Documentation of a Toy Tensor Processing Unit Implementation

Tamás Epres

May 2025

1 Introduction

The purpose of this document is to serve as the report for the Independent Laboratory 1 course of the Electrical Engineering MSc program at the Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics. In terms of structure, this document includes the problem to be solved, the tools and methods used, the implemented architecture and its components, and concludes with a few test cases and measurement results. The SystemVerilog files of the project can be found in the following GitHub repository[2].

The thesis consists of four main parts. First, the problem is introduced, followed by a brief presentation of the Google Edge TPU as the hardware platform. This is followed by an architecture implemented on a Xilinx FPGA, which primarily serves as a guideline for a later, more comprehensive project.

2 Problem Description

In recent years, it has become evident that artificial intelligence offers not only accurate but also fast and robust solutions in the field of computer vision. Typically, AI algorithms consist of two phases: a training phase and an evaluation phase (inference). While the training phase is a time-consuming process usually performed on GPUs, the evaluation phase does not require complex hardware and can even be executed on resource-constrained embedded systems.

The aim of this project is to design a hardware accelerator that offloads the embedded system and speeds up the evaluation of a neural network. Our goal is to develop a hardware solution capable of independently performing neural network inference, thereby freeing up the microprocessor to handle other tasks while the accelerator carries out its assigned computations.

3 Google Edge Tensor Processing Unit

In this section, I present the fundamental operating principles of the Tensor Processing Unit (TPU) [6], along with a brief benchmark related to the Google Edge TPU.

The TPU architecture is essentially a specialized hardware accelerator for matrix multiplication. It is based on the concept of a systolic array, which operates in a relatively simple manner[3]. The key idea is to parallelize matrix multiplication: multiple Arithmetic Logic Units (ALUs) work simultaneously, arranged in a square grid structure. By feeding the appropriate data to the correct computing units at the right time, the correct result is obtained.

This approach differs from traditional GPU architectures in that it utilizes so-called high-bandwidth memory (HBM), which reduces the need for frequent memory access. In addition to this, unlike a GPU, the TPU is a dedicated hardware accelerator, meaning it cannot be used for general-purpose computations. Nevertheless, it is typically characterized by low power consumption.

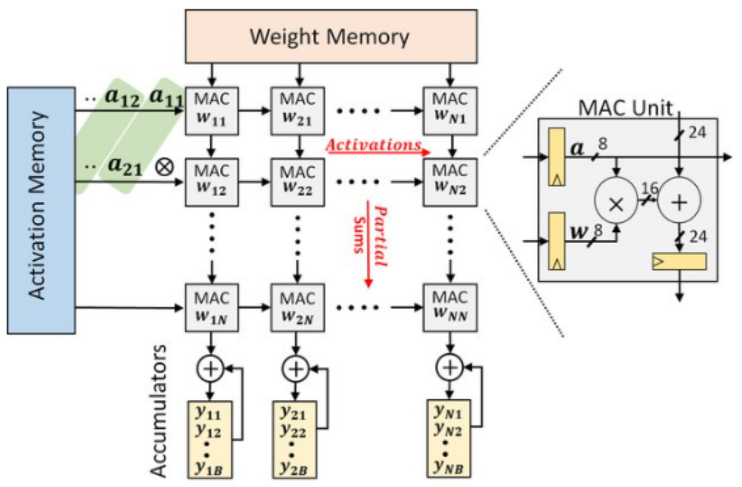
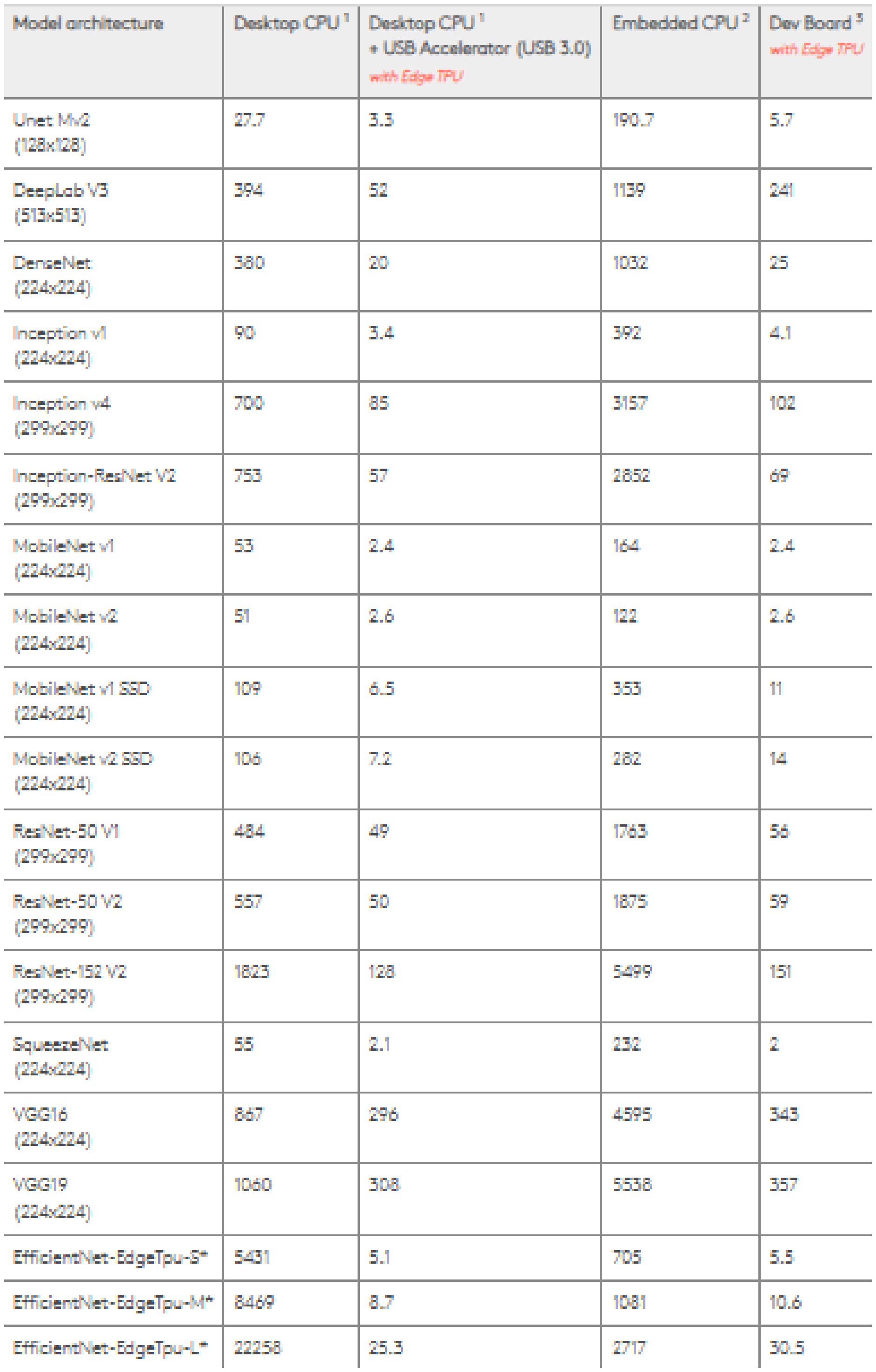


