SIMDify: Framework for SIMD-Processing with RISC-V Scalar Instruction Set

Mehmet Alp Şarkışla
Institute of Graduate Studies in Science and Engineering
Boğaziçi University
İstanbul, Turkey
Informatics and Information Security Research Center,
TUBİTAK
Kocaeli, Turkey
alp.sarkisla@boun.edu.tr

Arda Yurdakul
Institute of Graduate Studies in Science and Engineering
Boğaziçi University
İstanbul, Turkey
yurdakul@boun.edu.tr

ABSTRACT

In this work, we propose a parallel programming framework, SIMDify, which generates single-instruction-multiple-data (SIMD) processors that can achieve SIMD processing without using SIMD instructions. SIMDify takes an application machine code compiled for scalar RISC-V ISA and simulates it to determine the SIMD processing regions. Then, SIMDify configures and generates the application-specific SIMD processor that executes scalar RISC-V instructions concurrently on the SIMD datapath. SIMD processor consists of a single master and multiple slave processing elements (PE). Slaves focus on SIMD level tasks, whereas the master is responsible for the central control. Proposed architecture is the first SIMD capable RISC-V processor designed in HLS and can operate with a faster clock frequency than the existing SISD RISC-V HLS cores. SIMDify relieves the user from using custom instructions with rigid programming models and offers a flexible solution. The processor is designed and tested in Vivado High Level Synthesis 19.2. It operates at 78 MHz on Zynq Zedboard FPGA. Master PE uses 5% and each slave uses 3.5% of FPGA resources. Test results show that execution time can be improved by 8.5x with 9 slaves and 19x with 29 slaves.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; **High-level language architectures**.

KEYWORDS

SIMD, ASIP, RISCV, FPGA, HLS, Parallel Processing

ACM Reference Format:
Mehmet Alp Şarkışla and Arda Yurdakul. 2021. SIMDify: Framework for SIMD-Processing with RISC-V Scalar Instruction Set. In *Australasian Computer Science Week Multiconference (ACSW '21), February 1–5, 2021, Dunedin, New Zealand.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3437378.3444364

ACSW '21, February 1–5, 2021, Dunedin, New Zealand

ACM ISBN 978-1-4503-8956-3/21/02…$15.00
https://doi.org/10.1145/3437378.3444364

1 INTRODUCTION

In recent years the main attention has been optimizing general-purpose processors for a given application domain to make them more efficient [28]. With the adoption of application specific instruction set processors (ASIP), area-efficient [14, 40], power-efficient [22, 24, 25, 35] and performance-efficient [13, 14, 20, 33, 40] processors have been designed. One frequent way to enhance performance-efficiency of special purpose processors is to exploit inherent data parallelism in the algorithms and execute them simultaneously, which is called SIMD (Single instruction, multiple data) processing.

For hardware, SIMD instructions are inherently simple to implement, since they only require the duplicated structure of the main execution unit's datapath. But not all processors have built-in instructions for SIMD processing. Traditional approaches to this problem are solved by extending standard instruction set with non-standard custom instructions (compiler retargeting) [26] or using Just-in-time (JIT) compilers [9], both of which requires a non-standard compiler as well as non-standard instructions in the custom hardware. Since custom instructions are not standardized, each individual accelerator requires a different compiler modification. On top of the compiler modification, to properly introduce new instructions, simulators and debuggers must be additionally retargeted.

Accelerators are used in various fields such as machine learning [38], speech recognition [16], raw data processing [12], cryptography [23] and image detection and recognition [27] especially after the rise of IoT. Though designed accelerators may extremely speed up the execution, using them with custom instructions and compilers is a tedious process that discourages software programmers from using these accelerators [18]. SIMDify offers a flexible parallel processing solution that reduces the user burden and removes the custom instructions.

In this work, we present SIMDify [4], an open-source hardware-software parallelization framework to design special purpose SIMD processors without using any just-in-time compilation, extending the default instruction set or retargeting the compiler. SIMDify takes an application machine code compiled for scalar core and SIMD parameters, and generates a customization header files. Using Vivado High-Level Synthesis (HLS) [5], SIMDify processes the generated header files and automatically synthesizes the desired SIMD capable special purpose processor architecture. Processor is compatible with the RISC-V Instruction Set Architecture (ISA), and

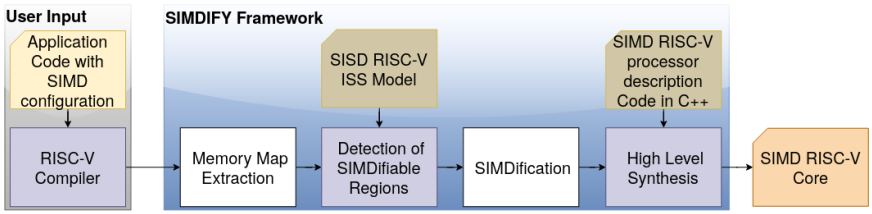

Figure 1: Block diagram of SIMDify Framework.

executes the native instruction set even during SIMD execution. The main contributions of this paper can be summarized as follows:

- A flexible parallel programming framework called SIMDify for generating, customizing and scaling SIMD capable processors with minimal software level modification and using a standard compiler. To generate the special purpose processor, user only requires to write an algorithm in C, and compile it in RISC-V compiler. SIMDify will generate custom header files for the HLS, then synthesizes the SIMD soft processor that accelerates the given algorithm. Generated processor then can be mapped to an FPGA.
- A new RISC-V soft processor architecture that enables in-memory SIMD processing is proposed. Generated processor is the first SIMD capable RISC-V core designed using HLS. Processor can achieve similar frequency with other HLS generated RISC-V cores even with 30 slaves, and it can execute applications as SIMD by using only the base RISC-V ISA without modifying the existing compiler.

Applicability of the SIMDify is tested on selected algorithms. Clock speed, area and performance-efficiency of the generated soft-processors are studied for Zynq Zedboard FPGA.

The rest of the paper is organized as follows. In sections 2 and 3 proposed hardware-software system is introduced. Section 4 provides an overview of SIMD processing, RISC-V, and HLS related works, and in section 5, detailed experimental analysis on resource usage and performance is given. Finally, section 6 concludes the paper.

