Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices

Michael Gautschi, *Student Member, IEEE*, Pasquale Davide Schiavone, *Student Member, IEEE*, Andreas Traber, Igor Loi, *Member, IEEE*, Antonio Pullini, *Student Member, IEEE*, Davide Rossi, *Member, IEEE*, Eric Flamand, Frank K. Gürkaynak, and Luca Benini, *Fellow, IEEE*

***Abstract*— Endpoint devices for Internet-of-Things not only need to work under extremely tight power envelope of a few milliwatts, but also need to be flexible in their computing capabilities, from a few kOPS to GOPS. Near-threshold (NT) operation can achieve higher energy efficiency, and the performance scalability can be gained through parallelism. In this paper, we describe the design of an open-source RISC-V processor core specifically designed for NT operation in tightly coupled multicore clusters. We introduce instruction extensions and microarchitectural optimizations to increase the computational density and to minimize the pressure toward the shared-memory hierarchy. For typical data-intensive sensor processing workloads, the proposed core is, on average, 3.5× faster and 3.2× more energy efficient, thanks to a smart L0 buffer to reduce cache access contentions and support for compressed instructions. Single Instruction Multiple Data extensions, such as dot products, and a built-in L0 storage further reduce the shared-memory accesses by 8× reducing contentions by 3.2×. With four NT-optimized cores, the cluster is operational from 0.6 to 1.2 V, achieving a peak efficiency of 67 MOPS/mW in a low-cost 65-nm bulk CMOS technology. In a low-power 28-nm FD-SOI process, a peak efficiency of 193 MOPS/mW (40 MHz and 1 mW) can be achieved.**

***Index Terms*— Instruction set architecture (ISA) extensions, Internet-of-Things, multicore, RISC-V, ultralow power (ULP).**

I. Introduction

In THE last decade, we have been exposed to an increasing demand for small, and battery-powered Internet-of-Things (IoT) endpoint devices that are controlled by a microcontroller (MCU), interact with the environment, and communicate over a low-power wireless channel. Such devices require ultralow-power (ULP) circuits, which interact with sensors. It is expected that the demand for sensors and processing platforms in the IoT segment will skyrocket over the next years [1]. Current IoT endpoint devices integrate multiple sensors, allowing for sensor fusion, and are built around an MCU, which is mainly used for controlling and lightweight processing. Since endpoint devices are often untethered, they must be very inexpensive to maintain and operate, which requires ULP operation. In addition, such devices should be scalable in performance and energy efficiency, because bandwidth requirements vary from ECG sensors to cameras, to microphone arrays and so does the required processing power. As the power of wireless (and wired) communication from the endpoint to the higher level nodes in the IoT hierarchy is still dominating the overall power budget [2], it is highly desirable to reduce the amount of transmitted data by doing more complex near-sensor processing, such as feature extraction, recognition, or classifications [3]. A simple MCU is very efficient for controlling purposes and lightweight processing, but not powerful nor efficient enough to run more complex algorithms on parallel sensor data streams [4].

One approach to achieve a higher energy efficiency and performance is to equip the MCU with digital signal processing (DSP) engines, which allow to extract the heart rate of an ECG signal more efficiently and reduce the transmission costs by only transmitting the extracted feature [5]. Such DSPs achieve a very high performance when processing data, but are not as flexible as a processor and also harder to program. Even higher energy efficiency can be achieved with dedicated accelerators. In a biomedical platform, for seizure detection, it has been shown that it is possible to speed up fast Fourier transformations (FFTs) by a dedicated hardware block, which is controlled by an MCU [6]. Such a combination of MCU and FFT-accelerator is superior in performance, but also very specialized and, hence, not very flexible nor scalable.

The question arises if it is possible to build a flexible, scalable, and energy-efficient platform with programmable cores consuming only a couple of milliwatts. We claim that, although the energy efficiency of a custom hardware block can never be achieved with programmable cores, it is possible to build a flexible and scalable multicore platform with a very high energy efficiency. ULP operations can be achieved by exploiting the near-threshold (NT) voltage regime where transistors become more energy efficient [7]. The loss in performance (frequency) can be compensated by exploiting parallel computing. Such systems can outperform single-core equivalents due to the fact that they can operate at a lower supply voltage to achieve the same throughput [8].

Manuscript received August 17, 2016; revised December 2, 2016; accepted January 8, 2017. Date of publication February 24, 2017; date of current version September 25, 2017. This work was supported in part by the FP7 ERC Advance Project MULTITHERMAN under Grant 291125, and in part by Micropower Deep Learning under Grant 162524 Project funded by the Swiss NSF.

M. Gautschi, P. D. Schiavone, A. Pullini, and F. K. Gürkaynak are with the Integrated Systems Laboratory, ETH Zürich, 8092 Zurich, Switzerland (e-mail: gautschi@iis.ee.ethz.ch; Schiavone@iis.ee.ethz.ch; pullinia@iis.ee.ethz.ch; kgf@iis.ee.ethz.ch).

A. Traber was with the Integrated Systems Laboratory, Electrical engineering and information technology, ETH Zürich, 8702 Zollikon, Switzerland. He is now with Advanced Circuit Pursuit, 8702 Zollikon, Switzerland (e-mail: atraber@iis.ee.ethz.ch).

L. Benini is with Integrated Systems Laboratory, ETH Zürich, 8092 Zurich, Switzerland, and also with the University of Bologna, 40126 Bologna, Italy (e-mail: lbenini@iis.ee.ethz.ch).

E. Flamand is with Integrated Systems Laboratory, ETH Zürich, 8092 Zurich, Switzerland, and also with GreenWaves Technologies, 38190 Villard-Bonnot, France (e-mail: eflamand@iis.ee.ethz.ch).

D. Rossi and I. Loi are with the University of Bologna, 40136 Bologna, Italy, and also with GreenWaves Technologies, 38190 Villard-Bonnot, France.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2654506

A major challenge in low-power multicore design is the memory hierarchy. Low-power MCUs typically fetch data and instructions from single-ported dedicated memories. Such a simple memory configuration is not adequate for a multicore system, but on the other hand, complex multicore cache hierarchies are not compatible with extremely tight power budgets. Scratchpad memories offer a good alternative to data caches as they are smaller and cheaper to access [9]. Another advantage is that such tightly coupled-data-memories (TCDMs) can be shared in a multicore system and allow the cores to work on the same data structure without coherency hardware overhead. One limiting factor in decreasing the supply voltage is memories, which typically start failing first. The introduction of standard cell–memories (SCMs), allows for NT operation and consumes fewer milliwatts at the price of additional area [10]. In any case, memory access time and energy are a major concern in the design of a processor pipeline optimized for integration in an ULP multicore cluster.

An open-source instruction set architecture (ISA) is a desirable starting point for an IoT core, as it can potentially decrease dependence from a single IP provider and cut cost, while, at the same time, allowing freedom for application-specific instruction extensions. Therefore, we focus on building a microarchitecture based on the RISC-V ISA [11], which achieves similar performance and code density to the state-of-the art MCUs based on a proprietary ISA, such as ARM Cortex-M series cores. The focus of this paper is on ISA and microarchitecture optimization, specifically targeting NT parallel operation, when cores are embedded in a tightly coupled shared-memory cluster. Our main contributions can be summarized as follows:

1) an optimized instruction fetch unit, featuring an L0 buffer with prefetch capability and support for hardware-loop handling, which significantly decreases bandwidth and power in the instruction memory hierarchy;
2) an optimized execution stage supporting flexible fixed-point and saturated arithmetic operations as well as single-intstruction—multiple-data (SIMD) extensions, including dot-product and shuffle instructions, and misaligned load support that greatly reduce the load–store traffic to data memory while maximizing computational efficiency;
3) an optimized pipeline architecture and logic, register transfer level (RTL) design, and synthesis flow, which minimize redundant switching activity and expose one full cycle for accessing instruction caches and two cycles for the shared TCDM, thereby giving ample timing budget to memory access that is most critical in NT shared-memory operation.

The back-end of the RISC-V GCC compiler has been extended with fixed-point support, hardware loops, postincrement addressing modes, and SIMD instructions.[1] We show that with the help of the additional instructions and microarchitectural enhancements, signal processing kernels, such as filters, convolutions, and so on, can be completed 3.5× faster on average, leading to a 3.2× higher energy efficiency. We also show that convolutions can be optimized in C with intrinsics by using the dot-product and shuffle instruction, which reduces accesses and contentions for the shared data memory utilizing the register file (RF) as L0-storage, allowing the system to achieve a near linear speedup of 3.9× when using four cores instead of one.

The following sections will first summarize the related work in Section II and explain the target multicore platform and its applications in Section III. The core architecture is presented in Section IV with programming examples in Section V. Finally, Section VI discusses the experimental results and in Section VII we draw our conclusions.

II. Related Work

The majority of IoT endpoint devices use single-core MCUs for controlling and lightweight processing. Single-issue in-order cores with a high Instructions per clock (IPC) are typically more energy efficient as no operations have to be repeated due to mispredictions and speculation [12]. Commercial products often use ARM processors from the Cortex-M families [13]–[15], which work above the threshold voltage. Smart peripheral control, power managers, and combination with nonvolatile memories allow these systems to consume only tens of milliwatts in active state, and a few microWatts in sleep mode. In this paper, we will focus on active power and energy minimization. Idle and sleep reduction techniques are surveyed in [16].

Several MCUs in the academic domain and a few commercial ones exploit the use of NT operation to achieve energy-efficiencies in active state down to 10 pJ/op [14], [17]. These designs take advantage of standard cells and memories, which are functional at very low voltages. It is even possible to operate in the subthreshold regime and achieve excellent energy-efficiencies of 2.6 pJ/op at 320 mV and 833 kHz [18]. Such systems can consume well below 1-mW active power, but reach their limit when more than a few MOPS of computing power are required as for near-sensor processing for IoT endpoint devices.

One way to increase performance while maintaining power in the mW range is to use DSPs, which make use of several optimizations for data-intensive kernels, such as parallelism through very long instruction words (VLIWs) and specialized memory architectures. Low-power VLIW DSPs operating in the range of 3.6–587 MHz, where the chip consumes 720 μW–113 mW, have been proposed [19]. It is also possible to achieve energy-efficiencies in the range of a couple of pJ/op. Wilson *et al.* [20] designed a VLIW DSP for embedded $F_{\max}$ tracking with only 62 pJ/op at 0.53 V, and a 16-b low-power fixed-point DSP with only about 5 pJ/op has been proposed by Le *et al.* [21]. DSPs typically have zero-overhead loops to eliminate branch overheads and can execute operations in parallel, but are harder to program than general purpose processors. In this paper, we borrow several ideas from the DSP domain, but we still maintain complete compatibility with the streamlined and clean load–store RISC-V ISA, and full C-compiler support (no assembly level optimization needed) with a simple in-order four-stage pipeline with Harvard memory access.