Figure 1: Operation of a systolic array in a TPU. Source: [7]

In the following, I present several benchmarks referenced in [1]. These benchmarks evaluate the inference latency (in milliseconds) of various neural networks at different image resolutions.



Model architecture	Desktop CPU [1]	Desktop CPU [1] + USB Accelerator (USB 3.0) *with Edge TPU*	Embedded CPU [2]	Dev Board [3] *with Edge TPU*
Unet Mv2 (128x128)	27.7	3.3	190.7	5.7
DeepLab V3 (513x513)	394	52	1139	241
DenseNet (224x224)	380	20	1032	25
Inception v1 (224x224)	90	3.4	392	4.1
Inception v4 (299x299)	700	85	3157	102
Inception-ResNet V2 (299x299)	753	57	2852	69
MobileNet v1 (224x224)	53	2.4	164	2.4
MobileNet v2 (224x224)	51	2.6	122	2.6
MobileNet v1 SSD (224x224)	109	6.5	353	11
MobileNet v2 SSD (224x224)	106	7.2	282	14
ResNet-50 V1 (299x299)	484	49	1763	56
ResNet-50 V2 (299x299)	557	50	1875	59
ResNet-152 V2 (299x299)	1823	128	5499	151
SqueezeNet (224x224)	55	2.1	232	2
VGG16 (224x224)	867	296	4595	343
VGG19 (224x224)	1060	308	5538	357
EfficientNet-EdgeTpu-S*	5431	5.1	705	5.5
EfficientNet-EdgeTpu-M*	8469	8.7	1081	10.6
EfficientNet-EdgeTpu-L*	22258	25.3	2717	30.5

Figure 2: Evaluation times (in milliseconds) of various neural networks on different hardware platforms. The hardware specifications are as follows: (1) Desktop CPU: Single 64-bit Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz; (2) Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz; (3) Dev Board: Quad-core Cortex-A53 @ 1.5GHz with Edge TPU.

It should be noted that the execution times are significantly influenced by how well TensorFlow Lite is optimized for the specific platforms, as well as by the implementation language of the test itself. Tests written in C++ may exhibit faster performance compared to those written in Python, due to the higher overhead associated with Python.

4 FPGA-based TPU

The idea of implementing such a TPU on an FPGA is not new, as Field-programmable gate arrays (FPGAs) are capable of performing highly parallel data processing with relatively low power requirements. The purpose of an FPGA is to realize a digital circuit or dedicated hardware that can exploit a high degree of parallelism on the input data. Therefore, it is considered an ideal platform for implementing such a system.

5 Specification

The device specification is as follows: The device consists of two main parts. One part, the `top_memory` module, is responsible for scheduling and storing data, while the other part, the `dot_product_multiplication_unit` module, handles data processing. The device expects 16-bit data on the `data_in` input. Depending on which memory we want to write to — whether it is the image data itself or the parameters characteristic of the neural network — the data is written to different memories. Using the input address signals, we can select which image memory to process and which parameters to multiply with the given image segment. This results in a device capable of multiplying arbitrary image memory contents with parameter memory contents.

The device has several parameters that allow the implemented hardware to be scaled. These parameters are as follows:

- `parameter DATA_WIDTH`: the data width (interpreted as 16-bit floating point numbers),
- `parameter IMAGE_WIDTH`: the width of the input image,
- `parameter IMAGE_HEIGHT`: the height of the input image,
- `parameter NUM_UNITS`: the number of processing units.