2 SIMDIFY FRAMEWORK

SIMDify can parallelize and accelerate an application with minimal software level modification and using the standard RISC-V compiler. It utilizes HLS pragmas and C like header structure of the HLS. Using HLS, SIMDify processes the RISC-V compiler machine code and HLS simulator outputs and automatically generates desired SIMD processor architecture. SIMDify is fully automated and it requires only 4 variables to configure the software, which reduces the design time.

Operation of SIMDify framework is shown in Fig. 1. It takes the compiled machine code that contains the algorithm and necessary configuration parameters. The machine code is fed to the Memory Map Extraction block to generate the Local Memory header file. Then, the Local Memory header and the SISD (Single-instruction-single-data) RISC-V ISS (Instruction Set Simulation) Model is fed to the next block to detect the regions that will be executed as SIMD (SIMDifiable Regions) and write them to the Address Header. After that, the SIMDification block generates the SIMD header file using the Address and Local Memory Header. Lastly, SIMD RISC-V processor description code in C++ and all header files are synthesized in Vivado HLS to generate SIMD RISC-V Core. All steps are automated inside the SIMDify Framework. A detailed explanation for each block in the figure is given in the rest of this section.

2.1 Application Code with SIMD configuration

Four variables must be included in the C code to generate and configure the SIMD processor successfully:

- StartPar: Determines the region which SIMD processing will be executed. The user has to set StartPar to 1 just before the loop begins and to 0 just after the loop ends.
- par_num: Unroll factor. Determines the number of SIMD processes. Number must exactly divide the loop count. Denoted by n.
- arr_str: Start local data memory address of the SIMD array. Used in SIMD slaves. Equals to &SMA[0]; where SIMD_memory_array (SMA) is the name of the array accessed in the SIMD loop with size X. Denoted by $A_{data,start}$.
- arr_end: Last local data memory address of the SIMD array. Used in SIMD slaves. Equals to &SMA[X-1] + (&SMA[X-1] - &SMA[X-2]); where SIMD_memory_array (SMA) is the name of the array accessed in the SIMD loop with a size X. Denoted by $A_{data,end}$.

In the local memory, variables have specific addresses which are generated using the "section" command. This command is a GCC variable attribute which is used for setting particular variables to appear in individual sections (address ranges). Only the unroll factor can be modified after compilation. To change the other three, code must be re-compiled. Section names and addresses are determined from the linker file.

Our SIMD processor template processing system consists of one master processing element (PE) and $n-1$ slave PEs. Master can access the complete local memory and executes the sequential code. During SIMD execution, master also executes concurrently with the slaves. So, during SIMD processing, n PEs execute concurrently. In order to fully benefit from SIMD operation, memory access range of each PE has to be contiguous as shown in Fig. 2. To achieve this, the user must write the SIMD loop part of the C code while considering memory adjacency. For example, consider a four iteration loop for matrix vector multiplication $A[i][0..4] \cdot v[0..4] = r[i]$ where A is the name of the 4×5 matrix, v is the multiplied vector with size 5 and r is the result vector, Fig. 3.a. In i-th iteration each element in

i-th row of A is multiplied with elements in the vector and summed up.

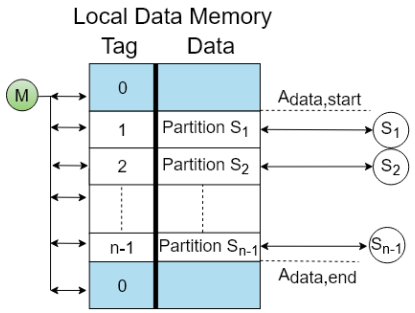

Figure 2: Accessible regions in the Local data memory for n-1 Slave PEs and the Master PE. Different Tag values are generated for each partition.

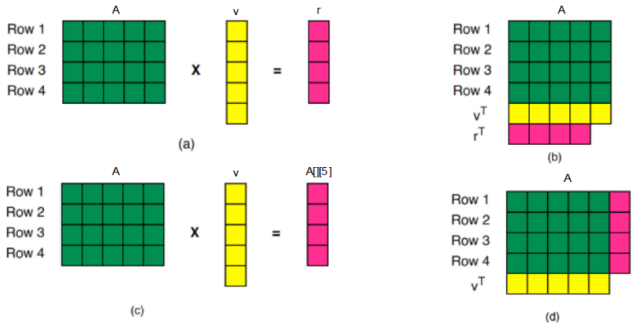

Figure 3: Coding example (a) Vector matrix multiplication (b) Default memory allocation of arrays after declaration. (c) Suggested SIMDifiable vector matrix multiplication (d) SIMDifiable memory allocation after suggestion

The example code results in one matrix, one vector, and one result block in the memory, Fig. 3.b. To design a SIMDifiable C code, all addresses accessed in only one iteration in the SIMD loop, i-th row of A and r, must be adjacent in the memory. So, code shown in Fig. 3.a, cannot be executed in our designed SIMD processor. We solve this problem by adding another column to the matrix to store the result vector by modifying the multiplication as $A[i][0..4] \cdot v[0..4] = A[i][5]$, Fig. 3.c. In this way, all arrays that are read and written in one iteration are compiled as adjacent memory partitions 3.d. Hence, each SIMD slave S_i will be able to execute in its own dedicated partition $Partition\ S_i$, as shown in Fig. 2. Note that the master M can access the entire local memory.

Local data memory in Fig. 2 consists of data and tag fields. Tag field is used for local data memory access and the data field stores the local data. It is a single block that contains address-to-partition mapping. Tag field makes a trade-off between memory access latency and area. By using tag field, area is increased. In return, the SIMD architecture does not require many comparison and multiplexer blocks, which increase the latency of the address-to-partition mapping process.

Size of the tag field is proportional to the size of the data field, and each tag contains values from 0 to $n-1$. 0 value is for regions that are only accessed by the master, and 1 to $n-1$ is for slave regions S_1 through S_{n-1}. How tag field is used for memory access is detailed in Section 3.1.

C code of the example matrix multiplication structure should be written as Fig. 4. Variables that are not accessed in only one iteration in the parallelized loop need not be adjacent in the memory. So, user only has to modify its SIMD execution loop and include the four variables. Rest of the code remains the same.

