IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 October 2023, accepted 20 October 2023, date of publication 25 October 2023, date of current version 1 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3327517

RESEARCH ARTICLE

An 8-bit Single Perceptron Processing Unit for Tiny Machine Learning Applications

MARCO CREPALDI, (Member, IEEE), MIRCO DI SALVO, AND ANDREA MERELLO
Electronic Design Laboratory, Istituto Italiano di Tecnologia, 16152 Genoa, Italy
Corresponding author: Marco Crepaldi (marco.crepaldi@iit.it)

ABSTRACT We present a tiny MultiLayer Perceptron (MLP) accelerator named Single Perceptron Linear Vector Processor (SPLVP) that aims at extending the capabilities of limited resources MCUs, enabling inference time speedup and main CPU off-load. It is based on a single perceptron hardware unit, enhanced with an additional accumulation input and scaling features, that is sequentially scheduled to cover all the nodes of the network. The accelerator supports both linear and Rectified Linear Unit (ReLU) activation and its firmware can be generated from 8-bit `tflite` quantized models. We also present a complete design toolchain that encompasses supervised learning, compilation, assembly, simulation, and device programming. The hardware support for extra accumulation input and scaling, together with the processor memory partitioning, are the key features that enable significant speedups. By solving a toy recognition problem based on image data captured from an infra-red camera, measurements show that the execution speed of SPLVP at 80 MHz outperforms an ARM Cortex-M4 STM32L476 microcontroller by a factor of 9.2 when the same ANN is translated to MCU code using the STM CubeMX-Ai converter at the same clock frequency. SPLVP is synthesized on a low-cost and gate-count Cyclone 10 LP FPGA resulting in an 18% logic and 77% memory occupation. The SPLVP assembly code can be directly converted into a VHDL description that directly hardcodes the ANN. The execution speed of an ANN model for Iris classification, fully synthesized, improves by a factor of 209 compared to firmware execution on the MCU. To verify the operation of SPLVP and its design framework, we have designed various tiny Machine Learning (ML) classifiers, for which we briefly discuss the obtained performance and the preprocessing techniques used. Across all these classifiers, the obtained speedup compared to the STM32 is 8.3–14.9 ×.

INDEX TERMS Neural processing unit, multilayer perceptron, single perceptron linear vector processor, fully connected neural network, FPGA, compiler, design toolchain, MCU, tiny machine learning.

I. INTRODUCTION

Artificial Neural Networks (ANNs) have revolutionized the way computers can help to solve problems, and are considered fundamental resources in terms of modeling and prediction capabilities. They are exponentially becoming more pervasive: the availability of software libraries developed by big technology companies and organizations [1], [2], [3], [4], [5], enables ANN models to be, more or less automatically, compiled and deployed for a wide variety of hardware. Such hardware encompasses powerful computing centers, custom accelerators, or Edge devices in which computing problems are partially addressed locally, where data are produced, and partially deferred and off-loaded in the cloud. On the low end, microcontrollers are particularly interesting [6], because ANNs can be potentially brought to limited hardware resources applications. This emerging but already vast and consolidated field is known as tiny Machine Learning (tinyML) [7] and paved the way for several research problems that are all currently open. Models developed in the past are indeed complex and require a very large number of both coefficients (e.g., on the order of millions for typical image classification problems, such as ImageNet [8]), and operations, and are typically trained using numbers represented in floating-point format. Substantial advantages in terms of hardware footprint, power

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

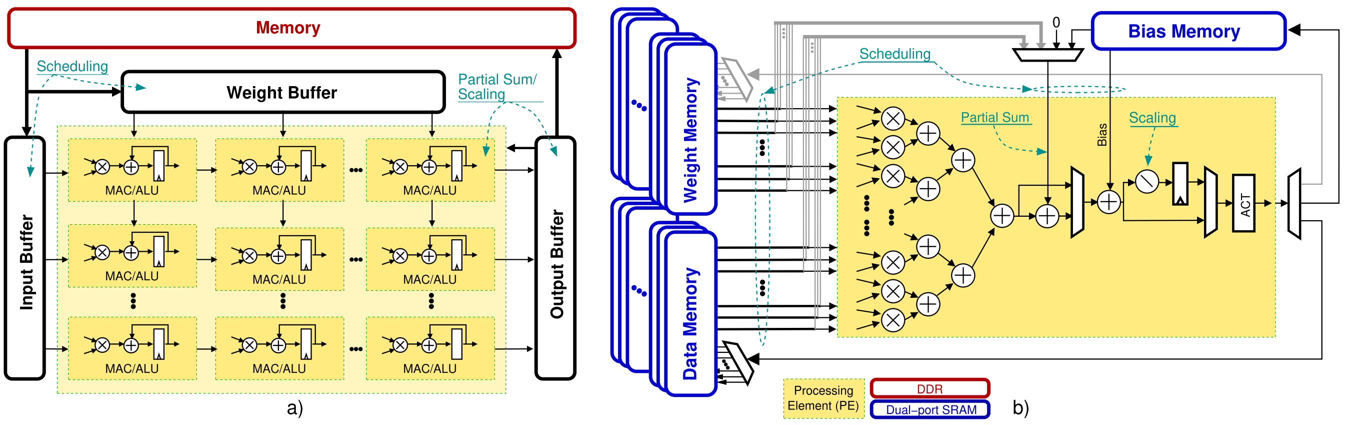


FIGURE 1. **a) Generic systolic array of PEs and b) SPLVP architecture. In systolic arrays, the buffers perform local data storage and operation scheduling and they are interfaced with a central Dynamic Random-Access Memory (DRAM), which is power-hungry. The partial sums, activations, and scalings can be implemented in the PEs or the output buffer. In SPLVP, the memory is partitioned into dual-port Static Random-Access Memories (SRAMs): partial sums can be any of the memory output, write-back can be implemented to any memory array element and a single PE concentrator is used to decrease complexity and save hardware resources. In this work, the toolchain considers as partial sum input the bias memory and implements write-back only to the bias and data memory, although all memories are wired and supported by the hardware.**

consumption, and memory requirements can be obtained with mixed-precision arithmetic, a fundamental topic when scalability of ML models becomes indispensable for both inference and training aspects [9], [10], [11], [12]. However, TinyML applications are highly resource-constrained and integer arithmetic implementation can result in even lower hardware resources and memory footprint compared to floating-point [13]. For instance, applications involving environmental sensing, and distributed sensor networks in general require lightweight ANNs, at most hundred of KB sized [14]. Current research efforts in microcontroller applications indeed focus on both model size reduction [15], and ANN quantization and pruning [16], which can result in negligible performance degradation.

The execution time of an ANN on a MicroController Unit (MCU) has to deal with the general-purpose architecture of their Central Processing Units (CPUs). Without specialized hardware, an embedded systems designer must sooner or later deal with the inherent limitations in real-time inference of ANNs posed by these CPUs. By construction, they typically do not embed a high level of arithmetic parallelism and rely on an inappropriate memory access mechanism. To enable low-power embedded systems to efficiently run ML models, two approaches are typically possible: i) the model conversion into optimized MCU machine code, and ii) the use of ad-hoc SoCs that embed an MCU with a specialized ML co-processor interfaced to the system bus. These approaches, however, do not enable the empowering of existing MCU-based designs targeting low-power sensor networks with ML capabilities: the converted ML models still perform with high latency due to their architectural limitations, and the ML models compiled for ad-hoc embedded accelerators cannot be directly ported to other platforms. In addition, the Google Edge Tensor Processing Unit (TPU) platform, designed to handle large models in Edge computing applications, needs a USB interface which is not always available on MCUs. It also requires a significant power budget [17], making it hard to use in battery-powered and resource-constrained systems such as a sensor network. In general, complex accelerators designed to support large Convolutional Neural Networks (CNNs) typically provide substantial latency for very small models because the internal logic is typically underutilized; moreover, they require interfacing with middle-end processors and operating systems. For instance, in the MAX78000 SoC, convolutional and linear filters with input data size of 16×16 provide a latency of $\sim 75\,\mu$s, irrespective of filter size, while a single two-dimensional convolutional layer with four output channels requires $\sim 150\,\mu$s, irrespective of the number of input channels [18]. Similarly, the Google Edge TPU platform suffers from extreme underutilization of its Processing Elements (PEs) and inefficient sequential scheduling of Fully Connected (FC) layers [19]. These aspects outline that a low-complexity specific solution providing tiny ANN models hardware acceleration in resource-constrained MCU applications can be advantageous.

In this paper, we present a tiny accelerator with the associated ANN design toolchain that fits a low-power and low-end FPGA, that is powerful enough to outperform a microcontroller clocked at the same frequency and can be interfaced with any MCU to provide inference acceleration. We name this device Single Perceptron Linear Vector Processor (SPLVP) given the presence of a single perceptron in its Arithmetic Logic Unit (ALU), enhanced with additional accumulation and scaling capabilities. SPLVP includes a parallel Input/Output (I/O) interface to implement programming and data streaming from any external MCU. The SPLVP prototype introduced in Sec. III has been implemented on a 10CL025 FPGA device and can run TensorFlow Lite (`tflite`) models, thanks to a customized toolchain detailed in Sec. IV. The unique contributions of this work can be summarized as follows: i) the accelerator has very low logic utilization, but this notwithstanding it enables an average of 9.6 × speedup compared to a STM32L4 MCU clocked at the same frequency, ii) it has an enhanced single perceptron computing unit that is scheduled to cover all the nodes of a FC ANN thanks to a specific internal memory partitioning, while in contrast, typical accelerators implement a systolic array of PEs (see Fig. 1) [20], [21], iii) it can be interfaced to any MCU thanks to a specific interface, whose pins can be directly driven by MCU General Purpose I/Os (GPIOs), iv) the associated portable toolchain enables Multi-Layer Perceptron (MLP) network design, compilation, assembly and simulation directly from TensorFlow (TF) and `tflite` though Post-Training Quantization (PTQ), and v) its assembly code, when considered as an intermediate representation of the ANN, can be automatically converted into hardcoded VHDL, thus enabling another order of magnitude inference latency reduction. Sec. V validates the processor prototype and the design toolchain using custom and publicly available datasets and discusses the performance of the obtained ANN models compared to an STM32L476 MCU running optimized code generated using its CUbeMX.Ai tool. There, we further position our work by comparing the inference performance of the obtained models with other acceleration solutions targeting power-constrained systems and the Google Coral platform. We outline further research steps and potentials in Sec. VI.

II. RELATED WORK

The implementation and deployment of ANNs in microcontrollers are major research trends. Manufacturers are releasing software capable of converting ANN models obtained using ML libraries into optimized machine code for their MCUs. A well-known example is STM32 CubeMX.Ai, which processes ANN models in various formats and outputs the microcontroller network implementation [22]. Semiconductor and software companies are also making significant efforts to optimize tinyML model execution from combined software and hardware perspectives. They are addressing the problem from both a compiler viewpoint [23], and by devising specific Neural Processing Units (NPUs) co-processors integrated into Application-Specific Integrated Circuits (ASICs) with a main CPU core [24]. A clear example is the MAX78000 microcontroller [25], which integrates custom hardware accelerators (including other co-processors), directly connected to the system bus of a standard CPU core. Another example is given by a class of SoCs integrating multiple dual-core 64-bit RISC-V processors with dedicated audio and CNN accelerators [26]. A different approach regards the use of Single Instruction Multiple Data (SIMD) processing in the main CPU by enabling up to eight parallel Multiply And Accumulate (MAC) 8-bit operations as in the Cortex-M55, for a 20% performance advantage compared to a Cortex-M4 [27]. Furthermore, another approach regards a specialized SoC that includes RISC-V processors and a dedicated FPGA that implements ad-hoc and reconfigurable accelerators supporting CNNs, MLPs, and basic operators [28]. These systems are supplied with their associated dedicated model design, compilation, and deployment toolchains. The ML models obtained with these toolchains, however, are platform-specific and cannot be used with other MCUs.

Deep Neural Network (DNN) FPGA accelerators have been extensively proposed to solve more or less generic ANN acceleration problems starting from power budgets (except rare cases) above 1/10 Watt [29]. While their vast majority speeds up CNNs, interestingly, ANN accelerators are very diverse in terms of supported quantization, which can range from `int1` to `float64` across all power ranges. The reported data show that the `int8` quantization is very prominent [29]. However, only very few of them are designed for applications on the Edge, and none of them are for direct interfacing with MCUs, that are capable of delivering only a few hundred Million OPerations per second (MOPs/s) [30]. Based on a recent survey [20], until 2022 very few non-spiking FPGA accelerators targeting low-power applications have been published, two of them designed for CNNs and one for Transformer networks. In addition, an accelerator targeting tinyML on the Edge has been very recently proposed [31].

Among the large variety of possible ANN models, MLPs have been massively researched so far, but this notwithstanding they still represent a fundamental building block for the implementation of sophisticated ANNs, and are effective in low-bandwidth time-series recording from low-power sensors [30]. MLPs are a FC class of feedforward ANNs, important in many classification and regression problems [32]. Recent works show that ANN architectures based only on MLPs can be considered as a valid alternative to Vision Transformers and CNNs for image classification [33]. The idea of accelerating an MLP is not new (see, e.g., [34]): many systems have been extensively implemented in the past years on FPGA for real-time human activity classification or ElectroCardioGram (ECG) anomaly detection [35], [36] with hardcoded networks, or as programmable accelerators through Network on Chips (NoCs) of Reduced Instruction Set Computer (RISC) cores [37]. The system in [38] demonstrated synthesis-time configurable and run-time programmable accelerators, that exhibited significant speedups on FPGA compared to MicroBlaze and ARM processors, but without focusing on power consumption nor MCU interfaceability. Towards a full hardware hardcoding, other recent works investigate the hardware implementation of MLPs through their direct FPGA synthesis (see, e.g., [39]). The literature shows that feedforward MLP networks can be also synthesized for ultra-low power ASIC aiming at accelerating both inference and training, by co-designing hardware and software to determine bit precision with approximate computing [40].

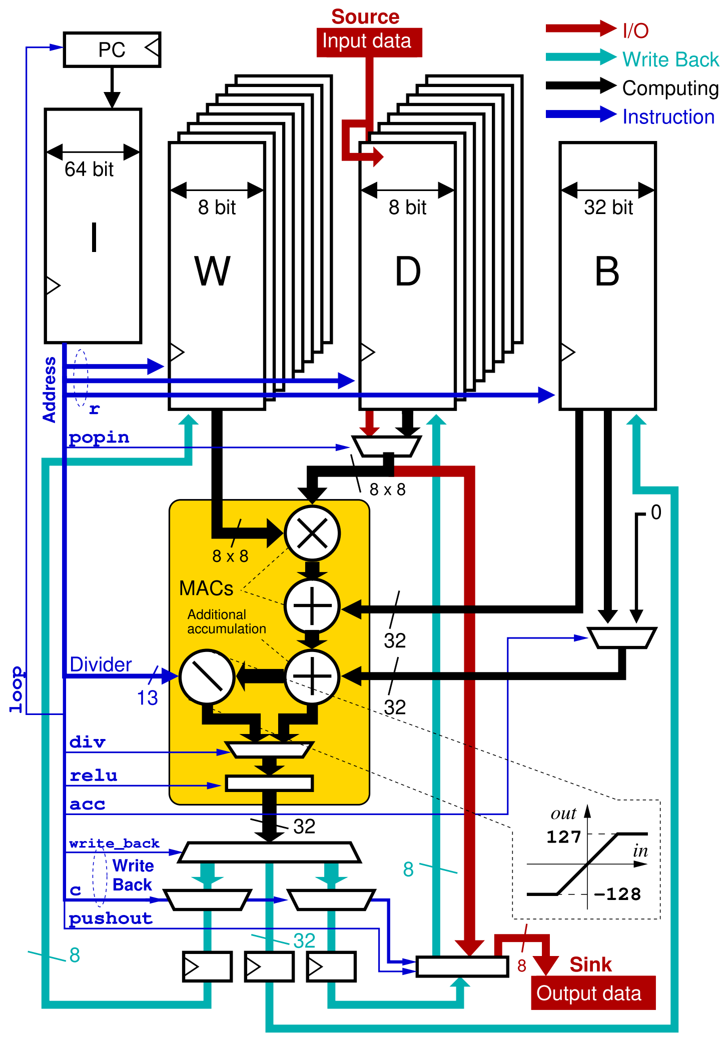


FIGURE 2. Simplified SPLVP architecture, with detail on the memory elements used to store FC layers coefficients and biases. The additional accumulation input here considers only bias memory.

Recently, in the context of tinyML, MLPs have been implemented on MCUs using a C-based framework, and extensively analyzed across Cortex-M4 and the recent RISC-V-based Parallel UltraLow Power platform (PULP) [30]. This analysis showed that an octa-core RI5CY can outperform a Cortex-M4 in three different applications by a factor 22 ×, 13.33 × and 7.5 × (the larger MLP, the better performance). This work suggests that MLPs are relevant in applications on the Edge of Internet-of-Things (IoT), where mW-powered MCUs are the most common computing engines.

The aspects outlined above motivated the development of this work, that regards the design of a low-complexity programmable solution to assist existing low-power MCUs in ML inference tasks, providing significant speedup.

III. SINGLE PERCEPTRON LINEAR VECTOR PROCESSOR

A. HIGH-LEVEL ARCHITECTURE AND ASSEMBLY

Fig. 2 shows the conceptual architecture of SPLVP. We consider ANN acceleration a combined local memory organization and computing problem. The main idea behind this processor is to dynamically schedule the execution of a specialized single perceptron PE of a given parallelism across a finely partitioned memory and by writing back values directly on it without further hierarchies. Conceptually, this simple processor inherits the simplicity of a One Instruction Set Computer (OISC) with absolute memory addressing and with a simple opcode (`com` for simplicity). The processor has four types of memories: the instruction memory `I`, the bias memory `B`, the data memory `D`, and the weight memory `W`. The instruction memory stores the address of the weight used for computing (in this implementation 11-bit), the address of the bias memory to be accumulated (11-bit width as well), the address of the data memory (7-bit), the options to be considered during executions, the memory element for write-back, and if required, the divisor used for scaling the accumulated data to `int8`. The instruction memory size is 2^{12} and it is 64-bit wide. The data memory stores the input layer data and the computed outputs of the internal layers, the weight memory stores the weights of the full MLP, and the bias memory stores biases, and it is also used as a temporary storage for intermediate accumations. Since the machine prototype presented here has a parallelism of eight, for computing efficiency purposes, both `D` and `W` memories are implemented using eight separate SRAMs to permit straightforward memorization of the output in a given column rather than rewriting specific bits of a larger word. The `B` memory (11-bit size) has a width of 32-bit, enough to implement accumulations and directly add the bias values according to the `tflite` quantization specification. To sequentially read the instructions from the `I` memory, the machine includes a program counter `PC` that can be reset to restart the execution of the program once the ANN is fully computed. As suggested in Fig. 1, the architecture is scalable and can be extended to support a higher perceptron parallelism (that is the number of simultaneous multiplications and additions executed per instruction) and larger internal memories, by consequently adjusting the width of the instructions.

The machine includes a single perceptron unit (box in the center) that works on eight weights and eight input elements from the `W` and `D` memory, performs their dot product, has two accumulation inputs, and adds all the elements to output a single value on a maximum width of 32-bit. The perceptron also has an internal divider that is used to scale down the accumulated output to the `int8` range, suitable for storage on the `D` memory. The machine operates with a type of processing similar to SIMD because the arithmetic operation is the same for all the elements of its vector inputs, but differently from a standard Vector Processing Unit (VPU) it combines them and outputs only a single value. SPLVP supports some optional parameters that are directly applied to the execution unit output, that slightly modify the computation datapath. A generic assembly line is given in Lst. 1.

```
com d(d_addr), w(w_addr), b(b_addr) act lin to m(r
    )(c) [opts]
```

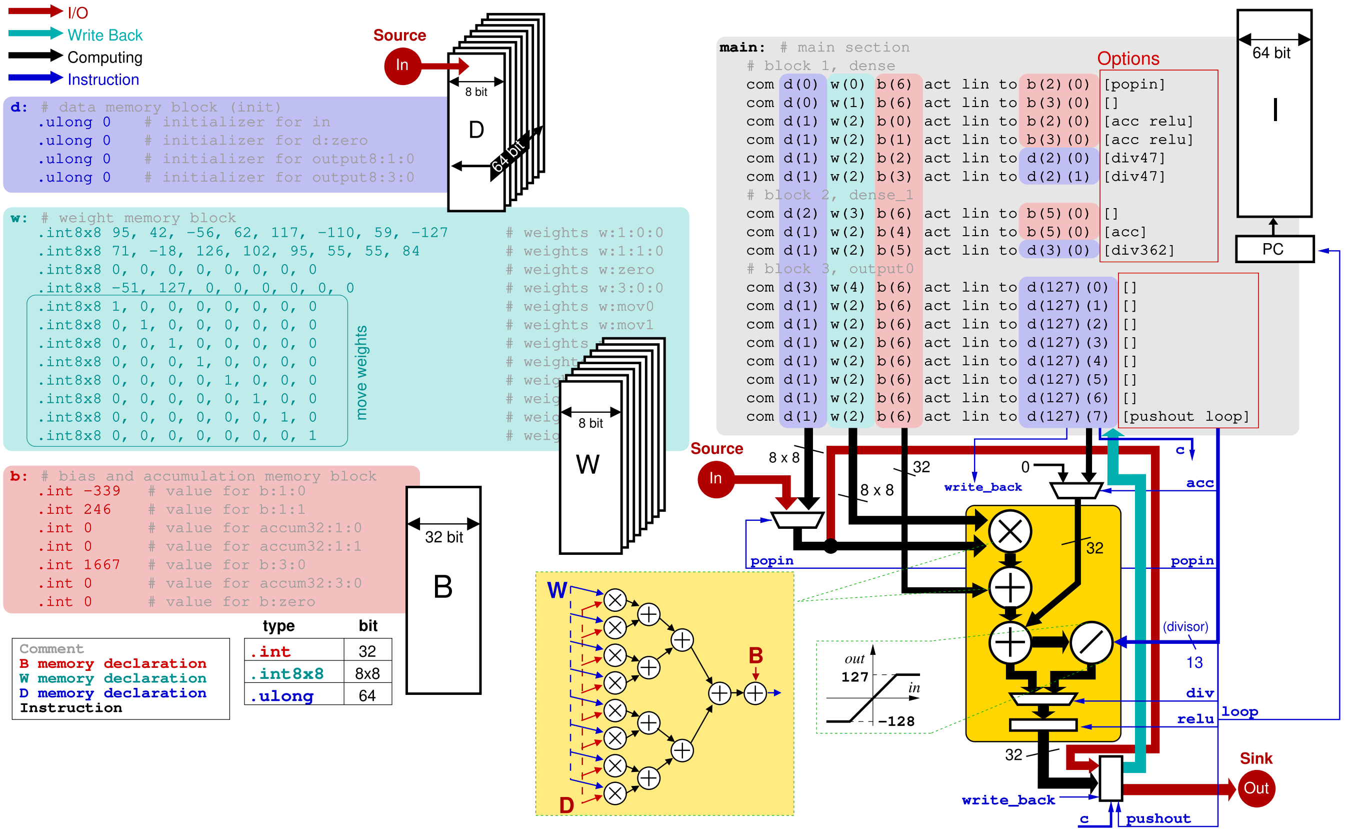
LISTING 1. Single perceptron vector processor generic assembly code.

This single line of code identifies the operation $\mathrm{w} \cdot \mathrm{d} + \mathrm{b} = \sum_{i=0}^{P-1} \mathrm{w}_i \mathrm{d}_i + \mathrm{b}$ and specifies to write-back the result to `m(r)(c)`, where $P = 8$, `d(d_addr)` and `w(w_addr)`, identify the content of `D` and `W` memory rows (vectors), `b(b_addr)` identifies the content of the `B` memory at the specified addresses, `r` and `c` are row and column of the destination memory `m` which can be any of the `D`, `W` or `B` memories. For instance, the string `d(76)(5)` means that the fifth column of `D` memory at row 76 is written. If write-back occurs on memory `B`, which is organized in 32-bit words rather than in eight parallel 8-bit memories, then `c` must be zero. The destination memory is encoded with a specific `write_back` field in the opcode, presented later. The `opts` field is a list of options that are additionally applied to the current computation. The possible options are provided below.