[1]Both the RTL HW description and the GCC compiler are open source and can be downloaded at http://www.pulp-platform.org

Since typical sensor data from ADCs use 16 b or less, there is a major trend to support SIMD operations in programmable cores, such as in the ARM Cortex M4 [22], which supports DSP functionalities while remaining energy efficient (32.8 μW/MHz in 90-nm low-power technology [22]). The instruction set contains DSP instructions, which offer a higher throughput as multiple data elements can be processed in parallel. ARM even provides a Cortex-M Software Interface Standard DSP library, which contains several functions, which are optimized with built-ins [22]. Performance can, for example, be increased with a dot-product instruction, which accumulates two 16-b$\times$16-b multiplication results in a single cycle. Such dot-product operations are suitable for media processing applications [23] and even versions with 8-b inputs can be implemented [24] and lead to high speedups. While 16-b dot products are supported by ARM, the 8-b equivalent is not.

Dedicated hardware accelerators, coupled with a simple processor for control, can be used for specific tasks offering the ultimate energy efficiency. As an example, a battery-powered multisensor acquisition system, which is controlled by an ARM Cortex M0 and contains hardware accelerators for heart rate detection, has been proposed by Konijnenburg *et al.* [4]. Ickes *et al.* [25] propose another system with FFT and Finite inpulse response (FIR) filter accelerators, which are controlled by a 16-b CPU and achieves an energy efficiency of 10 pJ/op at 0.45 V. Also convolutional engines [26]–[28], which outperform general purpose architectures, have been reported. Hardwired accelerators are great to speed up certain tasks and can be a great improvement to a general purpose system as described in [28]. However, since there is no standard to interface such accelerators, and the fact that such systems cannot be scaled up prompts us to explore a scalable multicore platform, which covers all kinds of processing demands and is fully programmable.

NXP, TI, and other vendors offer heterogeneous dual-core MCUs, featuring decoupled execution of an extremely energy-efficient Cortex M0+ for control tasks, and a Cortex M3 or M4 for computationally intensive kernels [15], [29]. Such systems are not scalable in terms of memory hierarchy and a number of cores as cores are embedded in their own subsystems and M0+ cannot run M3/4 executables. A few multicore platforms have already been proposed. For example, Neves *et al.* [30] proposed a multicore system with SIMD capabilities for biomedical sensors and Hsu *et al.* [31] implemented a reconfigurable SIMD vector machine, which is very powerful, energy efficient, and scalable in performance. Both are specialized processors, which are not general purpose, but optimized for a specific domain and very difficult to program efficiently. A very powerful multicore architecture consisting of 64 cores operating near the threshold voltage has been proposed in [32]. The system is designed for high-performance computing and its power consumption is in the range of a couple of hundreds milliwatts and not suitable for IoT applications.

III. Parallel Ultralow-Power Platform

In this section, we briefly review the parallel ULP (PULP) cluster architecture, which embeds our RISC-V cores. Interested readers are referred to [33]–[35] for more details. Our focus is on highlighting the key elements of the PULP memory hierarchy and the targeted workloads, which have been the main drivers of the core design. Many MCUs operate on instruction and data memories, and do not use caches.

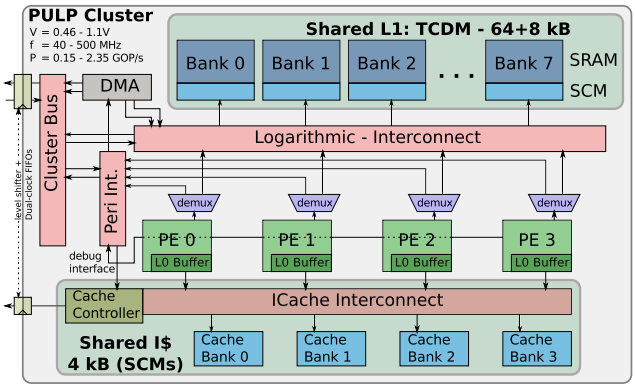

Fig. 1. PULP cluster with four cores, eight TCDM-banks, and a I$. Each TCDM-bank is split in 1-kB SCM and 8-kB SRAM. The I$ is fully implemented with SCMs allowing ultralow voltage operation. The processing elements (PEs) are RISC-V architectures with extended DSP capabilities.

Parallel execution in the PULP cluster requires a scalable memory architecture to achieve near-ideal speedups in parallel applications, while curtailing power. A PULP cluster supports a configurable number of cores with a shared instruction cache and scratchpad memory. Both memory ports have variable latency and can stall the pipeline. Fig. 1 shows a PULP cluster with four cores, and eight TCDM banks. A shared I$ is used to reduce the cost per core, leveraging the single-program-multiple-data nature of most parallel near-sensor processing application kernels. A tightly coupled DMA engine manages transfers between IO and L2 memory and the shared TCDM. Data access pattern predictability of key application kernels, the relatively low clock frequency target, and the tightly constrained area and power budget make a shared TCDM preferable over a collection of coherent private caches [9].

The data memories are split in area-efficient SRAM and energy-efficient SCM blocks. Since SCMs are built of standard cells, it is possible to scale the supply voltage and operate near the threshold voltage of transistors [10]. By clock gating the SRAM blocks and the individual cores, it is possible to scale down the system to a simple single-core architecture, which can act as an extremely energy-efficient controller operating near the threshold voltage where only SCMs are active. If more processing power is required, the power manager of the cluster can wake up more cores, and by dynamic-voltage and frequency-scaling (DVFS), the performance of a 28-nm Fully depleted Silicon on insulator (FDSOI) implementation can be adjusted from a couple of operations per second to 2.35 GOPS by scaling the voltage from 0.32 to 1.15 V where the cores run at 630 MHz (throughput = #cores$\cdot$ IPC$\cdot$ frequency). The PULP cluster has been successfully taped out with OpenRISC and RISC-V cores [33]–[35], and its latest version achieves a top energy efficiency in NT operation of 193 MOPS/mW in 28-nm FDSOI technology.

In this paper, we focus on ISA extensions, microarchitecture optimizations, and RTL design to balance pipeline stages and further improve the energy efficiency of the RISC-V

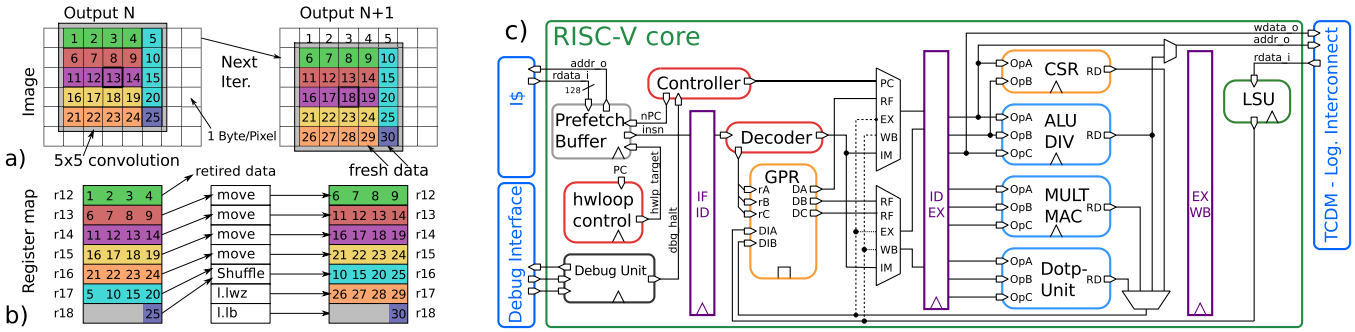

Fig. 2. (a) Example of a 5×5 convolution to compute output N and N+1 in the image domain and (b) how the register content is efficiently updated using the shuffle instruction. One 5×5 convolution requires exactly 4 *move*, one *shuffle*, and 2 loads to prepare the operands and 1 *dotp*-, and 6 *sdotp*-operations to compute one output pixel. (c) Simplified block diagram of the RISC-V core architecture showing its four pipeline stages and all functional blocks.

cluster. Memory accesses for both data and instructions are the most critical operations that contribute to energy consumption in a microprocessor as we will show in Section VI-B, and we will present several methods to reduce costly memory access operations for both data and instructions.

The RISC-V ISA used as a starting point, in this paper, already supports a compressed ISA extension that allows several common instructions to be coded in 16 b instead of 32 b, reducing the pressure on the instruction cache (I$). The PULP architecture is designed to use a shared I$ to decrease the per-core instruction fetch cost. A shared I$ is especially efficient when the cores execute the same parallel program (e.g., when using OpenMP). Since PULP, unlike GP-GPUs, does not enforce strict single-instruction-multiple data execution, the shared cache may produce stalls that add to energy losses. We have, therefore, added an L0-buffer into the core to reduce access contentions, reduce instruction read power, and shorten the critical path between I$ and the core. In addition, we have modified the L0-buffer to cope with nonaligned accesses due to compressed instructions more efficiently as discussed in Section IV-B. As for the data access, we will illustrate the major gains that can be achieved by our extensions using the example of a 2-D-convolution implementation that is widely used in many applications in the image processing domain [36]. The 2-D convolutions are not only very pervasive operators in the state-of-the-art sensor data processing, but they also contain the same iterative patterns as digital filtering (FIR, IIR) and statistical signal processing (matrix-vector product) operations. The main task of a convolution operation is to multiply subsequent values with a set of weights and accumulate all values for which multiply–accumulate (MAC) units can be used.

A straightforward implementation of a 5×5 2-D convolution of a 64×64 pixel image on the RISC-V architecture with compressed instructions takes 625k cycles, of which 389k are for various load/store (*ld/st*) operations to and from memory and 100k are actual operations calculating the values. The same algorithm when parallelized to four cores on the PULP requires only 173k cycles per convolution on average. However, increased parallelism also results in increased demand to the shared TCDM, which puts additional strain on the TCDM design. TCDM contentions can be reduced by increasing the number of banks, which unfortunately, in turn, increases power per-access due to more complex interconnection and increased memory area, because smaller and less dense memories have to be used. Improving the arithmetic capabilities of the cores will further exacerbate this problem as it will lead to a higher portion of *ld/st* instructions, which will result in even more contentions (up to 20%).

