Journal of Signal Processing Systems manuscript No.
(will be inserted by the editor)

Instruction Fetch Energy Reduction with Biased SRAMs

Joonas Multanen · Timo Viitanen · Pekka Jääskeläinen · Jarmo Takala

Received: date / Accepted: date

Abstract Especially in programmable processors, energy consumption of integrated memories can become a limiting design factor due to thermal dissipation power constraints and limited battery capacity. Consequently, contemporary improvement efforts on memory technologies are focusing more on the energy-efficiency aspects, which has resulted in biased CMOS SRAM cells that increase energy efficiency by favoring one logical value over another.

In this paper, xor-masking, a method for exploiting such contemporary low power SRAM memories is proposed to improve the energy-efficiency of instruction fetching. Xor-masking utilizes static program analysis statistics to produce optimal encoding masks to reduce the occurrence of the more energy consuming instruction bit values in the fetched instructions. The method is evaluated on LatticeMico32, a small RISC core popular in ultra low power designs, and on a wide instruction word high performance low power DSP. Compared to the previous "bus invert" technique typically used with similar SRAMs, the proposed method reduces instruction read energy consumption of the LatticeMico32 by up to 13% and 38% on the DSP core.

Keywords Asymmetric SRAM · Energy optimization · Instruction fetch · Low-power processors

Joonas Multanen
joonas.multanen@tut.fi

Customized Parallel Computing Group, Department of Pervasive Computing, Tampere University of Technology, Finland

1 Introduction

The *Internet of Things* (IoT) [1] era has lead to many modern devices that are wireless, intelligent and wearable. Small devices that can communicate wirelessly, process data and operate for extended periods of time without external power source create challenges for battery lifetime and processor technologies. The increased level of intelligence in devices often translates to increased battery consumption due to the additional required digital logic. Moreover, the *Thermal Dissipation Power* (TDP) becomes an issue in modern compute devices [2] with extremely small physical form factor limiting the size of cooling devices. It is now common that all the logic on a circuit cannot be switched on simultaneously in order to prevent malfunctions or failures. Since battery technology limits the maximum use time when not connected to an external power source, energy-efficient and low-power operation is often a mandatory requirement.

With advances in process technology, compute devices get smaller. Concurrently, the amount of data being processed is growing, making power consumption of memories a critical aspect in ultra-low-power compute devices. On-chip memories can consume up to half of total power in CPU-based digital designs [3, 4]. In order to keep memories up to speed with other system components, new technologies are researched. Possible replacements for SRAM include *Spin-Transfer Torque RAM* (STT-RAM) [5], hybrid *Non-Volatile Memories* (NVMs) [6] and *Embedded DRAM* (eDRAM). However, these technologies are not yet mature for wide adoption due to having challenges in cost, durability, or variability control. They are also *dedicated-process* [7] technologies, where the fabrication process used for rest

of the logic needs to be modified significantly, increasing design time and cost.

According to ISSCC [8], SRAM is still the most widely used technology for fast on-chip instruction and data memories and caches. However, while a replacement technology has not yet emerged, there is room for incremental improvement within SRAM designs. One such aspect is that normal, "symmetric" SRAM cells do not optimize the structure toward storing either high or low logic values found in instruction and program data. This bias can be exploited by designing memory cells in an *asymmetric* fashion, where power consumption is reduced when holding, writing or reading one logic value compared to the other. Such asymmetric cells have been designed and fabricated for memories and caches [9–12]. Information from statistical analysis of the data to be stored in such biased memories can be used to maximize the preferred values in memory using additional data encoding, leading to reductions in program power and energy consumption.

In this paper, an asymmetric SRAM designed and fabricated by Mori et al. [11] is taken under closer inspection. They implemented a chip on a 28 nm *Fully Depleted Silicon On Insulator* (FD-SOI) [13] technology, containing 8T memory cells, where reading the logical value '1' results in a lower energy consumption in comparison to value '0'. The SRAM was studied as an energy efficient data memory for image processors. Image data were stored in the memory, so that dark pixels contain more '0's and bright pixels contain more '1's. To reduce the total energy consumption in data memory, the authors added a majority voting logic to increase the amount of logical ones stored, and thus reduced read energy in their proposed memory. Compared to a standard 28nm FD-SOI SRAM of the same size, an 87% reduction in write energy and 52% reduction in read energy were reported.

In this article, we show that this type of asymmetric SRAM can be succesfully exploited also for program instruction memories in processor cores by utilizing offline program binary analysis, while only adding a very small hardware overhead. This paper extends our previous work [14] with the following main additions:

1. Analysis, implementation and evaluation of a scheme using multiple xor-masks instead of only a single one,
2. new heuristics for finding a mask set, and
3. evaluation of xor-masking on a wide-instruction word core, in addition to the small LM32 core.

The paper is organized as follows. Section 2 reviews previous methods for low-power instruction and data encoding. Section 3 introduces the proposed method. It is evaluated and compared to a previous state-of-the-art low-power encoding in Section 4. Section 5 concludes the paper.

2 Related Work

Majority of the previous work on low-power encoding has focused on reducing instruction address bus power and program data bus power. Existing methods can be divided into static and dynamic methods, depending on when the encoding is done. Static encoding is performed during program compile time, while dynamic is done on the fly at runtime, requiring additional hardware.

Bus-invert encoding [15] was introduced by Stan and Burleson in 1995 as a method to reduce power consumption in off-chip data buses. This has been extended in later work by further dividing the buses [16,17]. The bus-invert method dynamically encodes data based on consequtive data words. If the number of toggling bits between two consecutive data words (also known as the *Hamming distance*) is more than half, the data word is inverted with a logical NOT operation. In addition, a toggle bit is added to the data to indicate whether to invert the data at the other end or not. This is performed by logic connected to the memory block, at the end of the bus. At the end consuming the data (typically a load-store unit or an instruction fetch reading from memory), depending on the toggle bit, logical NOT is performed again to restore the original word. The obvious drawbacks of the method are the required extra bit and the power and area overhead of the logic.

Petrov and Orailoglu [18] proposed a static method where the instruction data bus was encoded with 16 possible data transformation operations. The transformations were selected using an exhaustive search by considering two consecutive instruction words in conjunction. The encoded words were written into instruction memory when loading the program, and the transformations were communicated to the processor's decode unit either at the start of a program, or before application hot-spots, such as loops. The performance of the method depends on frequent execution of the loops and small basic block sizes. Energy consumption is reduced due to the reduction in off-chip and on-chip bus toggling. However, the energy overhead of the additional implemented decoding logic was not analyzed and it is not clear if this overhead would counter the acquired savings. Moreover, the effect of on-chip bus energy reduction could be very small in a typical scenario, where the on-chip SRAM is close to the consumer.

Su et al. [19] used Gray coding and Cold scheduling to reduce the instruction address bus bit switching activity. Gray coding exploits the spatial locality

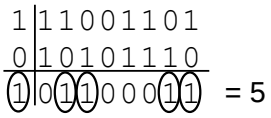


Fig. 1: Calculating the Hamming distance of successive words for bus-inverting. Left-most bits are *toggle bits*. The two words are XORed and the number of '1' bits in the resulting word is counted to make the toggle decision.