- `acc` Specifies that the destination memory `m` is accumulated, i.e., $\mathrm{m(r)(c)} \leftarrow \sum_{i=0}^{P-1} \mathrm{w}_i \mathrm{d}_i + \mathrm{b} + \mathrm{m(r)(c)}$;
- `relu` Applies a Rectified Linear activation Unit (ReLU) on the obtained single perceptron output. The `relu` option, activates a combinational logic at the end of the datapath to null any negative value;
- `popin` Reads 8×8-bit data from the input source and stores it at address `d(0)`. When a `popin` option is included in the opcode, the current `Input data` from the source (which is 8×8-bit overall) is immediately multiplexed to the perceptron to enable computation without latency and stored at the next cycle at address `d(0)`;
- `pushout` Writes all the content of `d(127)` to the sink. It enforces the redirection of the output of the perceptron (relative to a particular column `c`) to the sink and forwards the remainder columns of the address `d(127)`;
- `div`v Implements a division by an integer value v (unsigned 13-bit, therefore enabling division by a maximum of $2^{13} - 1$) after the single perceptron execution. The `div` option causes the internal muxes to divert the data through a datapath which includes a saturated integer divider that is used to scale the result. The 13-bit scaling factor is specified in the instruction opcode. The integer division is implemented with rounding, that is, assuming X_d as dividend, X_D as divisor and Y as result, $Y = \frac{X_d + \mathrm{sign}(X_d)\frac{X_D}{2}}{X_D}$, where the function $\mathrm{sign}(x)$, is 1 for $x > 0$ and -1 otherwise;
- `loop` Resets the program counter to zero and restart inference.

As a write-back field is included in the instructions, together with the other fields `r` and `c`, the instruction itself is used to directly demultiplex the single perceptron output to the corresponding destination memory `w`, `b`, and `d` (with truncation if necessary for 8-bit memories). The architecture supports write-back to any of the `D`, `B`, and `W` memories, but as we here focus on inference acceleration, the compiler implements only write-back on `D` and `B` memories. The possibility of writing the `W` memory, however, would be fundamental to implementing online training, which is a useful and only partially explored feature in MLP accelerators [41]. Observe also that the presence of a specific option `acc` to enable accumulation on a particular memory element goes in favor of the implementation of Recurrent Neural Networks (RNN), where accumulated values do not need to be reset.

We have chosen to implement ReLU because it requires low resources, it is energy efficient compared to hyperbolic tangent [42], and it is a first-to-go choice [43]. We have investigated the implementation of an approximation of the hyperbolic tangent $\tanh(x)$ in the form of LUTs, by assuming input and output in the `int8` range. Results show that



```
d: # data memory block (init)
   .ulong 0    # initializer for in
   .ulong 0    # initializer for d:zero
   .ulong 0    # initializer for output8:1:0
   .ulong 0    # initializer for output8:3:0

w: # weight memory block
   .int8x8 95, 42, -56, 62, 117, -110, 59, -127      # weights w:1:0:0
   .int8x8 71, -18, 126, 102, 95, 55, 55, 84         # weights w:1:1:0
   .int8x8 0, 0, 0, 0, 0, 0, 0, 0                    # weights w:zero
   .int8x8 -51, 127, 0, 0, 0, 0, 0, 0                # weights w:3:0:0
   .int8x8 1, 0, 0, 0, 0, 0, 0, 0                    # weights w:mov0
   .int8x8 0, 1, 0, 0, 0, 0, 0, 0                    # weights w:mov1
   .int8x8 0, 0, 1, 0, 0, 0, 0, 0                    # weights
   .int8x8 0, 0, 0, 1, 0, 0, 0, 0                    # weight
   .int8x8 0, 0, 0, 0, 1, 0, 0, 0                    # weig
   .int8x8 0, 0, 0, 0, 0, 1, 0, 0                    # weig
   .int8x8 0, 0, 0, 0, 0, 0, 1, 0                    # weig
   .int8x8 0, 0, 0, 0, 0, 0, 0, 1                    # weig

b: # bias and accumulation memory block
   .int -339    # value for b:1:0
   .int 246     # value for b:1:1
   .int 0       # value for accum32:1:0
   .int 0       # value for accum32:1:1
   .int 1667    # value for b:3:0
   .int 0       # value for accum32:3:0
   .int 0       # value for b:zero

main: # main section
   # block 1, dense
   com d(0) w(0) b(6) act lin to b(2)(0) [popin]
   com d(0) w(1) b(6) act lin to b(3)(0) []
   com d(1) w(2) b(0) act lin to b(2)(0) [acc relu]
   com d(1) w(2) b(1) act lin to b(3)(0) [acc relu]
   com d(1) w(2) b(2) act lin to d(2)(0) [div47]
   com d(1) w(2) b(3) act lin to d(2)(1) [div47]
   # block 2, dense_1
   com d(2) w(3) b(6) act lin to b(5)(0) []
   com d(1) w(2) b(4) act lin to b(5)(0) [acc]
   com d(1) w(2) b(5) act lin to d(3)(0) [div362]
   # block 3, output0
   com d(3) w(4) b(6) act lin to d(127)(0) []
   com d(1) w(2) b(6) act lin to d(127)(1) []
   com d(1) w(2) b(6) act lin to d(127)(2) []
   com d(1) w(2) b(6) act lin to d(127)(3) []
   com d(1) w(2) b(6) act lin to d(127)(4) []
   com d(1) w(2) b(6) act lin to d(127)(5) []
   com d(1) w(2) b(6) act lin to d(127)(6) []
   com d(1) w(2) b(6) act lin to d(127)(7) [pushout loop]
```

```
Comment
B memory declaration
W memory declaration
D memory declaration
Instruction
```

type	bit
.int	32
.int8x8	8x8
.ulong	64

FIGURE 3. SPLVP assembly with associated conceptual architecture elements during code execution.

the number of LUTs of tanh(x) depends on the considered function domain across x, in particular requiring 127, 117, 103, and 94 LUTs, for the intervals $[-1, 1]$, $[-2, 2]$, $[-3, 3]$ and $[-4, 4]$, respectively. The MAE of our approximation across all cases is always below 0.26. On the other hand, ReLU requires only 7 LUTs. In light of these results and its energy efficiency compared to hyperbolic tangent, we have considered ReLU as a viable choice for the implementation of our low-complexity and resources accelerator.

Fig. 3 shows an example of a complete SPLVP assembly file and illustrates how instructions internally work. Comments start with the character # and the text file is divided into four regions which identify and declare the content of a memory element in the architecture. The `d:` memory region declares the content of the `D` memory. The `b:` memory region declares the content of the bias and accumulation memory. Accumulators are initialized at zero while biases are constant values. The `w:` memory region declares the content of the weight memory of the ANN and additionally, it defines *move* weights. With these special weight values, data can be moved from one column `c` of the `D` memory to another, by optionally adding a constant bias. This edge usage of the perceptron (although resulting in low resource utilization) enables the compiler to generate memory-to-memory move operations. This operation is useful to move results to the last memory cell `d(127)` thus enabling output sink write. *Move* weights are sparse constant vectors that copy the column value where the corresponding coefficient is ’1’, and reject the content of the others where it is ’0’. Finally, the program memory is defined with the label `main:` which includes a sequence of one-instruction assembly lines, previously introduced in Lst. 1. The program memory sequentially addresses the content of the other three memories which provide inputs for the computing unit, and write-back results to one of those.

When an instruction comprises a `popin` option, the data arriving from the source (implemented in hardware using a First-In First-Out, FIFO, memory) is immediately used for computation and it is implicitly written to `d(0)`. This operational scheme is different compared to all other options: when a `popin` is issued, the source data are immediately redirected to the perceptron without first reading the input data from the `D` memory. For all the other options, instead, the `d` inputs of the perceptrons are sequentially read from the `D` memory. This functionality enables the vector processor to complete a single instruction within the same clock cycles, also in case new data are sourced. At assembly level, the only semantic constraint that must be ensured is that when a `popin` option is used the first operand of the `com` instruction is `d(0)`.

As exemplified in Fig. 3, the options in the assembly code operate on the datapath to route the output of the memories and enable specific functional units. For instance, for an

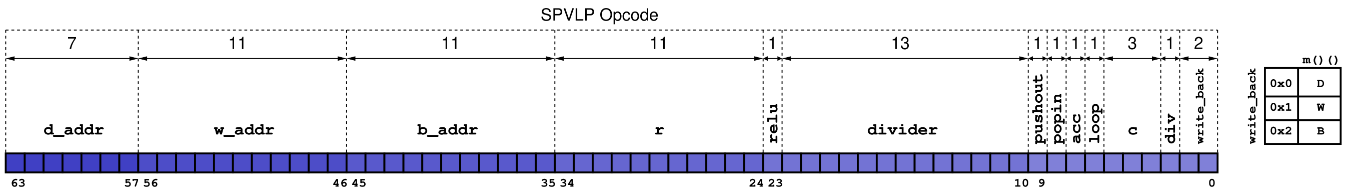


FIGURE 4. SPLVP 64-bit opcode stored in the `I` memory. The destination memory `m` for write-back is encoded using the table given on the right. Options are encoded directly using 1-bit flags, except from the divider.

option `pushout`, similar to the `popin` case, the current perceptron output is redirected to the output sink, and the full output vector (comprising overall eight values) is completed by the other seven values from the same memory row, which is 127. The multiplexer devoted to redirection indeed accepts both `pushout` and the column value `c`. Consequently, also in this case, the semantic constraint on the instruction consists of having, as the last argument (write-back) of the `com` instruction, a value of `d(127)` irrespective of the column `c`.

Fig. 4 shows the opcode of the vector processor. Instructions are encoded in a little-endian format and include the option bits, mostly in the lowest part (bit nine to zero). With such encoding, the instruction memory `I` can be seen as a simple addressing table that stores information on where to retrieve coefficients, data, and accumulations, how to modify the datapath, and where to write the results. The opcode straightforwardly encodes all the information included in the assembly command given in Lst. 1.

In this work, we have implemented integer arithmetic PEs. However, the machine could be synthesized using floating-point formats such as `bfloat16`, `float16`, and `TensorFloat32` [44], by implementing the required logic in the PEs, and by adjusting, in general, memory size. It would be also possible to maintain memory capacity unvaried and store intermediate layer results using 8-bit floating point arithmetic, for instance `e5m3` or `e4m4` [45]. In particular, `e5m3` can represent normalized `float16` numbers with lower accuracy but enable a small overhead for `float32` conversion. Consequently, the bias memory can be used to directly store floating point accumulations at 32 bit.

B. BLOCK SCHEME AND MCU INTERFACING

Fig. 5 depicts a high-level internal block scheme of SPLVP with the associated MCU interface. As with an Inter Integrated Circuit bus (I^2C), a Synchronous Peripheral Interface (SPI), or a Universal Asynchronous Transmit and Receive (UART) interface (normally available in all MCUs), the maximum data transfer speed would be limited, we have developed a custom parallel interface that has overall 28 pins (16 for input and 12 for output). Observe that low-power MCUs are available in packages with enough number of pins to host our interface. For instance, the STM32L4 MCU considered in this work is available in packages having 72 to 144 pins. With our custom interface, data can be simply bit-banged directly to the general-purpose I/O pins of an MCU without necessarily requiring specific

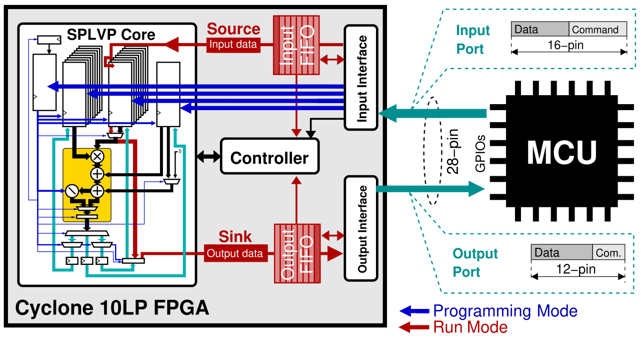
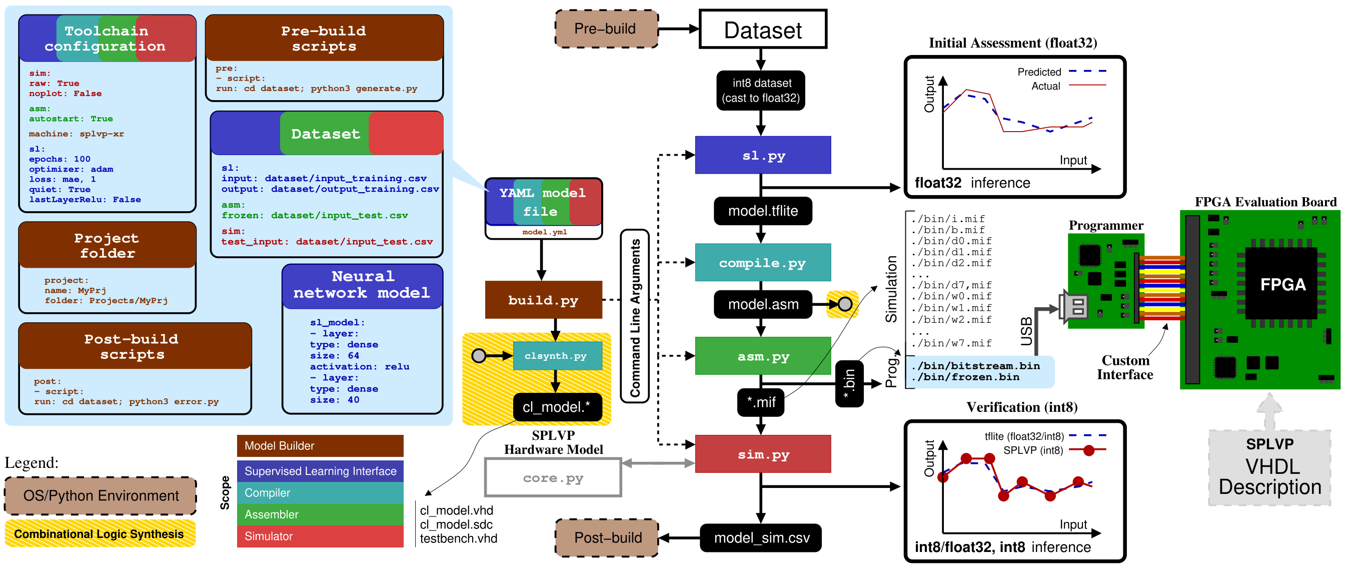


FIGURE 5. SPLVP high-level block scheme with the internal sub-systems devoted to controlling a custom interface that conveys MCU incoming and outgoing data.

bus hardware support. This portable and low-complexity mechanism has the potential advantage of being easily executed for instance by a Direct Memory Access (DMA) without burdening the microcontroller CPU. By carefully planning MCU pin connections, data can be organized in memory so that it is enough to perform straight DMA transfers directly on the GPIO peripheral registers. The interface is specifically designed to i) transfer the program and data to the SPLVP (to implement programming and inference), and ii) read computed output from it. It is split into input and output ports, and it is controlled by two dedicated sub-systems named `Input Interface` and `Output Interface` that implement a custom protocol. Both ports enable high-speed data transfer up to 10 Mbytes/s. In case the SPLVP program does not fit the internal MCU flash, cheap external serial memories can be used to retrieve the program at boot time.

We define two operational modes, `Programming Mode` and `Run Mode`. In `Programming Mode`, the data from the MCU is used to initialize all the internal memories of the SPLVP core. In `Run Mode`, the inference is executed; the data from the MCU is pushed to the `Input FIFO block`, i.e., the source given in Fig.2. The core writes the results to the `Output FIFO` which represents the sink. In this implementation, both FIFOs are eight elements deep, and input and output data are 64-bit wide. The `Controller`, based on the status of the `Input Interface` (`Programming` or `Run Mode`) and the empty/full status of the FIFOs, coordinates the SPLVP core and implements fetch and execute phases. The perceptron execution is normally completed in two clock cycles, while in the presence of a `div` option, it is prolonged by 12 clock cycles. The complete hardware implementation



```
sim:
raw: True
noplot: False

asm:
autostart: True

machine: splvp-xr

sl:
epochs: 100
optimizer: adam
loss: mae, 1
quiet: True
lastLayerRelu: False
```

```
pre:
- script:
run: cd dataset; python3 generate.py
```

```
sl:
input: dataset/input_training.csv
output: dataset/output_training.csv

asm:
frozen: dataset/input_test.csv

sim:
test_input: dataset/input_test.csv
```

```
project:
name: MyPrj
folder: Projects/MyPrj
```

```
sl_model:
- layer:
type: dense
size: 64
activation: relu
- layer:
type: dense
size: 40
```

```
post:
- script:
run: cd dataset; python3 error.py
```

```
./bin/i.mif
./bin/b.mif
./bin/d0.mif
./bin/d1.mif
./bin/d2.mif
...
./bin/d7,mif
./bin/w0.mif
./bin/w1.mif
./bin/w2.mif
...
./bin/w7.mif
./bin/bitstream.bin
./bin/frozen.bin
```

FIGURE 6. Complete design toolchain workflow that considers `int8` **quantized** `float32` **input data for both training and simulation to generate all the necessary files to operate SPLVP.**

of the processor is detailed in Appendix Sec. A and the interface is detailed in Appendix Sec. B. A detailed digital simulation encompassing both full hardware and interface is given in Appendix Sec. C.

IV. TOOLCHAIN

We have designed the complete toolchain of SPLVP using `python3`, which enables code portability across different operating systems. To increase the ease of operation and to allow the users to smoothly design and run neural network classifiers on our ML microprocessor we have designed an ad-hoc utility named `build.py`. This program reads a YAML file that includes all the information required for data pre-processing, supervised learning, compilation to binary assembly, and simulation of a given FC ANN. As an alternative implementation method, the toolchain can also generate a VHDL description of the ANN model. In other words, the ANN can be hardcoded in hardware without the need for the SPLVP core itself. This generated code includes a minimally customized testbench and an `sdc` file for synthesis.

Fig. 6 shows a compact view of the complete toolchain flow. The toolchain is invoked by running `python3 build.py model.yaml`. The YAML file lets the user link pre- and post-build scripts, that can be used to run data pre- and postprocessing or enable custom learning results visualization. Once the model is deployed in a final application, the pre-build script preprocessing needs to be implemented on the host MCU as well and applied to the real-time sensor data. Some examples of preprocessing encompass data normalization, offset correction, and subsampling. The goal of the preprocessing step (which is mandatory in ANN [46] and impacts the overall computing time), is the generation of training and test data quantized in an `int8` format. In our toolchain indeed, we assume that input data for the supervised learning is already quantized at 8-bit, but represented using `float32` numbers. We have chosen this approach (and not using full `float32` data) because input data from various sensors (for instance keyboard text output) in microcontroller applications, is inherently quantized (text can be directly converted into an ASCII representation). Our goal is the implementation of a solution that can efficiently run in limited resources MCU as an external component add-on.

The high-level `build.py` utility generates the command line arguments to all the sub-utilities, named `sl.py`, `compile.py`, `asm.py`, `sim.py`, and `clsynth.py`. These implement supervised learning, quantized `tflite` model compilation, assembly, simulation, and direct logic synthesis, respectively. The implementation details of the compiler, assembler, and simulator, with an example of direct VHDL synthesis, are reported in Appendix Sec. D, E, and F, respectively.

A. SUPERVISED LEARNING INTERFACE

`sl.py` is responsible for the execution of the supervised learning on the preprocessed data generated by the pre-build scripts, and for network quantization. To reduce the memory bandwidth and the computational cost of ANNs, two types of quantization approaches exist: PTQ and Quantization Aware Training (QAT) [47]. The compiler implements PTQ using the available `tflite` methods. `sl.py` is parametrized by the shape and activation functions of all the network layers, the loss function to be minimized, the maximum number of epochs used for training, the optimizer, and the loss limit. This last parameter is related to the network accuracy and is defined as the loss threshold below which supervised learning can be interrupted. In this work, we have used both Mean Absolute Error (MAE) and Mean Square Error (MSE) loss functions. Supervised learning is implemented by training the neural network assuming `float32` weights (randomly initialized), in particular, the preprocessed inputs at 8-bit are converted into a `float32` value. We do *not* normalize the input data with min-max or other common techniques at training time, rather we use data as is, to directly emulate a sensor output. The utility also permits the execution of an inference simulation on the obtained model and permits the extraction of Scalable Vector Graphics (SVG) simulation plots. After the `float32` model learning is completed, the network is fully quantized into an `int8` representation using the `tflite` stack. The obtained network is exported as a `tflite` file (in Fig. 6, named `model.tflite`), which is a flatbuffer representation of the network itself including quantized weights and bias values.

B. COMPILER

The `tflite` model file is compiled to generate an assembly file for SPLVP. The SPLVP compiler is based on the `tflite` 8-bit quantization specification [48], and the relative paper given in [49]. The quantization specification provides information on the way each operator in the model is represented, which ones are supported, and the corresponding tensor specifications. The operator's input and output tensors are quantized assuming a given scale factor and zero, which are fundamental to define integer quantization, in this case, from `float32` to `int8` and vice versa. In our work, we used classifiers that are based only on FC layers. According to the specifications, the TF's `FULLY_CONNECTED` operator's input, output, and weights tensors types are `int8`, while the bias tensors datatype is `int32`. To maintain compatibility, we store the operator's intermediate and final dot product results as `int32` numbers. The compiler must enforce write-back on the `B` memory, which is the only one that is organized into 32-bit rows. The `compile.py` program takes as input the generated `tflite` model and converts it into SPLVP assembly. According to the `tflite` specifications, at the input and output layers data needs to be *manually* quantized and de-quantized, thus involving the use of floating point operations. To keep MCU constraints as low as possible, in our implementation we do not consider the input and output scaling and zeros: we feed the network directly with the preprocessed inputs because we assume that scaling and zeros tend to be one and zero, respectively. `compile.py` accesses directly the flatbuffer file, extracts the tensors graphs and operators in the model, their weights, biases, scaling factors, and generates an internal representation of the operators' graph by assuming a single perceptron unit. It outputs a single `asm` file that includes all the instructions and the ANN data including intra-tensor scaling that is implemented using division.

C. ASSEMBLER AND PROGRAMMER

At this point of the flow, the assembly file needs to be translated into a binary machine code that can be directly downloaded to the SPLVP core. Moreover, it can be also useful to translate some of the testing input data as binary inputs to be streamed to SPLVP through its input interface, so that the operation of the system can be verified after programming. For these purposes, `asm.py` considers as input the compiled `asm` file, and optionally an input testing CSV file, to generate i) a firmware binary file to be streamed on the processor I/O interface (named `bistream.bin`), ii) the snapshots in a MIF format of the internal memory of the device useful to run simulations (named, `i.mif`, `b.mif`, `d0.mif–d7.mif`, and `w0.mif–w7.mif`), and iii) a frozen binary file containing input data taken from the testing CSV file (named `frozen.bin`). `asm.py` receives, as command line arguments, the input files and the output directory for binary data generation from the YAML description, and an `autostart` option to include a toggling instruction from programming to run mode at the end of the program, in the firmware binary file. This way the device is immediately ready to accept input data and run inference.

The `bin` files generated by the assembler are now ready to be transferred to the SPLVP core running on the FPGA evaluation board. For this purpose, we used a commercial MicroPython board that runs a firmware that drives the SPLVP interface signals, routed over a custom piggyback Printed Circuit Board (PCB). This programmer board sequentially streams groups of two bytes from the `bitstream.bin` file without modifications (data is read as is from the file) to the accelerator through its input interface.

