Call: **H2020-FETPROACT-2020-01**

Grant Agreement no. **101016776**

Deliverable D04.05a

Virtual scalable N^2C^2 design and Pareto-front data

Start date of the project: 1st January 2021

Duration: 60 months
Project Coordinator: Cristell MANEUX - University of Bordeaux
Contact: Cristell MANEUX - cristell.maneux@ims-bordeaux.fr

DOCUMENT CLASSIFICATION

Title	Virtual scalable N^2C^2 design and Pareto-front data
Deliverable	D4.05a
Estimated Delivery	31/08/2021 (M6+2)
Date of Delivery Foreseen	31/08/2021 (M6+2)
Actual Date of Delivery	31/08/2021 (M6+2)
Authors	Giovanni Ansaloni – P5 - EPFL
	David Atienza – P5 – EPFL
	Alberto Bosio – P3 – ECL-INL
	Ian O'Connor – P3 – ECL-INL
Approver	Cristell Maneux – P1 – UBx
Work package	WP4
Dissemination	PU (Public)
Version	V1.0
Doc ID Code	D4.05a_FVLLMONTI_P3-ECL-INL-20210831
Keywords	Neural network, logic design, specifications

DOCUMENT HISTORY

Version status	V1.0
Date	26/08/2021 (M6+2)
Document revision	NA
Date	NA
Reason for change	NA

DOCUMENT ABSTRACT

This document describes the first version of the virtual scalable Neural Network Compute Cube (N^2C^2). Its principal function is to carry out element-wise non-volatile matrix multiplication, accumulation and activation through a non-linear function. It features multiple means of configuration:

- number of inputs to each cell: configure the vertical routing of data between layers in both directions
- synaptic coefficients: program coefficients in memory elements and connect them to the multipliers
- various activation functions can be efficiently programmed in memory elements in a coarse-grain logic-in-memory approach

As technology development and logic cell design is still in early stages, this version of D4.05 is intended to serve as a reference document, containing a detailed description of functionality, high-level architecture and performance metrics. This information will be used mainly in WP4 (to focus logic cell design work towards a scaled down version of N2C2 in D4.4 scheduled for M30+2 as well as a second version of the virtual scalable N2C2 in D4.5b scheduled for M36+2) and WP5 (to enable architectural exploration in D5.2 scheduled for M20+2 and M36+2). As the FVLLMONTI project progresses, the content of this deliverable will be updated to reflect opportunities and limitations that appear according to the state of technology and logic circuit development.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101016776.

TABLE OF CONTENT

LIST OF FIGURES AND TABLES

LIST OF ACRONYMS / GLOSSARY

D: Deliverable
LUT: Look Up Table
M: Month of the project
MAC: Multiply Accumulate
N^2C^2: Neural Network Compute Cube
NN: Neural Network
P: Partner
PU: Public
V: Version
VNWFET: Vertical Nanowire Field Effect Transistor
WP: Work Package

1. Target Functionality

The Neural Network Compute Cube (N^2C^2) is a central concept to the FVLLMONTI project. It represents a flexible computing hardware block for transformer-based neural networks. As illustrated in Figure 1, it will be implemented based on a dedicated library of 3D logic cells leveraging VNWFET devices developed in T4.1 (Logic cell design, optimization and validation) and using technological hardware and data developed in WP1, WP2 and WP3. It also connects through a reconfigurable 3D interconnect framework developed in T4.2 (Inter-cube interconnect framework) to implement a scalable and versatile 3D architectural model in connection with WP5. Its fundamental properties of physical regularity, functional versatility and in-memory vector processing will make it suitable to explore hardware/software co-design techniques in the context of transformer-based neural networks for machine translation applications as well as quantization-based approximate computing to reduce resource usage and energy consumption as well as enable more complex network topologies.

