
  FPFMAVVS_MASK V2 <- V0, V1, R2
  STV D[i+0..63] <- V2
}
```

5.2.2 *Application-Based Microbenchmarks* - app-uBench

An *app-uBench* is a manually vectorized, floating-point intensive microbenchmark (kernel) extracted from an application. It is a representative part of the application and small enough (between 100k and 150k test vectors) to keep circuit simulation time reasonable. We use four different *app-uBenchs* extracted from the vectorized applications described in Table 7. We selected different types of applications to make the results more general. These applications are used in mobile devices and can also be found in server workloads.

5.3 A Fully Parameterizable FMA Generator

We developed FMAgen as a hardware generator written in Constructing Hardware in Scala Embedded Language (Chisel), a hardware construction language aimed at designing hardware by using parameterized generators [17]. Chisel is based on the Scala programming language, and it supports a combination of object-oriented and functional programming and good software engineering techniques. We find it as an optimal way to design and test parameterizable FUs. On one side it provides the ability to design and connect hardware blocks in the same way as in other hardware description languages (HDL)s (Verilog or VHDL), while on another side it is significantly more flexible (parameterizable) than existing HDL and provides significantly faster testing. Chisel allows users to code their designs in one source description and target multiple backends without rewriting their designs. The Chisel code is compact, due to its higher level of description than traditional HDL. Not surprisingly, as a general problem of high-level design approaches, a disadvantage of Chisel-based digital design is that it sometimes has worse quality of results than hand-crafted Verilog [39].

As a base for FMAgen, we take an open source floating-point library - Berkeley Hardware Floating-Point Units (BHFPU) [26]. This open source library internally uses a recoded format (the exponent has an additional bit) to detect and handle special cases, such as subnormal numbers, more efficiently[9]. BHFPU can produce FMAs for a configurable floating-point format, i.e. arbitrary number of mantissa and exponent bits.

FMAgen generates synthesizable Verilog code of 1-lane VFUs according to the input parameters (FMAgen parameters): Clock-Gating technique type (CG_{type}), latency - number of pipeline stages (n_S), and the input floating-point format. The presented advanced clock-gating techniques are compatible with each other and can be arbitrarily combined. Therefore, possible values for CG_{type} are any combination of the aforementioned clock-gating techniques (*IdleCG*, *MaskCG*, *ScalarCG*, *ImplCG*, and *InputCG*), including all of them together (*AllCG*) or none of them (*NoCG*). A combination of clock-techniques that is discussed below is *ActiveCG*, which combines all active clock-gating techniques from Table 2 (*MaskCG, ScalarCG, ImplCG, InputCG*). n_S can be an arbitrary number. In this study, we put 4 stages as a reasonable limit for a low power processor. Additionally, we set the VFU input floating-point format to double precision.

Apart from the mentioned features that we added to BHFPU (support for all the clock-gating techniques as well as support for combining them arbitrarily, pipelining, and different pipelining styles), we also added full IEEE754-2008 compliance [40] (which introduces some timing overhead). A simplified block diagram of modeled VFU is shown on Figure 2.

We paid special attention to ensure that clock-gating logic does not create a critical timing path. The only circuitry that could be on the critical path are bypassing multiplexors (see Figure 3).

TABLE 7: Vectorized application-based microbenchmarks (*app-uBench*). In brackets are given names of corresponding benchmark suites.

Sphinx3 (SPEC2006-ref [36]) is a widely known speech recognition system that includes both an acoustic trainer and various decoders, i.e., text recognition, phoneme recognition, N-best list generation, etc.
Facerec (SPEC2000-ref [36]) is an implementation of a face recognition system.
K-means (modified STAMP [37]) is one of the oldest and most commonly used clustering algorithms. It is a prototype based clustering technique defining the prototype in terms of a centroid which is considered to be the mean of a group of points and is applicable to objects in a continuous n-dimensional space.
Disparity Map - computeSAD (San Diego Vision Benchmark [38]) computes depth information using dense stereo. It is used for robot vision for stereo vision.

8. To explore potential savings we use the whole range of probabilities, including values that not represent a realistic case (e.g. 100% *NaN*s).

9. We assume recoding is done when loading and storing to memory.

However, compared to the rest of FMA submodules, their delay impact is fairly small. Additionally, we debug timing and apply retime[10] when necessary. Therefore, timing cost of added circuitry is almost negligible, especially in case of 4-stage VFU (the most relevant one).

Since we target low power, we do not incorporate any speculative hardware for improving performance, thus, no energy is wasted on precomputed results that get discarded.

6 Evaluation

This section presents an evaluation of the presented vector processing aware clock-gating proposals in terms of power savings (S) and area efficiency. Regarding power measurements, first we evaluate each technique separately using the benchmarking method from Section 5.2.1, and afterwards we evaluate combined scenarios using the method explained in Section 5.2.2.