D. SIMULATOR

Using the generated MIF files, SPLVP can be then simulated against the mixed `float32`/`int8` inference of `tflite`. The simulator module `sim.py` considers as inputs a directory where the MIF files are stored, a test input CSV file, and finally the `tflite` model. `sim.py` uses a hardware model `core.py` where all the components of the processor, i.e., the PE, internal memories, input and output FIFOs are modeled. The simulator first runs a software inference based on host `tflite` implementation, and next it runs again the same inference on the emulated SPLVP hardware. In particular, it loads the MIF files, pours the memory snapshots into the hardware model, and then feeds the input FIFO memory with data from the input CSV file every time a `popin` operation is required. When the input CSV file entries are over, the processor execution is stopped and both `tflite` and SPLVP outputs are stored in a CSV file, then simulation data is visualized. Optionally the simulator can save the obtained plot in an SVG file.

After the simulation is completed and both `tflite`/ SPLVP inference outputs are saved on disk, the toolchain can optionally launch specific post-build scripts to compute, for instance, the accuracy of the inference on the simulated data or other useful parameters. These scripts, which are executed as the last steps in the design flow, are indicated in the YAML project file using a specific key.

E. NEURAL NETWORK DIRECT SYNTHESIS

It is also useful to hardcode MLPs on FPGA for real-time acceleration applications, where very low latency is a mandatory requirement [35], [36]. For this reason, we have implemented a functionality in our toolchain to directly translate the `asm` file into a synthesizable VHDL description, through a specific `clsynth.py` tool. Such VHDL description represents a logic synthesis of the ANN, and it is obtained by assuming that the assembly is an intermediate representation (see Appendix Sec. D3). All the layer operations of the ANN are translated into combinational logic. The data from the input FIFO is stored into an 8-byte shift register with depth based on the size of the ANN input layer, and a single 8-byte register is used to sample the network output. The description considers two clocks, `clk` and `clk_out` that feed the shift register and the output register, respectively. To enable a correct synthesis and verification of the hardware the tool generates a Synopsys Design Constraints (SDC) file that assumes a `clk` to `clk_out` delay of 1 μs, and a testbench specifically designed to stimulate the hardware model with the input test CSV data, the same file used to implement the frozen dataset for hardware verification. The testbench is customized based on the number of inputs of the ANN, that are unrolled using a pipeline of registers. The generated output files are stored in a `synth` subdirectory in the project folder, and they can be opened by a synthesis tool, in our specific case Intel Quartus Lite 20.1, to synthesize the quantized ANN. The same toolchain can then be used for the design of the ANN model, for the code generation for SPLVP, and thanks to this automatic synthesis functionality, for further speeding up inference time on FPGA when latency requirements are very critical.

V. MEASUREMENTS AND VALIDATION

This section demonstrates the real-world applicability of our solution along with its design toolchain. Moreover, it presents synthesis results and performance figures of SPLVP. Our validation phase considered several open source datasets from [50], except in one case (the fingers recognition task in Sec. V-B), where the dataset has been manually collected using custom interfaces and electronic modules. The goal of all the examples presented in this section is to prove that SPLVP can be used in real ML applications. Here, we do not focus, as the primary objective, on the study of the best ANN to solve these problems and improve the state-of-the-art ML techniques. For all the considered problems, we designed custom MLPs assuming a binary coding for the output values. Such representation is discussed and detailed in Sec. V-B. The number of hidden layers and the network topologies have been determined based on a heuristic approach and on the necessity of testing the machine features. We have verified the operation of `build.py` on both MacOS Ventura 13.3 and CentOS Linux 7. We have observed slight variations in the `tflite` inference (typically ± 1 variation) depending on the version of TF, in particular 2.12.2 (MacOS) and 2.6.0 (CentOS Linux).

A. SPLVP SYNTHESIS

Including input and output interfaces, SPLVP has been synthesized on a 10CL025YU256I7G FPGA device of a Cyclone 10 LP Evaluation Board. The accelerator requires 18 % of the logic cells (4513 Adaptive Look-Up Tables, ALUTs, out of 24624, for 4140 LUT-only logic cells), it comprises 373 registers, 33 pins (that is 22 % of the available ones), 6 % of the embedded 9-bit multipliers (8 out of 132) and requires 1 built-in Phase Locked Loop (PLL) out of 4. The internal memories occupy 77 % of the overall device availability. The design has been synthesized to match the STM32L476 maximum clock frequency of 80 MHz, hence enabling a fair comparison. The maximum frequency achievable in the Cyclone 10 LP device is 84.56 MHz. The synthesized object has been converted into a `.jic` JTAG Indirect Configuration file (JIC) and written in the internal flash memory of the evaluation board so that the FPGA is automatically configured at system startup. The project is organized into five VHDL files, a VHDL constant definition file, and one SDC file. The VHDL files comprise the description of the entity and architectures of i) the top-level hierarchy, ii) the instruction decoder, iii) the single perceptron (including ReLU, accumulation, and scaling functionality), iv) the input interface, and v) the output interface. All the processor constants, e.g., opcode bit positions and memory width definitions are contained in a VHDL definition file. The memories and the PLL are based on proprietary FPGA IPs. By assuming the same memory capacity of the present prototype, increasing the perceptron parallelism P would only lead to an increase in the number of Digital Signal Processing (DSP) elements and multiplexers devoted to routing the perceptron output. The absolute addresses in the instruction memory `I` would simply identify a larger number of parallel `W` and `D` memory groups, in this case not limited to eight.

B. PHYSICAL PROGRAMMER AND FINGERS RECOGNITION

To verify the operation of SPLVP, we have developed a proof-of-concept module using a commercial MicroPython V1.1 module, that can be directly connected to the Cyclone 10 LP Evaluation Board and program the SPLVP. Contextually, we have developed a toy recognition application in which a commercial AMG8833 Infra-Red (IR) camera image (8×8 pixels, 12-bit per pixel, 10 frames per second) is used to detect three features, i.e., the presence of one or two fingers in front of it and additionally, their absence. Fig. 7 shows the complete setup used to build the dataset and implement this recognition task, which includes the programmer's proof-of-concept. The IR camera is read using another Micropython module named camera reader, (see Ⓐ) that is connected to a personal computer using a USB cable. The personal computer, Ⓑ, that runs the model builder introduced in Sec. IV, includes a specific interface for the IR camera named `cif.py`. The camera interface is used to collect raw data, indicated as p, used for the supervised learning of the model and its testing. The camera

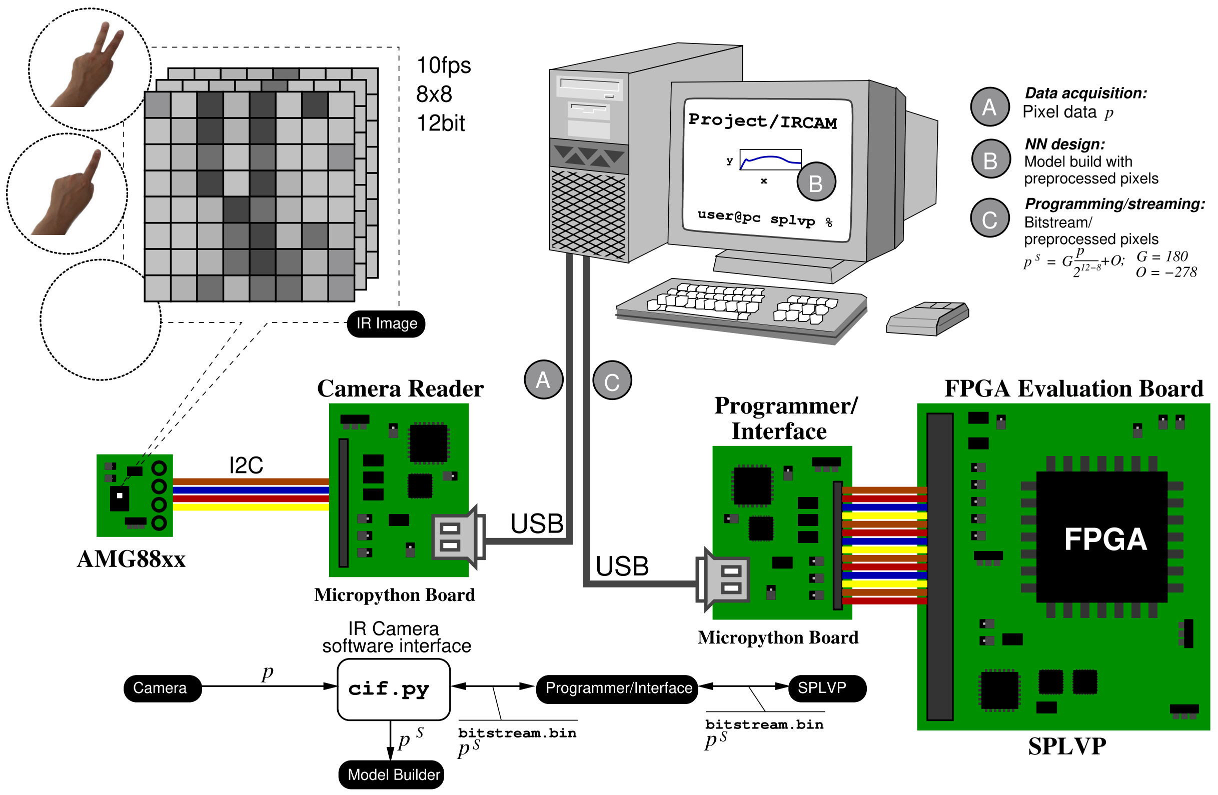


FIGURE 7. Experimental setup block scheme for both dataset acquisition and inference using the IR camera, including programmer and camera reader. The goal of this recognition task is the detection of three conditions, two given by hand gestures and the third given by their absence.

output is preprocessed by `cif.py` (to obtain p^S) using the formula indicated in Ⓒ. First, the raw data is divided by 2^{12-8} to rescale pixel depth to 8-bit, it is multiplied by a gain G, and finally, it is added to an offset O. These scalings have been implemented to increase image contrast and permit the visualization of fingers at nominal body temperature. For each pixel, the number of operations required for preprocessing are then i) a binary shift, ii) a multiplication, and iii) an addition, for an overall overhead of approximately $64 \times 3 = 192$ operations. These preprocessing operations need to be executed before sending data to SPLVP and therefore constitute an overhead that needs to be handled by an MCU.

Once the MLP is designed using the model builder, the programmer device can be used to program SPLVP. The programmer is another Micropython board that is directly connected to both the input and output interface of SPLVP using the onboard GPIOs. The input interface is connected to pins `A0–A7` for the low byte and `C0–C7` for the high byte. The output interface is connected to pins `B15–B6` and `B1–B0`, where `B0` is `FIFO_FULL`. The programmer is connected to the personal computer using another USB port and operates as a Virtual Communication Port (VCP) and Mass Storage Controller (MSC). Programming is achieved by transferring the `bitstream.bin` file content through the SPLVP input interface using Micropython scripts that implement a software version of the protocol. `bitstream.bin` can be transferred to the programmer by simply copying it from the computer to its mass storage device. After SPLVP programming, the core goes into run mode and the programmer can be used to test the processor or send frozen binary inputs to verify its functionality. In this specific recognition application, the programmer waits for new data p^S on the VCP (that are transmitted from the computer by `cif.py`) and directly applies them as input to SPLVP by implementing the input interface protocol. The camera interface, indeed, bridges the data from the AMG8833 camera reader to the programmer by opening two separate VCPs and directly forwards the camera output to SPLVP, thus enabling real-time recognition. This specific proof-of-concept implementation requires the PC, but the IR camera can be directly accessed by the same MCU that is interfaced with SPLVP.

With `cif.py` we have created a dataset for the three conditions exemplified in Fig. 7. The training dataset comprises 1071 entries, where the first 445 entries are relative to an absence of hands in front of the camera, the next 400 are relative to one finger and the remainder are relative to two fingers in front of it. Typically, in MLPs the number of outputs corresponds to the number of features to be detected (in this case three), and typically after FC layers a *softmax* layer is used to extract the probability of each feature. Although softmax is fundamental in Transformers and can be approximated [51], it is a very computationally expensive

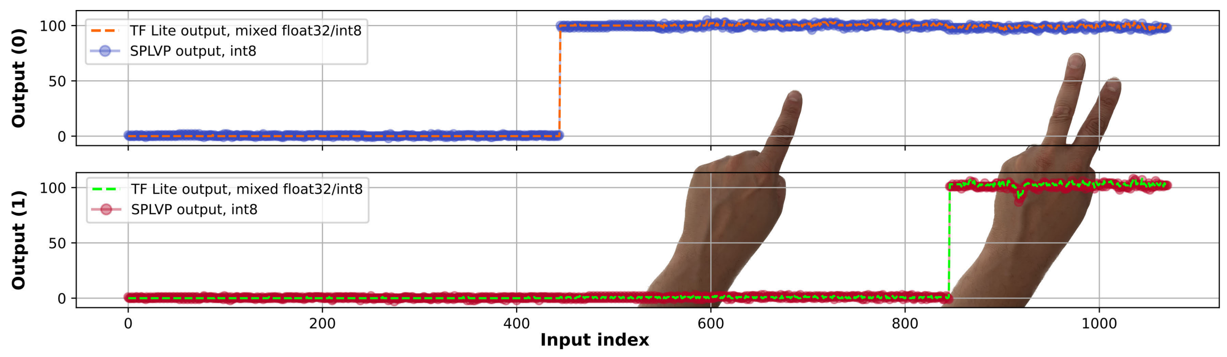


FIGURE 8. **Mixed** `float32`/`int8` **inference of** `tflite` **versus** `int8`**-only SPLVP simulation of the developed MLP model for the IR Camera.**

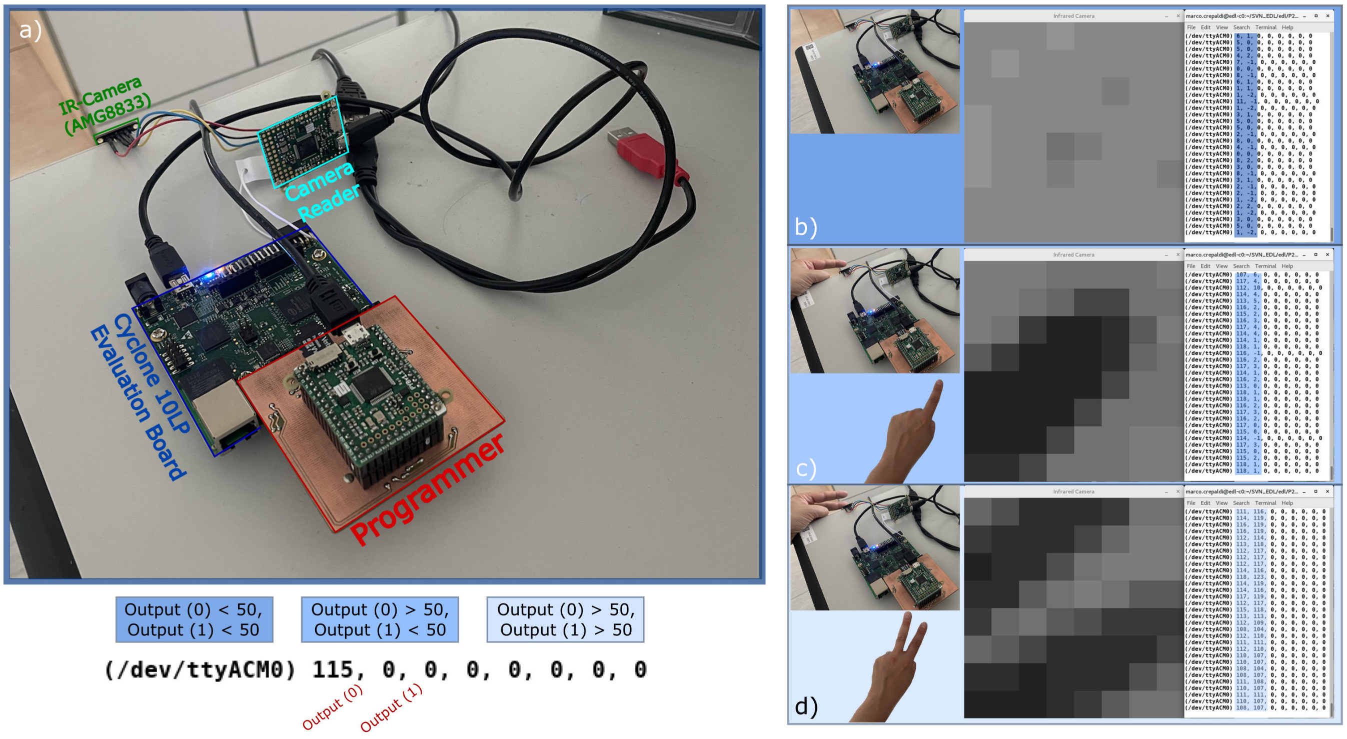
operator that makes little sense in our context. As our goal focuses on maintaining the minimum number of resources to run MLPs we decided to avoid the use of softmax layers in our models.

To additionally keep the hardware and software resources of MCUs low we use *binary coding* on the classes to be detected, assuming a big logical margin. In binary coding, classes are encoded as ordinals, the corresponding values are converted into binary code, and finally the digits of such binary code are split into separate columns. Here, we practically implement the digits '0' and '1' using the integer outputs of SPLVP. In literature, binary coding has been demonstrated to provide the same performance as one-hot encoding (typically used in combination with softmax layers) in the study of [52], although it is not exhaustive. With binary coding, the last layer size can be indeed two, and the three features of the present example can be encoded as integer values as `0–0`, `100–0`, and `100–100`, where `0` corresponds to a logical '0' and `100` to a logical '1'. The logical margin here is 100, and it is defined as the separation between logical '0' and logical '1' values in the integer domain. Enforcing logical margin has two main advantages, that is the possibility of i) controlling the robustness of the output values, similar to noise margin in logic gates, and ii) ensuring a degree of freedom for controlling the ratio between the input/output scaling factors of the quantized network. As the MLP output is still an ensemble of integer numbers, while logically they need to be converted to Boolean values, a discrimination threshold of 50 can be applied. Such threshold comparison requires that the downstream MCU runs, ideally, only compare assembly instructions on the outputs calculated by SPLVP, therefore impacting very little on CPU time. Observe that even in the commercial Google Coral platform, some of the operations can be still executed on the main CPU [53].

Using the model builder, we have designed an MLP comprising six layers overall, having sizes 60, 60, 60, 4, 3, and 2, where the first two layers have a ReLU activation function and all the others have linear activation functions. We have chosen to stack multiple linear layers to test the operation of SPLVP in the presence of multiple scalings between one layer and another (`div` option). However, being all linear, these can be collapsed into a single one. We have demonstrated that three layers of size 60, 60, and 2, where only the last one has linear activation, are enough to solve the problem. We have used an Adam optimizer, a fixed number of epochs of 300 for training (leaving the final accuracy target unspecified), and a MAE loss function. After 300 epochs, the obtained residual loss is below 0.5. The input layer is a flattened vector of 64 elements one byte each, represented in an `int8` format, whose input testing and training data are contained in the generated CSV files.

Fig. 8 shows a simulation of SPLVP inference against `tflite` assuming the same input dataset, obtained using our simulator `sim.py`, that was invoked with the model builder. As the model has been compiled to provide only two outputs we present only `Output (0)` and `Output (1)`, the two MLP active output values (all the others are zero) that correspond to the least significant bytes of the SPLVP output. Results show that the SPLVP inference is very close to those of `tflite`, although this last one results from a mixed `float32`/`int8` computing. For some inputs with two fingers, in both cases, the model outputs exhibit a variation compared to the expected values of `100–100`. The use of a large separation between the integer values, and therefore the possibility of using a final threshold comparison, helps in the mitigation of these spurs and at the same time relaxes the accuracy requirements of the MLP. The number of parameters of the model is 11487. When compiled, the model occupies 73% of the weight memory, 23% of the data memory, 15% of the bias/accumulation memory, and 44% of the instruction memory. To run a complete inference round (single 64-bit input frame), the 80 MHz-clocked processor takes 73.35 μ s, which corresponds to the execution of 1800 instructions. These instructions comprise, *inter alia*, `div` options with longer execution time compared to the standard perceptron unit operations, and move instructions with no data parallelism.

Fig. 9(a) shows a photo of the measurement setup presented in Fig. 7, including both the programmer and camera reader, and depicts the logical conditions for which detection can be achieved on the two SPLVP outputs, i.e., `Output (0)` and `Output (1)`. The figure provides also details on the SPLVP hardware output printed by `cif.py`


a)
IR-Camera (AMG8833)
Camera Reader
Cyclone 10LP Evaluation Board
Programmer
Output (0) < 50, Output (1) < 50
Output (0) > 50, Output (1) < 50
Output (0) > 50, Output (1) > 50
Output (0)
Output (1)
b)
c)
d)
Infrared Camera
marco.crepaldi@edl-c0:~/SVN_EDL/edl/P2...
File Edit View Search Terminal Help

```
(/dev/ttyACM0) 115, 0, 0, 0, 0, 0, 0, 0
```

```
(/dev/ttyACM0) 6, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 5, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 5, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 4, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 7, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 0, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 8, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 6, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, -2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 11, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, -2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 3, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 5, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 5, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 2, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 8, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 4, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 0, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 8, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 3, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 8, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 3, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 2, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 2, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, -2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 2, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, -2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 3, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 5, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 1, -2, 0, 0, 0, 0, 0, 0
```

```
(/dev/ttyACM0) 107, 6, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 4, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 10, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 4, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 113, 5, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 115, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 3, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 4, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 4, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 3, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 113, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 3, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 115, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, -1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 3, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 115, 0, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 115, 2, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 1, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 1, 0, 0, 0, 0, 0, 0
```

```
(/dev/ttyACM0) 111, 116, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 119, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 119, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 116, 119, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 114, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 113, 118, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 117, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 117, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 117, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 116, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 118, 123, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 119, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 114, 116, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 117, 119, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 117, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 115, 118, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 113, 113, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 109, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 108, 104, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 110, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 111, 111, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 112, 110, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 110, 107, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 110, 107, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 108, 104, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 108, 107, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 111, 108, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 110, 107, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 111, 111, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 110, 107, 0, 0, 0, 0, 0, 0
(/dev/ttyACM0) 108, 107, 0, 0, 0, 0, 0, 0
```

FIGURE 9. a) Experimental setup comprising the programmer, the Cyclone 10 LP Evaluation Board that runs SPLVP, and the camera reader board for interfacing the AMG8833 infrared camera (top). Three detection conditions correspond to the absence, or presence of one or two fingers in front of the camera, with detail on the SPLVP output (bottom). b, c, and d) Output of `cif.py` while bridging in real-time the output of the camera and SPLVP in the three conditions, with a graphical representation of the corresponding infrared image.

during inference. Fig. 9(b–d) shows screenshots of the output of `cif.py` during the real-time inference executed by SPLVP while the AMG8833 camera is streaming data at 10 fps. As shown in the screenshots, the output data of SPLVP is consistent with the training values, thus providing on its least significant bytes, the integer outputs in the range enforced during training. Observe that in the case of two fingers, the integer values reach ~ 120, above the set-point of 100 enforced during training. This notwithstanding, this output classification is still correct because it is required only that outputs are both above the threshold value (which is here 50).