As another example, we can present matrix matrix multiplication (MMM). Fig. 5 shows MMM for loop, $A[i][0..1] \cdot B[0..1][0..2] = A[i][2..4]$, is SIMDified and each PE executes $A[i][0..1] \cdot B[0..1][x] = A[i][x+2]$ operation. In Fig. 5.d, B is the common memory and can be accessed by all slaves and A is the partitioned memory and can only be accessed by single slave. Master can access to the entire local memory.

```
int X=4;  // size of the iteration and the SIMD array
// three variables must set before SIMD loop:
par_num=4; //can be 1,2, and 4
arr_str= &A[0];
arr_end=&A[X-1] + (&A[X-1] - &A[X-2]);
//startPar must be set before and after the SIMD loop.
startPar=1;
SIMD_loop:for(int i = 0; i < X; i++){
    for (int j = 0; j < 5; j ++) {
        A[i][5] = A[i][5] + A[i][j] * v[j];
    }}
startPar=0;
```

Figure 4: SIMD loop of the matrix multiplication example C code.

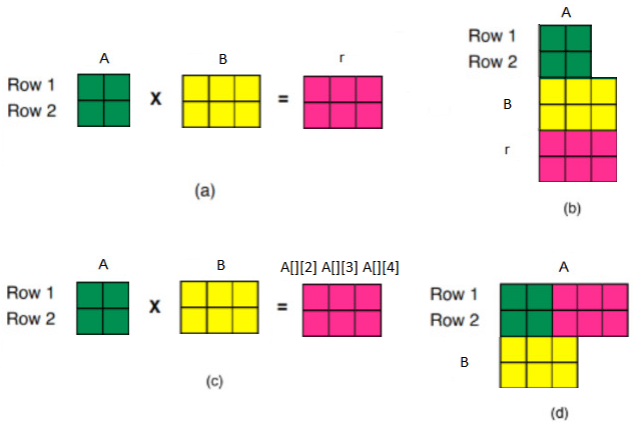

Figure 5: Coding example (a) Matrix matrix multiplication (b) Default memory allocation of arrays after declaration. (c) Suggested SIMDifiable matrix matrix multiplication (d) SIMDifiable memory allocation after suggestion

2.2 RISC-V-compiler

In this work standard RISC-V compiler such as riscv32-unknown-elf, riscv-none-embed, riscv64-unknown-elf GCC is used. To synthesize memory in a partitionable way, the compiler optimization level must be 3. The compiler generates the machine code, which consists of the data memory before the execution, and the instructions to be executed.

2.3 Memory Map Extraction

Memory Map Extraction block reads the machine code and generates the corresponding Local Memory header file for the HLS. Local Memory header contains instruction and data array. In the instruction array, each element contains 32-bit instructions. The size of the instruction array depends on the generated machine code. The data array contains 32 bits as 4x8, partitioned as 4 dual port 8 bit sized memory arrays. The length of the data array depends on the linker file. The header also contains macros for each instruction in the instruction binary. For example, if instruction binary contains an *ADDI* instruction, header contains *#define ADDI* directive. Macros are used in HLS to remove unused instructions of RISC-V and to create an area efficient core.

2.4 SISD RISC-V ISS Model

This model is written in C++ to be simulated with the Vivado HLS. With the use of HLS-specific constructs like ap_int library and HLS directives, overall design time is reduced. SISD model is only used in HLS C simulation to read instruction and data arrays in the local memory and generate the address header, as explained in the next part.

2.5 Detection of SIMDifiable Regions

Instructions in the local memory header are simulated in HLS without Register-Transfer Level (RTL) synthesis using the SISD RISC-V ISS model. While simulating, the model constantly reads the four variables (StartPar, n, $A_{data,start}$, $A_{data,end}$) from their respective local addresses, Fig. 6. When the StartPar is read as "1", it means that simulation is entering the SIMD loop and when it is read as "0", it means that simulation is exiting the SIMD loop. Meanwhile values $A_{data,start}$, $A_{data,end}$, and par_num, which are set before SIMD loop, are saved to the Address Header.

After exiting the loop, the model checks the branch instruction of the SIMD loop. The start of the SIMD loop, branch target address, is equal to the sum of sign extended immediate offset, imm[12:1], and branch PC address, Fig. 7. Together with the branch target address and next value after branch PC address, the register numbers and contents given in the source register (rs1 and rs2) fields of the branch instruction are saved to the Address Header. Detailed explanation about how register numbers and contents are used for transition between normal processing mode and SIMD processing mode is given in Section 3.1.

2.6 SIMDification

SIMDification block generates the HLS SIMD header file that consists of slave PE and cache parameters, partitions, and functions used in the SIMD execution. By default, this block uses the unroll

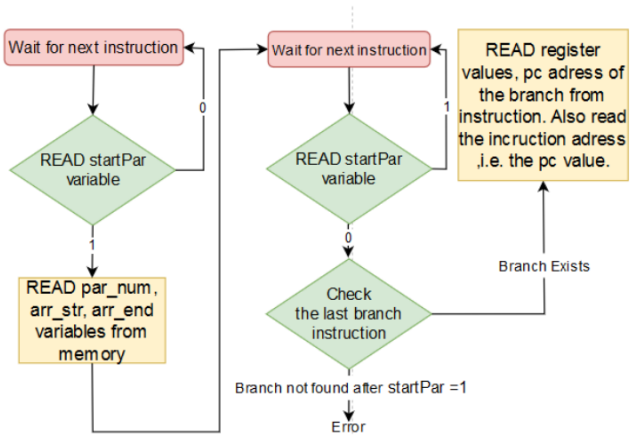

Figure 6: Flow diagram for Detection of SIMDifiable Regions block.

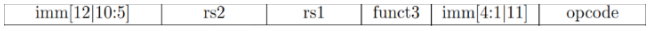

Figure 7: B-type instruction structure for RISC-V, rs1 and rs2 are the source registers for branch operation.

factor determined in the C code, but it can be overwritten to reconfigure SIMD processor without re-compiling it from scratch. SIMD processing can be applied to any memory partitionable loop in the application. SIMDified local data memory is generated by allocating all the data between the $A_{data,start}$ and $A_{data,end}$ into n equisized partitions, as shown in Fig. 8. The master PE acts as n-th slave during SIMD processing. Start and end addresses of the partitions are saved to the SIMD header. These are also used while transitioning between standard processing mode and SIMD processing mode. SIMDification block also generates constant memory tags for every word in the local data memory. CPU looks at the tags to determine which memory address belongs to which memory partition. Tagged memory architecture will be detailed in Section 3.1.