VFU designs with 1, 2, 3, and 4 stages are synthesized and run at 0.45, 0.85, 1.1, and 1.3 GHz respectively. We assume a *NoCG* VFU as a baseline. Its power in case of 2-lane VFU is 15.6, 30.9, 44.9, and 59.2 mW for 1, 2, 3, and 4 stages respectively.

We observe that the static (leakage) power is practically negligible compared to dynamic power. For *noCG* it is around 0.01% ot total power in average. The leakage is highest for *IdleCG* when it is up to 1%. It is practically negligible due to the following reasons: (1) arithmetic topologies produce high switching (high dynamic power), (2) the technology that we use has low leakage, and (3) we optimize non-critical path logic for leakage using high-V_{TH} cells. Although it is negligible when considering active operating modes (i.e. the execution inside a vector kernel), when the execution is outside a vector kernel (i.e. when the vector core is inactive), the leakage might be additionally suppressed via power gating.[11] However, power gating is out of the scope of this research since we target lowering power during active operating modes with no performance loss.

We focus on 4-stage results as they are the most important from the processor design perspective. Nonetheless, the 1-stage results are presented as a reference and in most cases it has the highest overhead in terms of power and area across all n_S. For the sake of simplicity, in the rest of this section we typically omit results for 2- and 3-stage designs, but we observe these results regularly scale between results for 1 and 4 stages.

6.1 Area Efficiency

Table 9 shows the area efficiency of the proposed techniques. Area for a *NoCG* 2-lane VFU configuration is 36191, 38060, 40693, and 43419 μm^2 for 1, 2, 3, and 4 stages respectively. Area overhead is in some cases higher than expected because: (1) during synthesis, we prioritized timing and power over area to assure power savings without spoiling timing and (2) Chisel generated Verilog code is sometimes less area efficient than equivalent manually written Verilog [39]. However, we observe this overhead has a strong decreasing trend as the n_S increases.

6.2 Per Technique Power Analysis

Figure 5 and Table 8 reveal the results for each of the presented vector processing aware clock-gating proposals separately, in terms of power savings for 4- and 1-stage VFU. In these experiments we set MV_L and the number of vector lanes (n_L) to 64 and 2 respectively.

TABLE 8: Evaluation of power savings for *ScalarCG* and *ImplCG* depending on the instruction type against the baseline (*NoCG*). *INSTYP, ADD/MUL, MULS,* and *ADDS uKernel* parameters are varied in these experiments to test all the instructions separately, while the rest of parameters are zero.

Vector Instruction	n_S	$S_{ScalarCG}(\%)$	$S_{ImplCG}(\%)$
FPFMAV	1	−34.35	−38.27
	4	1.87	0.39
FPFMAVSV	1	−10.11	−12.73
	4	16.47	14.55
FPFMAVVS	1	−31.47	−33.19
	4	4.07	4.74
FPFMAVSS	1	−6.77	−7.77
	4	18.90	18.83
FPADDV	1	−10.70	65.07
	4	16.54	42.42
FPADDVS	1	−6.72	70.30
	4	18.92	46.16
FPMULV	1	−31.36	−22.81
	4	4.12	9.00
FPMULVS	1	−6.69	2.52
	4	18.95	23.04

We observe that in most of the cases the savings grow with n_S, as more pipeline stages enable finer granularity of clock-gating. Due to its higher practical importance, in the rest of the discussion we focus on 4-stage results.

Figure 5 shows results for *MaskCG*, *InputCG*, and *IdleCG*:

✦ ***MaskCG***. Due to its simplicity, this technique comes with practically no overhead and the savings are between 8% and 52% depending on the p_m. The saving attainable when p_m=1 (S=52%) is the maximum possible power reduction for active 4-stage VFU.

✦ ***InputCG***. In order to isolate savings for each of the mentioned subtechniques (Table 4), we test all them separately by asserting the probabilities p_{inf}, p_{NaN}, and p_0 (Table 6) to the operands. In $InputCG_{\infty}$, $InputCG_{NaN}$, and $InputCG_{mul0}$, the corresponding probability affects operand A, while in $InputCG_{add0}$ it affects operand C, the addend.

The maximum saving of 48.3% is available when p_{inf} or p_{NaN} is 1 ($InputCG_{\infty}$ and $InputCG_{NaN}$). The same savings are available when an operand is *NaN* or ∞, as in both cases the same hardware is clock-gated. The minimum probability p_{NaN} or p_{inf} (of any operand) necessary for saving power is the spot where the savings graph crosses the probability axis (16%).

When considering $InputCG_{mul0}$, the maximum saving is 40%, and the minimum probability p_0 (of any multiplicand) necessary for saving power is 18.5%.