C. OTHER CLASSIFIERS

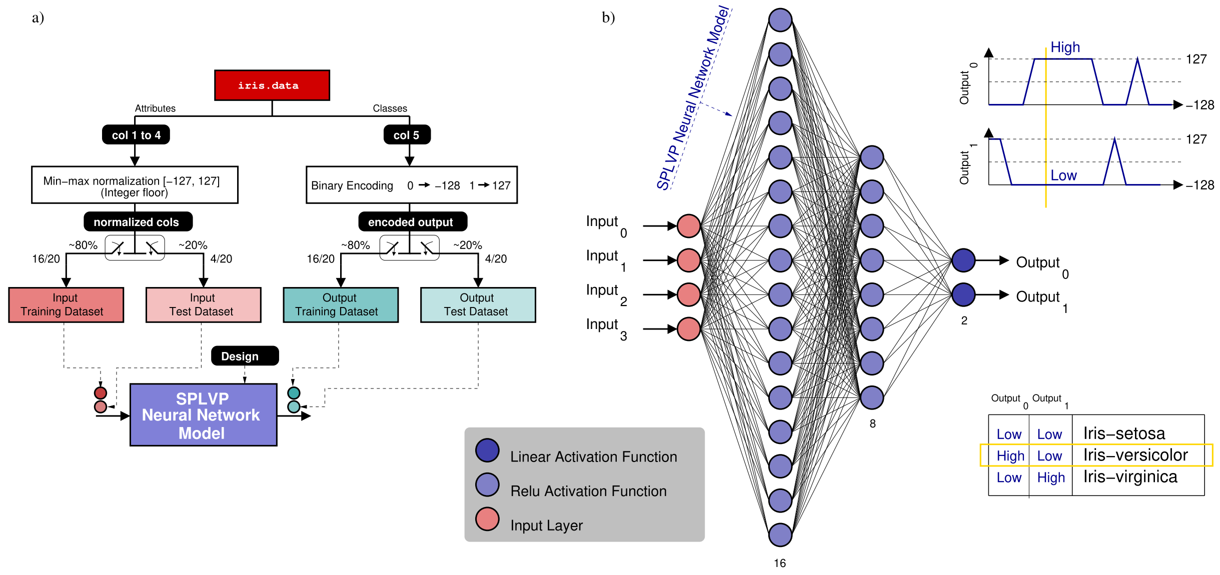
This section briefly details the design of a classifier for the Iris dataset [54] (for which we report its VHDL synthesis results in Sec. V-C2), and schematically summarizes all the classifiers used to validate our processor.

1) IRIS MODEL

Fig. 10(a) shows the scheme for the generation of both training and testing datasets, starting from the attributes and classes in the original archive `iris.data`. The first four columns of the file are attributes and the last column is the class in ASCII format. Data must be preprocessed to comply with our toolchain and processor: attributes and classes must be converted into an `int8` format. To this end, a script called by `build.py` before supervised learning, implements the preprocessing steps given in Tab. 2. Thanks to the particular numerical values in this dataset, the first and the fourth columns of the normalized attributes become always `127` and `-127`. This result is advantageous in terms of computing resources because supervised learning can actively operate only on the other two remaining non-constant attributes, that are sepal width and petal length, that vary across a fixed range.

Because in the original file, the classes are clustered, to build the training and testing sets, we consider chunks of 20 rows, where the first four are used for training, while the remainder are used for testing. Observe that in contrast to what is normally done in typical applications, here we do not normalize the complete dataset, but just one input at a time. While the normalization of a complete dataset can result in global scaling constants (that can be calculated once and in turn applied to new data), here we rely on a minimum preprocessing overhead on a hypothetical MCU. This way, preprocessing requires a limited number of subtractions and integer divisions on a very small number of attributes, which is still a viable choice for resource-constrained hardware. Compared to other ML models that consider 70% of the dataset for training [55], our network was trained with $\sim 80\%$ of it, leading to 100% accuracy. Fig. 10(b) shows a graph of the MLP designed using the model builder. The network comprises three layers of size 16, 8, and 2, with the first two having ReLU activation, and the last one having linear activation.

Fig. 11 shows inference outputs for both SPLVP and `tflite` using the training dataset generated according to



Output 0	Output 1	
Low	Low	Iris–setosa
High	Low	Iris–versicolor
Low	High	Iris–virginica

FIGURE 10. Designed neural network for Iris classification (right) with detail on dataset generation, inference, and output data convention (left).

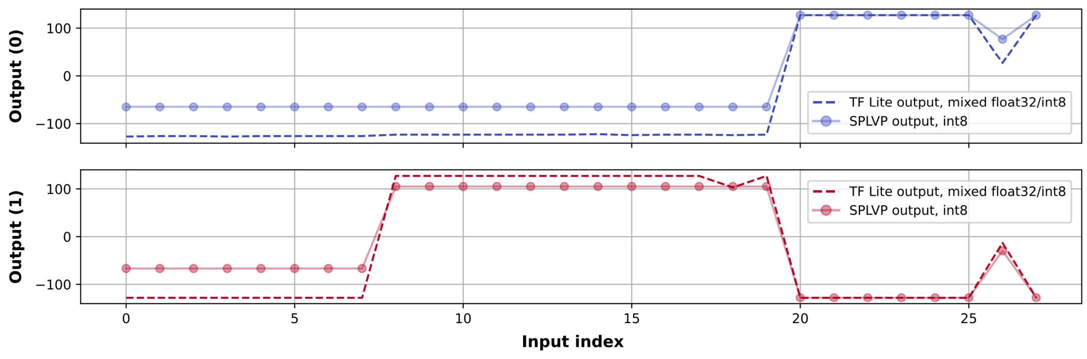


FIGURE 11. Mixed `float32`/`int8` inference of `tflite` versus `int8`-only SPLVP simulation of the designed FC model given in Fig. 10. The SPLVP outputs do not overlap with the `tflite` ones due to integer approximation in the division and to non-saturation of the 32-bit accumulators.

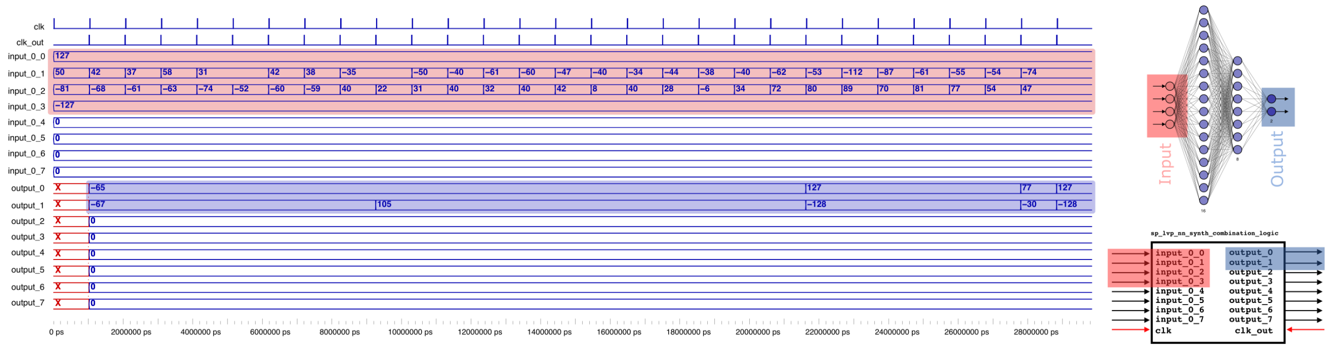


FIGURE 12. ModelSim Intel FPGA Starter Edition simulation of the MLP hardware description generated by `clsynth.py`, obtained with the autogenerated testbench that considers the same testing dataset used in Fig. 11.

what is presented in Fig. 10(a). The `tflite` outputs differ compared to those of SPLVP due to the following reasons: i) the implementation of scaling using an integer division instead of a fixed point multiplication and shift, ii) the non-application of the input scaling factor and the associated output scaling adjustment and iii) the consequent saturation of the 32-bit accumulators that may occur. To verify the performance of the obtained model, we have implemented a post-build script that compares the predicted outputs with the expected values, and the resulting accuracy is 100%. This result, which is consistent with the accuracy obtained with other ML classifiers [56] (above 96%), demonstrates that detection is robust and performance is not significantly impacted by quantization and numerical approximations.

TABLE 1. Dataset information and parameters used by the toolchain for supervised learning for different classification problems used to test the operation of SPLVP.

Dataset Information	Attribute Information/Format	Classes Information/Distribution	Optimizer	Maximum Epochs	Loss Function	Loss Threshold⦾
Iris [54]						
Small collection of measurements from three specimens of flowers originally proposed by Fisher in 1934. It is one of the most commonly found datasets in pattern recognition literature.	Sepal length, sepal width, petal length and petal width, measured in centimeters unit/fixed precision with one decimal point.	Iris Setosa, Iris Versicolour and Iris Virginica/33.3% each.	Adam	10000	MAE	9
SMS Spam/Ham (SMS) [57]						
Collection of text messages in English from various sources.	Text/ASCII.	Ham or Spam/86.6% Ham, 13.4% Spam.	Adam	10000	MSE	0.025
Banknote [58]						
Collection of four attributes and one class extracted using wavelet transform on images taken from an industrial camera used for print inspection.	Variance, skewness, kurtosis of the wavelet transformed image and the entropy of image/Fixed precision with four decimal points.	Valid or Non Valid/44.46% Valid, 55.54% Non Valid.	Adam	1000	MAE	–
Wi-Fi Indoor Localization (Wireless) [59]						
Collection of smartphone measurements attributes of signal strength from four different rooms.	Seven smartphone measurements in dBm units/Integer.	Room number (1-4)/25% each.	Adam	5000	MSE	0.25
Ionosphere [60]						
Data from a radar consisting of a phased array of 16 high-frequency antennas targeting free electrons in the ionosphere, with overall transmitted power on the order of 6.4 kW.	Complex values returned by the autocorrelation function resulting from the complex electromagnetic signal of 17 pulses./Fixed point precision, five decimals, that vary in the range $[-1, 1]$.	Good or Bad return/50% each.	Adam	5000	MAE	2.3
Avila⊖ [61]						
Collection of features built over a large number of digital images of the Avila Bible, a giant Latin copy of the whole Bible written during the XII century between Italy and Spain.	Upper and lower margin of the page, intercolumnar distance, number of rows in the column, column exploitation coefficient, weight, peaks, modular ratio, interlinear spacing and modular ratio/interlinear spacing ratio of each row/Fixed precision, six decimal points.	Letters A, B, C, D, E, F, G, H, I, W, X, Y/41.09%, 0.05%, 0.99%, 3.37%, 10.5%, 18.8%, 4.28%, 4.98%, 7.97%, 0.42%, 5.0%, 2.55%, respectively (percentage truncated at two decimals).	Adam	10000	MSE	150

⦾ = If not set (–), the training ends when the maximum number of epochs is reached, otherwise it ends when the corresponding loss limit is reached. MAE = Mean Absolute Error. MSE = Mean Square Error.

2) IRIS MODEL VHDL SYNTHESIS

Considering the limited size of the Iris model, we have validated the logic synthesis tool `clsynth.py` by generating an associated VHDL description (which includes testbench and SDC file) and by synthesizing it using Quartus Prime 20.1 on the same Cyclone 10 LP device family of SPLVP. Fig. 12 shows a simulation of the VHDL model obtained with ModeSim Intel FPGA Starter Edition 2020.1. The simulator runs the testbench generated by `clsynth.py` and reads the same testing dataset used in previous simulations. The testbench generates both clocks (`clk` and `clk_out`) in impulsive mode as they are needed only to i) feed the input shift register and ii) sample the computed output, respectively. The obtained output values correspond to those given in the graphical view of Fig. 11. The delay between the last positive edge of `clk` and `clk_out` needs to match the timing constraints, that we enforced to 1 μs. The time required for a complete hardware synthesis on a 3.6 GHz Intel Xeon CPU E5-1650 v4 running CentOS 7, is four minutes and 35 s. The synthesized model fits a 10CL120Y device, in particular requiring 50185 logic elements (42% occupation), 144 registers, and 204 fabric 9-bit multipliers. The synthesized VHDL model has a worst-case propagation delay of $\sim$ 445 ns, which is $\sim$ 14 $\times$ faster compared to SPLVP inference on the same model. However, the required hardware resources are by far greater compared to SPLVP (see Sec. V-A).

TABLE 2. Parameters and preprocessing for different classification problems used to test the operation of SPLVP, including state-of-the-art performance references.

# Attributes	# Classes	Training Set	Testing Set	MLP†	Preprocessing$^{\bigstar}$	Binary Coding	Threshold	Accuracy	Reference
Iris [54]									
4	3	122$^{\diamond}$	28$^{\diamond}$	16(r), 8(r), 2(l)	Min-max normalization in the range $[-127, 127]$ per entry.	'0'→`-128`, '1'→`127`	`0`	`tflite`: 100% SPLVP: 100%	[55], k-NN: 96.64%, SVM: 98.21%, LR: 96.43%
SMS Spam/Ham (SMS) [57]									
(variable text)	2	1674	3900	60(r), 40(l), 32(r), 16(r), 1(l)*	i) Use only the first 64 characters; ii) convert them to ASCII code (if unicode and $> 255 \rightarrow$ space); iii) min-max norm. in the range $[-100, 100]$.	'0'→`0`, '1'→`100`	`90`	`tflite`: 89.64% SPLVP: 89.69%	[62], EM + tok2: 85.54%, TR: 84.95%, Boolean NB + tok: 77.13% SVM + tok: 97.64%
Banknote [58]									
4	2	686$^{\diamond}$	686$^{\diamond}$	32(r), 32(r), 32(r), 32(r), 16(r), 10(r), 1(l)	Multiply each attribute by 9.	'0'→`-128`, '1'→`127`	`0`	`tflite`: 100% SPLVP: 100%	[63] (k-NN), GENILE: 95.92% LE: 94.46% LLE: 95.21% Isometric Projection: 94.52% IPCA: 77.23%
Wi-Fi Indoor Localization (Wireless) [59]									
7	4	1200$^{\diamond}$	800$^{\diamond}$	64(r), 32(r), 32(r), 32(r), 10(r), 2(l)	i) Add 60 to each attribute; ii) multiply the result by 2.5.	'0'→`-50`, '1'→`50`	0	`tflite`: 97.5% SPLVP: 97.625%	[64], PSO-NN: 64.66% GSA-NN: 77.53% FPSOGSA-NN: 95.16% SVM: 92.68% Naïve Bayes: 90.47%
Ionosphere [60]									
34	2	200	151	32(r), 32(r), 32(r), 1(l)	Multiply each attribute by 127.	'0'→`-128`, '1'→`127`	0	`tflite`: 100% SPLVP: 96.6%	[65]$^{\boxtimes}$, SFM: 94.6% SVML: 89.5% SVMG: 94.6%
Avila$^{\ominus}$ [61]									
10	12	10430	10437	64(r), 64(r), 64(r), 32(r), 32(l), 10(r), 4(l)*	i) Min-max normalization in the range $[-1, 1]$ per entry; ii) multiplication by 127.	'0'→`-100`, '1'→`100`	0	`tflite`: 83.8% SPLVP: 82.2%	[66], SVM: 82.67% NN: 94.56% k-NN: 75.61% DT: 98.25%

$\dagger$ = size of each layer (excluding input), with activation function in parenthesis (l is linear and r is ReLU). $\bigstar$ = At the end of all preprocessings, the obtained values are rounded to `int8`. $^{\diamond}$ = Training/testing set built by considering chunks of 20 rows as shown in Fig. 10(a). * = All the linear hidden layers can be avoided in principle but they have been used here to verify the correct operation of SPLVP in presence of multiple mixed activation layers scalings (`div` options). $\boxtimes$ = Mean values. $\ominus$ = Simple classification without rejection condition to drop unreliable samples.

3) CLASSIFIERS SUMMARY

Tab. 1 and 2 summarize the datasets with associated classifier parameters, training details, and references for the problems we have solved using SPLVP. The referenced documents present classifiers with diverse ML structures for which we report the acronyms as provided in the respective publications. SPLVP and its associated toolchain, support only FC networks. These models, together with the finger detection classifiers of Sec. V-B are used to assess the performance of SPLVP against the STM32L4 MCU. Tab. 3 shows the memory occupation breakdown on SPLVP of each model, with corresponding `bitstream.bin` programming file size. This binary file includes the interface pin values (including clocking) to be bit-banged on the input interface.

D. SPLVP PERFORMANCE AND COMPARISONS

1) L4-SERIES STM32

To compare the inference performance of SPLVP against a commercial microcontroller, we have considered an STM32L476 Nucleo Evaluation Board [67] and we have

TABLE 3. Memory occupation of the various classifiers on the SPLVP internal memory, with indicated capacity and corresponding programming file size (`bitstream.bin`) in Bytes.

Classifier	`D`, 1 KB	`W`, 16 KB	`B`, 8 KB	`I`, 32 KB	Programming File Size (B)
IR Camera	23%	73%	15%	44%	111162
Iris	5%	2%	3%	2%	5450
SMS	17%	52%	13%	33%	81770
Banknote	15%	19%	12%	15%	36938
Wireless	20%	31%	17%	24%	57578
Ionosphere	12%	21%	10%	15%	37258
Avila	29%	78%	25%	51%	127530

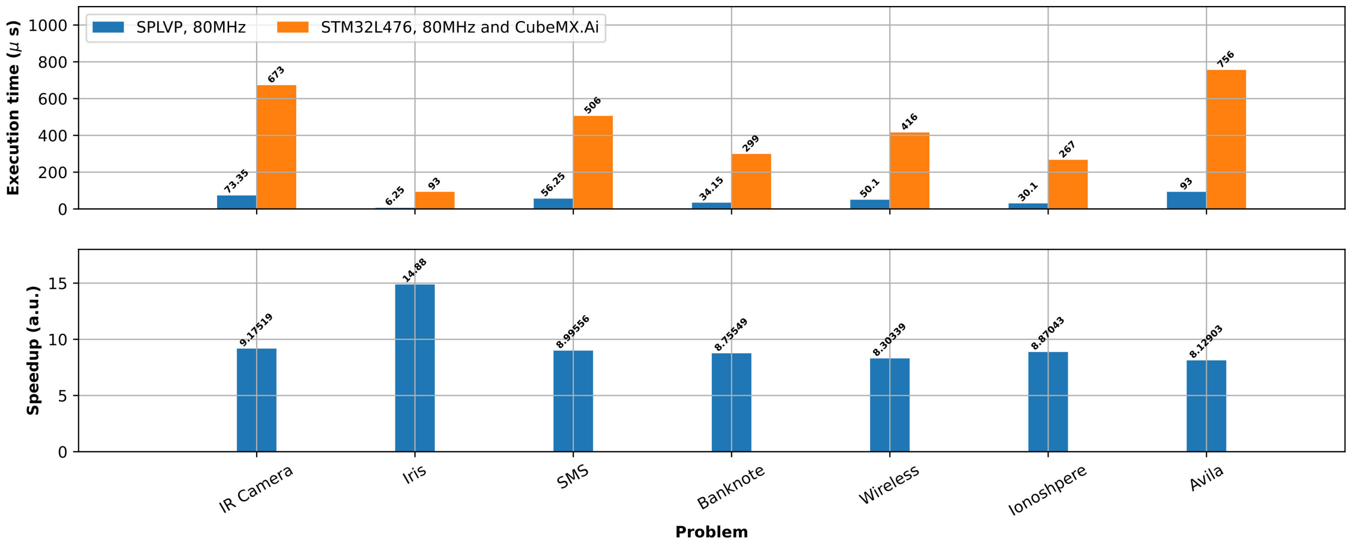


FIGURE 13. Comparative plot of the execution time for the classifiers previously presented when the `tflite` model is run on the STM32L4 MCU (IC, 90 nm process) and SPLVP (FPGA, 60 nm process), with associated speedup.

converted the `tflite` models obtained for the previously presented problems using STM CubeMX-Ai. This proprietary tool converts generic ANNs modeled in various software toolchains (for instance TF or PyTorch) into object files that can be directly linked to other compiled code and executed by STM32 microcontrollers. We have then generated a mainfile which calls the translated ANN code and toggles a GPIO immediately before and after the execution of the neural network. We applied the same testing inputs used for inference in the comparative plots given in the previous sections. The actual STM32 execution time is then compared with those obtained from SPLVP simulations. To verify the correctness of the simulations provided by the hardware model in `core.py` we have extracted measurements of the active computation time of the SPLVP through the reading of an additional pin (corresponding to the negated of the `empty` signal of the input FIFO), we have routed at synthesis time. An example measurement is given in Sec. V-D2. As simulations and measurements match, the execution time of `core.py` can be used to assess the hardware performance.

Fig. 13 compares the execution performance of an STM32L476 that runs the MLPs converted using the STM CubeMX.Ai utility. We have considered an STM32L476 ARM microcontroller (90 nm process) because its main CPU can be clocked at 80 MHz, that is the same clock frequency used for our SPLVP design. The speedup achieved with SPLVP is a function of the particular MLP type because the number of scalings (that require the internal integer divider) is a function of the number of outputs of each layer. Indeed, the Iris model that only requires 26 divisions performs significantly faster compared to the STM32. The average speedup across the models is ∼9.6 ×. Compared to STM32, when the Iris model is directly implemented in hardware on an FPGA (hence, the model is converted into a VHDL description and then synthesized using Quartus), the speed up reaches 209 ×.

2) HARDWARE INFERENCE SPEED MEASUREMENT

Fig. 14 shows an example measurement of the input FIFO `empty` signal which, according to the programmer operation, directly quantifies the single input inference time of SPLVP. As previously introduced, the programmer can stream frozen data generated by the assembler, in particular the content of the file `frozen.bin`. In this experiment, we program the device with the compiled Wireless classifier model, we send one input to the SPLVP and we check the output `DRDY` to retrieve inference. As the proof-of-concept programmer board runs a non-optimized Python code, that reads the content of `frozen.bin` from an external SD card filesystem and sequentially toggles the values of the GPIOs connected to the interfaces, the programmer requires ∼100 ms to send data to SPLVP and read the result. It can

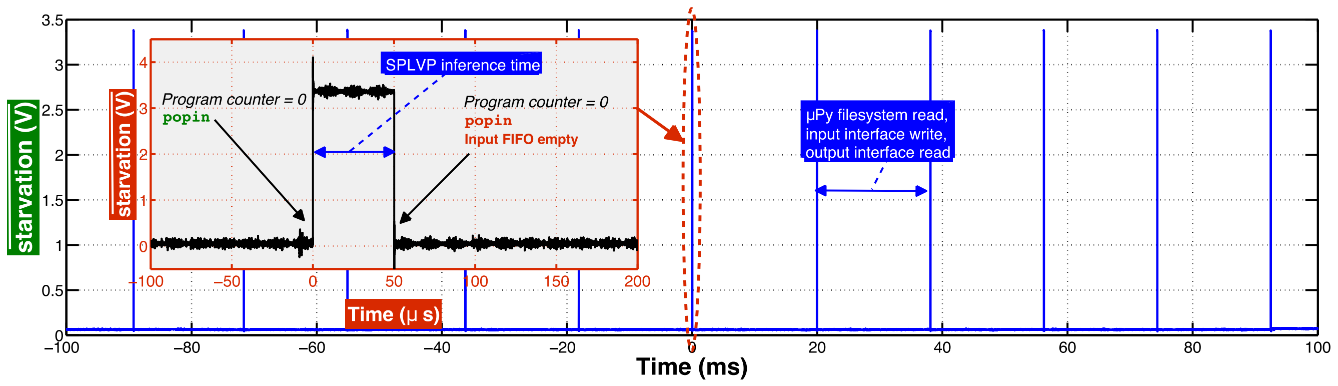


FIGURE 14. Example of physical measurement of the SPLVP internal $\overline{\texttt{starvation}}$ output of Fig. 16 during inference of the Wireless model. The measured inference time is ~50 μs. The proof-of-concept programmer board is limiting the overall throughput because its Python code runs significantly slower compared to SPLVP. The programmer indeed needs ~100 ms time to retrieve inference data from flash and drive the GPIOs to write and read the interfaces.

be demonstrated, for instance using bare-metal firmware, that the MCU can implement this GPIO toggling task significantly faster. This notwithstanding, SPLVP, which recovers execution immediately after a new 8×8-bit data is written on the input FIFO, completes inference in 50.1 μs. Observe that the Wireless Localization model we have designed using the model builder has an input layer of size seven, therefore from the execution viewpoint, writing a row of eight bytes to the input FIFO is enough to trigger program execution. After inference is finished, the program counter is reset, and as the first instruction has a `popin` option, the SPLVP controller remains frozen in the fetch state `F`, until an input FIFO entry (eight bytes) is not filled with new data.

3) POWER CONSUMPTION

We have estimated the power consumed by the FPGA chip by running a gate-level VHDL simulation of the SPLVP when it is programmed with the Iris model and by assuming the same dataset previously introduced for testing the model. To this end, the simulation output is then converted into a VCD file and used to extract signal activities for use in the Quartus Prime power estimator. The obtained static and dynamic power consumption of the 10CL025 FPGA is 248 mW. We have considered this approach because a physical measurement of the consumed current in the board as done in [68] includes the consumption of all the on-board chips of the evaluation board.

