SIMD Made Explicit

Luc Waeijen
Eindhoven University of Technology, The Netherlands
l.j.w.waeijen@student.tue.nl

Dongrui She
Eindhoven University of Technology, The Netherlands
d.she@tue.nl

Henk Corporaal
Eindhoven University of Technology, The Netherlands
h.corporaal@tue.nl

Yifan He
Eindhoven University of Technology, The Netherlands
y.he@tue.nl

***Abstract*—Low energy consumption has become one of the most important topics in computing. With single CPUs consuming as much as 115 Watt, engineers have been looking for ways to reduce energy consumption while maintaining high computational performance. Often wide SIMD architectures are used to achieve this, exploiting data parallelism to keep the required clock frequency low for a given compute constraint. In this paper, we propose a wide SIMD architecture with explicit datapath to further optimize energy efficiency without sacrificing computation power. To have a detailed comparison, both the proposed wide SIMD architecture and its transparent bypassing counterpart are implemented in HDL and synthesized with a TSMC 40nm low power library. The power estimation is derived from actual toggle rates generated by post-synthesis simulation. Our experimental results show that with explicit bypassing the overall energy consumption can be reduced up to 44% compared to the corresponding SIMD architecture with transparent bypassing.**

I. Introduction

There is an increasing demand for running applications with high performance requirements on systems that have relatively limited resources. For example, a smart phone has to run high-definition video codecs, wireless signal processing, and 3D graphics processing. A smart camera may combine high resolution video sensing, low-level to high-level vision processing, and communication within a single embedded device. All these applications require a large amount of computation, and yet system designers have to meet these requirements with a very limited energy budget. *Energy efficiency* is thus becoming a dominant determinant in the design, especially for systems that have to run on restricted energy sources like batteries or solar cells.

To provide the required high performance under a limited energy constraint, often a solution is found in wide SIMD (Single Instruction Multiple Data) architectures. In wide SIMDs, a *single* instruction operates on *multiple* data in parallel. This inherently enables SIMDs to exploit data-level parallelism present in an application. Because multiple operations are carried out simultaneously instead of sequentially, the same computational performance can be delivered at a much lower clock frequency, thereby reducing energy consumption [1].

Besides the high-performance characteristic which enables wide-range *Vdd* scaling, another important low-power feature of wide SIMDs is that a significant part of datapath and control path can be shared between multiple processing elements. A wide SIMD typically consists of a control processor (CP) and a large number of processing elements (PEs). Since the PEs execute the same instruction in parallel, the instruction fetch and decode hardware can be shared among all PEs. The energy consumption of the instruction memory, which contributes a significant part to the total energy consumption of a single core processor, is amortized among multiple PEs. Furthermore, as a programs' control flow is handled by the CP, this control cost is also effectively shared. For a wide SIMD with hundreds of PEs, the energy consumed by these shared parts, i.e. instruction fetch, instruction decode, and control flow, is now negligible. Energy consumption is concentrated on the real datapath, resulting in a higher energy efficiency.

As the energy consumption of a wide SIMD is dominated by PEs [1], [2], and the register file (RF) within the PE is one of the major contributors to the PE's energy consumption as will be shown in Section III, reducing the energy consumption of the register file has a large impact on the overall energy consumption of wide SIMD architectures. Because of this, reducing the register file's energy consumption in the context of a wide SIMD processor is of great importance.

Explicit bypassing is a technique that can be used to reduce the energy consumption of a register file. To minimize the penalty of read-after-write hazards in a pipelined architecture, results of preceding instructions can usually be bypassed to later instructions before they are written back to the register file. Traditional architectures handle this bypassing with hardware, which is transparent to the programmer and compiler. To the contrary, explicit bypassing directly controls the bypassing by software, which eliminates the need of hardware logic to determine bypass conditions. Explicit bypassing has the potential to greatly reduce the number of accesses to the RF, resulting a less power-hungry RF [3]–[5].

In this paper we propose a programmable, energy efficient wide SIMD architecture that exploits explicit bypassing. The complementary compiler for this architecture is introduced in [13]. To prove the effectiveness of explicit bypassing in wide SIMDs, the proposed architecture is compared to its transparently bypassed counterpart. Both architectures are implemented in HDL and synthesized with a TSMC 40nm low power library. Energy consumption of 7 different kernels is obtained based on post-synthesis simulation. The experimental results show that the proposed explicitly bypassed SIMD can reduce up to 44% of the total energy consumption compared to the transparently bypassed SIMD.

To further show the advantages of explicit bypassing in a wide SIMD context over a single core architecture, a 1-

978-1-4799-0103-6/13/$31.00 ©2013 IEEE

PE version of both the explicitly and transparently bypassed architecture is benchmarked. By comparing the normalized energy gain of a single PE in the SIMD and single core setup, it is concluded that explicit bypassing in a wide SIMD with 128 PEs on average saves an additional 9.8% of energy compared to a single core setting.