2.7 SIMD RISC-V processor description Code in C++

SIMD RISC-V processor description Code is a HLS code that is written in C++ and is responsible for generating processor system with dynamic branch prediction. It generates two types of datapath:

- *Master Datapath:* It is always executed, unique and responsible for branch prediction, stalls, and other control signals. Master datapath can access all the local memory (data and instruction), external memory, and register file array.
- *Slave Datapath:* There are $n-1$ slaves, which are executed only while SIMD processing. Each slave can only access its own register file and its own memory partition. In SIMD processing, slaves are not executed if the instruction is a branch or a jump, or an instruction is accessing a different memory address than its own partition (common memory).

Master and slave datapaths are entangled in the processor and not single blocks, but for the sake of clarity they will be referred as

Master and Slave processing elements throughout this paper. An illustrative partitioning is given in Fig. 8 for n=2 and n=4 for the matrix-vector multiplication code. In a memory partitionable loop, every load or store is accessing a different part of the memory or a common memory address. So, there are no dependencies between iterations. In each iteration SIMD loops either access to the common memory (like vector load) or they all access to a different part of the memory (like matrix load/store). In matrix multiplication, SIMD Slaves are not executed when the code is accessing the common memory (v block). Instead, master LOADs v array and writes to all n registers. Memory of the matrix is partitioned amongst PEs, the vector memory will reside in the non-partitioned common memory, and only the master PE can access it and write to all registers. If v must be STOREd inside SIMD loop, it must be part of partition matrix A.

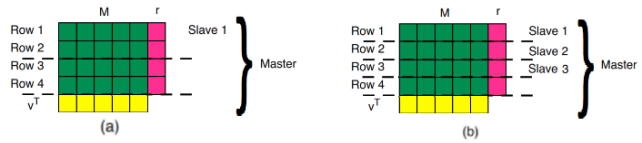

Figure 8: Example partitioning for a) n=2 , b) n=4

2.8 High Level Synthesis

Using HLS, SIMDify synthesizes the processor using generated headers and PE codes written in C++. For different applications the flow must start from the beginning. For the same application with different unroll factors, starting from the SIMDification step is enough.

3 CORE ARCHITECTURE

Our soft application-specific SIMD-processor consists of two main parts: A relatively large master PE and small slave PEs. Using HLS, SIMDify can combine and connect master and slaves to generate various SIMD processors for an application depending on the unroll factor. The processor is designed in C++ and synthesized in Vivado High-Level Synthesis 2019.2.

3.1 SIMD Processor

In our system, software loop is unrolled in hardware level to be executed in parallel as SIMD. Execution results in n times the latency gain for the SIMD executed part. The user guides the SIMDify, and the framework configures the processor accordingly. This process does not require inline assembly or custom instructions. It only requires modification on the SIMDified loop itself, thus, rest of the application does not need to be modified. Also, no extra instruction overhead is added to instructions generated by the compiler.

The overall SIMD processor architecture is shown in Fig. 9. SIMD processor consists of a master and $n-1$ slaves. Using HLS, SIMDify combines and connects these PEs to generate different SIMD processors for each application and unroll factor. Proposed processor architecture is the first RISC-V processor with SIMD support designed using HLS. In the figure, thin arrows are for single data and exclusive to master. Thick arrows indicate busses where both master and slaves execute. The designed processor runs in one of two different modes at any given time:

- Standard mode where the only active PE is the master.
- Parallel mode for SIMD processing where all PEs are active.

In the fetch stage, the master checks the PC value to start or end the SIMD processing. Before beginning the SIMD processing, the master initializes all slaves by writing different values of the SIMD loop iterator to the register files. These values are pre-calculated by the SIMDify tool as explained in the previous section.

In a SIMD loop machine code, rs1 and rs2 of branch source registers are set as initial and final addresses of the memory partition. Register of the initial_address is incremented until it's the same as final_address. This is purely done by compiler and similar for every SIMDifiable loop.

Consider an example where, unroll factor is 3, and SIMD loop accesses addresses 1 to 30. So, initial_address is 1 and final_address is 30. Master PE overrides "set rs1 and rs2" instruction and sets rs1 and rs2 values of the slave PEs as 1, 11 and 10, 20 and master PE as 21 and 30 respectively. This approach is similar to loop unrolling. This does not take additional time, since initialization is executed instead of "set rs1 and rs2" instruction. After the loop, master PE continues its normal operation. Since memory accessed in each iteration corresponds to different memory partitions, the system can be executed as SIMD.

The master determines execution mode by checking the program counter value in the fetch stage. System runs in the parallel mode if the PC value corresponds to the SIMD loop and runs in the standard mode if it does not. Additionally, only the master PE is active if the instruction is LUI, AUIPC, JUMP, or BRANCH, or requires access to non-partitioned (common) local memory.

Local data memory consists of a data field that has random access data and a tag field that identifies this data. Tag field is generated by the SIMDify and cannot be accessed by instructions. In the execute stage, the tag of the data is read from the tag field, and it is used to set the enable signals of the memory partitions. Then, in the memory stage, enable signals are used by the PEs to access the correct memory partition. There are three possible outcomes depending of the tag values and current mode:

- In standard mode: Only the master is active and tag value is used to give access to the master PE to the demanded memory partition.
- In parallel mode and all addresses are the same: That means PEs are reading from a common memory like v. In this case only the master PE accesses the memory and writes to all register files.
- In parallel mode and all addresses are different: That means core is executing as SIMD and every PE reads and writes to its own partition, by using their dedicated RFs.

Size of the tag field depends of size of the local data memory and the unroll factor as $\frac{LocalDataMemory}{32} * \lceil log_2(n) \rceil$. If the unroll factor is 1, tag field is not generated since there is only single memory partition and single PE.

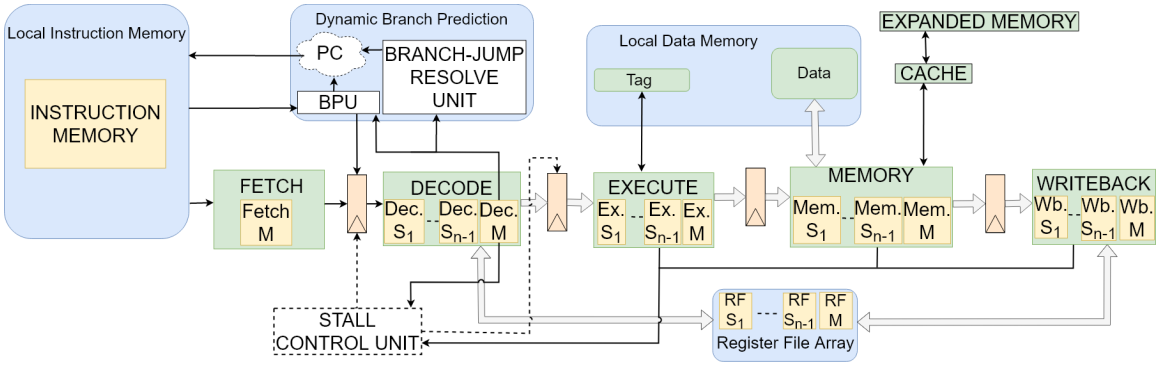

Figure 9: Block diagram of overall system with 1 master PE and n-1 slave PEs. Local data memory is detailed in Fig. 2.