4) GOOGLE CORAL EDGE TPU AND CPU

To further position our development, we have decided to compare the obtained inference performance with the Google Coral development board [69], which is not conceived for low power MCU-based devices. The system is conceived to speed up inference in Edge devices with intensive use of ANN, with millions of coefficients. The Dev Board embeds an Edge TPU with a host CPU that runs a Linux operating system. In terms of toolchains the Dev Board comprises `tflite`, which can be simply invoked as done in our simulator to run both quantized models on the CPU and the TPU. Code generation for the Edge TPU is obtained through a proprietary compiler, that is based on a quantized `tflite` model input and generates another `tflite` model tailored to the TPU hardware. We have loaded our `sim.py` in the Dev Board system and we have run inference for all the models of Fig. 13, using both the 64-bit ARM CPU core and the Edge TPU.

Fig. 15 shows the inference performance obtained in both cases. It is clear that both computing architectures (including in the count the underlying operating system, and the associated `tflite` software library as well), cannot reach the inference time of SPLVP because they are designed to manage high complexity ANNs with several orders of magnitudes larger number of coefficients. Simply put, the models used in this test are too small to justify the associated overhead required by these complex systems to operate. This result, however, further justifies the relevance of our work, as for applications requiring very low hardware resources, such systems, although providing orders of magnitude higher computational powers compared to our hardware, cannot be used to obtain the same performance. It is noteworthy that the onboard CPU performance is significantly better compared to the Edge TPU. This can be easily explained by the larger overhead required by the host CPU to load inputs on the TPU and retrieve them using the Linux operating system, which is by far larger compared to the active inference time needed by our models. Moreover, the TPU is designed to aggressively accelerate matrix multiplications, and we expect that if its hardware acceleration units are not fully used, data is filled with zeros, therefore hindering its capabilities. This fact is more evident for the Iris model, where the 6.25 μs active computation time does not justify the use of more complex hardware. The Edge TPU of course provides significant performance speedups when the ANN model is large and complex and can be hosted completely inside the device's internal memory. The metric provided herein refers to measurements with the same input test datasets described previously. In the case of larger datasets the inference time per single input decreases in both cases, still stabilizing across values of a few hundred μs. The performance obtained

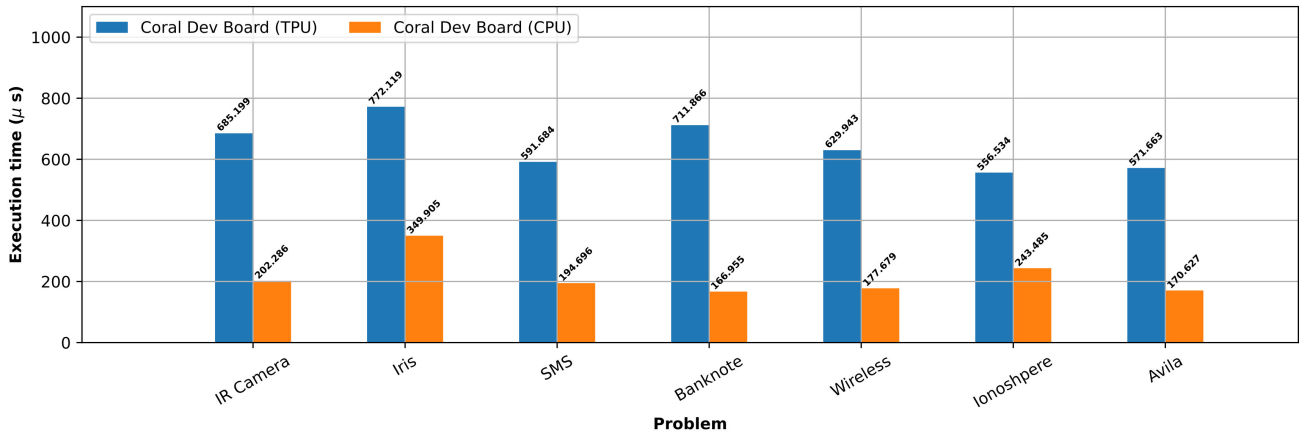


FIGURE 15. Comparative plot of the execution time for the problems previously presented when the `tflite` models are run on a Google Coral Dev Board CPU, and on the Edge TPU.

with the 400 MHz quad-core NXP iMX8 Cortex A53 CPU of the Coral Dev Board is significantly higher compared to the STM32L476, however, we can observe an overhead baseline that hides the real active inference performance. In the STM32 (and in the SPLVP) case, the inference time is directly proportional to the size of the ANN model, while for the Coral CPU, similar to the Edge TPU, this does not apply. We conclude that these high-performance solutions cannot be effectively used in very limited hardware applications with miniature-sized ANN, due to the hardware and software architectural overhead they require.

5) OPERATIONS PER SECOND

A common metric used to quantify computing performance in neural processors is the number of operations completed in a second, abbreviated in OP/s. This parameter is typically reported as peak performance as the real utilization of the hardware depends on the size and the topology of ANN used for inference. This and the related OPs/W metric, however, provided as they are without specifying the underlying hypotheses are considered harmful [70]. Thanks to our `core.py` hardware model used in `sim.py`, we can compute the exact number of operations run by the CPU while running the developed ANN models. Tab. 4 reports the real OP/s used for inference in the seven cases, obtained by referring to the real number of active operations according to the presence of weights equal to zero. To run a single perceptron without options, the number of operations is eight 8-bit multiplications, seven additions (up to 20-bit), and another 32-bit addition for accumulation, overall 16. By including the `acc`, `relu` and `div` options, the device can perform up to 19 operations, but the integer divider requires a larger number of clock cycles compared to standard execution. The peak OP/s of our architecture is indeed 760 MOP/s (1.52 GOP/s if the perceptron were executed in one 80 MHz clock cycle), but the presence of *move* instructions and integer division for scaling that both depend on the topology of the network, leads to lower OP/s. This result is consistent with other accelerators

TABLE 4. Throughput of SPLVP while running the devised MLP models accounting for a reduced active datapath when weights are equal to zero and the exact number of operations per instruction options.

Classifier	Operations per second (MOP/s)
IR Camera	340
Iris	102
SMS	332
Banknote	200
Wireless	228
Ionosphere	246
Avila	293

that in general cannot always exploit their peak performance capabilities, because these depend on the ANN topology they run. Notwithstanding the effective throughput is significantly lower compared to high-end accelerators, it is enough to outperform the MCUs.

6) OTHER ACCELERATORS

It was not possible to directly use the obtained `tflite` quantized files with the MAX78000 toolchain because its `tflite` support is deprecated and limited. However, when similar models are executed using such MCU integrating multiple convolutional engines, we can expect the same latency baseline of Google Coral, although lower in magnitude. Recent works in the literature show that the MAX78000 base execution units typically require two-dimensional inputs and in case less computing resources are required, data is filled with zeros, thus resulting in a latency baseline mostly independent from the operation size (for instance, a network with 4 input channels, 3×3 kernel size, padding of one, and 4 to 64 output channels, leads to a constant baseline latency of $\sim 150\,\mu$s, by far larger than SPLVP) [18]. The MLP models investigated in this work have a small size (the biggest Avila model counts 270 neurons). For such model sizes, PULP Mr. Wolf (Multi-RI5CY) in [30], although working in fixed point, provides a maximum speedup of $8\times$ compared to the Cortex-M4, while SPLVP exhibits slightly higher performance.

To provide a more thorough comparison with the RISC-V-based platform presented in [30], we have considered

TABLE 5. Inference time comparison of SPLVP with respect to PULP Mr. Wolf where MLPs are implemented using the FANN-on-MCU toolkit.

MLP	**SPLVP (60 nm FPGA, 80 MHz)**$^{\uplus}$	PULP Mr. Wolf (40 nm LP ASIC, 32 kHz–450 MHz)		
		IBEX	Single-RI5CY	Multi-RI5CY (octa core)
7, 6, 5	4.075 μs	20 μs	10 μs$^{\diamond}$	4 μs$^{\diamond}$

$\uplus$ = The sigmoid activation function is not supported in SPLVP and it has been substituted by ReLU.
$\diamond$ = In addition, 1–1.3 ms for cluster activation, initialization and activation.

TABLE 6. Inference time comparison of SPLVP with respect to MLPCP in [38] when synthesized with eight concurrent processing-elements (best case).

MLP$^{\oslash}$	Inference time (μs)	
	SPLVP^{a}, Intel Cyclone 10 LP (80 MHz, 60 nm)	MLPCP-4^{b}, Xilinx Zynq 7000 (100 MHz, 28 nm)
9, 2, 6	2.05	4.6
9, 4, 6	2.55	4.9
9, 16, 8, 6	7.55	8.7
9, 40, 6	12.15	13.92
9, 12, 27, 6	11.75	15.93

$\oslash$ = The activation functions of all layers in SPLVP are ReLU. The first entry is the input layer.
a = 8-bit weights/inputs, 32-bit accumulation. b = Fixed point precision (s,w,f), (1,17,8) weights, (1,29,16) weighted sum, and (1,8,4) outputs, where s is sign, w is word size, and f is fraction.

TABLE 7. Inference time comparison of SPLVP with respect to CNN-MLPA in [71] when synthesized for MLP operations, and for different number of PEs.

MLP$^{\oslash}$	Inference time (μs)			
	SPLVP^{a}, Intel Cyclone 10 LP (80 MHz, 60 nm)	CNN-MLPA^{b}, Xilinx Virtex-7 (100 MHz, 28 nm)		
		4 PEs	8 PEs	16 PEs
4, 10, 3	3.2	4.18	4.19	4.54
4, 7, 12, 3	5.22	6.1	6.0	5.88
14, 19, 19, 7	12.1	15.68	12.5	11

$\oslash$ = The activation functions of all layers in SPLVP are ReLU. The first entry is the input layer.
a = 8-bit weights/inputs, 32-bit accumulation. b = Floating point precision, and ReLU activation function.

the reported case C which refers to an MLP with layer sizes 7, 6, and 5, and we have compared the execution time of SPVLP with PULP Mr. Wolf with code generated from the Fast Artificial Neural Network (FANN) toolkit. Tab. 5 shows the obtained inference speed. SPLVP does not support sigmoid activation functions, which we have substituted here with ReLU. Our tiny accelerator outperforms both IBEX and a Single-RI5CY, while it provides almost the same performance as the octa-core Multi-RI5CY. Observe, however, that the RI5CY architectures require a non-recurrent extra latency of 1–1.3 ms for initialization, that however becomes not significant with an increasing number of inferences. On the other hand, SPLVP does not require initialization because the inference is executed while an upstream MCU is writing data to the input FIFO. Given these results, we conclude that our tiny SPLVP is a meaningful solution in applications requiring small-sized MLPs and avoiding the replacement of the MCUs with other more sophisticated architectures.

Tab. 6 compares the inference time of SPLVP with the programmable MLP Co-Processor (MLPCP) in [38], which is clocked at 100 MHz. Each PE comprises a MAC unit, an activation function, an accumulation register, and a separate controller. The significant speedup obtained by SPLVP can be attributed to the machine computing core and the machine's internal memory partitioning. The PE of MLPCP, although fully concurrent, do not provide the same performance as our SPLVP computing unit that sums in a single clock cycle eight computed values with the previous accumulated value without requiring additional scheduling. Furthermore, MLPCP requires external AXI lite and AXI4 stream buses to operate, which are not usually available in MCUs.

Tab. 7 compares the inference time of SPLVP with the programmable CNN-MLP Accelerator (CNN-MLPA) in [71], which can be synthesized using Xilinx Vivado High-Level Synthesis (HLS) to support MLP operations in floating point format (with selectable precision). The system runs at 100 MHz and each PE comprises similar sub-blocks as in MLPCP [38]. To operate with MLPs, the system has been synthesized using ReLU activation functions. For such very small models, the impact of increasing the number of PEs in CNN-MLPA is not significant because they remain underutilized, as the inference time only slightly decreases, while the hardware resources increase linearly. This fact, further emphasizes the usefulness of SPLVP. In the vast majority of cases (that is, except for 16 PEs), SPLVP provides better performance. Similar to MLPCP, this system requires an AXI bus to operate.

Tab. 8 compares SPLVP against low-end FPGA programmable accelerators that target embedded applications, and have comparable arithmetic. For our comparison, we consider architectures that enable inference of non-hardcoded MLP topologies, that can be defined at compilation and programming time. We do not consider the MLP FPGA accelerators that operate on fixed network topologies (for instance, [35], [36]) because the logic resources they require is a function of the implemented MLP size (the larger MLP and arithmetic representation, the larger logic resources). The number of specialized programmable FPGA accelerators targeting our application domain is not substantial because so far tinyML efforts are concentrated on the execution of ANN models directly on MCUs and the development of standalone ASIC or SoCs, that can be aggressively optimized for power consumption. From the comparison, it is evident that SPLVP is the smallest stand-alone low-end FPGA accelerator available in the literature. None of the reported work addresses the problem of MCU interfacing, which is here solved thanks to an ad-hoc 28-pin interface. The accelerators in [31] and [72] provide a configuration and data interface, but no specific solution for MCU interfacing is disclosed. Moreover, the accelerator in [73], which consumes 1.77 W, needs to be interfaced with an ARM processor and a DDR3 controller with an AXI4 bus (that can have a size 32–1024-bit). The maximum throughput achievable with SPLVP is limited by the low number of computing elements it integrates (8 multipliers overall in the proposed implementation), and by the requirement of two clock cycles for a single perceptron execution. SPLVP, however, is a scalable architecture that can be parallelized, or its perceptron parallelism can be expanded. The reported solutions are implemented in different FPGA technology processes which impact power efficiency and clock frequency. Moreover, these accelerators provide a very diverse number of computing elements and different memory sizes. For these reasons, we have elaborated a further parameter to standardize performance evaluation and better position our design. We define a quantity named Operational Density (OD), as $\mathrm{OD} = O_{\mathrm{cyc}}/r$, where O_{cyc} is the number of operations executed per accelerator cycle and r is the count of the 4-input LUTs and DSP blocks normalized with Logic Array Blocks (LAB) of a Stratix-III FPGA, by accounting for the relative ratios reported in [74] in Tab. 2. This quantity is irrespective of the memory utilization, which we assume has little impact on the associated additional logic. While DSP blocks occupy significant resources, registers can be excluded from the count as they are significantly smaller compared to LUTs [74]. This number quantifies the density of the design as a single operation cost in terms of FPGA resources and can be a useful metric in resource-constrained applications, where throughput is not necessarily the main aspect to be considered. According to this metric, SPLVP is the best one among the listed FPGA accelerators. The accelerator in [73] has a comparable figure of merit, indicating a high utilization of the available hardware resources, but consumes considerably higher power, and thus cannot be applied to embedded systems on the Edge.

Compared to MLPCP of Tab. 6, when synthesized using LUTs and eight PEs, SPLVP achieves lower resource occupation. On the considered Xilinx ZynQ 7000 SoC (XC7Z020 device), MLPCP requires 8.2% BRAM, 7.7% DSP, 2.9% registers, and 4.9% LUTs [38], which corresponds to about 11, 17, 3085 and 2606, BRAMs, DSPs, registers and LUTs. Assuming that eight MACs are executed at each cycle (one multiplication and one accumulation per PE), the estimated OD is $40{\cdot}10^{-3}$, which is still lower compared to SPLVP. CNN-MLPA of Tab. 7, requires 18218 LUTs, 6 DSPs, 11670 registers, and 222 BRAMs, for an estimated OD of $6{\cdot}10^{-3}$, where the number of operations has been calculated as throughput divided by the clock frequency reported in Tab. 4 in [71]. SPLVP provides the lowest resource usage. We conclude that SPLVP is the lowest resource occupation programmable MLP accelerator that can significantly outperform an STM32 Cortex-M4 clocked at the same frequency.

VI. DISCUSSION AND CONCLUSION

We demonstrated that tinyML acceleration is a meaningful problem for low-complexity and low-power devices. We designed the SPLVP accelerator to be easily connected to any microcontroller. Even with limited resource utilization, it proved to increase the reference MCU inference performance by a factor of 10×. The SPLVP prototype and its associated toolchain can be still improved in different areas. Thanks to its flexible memory architecture the accelerator can be extended to support online training and other type of feedforward networks. We have demonstrated successful applicability to MLPs through a series of test cases. However, the PE of SPLVP can be also used as is without hardware modifications to implement CNNs, thanks to the application of Toepliz matrices [75] and operation order rescheduling. The CNN support can be further optimized at the instruction level by considering conditional loop options to repeat the application of kernels on the same memory regions.

Given its low resource count, SPLVP can be synthesized also in other low-end and low-power FPGAs that can provide lower power consumption compared to the platform presented in this work. To obtain significant power consumption reduction, which is not possible so far using an FPGA platform, the processor can be synthesized on ASIC as a standalone chip or alongside an on-chip MCU, and scaled by extending the addressable memory space to support larger networks. Moreover, the implementation of SPVLP using mixed-precision floating-point arithmetic can be investigated. Other optimizations that regard both the supervised learning interface and the hardware itself, can be achieved by enforcing power-of-two scaling factors at training time, thus avoiding a hardware integer divider. Other further research work can focus on the implementation of a more aggressive quantization scheme, which can be applied *a posteriori* on the obtained assembly representation so that the hardcoded VHDL description can be synthesized with a minimum number of resources. Moreover, such direct hardware synthesis can be further optimized by plugging in gate-level descriptions of multipliers, adders, and dividers, so that the synthesizer can further descend in the hierarchy and perform an aggressive simplification of what is not needed.

APPENDIX

A. DETAILED HARDWARE ARCHITECTURE

Fig. 16 shows in detail the SPLVP microarchitecture implemented on the low-power FPGA. In the figure, signal naming is associated with each component, hence, we will consider in the description that follows a subset of the depicted signals

TABLE 8. Comparison of SPLVP with state-of-the-art Low/Middle-End FPGA programmable accelerators for Low-Power, tinyML and Edge applications.

	SPLVP (This work)	RAMAN [31]	[72]	[73]
Process (nm)	60	16	40	28
Accuracy (bit)	8	8, 4, 2	8 (activation), 16 (weights)	8
FPGA Device	10CL025YU256I7G Intel Cyclone 10 LP	Ti60 Efinix Titanium	XC7K410T Xilinx Kintex 7	XC7Z020 Xilinx Zynq 7000
Normalized Cost $C_i^{\star}$	1	$1.18^{\diamond}$	70^{f}	4.7^{h}
Design Core Frequency (MHz)	80	75	50	160
Supported Tensors	FC	CONV, DW, PW, Pool, FC	CONV, FC	CONV, Pool, FC
Validation Test Case	MLP	CNN	CNN	CNN
Operations per Cycle O_{cyc}	19 (w. `div`)/18 (w/o `div`)	96	64	256
FPGA Resources$^{\bigstar}$	LUTs: 4513 Registers: 373 DSPs: 8 (4) M9Ks: 66	LUTs: 37.2 k Registers: 8.6 k DSPs: 61 Memory Blocks: $118^{\oplus}$	LUTs: 62.473 k Registers:17.608 k DSPs: 64 BRAMs: 33	LUTs: 34.179 k Registers: – DSPs: 134 BRAMs: 133.5
Throughput S_i (GOP/s)	$0.76^{\sharp}$	$10.5^{\bullet}$	3.2	40.96
Estimated Operational Density (OD$^{\boxdot}$)	$70{\cdot}10^{-3}/66{\cdot}10^{-3}$	$37{\cdot}10^{-3}$	$12{\cdot}10^{-3}$	$62{\cdot}10^{-3}$
Power Consumption P_i (W)	0.287^{*}	$\sim 0.132^{\uplus}$	0.068	1.77
Throughput/Power $H_i^{\dagger}$ [(GOP/s)/W]	2.65	79.68	47	23.14
Estimated Energy per Inference (μJ)	1.79 (Iris)	151.3 (DS-CNN)	7.68 (KWS CNN)	64658 (AlexNet)
Real-time Programmability	Yes	Yes	No	No
Off-chip MCU Interface	Yes (custom, 28 pins)	No	No	No

$\bigstar$ = In the Intel FPGA, the M9K memory size is 9 Kbit, the LUTs have 4 inputs, the DSP blocks are one 18×18 bit or two individual 9×9 bit multipliers (in this design each DSP entry is a 9×9 bit multiplier). In the Efinix FPGA, the Memory Block size is 10 Kbit, the LUTs have 4 inputs and the DSP blocks are 19×18 bit multipliers with 48-bit addition/subtraction. In the Xilinx Kintex 7 and Zynq 7000 FPGA, the BRAM memory size is 36 Kbit, the LUTs have 6 inputs and the DSP blocks are 25×18 bit multipliers with 48-bit accumulator and pre-adder.

$\star$ = Defined as $C_i = \frac{\text{cost}_i}{\min\{\text{cost}\}}$. $\diamond$ = Referred to a manufacturer product number TI60F100S3F2I3. f = Referred to a XC7K410T-1FBG676C device. h = Referred to a XC7Z020-1CLG400C device. $\oplus$ = For DS-CNN. $\sharp$ = Maximum, two clock cycles per perceptron execution, i.e., 19/2·80 MHz. $\bullet$ = Maximum assuming sparsity for DS-CNN. $\boxdot$ = Referred to 4-input LUTs (the Xilinx LUTs are multiplied by a factor 6/4). The number of DSP blocks used for SPLVP is 4 (two individual 9×9 bit multipliers per DSP), to consider a comparable size with respect to the others. $*$ = Static and dynamic power consumption (excluding I/O buffers) estimated with the Quartus power analyzer assuming a switching activity derived from gate-level simulations of the Iris model. $\uplus$ = Estimated static and dynamic power for DS-CNN. $\dagger$ = Defined as $H_i = \frac{S_i}{P_i}$.

i is the device reported on the i^{th} column of the table, cost$_i$ is its the cost available at `digikey.com` in August 2023, P_i is its consumed power when inference corresponding to a throughput S_i is run. CONV = Convolution layer. DW = Depthwise convolution layer. PW = Pointwise convolution layer. FC = Fully connected layer. – = Not Available.

without losing generality. During our initial development steps, we implemented SPLVP assuming different memory capacity starting from a minimum of 8-bit. This allowed programs to address up to 255 memory locations for `D`, `B`, `W`, and `I`. However, from the results of practical applications presented here, we have expanded device memory allowing the execution of larger MLPs. Although the functional units implementation remain unvaried, the hardware implementation of SPLVP changes because the synthesizer infers pipeline stages in the memory blocks that are a function of memory capacity and routing. Memories implementation is typically derived from a built-in Intellectual Property (IP) of the manufacturer. Up to an 8-bit address, the number of pipeline stages on each memory was inferred to one, while for larger memory instances it shifted to two. To solve this issue, we have designed the clocking sub-circuit (top left side of the figure) to generate synchronized clocks, one main clock at 80 MHz, `clk`, and another at twice the frequency, `clk_mem`. Timing-driven synthesis succeeds by simply posing multicycle constraints.