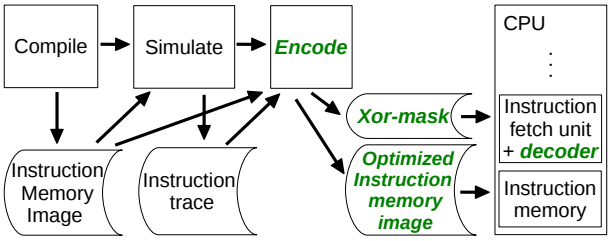


Fig. 2: Flow of the proposed encoding method. Stages and files for the proposed method are in bold italics.

found in memory accesses. That is, instructions are often fetched from consecutive memory addresses. These instructions can be reordered so that sequential addresses are located in Gray coded addresses, reducing the instruction address bus toggling. The authors combined Cold scheduling, a compiler energy reduction techique to schedule instructions depending on their relative energy, with Gray coding.

Musoll et al. [20] introduced *working zone* encoding based on the observation that programs usually operate on repeating sets of addresses (working zones). Similar to Gray coding, this allows the address bus toggling to be reduced. An identifier is given to each working zone and these combined with address offsets are signaled to the instruction control logic to indicate which working zone to use. The drawback of this method can be heavy hardware overhead, since the decoding and encoding algorithms are quite complex.

Benini et al. [21] created custom encoder and decoder logic for a given processor by analyzing instruction address traces. The authors claim that the method is generic and can be applied to various processors. However, the method showed little savings in total energy when taking into account the energy overhead of the additional encoding and decoding logic.

Yang et al. [22] noticed that certain data values on program data buses constituted a majority of the values on the buses. By exploiting this, they proposed a *frequent-value encoding*, where each end of the data bus had an identical codebook. If a value to be transmitted was found in the codebook, it was transported as an index of the codebook, otherwise as it was.

The previous work can be summarized as follows. Instruction memory power consumption optimizations mostly concentrate on address bus power reduction and, in particular, minimizing its toggling activity. Many of the reviewed methods focus on reducing off-chip data bus energy, while disregarding the energy overhead of the extra encoding and decoding logic. In contemporary compute devices, the instruction data bus energy consumption might not be so relevant due to instruction memories or caches integrated close to the consumer, the instruction fetch unit. In addition, L1 instruction cache miss rates in typical processor systems are low [23], indicating that when fetching from L2 cache to L1 cache, the bus energy consumption would not benefit much from encoding. It is very important to consider the energy and area overhead of the added logic due to required encoding and decoding. A complex implementation can nullify the benefits gained.

Compared to previous solutions, the proposed method, xor-masking, is able to statistically analyze individual bit positions from instruction execution during compile-time, resulting in a low-power encoding with minimal additional dynamic decoding logic.

3 Proposed Method

Many of the existing bus encoding algorithms concentrate on instruction address bus and program data bus encoding, where they aim to reduce the toggling activity. The proposed method instead tries to maximize the occurrence of one logical value over the other. For the instruction address bus, the existing approaches exploit the fact that instruction addresses are often accessed in a sequential order, where techniques such as Gray coding the addresses can help. The fetched instruction data, on the other hand, is usually less coherent [17] – the individual bits do not typically correlate with the instruction word fetched on the previous cycle.

The proposed method, *xor-masking*, analyzes statistical information about instruction memory accesses for individual programs by statically determining one or more optimal xor-masks to encode the instructions in each particular program to maximize the appearance of the desired logical value. However, the methods for controlling the use of the masks require careful design to avoid diminishing the benefits from the masking.

The Hamming distance for two words of data is the number of bits toggled in the successive words. It can be calculated by taking the logical XOR between the words, and counting the '1' bits of the result. An example is presented in Fig. 1. Here the previous (top)

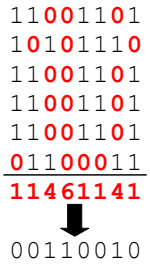


Fig. 3: Forming the xor-mask for an example instruction word trace. If the total amount of '0's for a given bit index is more than half (here three) of total bits for that index, it is xor-masked with a '1'.

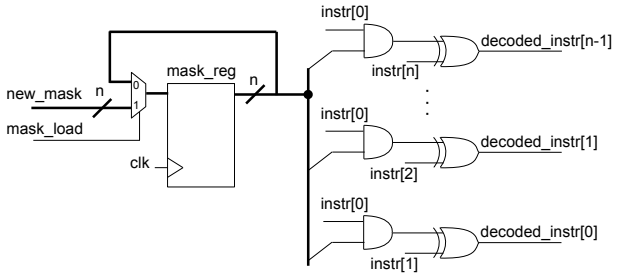


Fig. 4: Implementation of the decoding logic. The first word after reset is read in as a new mask. Mask is active, if first bit in instruction is logic '1'. Instructions are decoded by XORing them with the mask.

word has been inverted and we calculate the Hamming distance to the next word, including the added bit. In this case, we would invert the next (bottom) word, since the Hamming distance is greater than half of the word length. Counting the bits can be done with majority voters. Their implementation cost depends on the width of the data, and the type of the voter [24]. A word is decoded by XORing it with its added bit. For their SRAM circuit [11], Mori et al., modified the original bus-invert method by not minimizing the amount of toggles, but maximizing the amount of '1's instead. This approach corresponds to calculating the Hamming distance with the other word constantly being all '1's. This simplified the logic, since the previous data value was not compared to the current one, thus saving a register and a logical XOR between the words. They used this encoding for general data. In this paper, this method is referred to as *Majority Voter Encoding* (MVE).

Encoding words in this manner is not ideal for reducing the power consumption when the SRAM is used as an instruction memory. We have found that methods utilizing statistical instruction analysis can be used to increase the number of the low read-energy logic values in the memory. This paper concentrates on single and multiple mask scenarios, where the required additional hardware logic is minimized.

Like the static instruction transformation of Petrov and Orailoglu [18], the proposed method conveys application-specific information to the processor. In our case, this information is reduced to a compact *xor-mask*, which is used to decode the encoded words during the instruction fetch stage. Encoding the words incurs no additional hardware logic, since the encoding is performed offline as a compilation pass over the instruction bit image. As a side effect of an encoding that aims to maximize one logical value over another, the proposed method often reduces bit toggling between successive instruction words. Considering a multi-level instruction memory hierarchy, this reduced toggling can save additional energy since off-chip buses can consume notable amounts of energy.

Decoding the words is done by performing a logical XOR between the mask word and each encoded word. In addition to incurring extremely little additional logic, this allows easy implementation of the method to existing architectures. Fig. 4 depicts the logic required.

The update of the xor-mask can be integrated with the program image, so that after reset the first instruction address holds the xor-mask. This is fetched and treated as a mask word which is stored to the mask register. After this, the succeeding words are treated normally as instructions. For even easier integration with program compilers, the mask word and a jump to reset address can be placed to the last addresses in instruction memory to avoid relocating the program instructions.