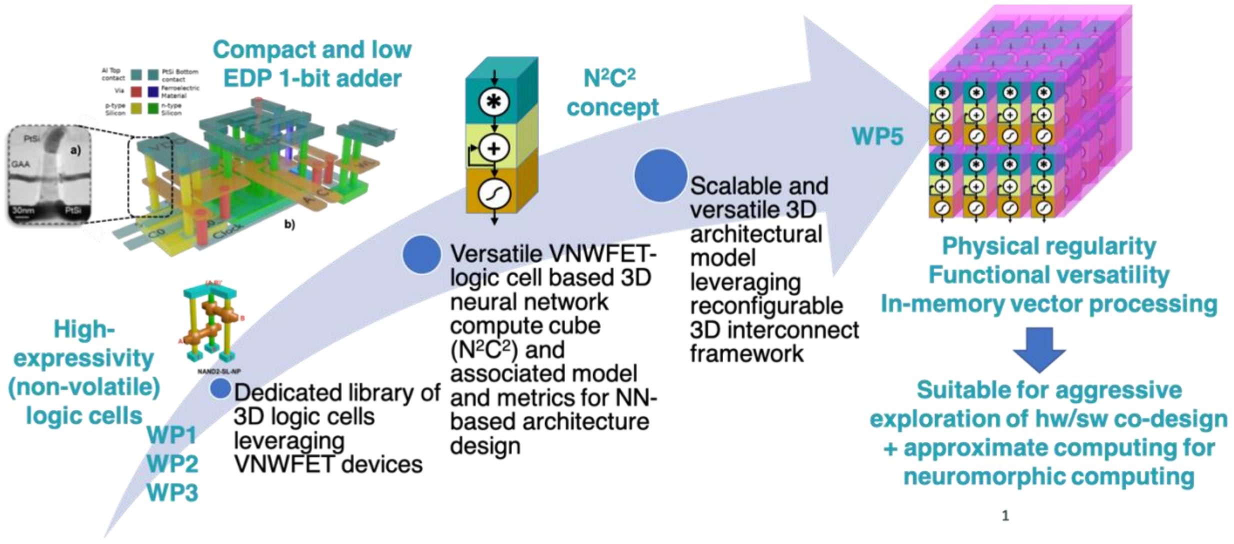
Compact and low EDP 1-bit adder
a)
b)
N²C² concept
WP5
Scalable and versatile 3D architectural model leveraging reconfigurable 3D interconnect framework
Versatile VNWFET-logic cell based 3D neural network compute cube (N²C²) and associated model and metrics for NN-based architecture design
Physical regularity
Functional versatility
In-memory vector processing
Suitable for aggressive exploration of hw/sw co-design + approximate computing for neuromorphic computing
High-expressivity (non-volatile) logic cells
Dedicated library of 3D logic cells leveraging VNWFET devices
WP1
WP2
WP3
1

Figure 1 : Neural Network Compute Cube (N^2C^2) – the big picture

The principal function of the N^2C^2 is to carry out element-wise non-volatile matrix multiplication, accumulation and activation through a non-linear function. It features multiple means of configuration:

- Firstly, it is function-configurable. As a baseline operation, we define a 32-bit integer multiply-accumulate function (MAC) which can also be broken down into its individual operations (multiply, addition, accumulation and combinations of these). We also include resources to efficiently program an activation function (e.g. sigmoid, tanh, rectified linear – ReLU, softplus …) that can be switched in and out of the datapath. It is intended for the activation function to be implemented in memory elements in a coarse-grain logic-in-memory approach.
- It is connectivity-configurable, meaning that it is possible to input from 2-8 operands as number of inputs to each cell. Further, it is compatible with routing resources outside of the N^2C^2 (T4.2 – Intercube interconnect framework) in order to (for example) handle feedback in recursive networks, or to configure the vertical routing of data between layers in both directions.
- It is coefficient-configurable, meaning that it is possible to program synaptic coefficients in memory elements and connect them to the multiplier function blocks.
- It is datawidth-configurable, in that it is possible to implement both intra-N^2C^2 scaledown from 1*32 bits to 2*16 bits, 4*8 bits or 8*4 bits; and that it is also possible to handle inter-N^2C^2 scaleup to 64 bits, 128 bits, 256 bits, 512 bits.

The following sections will detail the architectural specifications and description of planned implementations of the N^2C^2 block. In section 2, we will describe the proposed architecture in terms of its schematic and target behavior, as well as its intended use in matrix multiplication architectures. Section 3 will cover the intended implementations of the N2C2, including technological variants, target logic design styles and finally performance measurements which will be extracted from circuit-level schematics and injected into higher-level architectural models to enable performance assessment of complete architectures.

2. Proposed Architecture

I. SCHEMATIC AND BEHAVIOR

This section describes the schematic view of the N^2C^2 block as depicted in Figure 2.