Much lower maximum saving (2.3%) is available when the addend is a zero ($InputCG_{add0}$), as the adder consumes much less power than the multiplier (around 5 times in average). However, by combining these scenarios at the same time (which is reasonable to assume in a real application case), higher savings would be available. Therefore, even though the savings associated with detecting zero addend and clock-gating the adder and the corresponding aligner and input registers are not enough to justify its existence by itself, supporting this case improves overall savings of the complete *InputCG* technique when a real, combined scenario is considered. Since it shares some hardware with other *InputCG* subtechniques, the overhead of adding it is less then the saving it

10. Critical path optimization by adjusting the position of the flip-flops.

11. The gate signal in this case could be generated from vector kill instruction `KILLV` (similar to `VRIP` instruction in Cray X1 instruction set [41]).

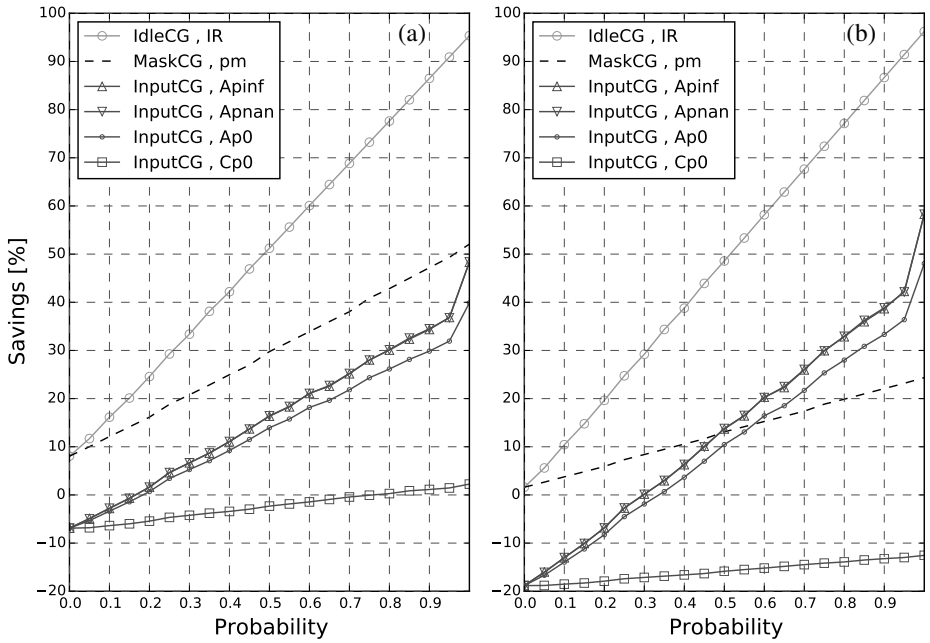


Fig. 5: Evaluation of power savings over the baseline (*NoCG*), as a function of *uKernel* parameters for *IdleCG*, *MaskCG*, and *InputCG* for 4-stage (a) and 1-stage (b) 2-lane VFU. For each graph, only one *uKernel* probability parameter from Table 6 is assumed to be variable while other parameters are zero. This is indicated in the legend with *technique*, *probability* pairs.

TABLE 9: Area efficiency of the proposed clock-gating techniques against the baseline (*NoCG*) for n_S =1 and 4.

CG_{type}	n_S	*IdleCG/MaskCG*	*ScalarCG*	*ImplCG*	*StageCG*	*InputCG*	*ALLCG*
Ratio(%)	1	98.1	153.8	184.6	98.1	129.0	185.4
	4	96.3	104.6	125.4	96.9	107.3	148.0

can achieve.

The power overhead of the added hardware can be identified in the case when the probability is 0, i.e. when *InputCG* is never active. The cost is a bit higher than expected taking into account the amount of additional logic that we include (detecting, bypassing, and clock-gating logic). In line with our discussion of area results, Chisel generated designs sometimes suffer from unexpected overhead, and our initial experiments confirm it. However, we observe it significantly decreases as n_S increases, thus, we expect this to be negligible for high n_S.

✦ ***IdleCG***. This technique provides savings between 8% and 95.3%. Although it uses the same hardware as *MaskCG*, the savings are higher in average because here the input data is stable, while in *MaskCG* it is not, thus incurring some switching albeit the first pipeline stage is gated. The saving attainable when *IR*=1 (95.3%) is the maximum possible power reduction when the VFU is idle.

The results for *ScalarCG* and *ImplCG* are shown in Table 8:

✦ ***ScalarCG***. Savings are available for all the combinations of *INSTYP, ADD/MUL, MULS,* and *ADDS uKernel* parameters, i.e. for all vector floating point instructions. Not surprisingly, as the internal multiplier dissipates more power than the adder, latching multiplicand's mantissas provides more savings than latching addend's mantissa (`FPFMAVSV` and `FPADDV` vs. `FPFMAVVS` and `FPMULV`).

✦ ***ImplCG***. This technique significantly improves savings for `FPADDV/FPMULV` instructions compared to *ScalarCG*. The highest savings, and the largest improvements compared to *ScalarCG*, are available for `FPADDV` and `FPADDVS` instructions as in these cases the multiplier is gated. As for *ScalarCG*, there are savings for all vector floating-point instructions. Incidentally, we can observe that for `FPADDV` and `FPADDVS` power savings are higher for 1- than for 4-stage VFU. The reason is that the mantissa multiplier contributes a higher percentage of total power when only one pipeline stage is present.