3.1 Single Xor-Mask

The simpler alternative includes only a single xor-mask for the whole program. For optimizing the selection of a single xor-mask, we propose to use static analysis enhanced with execution profiling data from a typical program run. The analysis (see Fig. 2) starts by taking the instruction memory bit image of the optimized program. An instruction address trace with an unmodified instruction memory image is produced by simulating the targeted program using typical input data. Then, for each bit index, the total amount of logical '1's and '0's weighted according to execution count is calculated. The mathematical presentation for this is shown in Eq. 1.

$$a_i = \sum_{c=1}^{C} b_{c,i}, \quad b_{c,i} \in \{0,1\} \tag{1}$$

An instruction trace is iterated over each cycle c, and C is the total amount of instructions, and $b_{c,i}$ is ith bit of the instruction at cycle c. a_i gives the average logic value of the ith instruction bit for an execution trace. This process is also illustrated in Fig. 3. If the amount of '0's is greater than half of total bits in the word, a '1' is assigned to that index of a *xor-mask* that is applied to the instruction word. This is described in Eq. 2.

$$m_i = \text{'1' if } a_i > 0.5 \text{ else '0'} \quad (2)$$

The xor-mask is optimized for each program separately. For the memory examined in this paper, logic '1' reads are preferred for low energy. However, the method can equally efficiently be applied to cases, where logic '0' is the preferred value.

3.2 Multiple Xor-Masks

To utilize multiple xor-masks, consideration is required for the switching granularity, mask storing/fetching, number of masks, and controlling the use of different masks.

3.2.1 Mask Switching Granularity

The use of a single xor-mask per program allows minimal decoding hardware overhead. However, this results in too coarse granularity when the program's instruction mix differs greatly across the hot basic blocks in the loaded program. For finer granularity, multiple selectable xor-masks can be used.

Theoretically, the best granularity in terms of minimizing instruction word read energy consumption occurs when each instruction has an individual mask that causes all the read bits to be flipped to the desired value. However, this would in the worst case require as many masks as there are instructions. For coarser granularity, groups of instructions withing the program should be masked with a single mask.

Basic Block Level Granularity Program *basic blocks* [23] are sections of code that execute in a sequential manner from start to end. There can be branches only to the beginning of the basic block, or from the end. Thus, they offer sequences of instructions that are grouped together naturally with a feasible point of switching the mask in the beginning of the block. When changing the mask only at the start of each basic block, there is no risk of accidentally decoding words with wrong masks. Thus, basic block level granularity is a logical choice for exploiting multiple masks.

Loop Level Granularity Programs typically consume most of their time in loops. Especially embedded applications tend to spend 90 percent of their execution time in only 10 percent of the code. This is known as the 90-10 rule. Of those 10 percent, approximately 85 percent are inner loops [25]. Of course, this depends highly on the nature of the program. This motivates masking a subset of program code, loops.

3.2.2 Mask Storing and Fetching

One possible storage scheme is storing the masks directly in the instruction memory. Since the masks have the same width as the instruction words and instruction memory depth may not need to be increased due to program length, this can be a convenient solution.

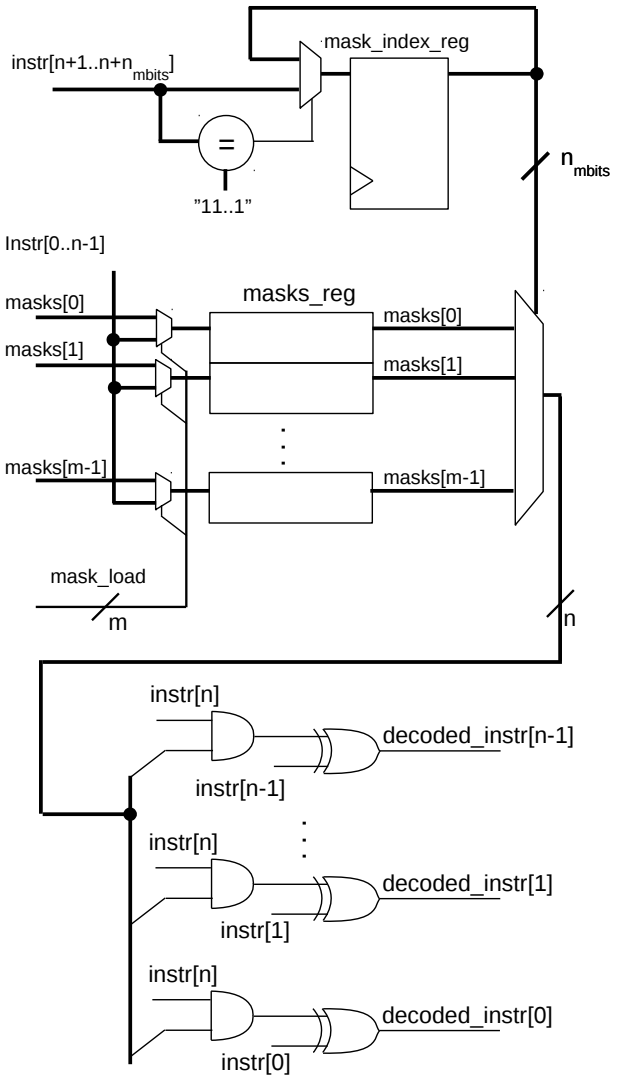


Fig. 5: An implementation of decoding logic using multiple masks.

However, this masking scheme increases the program execution time, since a stall is required when reading a mask from the instruction memory. Furthermore, the mask changing has to be identified somehow. In this scheme, there is no advantage in limiting the number of possible masks, as a new mask needs to be read in the beginning of each basic block in any case.

If loop level granularity is utilized for mask switching, a loop-specific mask could be used when executing the loop and a generic one otherwise. This mask could then be changed when entering the loop and when exiting it, possibly incurring only a small overhead in execution time and control energy in contrast to the instruction energy reduction. This scheme requires the instruction set of the architecture to have room for a custom instruction to indicate that the next instruction word is a new mask.

In this loop-specific masking scheme, a total of four instruction words are added to a loop. Two words are added before the loop: one to indicate that the next word is a mask for the loop and immediately after it the mask itself. When leaving the loop, this is repeated, but now the mask is the generic mask to be used outside loops. The instruction word indicating the mask change is architecture-dependent and is limited by the available bit combinations in the instruction set architecture.

In another scheme, the masks could be stored in the data memory of the processor. This could result in a smaller overhead in execution time, since a new mask could be fetched concurrently with the next instruction.

For this paper we evaluated a scheme where a number of masks are stored in *mask registers* in the instruction fetch component of the processor core. Storing the masks is done during reset, and during execution, mask changes are indicated by mask control bits added to each instruction word. This approach allows the masking of the words to be performed as a post-pass after the compiler's code generation. Each basic block's first control bits indicate the mask to be used, even if it is equal to the previous one. For the rest of the block's instructions, a *no-mask-change* bit sequence is used. The amount of control bits n_{mbits}, can be calculated with Eq. 3

$$n_{mbits} = \lceil log_2(m) \rceil, \quad m \geqslant 3, \tag{3}$$

where m is the amount of masks. In our approach, there are m sequences that include $m-2$ sequences for basic blocks whose energy can be reduced the most, one sequence indicating a generic mask for the rest of the basic blocks, and a no-mask-change sequence.