A következőkben

6 Detailed Operation

As described in the specification, the device consists of two main parts. Here, we present a detailed description of these components.

6.1 top_memory Module

The `top_memory` module is responsible for data storage and scheduling. The inputs of the device are defined as follows:

- `input logic clk,`
  `input logic reset,`
  `input logic step,`
  `input logic en,`
  These signals are responsible for memory reset, stepping, and enabling.

- `input logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] data_in,`
  The input data.

- `input logic read_mem1,`
  `input logic write_mem1,`
  `input logic [NUM_UNITS-1:0][clog2(IMAGE_WIDTH*IMAGE_HEIGHT)-1:0] start_addr_1,`
  `input logic read_mem2,`
  `input logic write_mem2,`
  `input logic [NUM_UNITS-1:0][clog2(IMAGE_WIDTH*IMAGE_HEIGHT)-1:0] start_addr_2,`
  `input logic [clog2(IMAGE_WIDTH)-1:0] kernel_dim,`
  `input logic simple_write,`
  `input logic simple_read,`
  `input logic [NUM_UNITS-1:0][clog2(IMAGE_WIDTH*IMAGE_HEIGHT)-1:0] simple_addr,`
  These signals select which memory to write to. `mem1` mainly stores image data, `mem2` stores neural network weights, and the simple memory is responsible for bias values.

The outputs towards the `dot_product_multiplication_unit` are:

- output logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] out_1,
- output logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] out_2,
- output logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] simple_mem_out,
- output logic en_out_1,
- output logic en_out_2,

The memories in the top_memory module can be loaded via the data_in inputs by setting the corresponding write signals high and specifying the appropriate memory addresses using the start_addr signals. Scheduling is performed by setting the read signal of the relevant memory segment high, selecting the memory regions from which data should be supplied to the dot_product_multiplication_unit, and specifying the length of data to be read. The top_memory module then outputs the required data to the dot_product_multiplication_unit. Once the calculations are completed, the dot_product_multiplication_unit signals this with an array_done signal, prompting the top_memory module to increment the given input addresses by one. This data fetching process continues until the desired amount of data has been processed.

The top_memory_unit consists of three main memory sections: mem1, which contains the image data; mem2, which stores the parameters or possibly system results; and simple_memory, which holds the individual bias values. Both mem1 and mem2 can simultaneously read and write NUM_UNITS data items, as these memories continuously supply data to our processing units. The memx memories are implemented within the memory_unit module. The en_1 and en_2 signals are reserved for future expansion. During later implementations, it might be important to consider that memory reads may not provide data immediately. Let us examine the operation of the module through an example, without using specific data.

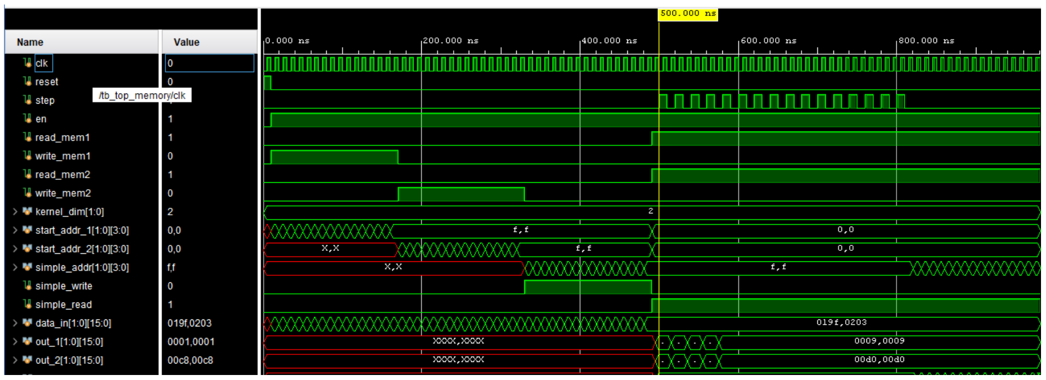


Figure 3: Read/write sequence of the `top_memory` module

As shown in the figure, asserting the individual write signals enables writing to the corresponding memory sections. The target memory address is specified via the `start_addr` signals. Once the memories have been filled with data, they can be read starting from the given `start_addr`. When the `step` signal is asserted, memory addresses are incremented while taking the image dimensions into account. Note that only as many values are read as needed by the kernel dimension. For example, with a 2×2 kernel, four values are read from both `mem1` and `mem2`.

6.1.1 memory_unit module

This module is responsible for correctly addressing both the parameters and the image data, as well as supplying the processing units with the appropriate input data. In practice, the module handles the parallel addressing of `NUM_UNITS` data items and computes the correct memory addresses based on the image information. One memory block is responsible for storing the image data, while another is used to store intermediate results and the parameters of the neural network.

6.2 dot_product_multiplication_unit

This module is responsible for processing the data. Its inputs and outputs are as follows:

- `input logic clk,`

- `input logic reset,`
- `input logic start,`
- `input logic [NUM_UNITS-1:0] active_units,`
  Control signals.
- `input logic [(clog2(IMAGE_WIDTH)-1) * (clog2(IMAGE_WIDTH)-1):0] length,`
  The length of the data sequence to be processed.
- `input logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] a_in_array,`
- `input logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] b_in_array,`
- `input logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] bias_array,`
  Input data arrays.
- `output logic [NUM_UNITS-1:0][DATA_WIDTH-1:0] relu_out,`
  Output data after ReLU activation.
- `output logic done,`
  Signal indicating the completion of the full data sequence.
- `output logic array_done`
  Signal indicating the completion of one multiply-and-accumulate operation, used by the `top_memory` module to increment memory addresses.