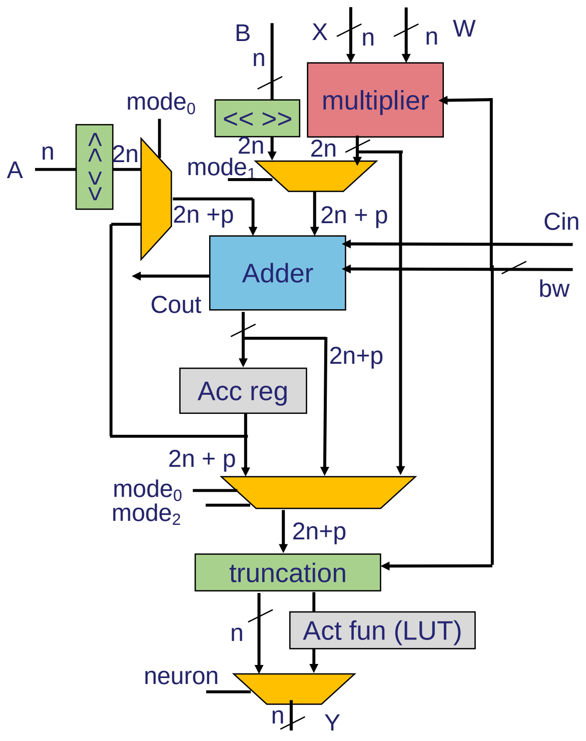

Figure 2 : N^2C^2 schematic view

Table 1 summarizes the input/output signals, with the related direction, bit-width and type (data or control signal).

Table 1: N^2C^2 I/O signals

Signal	Direction	Width (bit)	Type
X	input	N	data
W			
A			
B			
Y	output		
Cin	input	1	
Cout	output	1	
bw	input	2	control
$mode_0$		1	
$mode_1$		1	
$mode_2$		1	
neuron		1	

The N^2C^2 is a sophisticated Programmable Multiply and Accumulate unit. Table 2 reports the available arithmetic functions and the associated control signals values.

Table 2: N^2C^2 arithmetic functions

Output	Control Signals	Description

Y = X * Y
Acc = Y + A
	$mode_1$= 1
	$mode_0$= 1
	$mode_2$= 0
	neuron = 0
		multiplication

Y = X * Y + A
Acc = Y
	$mode_1$= 1
	$mode_0$= 0
	$mode_2$= 0
	neuron = 0
		multiplication addition

Y = X * Y + Acc
Acc = Acc + Y
	$mode_1$= 1
	$mode_0$= 1
	$mode_2$= 1
	neuron = 0
		multiplication accumulation (MAC)

Y = A + B
Acc = Y
	$mode_1$= 0
	$mode_0$= 0
	$mode_2$= 0
	neuron = 0
		addition

Y = B + Acc
Acc = Acc + Y
	$mode_1$= 0
	$mode_0$= 1
	$mode_2$= 1
	neuron = 0
		accumulation

Y = Act(R)	neuron = 1	neuron mode with activation function[1]. R can be the result of any of the above arithmetic functions

The N^2C^2 supports different levels of accuracy, in terms of bit-width of data processing. This is particularly useful in energy/resource-critical applications where it can be useful to explore accuracy/resource usage tradeoffs. The baseline accuracy is chosen to be n=32 (i.e. the bit-width of all data signals is 32 bits). By using the 'bw' control signal, it is possible to reduce the accuracy (and hence reduce the resource usage and energy consumption; or enable more complex NN architectures) as depicted in Table 3.

Table 3: N^2C^2 Precision Configuration

Precision (bit)	Guard bits	Control Signals	Description	Throughput
n=32	p=8	bw="00"	default precision	1x
n=16	p=4	bw="01"	half precision	2x
n=8	p=2	bw="10"	quarter precision	4x
n=4	p=1	bw="11"	octave precision	8x

Precision down-scaling is not the only option. It is also possible to up-scale the precision (e.g. to have n=64) by using two or more N^2C^2. Figure 3 shows a simple example where two N^2C^2 blocks are connected together to obtain a 64-bit adder. In a similar way, it is possible to perform 64-bit (or even higher) multiplications.

[1] The activation function is stored in a Look up Table. The details about LUT implementation and its programming mode will be further detailed.