A hardware implementation using multiple masks is illustrated in Fig. 5. In this case, logic '1' is the preferred value in terms of least SRAM read energy consumption.

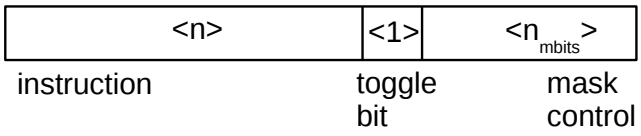


Fig. 6: For the use of xor-masks, our implementation adds a toggle bit and mask control bits to the instruction word.

A mask change is indicated by n_{mbits} *Most Significant Bits (MSBs)* of the current instruction word. When all MSB bits are ones, the last set mask is used. This implementation adds the mask control to the instruction words as depicted in Fig. 6. For the sake of clarity, the toggle bit and mask control bits are moved to the MSB indexes as opposed to the toggle bit being in the LSB end in the single xor-mask implementation. The nth bit is now the toggle bit.

In order to support pre-emptive multitasking or other hard interrupts, xor-masking requires additional hardware support for saving and restoring the masks during a context switch. This was left as a future work.

3.2.3 Multiple Mask Search Algorithm

Finding the best masks per program is not trivial. Exhaustive search eventually produces the optimal masks, but for an n-bit instruction word, there are 2^n cases to search. For a 32-bit instruction word, this already becomes unfeasible with the current computer technology.

Our proposed method for finding multiple masks per program is depicted in Algorithm 1. On lines 1–3, a single xor-mask is searched for each basic block. Next, absolute energy reduction for each block is calculated and the blocks are sorted by the most energy reduced. This sorting is described in Algorithm 2. The best $m-2$ blocks and their masks are assigned on line 6. For the rest of the blocks, a single generic mask is now searched and assigned on line 7. This is the same as finding a single mask for the whole program, but now the already assigned blocks are not included. Execution amounts of each basic block were counted on line 4 and are now used to sort the control word sequences on line 8. To minimize the energy consumption, the most executed block is assigned the least energy consuming sequence and so forth.

The generic mask may not be optimal for each of the blocks using it. The energy consumption can in some cases benefit from using the already assigned $m-2$ masks. Therefore, on lines 9–21, we next try each of the assigned masks for each of the blocks that the generic mask was previously assigned to. After the possible re-

assignment of these blocks, the generic mask is again searched on line 22 for the remaining blocks, since the previous one may not be optimal anymore. This may, however, still result in the previous generic mask. Finally, the input data is masked and returned along with the corresponding control words and masks.

Algorithm 1 Multiple mask search

Input: user-defined number of masks m, number of basic blocks *numBlocks*, execution *trace*
Output: *controlWords*, *maskedData*, *masks*

```
 1: for each basic block do
 2:     Find mask by statistical analysis of trace
 3: end for
 4: Count executions for each basic block from trace
 5: Sort masks by block energy reduction (Algorithm. 2)
 6: Assign m masks for m − 2 blocks with best energy reduction
 7: Find and assign genericMask for rest of blocks
 8: Assign controlWords according to block executions
 9: for each numBlocks−(m−2) basic block in energy-sorted blocks do
10:     bestReduction = reduction(genericMask)
11:     bestMask = genericMask
12:     for each mask in m − 2 best masks do
13:         if reduction(mask) > bestReduction then
14:             bestReduction = reduction(mask)
15:             bestMask = mask
16:         end if
17:     end for
18:     if bestReduction < reduction(genericMask) then
19:         Assign bestMask to basic block
20:     end if
21: end for
22: Find new genericMask for rest of blocks
23: Mask data in basic blocks
24: return controlWords, maskedData, masks
```

Control bit sequences add to the instruction read energy. To minimize the control bit energy overhead, different sequences are sorted by their execution amounts. The result is used to determine the order of masks in the mask register. The indexing should be done so, that the occurrence of the preferred logical value in the index field is largest in the most executed instruction/block, and smallest in the least executed instruction/block. As mask change control sequences only occur at the beginning of each basic block, the no-mask-change sequence is assigned the least energy-consuming bit sequence, since all the rest of the instructions in the block use it. That is, all bits are of the favoured value. The control bit sequences for the generic mask and the best masks are then sorted and assigned.

Algorithm 2 Basic block sorting by energy reduction

Input: basic *blocks*, execution amounts *executions* for each basic block, *masks*, energy *zeroEnergy* for reading a zero, energy *oneEnergy* for reading a one
Output: basic blocks sorted in descending order of energy reduction

```
 1: for each block in blocks do
 2:     for each instruction in block do
 3:         blockEnergy += ones*oneEnergy*executions
 4:         blockEnergy += zeroes*zeroEnergy*executions
 5:     end for
 6:     mask block
 7:     for each instruction in masked block do
 8:         maskedEnergy += ones*oneEnergy*executions
 9:         maskedEnergy += zeroes*zeroEnergy*executions
10:     end for
11:     reductions[i] = blockEnergy-maskedEnergy
12:     sort reductions
13: return basic blocks sorted by energy reduction
14: end for
```

4 Evaluation

Xor-masking was evaluated on two architectures. The first is LatticeMico32 [26] RISC architecture. It is an interesting choice for evaluation due to its open licensing agreement, freely available program compiler and generation of RTL code, and its use in a number of publications and projects [27,28]. Detailed specifications of the evaluated processor are listed in Table 1. The evaluation platform was a minimal setup, where instruction and data memories were scaled to be large enough to accommodate all of the individual bechmark program instructions and data at a time. The benchmark programs were chosen to represent typical applications in microcontrollers in low-power scenarios. The benchmarks are listed in Table 2.

Secondly, xor-masking was evaluated on a larger low power high performance DSP core designed by the authors for an *Image Signal Processing* (ISP) use case [29] and features a wide instruction word architecture. This core was implemented with *TTA-based Co-design Environment (TCE)* [30], a software toolset for design and programming of programmable processors based on the *Transport Triggered Architecture* (TTA) model. TTAs

Table 1: LatticeMico32 features.

Clock frequency	18.2MHz
Instruction set architecture	RISC
Instruction width	32 bits
Instruction memory	On-chip SRAM, 32kB
Data memory	On-chip SRAM, 400kB
Dedicated hardware	Hardware multiply unit Hardware divide unit Pipelined barrel shifter

Table 2: LatticeMico32 benchmark programs.

Suite	Programs
CHStone [32]	adpcm, aes, blowfish, gsm, jpeg, mips, sha
DSPStone [33]	matrix
Coremark [34]	coremark

exploit instruction-level parallelism found in programs by using a wide instruction word. They can also leverage *Single Instruction Multiple Data* (SIMD) operations. These qualities make them highly applicable to signal processing purposes. Xor-masking is applied to the DSP core to demonstrate the effectiveness of the method not only on RISC architecture, but also on signal processing cores.