The rest of this paper is organized as follows: Section II introduces the proposed architecture and elaborates on the differences between explicit and transparent bypassing. The experimental setup, energy benchmarks and their results are given in Section III. Section IV lists related work on explicit datapaths and low energy SIMDs. Finally the conclusions and future work are discussed in Sections V and VI respectively.

II. Proposed Wide SIMD Architecture

In this section an energy efficient wide SIMD architecture is introduced. In Section II-A general features of the architecture are treated. Section II-B discusses the inter-PE communication network and Section II-C further details the differences between the explicitly and transparently bypassed SIMDs, used to quantify the effectiveness of explicit bypassing in an SIMD context.

A. Wide SIMD with Control Processor: DLP + ILP

The proposed architecture is a wide SIMD. A wide SIMD contains hundreds of processing elements (PEs) that execute the same instruction on different data. Wide SIMDs inherently exploit data-level parallelism in an energy efficient manner [1], [6], by amortizing instruction fetch, instruction decode and control flow energy cost among all PEs.

The number of PEs in the proposed architecture is parametrized to better fit different target applications. As a typical size, a 128-PE instance is presented and discussed in this paper. An overview of the proposed wide SIMD architecture is shown in Fig. 1. The PE array performs vector operations, exploiting data level parallelism (DLP), while the control processor (CP) performs scalar operations in parallel to the PE array, exploiting instruction level parallelism (ILP). In the proposed architecture, all PEs execute at lock step, resulting in a uniform control flow. This uniform control flow can be handled by the CP at the same time when the PE array is processing vector data. Communication between the PE array and CP is through the circular neighbourhood network, which is described in the next section.

B. Neighbourhood Communication

If all operations on a single element are truly independent of other elements, a PE could compute these operations without exchanging data with another PE. In practice only a very selected number of applications have this property. In image processing for example, filters typically need pixels from a neighbourhood surrounding the target pixel that is operated on. These kind of dependencies make data exchanges between the different PEs required for practical cases. There exist many types of interconnect for wide SIMDs that can realize such communication needs. For example customized

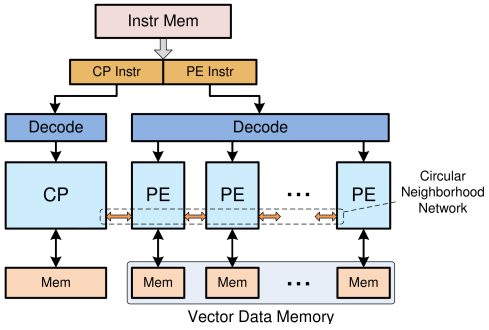


Fig. 1. Proposed wide SIMD architecture

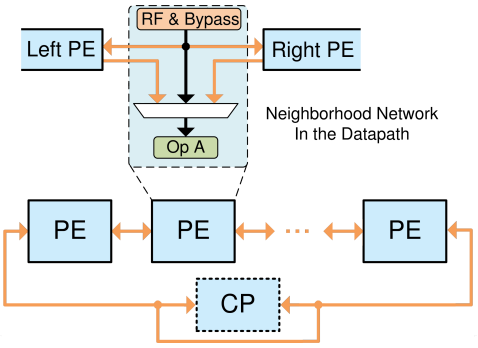


Fig. 2. Circular neighbourhood communication network

crossbars [7], XRAM based swizzle networks [8] and even completely dynamic interconnects [9]. However, in general these interconnects consume a lot of energy. In this SIMD an energy efficient neighbourhood network is used, that enables each PE to communicate with its two direct neighbours. The PEs are logically connected to each other in a circle and each PE can select the operands from its left and right neighbour. This is depicted in Fig. 2

The benefit of the neighbourhood network is that it is very energy efficient per operation compared to the alternatives. If the required communications are local, e.g. a 5x5 window or cross kernel, the performance of this relatively simple network is usually sufficient.

When the communication distances increase, the data has to be passed through all the PEs between the source PE and the destination PE. This can be quite costly in terms of cycles, which is the main reason why we are currently investigating other possibilities for the interconnect. To ensure the energy benchmarks are representable for realistic applications, both kernels with short as well as long communication distances are tested in this paper.

C. Processing Element Datapath

In order to quantify the effects of explicit bypassing compared to transparent bypassing, there are two versions of the datapath. One is transparently bypassed, the other one explicitly. The transparent and explicit datapaths are shown in Fig. 3 and Fig. 4 respectively.

Except from the bypassing, the two datapaths are similar.