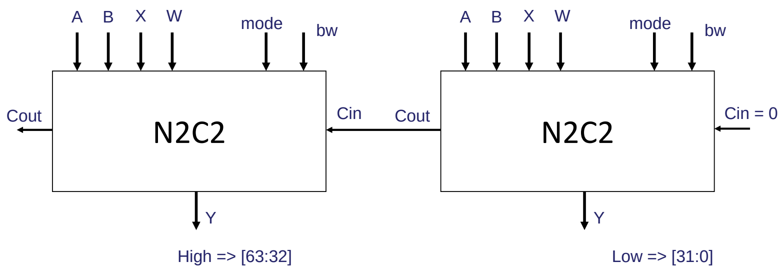

Figure 3 : N²C² 64-bit adder configuration

The N²C² has been implemented as a VHDL behavioral model in order to be able to carry out simulations to ensure the correct functionality and to obtain performances in terms of clock cycles. The full VHDL code is given in section 5 (appendix).

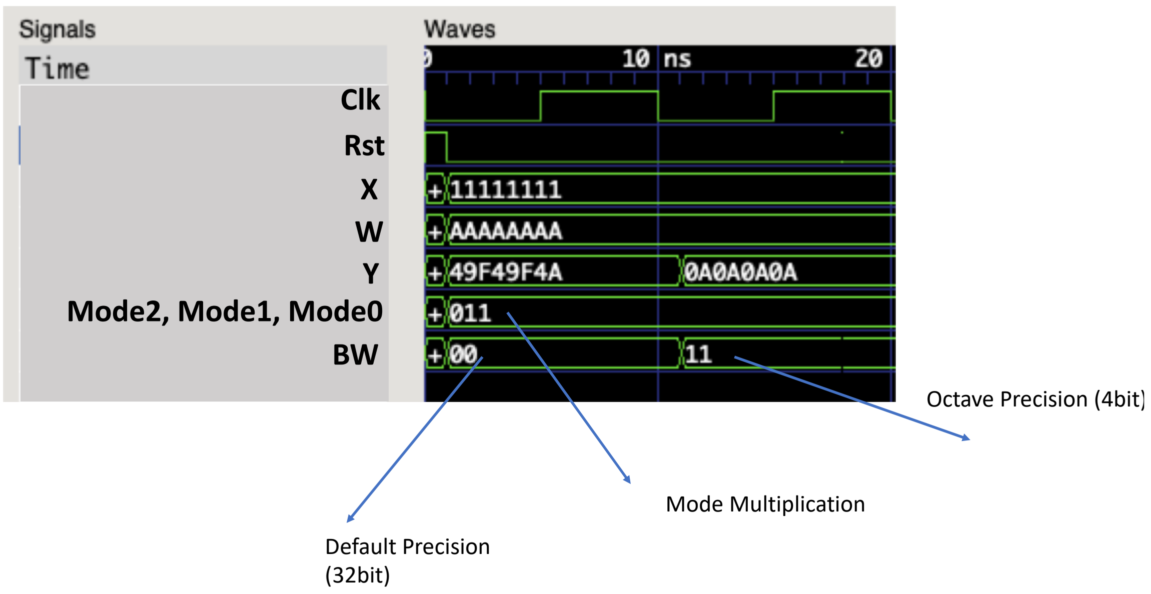

Figure 4 : N²C² example simulation multiplication mode

Figure 4 depicts the waveforms obtained from the simulation of the VHDL model. The simulation demonstrates the multiplication mode with two different levels of accuracy: (i) default (32-bit) precision and (ii) octave (32/8, i.e. 4-bit) precision. The result is thus Y=X*W depending on the precision. The throughput also depends on the precision: one multiplication at default precision, 8 multiplications in parallel executed at octave precision.