The operation of the module is relatively straightforward. It contains `NUM_UNITS` parallel processing units, each responsible for performing a multiplication and an addition. These units are implemented in the `processing_unit` module.

The appropriate input data are sequentially fed into the module from memory. Typically, image data are supplied to one input, while the corresponding neural network weights are fed to the other. Once the inner products of the relevant memory segments are computed using the processing units, the results are passed to the `vector_adder` module, which adds the bias values to the computed results. It is noteworthy that these arithmetic modules operate based on internal state machines, which change states when the individual subunits (multiplier and adder) complete their operations and signal their readiness.

Finally, the output values are processed by the `ReLU` module, which applies a non-linearity to the results.

6.2.1 processing_unit

The module multiplies the 16-bit floating-point numbers received on its inputs and then adds the result to a previously accumulated value, thus performing the *dot product* operation. The `processing_unit` includes a 16-bit floating-point multiplier and adder.

As these components involve complex combinational logic, the calculations are carried out over multiple clock cycles. The following test demonstrates how such a multiplication and subsequent addition are performed.

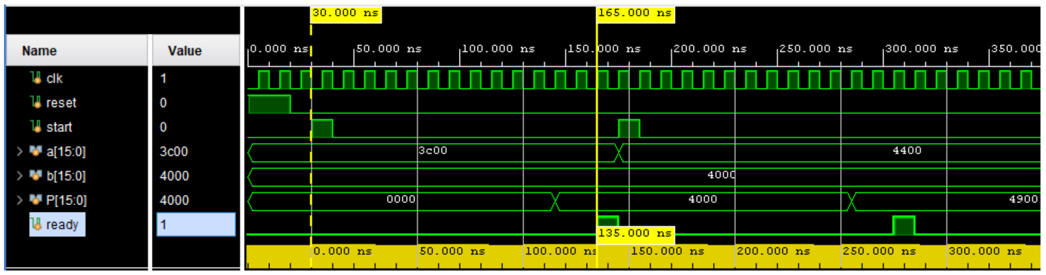
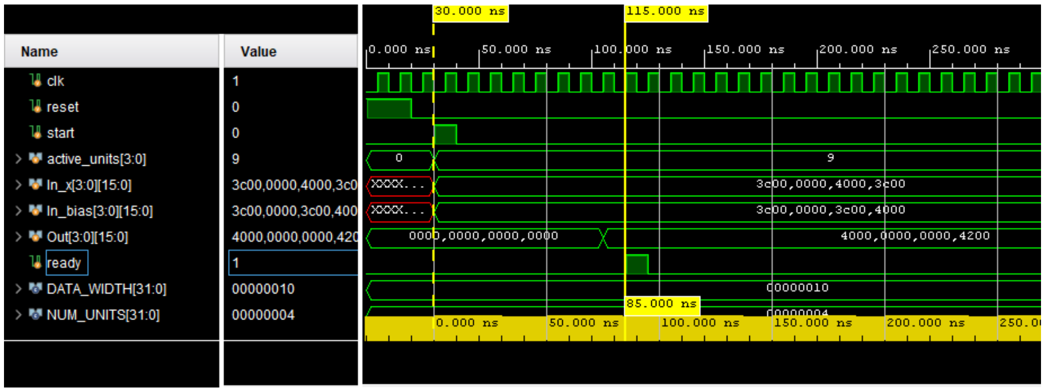


Figure 4: The "a" input receives first the value 1 and then 4, while the "b" input remains constant at 2. A new computation starts only after the previous one has completed.

In this example, the input `a` is first set to 1, then to 4, while the input `b` remains fixed at 2. As a result, we get $1 \cdot 2 + 4 \cdot 2 = 10$, which appears at the output after the second `ready` pulse. As shown in the figure, once the `start` signal pulse is applied to the module, the actual output appears only after a delay of approximately 135 ns. This delay is due to the pipelined nature of floating-point arithmetic: the operations are not completed within a single clock cycle but are instead performed over multiple stages.In this case, the clock frequency is 100 MHz, and the entire calculation completes in a total of 13.5 clock cycles.

6.2.2 vector_adder

The vector adder module is responsible for adding two input vectors element-wise. Typically, it sums the output of the `processing_unit` with the bias values. Consider the following test case:



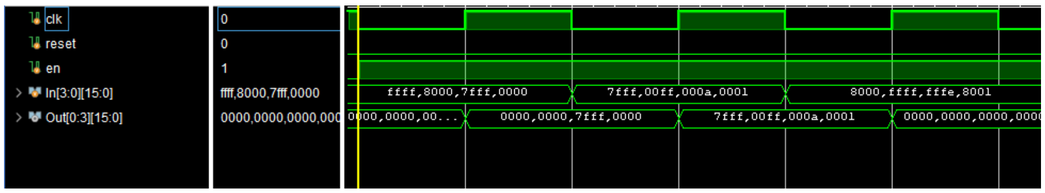
```
Name                Value
clk                 1
reset               0
start               0
active_units[3:0]   9
In_x[3:0][15:0]     3c00,0000,4000,3c0
In_bias[3:0][15:0]  3c00,0000,3c00,400
Out[3:0][15:0]      4000,0000,0000,420
ready               1
DATA_WIDTH[31:0]    00000010
NUM_UNITS[31:0]     00000004
```

Figure 5: The input vector `In_x` is set to (1, 0, 2, 1), while the input bias vector `In_bias` is (1, 0, 0, 2). Only the first and last additions are enabled using the `active_unit` mask.

