# nAIxt: A Light-Weight Processor Architecture for Efficient Computation of Neuron Models

Kevin Kauth(✉), Christian Lanius, and Tobias Gemmeke

IDS, RWTH Aachen University, Aachen, Germany
kauth@ids.rwth-aachen.de

**Abstract.** The simulation of biological neural networks holds immense promise for advancing both neuroscience and artificial intelligence. Due to its high complexity, it requires powerful computers. However, the high proportion of communication and routing makes general-purpose processing architectures, as used in supercomputers, inefficient. Dedicated hardware, such as ASICs, on the other hand, can be specifically adapted to this type of workload. However, integrated circuits are rigid, thereby eliminating the use of future neuron models.

To address this contradiction, this paper presents a programmable architecture for the computation of neuron models. Thanks to its Turing completeness, it enables embedding biological neural networks simulators into integrated circuits while simultaneously allowing adaptation of the neuron model. To assess suitability, both dedicated circuits and off-the-shelf processors are examined regarding AT efficiency. The proposed versatile architecture turns out to be up to 1800x more area efficient than a RISC-V processor, thereby playing a vital role in accelerating neuroscience simulation and research in AI.

**Keywords:** neuroAIx · Neuromorphic · Accelerator · Neuron · Processor · Biological Neural Network · Simulation · Software Pipelining · VLIW

## 1 Introduction

Improving our understanding of the human brain promises major advances in the fields of medicine [2,17] and artificial intelligence [12]. However, due to poor in vivo observability, research relies primarily on simulations [16]. These simulations are based on well-understood mechanisms of individual neurons and synapses and aim to enable controlling and studying their interplay on a large scale. During their execution, neuron states are continuously updated based on physical models, while generated action potentials must be distributed through

This work was supported in part by the Federal Ministry of Education and Research (BMBF), Germany, through the NEUROTEC II Project under Grant 16ME0399.


D. Fey et al. (Eds.): ARCS 2024, LNCS 14842, pp. 3–17, 2024.
https://doi.org/10.1007/978-3-031-66146-4_1

a large network to connected post-synaptic neurons. While communication and routing have shared common requirements across typical network models, which can be estimated precisely even for future systems [9], neuron and synapse models constantly evolve along with scientific progress and, therefore, suffer from extensive alterations.

Due to the high complexity of biological neural networks (BNNs), their simulation requires high computing power. In addition, modern general-purpose computing systems are not designed to handle the high amount of irregular and comparable small messages. This makes the use of dedicated architectures inevitable. For example, digital circuits based on FPGAs have led to significantly higher simulation speeds and energy efficiencies compared to general-purpose CPU or GPU systems [10]. Although such FPGA systems inherently support the benefit of adaptability to new findings in neuroscience through reconfigurability, porting digital circuits to application-specific integrated circuit (ASICs) can provide an even greater increase in speed and energy efficiency. For this purpose, all system parts subject to continuous evolution would require implementation as programmable parts rather than as fixed, dedicated digital circuits.

One of these affected parts is the calculation of the neuron model. There is a large variety of neuron models with different levels of biological realism, accuracy, and complexity. Their contribution to the computational load of the overall simulation is comparatively low, and their models can, unlike the communication part, be calculated very efficiently on GPUs [11], for example, due to their good parallelization capability. However, low latencies are crucial for simulations with high acceleration factors, making tight coupling between neuron calculation and network essential. While most dedicated systems are based on hardwired circuits, which come optimized for only one neuron model with mainly changeable parameters [4,5,8,14,18,19,22,24], only few support fully flexible processors for the implementation of arbitrary neuron models. One of the latter is SpiNNaker [6], which combines dedicated hardware for spike routing with a general-purpose CPU on *ARM* basis. Another one is Loihi2 [15], which implements a custom processor architecture for neuron computation. Here, we focus on comparing to general-purpose processors based on RISC-V, which are comparable with ARM processors while offering open source implementations.

This work aims to answer the research question of which processor architecture is suited for use in ASICs simulating BNNs and whether simple in-house developments can compete with established sophisticated architectures. To answer this question, this paper presents a configurable processor architecture that aims for the highest possible efficiency while maintaining Turing completeness. Considering the aforementioned possibility to compute neuron updates independently, the concept of scaling architectures applies, i. e., the product of area $A$ being proportional to the degree for parallelism and the computation time $T$ is to first order constant. In this study, area is quantified in terms of used lookup tables (LUTs). Hence, the classic $A \cdot T$ cost product is represented in this study as number of LUTs times the duration to finish a neuron update. In this classic way, optimization of $AT$-efficiency is independent of the actual

silicon resources dedicated to parallelization. The presented processor *"nAIxt"* is compared with processors based on the RISC-V instruction set, as well as the *Xilinx MicroBlaze*$^{TM}$ by calculating leaky integrate-and-fire neuron models with current-based (CUBA) synapses and conductance-based (COBA) synapses [23]. Furthermore, various numerical integration methods are considered. A dedicated, fully pipelined digital circuit, representing non-programmable systems, serves as the AT optimal reference in each case. Compared to all programmable processors tested, the AT efficiency achieved by nAIxt is at least one order of magnitude higher.