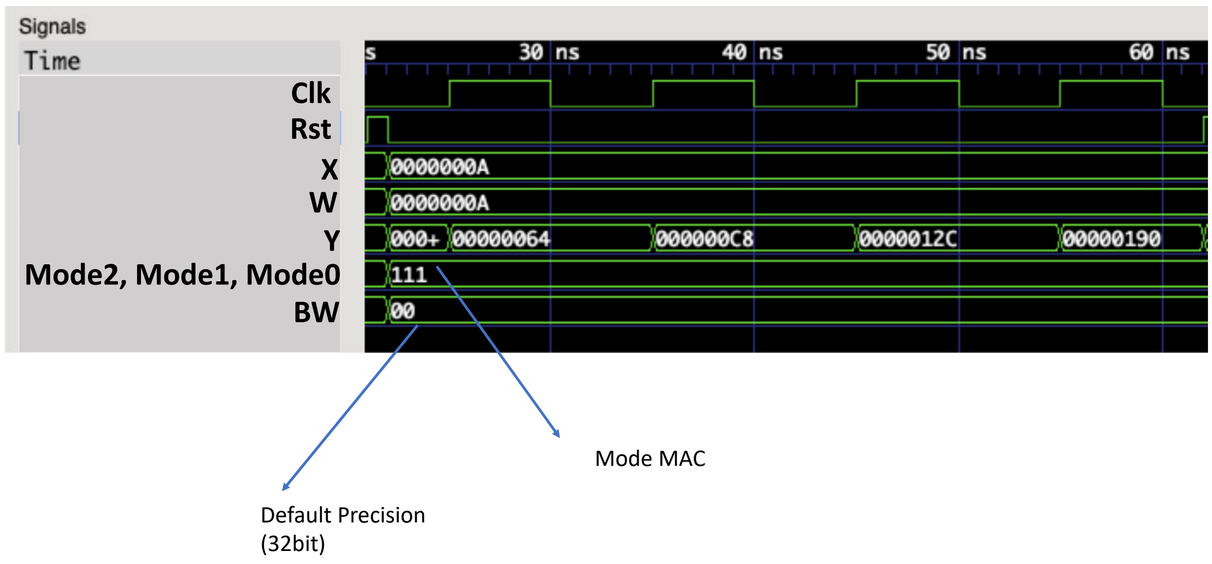

Figure 5 : N^2C^2 example simulation MAC mode

Figure 5 shows another example simulation, in which the N^2C^2 is configured as MAC at default precision. The output Y=X*W+Acc is updated at each clock cycle (in the example, X=W=0Ah).

II. INTENDED USE IN MATRIX MULTIPLICATION ARCHITECTURES

The N^2C^2 block can be used to parallelize matrix multiplication operation and thus speed up the computation of the transformers level. This provides a simple example of how to use an N^2C^2 network for matrix multiplication using default precision. In Figure 6, we show operator-level computation of two elements c_{12} and c_{33} as part of a 4x3 matrix *c* resulting from the multiplication of a 4x2 matrix *a* by a 2x3 matrix *b*. For the sake of simplicity, we will not cover the computation of the other elements of the *c* matrix, which can be found trivially from the given example.

$$\overset{4\times2\ \text{matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}} \overset{2\times3\ \text{matrix}}{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}} = \overset{4\times3\ \text{matrix}}{\begin{bmatrix} \cdot & c_{12} & c_{13} \\ \cdot & \cdot & \cdot \\ \cdot & c_{32} & c_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{33} = a_{31}b_{13} + a_{32}b_{23}$$

Figure 6 : Matrix multiplication example

Figure 7 sketches the N^2C^2 network and its configuration (structural connection). The four blocks are devoted to computing the results c_{12} and c_{33}. In the first cycle, each block is configured (see Table 2) to execute the multiplication between elements of *a* and *b* previously fetched from the memory and necessary for the computation.

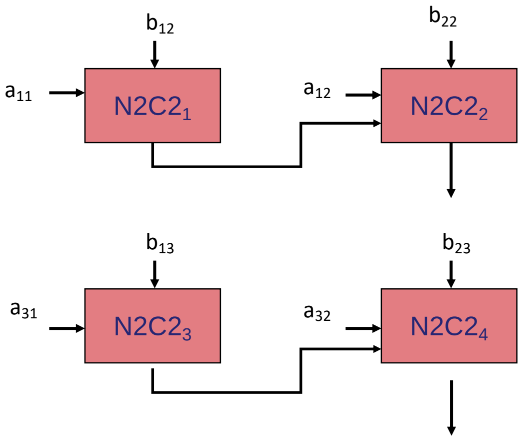

Figure 7 : N^2C^2 network first cycle

Figure 8 shows the same network but during the second cycle. Here, blocks $N^2C^2_2$ and $N^2C^2_4$ are each configured to execute an addition operation, resulting in c_{12} (from $N^2C^2_2$) and c_{33} (from $N^2C^2_4$).