As shown, the input `In_x` receives (1, 0, 2, 1), and `In_bias` receives (1, 0, 0, 1). Due to input masking, only the first and last coordinates are added; the others are disabled and output zero. As seen in the figure, after the `start` input pulse, it takes approximately 85 ns to perform such an addition, assuming a clock frequency of 100 MHz (which corresponds to about 8.5 clock cycles). The output at this time is (2, 0, 0, 3).

6.2.3 ReLU

The ReLU module is a simple multiplexer network which outputs zero if the input number is less than zero; otherwise, the input number appears at the output. Consider the following example:



```
clk             0
reset           0
en              1
In[3:0][15:0]   ffff,8000,7fff,0000
Out[0:3][15:0]  0000,0000,0000,000
```

Figure 6: Operation of the ReLU module: if the most significant bit is 1, the output is zero.

As shown, the operation of the ReLU is straightforward: for a 16-bit number representation, the most significant bit is the sign bit; therefore, when the sign bit is 1, the output is zero, otherwise the output is the input number itself.

7 Demos

After becoming familiar with the operation of the device, several demonstration examples are presented. These examples showcase the computation of simple convolutional filters applied to a few input images. In these demonstrations, the device parameters are set as follows:

```
    parameter DATA_WIDTH = 16;
    parameter IMAGE_WIDTH = 5;
    parameter IMAGE_HEIGHT = 5;
    parameter NUM_UNITS = 9;
```

This means that 9 convolution operations will be performed in parallel on a 5×5 image. The demonstrations aim to recognize hand-written digits composed of 0s and 1s by applying the Prewitt operator in the horizontal (x) direction [8]. The purpose of these examples is to illustrate the generation of activation maps and to showcase the functionality of the hardware.

It is worth noting that a similar approach could be applied to implement a multi-layer perceptron (MLP) on the input data. The key idea behind the architecture is that both convolutional filtering and MLP computations rely fundamentally on the inner product operation. In this case, the weights corresponding to a single neuron can simply be stored in `mem2`, and multiplied with the input or activation image accordingly.

7.1 First Demo

In this demonstration, we aim to generate the activation map of an image representing the digit "1" using the Prewitt operator. The input image stored in `mem1` looks as follows:

```
//  0  0  1  1  0
//  0  1  0  1  0
//  0  0  0  1  0
//  0  0  1  0  0
//  0  0  0  1  0
```

The kernel weights, stored in `mem2`, are:

```
//  1  0 -1  0  0
//  1  0 -1  0  0
//  1  0 -1  0  0
//  0  0  0  0  0
//  0  0  0  0  0
```

In this example, no bias is added, so the contents of `simple_memory` are uniformly zero. When addressing `mem2`, all `start_address_2` signals begin at address 0. For `mem1`, the `start_address_1` values are as follows: 0, 1, 2, 5, 6, 7, 10, 11, 12. These positions correspond to the image segments that we wish to convolve with the kernel.

Let us examine the output in this case. The first output value is computed as:

$$1 \cdot 0 + 0 \cdot 0 + (-1) \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot (-1) + 1 \cdot 0 + 0 \cdot 0 + 0 \cdot (-1) = -1$$

After applying the ReLU activation function, this value becomes 0. Performing the convolution across all 9 positions yields the following output values:

$$(0, 0, 1, 0, 0, 1, 0, 0, 1)$$

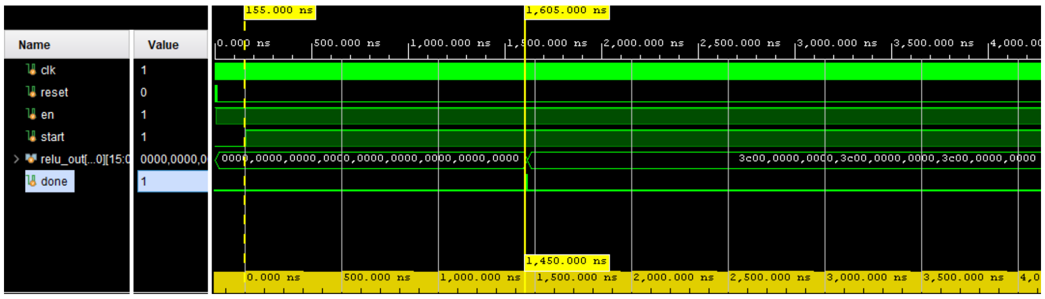


Figure 7: Output evolution during the first demo

As shown in the figure, the output values are $(0, 0, 1, 0, 0, 1, 0, 0, 1)$ — note that the values are displayed in reverse order. It can also be observed that the complete computation, i.e., the execution of all convolution operations, took 1450 ns with a 100 MHz clock, which corresponds to 145 clock cycles.