## 2 Processor

One design goal to optimize the AT efficiency is to utilize all instantiated computing units as intensively as possible and, conversely, to avoid instantiating units with poor utilization. The consequence of this goal is the simultaneous utilization of multiple of these units, i.e., a superscalar architecture, plus outsourcing control logic, such as scheduling and microcode, from the processor to the compiler. This corresponds to a very large instruction word (VLIW) architecture. Admittedly, this architecture never took hold in the general purpose area due to sub-optimal scheduling at compile time and the comparably low overhead of out-of-order execution units in competitive architectures. However, the specific metrics of the underlying use cases and the short executables required for a single neuron update enable high unit utilization at low hardware complexity.

A vector processor approach is not considered, as multiple instances of the processor developed here can also be operated in parallel. However, multiple instantiations of single units are examined to balance the unit utilization.

Although the decision against a micro opcode increases the executable size, this disadvantage is not decisive given the comparatively small size of neuron models already discussed. On the other hand, it reduces the processor's size and latency. Moreover, the resulting ability to directly control data flows offers possibilities like reusing data or realizing jumps without special commands.

### 2.1 Initial Architecture

The initial architecture of nAIxt is shown in Fig. 1. It is based on a 32-bit in-order Harvard architecture, which operates solely word-wise and has 16 registers, 13 of which are general-purpose. Register r13 contains the program counter (PC) and is equipped with its own adder for advancing the program. Jumps are made by writing to this register. Two 32-bit input/output ports, e.g., for processing incoming and transmitting generated spikes, are mapped onto the pseudo registers r14 and r15. Here, we explicitly opted against memory mapping, so input data is available immediately rather than one clock cycle later. In each cycle, a maximum of three registers can be written, and six can be read. All six read outputs can be used as ALU inputs, while the last two can also serve as memory addresses. The ALU contains six units that can operate fully in parallel:

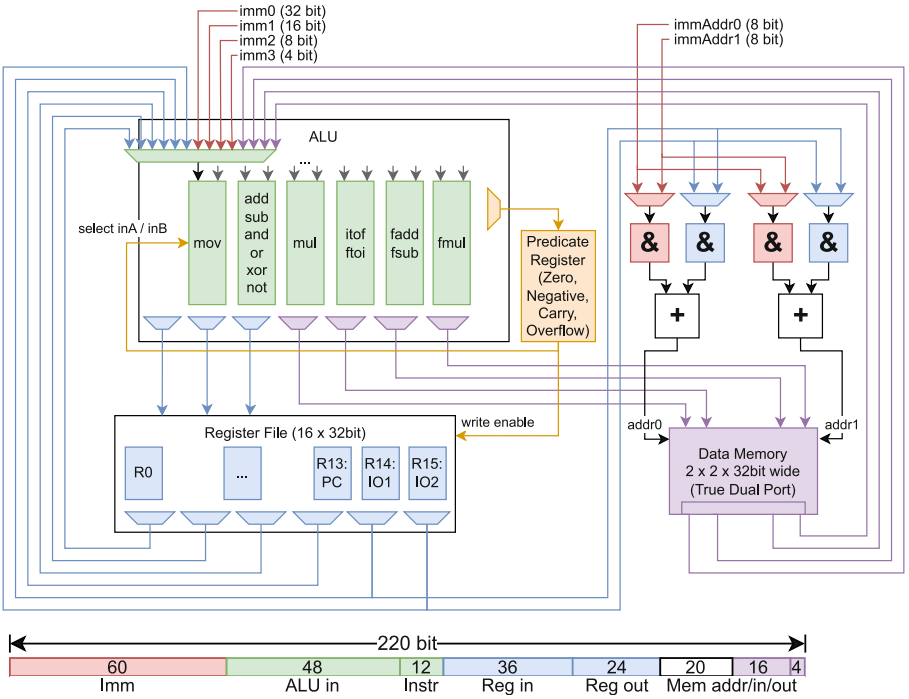


**Fig. 1.** Architecture and instruction word overview of nAIxt showing registers in blue, ALU in green, immediates in red, memory in purple, and predicate register in yellow. (Color figure online)

1. The move unit (**mov**) supports transferring the input, e.g., for copying registers, plus the opportunity to select either input A or input B (*select*) based on a condition. This function is essential to avoid conditional branches.
2. The integer unit allows the arithmetic operations **add**, **subtract**, as well as the logic operations **and**, **or**, **xor** and **not**.
3. The integer-multiplier (**mul**) performs 32-bit signed multiplications.
4. The **itof/ftoi** unit converts between floating-point and fixed-point numbers. One input specifies the decimal point position of the fixed-point number.
5. A floating-point adder/subtractor (**fadd/fsub**)
6. A floating-point multiplier (**fmul**)

Based on the knowledge about the dynamic range of numbers in neuron model computations, we streamlined the floating point units for better efficiency without changing their results compared to fully IEEE 754 compliant units.