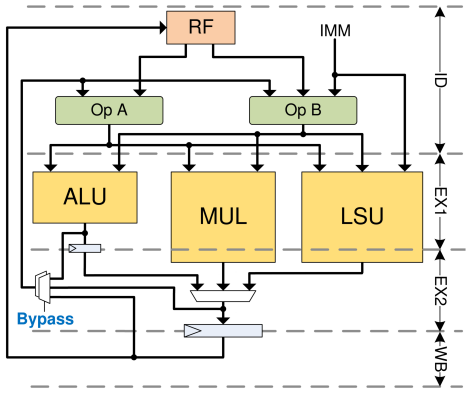


Fig. 3. Datapath with transparent bypass

Although in the proposed design the number of pipeline stages and the datapath width are configurable to match different target applications, in this paper the datapath is fixed to five pipeline stages with a 16-bit datapath as a typical configuration. As can be seen in Fig. 3 and Fig. 4 each PE has a local instruction decode (ID) stage, containing hardware which cannot be shared with other PEs, e.g. the register file. There are two execution stages (EX1 and EX2). The ALU finishes calculations in one cycle and places the result in a EX1/EX2 pipeline register. The multiplier (MUL) and load store unit (LSU) both utilize the two stages to produce a valid result.

Each functional unit in EX1 has private input registers that only get updated when the functional unit needs to be used. For example, the operands of an *add* instruction will not be written to the input registers of the multiplication unit. This prevents unnecessary toggling of the multiplier logic, reducing energy consumption.

The way the bypassing is implemented is the only difference between the two datapaths. The remaining part of this section elaborates the specific features of the transparent and explicit datapaths.

1) Transparent Datapath: The transparently bypassed datapath is shown in Fig. 3. As visualized in the figure, there exist three bypass sources. The first one is directly after the ALU in stage EX1. Next there is the result of EX2 which is provided by the output of the multiplexer at the end of that stage. Finally, the result that is located in the write back stage and still has to be written to the register file, can also be bypassed.

Whether a bypass is required or not is completely decided in hardware. To do this the hardware keeps track of the destination address of the data in the pipeline. When an operation has an operand address that matches one of the destination addresses, the pipeline register containing the result is automatically selected as the operand instead of the register file.

In the proposed architecture, the bypass requirements are checked in parallel with the register file read. Without adding an extra pipeline stage that first tests if a bypass is to be performed, the register file needs to be read speculatively. Consider the assembly code given in Code 1. Here instruction 1 consumes the result produced by instruction 0. Because r1 is not yet written back to the register file when instruction 1 executes, a bypass from the ALU output back to its input is required. The hardware figures this out at the same time the register file is being read. This results in r1 being read from in the register file, even though it is not being used. This read is not necessary for the computation, but the bypassing hardware cannot avoid it.

```
0: add r1, src1, src2
1: add dest, r1, src2
```

Code 1. Example code where bypassing is required for operation 1

A similar situation occurs when after instruction 1 in Code 1, r1 is never referenced again. In theory r1 does not need to be written to the register file. However, the hardware has no way to determine whether r1 will be referenced in the future or not. Therefore it is always speculatively written to the register file.

The two previous situations, unnecessary reads and writes of the register file, are inherent to transparent bypassing. In the next section the explicitly bypassed datapath is introduced which can avoid both these situations.

2) Explicit Datapath: The explicit datapath is very similar to the transparent datapath. The main difference is that in the explicit datapath the bypass sources are directly addressable by the instructions. This is accomplished by reserving a part of the register file address space for the relevant pipeline registers. In the proposed datapath 5 registers are removed from the register file in order to free the required number of bypass addresses. Usually less registers increases the register file pressure. In this case the loss of registers has only a limited impact, as with explicit bypassing short lived values do not need to allocate a register in the RF.

Because the bypass sources have to be addressed from the instruction, it is the responsibility of the compiler to handle bypassing in software. To do this, the compiler needs to have knowledge about the datapath, i.e. the datapath has to be *explicitly* known to the compiler. This in contrast to transparent bypassing, where the compiler does not need to know anything about the datapath organisation, i.e. the datapath is *transparent* to the compiler.

Consider again Code 1. In the case of explicit bypassing the compiler must detect that in instruction 1 the operand needs to be bypassed from the output of the ALU. The assembly in Code 1 would then be transformed into Code 2, where ALU1 is the bypass source directly after the ALU.

```
0: add r1, src1, src2
1: add dest, ALU1, src2
```

Code 2. Instruction 1 has an explicitly bypassed operand such that a speculative read from the register file can be avoided

Because the instruction specifies directly whether a bypass is required or not, the speculative read from the register file

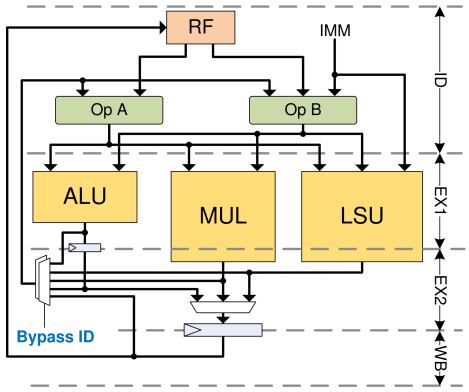


Fig. 4. Datapath with explicit bypass

can be avoided. In case the operand address is one of the special bypass source addresses, the read of the register file is disabled. This opposed to the transparently bypassed datapath, which always needs to read.