A common technique to reduce the data bandwidth is to make use of vector instructions that process instead of a single 32-b value, two 16-b values, or four 8-b values simultaneously. Most arithmetic and logic operations can be parallelized in subfields with very little overhead. As explained in Section IV, we have added a large variety of vector instructions to the RISC-V ISA, including dot-product (*dotp, sdotp*) instructions that can multiply four pairs of 8-b numbers and accumulate them in a single instruction.

Allowing one data word to hold multiple values directly reduces the memory bandwidth for data accesses, which is only useful if the data at subword level can be efficiently manipulated without additional instructions. Fig. 2 explains how a 5×5 2-D convolution can be computed with vector instructions. It can be seen that for each convolution step, 25 data values have to be multiplied with 25 constants and accumulated. If 8-b values are being used, registers can be used to hold vectors of four elements each. Once this calculation is completed, for the next step of the iteration, the five values of the first row will be discarded, and a new row of five values will be read. If these vectors are not aligned to word boundaries, an unaligned word has to be loaded from memory, which can be supported either in hardware or software. A software implementation requires at least five instructions to load two words and combine the pixels in a vector. In addition, it blocks registers from being used for actual computations, which is the reason why we support unaligned memory accesses directly in the load–store unit (LSU) by issuing two subsequent requests to the shared memory. Hence, unaligned words can be loaded in only two cycles. We also implement the *shuffle* instruction that can combine subwords from two registers in any combination. Fig. 2(b) shows how *move* and *shuffle* instructions are used to recombine the pixels in the right registers instead of loading all elements from memory. This allows to reduce RF pressure and the number of loads per iteration from 5 to 2. One

iteration can, therefore, be computed in about 20 instructions (four *move*, one *shuffle*, two *load*, seven *dotp*, one *store*, and five control flow), or 26 cycles on average. Thus, when all of the improvements are combined, the time to complete the operations can be reduced by 14.72× when compared with the original single-core implementation. Coupled by efficient DVFS methods, this performance gain can be used to increase energy efficiency by working at NT operation or to reduce computation time at the same operation voltage, allowing the system a wide range of tunability, which will be very important for future IoT systems that need to adapt to a variety of operating conditions and computation loads.

IV. RISC-V Core Microarchitecture

In this section, we will detail the extensions made to the RISC-V ISA and microarchitectural optimizations for increasing the efficiency of the processor core when working in a multicore platform with shared memory. The pipeline architecture will be described first in Section IV-A and the individual components of the core are discussed starting with Section IV-B where the IF-stage with the prefetch buffer is explained. Hardware-loop and postincrement extensions to the RISC-V ISA are explained in Sections IV-C and IV-D. The EX-stage with a more powerful dot-product-multiplier and a vector Arithmetic Logic Unit (ALU) with fixed-point support and a shuffle unit will be explained in Sections IV-E and IV-F.

A. Pipeline Architecture

The number of pipeline stages used to implement the processor core is one of the key design decisions. A higher number of pipeline stages allows for higher operating frequencies, increasing overall throughput, but also increases data and control hazards, which, in turn, reduces the IPC. For high-performance processors, where fast operation is crucial, optimizations, such as branch predictions, prefetch buffers, and speculation, can be introduced. However, these optimizations add to the overall power consumption and are usually not viable for ULP operation where typically a shallow pipeline of one to five stages with a high IPC is preferred.

The basic goal to keep the IPC high is to reduce stalls as much as possible. The ARM Cortex M4, for example, features a three-stage fetch-decode-execute pipeline with a single write-back port on the RF. The absence of a fourth pipeline stage and a separate write-back port for the LSU leads to stalls when executing load operations. ARM compilers reduce such stalls by grouping load/store operations as much as possible as this removes stalls for all subsequent load/store operations. A forwarding path from load operations to store operations allows to efficiently copy data hence and forth, but also limits the frequency of such a pipeline.

In the presented multicore setting, the shared TCDM can be accessed over a logarithmic interconnect, which adds delay for arbitration and MUXing to the data request and return paths. In contrast to a three-stage pipeline, our four-stage pipeline organization exposes two full cycles for memory accesses. This allows to balance TCDM request and return paths by employing *useful skew* techniques. Skewing the clock of the core allows to balance the request and return paths, thus achieving a higher clock frequency. The amount of skew depends on the available memory macros, the number of cores, and memory banks in the cluster. This 65-nm implementation with a 2.8-ns clock period for 1.08 worst case conditions uses a 0.5-ns skew. Even in a multicore context, the cluster is ultimately achieving frequencies of 350–400 MHz when implemented in 65 nm. Hence, thanks to a balanced pipeline, the PULP cluster reaches higher frequencies than commercially available MCUs operating in the range of 200 MHz (CortexM4-based MCUs in 90 nm).

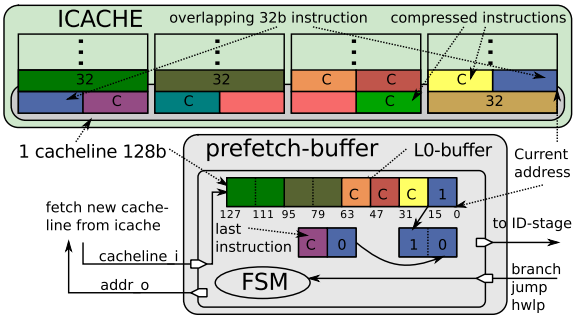

Fig. 3. Block Diagram of the prefetch buffer with an example when fetching full instructions over cache-line boundaries.

The four-stage pipeline requires a separate write-back port on the latch-based RF for the LSU, which increases its size by 1 kGE but also eliminates the write-back MUXes (ALU, Control Status Register (CSR), and Multiplier) from the critical path. The organization of the pipeline and all functional units are shown in Fig. 2. Since the critical path is mainly determined by the memory interface, it was possible to extend the ALU with fixed-point arithmetic and a more capable multiplier that supports *dotp* operations without incurring additional timing penalties.

B. Instruction Fetch Unit

Similar to the Thumb-2 instruction set of ARM, the RISC-V standard contains a specification for compressed instructions, which are only 16 b long and mirror full instructions with a reduced immediate size and RF address. As long as these 16-b instructions are aligned to 32-b word boundaries, they are easy to handle. A compressed instruction decoder detects and decompresses these instructions into standard 32-b instructions, and stalls the fetch unit for one cycle whenever two compressed instructions have been fetched.

Inevitably, some 32-b instructions will become unaligned when an odd number of 16-b instructions is followed by a 32-b instruction, requiring an additional fetch cycle for which the processor will have to be stalled. As described earlier in Section III, we have added a prefetch buffer, which allows to fetch a complete cache line (128 b) instead of a single instruction to reduce the access contentions associated with the shared I$ [37]. While this allows the core to access four to eight instructions in the prefetch buffer, the problem of fetching unaligned instructions remains, it is just shifted to cache-line boundaries, as shown in Fig. 3, where the current 32-b instruction is split over two cache lines.

To prevent the core from stalling in such cases, an additional register is used to keep the last instruction. In the case of an

TABLE I
INSTRUCTIONS DESCRIPTION

Instruction format	Description	Instruction format	Description
Hardware Loop Instructions		**Fixed Point Instructions**	
`lp.starti L, I`	Set the HW loop start address	`p.add[R]N rD, rA, rB, I`	Addition with round and norm. by I bits [a]
`lp.endi L, I`	Set the HW loop end address	`p.sub[R]N rD, rA, rB, I`	Subtraction with round and norm. by I bits [a]
`lp.count L, rA`	Set the number of iterations	`p.mul[hh][R]N rD, rA, rB, I`	Mult. with round and norm. by I bits [a b]
`lp.setup L, rA, I`	HW loop setup with registers	`p.mac[hh][R]N rD, rA, rB, I`	MAC with round and norm. by I bits [a b]
`lp.setupi L, I1, I2`	HW loop setup with immediate	`p.clip rD, rA, I`	Clip the value between -2^{I-1} and $2^{I-1}-1$
Extended Load/Store Instructions		**Vectorial Instructions**	
`p.l{b,h,w} rD, {rB,I}(rA)`	Load a value from address (rA+{rB,I}) [c]	`pv.inst.{b,h} rD, rA, rB`	Vectorial instruction between two registers [c]
`p.l{b,h,w} rD, {rB,I}(rA!)`	Load a value from address rA and increment rA by{rB,I} [c]	`pv.inst.{b,h} rD, rA, I`	Vectorial instr. between a register and an immediate [c]
`p.s{b,h,w} rB, {rD,I}(rA)`	Store a value to address (rA+{rD,I}) [c]		
`p.s{b,h,w} rB, {rD,I}(rA!)`	Store a value to address rA and increment rA by{rD,I} [c]		

[a] If R is not specified, there is no round operation before shifting. [b] If hh is specified, the operation takes the higher 16b of the operands.
[c] b, h, w specific the data lenght of the operands: byte (8b), halfword (16b), word (32b).

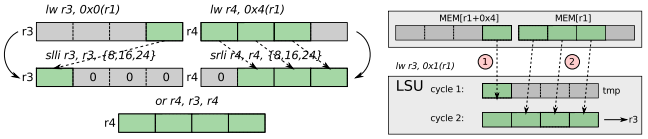

Fig. 4. (a) Support for unaligned access in software (five instructions/cycles) and (b) with hardware support in the LSU (one instruction and two cycles).

unaligned 32-b instruction, this register will contain the lower 16 b of the instruction, which can be combined with the higher part (1) and forwarded to the ID-stage. This addition allows unaligned accesses to the I$ without stalls unless a branch, hardware loop, or jump is processed. In these cases, the FSM has to fetch a new cache line to get the new instruction. The area cost of this prefetch buffer is mainly due to additional registers and accounts for 4 kGE.

C. Hardware Loops

Zero-overhead loops are a common feature in many processors, especially DSPs, where a hardware loop-controller inside the core can be programmed by the loop count, the beginning, and end address of a loop. Whenever the current program counter (PC) matches the end address of a loop, and as long as the loop counter has not reached 0, the hardware loop-controller provides the start address to the fetch engine to reexecute the loop. This eliminates instructions to test the loop counter and perform branches, thereby reducing the number of instructions fetched from I$.