Although the TTA core was designed for ISP purposes, it performed well with *Software Defined Radio* (SDR) programs with only minor modifications to the interconnect network and using a larger program data memory. The benchmark programs used were two SDR-specific programs, *Layered ORthogonal lattice Detector* (LORD) [31] and *Minimum Mean Square Error* (MMSE). They are used in modern wireless telecommunication, specifically in *Multiple-In, Multiple-Out* (MIMO) channel detection.

The energy consumption was evaluated by calculating energy costs for reading a single '1' and '0' based on the previously published measurements for the original SRAM [11]. In the original 64kB SRAM, for a 16-bit word, energy consumption for reading all '0's and reading all '1's was 1440 fJ/cycle and 148.5 fJ/cycle, respectively. To calculate read energy per bit, these were divided by 16 resulting in 90.00 fJ/cycle and 9.28 fJ/cycle for the cost of reading a single '1' bit and a '0' bit, respectively. Next, instruction address traces for the benchmark programs were produced from Modelsim simulations. Using the calculated read energies, instruction traces and instruction memory bit images encoded according to each of the two methods, the total SRAM energy consumption for each benchmark program was calculated.

Considering the write energy consumption is relevant especially for designs including dynamic instruction caches. Cache misses translate to writing cache lines. However, the difference in energy reduction for writing all '0's and all '1's to the referred SRAM, compared to a regular SRAM, was reported as negligible, 1%.

The proposed method does not affect the time of replacing cache lines compared to the referred method. In this sense, write energy evaluation is not interesting, since the difference between the proposed method and the reference method is not significant.

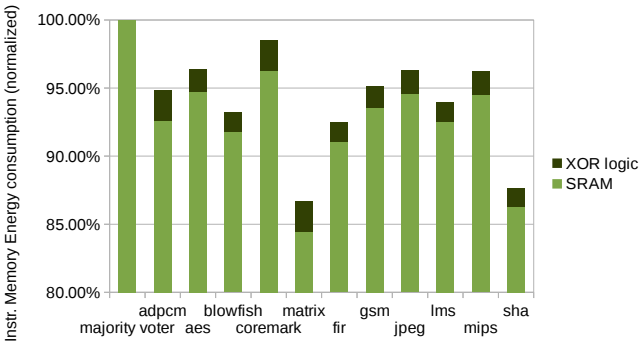


Fig. 7: Proposed method's instruction memory energy consumption for benchmark programs, normalized to the corresponding consumptions with MVE.

If both the instruction memory and the instruction cache are implemented with the referred SRAM technology, the total energy reduction depends on the combined energy consumption of the two. In this case, the decoding logic would be implemented after the instruction cache.

In the case of the proposed method, the results also include the energy consumed by the majority logic proposed with the referred SRAM technology. In our case, this majority logic is not used and instead, the decoding logic would be implemented in the instruction fetch unit. This overhead is present in our energy numbers for the proposed method. The improvement in energy consumption of our proposed method compared to the referred method would, therefore, be better than the results presented in this paper. However, this overhead is difficult to estimate, since the authors did not report the majority logic energy consumption individually, but rather the overall SRAM's. Moreover, the majority logic relies on a pull-down network and a sense amplifier. However, regardless of this overhead, our proposed method still achieves lower energy consumption in all 11 benchmark programs on the LatticeMico 32 and both benchmarks on the DSP core.

4.1 LatticeMico32 Results

Energy consumption normalized to MVE for the benchmark programs is presented in Fig. 7. As is expected, the energy consumption depends on the dynamic instruction mix in each of the benchmarks. The more there are instructions resembling each other on the bit level, the more the proposed method can save energy. The worst case is when the occurrence of '1' and '0' at each bit position is exactly the same. In this case, inverting the bit index results in no savings in energy.

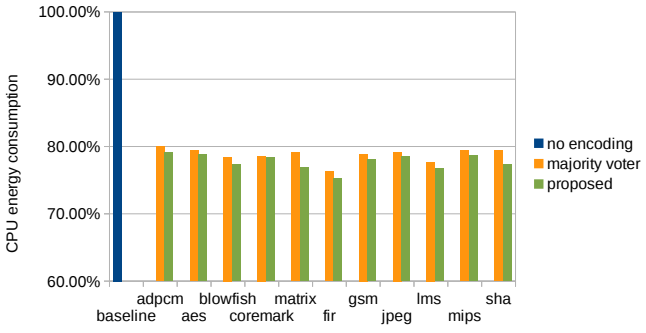


Fig. 8: Comparison of total CPU energy for MVE and the proposed method. Normalized to the level of CPU with unencoded instructions in SRAM.

The energy overhead of the added logic is small compared to the overall energy, on average 1.0% of the SRAM energy. The best energy reduction, 13.3%, was achieved in *matrix* benchmark and the lowest reduction, 1.5%, in *coremark*. On average, the reduction was 6.2%.

In order to estimate the CPU total energy consumption, the described LatticeMico32 was synthesized on a 28nm ASIC standard cell technology. The instruction memory consumed 37.7% of the total energy after synthesis for this particular implementation. Using this number, the effect on the total CPU energy was calculated. This is presented in Fig. 8. The largest total energy reduction, 24.8%, was achieved with *fir*. In *matrix*, total CPU energy consumption was 5.0% less compared to MVE. On average, this reduction was 2.4%.

The reduction in bit switching activity is compared to the majority-voter-encoded words, and the results are presented in Table 3. Both of the encoding methods add a toggle bit to the unencoded words, increasing the total amount of bits read. The proposed method reduces the bit switching activity in all but three benchmark programs compared to the MVE. The reductions are small and in the best case 3.3%. In a realistic implementation in an ultra-low-power IoT device, off-chip memory would be unlikely to be used for storing instructions and the effect in energy reduction for an on-chip bus would be negligible. The decoding logic size for LM32 was 377 equivalent NAND2 gates with the single mask scheme.

With the proposed multiple xor-mask scheme, LM32 results for the implementation presented in this paper are depicted in Fig. 9. X-axis values are the number of masks used, and translate to $x - 1$ basic blocks with the best energy reductions masked and a generic mask for the rest of the blocks. Y-axis is the reduction compared to using a single xor-mask. Some benchmark programs have less basic blocks than the maximum X-value, hence the missing values in some of the lines. Negative values indicate, that the overhead of masking was not beneficial to the energy consumption, but instead increased it.

Table 3: Bit switching activity.

Benchmark	MVE	Proposed Δ(%)
adpcm	764 000	3.3
aes	413 000	-0.3
blowfish	6 257 000	0.1
coremark	4 798 000	-0.7
matrix	2 996 000	0.0
fir	5 000	-1.4
gsm	231 000	0.4
jpeg	25 016 000	1.0
lsm	8 000	0.2
mips	213 000	1.1
sha	5 819 000	2.9

In two benchmark programs, *matrix* and *mips*, the reduction over a single xor-mask is notable, 8-9% with 3 and 14 masks. Rest of the programs either gained small reductions in energy (up to 1.6% in *jpeg*), or did not benefit from multiple masks (at best 1.2% decrease in *fir*). Depending on the benchmark, the benefit of multiple masks is reduced in steps as the number of masks surpasses a power of two. At these points, this is expected, since the number of control bits is increased. Some benchmarks show more sporadic behaviour, which seems to result from re-assigning the generic mask as described in Algorithm 1.