The explicit datapath also avoids unnecessary writes. Consider the situation that the result produced by instruction 0 in Code 2 is only referenced by instruction 1. In the transparently bypassed datapath the hardware cannot determine the liveliness of a variable. In the explicitly bypassed version, the compiler has this information readily available. This can be exploited by encoding in the instruction that the result of instruction 0 does not need to be written to the register file. See Code 3, where the result of instruction 0 is not written to the register file. In practice this is done by encoding destination address r0, which statically always reads 0. The hardware then avoids wasting energy by disabling the write to the register file.

```
0: add --, src1, src2
1: add dest, ALU1, src2
```

Code 3. Instruction 0 specifies that the result need not to be written to the register file

It is clear that explicit bypassing can save energy by avoiding these unnecessary reads and writes to the register file. Yet by expanding the number of available bypass sources, explicit bypassing provides even more opportunities to bypass and thus save energy. The explicit datapath drawn in Fig. 4 has 2 extra sources compared to the transparent datapath given in Fig. 3. These additional bypass sources increase the chance of a result being bypassed instead of being written back to the register file.

The chance of a result being bypassed instead of written to the RF increases because each unit (ALU, MUL and LSU) has private input registers, which keep the results at the output of a compute unit valid as long as no new operation/input is assigned to it. The effect of this is that the window for bypassing a result is increased, which the explicit datapath can exploit.

An example of how the explicit datapath saves register file accesses with the additional bypass sources is given in Code 4 to 6. Code 4 lists the original transparently bypassed code. Code 5 lists the same code, but for an explicit datapath with the same number of bypass sources exposed as the transparent datapath, see Fig. 3. Here EX1 is the result of the ALU and EX2 is the result of the second execution pipeline stage, which either holds the result of the ALU, the multiplier or the load store unit. Finally Code 6 lists the case where the ALU and multiplier (MUL) outputs are exposed as bypass sources, just as in Fig. 4. As can be seen, the additional bypass source saves a write, a read and a register allocation.

```
0: add r1, src1, src2
1: mul r2, r1, src2
2: add dest, src1, src2
3: add r3, r2, r1
```

AssemblyCode 4. Transparently bypassed code where the ALU and multiplier consume each others results

```
0: add r1, src1, src2
1: mul --, EX1, src2
2: add dest, src1, src2
3: add r3, EX2, r1
```

AssemblyCode 5. Explicit code where only the same bypass sources as in the transparent datapath are exposed

```
0: add --, src1, src2
1: mul --, ALU1, src2
2: add dest, sr1, src2
3: add r3, MUL, ALU2
```

AssemblyCode 6. Explicit code that handles bypassing when the ALU and multiplier are exposed bypass sources

The explicitly bypassed datapath has these extra bypass sources while the transparently bypassed version does not, because there is no gain in having them in the transparent case. In the explicit datapath an extra bypass source can avoid a read or write to the register file. In the transparent datapath reads and writes are always performed speculatively, thus extra bypass sources provided no energy or performance gain and would only complicate the bypassing logic.

III. Measurements and Analysis

The goal of the experiments is to quantify effects of explicit bypassing versus transparent bypassing on energy consumption and compare the effectiveness of explicit bypassing in a wide SIMD and a single core environment. This section contains a description of the experimental setup and used benchmarks, presentation of obtained results and a detailed analysis of those results. Section III-A treats the experimental setup, describing the tools and techniques used. Section III-B describes the various kernels that were tested, providing example code to illustrate the difference in code for the transparent and explicit SIMD. The results are presented in Section III-C and finally analysed in Section III-D.

TABLE I
CONFIGURATION OF THE TARGET ARCHITECTURE

Data width	16 bits
Pipeline stages	5
Number of PEs	128
Instruction memory	56b × 1k
PE data memory	16b × 1k

TABLE II
ENERGY CONSUMPTION OF ACCESSING DIFFERENT MEMORIES

	16b × 1kB	56b × 1kB
Access Energy (pJ)	0.7564	0.9185

TABLE III
KERNEL DESCRIPTION

Kernel	Description
FIR5	5-tap FIR filter on a 1D data stream
Erosion	Logic AND in a 3x3 cross
Binarization	Clipping according to a threshold
Rotate	Square matrix rotation
Mirror	Image mirroring in the vertical axis
Transpose	Matrix transpose
Conv3x3	Convolution with a 3x3 window

A. Experimental Setup

Both the transparent and explicit datapath SIMDs presented in Section II are implemented in HDL. The configurable architecture allows for automated HDL code generation of various variations of the SIMD. The configuration used in the experiments is given in Table I.