7.2 Second Demo

In the second demo, we aim to generate the activation map of an input image representing the digit "0". The content of `mem1` is as follows:

```
//  0  1  0  0  0
//  1  0  1  0  0
//  1  0  0  1  0
//  1  0  0  1  0
//  0  1  1  0  0
```

The content of `mem2` is identical to the one used in the first demo. When computing the dot products for each receptive field, the resulting activation values are:

$$(1, 0, 1, 2, 0, 1, 1, 0, 1)$$

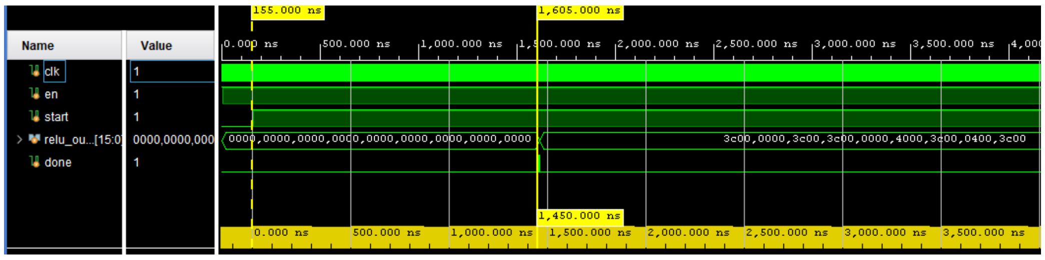
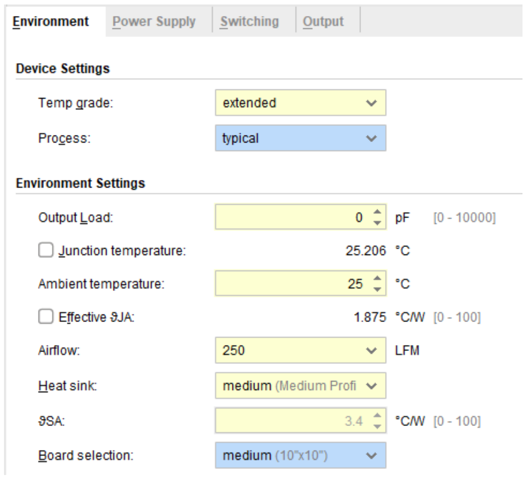


Figure 8: Output evolution during the second demo

As shown in the figure, the output values are indeed $(1, 0, 1, 2, 0, 1, 1, 0, 1)$. However, it is worth noting that the second coordinate is not exactly zero but slightly off (specifically, `0.00006104`). This small numerical discrepancy was not further investigated in this demo.

8 Power Consumption by Parameters

In this section, we analyze the power consumption of the system as a function of its key parameters. Power usage depends primarily on the number of active processing units (`NUM_UNITS`), the clock frequency, and the characteristics of the input images. It is important to note that for accurate performance estimation, the target FPGA model must be specified. In this case, the selected device is the `xc7k160tfbg676-3` FPGA [5].The environmental and physical conditions were also set to fixed values, as summarized in the following table:



Device Settings

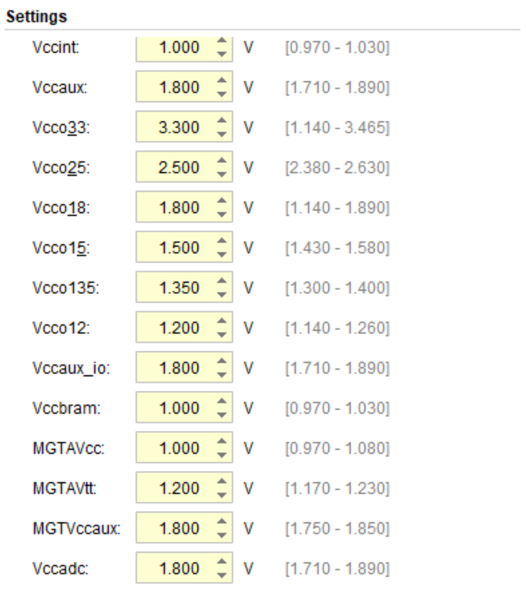
Temp grade:	extended		
Process:	typical		

Environment Settings

Output Load:	0	pF	[0 - 10000]
☐ Junction temperature:	25.206	°C	
Ambient temperature:	25	°C	
☐ Effective ϑJA:	1.875	°C/W	[0 - 100]
Airflow:	250	LFM	
Heat sink:	medium (Medium Profi		
ϑSA:	3.4	°C/W	[0 - 100]
Board selection:	medium (10"x10")		

Figure 9: Environmental and physical parameter settings

All supply and signal voltage levels were also treated as constant factors, just like the clock frequency, which was set to the default value of 100MHz.



Settings

Vccint:	1.000	V	[0.970 - 1.030]
Vccaux:	1.800	V	[1.710 - 1.890]
Vcco33:	3.300	V	[1.140 - 3.465]
Vcco25:	2.500	V	[2.380 - 2.630]
Vcco18:	1.800	V	[1.140 - 1.890]
Vcco15:	1.500	V	[1.430 - 1.580]
Vcco135:	1.350	V	[1.300 - 1.400]
Vcco12:	1.200	V	[1.140 - 1.260]
Vccaux_io:	1.800	V	[1.710 - 1.890]
Vccbram:	1.000	V	[0.970 - 1.030]
MGTAVcc:	1.000	V	[0.970 - 1.080]
MGTAVtt:	1.200	V	[1.170 - 1.230]
MGTVccaux:	1.800	V	[1.750 - 1.850]
Vccadc:	1.800	V	[1.710 - 1.890]

Figure 10: Voltage levels used in the system.