Both inputs of each unit can be selected from 14 different sources. These include the six register outputs, four immediate values of various sizes, and four memory outputs. The immediate values are part of each instruction word and always get sign extended. The four memory outputs consist of two consecutive 32-bit words of a 64-bit wide main memory with two independent ports featuring

separate addresses. For addressing the two memory ports, the last two register outputs can each be combined with a separate immediate value, for example, to apply an offset to variables such as counters. The ALU supports seven outputs, each able to select the result of any instruction unit. Three of the outputs lead to the register file inputs, and four others to the two 64-bit ports of the main memory described above. Another 4-bit output of the ALU selects the predicates *zero*, *negative*, *carry*, and *overflow* of the result of one unit. These can be stored to the *predicate register* using a flag in the instruction word and then be used for both the select operation and conditional register write instructions. The latter also serves the purpose of avoiding program branches, thus enabling more efficient program loops. In both cases, the four predicates can be combined to form the following conditions (or integer relations after subtraction):

- always
- never
- if zero (equal)
- if negative (lower than)
- if negative or zero (lower or equal)
- if positive (greater than)
- if positive or zero (greater or equal)
- if not zero (not equal)
- if carry occurred
- if overflow occurred

The jump avoidance measures are intended to compensate for the fundamental disadvantages of the VLIW architecture by leading, in combination with software pipelining, to a high computing unit utilization. In addition, these measures enable significant potential to increase the maximum clock frequency ($F_{\max}$) by avoiding pipeline stalls even for extensively pipelined hardware.

The optimization of the processor for the highest clock frequency is not part of this work. Hence, no extensive hardware pipelining was applied to cut any timing-critical path. However, the floating-point adder and multiplier each contain one pipeline stage, which delays their result by one clock cycle. In addition, all memories have a register stage on both the input (address bus) and output side (data read bus). This also applies to the instruction memory, so the nAIxt toolchain already includes all necessary features to support hardware pipelining.

## 3 Toolchain

Within this work, the focus was on architectural exploration. In this context, the realized toolchain focuses on elements ultimately necessary to evaluate the proposed architecture's performance. Since software pipelining is a key optimization opportunity in the context of running many similar neuron updates in sequence, related support by mapping tools is compulsory.

### 3.1 Assembler

Since nAIxt is considered an evaluation platform, the most important requirement for the assembler is to support varying architectures, e.g., an adaptation of the computing units, registers, and data buses. Therefore, all supported instructions, including possible inputs, outputs, and bit widths, are declared in the assembler in the form of a table. The assembler realizes the conversion of simple instructions, such as `r0 = r1 + r2` into the atomic multiplexer configurations.

### 3.2 Scheduler

To evaluate a processor's performance, a program's instructions need to be arranged to fully exploit all features of an architecture. The scheduler is intended to automate this arrangement to support explorations on nAIxt. It comprises the three stages analysis, scheduling, and register allocation, as detailed in the following.

**Analysis.** In this first stage, the code is loaded into suitable data structures and analyzed for dependencies. First, static definitions are separated, and the remaining code is divided into basic blocks based on jumps and jump labels. Afterwards, relationships between these blocks are identified. Data dependencies between individual instructions are then created for each basic block. Finally, these dependencies are optimized in two ways. Firstly, dependencies between identical instructions are combined. Secondly, in a multi-stage process, new dependencies are introduced between two instructions that are only indirectly interdependent through a third instruction. This significantly reduces the number of combinations for the next step, resulting in faster scheduling.

**Scheduling.** Since this work aims to compare the architectures' full potential, the scheduling algorithm is not based on a heuristic but on back-tracking. In other words, all combinations are traversed while attempting to exclude impossible schedules early. The recursive procedure has the following structure:

1. For each unplaced instruction: Calculate a set of possible target addresses at which this instruction satisfies all dependencies from/to placed instructions.
2. If no instruction is left, a schedule is found.
3. If even one of the sets from step 1 is empty, scheduling is no longer possible. Return from one recursion level.
4. Heuristic: Choose the instruction with the smallest set.
5. For each address in the set, starting with the smallest:
   (a) Place the instruction.
   (b) If all resource dependencies in this cycle are fulfilled, continue with the next instruction (point 1 in the next recursion step).
6. If no schedule was found for any address, return from one recursion level.

This algorithm has the disadvantage of a relatively long runtime. However, back-tracking guarantees an optimal solution and is therefore well suited for evaluating an architecture. In any case, the schedule of each benchmark was calculated in a few minutes at most.