3.2 Master (Scalar) PE

The scalar PE is master for slave PEs and supports *riscv32i* instruction set and MUL, MULH, MULHSU, MULHU multiplication instructions. It has a standard five-stage pipeline consisting of fetch, decode, execute, memory, and writeback stages [19]. Instructions are fetched and issued for execution in program order. Using the aforementioned directives unused instruction blocks are removed, which reduces the area. All data dependency hazards are solved via stalling.

Branch hazards are handled with dynamic one-level branch prediction with a 1-bit saturating counter. Jump instructions are resolved in the decode stage, which results in 1 cycle overhead. Branch instructions are also resolved in the decode stage to reduce misprediction penalty. In the figure, top input of the pipe registers indicates flush, and bottom indicates stall. The core has a local memory for faster processing, and memory can be expanded with a cache connected to external memory. All instructions are stored in the local instruction memory. Latency for memory stage is a single cycle for local memory. If the cache is implemented, the overall pipeline depth does not change, but the memory stage may take multiple cycles to execute. All local memories and cache memories are asynchronous read and synchronous write. It should be noted that block diagram is behaviorally correct, however, in HLS register file, writeback, and decode stages are written as one block. This is done to generate two port asynchronous read, one port synchronous write register file (RF). Operands are sent to the decode first, and sent to the execute from there.

The master PE is also responsible for starting and ending the SIMD processing. Before beginning the SIMD processing, the master PE initializes slave PEs as explained in the Section 3.1.

3.3 Slave PE

Slave PEs only consist of decode, execute, memory, and writeback units. All slave PEs can only access to their individual 32-bit register files, and their partition in the main memory. Since it is guaranteed that all PEs will execute the same instruction at any given time, redundant signals are trimmed to reduce area. Slave PEs also do not have a stall, fetch, or branch units. They are generated only when the user demands a SIMD processing and are fully controlled by the master PE.

Slave PEs are generated by using HLS loop unroll pragmas with case blocks. So, they use the same blocks and same unified local memory as the master PE. SIMDify detects SIMD loops and guides HLS to generate slave PEs accordingly. This approach generates general purpose slave PEs and removes the need for designing custom modules per application. Slave PEs can execute most instructions of the supported ISA. Slave and master PEs are further reduced to only execute necessary instructions per application basis, which reduces area. If master PE is modified to include extra instructions, slave PEs can be easily scaled to include these instructions as well.

4 BACKGROUND AND RELATED WORKS

In this work, RISC-V [41] ISA is chosen due to its open-source, free, active, and well-documented nature. Even though RISC-V ISA defines two types of extended instruction sets for data parallel processing, "P" (Packed SIMD) extension and "V" (Vector) extension, "V" extension is not tailored for small scale SIMD applications, and the "P" extension cannot be scaled without recompiling the application (not scalable). Scalable SIMD instructions can be implemented in RISC-V processors by extending standard instruction set with non-standard custom instructions [17, 21, 36].

V extension can work with small scale vector lengths, however it's intended for high performance computing with its OpenMP support. Allowing vector processing requires significant changes in the processor architecture, whereas SIMDify can unroll loops with its simple core architecture. Widths of vector can only be a power of 2, whereas SIMDify can take custom unroll factors as an input.

P extension requires new machine code for every time when number of parallel computing units is changed. SIMDify can use same machine code for any number of parallel computing units. Hence, user can easily explore design space to optimize overall design without recompiling the software. Currently, P extension is not supported by standard RISC-V compiler. Both extensions are not standardized and there are no existing open-source designs with these extensions. Note that, SIMDify is an open-source project.

In conventional SIMD processing, SIMD instructions are used, which necessitates either inline assembly or modification of the RISC-V GCC toolchain. Intermediate representation generated by the compiler front end can also be used to detect SIMDifiable regions with cost of forcing users to a custom compiler. SIMDify

does not force users which compiler to use and it automates the process after the compilation. In the literature, SIMD computation is achieved using custom instructions in [17, 21, 36] by extending RISC-V ISA, and in [15] by extending SimpleRISC ISA. Automated tools like [3] or [1] where ISA extended processors can be generated together with SDK exist, but this also limits the user by forcing one IP ecosystem. Chipyard RoCC [2] is another commonly used framework for designing accelerators for Rocket processor. However, communicating accelerators with its RoCC interface also requires a custom software toolchain. SIMDify solution can be applied to any SIMD loop that satisfies the memory constraints, whereas, contemporary approaches might require different custom instruction for each new application.

HLS tools give designers better authority over-optimization of their design architecture. However, HLS often requires guidance from the user to generate the architecture through the use of pragmas. Quality of the design is directly dictated by the selected pragmas. Hence, iterative design process for finding the best solution takes a considerable design effort and time. In our approach, we have already designed the template SIMD processor architecture. The SIMDify framework, which generates application specific SIMD architecture, greatly reduces the design effort and time of the user. SIMDify fully utilizes HLS and its C like header structure to reduce design time.

In [29] it is shown that HLS can reduce the design effort while still achieving a good quality of results. In terms of area, the processor designed in HLS is %50 larger than its RTL equivalent. Several implementations of RISC-V have been made in Chisel [8], a Scala-embedded language that allows functional and object-oriented descriptions of hardware circuits, like [11, 39]. Other RISC-V processors designed using a high-level synthesis are Comet core [32] in Catapult HLS, HL5 [29] in SystemC and approximate CPU [37] in Vivado HLS. All mentioned processors does not have a SIMD support, but Comet does allow instruction extensions by modifying the HLS code. However, compiler modification must be done by the user. HL5 and Comet have stable $riscv32im$ instruction support. [34] and [6] are MIPS architecture based processors, utilizing Vivado and LegUp [10] HLS tools, respectively. In [22] SIMD processor for software-defined radio (SDR) applications is designed using high-level OpenCL language [30].