Each SIMD is synthesized with a 40nm TSMC low power library and targeted at 100MHz. Post-synthesis simulation is used to obtain actual toggle rates of the synthesized hardware and power reports are constructed from these simulation results. The tool used to perform the synthesis, simulation and power reporting is Cadence Encounter®RTL Compiler Version v11.20.

All the hardware of the processors is synthesized, with exception of data and instruction memories. This is done because these memories need a specialized memory synthesis tool. Instead of synthesis, the CACTI tool [10] is used to obtain energy per access numbers. The obtained access energies are listed in Table II.

B. Benchmarks

To get a good overview of the effects of explicit bypassing versus transparent bypassing, in total 7 different kernels were tested. The names of the tested kernels and a short description are given in Table III. Included are kernels from the image processing domain and matrix calculus. The rotate, mirror and transpose kernels are focussed on long distance communication, while the other kernels use short/local communication and are more compute intensive.

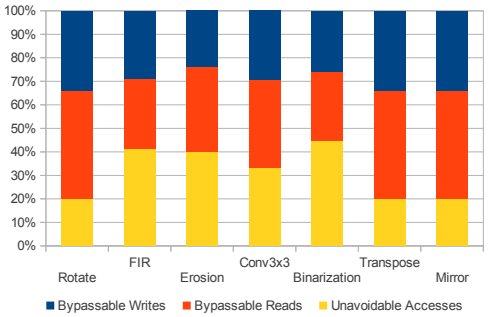


Fig. 5. Register file access breakdown for the tested kernels. The bypassable reads and writes are the register file accesses that can be avoided with explicit bypassing.

All kernels are tested on 128x128 datasets. In the case of the FIR filter and convolution kernel all coefficients are assumed to be stored in the register file and not hard-compiled into the code.

Code 7 and Code 8 show a snippet of the code for the transparently and explicitly bypassed erosion kernel respectively. Whenever an operand is prefixed with either 'l.' or 'r.', that operand is fetched from either the left or the right neighbour PE.

```
lw   r3 , r6 ,    2
and  r4 , r1 ,    r2
and  r4 , r4 ,    r3
and  r4 , l.r2 ,  r4
and  r4 , r.r2 ,  r4
sw   r6 , r4 ,    1
```

AssemblyCode 7. Code snippet from the transparently bypassed erosion kernel

```
lw   r3 , ALU1, 2
and  --, r1 ,   r2
and  --, ALU1,  LSU
and  --, l.r2 , ALU1
and  --, r.r2 , ALU1
sw   r0 , ALU1, 1
```

AssemblyCode 8. Code snippet from the explicitly bypassed erosion kernel

Comparing Code 7 and Code 8 it can be observed that the explicit version avoids 4 out of 5 writes and 6 out of 11 reads. The effects of this will translate into energy savings for the explicitly bypassed version.

An overview of the percentage of accesses that are avoided by the explicit code per kernel is given in Fig. 5. Because the frequency of register file accesses varies heavily per kernel, the average number of accesses per cycle is used as a metric for register file activity per kernel. The difference in accesses per kernel with transparent and explicit bypassing is used to estimate the potential energy saving. The accesses per cycle numbers are listed in Table IV.

In terms of performance the explicit and transparent codes are almost equivalent. In some cases the explicit code needs

TABLE IV
NUMBER OF RF ACCESSES PER CYCLE (READS AND WRITES)

Kernel	Accesses per Cycle		improvement
	Transparent	Explicit	factor
FIR5	2.81	1.17	2.40×
Erosion	2.58	1.05	2.46×
Binarization	1.74	0.78	2.23×
Rotate	1.92	0.38	5.05×
Mirror	1.93	0.38	5.08×
Transpose	1.94	0.38	5.11×
Conv3x3	2.83	0.94	3.01×

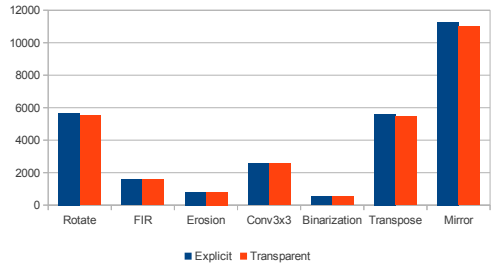


Fig. 6. Cycle count per kernel. Only the counts of the transpose, mirror and rotate kernels differ between the explicit and transparent SIMDs

a couple of instructions more, to fill the bypass registers with correct data when entering a loop. An example of such a joint-point issue can be seen in Code 8. The load instruction at the top uses bypass source ALU1. In the full code, this snippet of code is part of the prologue of a loop body. Before entering the loop the bypass source ALU1 must be initialized correctly. In some cases this can require one or two extra cycles compared to transparent bypassing, which always can address the registers directly as if there are no pipeline stages. This effect is not profound, as can be observed in Fig. 6 which lists the cycle counts of each kernel for both the explicit and transparent cases.