The impact of hardware loops can be amplified by the presence of a loop buffer, i.e., a specialized cache holding the loop instructions, which removes any fetch delay [38]; in addition, fetch power can also be reduced by the presence of a small loop cache [39]. Nested loops can be supported with multiple sets of hardware loops, where the innermost loop always gives the highest speedup as it is the most frequently executed. We have added hardware loop support to the RISC-V cores at the microarchitectural level with only two additional blocks: a controller and a set of registers to store the loop information. Each set has associated three special registers to hold the loop counter, the start address, and the end address. The registers are mapped in the CSR space, which allows to save and restore loop information when processing interrupts or exceptions. A set of dedicated instructions have been provided that will initialize a hardware loop in a single instruction using *lp.setup* (or *lp.setupi*). Additional instructions are provided to set individual registers explicitly (*lp.start*, *lp.end*, *lp.count*, and *lp.counti*).

Since the performance gain is maximized when the loop body is small, supporting many register sets only brings marginal performance improvements at a nonnegligible cost in terms of area ($\approx$ 1.5 kGE per register set). Our experiments have shown that two register sets provide the best tradeoff. As mentioned earlier, our improved cores feature a prefetch buffer able to hold four to eight instructions of the I$. This prefetch buffer can act as a very small loop cache and if the loop body fits into the prefetch buffer, I$ accesses can be eliminated during the loop, reducing the power considerably. The GCC compiler has been modified to automatically insert hardware loops by using the instructions provided in Table I.

D. Load–Store Unit

The basic RISC-V architecture only supports one type of *ld/st* operation where the effective address is computed by adding an offset coming from an immediate to the base address stored in a register. We have first added an additional addressing mode where the offset can be stored in a register instead of an immediate, and then added a postincrement addressing mode with an immediate or register offset to automatically update pointers. A preincrement *ld/st* mode was not deemed necessary as every preincrement *ld/st* operation can be rewritten in a postincrement *ld/st* operation. Support for postincrement instructions leads to high speedup of up to 20% when memory access patterns are regular as it is the case, for example, in a matrix multiplication. To support *ld* operations with postincrement, two registers have to be updated in the RF: the data from memory and the incremented address pointer, which is computed in the ALU. Since ALU and LSU have separate RF write ports, both values can be written back without experiencing any contentions. Table I shows the various forms of additional *ld/st* instructions.

The LSU has also been modified to support unaligned data memory accesses, which frequently happen during vector operations, such as the 5×5 2-D convolution described earlier in Section III. If the LSU detects an unaligned data access, it first issues a request to the high word and stores the data in a temporary register. In a second request, the lower bytes

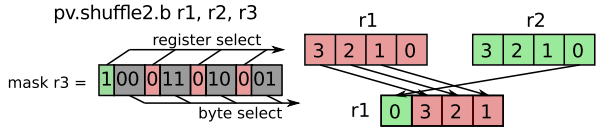

Fig. 5. Shuffle instruction allows to efficiently combine 8- or 16-b elements of two vectors in a single one. For each byte, the mask encodes which byte (index) is used from which register (select).

are accessed, which are then on the fly combined with the temporary register and forwarded to the RF. This approach not only allows to reduce the pressure on the number of used registers, but also reduces code size, as shown in Fig. 4, and the number of required cycles to access unaligned vector elements. In addition, this low-overhead approach is better than full hardware support as this would imply to double the width of the interconnect and change to the memory architecture.

E. EX-Stage: ALU

1) Packed-SIMD Support: To take advantage of applications in the IoT domain that can work with 8- and 16-b sensor data, the ALU can be modified to work on vectors of four and two elements using a vectorized datapath segmented into two or four parts, allowing to compute up to 4 B in parallel. Such operations are also known as subword parallelism [40], packed-SIMD or micro-SIMD [41] instructions.

We have extended the RVC32IM ISA with subword parallelism for 16- (halfword) and 8-b (byte) operations in three addressing variations. The first variation uses two registers, the second uses an immediate value, and the third replicates the scalar value in a register as the second operand for the vectorial operation. Vectorial operations, such as additions or subtractions, have been realized by splitting the operation in four suboperations, which are connected through the carry propagation signals. For example, the full 32-b result is computed by enabling all the carry propagation signals while in vector addition at byte level, the carry propagation is terminated between suboperations. The final vectorial adder is, therefore, a 36-b architecture (32 b for operands and four carry bits). Vectorial comparison and shift operations have similarly been realized by splitting the datapath in four separate segments. A 32-b comparison is then computed using the four 8-b comparison results. Logic vectorial operations, such as *and, or*, and *xor*, are trivial, since the exact similar datapath can be used.

Additional subword data manipulation instructions are needed to prepare vector operands [42] for vector ALUs. We have implemented a three operand *shuffle* instruction that can generate the output as any combination of the subwords of two input operands, while the third operand, sets selection criteria either through an immediate value or a register as seen in Fig. 5. The *shuffle* instruction is supported through a tree of shared multiplexers and can also be used to implement further subword data manipulation instructions such as in *insert*, *extract*, and *pack* instructions

2) Fixed-Point Support: There are many applications, such as speech processing, where floating-point accuracy is not always needed, and simpler fixed-point arithmetic operations can be used instead [43]. This has the advantage of reusing

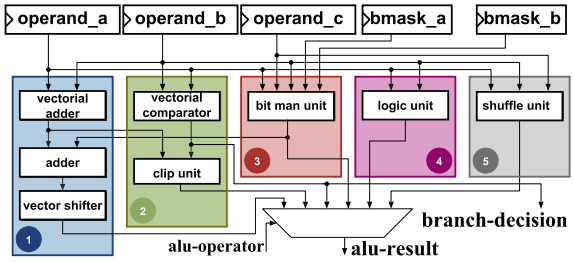

Fig. 6. Simplified block diagram of the RISC-V ALU.

TABLE II
ADDITION OF FOUR Q1.11 FIXED-POINT NUMBERS

Without Add Norm Round	With Add Norm Round

`add r3, r4, r5`
`add r3, r3, r6`
`add r3, r3, r7`
`addi r3, r3, 2`
`srai r3, r3, 2`
	`add r3, r4, r5`
	`add r3, r3, r6`
	`p.addRN r3, r3, r7, 2`

the integer datapath for most operations with the help of some dedicated instructions for fixed-point support, such as saturation, normalization, and rounding.

Fixed-point numbers are often given in the Q-format where a Qn. m number consists of n integer bits and m fractional bits. Some processors support a set of fixed-point numbers encoded in 8, 16, or 32 b, and provide dedicated instructions to handle operations with these numbers. For example, the ARM Cortex-M4 ISA provides instructions as QADD or QSUB to add two numbers and then saturate the results to 8, 16, or 32 b.

Our extensions to the RISC-V architecture have been designed to support fixed-point arithmetic operations in any Q-format with the only limitation that $n + m < 32$. We provide instructions that can add/subtract numbers in fixed-point arithmetic and shift them by a constant amount to perform normalization. The two code examples in Table II show how the combined add-round-normalize (*p.addRN*) instruction can save both code size (three instead of five instructions) and execution time (two cycles less). In this example, four numbers represented by Q1.11 are summed up. The result, if not normalized, will be a 14-b-long Q3.11 number. To keep the result in 12 b, rounding can be achieved by adding two units of least precision (ULP) to the result and shift the number right by two places. The result can then be interpreted as a 12-b number in Q3.9 format. The final *p.addRN* instruction achieves this rounding in a single step by first adding the two operands using the vectorial adder, and adding $2^{(I-1)}$ to the intermediate result before it is shifted by I bits, utilizing the shifter of the ALU. An additional 32-b adder was added to the ALU to help with the rounding operation as seen in highlighted region 1 of Fig. 6.

For fixed-point operations, a *clip* instruction has been implemented to check if a number is between two values and saturates the result to a minimum or maximum bound otherwise. No significant hardware has been added to the ALU to implement the clip instruction; indeed, the *greater than* comparison is done using the existing comparator and the *less than* comparison is done in parallel by the adder. The *clip* instruction relies on the input data; therefore, handling overflows is left to the programmer. Unlike the ARM Cortex-M4 implementation, our implementation requires an additional

TABLE III
ADDITION OF n ELEMENTS WITH SATURATION

Without clip support		With clip support	
addi	r15, r0, 0x800	addi	r3, r0, *n*
addi	r14, r0, 0x7FF	lp.setup	r0, r3, endL
addi	r3, r0, *n*	p.lh	r4, 0(r10!)
lp.setup	r0, r3, endL	p.lh	r5, 0(r11!)
p.lh	r4, 0(r10!)	add	r4, r4, r5
p.lh	r5, 0(r11!)	p.clip	r4, r4, 12
add	r4, r4, r5	endL: sw	0(r12!), r4
blt	r4, r15, lb		
blt	r14, r4, ub		
j	endL		
lb: mv	r4, r15		
j	endL		
ub: mv	r4, r14		
endL: sw	0(r12!), r4		

clip instruction, but has the added benefit of supporting any Q-number format and allows to round and normalize the value before saturating, which provides higher precision.

Table III shows an example of compiler-generated code where two arrays, each containing n Q1.11 signed elements, are added together; then, the result is normalized between -1 and 1 represented in the same Q1.11 format. The example clearly illustrates the difference between the RISC-V ISA with and without clip support. Table I shows the added instructions for fixed-point support. Note that the code to the right is not only shorter, and it also does not have control flow instructions, thereby achieving better IPC.

3) Bit Manipulation Support: There are many instances where a single bit of a word needs to be accessed e.g., to access a configuration bit of a memory-mapped register. We have enhanced the RISC-V ISA with instructions, such as *p.extract* (read a register set of bits), *p.insert* (write to a register set of bits), *p.bclr, p.bset* (clear/set a set of bits), *p.cnt* (count number of bits that are 1), *p.ff1,p.fl1* (find index of first/last bit that is 1 in a register), and *p/clb* (count leading bits in a register).

4) Iterative Divider: To fully support the RVC32IM RISC-V ISA, we have opted to support division by using long integer division algorithm by reusing existing comparators, shifters, and adders of the ALU. Depending on the input operands, the latency of the division operation can vary from 2 to 32 cycles. While it is slower than a dedicated parallel divider, this implementation has low area overhead (2 kGE).