The clocking circuit generates also the reset signal `rst` for all the sequential logic in the processor. The raw clocks `c0` and `c1` are derived from a built-in fabric PLL, which considers the on-board 50 MHz oscillator output as a reference for frequency synthesis. The `Reset Logic` gates these raw clocks until the PLL has reached a steady state by counting for a sufficient number of cycles (resulting in a 1.3 ms delay) after the `locked` signal is asserted. Once this signal is active for the first time it can indeed still toggle before stabilizing. The data incoming from the input interface [`Input Finite State Machine (FSM)`] – herein referred as `data8` – is routed to the proper functional unit depending on the current operational mode. During programming, the `Input Finite State Machine` activates the `prog` signal, and data is routed to the `Programming Path` given in the figure. The interface generates all the addresses required to fill the internal memories. These, are all dual ports RAMs implemented using manufacturer IPs (except for `I` memory which is a single port RAM), to easily handle the particular datapath reconfiguration required for the options `popin` and `pushout`. These dual port RAMs comprise two inputs and output ports named `a` and `b`. Programming is implemented using port `a`. For instance, the firmware in `I` and `B` is uploaded using the signals `i_address_programming` or `b_address_a_programming`. The programming flag `prog` controls several multiplexers to drive all memories' write enable signals `wren` with the corresponding values generated by the interface. During normal operation, `prog` is set to zero therefore leaving the datapath configured in normal execution mode. The `prog` signal also causes the processor controller to be held in reset (until the programming procedure finishes). The system can

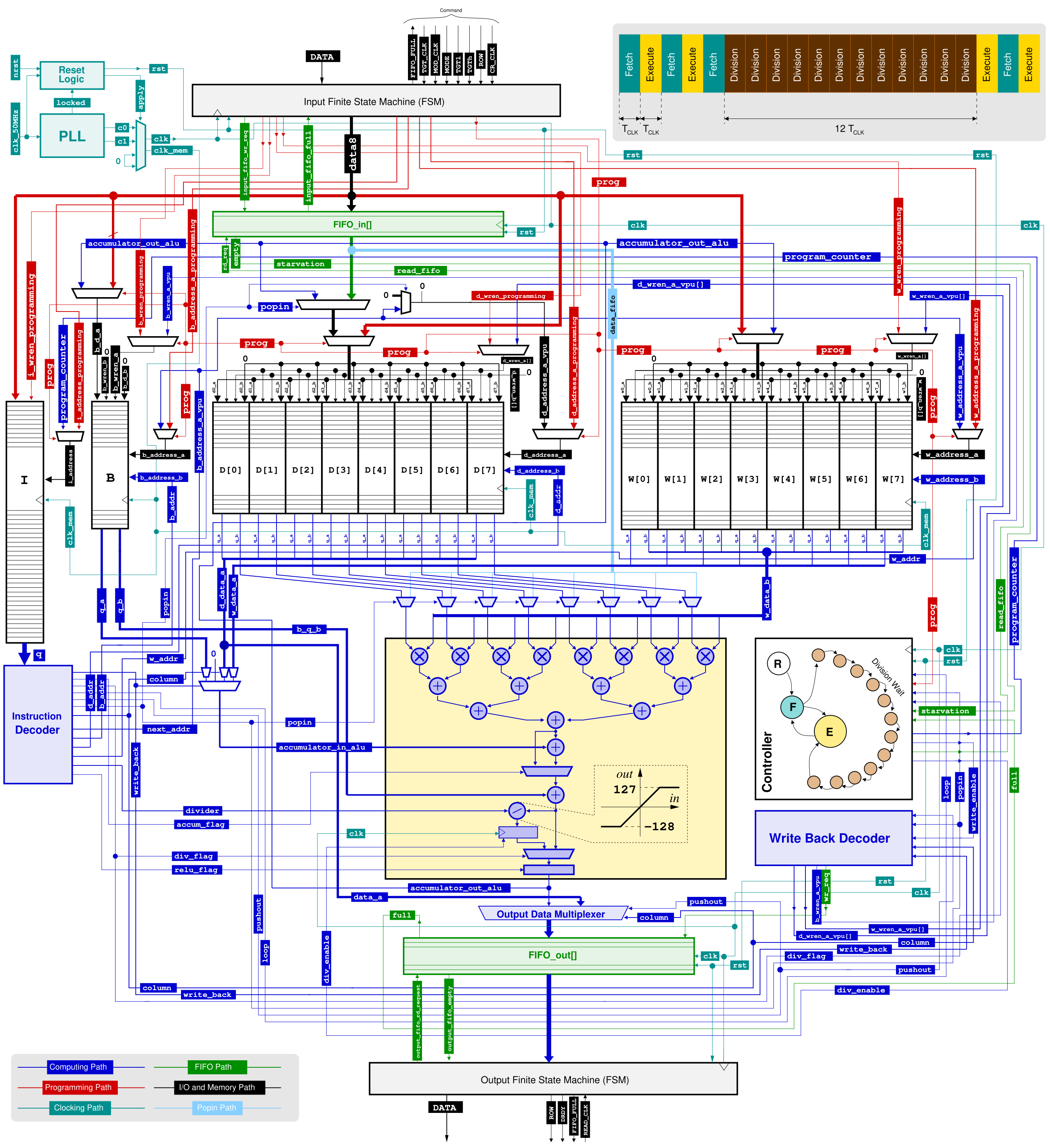


FIGURE 16. SPLVP detailed hardware architecture including input and output FIFO memories and input/output FSM.

be brought in programming mode anytime, even during program execution, based on the commands received by the Input Finite State Machine (FSM). During programming all memories of SPLVP can be selected and incrementally written. The input interface indeed enables the selection of the target memory to be sequentially written, always starting from address 0 × 000.

When programming is finished and the input interface is configured to implement normal operation (compute or run mode), prog toggles to zero. When the signal prog toggles to an active state, the Controller immediately moves to the reset state R. This transition to programming mode is encoded in every active state of the controller, i.e., F, E and Division Wait states. The datapath is reconfigured to address each memory with the pointer specified in the opcode (stored in the instruction memory) and to route the single perceptron arithmetic unit output to the data ports. The `program_counter` is generated by the controller as soon as it leaves its reset state `R` and enters the fetch phase `F`. On the left side of the figure, the `program_counter` is used to address the `I` memory, whose content is available at the next `clk` cycle (corresponding to two `clk_mem` cycles). The opcode outputted on `q` is passed to the `Instruction Decoder` which generates `d_addr`, `w_addr`, `b_addr`, `column` (that is `c` in Lst. 1), `next_addr` (that is `r` in Lst. 1), the write-back code `write_back` which identifies where the perceptron output needs to be saved to, and all the option flags encoded as a single signal each, i.e., `loop`, `popin`, `pushout`, `accum_flag` (for option `acc`), `relu_flag` (for option `relu`), and a division flag `div_flag` with corresponding integer `divider`. These signals are passed to the `Controller` for control execution and to the `Write Back Decoder` that activates the corresponding `wren` signal at port `b` of the corresponding memory to store the output of the single perceptron `accumulator_out_alu`, with truncation in case of 8-bit memory target. To activate the correct `wren` signal the `Write Back Decoder` needs to receive both `column` and `write_back` code. For instance, if write-back occurs on `d(8)(5)`, the decoder activates `d_wren_a_vpu[5]` and keeps all the others to zero, and similarly for the other memories. The `Controller` coordinates the execution of each instruction which is typically completed in two clock cycles, (fetch `F` and execute `E`). However, the implementation of the division (i.e., the fetched instruction includes a `div` option) requires 12 clock cycles of delay due to physical hardware implementation of the integer divider of the FPGA. The `Controller` in this case leaves fetch to run a sequence of `Division Wait` states before entering again the execute state `E`. As shown in the top right diagram of Fig. 16, when division occurs fetch is prolonged by 12 clock cycles, where T_{clk} is 12.5 ns = $1/f_{\texttt{clk}}$. All the other instructions are completed in 25 ns.

In run mode, data sink and sources are implemented using FIFO memories, in the diagram named `FIFO_IN[]` and `FIFO_OUT[]`. These, are written and read based on the external microcontroller activity and their operation is strongly related to `popin` and `pushout` options. During run mode, the input interface atomically writes all the incoming data rows (8 × 8-bit wide) in the `FIFO_IN[]` and SPLVP pops data out of it every time a `popin` option is present in the program. By referring to the `FIFO Path` in the figure, the `Controller` stops execution if two particular conditions occur: i) `FIFO_IN[]` is empty or ii) `FIFO_OUT[]` is full. The `Controller` stops execution and remains frozen in either `F` or `E` states when one of the above conditions is true and restores execution when both become false again. If one of the above conditions is met during a `Division Wait` state, the controller moves to `F` and waits there for the above conditions to become false.

B. INPUT AND OUTPUT INTERFACES

The interfaces are implemented using FSMs. Fig. 17 details the operation of the input interface that implements both programming and writing of input data during inference (`Programming Mode`/`Run Mode`). The interface wires (that are physically connected to the microcontroller) comprise a high byte `DATA` that carries data on a single `int8` byte, and a low byte which comprises control signals. We have chosen to implement mode, data, and target memory write transitions using specific bits and by associating a clock to each one. For instance, the mode is selected using the `MODE` input, but it is validated by positive edge transitions on its corresponding `MOD_CLK`. The target is selected by driving pins `TGTh` and `TGTl` and by validating them through positive edge transitions on `TGT_CLK`. The same logic applies when data is pushed in the processor, through an associated `CR_CLK` (column and row clock). The `ROW` pin indicates the reaching of the last 8th column, and it is used to implement address increment. The input interface provides all signals as inputs, except for `FIFO_FULL` which is output.

During `Programming Mode`, the `MODE` signal is brought high by an external device, and it is validated with a positive edge of `MOD_CLK`. Next, data can be written in the internal memories. To do this, the target data memory is selected using `TGTh`/`TGTl`, which can assume four values (0 × 0–0 × 3 for `D`, `W`, `B`, and `I` memories, respectively). After the target memory selection is validated with a `TGT_CLK` transition, the interface resets its internal address counters to write the specified internal memory. The `CR_CLK` is now used to transfer the data in the high byte of the interface (herein referred to `DATA`) to the target memory (sequentially, column one to column eight). When the last column is transmitted (that can be the 8th, or the 4th for the `B` memory as its width is 4 bytes), the `ROW` signal is also asserted to enable address increment at the next cycle. This process is iterated for each target memory of the processor by setting the `TGTh`/`TGTl` signals accordingly.

After programming is finished, the input interface can be brought into `Run Mode` by setting the `MODE` signal to zero and by validating it using `MOD_CLK`. In `Run Mode`, a target memory does not need to be set because input data needs to be written to `INPUT_FIFO`. Similar to the `Programming Mode`, data can be sequentially presented at the `DATA` input of the interface, and `ROW` can be asserted to indicate the reaching of the 8th column (input data parallelism is 64-bit). During data streaming the input interface can assert the `FIFO_FULL` signal to inform an external system that the `FIFO_IN[]` is full. The streaming must then be suspended until the `FIFO_FULL` signal is de-asserted. From an implementation viewpoint, this signal shall be checked after a complete 64-bit data input is fully transferred, and when `CR_CLK` is deasserted. The signal `FIFO_FULL` is asserted by the device only at the first byte of new data and not in the middle of a transfer. The data streaming speed (ideally 40 Mbyte/s) is limited by the implementation of the input interface state machine, for a maximum speed of

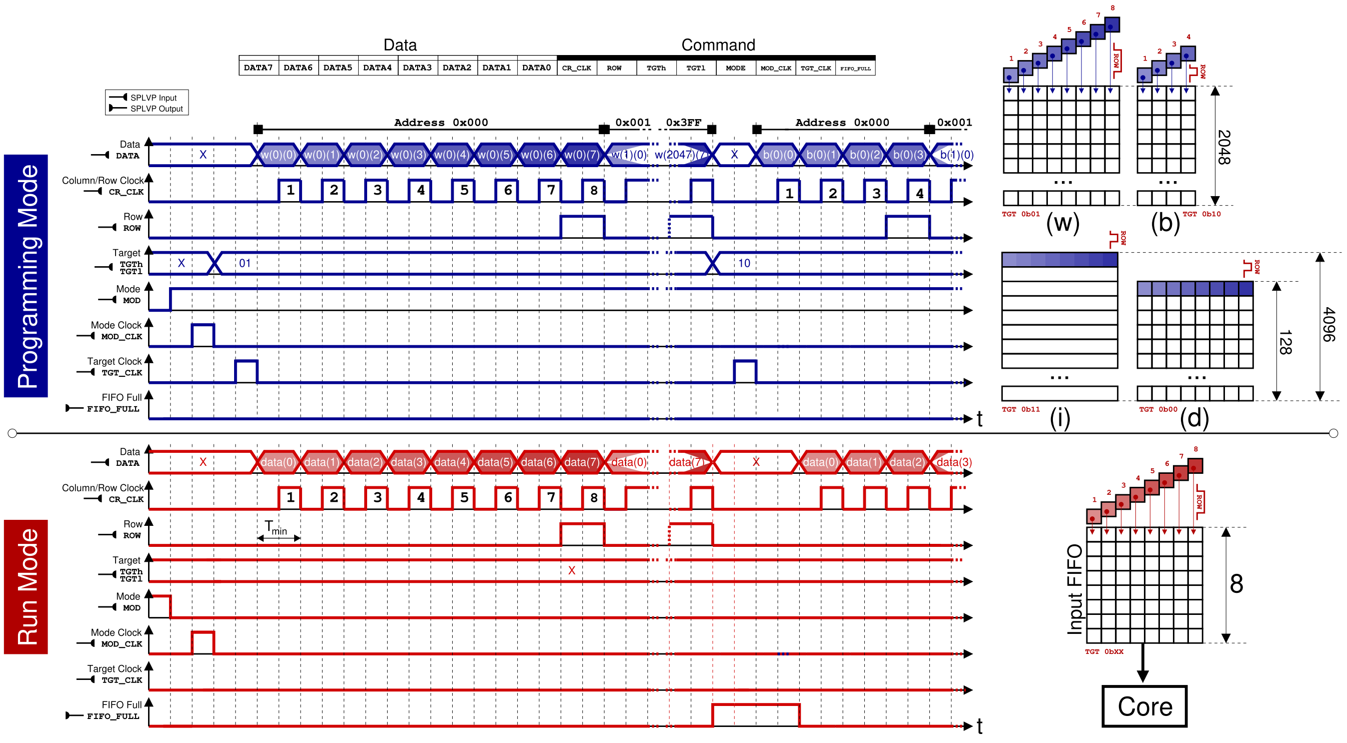


FIGURE 17. Conceptual input interface operation while receiving data from an external device for both programming and running ANN inference.

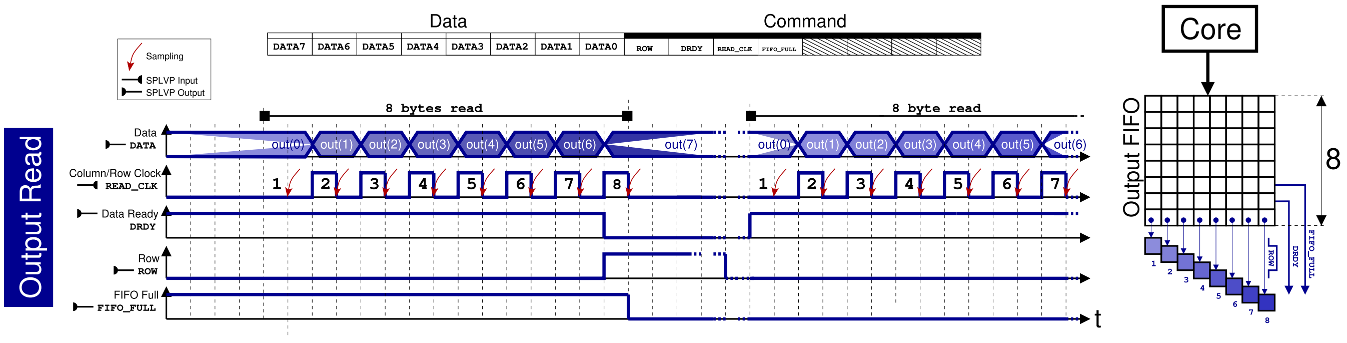


FIGURE 18. Conceptual output interface operation, that is used to extract last layer outputs from the processor to an external system.

$\frac{1}{T_{\min}} = 10\,\text{MHz}$, thus resulting in a throughput of 10 Mbyte/s. Considering the maximum clock frequency of common low-power microcontrollers (for example the STM32L476 used in our comparison measurements that runs at 80 MHz) and that some clock cycles are required to drive the GPIO signals, we believe that the speed of our interface is adequate, and does not represent a bottleneck. GPIOs data registers indeed typically group multiple pins, hence a single instruction can set or read several pins.

Fig. 18 details the signal timing of the output interface of SPLVP. The output interface is used to read the output data from the processor as soon as it becomes available. The interface has one input, READ_CLK, which is used, similar to CR_CLK, to pop each byte out from the output FIFO. Data is presented on the DATA output port. An external system needs to assert positive edges on CR_CLK every time it is ready to accept a new byte. The last byte (corresponding to the 8th column of d(127)), is indicated by the device by asserting the ROW signal. Control flow is implemented by two separate signals, FIFO_FULL and DRDY (data ready). When FIFO_FULL is asserted, data is available for reading, but it indicates that the processor cannot calculate further results because the FIFO_OUT[] needs to be flushed first. The signal DRDY, instead, indicates the presence of new data on the FIFO_OUT[]. The downstream logic shall not assert the READ_CLK as long as DRDY is not asserted. On the other hand, FIFO_FULL is activated by SPLVP only in connection with the first byte of a new transfer, therefore making a single 64-bit data atomically identifiable and thus avoiding interruptions in the middle of an 8-byte transfer.

The assembler tool can generate both a binary program file to be used for programming the internal SRAMs, and a frozen binary file that can be used to stream a fixed sequence

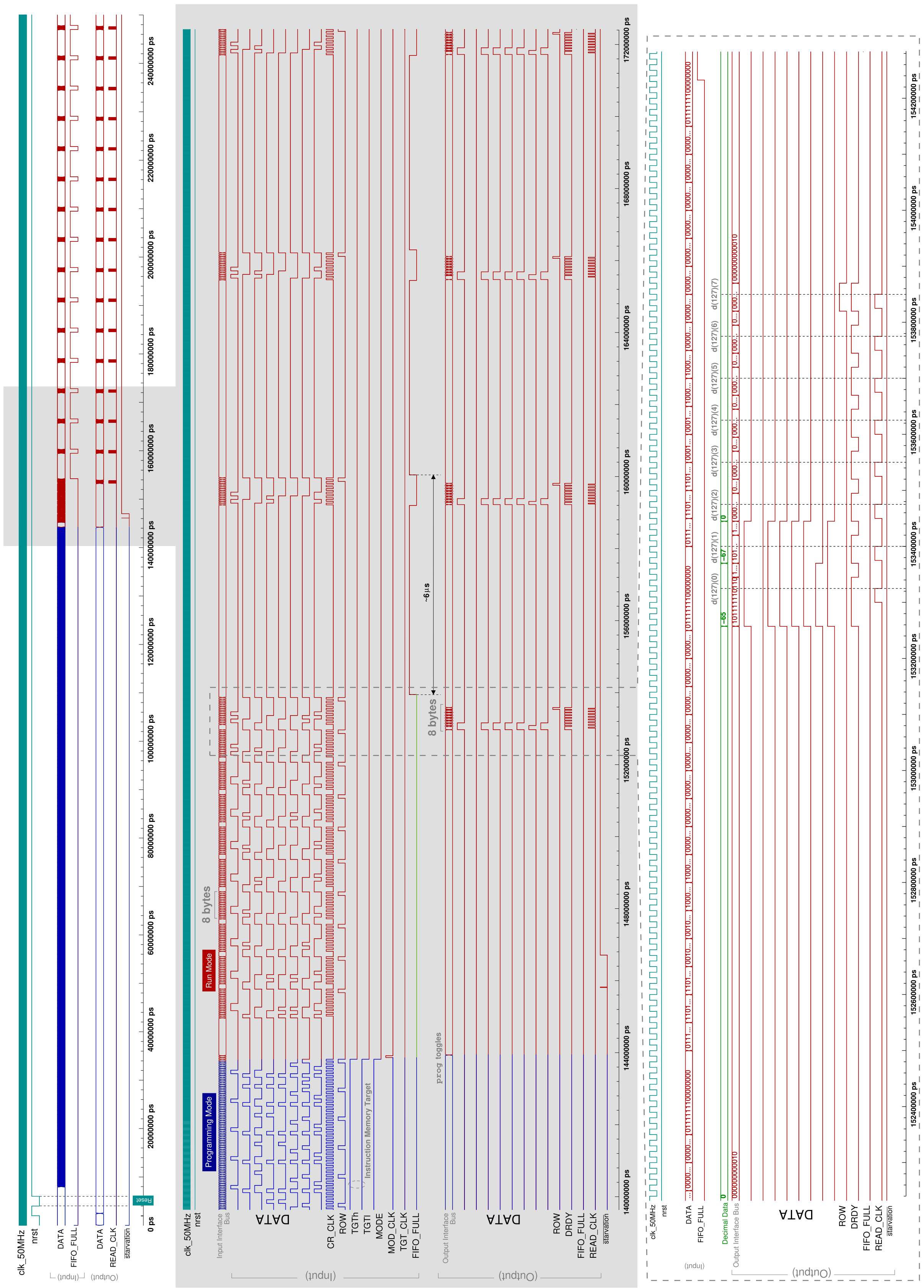


FIGURE 19. Gate-level post-synthesis simulation of SPLVP obtained using ModelSim Intel FPGA Starter Edition while running the Iris model of Sec. V-C1, during both programming and run mode, depicting a correct `Decimal Output`. The signal `prog` which toggles when `Run Mode` is set, is included in the `Output Interface Bus`, but it is not shown.

of inputs for testing purposes. The content of these files represents the direct encoding of the signaling introduced in Fig. 17 in a binary format, including clocking as well, that is simply implemented by duplicating each data entry, with zero-to-one and one-to-zero transitions on CR_CLK, TGT_CLK and MOD_CLK. This way, the content of the binary files can be simply streamed across the pins of a microcontroller.

C. DIGITAL SYSTEM-LEVEL SIMULATION

Fig. 19 shows a gate-level simulation of the complete SPLVP, after synthesis. In this simulation, we have tested the complete operation of the system, including input and output interfaces and we have included a particular signal $\overline{\text{starvation}}$ in an available pin to measure the active computation time of SPLVP. We have considered the Iris model of Sec. V-C1 as a test case, and we directly applied the program code and the inputs generated by our toolchain as presented in Sec. IV. This is done, in particular, by directly streaming the binary files generated by the assembler with a custom testbench that emulates a system connected to the microprocessor. Not to waste simulation time, the SPLVP description here includes a reset logic that simply counts for a few cycles to permit the PLL model to lock. The simulation starts at $t = 0$ s with signal nrst low, and the testbench performs another reset cycle to make sure all internal logic is correctly initialized. After 8 μs, the testbench opens the programming file bitstream.bin containing the SPLVP firmware and streams it directly on the input interface at the maximum speed possible. Using a 20 MHz clock, we obtain 10 Mbyte/s of effective throughput (each of the 16 input interface bits is updated every 50 ns, but CR_CLK needs two transitions per byte transfer). During programming, all target memories are updated using the signals TGTh, TGTl, and TGT_CLK. As shown, the last one that is written is instruction memory. After the last instruction memory byte is transmitted, the system is immediately set in Run Mode by de-asserting the MODE signal and by enforcing a double edge transition on MOD_CLK.

Similar to programming, during Run Mode the testbench streams input data from the frozen file frozen.bin directly on the input interface at maximum speed. After the ROW signal is asserted, eight bytes are fully transmitted to the input interface, and the testbench needs to wait for a further 100 ns to check if the FIFO_FULL signal is active. If this condition occurs, it indicates there is no room for further data in the input FIFO, and the testbench needs to wait until FIFO_FULL is de-asserted to continue streaming. At the output interface, after the first eight bytes are transmitted, the testing signal $\overline{\text{starvation}}$ has a glitch, that would indicate that the input FIFO is empty. However, this glitch is present simply because the signal is not sampled by the main clock, and does not impact the functional behavior of the machine. When eight bytes are completely transmitted, $\overline{\text{starvation}}$ is asserted and within the streaming of the 9th byte, signal DRDY goes high at the output interface to indicate that eight new bytes are available to be retrieved. The testbench, which checks every 50 ns the value of this signal, can now toggle the READ_CLK to sequentially acquire the eight bytes d(127)(0)–d(127)(7) that correspond to the eight outputs of the ANN model. Immediately after data is retrieved, the FIFO_FULL signal at the input interface is asserted, to indicate that the processor cannot continue execution. As previously introduced, the testbench stops streaming new input data. Within $\sim 6\,\mu$s, a new vector is made available at the output interface, and the testbench after reading it permits SPLVP to continue execution. Consequently, the FIFO_FULL signal at the input interface is de-asserted. Execution continues periodically every $\sim 6\,\mu$s, that is the inference time of the model reported in Sec. V-D. The signal $\overline{\text{starvation}}$ is never deasserted to indicate that the SPLVP is continuously running without interruptions thanks to the responsiveness of the testbench in the implementation of data streaming and output reading.