To investigate the effectiveness of explicit bypassing in a wide SIMD context compared to a single core architecture, a second series of tests is conducted. Instead of 128 PEs, both the explicit and transparent architectures are synthesized with just one PE. The goal is to compare the relative energy saving in the PE-array. Expected is that the 128 PE version will save more energy compared to the one PE version, because the energy of the instruction fetch and decode are amortized over the 128 PEs. In other words, in the PE-Array with 128 PEs the register files contribute a larger percentage to the energy consumption than in the one PE configuration. Saving energy in the register file by explicit bypassing is therefore deemed to be more effective in the 128-PE/SIMD configuration. To ensure a fair comparison, the exact same code used for the 128 PEs is also executed on the one PE configuration. This ensures that the energy saving in the register file accomplished by explicit bypassing is the same for both cases.

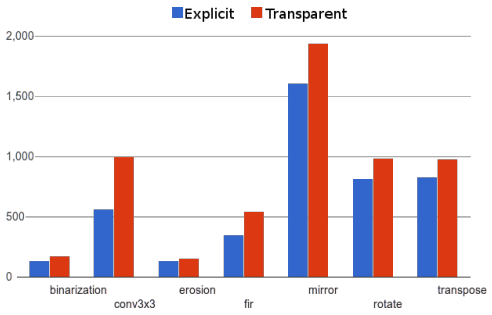
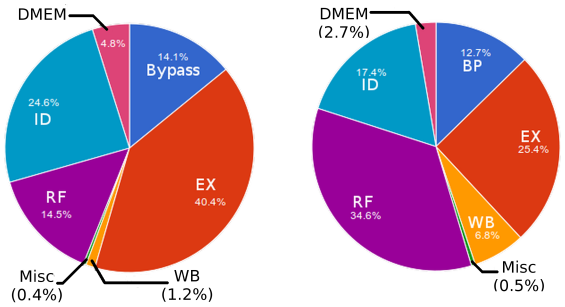


Fig. 7. Total energy consumption per kernel (nJ)

TABLE V
ENERGY SAVING PER KERNEL

Kernel	Saving RF	Saving Total
FIR5	66.2%	35.6%
Erosion	33.1%	14.4%
Binarization	56.0%	22.7%
Rotate	36.8%	17.2%
Mirror	37.3%	17.1%
Transpose	35.3%	15.6%
Conv3x3	76.7%	43.6%



(a) Explicit PE (1.56 pJ) (b) Transparent PE (2.81 pJ)

Fig. 8. Breakdown of the energy consumption per cycle in a PE for the conv3x3 kernel

C. Results

Fig. 7 shows the energy consumption per kernel of both the explicit and transparent SIMDs. The percentage of overall energy saved is listed in Table V, together with the savings in the register file alone. The maximum overall saving is 43.6%, with an average of 23.7%. The maximum energy saved in the register file is 76.7% with an average of 48.8%.

Fig. 8 shows the energy breakdown inside a PE for the conv3x3 kernel. In this figure, the write back stage consists only of the pipeline registers from EX/WB and the ID includes the ID/EX pipeline registers. For a more complete overview of the energy consumption, Fig. 9 shows the absolute energy consumption inside a PE for the conv3x3 kernel. The energy numbers are in nJ, and they are the sum of all 128 PEs. In this way, any small variations between the PEs are averaged. The energy consumption for a single PE can be obtained by dividing the number in Fig. 9 by 128.

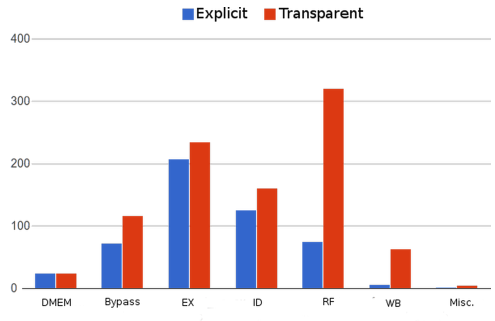


Fig. 9. Absolute energy breakdown in nJ for the conv3x3 kernel (energy is cumulative of all 128 PEs in the SIMD)

TABLE VI
ENERGY SAVING OF EXPLICIT BYPASSING COMPARED TO TRANSPARENT BYPASSING IN THE PE-ARRAY

Kernel	1 PE	128 PEs	Difference
FIR5	23.8%	36.3%	12.5%
Erosion	12.4%	15.9%	3.5%
Binarization	13.6%	23.8%	10.2%
Rotate	7.8%	18.5%	10.7%
Mirror	7.7%	18.6%	10.9%
Transpose	7.2%	17.0%	9.8%
Conv3x3	33.1%	44.2%	11.1%

To compare the effectiveness of explicit bypassing in a wide SIMD context and a single core architecture, the same code as for the 128 PE SIMDs is benchmarked on a one PE version of the transparently and explicitly bypassed architectures. For a fair comparison, only the energy savings in the PE-array and instruction memory of the PEs is taken into account (leaving out the CP). The PE-Array energy savings of explicit bypassing compared to transparent bypassing for the 128 PE and 1 PE configurations are given in Table VI.