**Register Allocation.** A simple greedy algorithm is used to carry out register allocation. It determines the validity range of all variables and then assigns each variable a register that is unoccupied in this range. To keep this step as simple as possible, variables are not offloaded to the main memory. Instead, the process is aborted if there are insufficient registers, and offloading is left to the developer.

### 3.3 Software Pipelining

As already discussed, the processor architecture presented here aims for the highest possible throughput per resource, which implies a high utilization of all computing units. In general, VLIW architectures support this, but only if the dependencies between the instructions allow dense placement. Using the example of one benchmark, a program with 27 arithmetic and logical instructions and a critical path of 13 cycles, it can already be seen that even without considering congestion, utilization of the six computing units is upper limited by $\frac{27}{13 \cdot 6} \approx 35\%$.

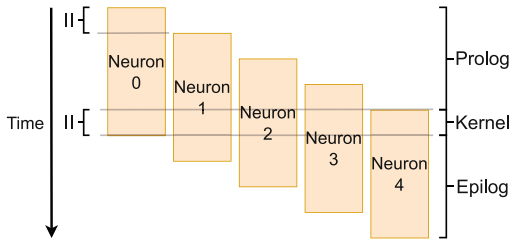


**Fig. 2.** Scheme of software pipelining: After splitting a loop, e.g., of neuron calculations, into prologue, kernel, and epilogue, several iterations are executed in parallel, whereby the kernel is repeated, and a new calculation is started every II cycles.

Improving this is generally difficult, but not in the case of simulating BNNs. Since each processor calculates a large number of neurons, the calculation of several neurons can be interleaved. As there are no dependencies between different neurons, latencies can be hidden, and schedule gaps filled.

In our case, it is even assumed that many neurons are based on the same model. This allows so-called software pipelining [13] to be used, in which the compiler splits the loop across all neurons into a prologue, a kernel, and an epilogue (cf. Fig. 2). With this method, the kernel still represents a loop with all the original instructions, but these now relate to different neurons. The advantage is that instructions not belonging to the calculation of the same neuron have no interdependencies and can, therefore, be placed freely. While the latency of a neuron calculation does not benefit from this approach, the throughput is increased because the kernel becomes more densely packed, reducing the distance between two iterations, i.e., the initiation interval (II). Prolog and epilog are only needed to fill and finalize the pipeline. An essential condition for the applicability of software pipelining is a sufficiently high number of available registers, as more data is now processed simultaneously.

**Extension of the Scheduler.** Since the software pipelining of a loop is another scheduling problem, the aim is to reuse the presented scheduling algorithm and its implementation with minimal changes. To this end, various approaches are compared in the following.

1. Since software pipelining of BNN computation involves executing instructions of different loop iterations, which are independent, an obvious approach is to delete all dependencies. The subsequent scheduling becomes almost trivial, as it is only constrained by resource dependencies. However, the prolog and epilog subsequently need to be fitted to the resulting kernel using a further algorithm yet to be developed. It is also possible that this schedule leads to the interlacing of an unnecessarily large number of loop iterations. This should be avoided, as more parallel iterations are associated with a higher register requirement and greater overall latency.
2. On the other hand, the solution with presumably the least implementation effort is to duplicate the source code before starting the scheduler, for example, in the form of a preprocessor, and to apply the unmodified scheduler to it. This corresponds to loop unrolling. The major disadvantage here is that the runtime of the scheduler can be increased dramatically due to a significantly larger problem size. Since a scheduling algorithm based on back-tracking is used here, the runtime increases exponentially with the problem size.
3. A modification of the modulo scheduling [20] represents the use of the original scheduling algorithm with the following adaptations:
   (a) An eligible II is selected iteratively or based on a heuristic.
   (b) In the scheduling step 5(b) for checking resource dependencies, not only the units at the placement address $addr_{\text{placed}}$ of an instruction are checked, but also the units of all alias addresses (i.e., $addr_{\text{placed}} + x * II, x \in \mathbb{R}$) are marked occupied.
   (c) Some additional data dependencies must be added to ensure a correct sequence of data accesses even after the code is rolled up to a loop. For example, variables can be written in the lower part of a loop before they are read in the upper part. In this case, an overlap would lead to the use of a value of a different neuron.
   (d) Finally, as illustrated in Fig. 2, the resulting schedule for one neuron is placed next to each other several times, shifted by II, until the obtained new schedule can be divided into a kernel of length II, as well as a prologue and an epilogue. The additional check of all alias addresses in 3(b) ensures that no resource can be reused despite parallelization.

The last proposed approach combines the advantages of the previous two: After applying these minimal adaptations to the original algorithm, dense software pipelined schedules with a small number of interlaced iterations can now successfully be found without increasing the problem size.