F. EX-Stage: Multiplication

While adding vectorial support for add/sub and logic operations was achieved with relative ease, the design of the multiplier was more involved. The simplified final multiplier architecture shown in Fig. 7 contains four modules: a 32-b×32-b multiplier, a fractional multiplier, and two dot-product (dotp) multipliers.

The proposed multiplier has the capability to multiply two vectors and accumulate the result in a 32-b value in one cycle. A vector can contain two 16-b elements or four 8-b elements. To perform signed and unsigned multiplications, the 8-/16-b inputs are sign extended; therefore, each element is a 17- or 9-b signed word. A common problem with an N bit multiplier is that its output needs to be 2N bits wide to be able to cover the entire range. In some architectures, an additional register is used to store part of the multiplication result. The dot-product operation produces a result with a larger dynamic than its operands without any extra register

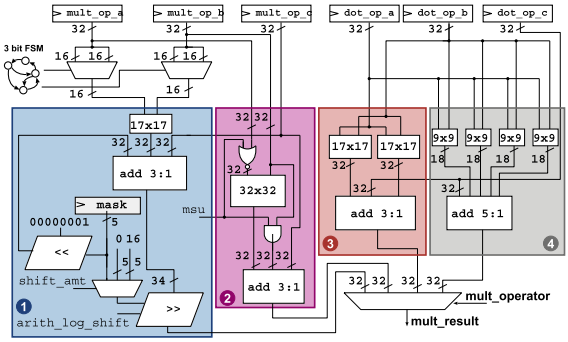

Fig. 7. Simplified block diagram of the multiplier in the RISC-V core implementation reflecting the behavioral implementation.

due to the fact that its operands are either 8 or 16 b. Such dot-product (*dotp*) operations can be implemented in hardware with four multipliers and a compression tree and allow to perform up to four multiplications and three additions in a single operation as follows:

$$d = a[0] \cdot b[0] + a[1] \cdot b[1] + a[2] \cdot b[2] + a[3] \cdot b[3]$$

where $a[i]$ and $b[i]$ are the individual bytes of a register and d is the 32-b accumulation result. The MAC equivalent is the sum-of-dot-product (*sdotp*) operation, which can be implemented with an additional accumulation input at the compression tree. With a vectorized ALU and *dotp*-operations, it is possible to significantly increase the computational throughput of a single core when operating on short binary numbers.

The implementation of the dot-product unit has been designed, such that its longest path is shorter or equal to the critical path of the overall system. In our case, this path is from the processor core to the memories and vice versa. Additional pipeline registers would have resulted in additional stalls when computing back-to-back *dotp* operations. Since *dotp* operations are near to be timing critical, multiplication resources are not shared between the four regions but only within regions, allowing the additional circuitry to have no impact on the overall operation speed. Fig. 7 shows the 16-b dotp unit (region *3*) and 8-b dotp unit (region *4*), which have been implemented by one partial product compressor, which sums up the accumulation register and all partial products coming from the partial product generators. The multiplications exploit carry-save format without performing a carry propagation before the additions. The stand-alone multiplier has been analyzed in detail to minimize the area-delay product. Area can be saved when sharing compression trees of the two dot multipliers. However, sharing resources also makes the unit slower, which is why the dot-product-units are described behaviorally following Synopsys Design rules to give the synthesizer maximum freedom for optimizations.

The proposed multiplier also supports fixed-point numbers. In this mode, the *p.mul* instruction accepts two 16-b values (signed or unsigned) as input operands and calculates a 32-b result. The *p.mac* multiply–add instruction allows an additional 32-b value to be accumulated to the result. Both instructions produce a 32-b value, which can be shifted to the right by I bits; moreover, it is possible to round the result (adding 2^{I-1}) before shifting, as shown in Fig. 7, where the fractional multiplier is shown in region *1*.

TABLE IV
ELEMENTWISE MULTIPLICATION OF n Q1.11 ELEMENTS

Without Mul Norm Round		With Mul Norm Round	
addi	r3, r0, *n*	addi	r3, r0, *n*
lp.setup	r0, r3, endL	lp.setup	r0, r3, endL
p.lh	r4, 0(r10!)	p.lh	r4, 0(r10!)
p.lh	r5, 0(r11!)	p.lh	r5, 0(r11!)
mul	r4, r4, r5	p.mulsRN	r4, r4, r5, 12
addi	r4, r4, 0x800	endL: sw	0(r12!), r4
srai	r4, r4, 12		
endL: sw	0(r12!), r4		

Consider the code example given in Table IV that demonstrates the difference between the RISC-V ISA with and without fixed-point multiplication support. In this example, two vectors of n Q1.11 elements are multiplied with each other (a common operation in the frequency domain to perform convolution). The multiplication results in a Q2.22 number and a subsequent rounding and normalization step will be needed to express the result with 12 b as a Q1.11 number. It is important to note that, for such operations, performing the rounding operation before normalization reduces the error.

The *p.mulsRN* multiply-signed with round and normalize operation is able to perform all three operations (multiply, add, and shift) in one cycle, thus reducing both the code size and the number of cycles. The two additional 32-b values need to be added to the partial-products compressor, which does not increase the number of the levels of the compressor tree and, therefore, does not add delay to the circuit [44].

Naturally, the multiplier also supports standard 32-b×32-b integer multiplications, and it is also possible to support 16-b×16-b + 32-b MAC operations at no additional cost. Similar to the ARM Cortex-M4 *MLS* instruction, a multiply with subtract using 32-b operands is also supported.

The proposed ISA extensions are realized with separate execution units in the EX-stage, which have contributed to an increase in area (8.3-kGE ALU and 12.6-kGE multiplier). To keep the power consumption at a minimum, switching activity at unused parts of the ALU has to be kept at a minimum. Therefore, all separate units: the ALU, the integer and fractional multiplier, and the dot-product unit all have separate input operand registers, which can be clock gated. The input operands are controlled by the instruction decoder and can be held at constant values to further eliminate propagation of switching activity in idle units. The eliminated switching activity decreases the power consumption of the core by 50%.

V. TOOLCHAIN SUPPORT

The PULP compiler used in this paper has been derived from the original GCC RISC-V version, which is itself derived from the MIPS GCC version. GCC-5.2 release and the latest binutils release have been used. The binutils have been modified to support the extended ISA as well as a few new relocation schemes have been added.

Hardware loop detection and mapping has been enabled as well as postmodified pointers detection. The GCC internal support for hardware loops is sufficient but the module taking care of postmodified pointers is rather old and primitive. As a comparison, more recent modules geared toward vectorization and dependence analysis are way more sophisticated. One of the consequences is that the scope of induced pointers is limited to a single loop level and we miss opportunities that can be exposed across loop levels in a loop nest.

Furthermore, sum of products and sum of differences are automatically exposed, allowing the compiler to always favor the shortest form of mac/msu (use 16 b × 16 b into 32 b instead of 32 b×32 b into 32 b) to reduce energy.

Vector support (4 B or two shorts) has been enabled to take advantage of the SIMD extensions. The fact that we support unaligned memory accesses plays a key role into the exposition of vectorization candidates. Even though autovectorization works well, we believe that for deeply embedded targets, such as PULP, it makes sense to use the GCC extensions to manipulate vectors as C/C++ objects. It avoids overly conservative prolog and epilog insertion created by the auto vectorizer that are having a negative impact on code size. We have opted for a less specialized scheme for the fixed-point representation, to give more flexibility. The architecture supports any fixed-point format between Q2 and Q31 with optimized instructions for normalization, rounding, and clipping.

All these instructions fit nicely into the instruction combiner pass of GCC. Normalization and rounding can be combined with common arithmetic instructions, such as addition, subtraction, and multiplication/mac, everything being performed in one cycle. Dot-product instructions are more problematic, since they are not a native GCC internal operation and the depth of its representation prevent it from being automatically detected and mapped by the compiler, which is the reason why we rely on built-in support. More generally, most of our extensions can also be used through built-ins as an alternative to automatic detection (in contrast with assembly insertions built-ins can easily capture the precise semantic of the instructions they implement). An example of using *dotp* instructions is given in the following.

```
// define vector data type and dotp instruction
typedef short PixV __attribute__((vector_size(4)));
#define SumDotp16(a,b,c) __builtin_sdotsp2(a,b,c);

PixV VectA, VectB; // vectors of shorts
int S;
...
S = 0;
// each iteration is computing two mult and 2 accum
for (int k = 0; k < (SIZE>>1); k++) {
  S = SumDotp16(VectA[k], VectB[k], S);
}
C[i*N+j] = S;
...
```

Finally, the bit manipulation part of the presented ISA extensions fits well into GCC, since most instructions have already an internal GCC counterpart.[2]

VI. EXPERIMENTAL RESULTS

For hardware, power, and energy efficiency evaluations, we have implemented the original and extended core in a PULP cluster with 72-kB TCDM memory and 4-kB I$. The two clusters (cluster A with an RVC32IM RISC-V core, and cluster B with the same core plus the proposed extensions) have been synthesized with Synopsys Design Compiler-2016.03 and

[2]PULP Parallel Programming is possible thanks to OpenMP three support integrated in our GCC compiler. The interested reader is referred to [45] for more information.

TABLE V
COMPARISON OF DIFFERENT CORE ARCHITECTURES

Processor Core	This Work RISC-V Basic	This Work RISC-V Ext.	OpenRISC [46]	CortexM4 [22]	
Technology	65 nm	65 nm	65 nm	90 nm	40 nm
Vdd [V]	1.08	1.08	1.2	1.2	1.1
Freq. [MHz]	357	357	362	216	216
Dyn. Power [µW/MHz]	26.28	28.68	33.8	32.8	12.26
Area [kGE]	46.9	53.5	44.5	-	-
[mm^2]	0.068	0.077	0.064	0.119	0.028
CoreMark/MHz	2.94[a]	3.19[a]	2.66	3.40[b] / 2.55[c]	

[a] GCC-5.2.0, Flags: "-O2 -g -falign-functions=16 -funroll-all-loops"
[b] IAR v6.50, Flags: "-e -Ohs –use_c++_inline –no_size_constraints"
[c] GCC-5.2.1, Flags: "-O3 –g -falign-functions=16 -falign-loops=16 -mno-unaligned-access -finline-functions -fno-strict-aliasing"

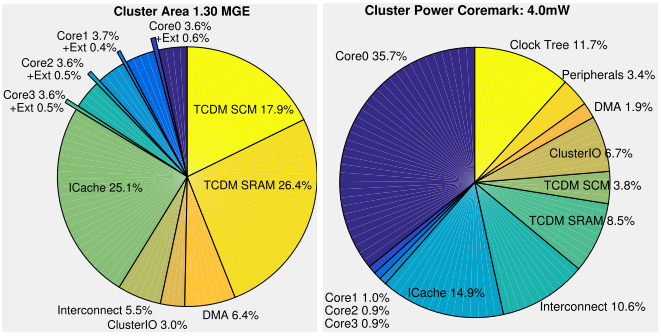

Fig. 8. (a) Area distribution of the cluster optimized for 2.8-ns cycle time. The DSP extensions of each core are highlighted. (b) Power distribution when running CoreMark on a single core at 50 MHz and 1.08 V.

complete back-end flows have been performed using Cadence Innovus-15.20.100 in a eight-metal UMC 65-nm LL CMOS technology. A set of benchmarks has been written in C (with no assembly level optimization) and compiled with the modified RISC-V GCC toolchain, which makes use of the ISA extensions. The switching activity of both clusters has been recorded from simulations in Mentor QuestaSim 10.5a using backannotated postlayout gate-level netlists and analyzed with the obtained value change dump files in Cadence Innovus-15.20.100.