D. Analysis

It is clear from Fig. 7 and Table V that explicit bypassing can lead to significant reduction of the overall energy consumption. Even though it follows from Fig. 6 that the explicitly bypassed SIMD sometimes needs a few cycles more, the energy savings outweigh these extra cycles by far. In fact the average reduction of the overall energy consumption is 23.7% with an average loss in performance of less than 1%.

There are some notable observations that can be made from these results. Firstly one would expect a strong correlation between the improvement in register file accesses per cycle in Table IV and the energy savings in Table V. Observing in Table IV that the rotate, mirror and transpose kernels have the highest improvement factor, it would be expected that these kernels have the highest energy savings. However these kernels reduce the total energy by $15.7 - 17.2\%$ which is less than for example the FIR filter which saves 35.5% while its improvement factor in Table IV is twice as low.

This somewhat unexpected result can be explained by the sequence of registers that are accessed in the register file. The rotate, transpose and mirror kernels perform long distance communication by repeating the instruction in Code 9. The explicit datapath can avoid both the read and the write in this instruction. Because this instruction is evaluated often in these kernels, the explicit version saves a lot of register file accesses. Yet in the transparent datapath the register that is speculatively being read and written is always the same (r1). Because of this the word lines do not need to toggle and these speculative reads and writes are relatively energy efficient. This reduces the overall gain of the explicit datapath for these kernels.

```
0: add r1, l.r1, 0
```

AssemblyCode 9. Transparent code for communication

The second observation is that in the explicitly bypassed SIMD, not only the register file consumes less energy, but also other hardware has become more efficient. This can be calculated from Table V. For example the conv3x3 saves 76.7% of energy in the register file and the overall savings are 43.6%. This suggests that the register file in the transparent situation would contribute $\frac{43.6}{76.7} * 100\% = 56.8\%$ to the overall energy consumption. However from Fig. 8b it is clear that the register file only contributes 34.6% of the energy consumption of the PE alone. Therefore it must be that other parts of the hardware also have become more energy efficient.

This effect can be explained by three main reasons:

1) The RF write address is often zero in the explicit datapath, which results less toggles in pipeline registers that carry this address from the ID to the WB stage. Also the word select lines of the RF are toggled less.
2) Results are not written into the write back stage if no RF write is needed, this saves toggling in the EX/WB pipeline register and in the WB stage itself.
3) Speculative RF reads are avoided in the explicit datapath, which avoids unnecessary toggling in the instruction decode stage.

These effects can be seen in Fig. 9 where almost every section of the explicit datapath is more energy efficient than in the transparent datapath.

Lastly, examining the results in Table VI, it can be concluded that explicit bypassing in a wide SIMD context with 128 PEs is more effective than explicit bypassing in a single core architecture. Looking only at the energy saved in the PE-Array, thus excluding the CP, the 128 PE SIMD on average saves 9.8% more energy than the single PE configuration. This is because the instruction fetch and instruction decode costs are amortized over all the PEs in the array. As a result the register files contribute a larger percentage to the energy consumption of the PE-Array than the single register file in the 1 PE configuration. Energy saving in the register file is therefore more effective in the SIMD context.

IV. RELATED WORK

Earlier work has shown that in a conventional configuration the register file in a single core machine consumes between 10 and 30 percent of the total energy and that bypassing

can reduce this large consumption [4]. For example in Very Long Instruction Word processors (VLIWs) it has been shown that storing short lived values in pipeline registers can reduce energy while sustaining the compute performance [5]. Similar in a Transport Triggered Architecture (TTA), the savings induced by explicit bypassing have been shown to reduce as much as 80% of the register file energy consumption, leading to an overall energy reduction of 11% [3]. A compiler that was developed for this TTA, shows that it is possible to fully automate explicit bypassing and achieve the same energy savings [11], which proves the practical value of explicit bypassing. The efforts made in the development of the Xetal-Pro [2] also show how important the energy consumption of memory is in an SIMD setting. Furthermore the benefits of explicit bypassing for realistic Mobile Signal Processing algorithms, with explicit bypassing applied in an SIMD architecture have also been proven [12].