4.2 DSP Core Results

For the DSP core, the instruction memory energy consumption is reduced by 38.1% in *LORD* and 38.4% in *MMSE*. This is illustrated in Fig. 10a. As listed in Table 4, both of the the benchmarks have a high proportion of *No Operations* (NOPs) of all instructions, 74.6% in *LORD* and 69.6% in *MMSE*. When searching for the masks, the instruction slots in basic blocks that mainly contain NOPs heavily affect the mask. In the case of the DSP core, the instruction set encoding is such that NOP encodings naturally contain mostly non-preferred

Table 4: DSP core benchmark programs.

Benchmark	Instructions	Basic blocks	NOPs per all instructions
LORD	975	36	74.6
MMSE	1136	16	69.6

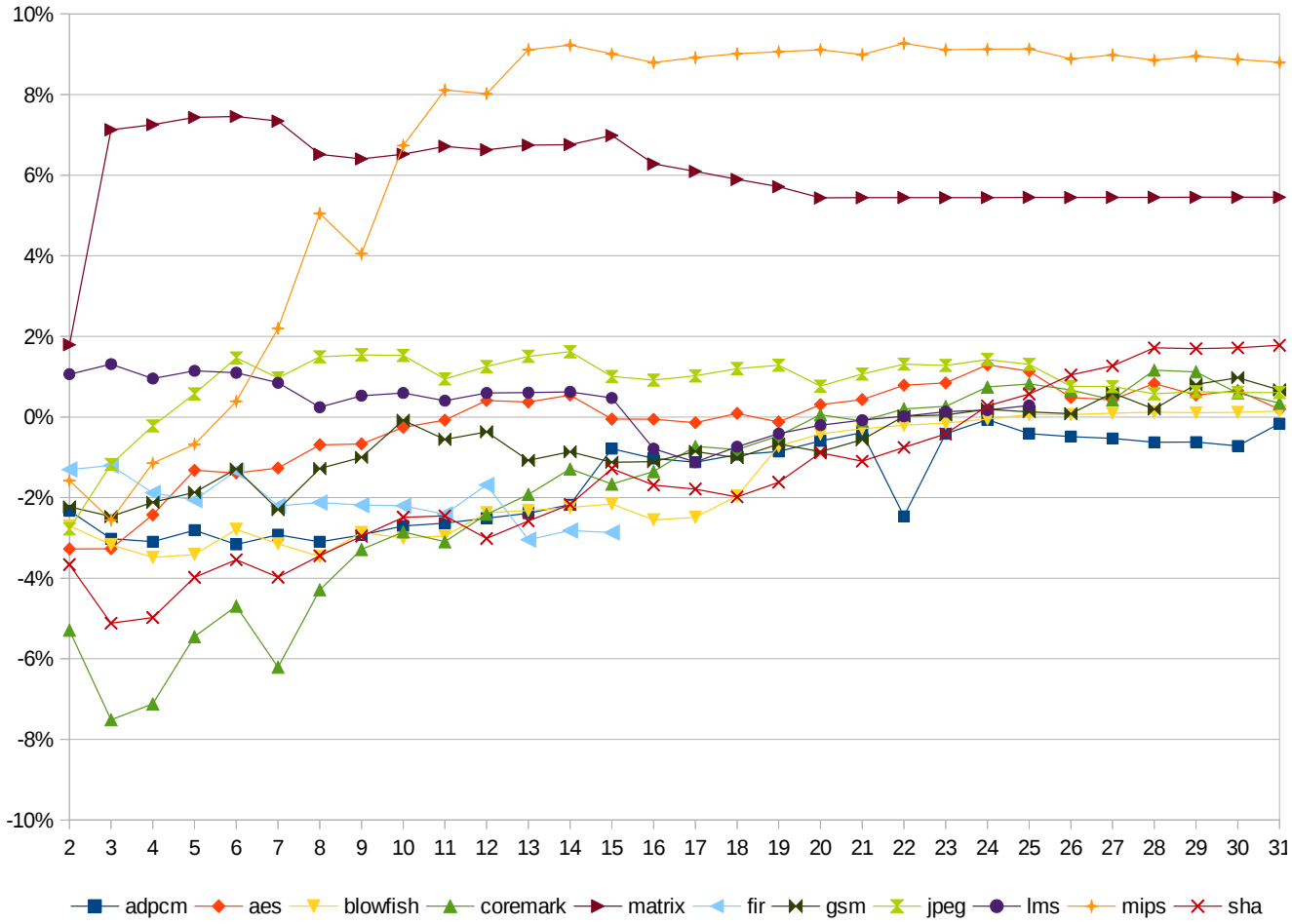


Fig. 9: LatticeMico32 instruction memory energy reduction as a function of number of masks. Improvement over xor-masking with a single mask. Positive numbers indicate a reduction in energy consumption.

bit values, reducing energy consumption notably. Area overhead of xor-masking with a single mask was 1544 equivalent NAND2 gates.

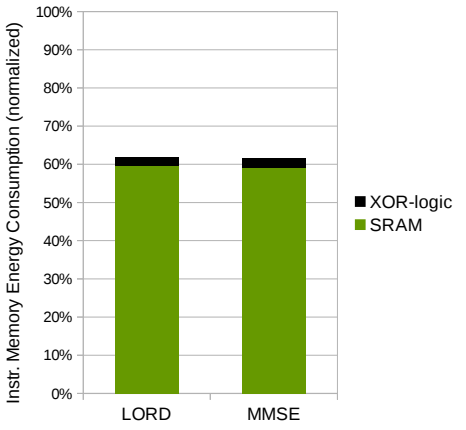
For the DSP core, results of using multiple masks are shown in Fig. 10b. The energy reductions compared to using only one mask are small, less than 1% in all cases. This is due to the high proportion of NOPs. Because of the NOPs, many of the different basic blocks' masks are very similar. In this case, using multiple masks does not provide energy reductions. The control bit overhead is still added, however, since the amount of control bits per word is small compared to the instruction words, and the control bits mostly contain the preferred values, the added energy consumption is quite small.

If the NOPs do not dominate the mask search, the efficiency of xor-masking when using multiple masks and our proposed implementation seems to depend on the distribution of energy per basic block. If the energy is distributed evenly among all basic blocks, overhead from the mask control seems to counter the benefit of masking. If the best basic blocks consume majority of the total energy, using multiple masks seems more beneficial.