In Section VI-A, the area, frequency, and power of the cluster will be discussed. Then, the energy efficiency of several instructions is discussed in Section VI-B. Speedup and energy efficiency gains of cluster B are presented in Section VI-C, followed by an in-depth discussion about convolutions in Section VI-D. While Sections VI-A–VI-D focus on relative energy and power comparisons, which are best done in a consolidated technology in super threshold conditions, the overall cluster performance is analyzed in NT conditions and compared with previous PULP architectures in Section VI-E.

A. Area, Frequency, and Power

Table V shows a comparison of the basic and extended RISC-V cores with an OpenRISC architecture, and the ARM Cortex M4. The RISC-V architecture is similar in performance to an OpenRISC architecture, but with a smaller dynamic power consumption due to support of compressed instructions (less instruction fetches) and low-level power optimizations to reduce switching activity. The extended RISC-V core

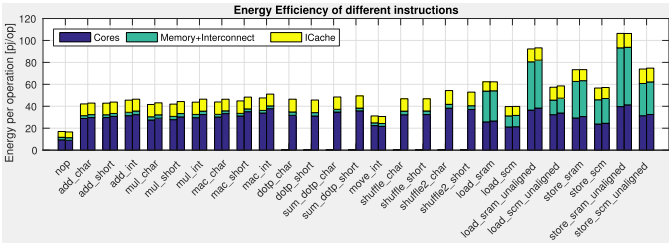

Fig. 9. Energy per operation for different instructions with random inputs, 1.08 V, worst case conditions, 65-nm CMOS.

architecture increases in size by 6.6 kGE as a result of additional execution units (dot-product unit, shuffle unit, and fixed-point support) but these extensions allow the core to reach a higher CoreMark score of 3.19.

With respect to ARM architectures, the proposed core is comparable in size and reaches better CoreMark scores than ARM when relying on GCC compilers. ARM's top score cannot yet be achieved due to 11% of cycle penalty for taken branches, 3% for jumps, and 7% for load-use penalties. These penalties could be removed to some degree if the compiler would reorder instructions (load use), and with a branch prediction circuit. The proposed extensions are also not utilized best in CoreMark as it is a rather control-intensive benchmark with up to 20% branch instructions. The real performance of the proposed core becomes apparent in a shared-memory multicore system, where even in a four-core configuration, it reaches higher operating frequencies on comparable technologies.

The area distribution of the cluster is shown in Fig. 8(a). The total area of the cluster is 1.30 MGE of which the additional area for the dot-product unit and ALU extensions accounts for only 2%. Fig. 8(b) shows the power distribution when running CoreMark on a single core. The total power consumption at 50 MHz and 1.08 V is only 4 mW of which 35% is consumed by the active core. The three idle cores consume only 2.8% of which 83% is leakage.

B. Instruction Level Performance

The power consumption and energy efficiency of the two core versions have been analyzed at instruction level. To determine the power and energy of each instruction, they have been executed in a loop with 100 iterations each containing 100 instructions. The energy of a single instruction is then the product of the execution time and the power divided by the number of executed instructions (10 000) and cores (4). The power of each instruction consists of TCDM power (including interconnect), core power, and I$ power. The execution time of the loop depends on the latency of the instruction (two cycles for unaligned memory access instructions and one cycle for others).

Fig. 9 shows the resulting energy per operation for different types of instructions. As expected, *nop* consumes the least power, which is not 0, because it has to be fetched from the I$/L0-buffer. Then comes a set of arithmetic instructions (*add, mul, mac, dotp, sdotp*) for different data types (char, short, and int). The extended core is bigger, and hence, it also consumes more power. For the arithmetic operations, we observe a slight power increase of 4%.

While some of the new instructions (*mac*, *sdotp*, and *shuffle*) consume slightly more power than other arithmetic instructions, they actually replace multiple instructions, which makes them more energy efficient. For example, the *shuffle* instruction is capable of reorganizing bytes or shorts in any combination by utilizing the shuffle unit. Executing a shuffle operation (50 pJ) is equivalent to executing three to four simple ALU operations for 90–120 pJ. Similarly, the proposed LSU allows to perform unaligned memory accesses from SRAMs for only 93–106 pJ whereas software only support would require to execute five instructions in sequence, which would cost about 3× as much energy. We also observe that *ld/st* operations from SCMs are on average 46% more energy efficient than from SRAMs. Unfortunately, SCMs are not very area efficient and, therefore, limited in size.

C. Function Kernel Performance

To evaluate the performance gain of the proposed extensions, a set of benchmarks ranging from cryptographic kernels (crc, sha, aes, and keccak), control-intensive applications (fibonacci and bubblesort), transformations (FFT and FDCT), and over more data-intensive linear algebra kernels (matrix additions and multiplications), to various filters (fir, 2-D filters, and convolutions) has been compiled. We have also evaluated a *motion detection* application, which makes use of linear algebra, convolutions, and filters.

Fig. 10(a) shows the IPC of all applications and Fig. 10(b) shows the speedup with hardware loops and postincrement extensions, versus a plain RISC-V ISA. In this case, an average speedup of 37% can be observed. As expected, filters and linear algebra kernels with very regular data access patterns benefit the most from the extensions. A second bar (+built-ins) shows the full power of the proposed RISC-V ISA extensions. Data-intensive kernels benefit from vector extensions, which can be used with vector data types and the C built-in *dotp* instruction. On these kernels, a much larger speedup, up to 13.2×, can be achieved, with an average gain of 3.5×.

The overall energy gains of the extended core are shown in Fig. 10(c), which shows an average of 3.2× gains. In an ideal case, when processing a matrix multiplication that benefits from *dotp*, hardware loop and postincrement extensions the gain performance gain can reach 10.2×.

Finally, the proportion of compressed instructions is shown in Fig. 10(d). We see that 28.9%–46.1% of all executed instructions were compressed. Note that, the ratio of compressed instructions is less in the extended core, as vector instructions, and built-ins do not exist in compressed form.

D. Convolution Performance

Convolutions are common operations and are widely used in image processing. In this section, we compare the performance of *basic* RISC-V implementation to the architecture with the *extensions* proposed in this paper. Since not all instructions can be efficiently utilized by the compiler, we also provide a third comparison called *built-ins*, which runs on the extended architecture by directly calling these extended instructions. For the convolutions, a 64×64 pixel image has been used with a Gaussian filter of size 3×3, 5×5, and 7×7. Convolutions are shown for 8-b (char) and 16-b (short) coefficients.

Fig. 11(a) shows the required cycles per output pixel using the three different versions of the architecture. In this evaluation, the image to be processed is divided into four strips and each core working on its own strip. Enabling the extensions of the core allows to speedup the convolutions by up to 41% mainly due to the use of hardware loops, and postincrement instructions. Another 1.7–6.4× gain can be obtained when using the *dotp* and *shuffle* instructions resulting in an overall speedup of 2.2–6.9× on average. *Dotp* instructions are processing four multiplications, three additions, and an accumulation all within a single cycle and reduce the number of arithmetic instructions significantly. A 5×5 filter would require 25 *mac* instructions, while 7 *sdotp* instructions are sufficient when the vector extensions are used. As seen in Fig. 11(b), the acceleration also directly translates into energy savings, and the extended architecture computing convolutions on a 64×64 image are 2.2–7.8× more energy efficient.

Utilizing vector extensions and the RF as a L0-storage allows to reduce the number of load words, as shown in Fig. 11(c), because it is possible to keep more coefficients in the RF. When computing 3×3 and 5×5 convolutions, it is even possible to store all required elements in the RF, and only load the new pixels. The remaining pixels can be reorganized using *move* and *shuffle* instructions as described in the 2-D-convolution example of Section III. This not only reduces the number of *ld/st* operations by 8.3×, but also reduces contentions at the eight-bank shared memory. The extended cores have a higher *ld/st* density, and thus experience more contentions per TCDM access, 17.8% on average. With vector operations, this number goes down to only 6.2%, which is a reduction from 11 100 contentions to only 390.

Fig. 11(d) shows the power distribution of the cluster. Note that, although for all examples the power of the core increases, the overall system power is reduced in all but one case, where it marginally increases by 5.3% (*Conv_3x3_short*).

The speedup and energy saving of the vector convolutions on the four-core system are shown in Fig. 11(e) and (f) with respect to a single-core configuration. Overheads at the strip boundaries of the multicore implementation are negligible as the speedup with 3.9× is almost ideal. Using four instead of one core requires only 2.4× more power leading to energy savings of 1.6×. Hence, the system is very well scalable both in execution speed and energy.