Yet none of the related works provide a detailed head-to-head energy comparison of explicit versus transparent bypassing in an SIMD setting. In this paper power estimation guided by post-synthesis simulation is used to obtain energy numbers for both explicit and transparent bypassing in an SIMD architecture. The comparison of transparent and explicit datapaths shows that not only the register file becomes more energy efficient due to less accesses, but also other hardware saves energy by toggling less. Furthermore this work shows the advantages of explicit bypassing in a wide SIMD context over explicit bypassing in a single core architecture. None of this has been shown in any of the related works.

V. Conclusions

In this paper, we proposed a low-energy, wide SIMD architecture with explicit datapath. This architecture and its transparently bypassed counterpart were implemented in HDL. Energy consumption of both architectures was measured using post-synthesis simulation of 7 different kernels. The advantages of explicit bypassing in a wide SIMD over explicit bypassing in a single core architecture are examined by comparing a 128 PE SIMD with a 1 PE configuration. Detailed analysis of the experimental results shows that Explicit bypassing in an SIMD context saves on average 23.7% of the total energy consumption compared to transparent bypassing. For a single kernel the overall saving was shown to be up to 43.6%. The energy savings of explicit bypassing in an SIMD context with 128 PEs are shown to be on average 9.8% higher than the savings of explicit bypassing in single core processors, proving the importance of reducing the energy consumption of the register file in an SIMD context. Furthermore it is shown that explicit bypassing does not only save energy by reducing the number of accesses to the register file, but as a secondary effect the remaining hardware also is toggled less and becomes more energy efficient as well.

VI. Future Work

An interesting topic for future work is low-power interconnects in SIMDs to improve performance of kernels that need communication over long distances. If the reduction in running time is large enough, more energy will be saved than is spent on any additional hardware needed for the network. Communication dominated kernels such as mirror, transpose and rotate are expected to benefit greatly from such a kind of interconnect.

Other energy saving extensions of the proposed architecture are currently also being researched. Predication of instructions and a broadcast possibility from the CP to the PEs are a couple of the features that are being considered.

Acknowledgments

This work is supported by the Ministry of Economic Affairs of the Netherlands, project EVA PID07121, and the Dutch Technology Foundation STW, project NEST 10346.

References

[1] Y. He, Y. Pu, Z. Ye, S. Londono, R. Kleihorst, A. Abbo, and H. Corporaal, "Xetal-pro: An ultra-low energy and high throughput simd processor," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, june 2010, pp. 543 –548.

[2] Y. Pu, Y. He, Z. Ye, S. Londono, A. Abbo, R. Kleihorst, and H. Corporaal, "From xetal-ii to xetal-pro: On the road toward an ultralow-energy and high-throughput simd processor," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 21, no. 4, pp. 472 –484, april 2011.

[3] Y. He, D. She, B. Mesman, and H. Corporaal, "Move-pro: A low power and high code density tta architecture," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, july 2011, pp. 294 –301.

[4] X. Guan and Y. Fei, "Reducing power consumption of embedded processors through register file partitioning and compiler support," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, 2008, pp. 269–274.

[5] N. Goel, A. Kumar, and P. Panda, "Power reduction in vliw processor with compiler driven bypass network," in *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, jan. 2007, pp. 233 –238.

[6] A. Abbo, R. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, B. Vermeulen, and M. Heijligers, "Xetal-ii: A 107 gops, 600 mw massively parallel processor for video scene analysis," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 192 –201, jan. 2008.

[7] P. Raghavan, S. Munaga, E. Ramos, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest, "A customized cross-bar for data-shuffling in domain-specific simd processors," in *Architecture of Computing Systems - ARCS 2007*, ser. Lecture Notes in Computer Science, P. Lukowicz, L. Thiele, and G. Tröster, Eds. Springer Berlin / Heidelberg, 2007, vol. 4415, pp. 57–68. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71270-1_5

[8] S. Satpathy, Z. Foo, B. Giridhar, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw, "A 1.07 tbit/s 128x128 swizzle network for simd processors," in *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, june 2010, pp. 81 –82.

[9] R. Frijns, H. Fatemi, B. Mesman, and H. Corporaal, "Dc-simd : Dynamic communication for simd processors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April, pp. 1–10.

[10] CACTI, "cacti 5.3, rev 174," http://quid.hpl.hp.com:9081/cacti/.

[11] D. She, Y. He, B. Mesman, and H. Corporaal, "Scheduling for register file energy minimization in explicit datapath architectures," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 388 –393.

[12] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Anysp: Anytime anywhere anyway signal processing," *Micro, IEEE*, vol. 30, no. 1, pp. 81 –91, jan.-feb. 2010.

[13] D. She et al., "OpenCL code generation for low energy wide SIMD architectures with explicit datapath," in *Proceedings of the 13th International Conference on Embedded Computer Systems (SAMOS-XIII)*, 2013.