6.3 Application-Based Combined Power Evaluation

Table 10 shows the evaluation of the presented clock-gating proposals as well as workload profiling results for all combinations of *app-uBench* and vector parameters (MV_L and n_L), while figures 6, 7, and 8 visualize the key metrics from the table. For the reasons explained above, we focus on VFUs with 4 stages. The results shown in the table and the figures are:

- *S* - power savings percentage,
- *Active Execution / Idle Execution* - Active/Idle VFU execution, i.e percentage of *app-uBench* execution time that VFU is active/idle,
- *IdleCG, ScalarCG, ImplCG, InputCG* and *InputCG* subtechniques efficiency - percentage of execution time that the corresponding technique is used,
- *AllCG* efficiency - percentage of execution time that any clock-gating technique is used,
- *ActiveCG* efficiency - percentage of execution time that any active clock-gating technique is used, and
- *ActiveExeCG* efficiency - percentage of active VFU execution that any active clock-gating technique is used. Therefore, it disregards idle cycles and considers only the cycles in which

there is a vector instruction executing in the unit, rather than the total execution time like in ActiveCG.

We show the profiling results as well to understand the VFU behavior and where the savings come from. Data for *MaskCG*, $InputCG_{NaN}$, and $InputCG_{inf}$ are not present in the Table 10 as they are 0%. The reason is that the selected *app-uBenchs* do not have vector mask instructions. Also, none of the input values are *NaNs* nor infinities. However, abundant vector mask instructions could be found in any vector workload that has conditional execution [42], so in this kind of workloads we can expect fair savings as the result of *MaskCG* technique. Regarding *NaNs* and infinities, for some other applications and/or input data sets their occurrence might be more common, thus, the benefit of $InputCG_{NaN}$ and $InputCG_{inf}$ subtechniques will be visible. Common cases of *NaN* and infinity processing are explained in [21].

Figure 6 and Table 10 reveal that *AllCG* efficiency is very high, i.e. clock-gating is often used almost 100% of total execution time. This is a consequence of the fact that the proposed clock-gating techniques are used during both idle and active VFU execution (i.e. both *ActiveCG* and *IdleCG* are used). Due to this very high *AllCG* efficiency, the power savings are also fairly high. We observe that power savings are available for practically all the combinations of *app-uBenchs* and vector parameters. The highest savings are obtained for computer vision *app-uBenchs* (Facerec and Disparity) and are between 60% and 80%. The only case in which the techniques do not result in savings is K-Means, MV_L=16 and n_L=4. There are two reasons for that: (1) the clock-gating efficiency (i.e. the percentage of execution time that any clock-gating technique is used) is not high and (2) with n_L=4 and MV_L=16 the effective vector length per lane is 4 which makes *ImplCG* (the most used technique in this case) less fruitful since it is used only 3 consecutive cycles per vector on each lane (which as a result has more switching activity in clock-gating and bypassing logic). Additionally, from Table 10, we observe that presented novel ideas/approaches (*ActiveCG*) provide significant savings in addition to the standard one (*IdleCG*).

Figure 7 shows that ratio of active and idle VFU execution varies across *app-uBenchs* and vector parameters, and explains the nature for each combination of parameters. There are situations in which the VFU is most of the time active and vice versa. However, we can notice there is a trend that vector processors with MV_L of 128 have its VFU most of the time active (busy) as with longer vectors the effects of cache misses are diminished. *IdleCG* is used whenever the VFU is idle, thus, *IdleCG* efficiency inside these idle periods is 100%. When considering active VFU execution, the efficiency (*ActiveExeCG*) varies across the *app-uBenchs* and vector parameters and is shown on Figure 8. As we can observe from the figure, a very high percentage of the time at least one of the active clock-gating techniques is used and depending on the benchmark it goes up to 100%. Table 10 shows that in all these cases the used techniques are some variants of *ImplCG*[12] and *InputCG*. Also, there are cases when these techniques overlap.

ActiveCG techniques can arbitrarily overlap and there are two potential kinds of overlaps. The first group of overlaps refers to the cases when the techniques jointly produce higher savings than each technique separately. This happens when the techniques target different hardware. For example, when we have a zero addend inside a `FPADDV` instruction, $InputCG_{add0}$ gates the mantissa adder and the aligner, while *ImplCG* gates the input register of the *A* operand and the mantissa multiplier. The second group of overlaps happens when one technique gates just part of the hardware that another technique gates. In these cases, the savings are equal to the savings of the technique that has a larger scope. For example, if the corresponding bit in VMR is '0' and the current instruction is `FPFMAVVS`, the savings are going to be equal to the savings achieved by *MaskCG* alone.