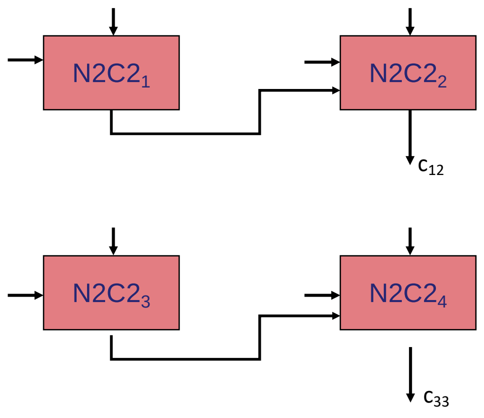

Figure 8 : N^2C^2 network second cycle

Figure 9 presents the simulation waveforms of the matrix multiplication presented above. Here we fix the following input values:

- $a_{11} = 2$; $a_{12} = 4$; $a_{31} = 5$; $a_{32} = 1$
- $b_{12} = 3$; $b_{22} = 3$; $b_{13} = 4$; $b_{23} = 10$;

Consequently, the outputs are:

- $c_{12} = a_{11}*b_{12} + a_{12}*b_{22} = 2*3 + 4*3 = 6 + 12 = 18$ (12h)
- $c_{33} = a_{31}*b_{13} + a_{32}*b_{23} = 5*4 + 1*10 = 20 + 10 = 30$ (1Eh)

In the first cycle, all N^2C^2 blocks are configured to multiplication mode in order to compute the intermediate values. In the second and final cycle, blocks 2 and 4 are configured to addition mode. The figure highlights the final results.

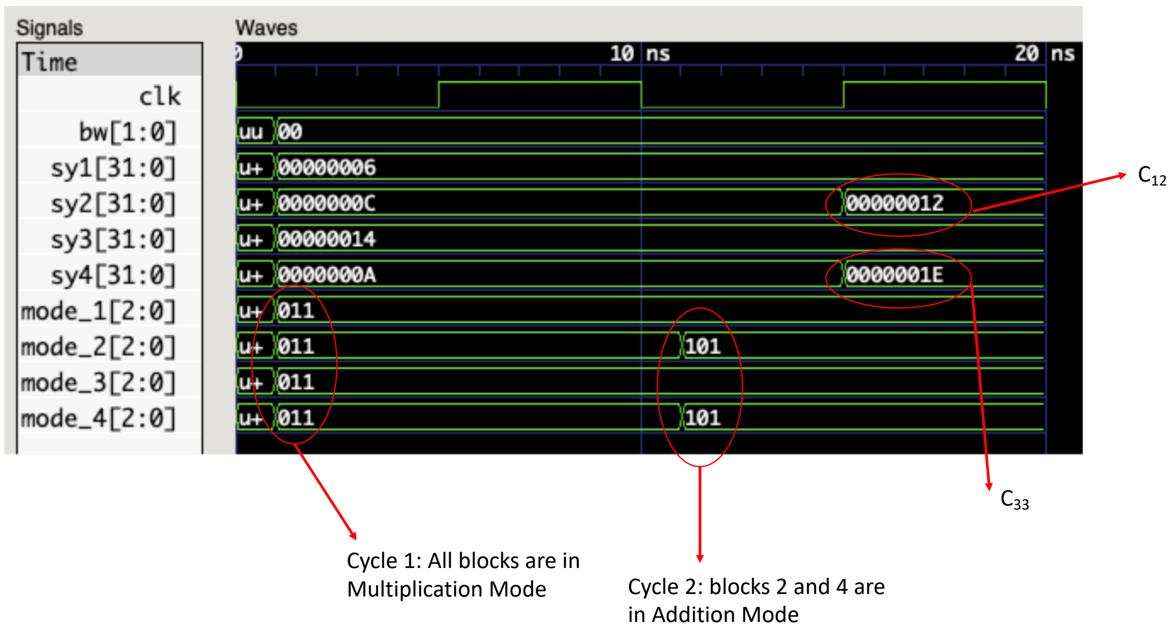

Figure 9 : N^2C^2 matrix multiplication example

3. Implementation and performance metrics

There are multiple technological variants and logic design styles that can implement the N^2C^2; and there are also multiple configurations (i.e. N^2C^2 network structures) that can execute the same operations. Each configuration is characterized by its latency (i.e. how many cycles are required to execute the operation), the area (i.e. how many N^2C^2 blocks are required) and finally the power/energy.