D. COMPILER IMPLEMENTATION

1) QUANTIZATION AND EXECUTION MODEL

This section provides a high-level overview of the operation of the compiler with an example FC classifier with three layers. The goal of the compiler is to provide compatibility with tflite with a low number of approximations.

Fig. 20(a) shows a logical execution model of an example FC ANN having three layers, Dense$_0$–Dense$_2$, with $N_0 = 4$, $N_1 = 3$, and $N_2 = 4$ outputs, respectively, where the input layer has size 4. Overall this example model has three operators, each with a given input and output scale factor. According to the specification, the TF quantization scheme is an affine mapping of quantized integer values q to real numbers r, according to the equation $r = S(q - Z)$, where S and Z are some constants, that in this context are named scale factor and zero, respectively. This mapping applies to each operator's input and output, in our case the layers of the neural network. In this example, we can then identify the quantities r_i and q_i, $\forall i \in [0, 3]$, where we define i as level. The Q and Q^{-1} formulas given in the figure can be used anytime to pass from a real (float32) representation to int8 for a given level. The quantized tflite model is made (by the TF software library) such that the output scale factor of one operator corresponds to the input scale factor of the next one. This way inference execution can avoid the reconversion of each operator's quantized output into its real counterpart, at least for the internal layers. To achieve inference using the tflite methods, it is required that input data is *manually* quantized before feeding it at the input of the first layer and the inference output is required to be reconverted back to real numbers at the output of the last layer (refer to tflite input and tflite output in the figure). This conversion and re-conversion process implies that inference on a quantized tflite model, when considered from an *input data perspective*, is a mixed integer/float one as scaling is in general a real number [13].

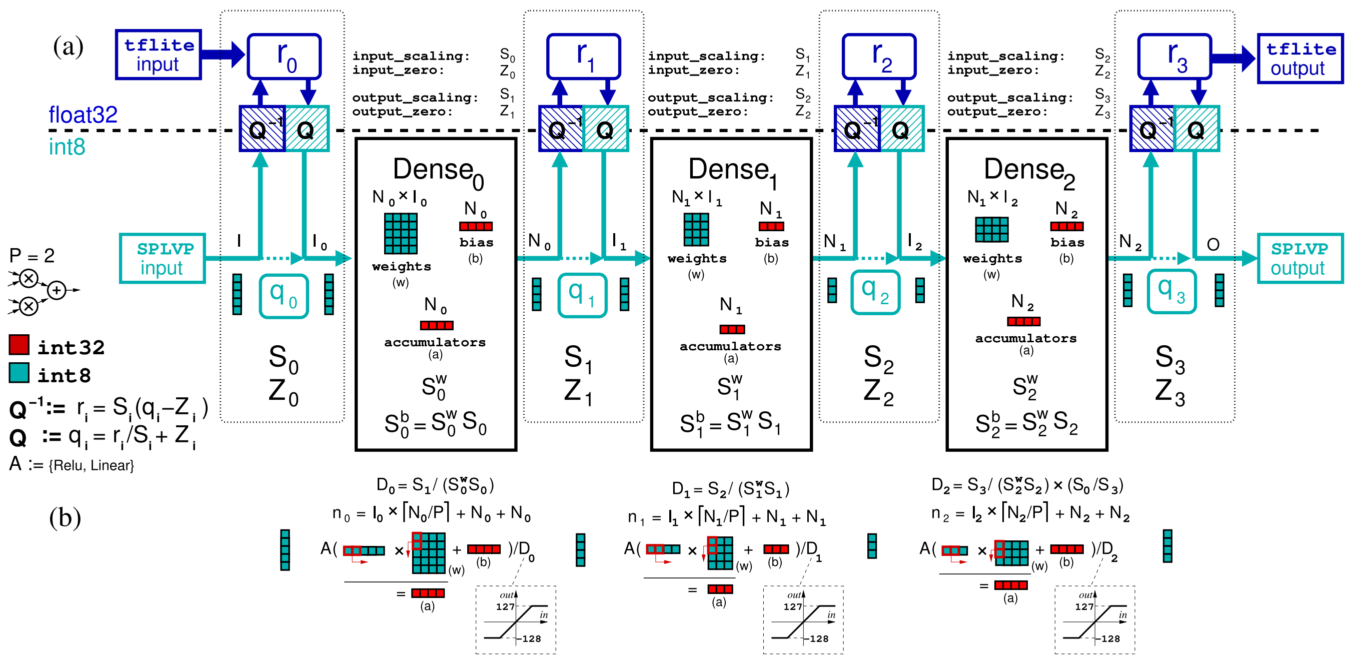


FIGURE 20. Conceptual workflow of the compiler on a `tflite` model translation to SPVLP assuming for simplicity a perceptron with parallelism $P = 2$ (a) and relative compiled output representation (b), as a function of on the internal quantized model parameters.

We assume that supervised learning is achieved with input data already pre-quantized in the `int8` range. We have chosen here to *directly apply* the quantized inputs to the network, therefore avoiding this first conversion step (refer to `SPLVP` input and `SPLVP` output in the figure). This way the output of the ANN *may not* exactly correspond to the output of `tflite`, but this way we can avoid the need for a floating-point arithmetic logic unit in the microcontroller to be accelerated. It is worth observing that the SPLVP output difference compared to the `tflite` output can be considered negligible, if training is made such as S_0 approximates one, and Z_0 approximates zero as much as possible. As shown in Fig. 20(a), in each operator, biases are `int32` numbers, and consequently, intermediate accumulations need to be memorized in an `int32` format because biases are applied directly to the dot multiplication output. According to the quantization specifications in [49], the scaling factors of weights and bias tensors are inter-related by construction through the input scaling factors, in particular $S_j^b = S_j^w S_j$, where S_j^b is the bias scale factor, S_j is the input scaling factor and S_j^w is the weights scale factor, at the operator jth.

After the accumulations in the `int32` range and the application of bias and activation function are all completed, the final output needs to be rescaled back to `int8`. To do this, in [49] the authors propose to multiply these values by a quantity (corresponding to the inverse of D_j in our scheme) that can be easily split into a first fixed point multiplication of a number in the range [0.5, 1] and a second 2^n division. This last one, in turn, can be easily implemented in hardware using a barrel shifter. However, as our goal is computing using only integer arithmetic, we have chosen to rescale accumulations using the SPLVP integer divider where the divisor approximates the real quantity D_j. This solution introduces a further variation on SPLVP inference compared to the original `tflite` model, but as demonstrated in our tests this is not significant. The divisors depicted in Fig. 20(b) are calculated for each operator using both the input and the weights scale factor S_j and S_j^w, except for the last layer in which the divisor is computed differently. This exception is done to recover the variation introduced by avoiding the first quantization step at the input layer. The last layer rescaling factor D_2 is indeed multiplied by S_0/S_3. This way, even if the input layer scaling factor is not one, we can partially recover its effect by scaling the output values. For zeros, we assume they are small and hence that they do not significantly impact accuracy. Observe that this is an approximation and this technique does not lead to the same accuracy obtained with the mixed `tflite` inference, compared to the original `float32` model. Observe, however, that S_0 cannot be very small by construction because we directly apply fake `int8` quantized data to our supervised learning interface when training the `float32` model. Finally, the quantization specification in [49] requires saturated arithmetic, and this constraint is implicitly met by the saturation features of the integer divider of SPLVP. We do not, however, implement saturation for `int32` numbers because we assume this condition is unlikely to happen. Fig. 20(b) shows a graphical representation of the execution of the inference assuming a single perceptron capable of handling two weights at a time (parallelism $P = 2$) that we have considered in this example. In general, given a jth dense operator, the number of required perceptron executions n_j can be calculated with the formula given in the figure.

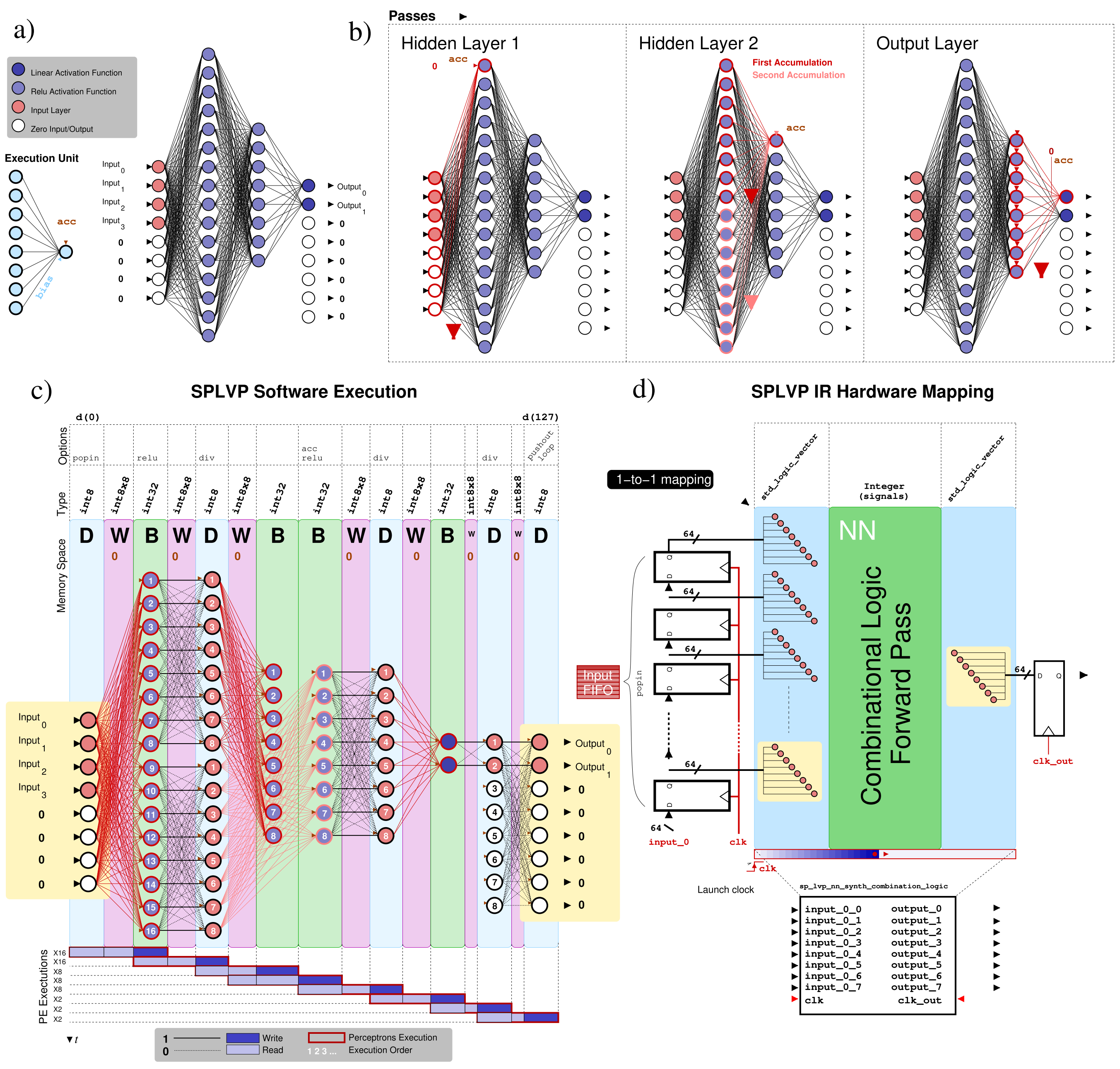


FIGURE 21. **Workflow for the execution of the Iris MLP classifier (see Sec. V-C1) and for the direct translation of the ANN into a combinational VHDL description, assuming that inputs are stored in dedicated registers.**

2) OPERATION SCHEDULING

To describe the internal operations of the compiler, we first summarize the technical operations necessary to decode the flatbuffer input. Next, we consider an example MLP and we detail the conceptual execution of its operations by assuming the single perceptron unit with parallelism $P = 8$ of our physical implementation.

The operations performed by `compile.py` after reading the `tflite` file as input can be summarized as follows. First, the `tflite` flatbuffer is loaded using the standard `tflite` interpreter that returns a list of dictionaries where each one refers to a tensor, and not necessarily an operator (layer) used in the multilayer perceptron. Depending on the type of activation functions, the version of TF, and in general the structure of the neural network, information on each operator can be fragmented and needs to be reconstructed, so that each layer in the network has all the parameters of Fig. 20 defined and stored in an internal descriptor. For instance, signatures and activation functions may be represented with different entries, but in any case, they need to be associated with the same layer. A possible approach to rearrange these entries consists of parsing the names of each tensor (which typically embeds a hierarchical tree relationship), but some input and output scaling information may be impossible to reconstruct. The standard `tflite` interface used for inference can be used to extract internal information on the ANN, but data needs to be completed with other information that is typically not accessible using the standard methods. To complete the compiler's internal descriptors we then wrote an internal `flat_buffer_helper` module that considers the complete information on the flatbuffer file and reconstructs the graph of the network with fused activation functions. During this first step, `compile.py` stops execution and outputs errors if unknown tensors are present in the network.

The next operation of the compiler is the sequential processing of all the layers in the ANN structure to convert the `tflite` structure to an intermediate representation. This way, the ANN can be described in terms of a single perceptron unit operation scheduling, to be executed in a single forward pass. The compiler here works only for a single ANN subgraph (in complex networks many subgraphs are possible), so we always expect that the multilayer perceptron has a single input layer and a single output layer. The tool starts from the input layer and groups the inputs in zero-padded chunks of size eight, then it schedules one inference for each chunk. The number of chunks obtained this way is indeed the number of `popin` options used in the code because the input FIFO of our hardware has a parallelism of eight as well. Observe that this chunking operation is scalable and can be applied for different sizes as it only requires that both the PE and the input FIFO data parallelism match. As we have decided that the maximum output parallelism the compiler can handle is eight (the hardware, however, does not pose limitations), the output layer is not divided into chunks. `compile.py` raises an error if the number of outputs of the `tflite` ANN exceeds this limit.

After input rearrangement, the compiler executes the operation scheduling that is exemplified in the workflow of Fig. 21. Let us consider the MLP given in Fig. 21(a), which we have used to solve the classification problem of the Iris dataset. The ANN has an input layer of size eight, two hidden layers of size 16 and eight with ReLU activation function, and an output layer of size two with linear activation. The objective of the compiler is to sequentially schedule the computing unit named `Execution Unit` (that includes an accumulation `acc` input as well) in the graph to perform a feedforward pass starting from the input layer to the output layer. As shown in the graph, both input and output layers are zero-padded, that is, the non-used inputs and outputs are zero. For simplicity, we here refer to a simple perceptron computing unit, while the hardware counterpart includes also the integer divider and the ReLU arithmetic logic.

Computation can be graphically represented as a sequence of passes per layer as shown in Fig. 21(b). If we refer to the `Hidden Layer 1`, the single perceptron can be sequentially applied following the arrow to cover all the nodes of the hidden layer, by enforcing `acc` $= 0$. In this example, the size of this first layer is larger than the input layer, therefore the single perceptron is applied only once per node, by using multiple times the input layer. Observe that in this forward pass, the order with which nodes are computed is not important because the computing unit is linear. When the computing of this first layer is over (that is the system has eventually applied the `relu` option and scaling), the results can then be used as inputs for the second layer `Hidden Layer 2`. In this case, however, the parallelism of the input data is larger than the one of the output, and therefore the computing unit needs to be scheduled more than once per output node. The `acc` input is zero for the first accumulation, but it is the previous accumulated value for the second one. After the application of the activation function and scaling for each node, finally, the output layer is processed in a similar way compared to `Hidden Layer 1` using a linear activation function.

The operations outlined above need to be executed in our single perceptron-based accelerator by exploiting its internal architecture and hardware units. Fig. 21(c) shows the same ANN executed according to the high-level scheme of Fig. 21(b), on SPLVP. This plot shows the execution timeline of the forward pass. It details, for each step, the logic utilization of memories, the associated data types, and the options used in the assembly instruction. From left to right, inputs are presented in an `int8` format from the input FIFO and are contextually stored in the `D` memory at address `d(0)`. To run the computation of the first hidden layer, all inputs are taken in parallel by the perceptron (four inputs are zero-padded) in the order given in the figure, from 1 to 16, with no accumulation (`acc` option is absent). The results are stored in the `B` memory as all the accumulations are 32-bit integers. Here, we have chosen to implement the activation function directly during this accumulation pass (`relu` option). However, `relu` could have been computed in the next step as well, because it is simply implemented by checking the data sign, which in any case remains unchanged after an unsigned division. After accumulation, the hidden layer outputs must be rescaled to `int8` and therefore the single perceptron is executed once again to move accumulator values from the `B` memory to the `D` memory by sequentially applying *move* weights. These, are simply sparse weights vectors with '1's in the positions that need to be copied from source to destination and '0's otherwise. During this operation rescaling is executed with the `div` option (the divisor is not made explicit for the sake of brevity) so that the accumulated data is normalized in the `int8` range.

The computation of the second hidden layer is performed similarly compared to the first one, with a difference in the accumulation process. During the first accumulation pass the `B` memory is overwritten with `acc` $= 0$, in the order given in steps 1–8. For the second pass, the previous accumulation is used as `acc` input, and data is rewritten in the same `B` memory locations by applying the activation function. For this second step, indeed the options used in the assembly are `acc` and `relu`, and this last one finally applies activation. It is noteworthy to observe that using such computation flow and by assuming that data is

```
com d(0) w(0) b(6) act lin to b(2)(0) [popin]
com d(0) w(1) b(6) act lin to b(3)(0) []
com d(1) w(2) b(0) act lin to b(2)(0) [acc relu]
com d(1) w(2) b(1) act lin to b(3)(0) [acc relu]
com d(1) w(2) b(2) act lin to d(2)(0) [div47]
com d(1) w(2) b(3) act lin to d(2)(1) [div47]
```

LISTING 2. Example SPLVP assembly of a layer with size two with ReLU activation function and integer scaling 47.

```
accum_2_stage_0 <= data_0_0_0 * weight_0_0 + data_0_1_0 * weight_0_1 + data_0_2_0 * weight_0_2 +
  data_0_3_0 * weight_0_3 + data_0_4_0 * weight_0_4 + data_0_5_0 * weight_0_5 + data_0_6_0 * weight_0_6
   + data_0_7_0 * weight_0_7 + bias_6;
accum_3_stage_0 <= data_0_0_0 * weight_1_0 + data_0_1_0 * weight_1_1 + data_0_2_0 * weight_1_2 +
  data_0_3_0 * weight_1_3 + data_0_4_0 * weight_1_4 + data_0_5_0 * weight_1_5 + data_0_6_0 * weight_1_6
   + data_0_7_0 * weight_1_7 + bias_6;
accum_2_stage_1 <= data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 *
  weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 *
  weight_2_7 + bias_0 + accum_2_stage_0 when (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2
  * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 *
  weight_2_6 + data_1_7 * weight_2_7 + bias_0 + accum_2_stage_0) > 0 else 0;
accum_3_stage_1 <= data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 *
  weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 *
  weight_2_7 + bias_1 + accum_3_stage_0 when (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2
  * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 *
  weight_2_6 + data_1_7 * weight_2_7 + bias_1 + accum_3_stage_0) > 0 else 0;
data_2_0_div <= (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 *
  weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 *
  weight_2_7 + accum_2_stage_1 + 47/2)/47 when (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 +
  data_1_2 * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 +
  data_1_6 * weight_2_6 + data_1_7 * weight_2_7 + accum_2_stage_1) > 0 else (data_1_0 * weight_2_0 +
  data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 +
  data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 * weight_2_7 + accum_2_stage_1 - 47/2)/47;
data_2_0 <= 127 when data_2_0_div > 127 else -128 when data_2_0_div < -128 else data_2_0_div;
data_2_1_div <= (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 *
  weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 *
  weight_2_7 + accum_3_stage_1 + 47/2)/47 when (data_1_0 * weight_2_0 + data_1_1 * weight_2_1 +
  data_1_2 * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 + data_1_5 * weight_2_5 +
  data_1_6 * weight_2_6 + data_1_7 * weight_2_7 + accum_3_stage_1) > 0 else (data_1_0 * weight_2_0 +
  data_1_1 * weight_2_1 + data_1_2 * weight_2_2 + data_1_3 * weight_2_3 + data_1_4 * weight_2_4 +
  data_1_5 * weight_2_5 + data_1_6 * weight_2_6 + data_1_7 * weight_2_7 + accum_3_stage_1 - 47/2)/47;
data_2_1 <= 127 when data_2_1_div > 127 else -128 when data_2_1_div < -128 else data_2_1_div;
```

LISTING 3. Synthesized VHDL description corresponding to the assembly code given in Lst. 2. The `popin` option is intrinsically embedded in the values of the input data.

otherwise stored in different memory locations, the network can be considered combinational, even in the presence of accumulation. This fact is useful to automatically convert the MLP into synthesizable code. Exactly as for the first hidden layer, *move* weights here are applied to write the `D` memory again and scale the accumulations to `int8` (`div` option). The last output layer is executed similarly to the previous ones and the output result is stored in the `B` memory by executing twice the computing unit (marked as 1 and 2). Finally, the accumulations are scaled down and moved to the `D` memory space using the *move* weights (unused outputs are zero-padded). As a final mandatory step for the execution of the MLP, the output layer data needs to be stored at memory address `d(127)`, and a `pushout` option is issued to write the output FIFO. Contextually, a `loop` option is used to restart the program counter and therefore execute the MLP with new data from the input FIFO. Observe that the internal `D` and `B` accumulated values do not need to be reset because they are overwritten thanks to the absence of the `acc` option during the first accumulation stages. This strategy is useful for the implementation of feedforward MLP, although keeping accumulation active may be useful to implement Recurrent Neural Networks (RNNs). These, however, are not part of this work.

3) DIRECT LOGIC SYNTHESIS

All the assembly `com` instructions can be directly mapped one by one to implement a synthesizable VHDL description of the network, hence avoiding the use of the SPLVP core. Such a VHDL network simply needs to receive input data in the same order given by the model to implement inference. Fig. 21(d) shows a high-level scheme of the VHDL description. Because the accumulator steps can be unrolled as shown in Fig. 21(c), assuming data is all saved in different memory locations every time the computing unit is applied, the ANN assembly can be seen as a Direct Acyclic Graph (DAG). In the depicted example, the generic shift register used to feed the combinational logic network is made of only one register because the input layer has a size four, and the MLP requires a single input to complete the inference. In our proof-of-concept converter, we have mapped each node as an `Integer`, whose implemented number of bits, is synthesizer-dependent. The Quartus synthesizer considers integers at 32-bit, but in general integer mapping impacts the inference accuracy. To overcome this issue, signals can be declared using the VHDL `numeric_std` library in an arbitrary number of bits, thus providing a fully disclosed mapping. Our choice goes in favor of further area optimization that may be required while deploying the neural network. Constraining the accumulations (for instance at 16-bit relying on the synthesizer), contributes to the reduction of circuit area, provided that simulations show that accuracy constraints are still met.

The eight-byte data received from the SPLVP input FIFO can be sequentially fed to a shift register having parallelism 64-bit and depth $\lceil \frac{n_I}{8} \rceil$, where n_I is the size of the input layer. For ease of implementation and timing analysis, in our description, we enforce two different clocks `clk` and `clk_out` for feeding the input data and acquiring the output data. `clk_out` is used to sample the MLP output using a single dedicated register of size 64-bit (i.e., 8×8-bit), as the compiler supports only eight parallel outputs. With these assumptions, the timing analyzer of the FPGA synthesis tool can efficiently compute propagation delay using multicycle path and false path constraints. By default, the SDC code is generated assuming 50 MHz and 1 MHz frequency for `clk` and `clk_out`, respectively. Providing data to the hardware unit through a register pipeline has the advantage of maintaining a low number of pins and the same entity declaration irrespective of the complexity of the internal ANN graph. The VHDL testbench automatically generated by `clsynth.py` implements the feeding of the internal pipeline whose depth is based on the input test data parallelism, and hence needs to be customized at compile time because it depends on the ANN model. Furthermore, it instantiates the component whose entity is given in Fig. 21(d). At runtime, it reads the CSV files associated with the testing inputs at build time, converts them into input logic signals, applies them in the correct order, and saves output data from the output register to another CSV file.