12. As explained before, *ScalarCG* is integrated in *ImplCG*.

7 Conclusions

In this research, we extensively identify, propose, and evaluate the most suitable clock-gating techniques for vector FMA (VFU) considering peak performance, and focusing on the active operating mode. We propose techniques that are either (1) completely novel ideas to lower the power of VFU using active clock-gating (e.g. vector instructions with scalar operand or vector masking) or (2) ideas that exist in some form in scalar architectures and that we extend to achieve more savings by taking advantage of vector processing characteristics. We find that each of the proposed optimizations achieves power reductions while maintaining the performance. As a consequence of this fact, sometimes an area increase is observed.

An in-depth evaluation is performed, and each of the techniques is evaluated separately as well as combined with other techniques. For this evaluation, both synthetic and real application-based benchmarks are employed. We considered a variety of benchmarks with different behavior to assure a fair evaluation and general conclusions.

In the case of active 4-stage vector fused multiply-add unit (VFMA) with 2 lanes actively operating at the peak performance, power savings are up to 52% are available when using a single technique. Regarding the vector instruction-dependent techniques that we propose, we observe savings for all floating-point vector instructions.

Testing all the techniques together and using real application benchmarks (especially computer vision ones) reveals fairly high power reductions that go up to 80%. Clock-gating efficiency (percentage of time that some of the proposed techniques are used) is quite high, often close to 100%. When considering the efficiency of only active clock-gating techniques, this number is usually between 70% and 100%. We observe that these novel ideas/approaches (applied when VFU is active) provide significant savings in additional to the standard ones (idle VFU). Moreover, we notice the trend that savings for the proposed techniques rise with the number of pipeline stages.

We performed this research in a fully parameterizable, scalable and automated manner using simulators and tools at many levels. Although targeting floating-point FMA, as the major consumer among all functional units, similar low power techniques as well as the framework could be re-tailored for other vector functional units as well. We would also like to stress that the combination of Chisel-based generators and state-of-the-art synthesis and PnR tools is a powerful tool for flexible hardware generation with Verilog-like quality of results.

References

[1] I. Ratković, O. Palomar, M. Stanić, O. Unsal, A. Cristal, and M. Valero, "A Fully Parameterizable Low Power Design of Vector Fused Multiply-Add Using Active Clock-Gating