## 4 Results and Discussion

In this chapter, different configurations of nAIxt are first compared with each other. Then, the best configuration is compared to state-of-the-art solutions.

### 4.1 Benchmarks

For the following comparisons, four benchmarks are used, all calculating a leaky integrate-and-fire neuron model. While the first one uses CUBA synapses and an exact integration method [21], the others use COBA synapses and three different numerical integration methods *forward Euler method* and *Runge-Kutta methods* of order three (RK3) and four (RK4). The CUBA model corresponds to the NEST:: [7] neuron model *iaf_psc_exp*, while the COBA models correspond to the NEST:: neuron model *iaf_cond_exp* with a customized solver.

A numerical solver is required since the COBA synapse model does not allow for analytical solutions. In the following, three different solvers are evaluated because of significant differences in accuracy, stability, and computation effort. While the explicit Euler method is the simplest, the Runge-Kutta method represents a good compromise and is, therefore, often used in the calculation of neuron models. Moreover, it can be adjusted in the number of order and thus in the precision and computation effort. In our case, the highest order considered was set to 4, as in this case, the computational effort required to calculate the Runge-Kutta method already exceeds that of the rest of the neuron model, and the entire program is based almost entirely on floating-point additions. A significant change through higher orders is therefore not expected.

The memory accesses for neuron variables and excitations by spikes, arriving as fixed-point numbers, thus requiring conversion to floating-point, were added to all benchmarks. In the case of the RISC-V and MicroBlaze processors, the compilation was carried out without debug information and optimization enabled (`-O2`). For the two Runge-Kutta benchmarks on nAIxt, the original neuron calculation loop had to be split into two parts, each running through software pipelining independently. This step became necessary due to insufficient registers and can reduce optimization potentials.

### 4.2 Design Space Exploration

To optimize nAIxt, design changes are applied to the initial architecture shown in Fig. 1 and the AT efficiency is computed as $\eta_{AT} = \frac{F_{\max}}{II \cdot \#LUTs}$ for each case. The changes include a variation of the register file size and a gradual increase in the number of floating-point adders and multipliers (cf. Fig. 3). To cover possible bottlenecks and downsizing, the number of register file in- and output ports was also examined in the two edge cases - in the case of the maximum floating point units to 12, respectively six, and in the case of only one unit each, to four, respectively two. The initial architecture from Fig. 1 clearly represents the optimal solution of this comparison for all benchmarks. However, its margin reduces for larger benchmarks. The number of registers is also already optimal at 16–24 is preferable only in some cases of larger configurations. In addition to the experiments shown with 16 and 24 registers, 12 and 32 were also examined. However, these were not plotted to enhance clarity as they performed significantly worse. In the following sections, the initial architecture is used.

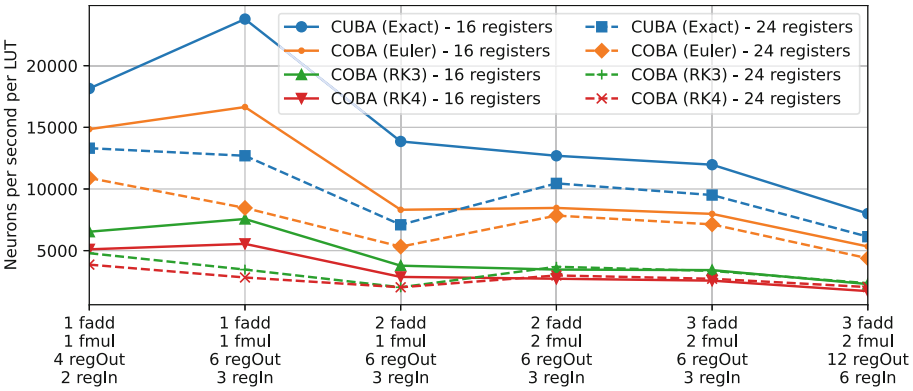


**Fig. 3.** AT efficiencies for benchmarks on different nAIxt configurations, comprising different numbers of floating-point units, register file in-, and output ports.

### 4.3 Comparison

In this section, the AT efficiency of the derived nAIxt architecture will be compared to that of other freely programmable processors. These are firstly the MicroBlaze using two optimization targets *Area* and *Performance*, and secondly four different RISC-V open source implementations *Rocket Chip* [3], *CVA6* [25] and *Berkeley Out-of-Order Machine (BOOM)* [26] in the configurations *Large* and *Mega* from the chipyard [1] v1.11.0 stable release. All processors contain floating-point units and hardware multipliers. Table 1 shows the implementation results of all processors in terms of occupied LUTs and maximum frequency. The synthesis results do not contain the main memory and memory controller. Here and for all following results, the synthesis tool *Xilinx Vivado™ 2022.1* was used, and the Xilinx ZCU106 evaluation board was selected as the target FPGA.