E. Near-Threshold Operation

The ISA extensions of the RISC-V cores decrease the execution time significantly, which, at the same time, reduces the energy. Fig. 12 shows the energy efficiency of several successfully taped out PULP chips when processing a matrix multiplication on four cores. First, PULPv2 [34]—an implementation of PULP with basic OpenRISC cores without ISA extensions in 28-nm FDSOI, and second Mia [35]—a PULP cluster in 65 nm with an OpenRISC core, featuring a first set of ISA extensions (hardware loops and postincrement instructions) [46]. While Mia had no support for dot products

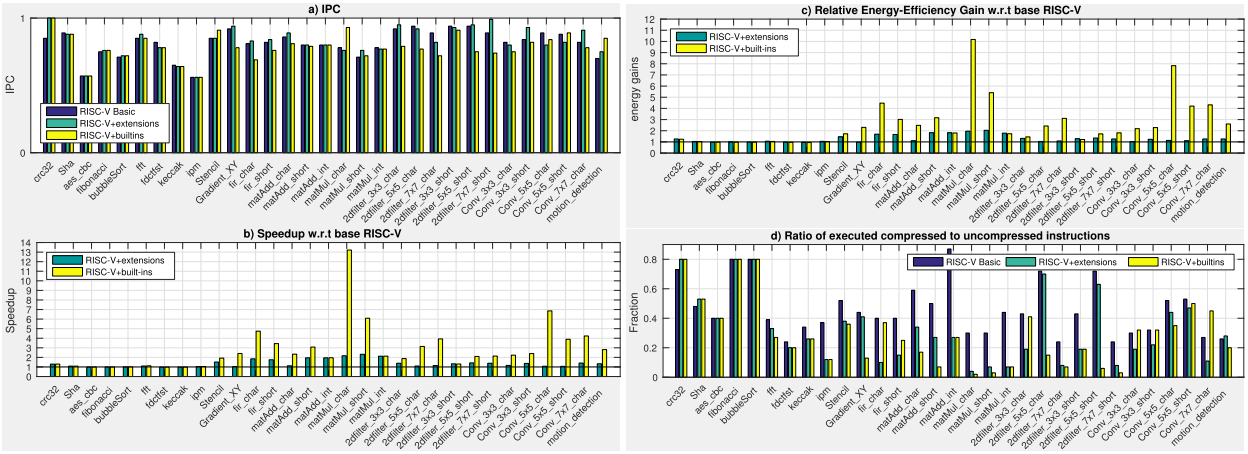

Fig. 10. (a) IPC of all benchmarks with and without extensions, and with built-ins. (b) Speedup. (c) Energy-efficiency gains with respect to a RISC-V cluster with basic extensions. (d) Ratio of executed instructions (compressed/normal).

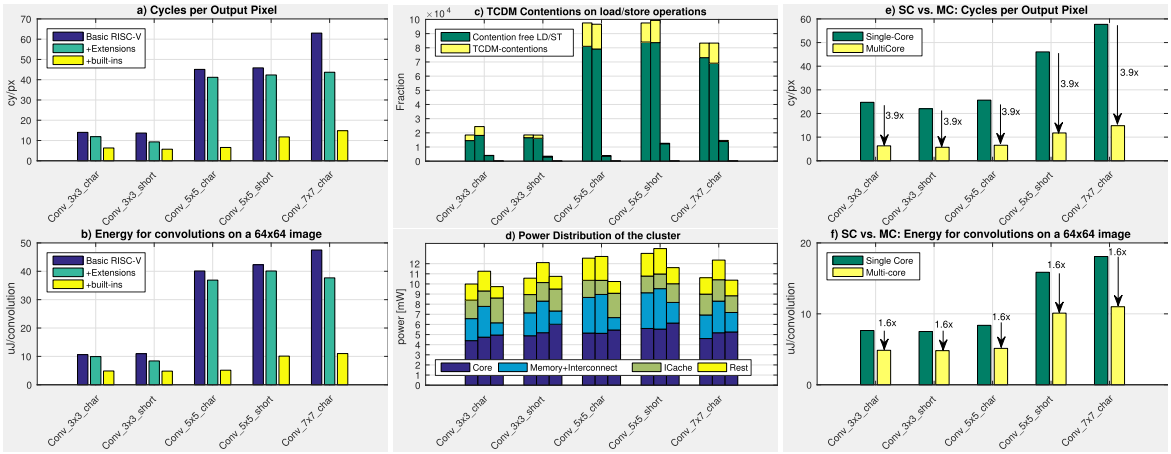

Fig. 11. (a) Required cycles per output pixel. (b) Total energy consumption when processing the convolutions at 50 MHz at 1.08 V. (c) Number of TCDM contentions and (d) power distribution of the cluster when using the three different instruction sets. (e) and (f) Compare single-core and multicore implementations in speedup and energy.

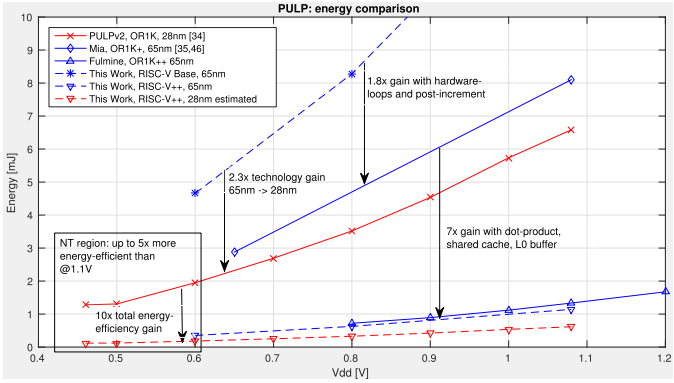

Fig. 12. Energy consumption of different PULP architectures when processing a matrix multiplication on four cores that shows the potential of the vector extensions.

and shuffle, Fulmine is already supporting a very similar ISA as the presented RISC-V core in this paper. All chips are silicon proven and measured.

PULPv2, Mia, and this paper feature SCMs, which allow for NT operation. PULPv2 works down to 0.46 V where it consumes only 1.3 mJ for a matrix multiplication, which is 5× less than at 1.1V. This also means that the average energy per operation of 52 pJ/op at 1.08 V translates to approximately 10 pJ/op in NT operation.

Fig. 12 shows the evolution of the architecture. First, we observe a 2.3× gain of the 28-nm PULPv2 versus a 65-nm RISC-V-based cluster with comparable instruction sets (both architectures feature no ISA extensions). Then, we see that the introduction of hardware loops and postincrement instructions in Mia almost closes this gap as the execution speed improved by 1.8×. The ISA extensions presented in this paper bring the largest speedup due to dot-product instructions and accounts for a factor of 10× with respect to a RISC-V or OpenRISC core without extensions.

The graph also shows the expected energy efficiency of these extensions in an advanced technology, such as 28-nm FDSOI. These estimations show that moving to an advanced technology and operating at the NT region will provide another 1.9× advantage in energy consumption with respect to the 65-nm implementation reported in this paper. In fact, a 28-nm implementation achieving 193 MOPS/mW in NT operation equals an energy consumption of 5.2 pJ/op, which is better than best MCUs, which consume 10 pJ/op [14], [17] but typically have less powerful instructions than our core. The proposed RISC-V cluster will work in the same conditions (0.46–1.1 V), and achieve the same throughput

(0.15–2.35 GOPS) while consuming 10× less energy due to the reduced runtime and only moderate increase in power. The increased efficiency, as well as low-power consumption (1 mW at 0.46 V), and still high computation power (0.15 GOPS at 0.46 V and 40 MHz) allow the cluster to be perfectly suited for data processing in IoT endpoint devices.

VII. Conclusion

We have presented an extended RISC-V -based processor core architecture, which is compatible with state-of-the art cores. The processor core has been designed for a multicore PULP system with a shared L1 memory and a shared I$. To increase the computational density of the processor, ISA extensions, such as hardware loops, postincrementing addressing modes, and fixed-point and vector instructions, have been added. Powerful dot-product and sum-of-dot-products instructions on 8- and 16-b data types allow to perform up to four multiplications and accumulations in a single cycle, while consuming the same power as a single 32-b MAC operation. A smart L0 buffer in the IF-stage of the cores is capable of fetching compressed instructions and buffering one cache line, reducing the pressure on the shared I$.

On a set of benchmarks, we observe that the core with the extended ISA is on average 37% faster on general purpose applications, and by utilizing the vector extensions, another gain of up to 2.3× can be achieved. When processing convolutions on the proposed core, the full benefit of vector and fixed-point extensions can be used, leading to an average speedup of 3.9×. The use of vector instructions in combination with an L0-storage allows to decrease the shared-memory bandwidth by 8.3×. Since *ld/st* instructions require the most energy, this decrease in bandwidth leads to energy gains up to 7.8×.

In addition, multicore implementations feature significantly fewer shared-memory contentions with the new ISA extensions, allowing a four-core implementation to outperform a single-core implementation by 3.9× while consuming only 2.4× more power.

Finally, implemented in an advanced 28nm technology, we observe a 5× energy efficiency gain when processing NT at 0.46 V where the cluster is achieving 0.15 GOPS while consuming only 1 mW. The cluster is scalable as it is operational from 0.46 to 1.1 V where it consumes 1–68 mW and achieves 0.15–2.35 GOPS, making it attractive for a wide range of IoT applications.

Acknowledgment

The authors would like to thank G. Haugou for the tool and software support.

References

[1] G. Lammel, "The future of MEMS sensors in our connected world," in *Proc. Int. Conf. Micro Electro Mech. Syst.*, 2015, pp. 61–64.

[2] V. Shnayder, M. Hempstead, B.-R. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proc. 2nd Int. Conf. Embedded Netw. Sens. Syst. (SenSys)*, Nov. 2004, pp. 188–200.

[3] E. F. Nakamura, A. A. F. Loureiro, and A. C. Frery, "Information fusion for wireless sensor networks: Methods, models, and classifications," *ACM Comput. Surv.*, vol. 39, no. 3, 2007, Art. no. 9.

[4] M. Konijnenburg *et al.*, "A battery-powered efficient multi-sensor acquisition system with simultaneous ECG, BIO-Z, GSR, and PPG," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 480–482.

[5] F. Zhang *et al.*, "A batteryless 19μW MICS/ISM-band energy harvesting body area sensor node SoC," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 55. Feb. 2012, pp. 298–299.

[6] S. R. Sridhara *et al.*, "Microwatt embedded processor platform for medical system-on-chip applications," *IEEE J. Solid-State Circuits*, vol. 46, no. 4, pp. 721–730, Apr. 2011.

[7] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proc. IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[8] A. Y. Dogan, J. Constantin, D. Atienza, A. Burg, and L. Benini, "Low-power processor architecture exploration for online biomedical signal analysis," *IET Circuits, Devices Syst.*, vol. 6, no. 5, pp. 279–286, 2012.

[9] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Symp. Hardw./Softw. Codesign*, May 2002, pp. 73–78.

[10] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, "Power, area, and performance optimization of standard cell memory arrays through controlled placement," *ACM Trans. Design Autom. Electron. Syst.*, vol. 21, no. 4, Sep. 2016, Art. no. 59.