TABLE 10: Power savings and clock-gating statistics. All the results are expressed in percentages.

app-uBench	MVL	nL	S	AllCG	Active Exe	ActiveCG	ActiveExeCG	Idle Exe, IdleCG	ScalarCG/ImplCG	InputCG	InputCG_{mul0}	Input_{add0}
Facerec	16	1	70.32	98.96	16.68	15.63	93.74	83.32	15.63	7.50	0.00	7.50
		2	73.99	98.83	9.36	8.19	87.49	90.64	8.19	4.21	0.00	4.21
		4	76.48	98.77	4.91	3.68	74.99	95.09	3.68	2.21	0.00	2.21
	128	1	59.84	99.65	44.33	43.98	99.21	55.67	43.98	19.90	0.00	19.90
		2	65.50	99.55	28.81	28.35	98.43	71.19	28.35	12.93	0.00	12.93
		4	71.04	99.47	16.83	16.30	96.87	83.17	16.30	7.55	0.00	7.55
Sphinx3	16	1	37.54	84.96	49.48	34.43	69.59	50.52	32.06	17.67	2.37	15.29
		2	38.61	85.05	43.04	28.10	65.27	56.96	26.03	15.37	2.07	13.30
		4	46.13	87.83	28.07	15.90	56.63	71.93	14.55	10.02	1.35	8.68
	128	1	30.55	82.25	69.67	51.92	74.53	30.33	48.31	25.27	3.61	21.66
		2	22.09	76.98	88.49	65.47	73.98	11.51	60.89	32.10	4.58	27.52
		4	23.43	77.70	82.26	59.96	72.89	17.74	55.70	29.84	4.26	25.58
Disparity	16	1	75.55	97.85	38.41	36.26	94.41	61.59	36.26	25.90	0.00	25.90
		2	74.20	97.10	24.38	21.49	88.11	75.62	21.49	16.44	0.00	16.44
		4	73.17	96.69	13.51	10.20	75.52	86.49	10.20	9.11	0.00	9.11
	128	1	79.69	100.00	81.38	81.38	100.00	18.62	82.60	60.81	0.00	60.81
		2	78.90	100.00	76.24	76.24	100.00	23.76	76.77	56.97	0.00	56.97
		4	77.71	99.45	61.60	61.05	99.10	38.40	61.05	46.03	0.00	46.03
K-Means	16	1	19.56	60.22	76.26	36.49	47.84	23.74	35.87	0.61	0.00	0.61
		2	4.61	47.97	94.05	42.02	44.68	5.95	41.27	0.76	0.00	0.76
		4	−11.24	45.00	88.93	33.93	38.15	11.07	33.21	0.71	0.00	0.71
	128	1	24.62	60.53	96.44	56.97	59.07	3.56	50.46	6.51	0.00	6.51
		2	22.91	58.65	100.00	58.65	58.65	0.00	51.90	6.75	0.00	6.75
		4	20.77	57.81	100.00	57.81	57.81	0.00	51.05	6.75	0.00	6.75

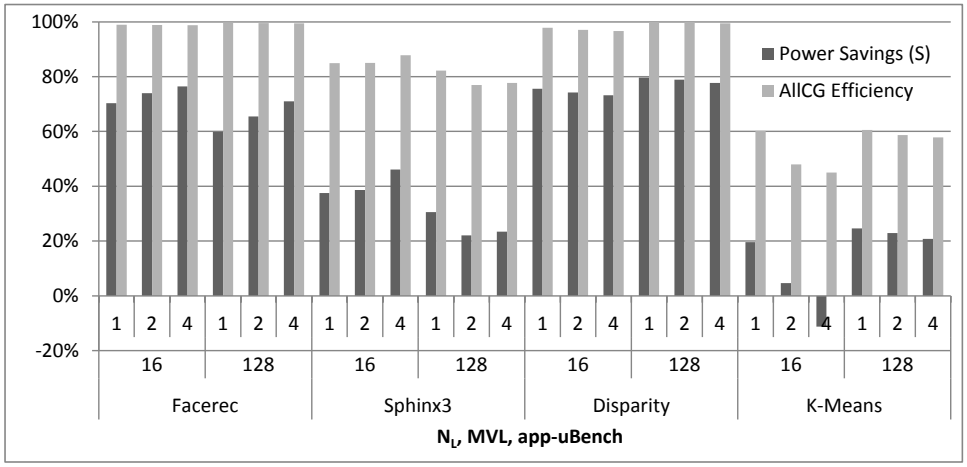


Fig. 6: Power savings (*S*) and *AllCG* efficiency.

Techniques," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2016, pp. 362–367.

[2] K. Asanović, "Vector Microprocessor," 1998, *PhD Thesis*, UC Berkeley.

[3] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Rfficiency in Data-Parallel accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 129–140.

[4] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtić, B. Keller, S. Bailey, M. Blagojević, P.-F. Chiu, H.-P. Le *et al.*, "A RISC-V Vector Processor With Simultaneous-Switching Switched-Capacitor DC–DC Converters in 28 nm FDSOI," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 930–942, 2016.

[5] R. Espasa, M. Valero, and J. E. Smith, "Vector Architectures: Past, Present and Future," in *Proceedings of the 12th International Conference on Supercomputing (SC)*, 1998, pp. 425–432.

[6] H. Li, S. Bhunia, Y. Chen, T. Vijaykumar, and K. Roy, "Deterministic Clock Gating for Microprocessor Power Reduction," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (ISCA)*, 2003,

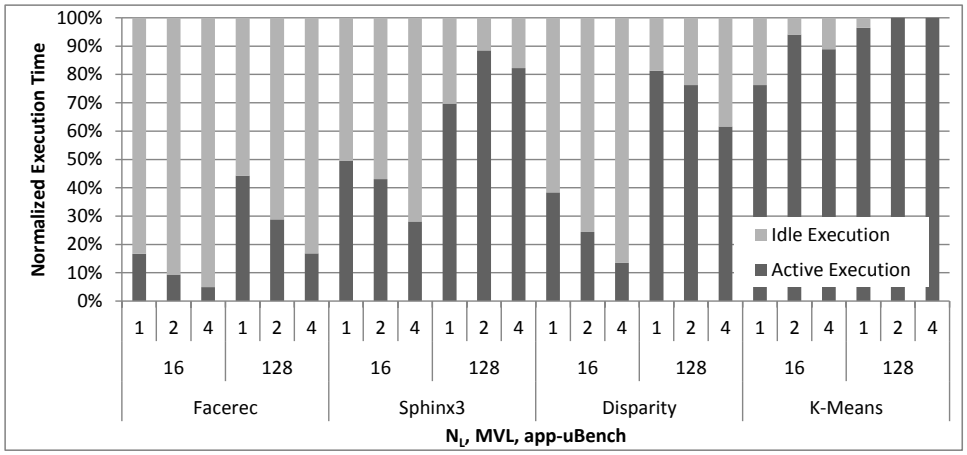


Fig. 7: *ActiveCG/IdleCG* - Active/Idle VFU execution.