8.1 Effect of the `NUM_UNITS` Parameter

In this section, we analyze how changes in the `NUM_UNITS` parameter affect the total on-chip power consumption. Throughout this analysis, both `IMAGE_WIDTH` and `IMAGE_HEIGHT` are fixed to 5, and the data width is consistently set to 16 bits.

- `NUM_UNITS` = 1, Total On-Chip Power = 0.132 W
- `NUM_UNITS` = 2, Total On-Chip Power = 0.147 W
- `NUM_UNITS` = 3, Total On-Chip Power = 0.156 W
- `NUM_UNITS` = 4, Total On-Chip Power = 0.165 W
- `NUM_UNITS` = 5, Total On-Chip Power = 0.175 W
- `NUM_UNITS` = 6, Total On-Chip Power = 0.186 W
- `NUM_UNITS` = 7, Total On-Chip Power = 0.198 W

- `NUM_UNITS` = 8, Total On-Chip Power = 0.206 W

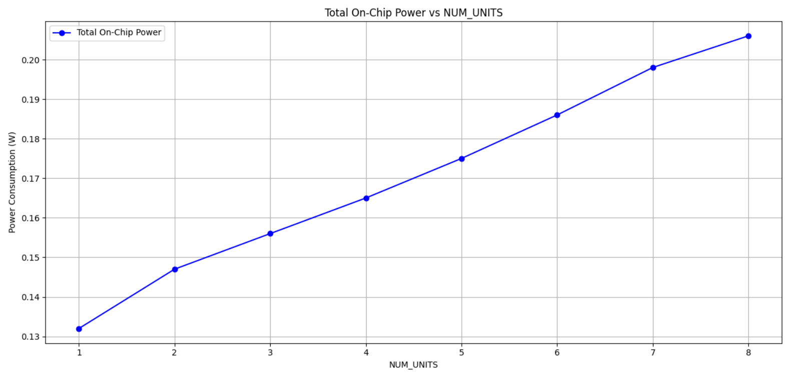


Figure 11: `NUM_UNITS` vs power consumption. As shown in the figure, the consumed energy is approximately linearly proportional to `NUM_UNITS`.

This parameter typically leads to an increase in the logic's power consumption. A noticeable trend can be observed: the more `NUM_UNITS` are used, the larger the dynamic components contribute to the total power consumption. In Vivado terminology, **dynamic power consumption** refers to the energy consumed by switching activities and logic circuits during operation, whereas **static power consumption** encompasses all other sources of power usage. To better understand the system's behavior, the chip itself is primarily responsible for dynamic power consumption, while the board accounts for the static power consumption [4].

8.2 Effect of the `IMAGE_WIDTH` and `IMAGE_HEIGHT` Parameters

In this section, we analyze how simultaneous changes in the `IMAGE_WIDTH` and `IMAGE_HEIGHT` parameters affect the total on-chip power consumption. For all measurements, `NUM_UNITS` is fixed to 1, and the data width remains at 16 bits. Note that in this case, the image width and height are always equal, i.e., `IMAGE_WIDTH` = `IMAGE_HEIGHT`.

- `IMAGE_WIDTH` = 5, Total On-Chip Power = 0.132 W

- IMAGE_WIDTH = 10, Total On-Chip Power = 0.144 W
- IMAGE_WIDTH = 15, Total On-Chip Power = 0.162 W
- IMAGE_WIDTH = 20, Total On-Chip Power = 0.188 W
- IMAGE_WIDTH = 25, Total On-Chip Power = 0.218 W
- IMAGE_WIDTH = 30, Total On-Chip Power = 0.248 W

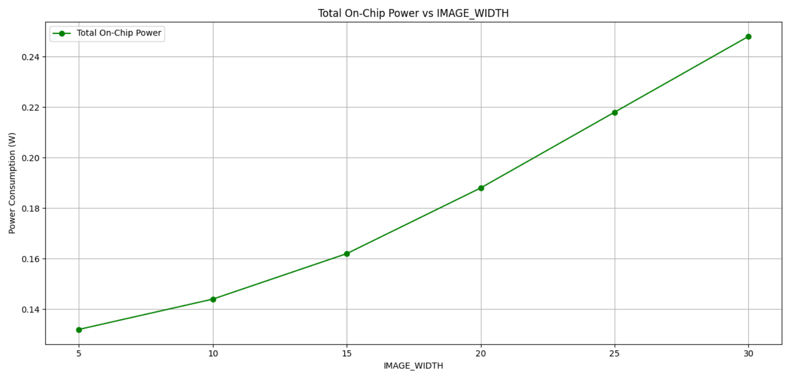


Figure 12: IMAGE_WIDTH and IMAGE_HEIGHT vs. power consumption. As shown in the figure, the consumed energy is approximately quadratically proportional to IMAGE_WIDTH and IMAGE_HEIGHT.

This parameter typically leads to an increase primarily in the signal switching power consumption.