**Table 1.** Full synthesis results of nAIxt, MicroBlaze (MB) and RISC-V processors under Xilinx Vivado™2022.1 for a ZCU106 FPGA board.

| Processor | nAIxt | MB (A) | MB (P) | Rocket | CVA6 | BOOM (L) | BOOM (M) |
|---|---|---|---|---|---|---|---|
| Max. Freq. [MHz] | 140 | 380 | 350 | 154 | 117 | 119 | 93 |
| Occupied LUTs | 841 | 1730 | 2788 | 39600 | 41143 | 226124 | 402389 |

To obtain a quasi-optimal design point as a reference, a dedicated digital circuit was created for each benchmark with the help of *AMD Vitis™ HLS*. For this purpose, full pipelining was enforced using the pragma `II=1`. The target frequency of 200 MHz was chosen as a compromise in the mid-range between the processors. All neuron and synapse parameters, like membrane capacitance or threshold voltage, were implemented as constants and are not changeable at runtime. The evaluation of the occupied LUTs is based on a complete synthesis,

as HLS synthesis only provides a coarse estimate. Table 2 shows the resulting sizes of these reference circuits.

**Table 2.** Implementation results of the dedicated digital circuits for all benchmarks using HLS and Xilinx Vivado 2022.1 for implementation on the target board ZCU106.

| Benchmark | CUBA (Exact) | COBA (Euler) | COBA (RK3) | COBA (RK4) |
|---|---|---|---|---|
| Occupied LUTs | 4507 | 5066 | 11602 | 13573 |

To compare the AT efficiency of the processors, the II of the neuron calculation is first captured cycle accurately. In the case of nAIxt and the RISC-V processors, this was done with the help of a Verilog simulation in *Verilator 5.020*. In the case of the MicroBlaze, a measurement was carried out on an FPGA using an (ILA) IP core, which observes a certain memory access generated in each neuron calculation.

**Table 3.** Schedule of software pipelined kernel of the **COBA (Euler)** benchmark for nAIxt. White fields indicate nop operations, others the corresponding neuron index.

| Cycle | Unit | | | | | |
|---|---|---|---|---|---|---|
| | Move | Int. Arith. | Int. Mul. | FP Conv. | FP Add. | FP Mul. |
| 0 | i | | i-1 | | i-1 | i-1 |
| 1 | i | i | | i-2 | i-1 | i-1 |
| 2 | i | i | i-2 | | | i-1 |
| 3 | | | i-1 | | i-2 | |
| 4 | | i-1 | | | i-2 | |
| 5 | | | i-2 | | i-1 | i-2 |
| 6 | i-2 | | | i-2 | i-1 | i-2 |
| 7 | | | | | i-1 | i-2 |
| 8 | i | i | | | i-2 | |
| 9 | | i-1 | | | i-1 | |

Given the kernel schedule of the **COBA (Euler)** benchmark for nAIxt in Table 3, it is apparent that floating-point additions are the limiting instructions of this architecture. This also applies to the other benchmarks. Since the exploration in Sect. 4.2 did not determine any benefit for adding extra floating-point units, this approach is not pursued. Consequently, the number of floating-point additions per benchmark can be considered a minimum bound for the II.

Table 4 shows the obtained II for the four different benchmarks on all processors. The II of the digital reference circuit is not listed, as it was forced to

**Table 4.** Measured II of all benchmarks executed on each processor between two neuron computations and the number of required floating-point additions as reference.

| Benchmark | CUBA (Exact) | COBA (Euler) | COBA (RK3) | COBA (RK4) |
|---|---|---|---|---|
| No. of FP Additions | 6 | 9 | 21 | 27 |
| MicroBlaze (Area) | 241 | 255 | 443 | 490 |
| MicroBlaze (Perf.) | 189 | 201 | 358 | 398 |
| Rocket Chip | 51 | 65 | 131 | 142 |
| CVA6 | 79 | 85 | 151 | 166 |
| Large BOOM | 16 | 19 | 67 | 81 |
| Mega BOOM | 12 | 15 | 54 | 65 |
| **nAIxt** | **7** | **10** | **23** | **30** |

be always one. As expected, the larger RISC-V processors achieve significantly lower IIs, i.e., higher throughputs than their smaller counterparts or the MicroBlaze processors. However, despite its small size, nAIxt can outperform even the BOOM in *Mega* configuration by a factor of around two. Comparing the II to the number of floating-point additions indicates that nAIxt works almost optimally regarding its number of floating-point units.