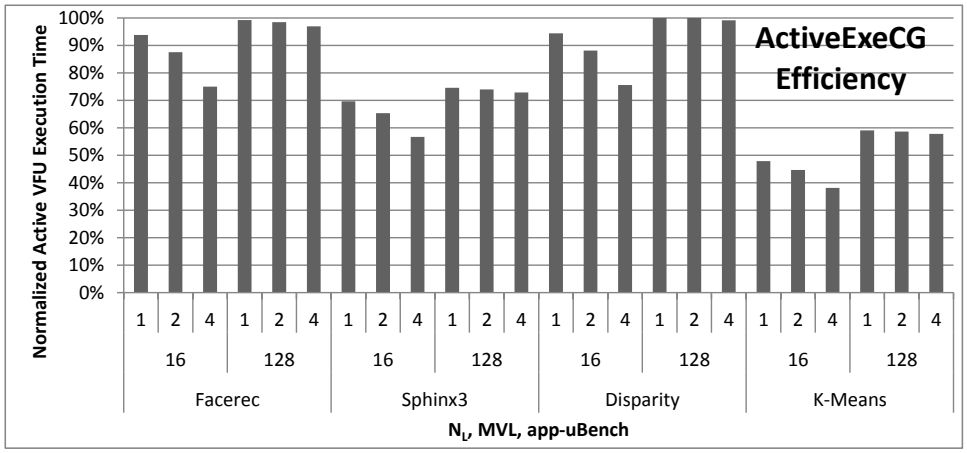


Fig. 8: *ActiveExeCG* efficiency.

pp. 113–122.

[7] N. Mohyuddin, K. Patel, and M. Pedram, "Deterministic Clock Gating to Eliminate Wasteful Activity due to Wrong-Path Instructions in Out-Of-Order Superscalar Processors," in *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 2009, pp. 166–172.

[8] J. Preiss, M. Boersma, and S. M. Mueller, "Advanced Clock-gating Schemes for Fused-Multiply-Add-Type Floating-Point Units," in *Proceedings of 19th IEEE Symposium on Computer Arithmetic (ARITH)*, 2009, pp. 48–56.

[9] I. Ratković, O. Palomar, M. Stanić, O. S. Unsal, A. Cristal, and M. Valero, "On the Selection of Adder Unit in Energy Efficient Vector Processing," in *Proceedings of 14th International Symposium on Quality Electronic Design (ISQED)*, 2013, pp. 143–150.

[10] I. Ratković, O. Palomar, M. Stanić, M. Duric, D. Pešić, O. Unsal, A. Cristal, and M. Valero, "Joint Circuit-System Design Space Exploration of Multiplier Unit Structure for Energy-Efficient Vector Processors," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2015, pp. 19–26.

[11] M. Stanic, O. Palomar, I. Ratković, M. Duric, O. Unsal, and A. Cristal, "VALib and SimpleVector: Tools for Rapid Initial Research on Vector Architectures," in *Proceedings of the 11th ACM Conference on Computing Frontiers (CS)*, 2014, p. 7.

[12] (2016) http://arm.com/.

[13] S. Momose, "NEC Vector Supercomputer: Its Present and Future," in *Sustained Simulation Performance*. Springer, 2015, pp. 95–105.

[14] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina *et al.*, "Tarantula: a Vector Extension to the Alpha Architecture," in *Proceedings of 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002, pp. 281–292.

[15] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B.-W. Lee *et al.*, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, 2010.

[16] B. Nikolic, "Simpler, more efficient design," in *Proceedings of 41st European Solid-State Circuits Conference (ESSCIRC)*, 2015, pp. 20–25.

[17] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012, pp. 1216–1225.

[18] K. R. Gandhi and N. R. Mahapatra, "A Study of Hardware Techniques that Dynamically Exploit Frequent Operands to Reduce Power Consumption in Integer Function Units," in

Proceedings of 21st International Conference on Computer Design (ICCAD), 2003, pp. 426–428.

[19] (2016) https://software.intel.com/en-us/blogs/2013/avx-512-instructions/.

[20] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[21] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[22] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, 2000.

[23] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[24] E. Hokenek, R. K. Montoye, and P. W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *Solid-State Circuits, IEEE Journal of*, vol. 25, no. 5, pp. 1207–1213, 1990.

[25] T. Xanthopoulos and A. P. Chandrakasan, "A Low-Power IDCT Macrocell for MPEG-2 MP@ ML Exploiting Data Distribution Properties for Minimal Activity," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 5, pp. 693–703, 1999.

[26] (2016) https://github.com/ucb-bar/berkeley-hardfloat/.

[27] S. Balakrishnan and G. S. Sohi, "Exploiting Value Locality in Physical Register Files," in *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture (2003)*. IEEE, 2003, pp. 265–276.

[28] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005, pp. 74–85.

[29] J. Hennessy, D. Patterson, and K. Asanović, *Computer Architecture, Appendix G*. MK, 2011.

[30] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye, and G. Schwoerer, "The Cray BlackWidow: a Highly Scalable Vector Multiprocessor," in *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, 2007, p. 17.

[31] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *40th European Solid State Circuits Conference (ESSCIRC)*, 2014, pp. 199–202.

[32] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtic, B. Keller, S. Bailey, M. Blagojevic, P.-F. Chiu, H.-P. Le *et al.*, "A RISC-V Vector Processor with Tightly-Integrated Switched-Sapacitor DC-DC Converters in 28nm FDSOI," in *Proceedings of Symposium on VLSI Circuits*. IEEE, 2015, pp. C316–C317.