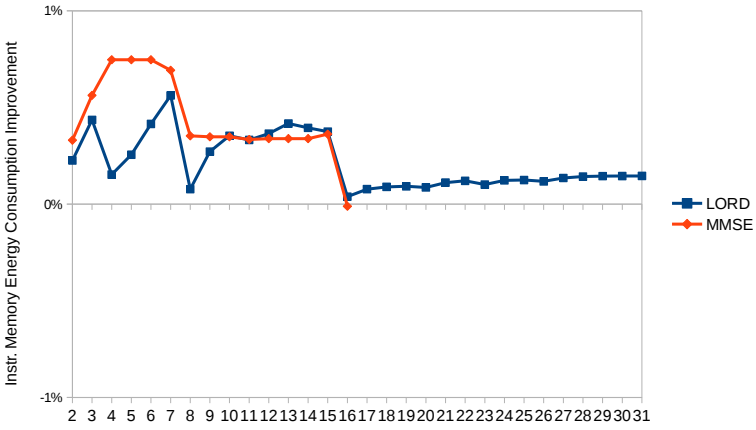
4.3 On the Optimality of the Proposed Method

Theoretical best-case scenario for any given instruction trace occurs, when all bits in the trace are of the preferred value. To estimate the remaining energy saving potential of our proposed technique on the two cores, we calculated the instruction memory energy consumption for each benchmark when all bits in its trace are of the less energy consuming logic value. Because using multiple masks only brought significant improvements in two benchmarks, this evaluation was done using a single mask per benchmark.

Comparison to energy consumption achieved with the proposed method per benchmark is presented in



(a) One mask.



(b) DSP core instruction memory energy reduction as a function of number of masks. Positive numbers indicate a reduction in energy consumption.

Fig. 10: Energy reduction with xor-masking for DSP core.

Fig. 11. For LM32, *coremark* was closest to its minimum, consuming 17% more energy. Worst benchmark was *matrix*, consuming 164% more energy. On average, the 11 benchmarks consumed 61% more energy compared to the minimum. The DSP core consumed 94% and 92% more energy with *LORD* and *MMSE*, respectively. On average they consumed 93% more energy than the minimum.

This suggests that there is still potential for further energy savings. However, additional improvements depend on the instruction mixes of the benchmark programs. If the instructions utilized by a benchmark differ greatly on the bit-level, it is hard to gain additional reductions in energy.

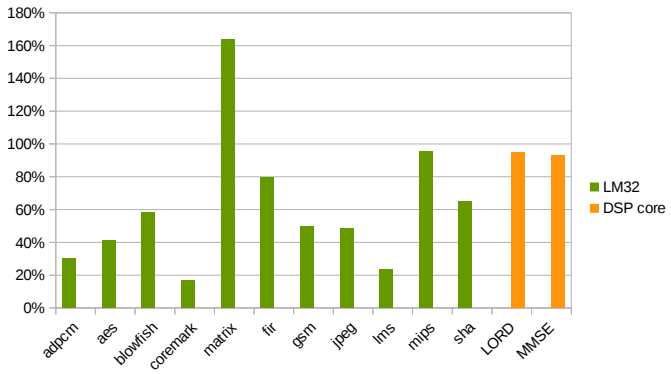


Fig. 11: Increase in energy consumption of benchmark programs compared to the optimal case.

4.4 Limitations in Evaluation

As the results obtained with xor-masking are dependent on the program execution and individual instructions, programs whose control flow is data-dependent can perform worse with other input values than the ones that were used to form the xor-masks. In cyclic applications, where the amount of loop iterations is not known at compile-time, it is difficult to conclude the optimal masks. Estimating these worst-case scenarios shares similar issues with *Worst-Case Execution-Time* (WCET) analysis [35]: if the amount of loop iterations is not constrained or the program execution time depends on the input data, it is very hard to accurately estimate the WCET. In our case, this leads to estimating the worst-case energy consumption being difficult.

4.5 Estimation of Loop Masking

In this evaluation, we consider loops with exit points only at the end of the loop found in the LM32 benchmarks. This is to simplify the mask changing by only allowing it at entering and exiting the loop and, therefore, avoid accidentally decoding instructions with the wrong mask.

In order to estimate the savings, a mask was searched for each loop similar to the multiple mask case, but using only one control bit to indicate whether a word in memory is masked or not. Then, a generic mask was searched for the rest of the instructions. The calculated results are presented in Table 5. The energy consumption is compared to the single xor-mask case. The largest energy savings were reached in *lms*, *fir* and

Table 5: Energy reduction with only loops masked.

Benchmark	Avg. loop length	Avg. loop executions	Loop cycles per total cycles (%)	Energy reduction (%)
adpcm	2.2	95.7	0.3	0.8
aes	2.3	124.6	0.6	1.0
blowfish	2.6	187.4	0.1	0.1
coremark	3.3	33.0	0.0	0.0
matrix	5.0	2769.0	4.5	0.4
fir	5.0	101.0	71.8	1.8
gsm	2.8	123.2	1.3	-0.6
jpeg	2.5	7912.6	0.7	1.5
lms	5.0	101.0	47.8	3.8
mips	5.0	70.0	1.5	0.2
sha	3.7	52.7	0.0	0.0

jpeg enchmarks, 3.8%, 1.8% and 1.5%, respectively. Overall, the savings were small. The saving depends somewhat on loop lengths and number of iterations, but more on the amount of cycles executed inside loops compared to the overall cycles.

4.6 Impact on Memory Size

The masking schemes introduced in this paper each have a different effect on the instruction memory size. When using a single mask per program, the effect is minimal: if a mask is stored as a first instruction in the memory, the memory size only increases by one. In the multi-mask scheme, a mask is required for each basic block in a program and the instruction memory size is increased by the amount of basic blocks. For the loop masking scheme described in this paper, changing the mask for a loop requires four extra instructions. Thus, for the loop masking scheme the memory size overhead is obtained by multiplying the amount of loops by four. For the multi-mask and loop masking scheme applied to LM32, the effect on memory size is presented in Table 6. The relative memory size increase in multi-mask scheme depends on the amount of basic blocks in a program in relation to its original instruction amount. The largest overhead, 26.9%, was incurred in *coremark*, which had 624 basic blocks in relation to 2320 instructions. This is explained due to *coremark* being quite program control oriented. *Fir* and *lms* had the smallest overhead, 3.2% and 3.3% respectively. The loop masking scheme incurred only slight memory size overheads, with a maximum of 2.3% in *adpcm* and a minimum of 0.5% in *lms* and *mips*. This is due to the scheme only selecting loops that only have exit points in the end. Therefore, the amount of loops that a mask is applied to is typically small.

Table 6: Impact on LM32 instruction memory size when using multiple xor-masks.

Benchmark	num. instructions	multi-mask Δ(%)	loop masking Δ(%)
adpcm	2949	+6.4	+2.3
aes	4212	+10.0	+1.1
blowfish	2306	+6.5	+1.2
coremark	2320	+26.9	+1.0
matrix	603	+9.8	+0.7
fir	472	+3.2	+0.8
gsm	1912	+13.0	+1.0
jpeg	4308	+12.7	+0.7
lms	764	+3.3	+0.5
mips	819	+7.8	+0.5
sha	1638	+7.5	+0.7

5 Conclusions

Energy consumption of on-chip and off-chip memories offers optimization opportunities in pervasive compute devices. In this paper, a novel statistical method, xor-masking, was proposed to reduce the instruction fetch energy consumption in asymmetric SRAM technologies. The proposed method was evaluated on LatticeMico32 RISC soft-core with 11 benchmark programs, and a wide instruction word architecture DSP core with two typical programs for software defined radio use.