5 EXPERIMENTAL RESULTS

5.1 Setup

Experiments are carried out on Zynq-7020-2CLG484-1 FPGA as hardware simulation. Each tested algorithm uses less than 32KB of total memory, which fits in the local memory. C code is compiled with riscv64-unknown-elf-gcc 7.2.0 with following options $-mabi = ilp32 - g0 - O3 - march = rv32im - Wl, --no - relax - nostartfiles$.

We implemented algorithms of matrix vector multiplication (MVM), sum of absolute distances (SAD), sum of squared distances (SSD), artificial neural networks (ANN), k-nearest neighbors with selective sort (KNS) and k-nearest neighbors with qsort (KNQ). Both massively parallelizable algorithms with large parallel portions (MVM, SAD, SSD, ANN) and partially parallelizable algorithms with smaller parallel portions (KNS, KNQ) are tested. Both large and small scale applications only requires user modification on the SIMD loop, rest of the application does not change. SIMDify focuses on unrolling user picked critical loops in the application.

5.2 Results

We verified the HDL generated by SIMDify against SISD RISC-V ISS Model and confirmed that their outputs agree. To measure the latency, HLS cosimulation results are used, which are based on synthesized HDL code. Resource usage and clock speed values are taken from the synthesis report. Processor generation time is around 4 min on a four-core Intel Xeon server.

Clock speeds are calculated when the target clock is 15ns, and uncertainty is %12.5. Three different parameters affect the clock speed: multiplication, cache and the number of PEs, i.e. n. If an algorithm uses one of multiplication instructions, MUL block is inserted, and its 11 ns slack causes the bottleneck. However, if it does not use any multiplication instructions, its period changes depending on the number of slave PEs, Table 1. In the algorithms mentioned above, only SAD does not use any multiplication instructions. If the unroll factor is one, only the scalar PE with one partition is used, so extra logic for slave PE routing is removed. Cached clock-speed, 11.827 ns, is faster than the non-cached core, but, it requires three times more clock cycles to complete. Different master and slave PE architectures might change the clock speed of the generated processor. However, SIMDify framework itself is independent from the master and slave PEs.

Speed-up and latency values for each algorithm are given in Table 1. Latency without SIMD processing is given in terms of the number of clock cycles, and the latency speedup is calculated as $\frac{Lat_1}{Lat_n}$. 150 iteration MVM, SAD, SSD, KNS, and KNQ and 75 iteration ANN algorithms are SIMDified with 5, 15, and 25 unroll factors. "Max" is used to show maximum achievable parallelism. The processor does not lose any clock cycles when going into or exiting the SIMD mode. So, calculated latency values are in correspondence with the Amdahl's law [31]. However, since clock speed is different with and without the slave PEs, it must also be considered when calculating the true speed-up, which is also given separately in the Table 1.

The resource used for MVM algorithm is given in the Table 2. The number of BRAM required for each algorithm is dependent on the number of instructions in the algorithm. The number of LUTs and FFs required is roughly similar for each tested algorithm. The number of DSP blocks required is dependent on the number of multiplication instructions used in the algorithm. So if all of MUL, MULH, MULHSU, MULHU instructions are used, the processor will require 12 DSPs per PE. For applications with no multiplication instructions, such as the SAD algorithm, no DSP blocks are used. DSP usage improved drastically by application specific block removal mentioned in the Scalar PE subsection. With the same technique, BRAM and FF usage does not change, and LUT usage is improved by ~4%. In all test cases, number of LUTs was the limiting factor in deciding the maximum number of slave PEs (unroll factor). Maximum 25-30 PEs can be implemented inside the Zynq-7020-2CLG484-1 FPGA. It can be seen that DSP increase is linear w.r.t unroll factor. For MVM with unroll factor 25, the application finishes 16.9 times faster by using 8.53 times more BRAM, 25 times more DSP blocks, 12.09 times more FF, and 13.66 times more LUT.

Table 1: Latency (clock cycles), Clock Speed and Speed-up for unroll factor 5, 15, 25, and maximum achievable parallelism

Algorithm	Single PE (n=1) Latency (Cycles)	Clock Period (ns)				Speed-up (Cycles)				Speed-up (Time)			
		1	5	15	25	5	15	25	Max	5	15	25	Max
MVM	6580	12.75	12.78	12.78	12.78	4.63	11.75	16.96	50.62	4.62	11.72	16.92	50.50
SAD	10640	11.92	12.69	12.64	12.66	4.75	12.67	19.00	76.00	4.46	11.95	17.89	71.56
SSD	8680	12.75	12.78	12.78	12.78	4.72	12.40	18.39	66.77	4.71	12.37	18.34	66.58
ANN	14008	12.75	12.78	12.78	12.78	4.76	12.76	19.22	79.59	4.75	12.73	19.17	79.38
KNS	156622	12.75	12.78	12.78	12.78	1.36	1.45	1.47	1.50	1.35	1.44	1.46	1.49
KNQ	118332	12.75	12.78	12.78	12.78	1.55	1.70	1.73	1.79	1.54	1.69	1.72	1.78

Table 2: Resource usage of Matrix Vector Multiplication for unroll factor 5, 15, 25

Type	Available	Utilization			
		1	5	15	25
BRAM	280	13	46	70	111
DSP	220	3	15	45	75
FF	106400	637	1619	4787	7699
LUT	53200	3406	9958	27649	46541

Table 3: FPGA Frequency Comparison table for RISC-V HLS cores

Core Name	Frequency	SIMD	Language
Comet [32]	70 MHz	No	Catapult HLS
Rocket [7]	76 MHZ	No	Chisel
HL5 [29]	Unknown	No	SystemC
SIMDify with No Slave PEs	78.4 MHz	No	Vivado HLS
SIMDify with 30 slave PEs	78.2 MHz	Yes	Vivado HLS

In MVM, SAD, SSD and ANN, we showed experimentally that if the latency is mainly due to partitionable loop, SIMDify can speed-up the design drastically. However, this isn't the case with KNS and KNQ, which can only be reduced to 60% of single cycle latency due to Amdahl's law. Users must decide if SIMDifing can improve the application, and how much.