[11] A. Waterman *et al.*, "The RISC-V instruction set manual, volume I: Base user-level ISA," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2011-62, 2011.

[12] O. Azizi *et al.*, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proc. ISCA*, 2010, pp. 26–36.

[13] STMicroelectronics. *STM32F Datasheet*, accessed on Oct. 10, 2016. [Online]. Available: http://www.st.com/resource/en/datasheet/stm32f405og.pdf

[14] Ambiqmicro. *Apollo Datasheet*, accessed on Oct. 16, 2016. [Online]. Available: https://www.eembc.org/ulpbench/Apollo_MCU_Data_SheetDS0010V0p90.pdf

[15] NXP. *LPC5410x Product Data Sheet*, accessed on Sep. 25, 2016. [Online]. Available: http://www.nxp.com/documents/data_sheet/LPC5410X.pdf

[16] S. Rele, S. Pande, S. Onder, and R. Gupta, "Optimizing static power dissipation by functional units in superscalar processors," in *Proc. Int. Conf. Compiler Construction*, 2002, pp. 261–275.

[17] N. Ickes, Y. Sinangil, F. Pappalardo, E. Guidetti, and A. P. Chandrakasan, "A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip," in *Proc. Eur. Solid-State Circuits Conf.*, Sep. 2011, pp. 159–162.

[18] B. Zhai *et al.*, "Energy-efficient subthreshold processor design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 8, pp. 1127–1137, Aug. 2009.

[19] G. Gammie *et al.*, "A 28 nm 0.6 V low power DSP for mobile applications," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2011, pp. 132–133.

[20] R. Wilson *et al.*, "A 460MHz at 397mV, 2.6GHz at 1.3V, 32b VLIW DSP, embedding F_{MAX} tracking," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 57. Feb. 2014, pp. 452–453.

[21] D.-H. Le, N. Sugii, S. Kamohara, X.-T. Nguyen, K. Ishibashi, and C.-K. Pham, "Design of a low-power fixed-point 16-bit digital signal processor using 65nm SOTB process," in *Proc. Int. Conf. IC Design Technol. (ICICDT)*, Jun. 2015, pp. 1–4.

[22] "ARM cortex M-4 technical reference manual," ARM, Cambridge, U.K., Tech. Rep., 2015.

[23] A. A. Faroqui and V. G. Oklobdzija, "Impact of architecture extensions for media signal processing on data-path organization," in *Proc. Conf. Rec. 34th Asilomar Conf. Signals, Syst. Comput.*, Oct. 2000, pp. 1679–1683.

[24] X. Zhang, Z. Li, and Q. Zheng, "Design of a configurable fixed-point multiplier for digital signal processor," in *Proc. Asia Pacific Conf. Postgraduate Res. Microelectron. Electron., PrimeAsia*, Jan. 2009, pp. 217–220.

[25] N. Ickes, D. Finchelstein, and A. P. Chandrakasan, "A 10-pJ/instruction, 4-MIPS micropower DSP for sensor applications," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2008, pp. 289–292.

[26] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 24–35, 2013.

[27] L. Cavigelli and L. Benini, "A 803 GOp/s/W convolutional network accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, vol. PP, no. 99, p. 1, doi: 10.1109/TCSVT.2016.2592330.
[28] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *Proc. DATE*, Mar. 2015, pp. 683–688.
[29] Texas Instruments. *CC2650 SimpleLink Multistandard Wireless MCU*, accessed on Sep. 12, 2016. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2650.pdf
[30] N. Neves, N. Sebastião, D. Matos, P. Tomás, P. Flores, and N. Roma, "Multicore SIMD ASIP for next-generation sequencing and alignment biochip platforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 7, pp. 1287–1300, Jul. 2015.
[31] S. Hsu *et al.*, "A 280mV-to-1.1V 256b reconfigurable SIMD vector permutation engine with 2-dimensional shuffle in 22nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2012, pp. 178–180.
[32] D. Fick *et al.*, "Centip3De: A cluster-based NTC architecture with 64 ARM cortex-M3 cores in 3D stacked 130 nm CMOS," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 104–117, Jan. 2013.
[33] D. Rossi *et al.*, "A 60 GOPS/W, −1.8 V to 0.9 V body bias ULP cluster in 28 nm UTBB FD-SOI technology," *Solid State Electron.*, vol. 117, pp. 170–184, Mar. 2016.
[34] D. Rossi *et al.*, "193 MOPS/mW 162 MOPS, 0.32V to 1.15V voltage range multi-core accelerator for energy efficient parallel and sequential digital processing," in *Proc. IEEE Symp. Low-Power High-Speed Chips (COOL CHIPS)*, Apr. 2016, pp. 1–3.
[35] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A heterogeneous multi-core system-on-chip for energy efficient brain inspired vision," in *Proc. ISCAS*, May 2016, p. 2910.
[36] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis and Machine Vision*. New York, NY, USA: Springer, 1993, doi: 10.1007/978-1-4899-3216-7.
[37] I. Loi, D. Rossi, G. Haugou, M. Gautschi, and L. Benini, "Exploring multi-banked shared-L1 program cache on ultra-low power, tightly coupled processor clusters," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, May 2015, Art. no. 64.
[38] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao, "Techniques for effectively exploiting a zero overhead loop buffer," in *Proc. Int. Conf. Compiler Construction*, 2000, pp. 157–172.
[39] R. S. Bajwa *et al.*, "Instruction buffering to reduce power in processors for signal processing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 5, no. 4, pp. 417–424, Dec. 1997.
[40] R. B. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, Aug. 1996.
[41] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "A comparison between processor architectures for multimedia application," in *Proc. 15th Annu. Workshop Circuits, Syst. Signal Process., ProRisc*, 2004, pp. 138–152.
[42] H. Chang, J. Cho, and W. Sung, "Performance evaluation of an SIMD architecture with a multi-bank vector memory unit," in *Proc. IEEE Workshop Signal Process. Syst. Design Implement. (SIPS)*, Oct. 2006, pp. 71–76.
[43] C.-H. Chang, S.-H. Chen, B.-W. Chen, W. Ji, K. Bharanitharan, and J.-F. Wang, "Fixed-point computing element design for transcendental functions and primary operations in speech processing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 5, pp. 1993–1997, May 2016.
[44] B. Parhami, "Variations on multioperand addition for faster logarithmic-time tree multipliers," in *Proc. Conf. Rec. 30th Asilomar Conf. Signals, Syst. Comput.*, Nov. 1996, pp. 899–903.
[45] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu, "Energy efficient parallel computing on the PULP platform with support for OpenMP," in *Proc. IEEE 28th Conv. Elect. Electron. Eng. Israel (IEEEI)*, Dec. 2014, pp. 1–5.
[46] M. Gautschi *et al.*, "Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2015, pp. 25–30.

Michael Gautschi received the M.Sc. degree in electrical engineering and information technology from ETH Zürich, Zürich, Switzerland, in 2012, where he is currently pursuing the Ph.D. degree with the Integrated Systems Laboratory.

His current research interests include energy-efficient systems, multi-core SoC design, mobile communication, and low-power integrated circuits.

Pasquale Davide Schiavone received the B.Sc. and M.Sc. degrees in computer engineering from Polytechnic of Turin, Turin, Italy, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree with the Integrated Systems Laboratory, ETH Zürich.

His current research interests include data path blocks design, low-power microprocessors in multicore systems, and deep-learning architectures for energy-efficient systems.

Andreas Traber received the M.Sc. degree in electrical engineering and information technology from ETH Zürich, Zürich, Switzerland, in 2015.

He was a Research Assistant with the Integrated System Laboratory, ETH Zürich. He is currently with the Advanced Circuit Pursuit AG, Zürich, Switzerland. His current research interests include energy-efficient systems, multicore architecture, and embedded CPU design.

Igor Loi received the B.Sc. degree in electrical engineering from the University of Cagliari, Cagliari, Italy, in 2005, and the Ph.D. degree from the Department of Electronics and Computer Science, University of Bologna, Bologna, Italy, in 2010.

He currently holds a researcher position in electronics engineering with the University of Bologna. His current research interests include ultralow-power multicore systems, memory system hierarchies, and ultralow-power on-chip interconnects.

Antonio Pullini received the M.S. degree in electrical engineering from University of Bologna, Bologna, Italy.

He was a Senior Engineer with iNoCs Sàrl, Lausanne, Switzerland. His current research interests include low-power digital design and networks on chip.

Davide Rossi received the Ph.D. degree from the University of Bologna, Bologna, Italy, in 2012.

He has been a Post-Doctoral Researcher with the Department of Electrical, Electronic and Information Engineering Guglielmo Marconi, University of Bologna, since 2015, where he is currently an Assistant Professor. His current research interests include energy efficient digital architectures in the domain of heterogeneous and reconfigurable multicore systems. He has authored over 30 papers in international peer-reviewed conferences and journals in these areas.

Eric Flamand received the Ph.D. degree in computer science from Institute national polytechnique de Grenoble, France, in 1982.

He was a Researcher with Centre national d'études des télécommunications and Centre national de la recherche scientifique in France. He then held different technical management positions with the semiconductor industry, Motorola, and ST microelectronics. He is currently a co-founder and the chief technical officer of Greenwaves Technologies, Villard-Bonnot, France, a startup, where he is involved in developing an IoT processor derived from Parallel Ultra-Low Power Platform. He is acting as a part time consultant with ETH Zürich.

Frank K. Gürkaynak received the B.Sc. and M.Sc. degrees in electrical engineering from Istanbul Technical University, and the Ph.D. degree in electrical engineering from ETH Zürich, Zürich, Switzerland, in 2006.

He is currently a Senior Researcher with the Integrated Systems Laboratory, ETH Zürich. His current research interests include digital low-power design and cryptographic hardware.

Luca Benini (F'07) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 1997.

He has served as the Chief Architect of the Platform 2012/STHORMProject with STMicroelectronics, Grenoble, France, from 2009 to 2013. He held visiting/consulting positions with the École Polytechnique Fédérale de Lausanne, Stanford University, and imec. He is currently a Full Professor with the University of Bologna, Bologna, Italy. He has authored over 700 papers in peer-reviewed international journals and conferences, four books, and several book chapters. His current research interests include energy-efficient system design and multicore system-on-chip design.

Dr. Benini is a member of Academia Europaea and a fellow of the ACM. He is currently the Chair of Digital Circuits and Systems with ETH Zürich.