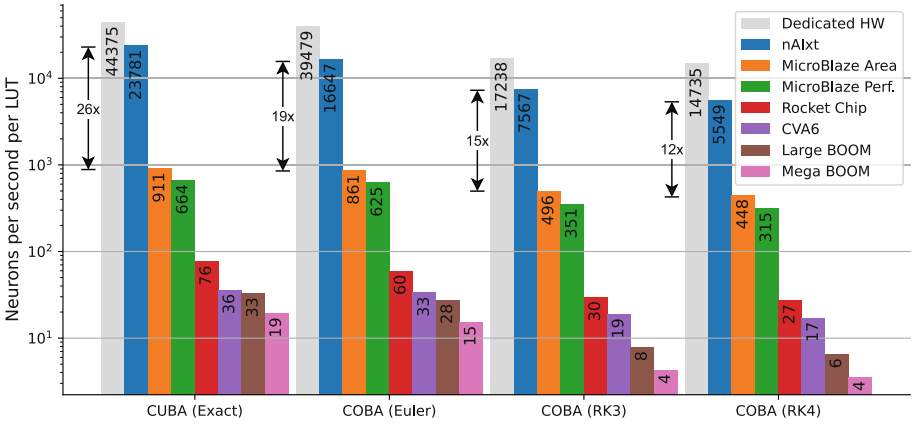


**Fig. 4.** Performance comparison of all processors in relation to their size based on four neuron models as benchmarks.

Finally, Fig. 4 shows the comparison in terms of AT efficiency. Now, a different picture emerges for larger RISC-V and MicroBlaze processors. The higher throughput cannot compensate for the higher resource requirements. The instantiation of many low-performance processors would therefore be the better choice in systems for BNN simulations. Interestingly, the AT efficiency of the two MicroBlaze processors exceeds the one of the RISC-V processors by one to

two orders of magnitude, presumably due to their optimization for FPGAs. As expected, the dedicated digital circuits represent the fastest solution. However, the number of pipeline stages to reach a frequency of 200 MHz is significant here (136 in case of **COBA (RK4)**) and, therefore, may not meet possible restrictions depending on the use case. Compared to this reference, nAIxt is two to three times worse, indicating an opportunity for further optimization. Comparing only programmable solutions, both RISC-V and MicroBlaze processors are not competitive with nAIxt, with an AT efficiency of two to three orders lower.

### 4.4 Analysis and Optimization

Since one way of optimizing the processor is to reduce its resource requirements, Fig. 5 illustrates the breakdown of LUTs between the functional units **register file**, **floating-point adder**, **floating-point multiplier** and **integer multiplier**. All other units are included in **others**. It is worth noting that the floating-point units, which are mainly responsible for the calculations, only occupy less than a quarter of the resources. Instead, a large part of the resources (here: register file) is required for the multiplexers, i.e., the data management between ALU and register file. This is indispensable in superscalar architectures, as large amounts of data need to be accessed selectively and in parallel by the ALU. However, one optimization approach could be reducing supported sources and/or sinks, for example, by giving specific units access to only a subset of all registers. This could theoretically increase the throughput per resource at the cost of a higher scheduling complexity and, thus, longer compiler runtimes.

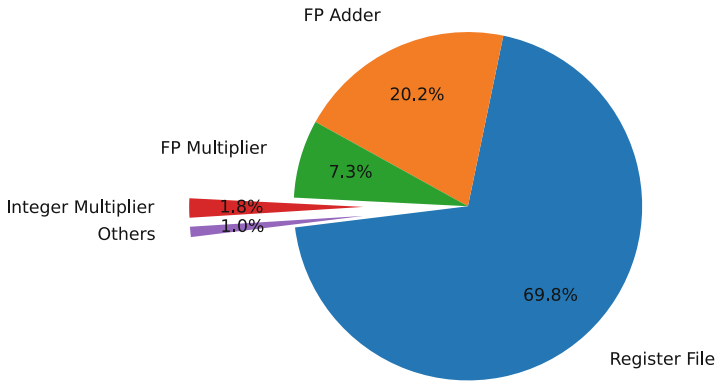


**Fig. 5.** Breakdown of utilized LUTs over main parts of nAIxt. Due to the flattening of the synthesis, the "Register File" section also contains the multiplexers of the ALU.

## 5 Conclusion

This work demonstrates that conventional general-purpose CPUs are well suited for the computation of neuron models but at the expense of high resource require-

ments. If high parallelization capability is combined with jump avoidance, software pipelining allows an enormous leap in performance even for basic VLIW architectures with extremely low resource requirements. Exactly these conditions are fulfilled in large digital systems for the simulation of BNNs and can, therefore, be exploited by processor architectures of the kind presented here.

Contrary to the often problematic utilization of a VLIW architecture in the past, even rudimentary scheduling algorithms can deliver good results for the very regular and mathematical structures of neuron models, as shown here.

The presented solution achieves 200 to 1800 times higher AT efficiencies than open source RISC-V processors while maintaining full programmability. This brings it close to highly optimized dedicated circuits and thus facilitates the transition of BNNs simulators from reconfigurable hardware to integrated circuits, still supporting rapid adaptation to future findings in neuroscience.