Including the energy overhead of the decoding logic, the proposed method consumes up to 13% less energy compared to the state-of-the-art majority voter encoding on the same SRAM. The total CPU energy reduction is up to 5% compared to majority voter encoding. The energy consumption with the proposed method was smaller in all benchmark programs compared to the majority-voter-encoding. In addition, the proposed method reduces instruction data bus toggling up to 3.3% compared to the referred method. When using multiple masks, there were notable energy savings of 7-9% compared to the single xor-mask scheme in two benchmarks, and minor improvements in majority of the benchmarks. Instruction memory read energy reduction for the DSP core was 38.4% in the best case when using one mask.

Future work involves researching methods for applying xor-masking to dynamically linked code, system calls and context switches. Different schemes, implementation and its logic overhead need to be carefully considered.

6 Acknowledgments

The authors would like to thank the TUT Graduate School, Academy of Finland (project PLC), Finnish Funding Agency for Technology and Innovation (project "Parallel Acceleration 3", funding decision 1134/31/2015), and ARTEMIS JU under grant agreement no 621439 (ALMARVI).

References

1. Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15), 2010.
2. M. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, June 3-7 2012.
3. D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. Legat. SleepWalker: A 25-MHz 0.4-V Sub-mm^2 7- μm^2 μW/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor nodes. *IEEE Journal of Solid-State Circuits*, 48(1), Jan. 2013.
4. A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 23-25 2010.
5. X. Fong, Y. Kim, K. Yogendra, D. Fan, A. Sengupta, A. Raghunathan, and K. Roy. Spin-transfer torque devices for logic and memory: Prospects and perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1), Jan 2016.
6. J. Hu, C.J. Xue, Q. Zhuge, WC. Tseng, and E.H.-M. Sha. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *Design, Automation Test in Europe Conference Exhibition*, Mar. 14-18 2011.
7. L. Benini, A. Macii, and M. Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *Transactions on Embedded Computing Systems*, 2(1), Feb. 2003.
8. ISSCC: ISSCC 2016 tech trends, Feb. 2016. http://isscc.org.
9. N. Azizi and F.N. Najm. An asymmetric SRAM cell to lower gate leakage. In *Proceedings of the 5th International Symposium on Quality Electronic Design*, Hangzhou, China, Mar. 15-16 2004.
10. M. Imani, S. Patil, and T.S. Rosing. Hierarchical design of robust and low data dependent FinFET based SRAM array. In *Proceedings of the International Symposium on Nanoscale Architectures*, Boston, MA, July 8-10 2015.
11. H. Mori, T. Nakagawa, Y. Kitahara, Y. Kawamoto, K. Takagi, S. Yoshimoto, S. Izumi, K. Nii, H. Kawaguchi, and M. Yoshimoto. A 298-fJ/writecycle 650-fJ/readcycle 8T three-port SRAM in 28-nm FD-SOI process technology for image processor. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Jose, CA, Sep. 28-30 2015.
12. A. Teman, A. Mordakhay, J. Mezhibovsky, and A. Fish. A 40-nm sub-threshold 5T SRAM bit cell with improved read and write stability. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(12), Dec. 2012.
13. K. K. Young. Short-channel effect in fully depleted soi mosfets. *IEEE Transactions on Electron Devices*, 36(2), Feb 1989.
14. J. Multanen, T. Viitanen, P. Jääskeläinen, and J. Takala. Xor-masking: A novel statistical method for instruction read energy reduction in contemporary SRAM technologies. In *International Workshop on Signal Processing Systems*, Dallas, TX, Oct. 26-28 2016.
15. M.R. Stan and W.P. Burleson. Bus-invert coding for low-power I/O. *IEEE Transactions on Very Large Scale Integration Systems*, 3(1), Mar. 1995.
16. Y. Shin, S-I. Chae, and K. Choi. Partial bus-invert coding for power optimization of application-specific systems. *IEEE Transactions on Very Large Scale Integration Systems*, 9(2), Apr. 2001.
17. Ji G. and Hui G. A segmental bus-invert coding method for instruction memory data bus power efficiency. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, Taipei, Taiwan, May 24-27 2009.
18. P. Petrov and A. Orailoglu. Application-specific instruction memory customizations for power-efficient embedded processors. *IEEE Design Test of Computers*, 20(1), Jan. 2003.
19. C. Su, C. Tsui, and A. Despain. Saving power in the control path of embedded processors. *IEEE Design and Test of Computers*, 11(4), Winter 1994.
20. E. Musoll, T. Lang, and J. Cortadella. Working-zone encoding for reducing the energy in microprocessor address buses. *IEEE Transactions on Very Large Scale Integration Systems*, 6(4), Dec. 1998.
21. L. Benini, G. De Micheli, E. Macii, M. Poncino, and S. Quez. System-level power optimization of special purpose applications: the beach solution. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Monterey, CA, Aug. 18-20 1997.
22. J. Yang, R. Gupta, and C. Zhang. Frequent value encoding for low power data buses. *ACM Transactions on Design Automation of Electronic Systems*, 9(3), Jul. 2004.
23. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.
24. B. Parhami. Design of m-out-of-n bit-voters. In *Conference Record of the Twenty-Fifth Asilomar Conference on Signals, Systems and Computers*, volume 2, Pacific Grove, CA, Nov. 4-6 1991.
25. D.C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt. Profiling tools for hardware/software partitioning of embedded applications. *SIGPLAN Notices*, 38(7), July 2003.
26. Lattice Semiconductor. Latticemico32, Feb. 2016. http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/ IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx.
27. Z. Ben Salem, M. W. Youssef, and M. Abid. Prototyping cost-effective secure application server on a chip (sasoc) a case study for monitoring sensor network. In *International Conference on Wireless and Ubiquitous Systems*, Sousse, Tunisia, Oct. 18-10 2010.
28. P. Schleuniger, S. McKee, and S. Karlsson. *Design Principles for Synthesizable Processor Cores*. Feb. 28-Mar. 2 2012.
29. J. Multanen, H. Kultala, M. Koskela, T. Viitanen, P. Jääskeläinen, J. Takala, A. Danielyan, and C. Cruz. Opencl programmable exposed datapath high performance low-power image signal processor. In *IEEE Nordic Circuits and Systems Conference*, Nov. 1-2 2016.

30. O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez. Customized exposed datapath soft-core design flow with compiler support. In *Proceedings of International Conference on Field Programmable Logic and Applications*, Washington, DC, Aug. 31-Sep. 2 2010.
31. M. Siti and M. P. Fitz. A novel soft-output layered orthogonal lattice detector for multiple antenna communications. In *International Conference on Communications*, volume 4, Istanbul, Turkey, June 11-15 2006.
32. Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17, Oct. 2009.
33. V. Zivojnovic, J. Martinez, C. Schlger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, Dallas, TX, Oct. 18-21 1994.
34. EEMBC – The Embedded Microprocessor Benchmark Consortium. Coremark benchmark, Feb. 2016. http://www.eembc.org/coremark.
35. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), May 2008.