[33] (2016) https://www.arm.com/products/processors /cortex-a.

[34] (2016) http://www.anandtech.com/show/6936/intels-silvermont-architecture-revealed-getting-serious-about-mobile.

[35] (2016) https://www.qualcomm.com/products/snapdragon/cpu-specifications.

[36] J. L. Henning, "SPEC CPU2006 enchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[37] S. K. Rethinagiri, O. Palomar, A. Sobe, G. Yalcin, T. Knauth, R. T. Gil, P. Prieto, M. Schneegaß, A. Cristal, O. Unsal *et al.*, "ParaDIME: Parallel Distributed Infrastructure for Minimization of Energy for Data Centers," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1174–1189, 2015.

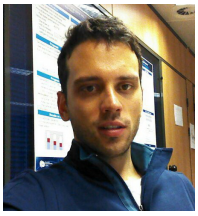
[38] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego vision benchmark suite," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 55–64.

[39] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján, "An Empirical Evaluation of High-Level Synthesis Languages and Tools for Database Acceleration," in *Proceedings of 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[40] I. S. Committee *et al.*, "754-2008 IEEE standard for floating-point arithmetic," *IEEE Computer Society Std*, vol. 2008, 2008.

[41] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[42] J. E. Smith, G. Faanes, and R. Sugumar, "Vector Instruction Set Support for Conditional Operations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 260–269.



Ivan Ratković received the BS and MS degree in Electrical Engineering and Computer Science from the University of Belgrade (Serbia) and the PhD degree in Computer Architecture from Polytecnic University of Catalonia (Spain). He is currently CPU R&D engineer at Esperanto Technologies and Semidynamics. In the past, Dr. Ratković was involved with BSC Microsoft Research Center, Barcelona Supercomputing Center, and Berkeley Wireless Research Center. His research interests include low power design, computer architecture, vector and SIMD processors, digital arithmetic, VLSI design flows, and embedded systems.

Oscar Palomar received a B.S. in Computer Science and PhD degree on Computer Architecture from the Polytechnic University of Catalonia (UPC), Spain. He joined the Computer Architecture for Parallel Paradigms research group at the Barcelona Supercomputing Center as a post-doc. There he led research on vector architecture and worked on the hardware architecture workpackage in the ParaDIME project. He was granted a Newton International Fellowship from the Royal Society, joining the Advanced Processor Technologies research group at the University of Manchester. His current research interests are computer vision algorithms, FPGA acceleration, low power computer architectures and vector processors.

Milan Stanić received the BS degree in Electrical Engineering and Computer Science from the University of Belgrade (Serbia) and the MS and PhD degree in Computer Architecture from Polytecnic University of Catalonia (Spain). He worked as a researcher at Barcelona Supercomputing Center and he is currently software developer at ASML His research interests include Computer architecture, SIMD and vector architectures, high-performance computing, many-core and heterogeneous architectures, workload characterization, simulation, ISA and microarchitecture development, power and energy efficiency, code optimization.

Osman Sabri Ünsal is co-leader of the Architectural Support for Programming Models group at the Barcelona Supercomputing Center. In the past, Dr. Ünsal was involved with Intel Microprocessor Research Labs, BSC Microsoft Research Center, and Intel/BSC Exascale Lab. He holds BS, MS, and PhD degrees in electrical and computer engineering from Istanbul Technical University, Brown University, and University of Massachusetts, Amherst, respectively. His research interests are in computer architecture, low-power and energy-efficient computing, fault-tolerance and transactional memory.

Adrian Cristal received the licenciatura in Compuer Science from Universidad de Buenos Aires (FCEN) in 1995 and the PhD. degree in Computer Science in 2006, from the Universitat Politecnica de Catalunya (UPC), Spain. From 1992 to 1995 he has been lecturing in Neural Network and Compiler Design. In UPC, from 2003 to 2006 he has been lecturing on computer organization. Currently, and since 2006, he is researcher in Computer Architecture group at Barcelona Supercomputing Center. He is currently co-manager of the Computer Architecture for parallel paradigms. His research interests cover the areas of microarchitecture, multicore architectures, and programming models for multicore architectures. He has published around 60 publications in these topics and participated in several research projects with other universities and industries, in framework of the European Union programmes or in direct collaboration with technology leading companies.

Mateo Valero is full professor at Computer Architecture Department at the Universitat Politcnica de Catalunya and the Director Barcelona Supercomputing Center. His research focuses on high-performance computer architectures. Published 700 papers. Served in organization of 300 international conferences. Main awards: Seymour Cray, Eckert-Mauchly, Harry Goode, ACM Distinguished Service, ”Hall of Fame” member IST European Program, King Jaime I in research, two Spanish National Awards on Informatics and Engineering. He is Honorary Doctorate by 9 Universities. He is a fellow of IEEE and ACM and is and Intel Distinguished Research Fellow. He is member of 5 academies.