It should also be mentioned that Comet [32] also reports around 70 MHz on Artix 7 FPGA and takes around 2 minutes to synthesize. In [32], Rocket Core [7], another core written in Chisel HDL, has been mentioned to have 76 MHZ on Artix 7 FPGA. Finally, HL5 has clock frequency between 700 MHz and 2GHz in 32 nm CMOS. Proposed SIMD processor architecture has a similar clock frequency with aforementioned HLS cores even with 30 slaves, Table 3. The solution proposed in this paper is scalable, open-source, and does not depend on non-standard compilers to minimize the user workload. Using SIMDify, hardware-level parallelization is achieved without the use of additional instructions.

6 CONCLUSION AND THE FUTURE WORK

In this paper, SIMDify, hardware-software parallelization framework for generating SIMD capable application-specific RISC-V instruction set processors, and the generated application specific SIMD processor structure are presented. SIMDify combines HLS with standard RISC-V compiler to generate a five-stage pipelined SIMD processor written in C++. SIMD processor consists of master and slave PEs. Using HLS, SIMDify combines and connects these PEs to generate different SIMD processors for each application. SIMD processor architecture is the first HLS designed RISC-V processor with SIMD support.

Applicability of the SIMDify is tested on selected algorithms. System runs on Zynq-7020-2CLG484-1 FPGA and it operates on approximately 78 MHz. Processor is based on an FPGA, so it can be combined with other applications as an accelerator. Since it's designed in HLS it can be easily modified and improved by many users.

In terms of scalar PE, cache implementation can be improved. Also, forwarding structure can be implemented to reduce the number of stalls, and multi-cycle instructions such as DIV and REM can be implemented for full *riscv32im* support. The main bottleneck of the core is 11 ns single cycle 32x32 multiplication instruction, which can be improved by using a custom multiplication block or multi-cycle multiplication operation.

Existing external memory and cache structure can be used to increase the total data memory size. However, data in the external memory cannot be used in SIMD processing. So, to increase the size of the SIMD processed memory, tag field size must also increase. Since, SIMDification block generates constant memory tags for every word in the local data memory, generated tag field size increases proportionally with the local data field size. This problem can be solved by decreasing tag field size per word or by changing the memory structure to extend local memory without increasing the tag size.

Designed ASIP and SIMDify framework can be applied to any iterative loop if the loop does not include any conditional branching and if the loop satisfies the memory constraints. We believe SIMDify solution is better and more comprehensive than the alternative: modifying an each application to make it compatible with each custom instruction. SIMDify automates processor generation and creates an open-source framework that can easily be used by anyone to achieve SIMD computation. Despite the limitations of the current HLS tools, the time to design and optimize the processor

significantly decreased compared to traditional RTL flow. Addressing these limitations represents a research driver for future HLS tools.

ACKNOWLEDGMENTS

Authors would like to thank Omer Faruk Irmak for his support in machine code generation using compilers.

The authors would also like to thank anonymous referees for their valuable comments and helpful suggestions. This work is supported by the Turkish Ministry of Science, Industry and Technology under Grant No. 58135.

REFERENCES

[1] [n.d.]. *ASIP Designer*. Ph.D. Dissertation. https://www.synopsys.com/designware-ip/processor-solutions/asips-tools.html

[2] [n.d.]. Chipyard. https://chipyard.readthedocs.io/en/latest/Customization/RoCC-Accelerators.html

[3] [n.d.]. Codasip Studio. https://codasip.com/codasip-studio/

[4] [n.d.]. SIMDify Framework. https://github.com/alpsark/SIMDify

[5] [n.d.]. Vivado High-Level Syntesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[6] Tanvir Ahmed, Noriaki Sakamoto, Jason Anderson, and Yuko Hara-Azumi. 2015. Synthesizable-from-C embedded processor based on MIPS-ISA and OISC. *Proceedings - IEEE/IFIP 13th International Conference on Embedded and Ubiquitous Computing, EUC 2015* (2015), 114–123. https://doi.org/10.1109/EUC.2015.23

[7] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Technical Report* UCB/EECS-2016-17 (2016). https://doi.org/10.1023/A:1010000313106

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. *Proceedings - Design Automation Conference* (2012), 1216–1225. https://doi.org/10.1145/2228360.2228584

[9] Andrew Becker, Scott Sirowy, and Frank Vahid. 2011. Just-in-time compilation for FPGA processor cores. In *2011 Electronic System Level Synthesis Conference (ESLsyn)*. IEEE, 1–6. https://doi.org/10.1109/ESLsyn.2011.5952282

[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*. ACM Press, New York, New York, USA, 33. https://doi.org/10.1145/1950413.1950423

[11] Christopher Celio, Pi Feng Chiu, Krste Asanović, Borivoje Nikolić, and David Patterson. 2019. BROOM: An Open-Source Out-of-Order Processor with Resilient Low-Voltage Operation in 28-nm CMOS. *IEEE Micro* 39, 2 (2019), 52–60. https://doi.org/10.1109/MM.2019.2897782

[12] Yao Chen, Kai Zhang, Cheng Gong, Cong Hao, Xiaofan Zhang, Tao Li, and Deming Chen. 2019. T-DLA: An Open-source Deep Learning Accelerator for Ternarized DNN Models on Embedded FPGA. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 13–18. https://doi.org/10.1109/ISVLSI.2019.00012

[13] Sivakumar Chidambaram, Alexandre Riviello, J.M. PierreLanglois, and Jean-Pierre David. 2018. Accelerating the Inference Phase in Ternary Convolutional Neural Networks Using Configurable Processors. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 94–99. https://doi.org/10.1109/DASIP.2018.8596860

[14] Peter Figuli, Carsten Tradowsky, Nadine Gaertner, and Juergen Becker. 2013. ViSA: A highly efficient slot architecture enabling multi-objective ASIP cores. In *2013 International Symposium on System on Chip (SoC)*. IEEE, 1–8. https://doi.org/10.1109/ISSoC.2013.6675270

[15] V. Ganesh and B. V.H. Sandilya. 2019. Implementation of SIMD Instruction Set Extension for BLAKE2. *2019 10th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2019* (2019). https://doi.org/10.1109/ICCCNT45670.2019.8944835

[16] Chang Gao, Stefan Braun, Ilya Kiselev, Jithendar Anumula, Tobi Delbruck, and Shih-Chii Liu. 2019. Real-Time Speech Recognition for IoT Purpose using a Delta Recurrent Neural Network Accelerator. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5. https://doi.org/10.1109/ISCAS.2019.8702290