8.3 Example Resource Allocation

The specific resource allocation of the device, depending on the following parameters, is as follows:

- DATA_WIDTH = 16
- IMAGE_WIDTH = 15
- IMAGE_HEIGHT = 15
- NUM_UNITS = 5

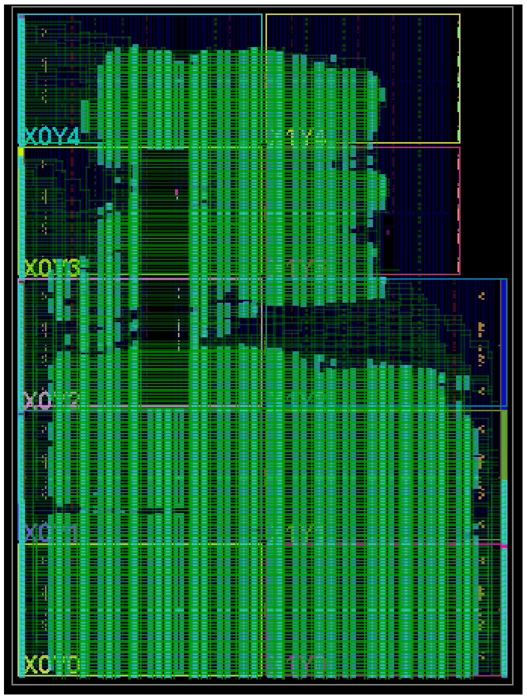
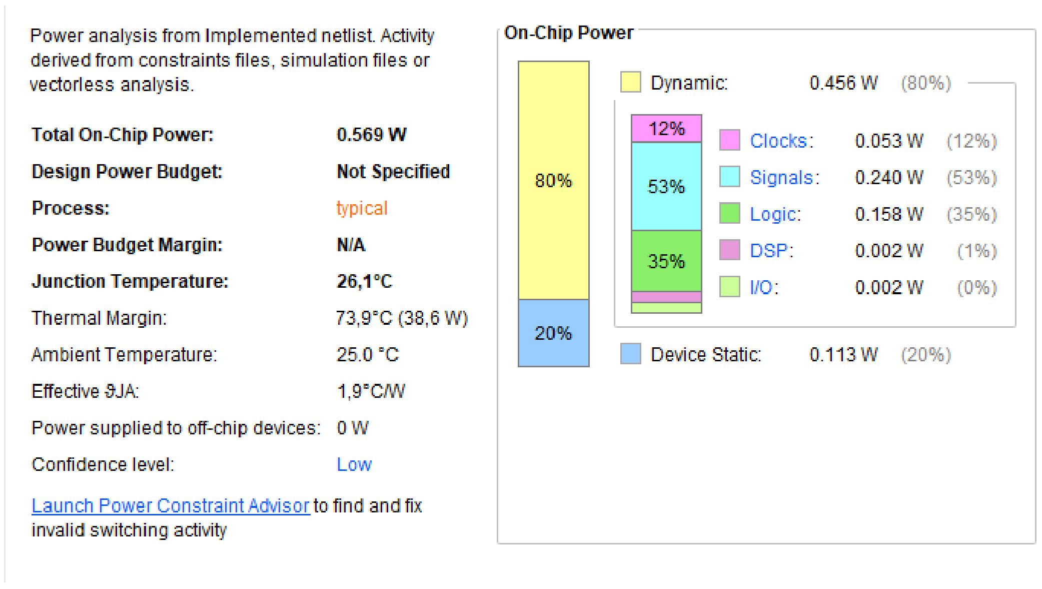


Figure 13: Resource allocation of the device

As can be seen, this parameter distribution covers nearly the entire resource demand of the device, making the optimization of the architecture one of the most important directions for further development. In this case, the power consumption of the device would take the following form:


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.569 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26,1°C**
Thermal Margin:	73,9°C (38,6 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1,9°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.456 W	(80%)	
Clocks:	0.053 W	(12%)	
Signals:	0.240 W	(53%)	
Logic:	0.158 W	(35%)	
DSP:	0.002 W	(1%)	
I/O:	0.002 W	(0%)	
Device Static:	0.113 W	(20%)	

Figure 14: Power consumption of the device with the given parameters

9 Possible Further Developments

The main limitation of the current architecture is its poor scalability in terms of input and output I/O pin allocation. A potential solution to this problem would be the use of a fast and reliable serial communication interface through which the image could be transferred. As the system scales, it becomes evident that with increasing image data, the performance of signal transmission will dominate power consumption. Therefore, a development is needed that makes the processing more sequential in nature, which currently presents a challenge.

Further considerations could include the use of fixed-point arithmetic. While this would reduce the flexibility of the device, it would simplify the hardware, allowing more computations to be performed in the same amount of time, ultimately contributing to the acceleration of the system.

Another promising direction for development would be the implementation of the device on actual hardware.

References

[1] Edge tpu performance benchmarks. https://coral.ai/docs/edgetpu/benchmarks/.

[2] Github repository. https://github.com/Eper00/Toy-Tensor-Processing-Unit-TTPU.

[3] Systolic array overwiev. https://en.wikipedia.org/wiki/Systolic_array.

[4] Xilinx power estimation. https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug440-xilinx-power-estimator.pdf.

[5] AMD. Xc7k160t data sheet. https://docs.amd.com/v/u/en-US/ds180_7Series_Overview.

[6] Google. Tpu architecture. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm.

[7] Zahra Ghodsi Siddharth GargSiddharth Garg Jeff J. Zhang, Kartheek Rangineni. Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators. https://www.researchgate.net/publication/325861610_Thundervolt_enabling_aggressive_voltage_underscaling_and_timing_error_resilience_for_energy_efficient_deep_learning_accelerators/figures?lo=1.

[8] Wikipedia. Prewit operator. https://en.wikipedia.org/wiki/Prewitt_operator.