Lst. 2 shows a snippet of the assembly code of Fig. 3, implementing a perceptron with size two, ReLU activation function, and integer scaling 47. The code has been synthesized with `csynth.py`, and the corresponding output is shown in Lst. 3 (we do not show the entire code here for the sake of brevity). Lines 1–4 of Lst. 2 are mapped to lines 1–4 of Lst. 3, while lines 5 and 6 are mapped to lines 5–6 and 7–8, respectively. The referred ANN has an input layer of size eight. Our tool statically mangles the memory names of the assembly, using the following convention. `d(x)(y)` memory for `x` $\neq 0$ is mapped to `data_x_y`, while for `x` $= 0$, it is mapped as `data_0_y_w`, where `w` identifies the shift register depth where input data is written. In this example, we have only eight inputs, and therefore the synthesized shift register is simply a register. The eight values of `d(0)` are then mapped to `data_0_0_0`–`data_0_7_0`. The input signals of Fig. 21(d) are simply sampled on `clk` positive edges and stored in these signals. A generic weight `w(x)` is encoded as `weight_x_c`, where `c` ranges from 0 to 7. The accumulators `b(x)(0)` are mapped to `accum_x_stage_z` where `z` is a number that starts from zero and it is sequentially incremented every time a new write on the same accumulator occurs in the assembly. With these defined signals accumulation can be identified as a unique signal in the VHDL code, thus permitting a DAG description of the network. For the same reason, after rescaling (given by the presence of a `div` option in the assembly), a suffix `_div` is appended to the data memory names to maintain uniqueness in the signal naming. Given such a naming convention, the translation of the assembly code is straightforward, and the network can be easily generated. When an `acc` option is used, for instance, it is sufficient that an `accum_x_stage_`$i+1$ considers `accum_x_stage_`i as input. When a `relu` option is present, `csynth.py` uses a `when/else` conditional statement to implement the greater than zero comparisons. With rescaling, in the same way, a conditional statement is used to implement rounding in the division based on the sign of the dividend. After rescaling, the final values `data_2_0` and `data_2_1` are declared with a conditional statement to clamp the output value in the range $[-128, 127]$ thus implementing `int8` saturation.

E. ASSEMBLER IMPLEMENTATION

The assembler tool `asm.py` reads a text file with extension `.asm` and generates i) the firmware binary file suitable for direct streaming across the input interface for programming the processor, ii) an optional frozen binary file to stream fixed data to the input interface for testing purposes and iii) the MIF file snapshot of the memory. Internally it comprises a tokenizer, a multi-pass parser, and a code generator. The parser checks the syntax of the assembly file, and detects the presence of errors, for example memory cells other than `d(0)` in the presence of `popin` options. Both the bitstream file and the frozen input file include a dump of the logical values of the pins of the input interface. Generating a binary file ready to be bit-banged simplifies the testing of the interface and allows an MCU to easily implement programming. The MIF files, instead just include all the content of the `D`, `W`, `B`, and `I` memories ready to be loaded by the simulator. These files are used to populate the memory models of the physical hardware emulator in `core.py`.

F. SIMULATOR IMPLEMENTATION

Fig. 22 shows a high-level flow chart of the operation of `sim.py`. Its execution is customizable using specific command line arguments. In its standard flow, it runs inference for both a quantized `tflite` model (using the locally installed TF library) and for the corresponding compiled version for SPLVP to obtain performance comparison. At program start, after command line argument parsing, the MIF files including the processor internal memory snapshot are read and the internal device memory of the hardware model in `core.py` is virtually programmed. Next, the simulator opens the CSV file containing the input testing data to be used for inference. The file can be specified by the user at the command line

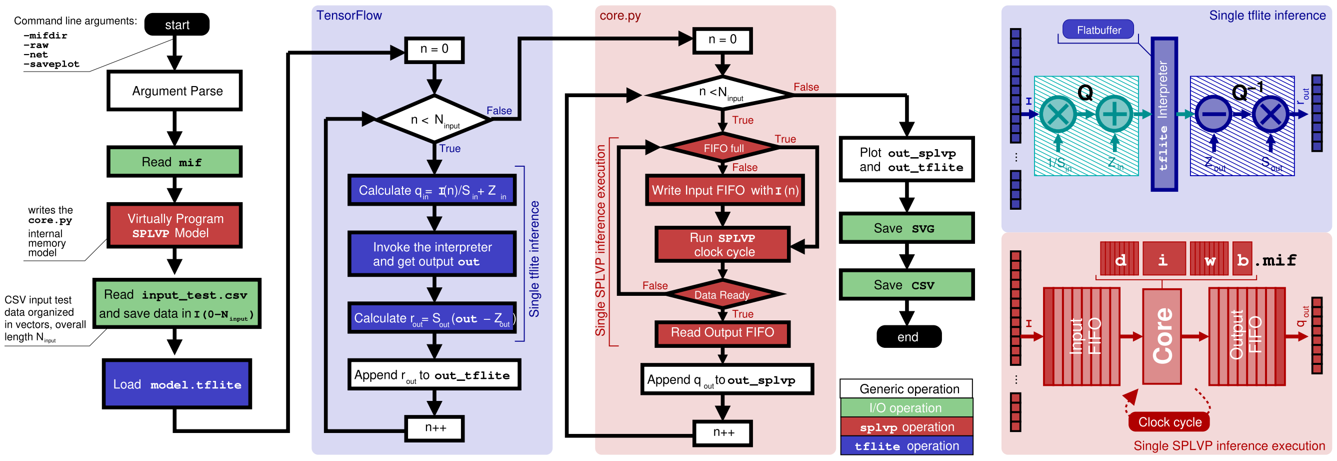

Command line arguments:
-mifdir
-raw
-net
-saveplot
start
Argument Parse
Read mif
writes the core.py internal memory model
Virtually Program SPLVP Model
CSV input test data organized in vectors, overall length N_input
Read input_test.csv and save data in I(0–N_input)
Load model.tflite
TensorFlow
n = 0
n < N_input
False
True
Calculate q_in = I(n)/S_in + Z_in
Invoke the interpreter and get output out
Calculate r_out = S_out(out – Z_out)
Append r_out to out_tflite
n++
Single tflite inference
core.py
n < N_input
FIFO full
Write Input FIFO with I(n)
Run SPLVP clock cycle
Data Ready
Read Output FIFO
Append q_out to out_splvp
Single SPLVP inference execution
Plot out_splvp and out_tflite
Save SVG
Save CSV
end
Generic operation
I/O operation
splvp operation
tflite operation
Single tflite inference
Flatbuffer
tflite Interpreter
Q
Q^-1
1/S_in
Z_in
Z_out
S_out
I
out
d
i
w
b
.mif
Input FIFO
Core
Output FIFO
q_out
Clock cycle
Single SPLVP inference execution

FIGURE 22. **High-level flow chart of the simulator which performs a comparative `tflite` versus SPLVP MLP inference. Inference is ran using TF in one case and a Python hardware model in the other.**

or it can be automatically retrieved by `build.py` from the YAML description. Using the standard Python interface of `tflite`, the quantized TF model is read and loaded, and it is ready to be used for inference. The simulation here is executed for all rows of the CSV file (overall N_{input}). Each row must be a vector of the same parallelism of the first input layer of the MLP. For all rows of the CSV files, data is quantized using the input scaling factor and zeros (S_{in} and Z_{in}) that are extracted by the `tflite` interface dictionary. After data is quantized using these constants, the interpreter is invoked, the inference is run and the interpreter output is converted back to real values using output scaling and zeros (S_{out} and Z_{out}), which are also available from the standard methods of `tflite`. A vector named `out_tflite` is populated with the obtained output data. This operation is repeated for all entries of the CSV files until all data is used.

Next, the simulator runs SPLVP inference on the same CSV data. During simulation, `sim.py` does not manually quantize the input data as in the previous case of `tflite` but it passes CSV rows as they are to the processor model `core.py`. The simulator feeds the processor `FIFO_IN[]` with new data until it is full and then it emulates a hardware clock cycle. If the input FIFO is full the simulation continues execution to flush it, until a new result is present in the `FIFO_OUT` memory. When a new result is available, the output FIFO is read and data is appended in a vector, here named `out_splvp`. The iteration is repeated until all the data is applied to the processor and all data from `FIFO_IN` are used. These processes of feeding `FIFO_IN[]` and reading data from `FIFO_OUT[]`, emulate the operation of an external MCU and the SPLVP I/O interfaces. The complementary read and write FIFO operations are executed instead by the program options `popin` and `pushout`.

Finally, the simulator generates plots of both results (i.e., `out_tflite` and `out_splvp`) and saves both inference outputs in a CSV file. The generated graphical data is saved in an SVG file. As `core.py` strictly matches the processor hardware implementation, the simulator is also capable of outputting the number of clock cycles required to run the inference, either for a single input or for all the CSV file rows. This active execution time is used to compare the inference speed of a given ANN model when executed on SPLVP versus an STM32L476 microcontroller in Sec. V-D.

ACKNOWLEDGMENT

The authors would like to thank Alessandro Barcellona, IIT-EDL, for the prototyping of the piggyback module for the Cyclone 10 LP evaluation board, Davide Dellepiane, IIT-EDL, for the wiring of a preliminary test set-up, and Claudio Lorini, IIT-EDL, for technical support in the state-of-the-art comparisons.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

REFERENCES

[1] (Jun. 2023). *An End-to-End Machine Learning Platform*. [Online]. Available: https://www.tensorflow.org
[2] (Jun. 2023). *PyTorch*. [Online]. Available: https://pytorch.org
[3] (Jun. 2023). *Caffe—Deep Learning Framework*. [Online]. Available: https://caffe.berkeleyvision.org
[4] (Jun. 2023). *Scikit-Learn—Machine Learning in Python*. [Online]. Available: https://scikit-learn.org/stable/
[5] (Jun. 2023). *MATLAB for Machine Learning*. [Online]. Available: https://mathworks.com/solutions/machine-learning.html
[6] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, Apr. 2021.
[7] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, Apr. 2022.
[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, 2012.
[9] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1740–1750.
[10] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi, "8-Bit numerical formats for deep neural networks," 2022, *arXiv:2206.02915*.

[11] A. Abdelfattah et al., "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 4, pp. 344–369, Jul. 2021.

[12] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," 2017, *arXiv:1710.03740*.

[13] (Jun. 2023). *TensorFlow 2.x Quantization Toolkit—1.0-Beta*. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/tensorflow-quantization-toolkit/docs/docs/intro_to_quantization.html

[14] F. Alongi, N. Ghielmetti, D. Pau, F. Terraneo, and W. Fornaciari, "Tiny neural networks for environmental predictions: An integrated approach with miosix," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Sep. 2020, pp. 350–355.

[15] A. Velichko, "Neural network for low-memory IoT devices and MNIST image recognition using kernels based on logistic map," *Electronics*, vol. 9, no. 9, p. 1432, Sep. 2020.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.

[17] Y. Ni, Y. Kim, T. Rosing, and M. Imani, "Online performance and power prediction for edge TPU via comprehensive characterization," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 612–615.

[18] A. Moss, H. Lee, L. Xun, C. Min, F. Kawsar, and A. Montanari, "Ultra-low power DNN accelerators for IoT: Resource characterization of the MAX78000," in *Proc. 20th ACM Conf. Embedded Networked Sensor Syst.* New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 934—-940.

[19] A. Boroumand, S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu, "Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating Google edge models," 2021, *arXiv:2103.00768*.

[20] C. Åleskog, H. Grahn, and A. Borg, "Recent developments in low-power AI accelerators: A survey," *Algorithms*, vol. 15, no. 11, p. 419, Nov. 2022.

[21] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*. New York, NY, USA: Association for Computing Machinery, Feb. 2016, pp. 26—-35.

[22] (Jun. 2023). *Free Tool for Edge AI Developers*. [Online]. Available: https://stm32ai.st.com/stm32-cube-ai/

[23] (Jul. 2023). *EON Compiler—tinyML Summit 2021 Proceedings*. [Online]. Available: https://cms.tinyml.org/wp-content/uploads/summit2021/tinyMLSummit2021d1_Awards_EdgeImpulse.pdf

[24] (Jul. 2023). *ARM Ethos U55—Embedded ML Inference for Cortex-M Systems*. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55

[25] (Jun. 2023). *Artificial Intelligence Microcontroller With Ultra-Low-Power Convolutional Neural Network Accelerator*. [Online]. Available: https://www.stg-maximintegrated.com/en/products/microcontrollers/MAX78000.html

[26] (Aug. 2023). *Kendryte—A Series of AI Chips Which Focuses on IoT, and the 1st-Gen is Named K210*. [Online]. Available: https://www.canaan.io/product/k230

[27] M. Giordano, L. Piccinelli, and M. Magno, "Survey and comparison of milliwatts micro controllers for tiny machine learning at the edge," in *Proc. IEEE 4th Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Jun. 2022, pp. 94–97.

[28] (Sep. 2023). *Efinix—TinyML Platform*. [Online]. Available: https://www.efinixinc.com/solutions-tinyml.html

[29] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2020, pp. 1–12.

[30] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4403–4417, May 2020.

[31] A. Krishna, S. R. Nudurupati, C. D G, P. Dwivedi, A. van Schaik, M. Mehendale, and C. S. Thakur, "Raman: A re-configurable and sparse tinyML accelerator for inference on edge," 2023, *arXiv:2306.06493*.

[32] R. Caruana, S. Lawrence, and C. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances in Neural Information Processing Systems*, vol. 13, T. Leen, T. Dietterich, and V. Tresp, Eds. Cambridge, MA, USA: MIT Press, 2000.

[33] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "MLP-mixer: An all-MLP architecture for vision," 2021, *arXiv:2105.01601*.

[34] Y. Taright and M. Hubin, "FPGA implementation of a multilayer perceptron neural network using VHDL," in *Proc. 4th Int. Conf. Signal Process.*, vol. 2, 1998, pp. 1311–1314.

[35] N. B. Gaikwad, V. Tiwari, A. Keskar, and N. C. Shivaprakash, "Efficient FPGA implementation of multilayer perceptron for real-time human activity classification," *IEEE Access*, vol. 7, pp. 26696–26706, 2019.

[36] M. Wess, P. D. Sai Manoj, and A. Jantsch, "Neural network based ECG anomaly detection on FPGA and trade-off analysis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.

[37] F. Gao, Z. Huang, S. Wang, and X. Ji, "A manycore processor based multilayer perceptron feedforward acceleration framework for embedded system," in *Proc. 3rd Int. Conf. Inf. Sci. Control Eng. (ICISCE)*, Jul. 2016, pp. 49–53.

[38] Z. Aklah and D. Andrews, "A flexible multilayer perceptron co-processor for FPGAs," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham, Switzerland: Springer, 2015, pp. 427–434.

[39] I. Westby, X. Yang, T. Liu, and H. Xu, "FPGA acceleration on a multi-layer perceptron neural network for digit recognition," *J. Supercomput.*, vol. 77, no. 12, pp. 14356–14373, Dec. 2021.

[40] D. Kim, J. Kung, and S. Mukhopadhyay, "A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 2, pp. 164–178, Apr. 2017.

[41] T. Senoo, A. Jinguji, R. Kuramochi, and H. Nakahara, "A multilayer perceptron training accelerator using systolic array," in *Proc. IEEE Asia Pacific Conf. Circuit Syst. (APCCAS)*, Nov. 2021, pp. 77–80.

[42] T. Yang, Y. Wei, Z. Tu, H. Zeng, M. A. Kinsy, N. Zheng, and P. Ren, "Design space exploration of neural network activation function circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 10, pp. 1974–1978, Oct. 2019.

[43] T. Szandała, *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks*. Singapore: Springer, 2021, pp. 203–224.

[44] (Oct. 2023). *Training Neural Networks With Tensor Cores*. [Online]. Available: https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

[45] H. Ootomo and A. Naruse, "Custom 8-bit floating point value format for reducing shared memory bank conflict in approximate nearest neighbor search," 2023, *arXiv:2301.06672*.

[46] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[47] E. Soufleri and K. Roy, "Network compression via mixed precision quantization using a multi-layer perceptron for the bit-width allocation," *IEEE Access*, vol. 9, pp. 135059–135068, 2021.

[48] (Jun. 2023). *TensorFlow Lite 8-Bit Quantization Specification*. [Online]. Available: https://www.tensorflow.org/lite/performance/quantization_spec

[49] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017, *arXiv:1712.05877*.

[50] (Jun. 2023). *University of California Irvine (UCI)—Machine Learning Repository*. [Online]. Available: https://archive.ics.uci.edu/datasets

[51] J. R. Stevens, R. Venkatesan, S. Dai, B. Khailany, and A. Raghunathan, "Softermax: Hardware/software co-design of an efficient softmax for transformers," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 469–474.

[52] K. Potdar, T. S. Pardawala, and C. D. Pai, "A comparative study of categorical variable encoding techniques for neural network classifiers," *Int. J. Comput. Appl.*, vol. 175, no. 4, pp. 7–9, Oct. 2017.

[53] (Oct. 2023). *TensorFlow models on the Edge TPU*. [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

[54] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Ann. Eugenics*, vol. 7, no. 2, pp. 179–188, Sep. 1936.

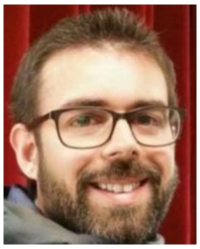
[55] T. Gupta, P. Arora, R. Rani, G. Jaiswal, P. Bansal, and A. Dev, "Classification of flower dataset using machine learning models," in *Proc. 4th Int. Conf. Artif. Intell. Speech Technol. (AIST)*, Dec. 2022, pp. 1–6.

[56] H. Nugroho, N. P. Utama, and K. Surendro, "Performance evaluation for class center-based missing data imputation algorithm," in *Proc. 9th Int. Conf. Softw. Comput. Appl.* New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 36–40.

[57] T. Almeida and J. Hidalgo, "SMS spam collection," UCI Mach. Learn. Repository, Univ. California, Irvine, Irvine, CA, USA, 2012. [Online]. Available: https://archive.ics.uci.edu/dataset/228/sms+spam+collection, doi: 10.24432/C5CC84.
[58] V. Lohweg, "Banknote authentication," UCI Mach. Learn. Repository, Univ. California, Irvine, Irvine, CA, USA, 2013, doi: 10.24432/C55P57.
[59] R. Bhatt, "Wireless indoor localization," UCI Mach. Learn. Repository, Univ. California, Irvine, Irvine, CA, USA, 2017, doi: 10.24432/C51880.
[60] V. Sigillito, S. Wing, L. Hutton, and K. Baker, "Ionosphere," UCI Mach. Learn. Repository, Univ. California, Irvine, Irvine, CA, USA, 1989, doi: 10.24432/C5W01B.
[61] C. Stefano, F. Fontanella, M. Maniaci, and A. Freca, "Avila," UCI Mach. Learn. Repository, Univ. California, Irvine, Irvine, CA, USA, 2018, doi: 10.24432/C5K02X.
[62] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami, "Contributions to the study of SMS spam filtering: New collection and results," in *Proc. 11th ACM Symp. Document Eng.* New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 259–262.
[63] Z. K. Malik, A. Hussain, and J. Wu, "An online generalized eigenvalue version of Laplacian eigenmaps for visual big data," *Neurocomputing*, vol. 173, pp. 127–136, Jan. 2016.
[64] J. G. Rohra, B. Perumal, S. J. Narayanan, P. Thakur, and R. B. Bhatt, "User localization in an indoor environment using fuzzy hybrid of particle swarm optimization & gravitational search algorithm with neural networks," in *Proc. 6th Int. Conf. Soft Comput. Problem Solving*. Singapore: Springer, 2017, pp. 286–295.
[65] T. Maszczyk and W. Duch, "Support feature machines: Support vectors are not enough," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2010, pp. 1–8.
[66] C. De Stefano, M. Maniaci, F. Fontanella, and A. Scotto di Freca, "Reliable writer identification in medieval manuscripts through page layout features: The 'Avila' bible case," *Eng. Appl. Artif. Intell.*, vol. 72, pp. 99–110, Jun. 2018.
[67] (Jun. 2023). *STM32 Nucleo-64 Development Board With STM32L476RG MCU*. [Online]. Available: https://estore.st.com/en/nucleo-l476rg-cpn.html
[68] M. Crepaldi, A. Merello, and M. Di Salvo, "A multi-one instruction set computer for microcontroller applications," *IEEE Access*, vol. 9, pp. 113454–113474, 2021.
[69] (Jun. 2023). *Dev Board—A Development Board to Quickly Prototype On-Device ML Products. Scale From Prototype to Production With a Removable System-On-Module (SoM)*. [Online]. Available: https://coral.ai/products/dev-board/
[70] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "How to evaluate deep neural network processors: TOPS/W (Alone) considered harmful," *IEEE Solid State Circuits Mag.*, vol. 12, no. 3, pp. 28–41, Summer 2020.
[71] E. Kabir, A. Poudel, Z. Aklah, M. Huang, and D. Andrews, "A runtime programmable accelerator for convolutional and multilayer perceptron neural networks on FPGA," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, L. Gan, Y. Wang, W. Xue, and T. Chau, Eds. Cham, Switzerland: Springer, 2022, pp. 32–46.
[72] M. Wang and A. P. Chandrakasan, "Flexible low power CNN accelerator for edge computing with weight tuning," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2019, pp. 209–212.
[73] B. Khabbazan and S. Mirzakuchaki, "Design and implementation of a low-power, embedded CNN accelerator on a low-end FPGA," in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2019, pp. 647–650.
[74] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the impact on processor microarchitecture," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Feb. 2011, pp. 5–14.
[75] S. Liao, A. Samiee, C. Deng, Y. Bai, and B. Yuan, "Compressing deep neural networks using Toeplitz matrix: Algorithm design and FPGA implementation," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1443–1447.

MARCO CREPALDI (Member, IEEE) received the engineering degree (summa cum laude) and the Ph.D. degree in electronic engineering from Politecnico di Torino (Polito), Turin, Italy, in 2005 and 2009, respectively. In 2008, he was a Visiting Scholar with the Department of Electrical Engineering, Columbia University, USA. After the Ph.D. degree, he was a Postdoctoral Researcher with the VLSI-Laboratory, Department of Electrical Engineering, PoliTo, and then as a Postdoctoral Researcher with the former Istituto Italiano di Tecnologia@PoliTo—Center for Space Human Robotics (IIT-CSHR). He is currently the coordinator of the Electronic Design Laboratory (edl.iit.it), IIT Center for Human Technologies, Genoa. He is the author and coauthor of more than 100 publications and two international patents. His research interests include the development of all-digital impulse-radio ultra-wideband (IR-UWB) systems, electronic systems design, neural network-based synchronization algorithms, in-memory computing with ferrofluids, and resource-constrained microprocessors, including one instruction set computers.

MIRCO DI SALVO received the degree in computer engineering from Universitá di Genova, Italy, in 2011. He worked for automotive, telecommunication, and aerospace industries before joining the Istituto Italiano di Tecnologia, in 2014. He first worked with Rehab Technologies and in 2017, he joined the Electronic Design Laboratory, Istituto Italiano di Tecnologia. His current research interests include the development of control software for different types of robots and firmware for motor control for robotic applications, radio transceivers, experimental medical devices, and smart sensors. His technical interests regard parallel computing architectures and 3D graphics.



ANDREA MERELLO received the degree (summa cum laude) in computer science from Universitá di Genova, Italy, in 2008. Since 2008, he has been a Software Engineer with the Electronic Design Laboratory, Istituto Italiano di Tecnologia. He contributed to the development of several open-source projects, including the Linux kernel with several patches. He designed electronics boards for experimental and scientific setups and several firmware and software for custom modules in the field of wireless systems, robotics, and miscellaneous systems. He has coauthored one conference, one international journal article, and one patent. His research interests include software development in the field of Linux drivers, firmware for bare-metal and ultra-low power electronics, motor control, CAN bus, and wireless communication.

Open Access funding provided by 'Istituto Italiano di Tecnologia' within the CRUI CARE Agreement