## References

1. Amid, A., et al.: Chipyard: integrated design, simulation, and implementation framework for custom SoCs. IEEE Micro **40**(4), 10–21 (2020). https://doi.org/10.1109/MM.2020.2996616
2. Andalman, A.S., et al.: Neuronal dynamics regulating brain and behavioral state transitions. Cell **177**(4), 970–985 (2019)
3. Asanovic, Ket al.: The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 **4**, 6–2 (2016)
4. Cheung, K., Schultz, S.R., Luk, W.: NeuroFlow: a general purpose spiking neural network simulation platform using customizable processors. Front. Neurosci. **9**, 516 (2016)
5. Frenkel, C., Legat, J.D., Bol, D.: MorphIC: a 65-nm 738k-Synapse/mm2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning. IEEE Trans. Biomed. Circuits Syst. **13**(5), 999–1010 (2019)
6. Furber, S.B., Galluppi, F., Temple, S., Plana, L.A.: The SpiNNaker project. Proc. IEEE **102**(5), 652–665 (2014)
7. Gewaltig, M.O., Diesmann, M.: NEST (NEural Simulation Tool). Scholarpedia **2**(4), 1430 (2007)
8. Heittmann, A., et al.: Simulating the cortical microcircuit significantly faster than real time on the IBM INC-3000 neural supercomputer. Front. Neurosci. **15**, 728460 (2022)
9. Kauth, K., Stadtmann, T., Brandhofer, R., Sobhani, V., Gemmeke, T.: Communication architecture enabling 100x accelerated simulation of biological neural networks. In: Proceedings of the Workshop on System-Level Interconnect: Problems and Pathfinding Workshop, SLIP 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3414622.3431909
10. Kauth, K., Stadtmann, T., Sobhani, V., Gemmeke, T.: neuroAIx: FPGA cluster for reproducible and accelerated neuroscience simulations of SNNs. In: 2023 IEEE Nordic Circuits and Systems Conference (NorCAS), pp. 1–7 (2023). https://doi.org/10.1109/NorCAS58970.2023.10305473
11. Knight, J.C., Nowotny, T.: Gpus outperform current hpc and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. Front. Neurosci. **12**, 941 (2018)

12. Kudithipudi, D., et al.: Biological underpinnings for lifelong learning machines. Nat. Mach. Intell. **4**(3), 196–210 (2022)
13. Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. SIGPLAN Not. **23**(7), 318–328 (1988). https://doi.org/10.1145/960116.54022
14. Moore, S.W., Fox, P.J., Marsh, S.J., Markettos, A.T., Mujumdar, A.: Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 133–140. IEEE (2012)
15. Orchard, G., et al.: Efficient neuromorphic signal processing with Loihi 2. In: 2021 IEEE Workshop on Signal Processing Systems (SiPS), pp. 254–259. IEEE (2021)
16. O'reilly, R.C., Munakata, Y.: Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain. MIT press, Cambridge (2000)
17. Pandarinath, C., et al.: Inferring single-trial neural population dynamics using sequential auto-encoders. Nat. Methods **15**(10), 805–815 (2018)
18. Pani, D., et al.: An FPGA platform for real-time simulation of spiking neuronal networks. Front. Neurosci. **11**, 90 (2017)
19. Park, J., Ha, S., Yu, T., Neftci, E., Cauwenberghs, G.: A 65k-neuron 73-Mevents/s 22-pJ/event asynchronous micro-pipelined integrate-and-fire array transceiver. In: IEEE Biomedical Circuits and Systems Conference (BioCAS), pp. 675–678. IEEE (2014)
20. Rau, B.R., Glaeser, C.D.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. ACM SIGMICRO Newsl. **12**(4), 183–198 (1981)
21. Rotter, S., Diesmann, M.: Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. Biol. Cybernet. **81**(5–6), 381–402 (1999)
22. Trensch, G., Morrison, A.: A system-on-chip based hybrid neuromorphic compute node architecture for reproducible hyper-real-time simulations of spiking neural networks. Front. Neuroinf. **16**, 884033 (2022)
23. Vogels, T.P., Abbott, L.F.: Signal propagation and logic gating in networks of integrate-and-fire neurons. J. Neurosci. **25**(46), 10786–10795 (2005)
24. Yeh, Z.W., et al.: Poppins: a population-based digital spiking neuromorphic processor with integer quadratic integrate-and-fire neurons. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5 (2021)
25. Zaruba, F., Benini, L.: The cost of application-class processing: energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm fdsoi technology. IEEE Trans. Very Large Scale Integrat. (VLSI) Syst. **27**(11), 2629–2640 (2019)
26. Zhao, J., Korpan, B., Gonzalez, A., Asanovic, K.: Sonicboom: the 3rd generation berkeley out-of-order machine. In: Fourth Workshop on Computer Architecture Research with RISC-V, vol. 5 (2020)