[17] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. 2020. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 186–191. https://doi.org/10.23919/DATE48585.2020.9116529

[18] Yuchen Hao, Zhenman Fang, Glenn Reinman, and Jason Cong. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 37–48. https://doi.org/10.1109/HPCA.2017.19

[19] David Money Harris and Sarah L. Harris. 2012. *Digital design and computer architecture, 2nd edition*. Morgan Kaufmann. 1–690 pages. https://doi.org/10.1016/C2011-0-04377-6

[20] Yuanhong Huo and Dake Liu. 2017. High-Throughput Area-Efficient Processor for Cryptography. *Chinese Journal of Electronics* 26, 3 (5 2017), 514–521. https://doi.org/10.1049/cje.2017.03.004

[21] Yoshiki Kimura, Tomoya Kikuchi, Kanemitsu Ootsu, and Takashi Yokota. 2019. Proposal of Scalable Vector Extension for Embedded RISC-V Soft-Core Processor. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 435–439. https://doi.org/10.1109/CANDARW.2019.00082

[22] Heikki Kultala, Timo Viitanen, Heikki Berg, Pekka Jaaskelainen, Joonas Multanen, Mikko Kokkonen, Kalle Raiskila, Tommi Zetterman, and Jarmo Takala. 2019. LordCore: Energy-Efficient OpenCL-Programmable Software-Defined Radio Coprocessor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 5 (5 2019), 1029–1042. https://doi.org/10.1109/TVLSI.2019.2897508

[23] Carlos Andres Lara-Nino, Arturo Diaz-Perez, and Miguel Morales-Sandoval. 2020. Lightweight elliptic curve cryptography accelerator for internet of things applications. *Ad Hoc Networks* 103 (6 2020), 102159. https://doi.org/10.1016/j.adhoc.2020.102159

[24] Chi-Ming Lee, Yong-Jyun Huang, Chih-Wei Liu, and Yarsun Hsu. 2016. DeAr: A framework for power-efficient and flexible embedded digital signal processor design. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 658–661. https://doi.org/10.1109/APCCAS.2016.7804083

[25] Meng Li, Frederik Naessens, Min Li, Peter Debacker, Claude Desset, Praveen Raghavan, Antoine Dejonghe, and Liesbet Van der Perre. 2013. A processor based multi-standard low-power LDPC engine for multi-Gbps wireless communication. In *2013 IEEE Global Conference on Signal and Information Processing*. IEEE, 1254–1257. https://doi.org/10.1109/GlobalSIP.2013.6737136

[26] Gai Liu, Joseph Primmer, and Zhiru Zhang. 2019. Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3316781.3317890

[27] Zhiqiang Liu, Jingfei Jiang, Guoqing Lei, Kai Chen, Buyue Qin, and Xiaoqiang Zhao. 2020. A Heterogeneous Processor Design for CNN-Based AI Applications on IoT Devices. *Procedia Computer Science* 174 (2020), 2–8. https://doi.org/10.1016/j.procs.2020.06.048

[28] Steffen Malkowsky, Hemanth Prabhu, Liang Liu, Ove Edfors, and Viktor Öwall. 2019. A programmable 16-lane SIMD ASIP for massive MIMO. In *Proceedings - IEEE International Symposium on Circuits and Systems*, Vol. 2019-May. 1–5. https://doi.org/10.1109/ISCAS.2019.8702770

[29] Paolo Mantovani, Robert Margelli, Davide Giri, and Luca P. Carloni. 2020. HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis. *Proceedings of the Custom Integrated Circuits Conference* 2020-March (2020). https://doi.org/10.1109/CICC48029.2020.9075913

[30] Aaftab Munshi. 2009. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[31] David P. Rodgers. 1985. Improvements in multiprocessor system design. *ACM SIGARCH Computer Architecture News* 13, 3 (6 1985), 225–231. https://doi.org/10.1145/327070.327215

[32] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. 2019. What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Vol. 2019-Novem. IEEE, 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942177

[33] Shanlin Xiao, Dongju Li, Hiroaki Kunieda, and Tsuyoshi Isshiki. 2016. Design of an efficient ASIP-based processor for object detection using AdaBoost algorithm. In *2016 7th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*. IEEE, 96–99. https://doi.org/10.1109/ICTEmSys.2016.7467129

[34] Sam Skalicky, Tejaswini Ananthanarayana, Sonia Lopez, and Marcin Lukowiak. 2016. Designing customized ISA processors using high level synthesis. *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015* (2016), 0–5. https://doi.org/10.1109/ReConFig.2015.7393299

[35] Tomoki Sugiura, Shoko Nakatsuka, Jaehoon Yu, Yoshinori Takeuchi, and Masaharu Imai. 2014. An efficient data compression method for artificial vision systems and its low energy implementation using ASIP technology. In *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*. IEEE, 81–84. https://doi.org/10.1109/BioCAS.2014.6981650

[36] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2019. Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 654–657. https://doi.org/10.23919/DATE.2019.8714897

[37] İbrahim Taştan, Mahmut Karaca, and Arda Yurdakul. 2020. Approximate CPU Design for IoT End-Devices with Learning Capabilities. *Electronics* 9, 1 (1 2020), 125. https://doi.org/10.3390/electronics9010125

[38] Tsung-Han Tsai, Yuan-Chen Ho, and Ming-Hwa Sheu. 2019. Implementation of FPGA-based Accelerator for Deep Neural Networks. In *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 1–4. https://doi.org/10.1109/DDECS.2019.8724665

[39] Angie Wang, Woorham Bae, Jaeduk Han, Stevo Bailey, Orhan Ocal, Paul Rigge, Zhongkai Wang, Kannan Ramchandran, Elad Alon, and Borivoje Nikolic. 2019. A Real-Time, 1.89-GHz Bandwidth, 175-kHz Resolution Sparse Spectral Analysis RISC-V SoC in 16-nm FinFET. *IEEE Journal of Solid-State Circuits* 54, 7 (7 2019), 1993–2008. https://doi.org/10.1109/JSSC.2019.2913099

[40] Yi Wang and Yajun Ha. 2014. A Performance and Area Efficient ASIP for Higher-Order DPA-Resistant AES. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 4, 2 (6 2014), 190–202. https://doi.org/10.1109/JETCAS.2014.2315877

[41] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. 2011. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. *Electrical Engineering* I (2011), 1–34. https://doi.org/10.1109/ESSCIRC.2014.6942056