I. TECHNOLOGICAL VARIANTS

In this section we detail the various options to be explored in terms of technological implementation, which will then be evaluated at circuit-level to extract design data to be used in higher-level system simulations.

The FVLLMONTI project will explore several avenues of research at the technological level.

The baseline technology consists of the vertical nanowire field effect transistor (VNWFET) with a single gate (variant α). The design parameter at this level is essentially the number of nanowires per transistor, where the pitch between nanowires is a technological parameter that will be optimized according to tradeoffs between density (footprint), inter-nanowire capacitance, reliability and yield.

Gate stacking is advantageous for logic density where multiple transistors in series are needed. This will be explored at the device hardware level with a 2-gate stack (variant β) as well as virtually (using TCAD simulations) with a 3-gate stack (variant χ).

Ambipolarity (electrostatic doping of the VNWFET channel) enables fine-grain logic reconfigurability but also requires a polarity gate to control the type of majority carriers in the channel. It therefore requires two gates on a single device. This can be achieved either with a 1-gate stack using a U-type configuration (variant δ) or with a 2-gate stack, where one of the gates is the polarity gate (variant ε).

The integration of a ferroelectric layer in the transistor gate stack enables non-volatile behavior (memory or configuration) directly within the transistor. This will be explored at the device hardware level with a 1-gate stack (variant ϕ) as well as virtually (using TCAD simulations) with a 2-gate stack (variant γ).

Finally, the mixing of these variants will also be explored virtually and used in the design of dense, fine-grain reconfigurable, non-volatile logic gates (variants η and *fvllmonti*).

Table 4 summarizes the technological variants that will be used to build logic cell libraries in the context of implementation of N^2C^2.

Table 4: Summary of technological variants

Variant	Gate stack	Ambipolar	Ferroelectric	hardware
α	1	no	no	yes
β	2	no	no	yes
χ	3	no	no	no
δ	1	yes (U)	no	yes
ε	2	yes	no	no
ϕ	1	no	yes	yes
γ	2	no	yes	no
η	1	yes (U)	yes	no
fvllmonti	2	yes	yes	no

II. LOGIC DESIGN STYLES

Several design styles can be considered for implementation of logic functions. Each design style has its own merits and shortcomings, and thus a proper choice has to be made by designers in order to provide the correct functionality. This is true in all technologies, and is given a further degree of importance with the 3D VNWFET technology due to (a) the possibility of vertical stacking, (b) the possibility of reorienting the channel direction by 90°, (c) fine-grain reconfigurability and (d) non-volatile behavior.

Candidate design styles considered of interest for the VNWFET technology are:

- Static CMOS-like: this is the baseline design style using pull-up and pull-down switching networks to enable propagation of the voltage of one of the two power rails (gnd='0', Vdd='1') based on the state of the inputs, which can only access transistors via the gate terminals.
- Pass Transistor Logic (PTL): this design style propagates data directly through transistor channels by allowing inputs to access the transistor either on the gate terminal or on one of the source/drain terminals. While this leads to more compact logic structures, the transistor channel resistance can lead to limited fanout and logic level degradation. However, as gate stacking is naturally suited to multiple transistors in series, the PTL approach presents an opportunity for exploration of such compact structures.
- Non-volatile (NV) logic: non-volatile ferroelectric transistor devices enable the storage of an operand data value within the device itself, followed by the arrival of a second operand data value. This approach is particularly well adapted to applications where one operand varies rarely (e.g. NN weight coefficient) while the other varies often.
- Ambipolar logic: electrostatic doping enables a single hardware device to achieve either n-type or p-type switching functionality, and therefore leads to attractive solutions for fine-grain reconfigurability, flexible hardware substrates and dense, regular architectures. The combination of ambipolarity and ferroelectric behavior also enables the non-volatile storage of a configuration value. A generic tile-based structure

presents opportunities for exploration. Particular points of concern will be leakage current and interconnect limitations.

Examples of these four types of logic design style are given in Figure 10.

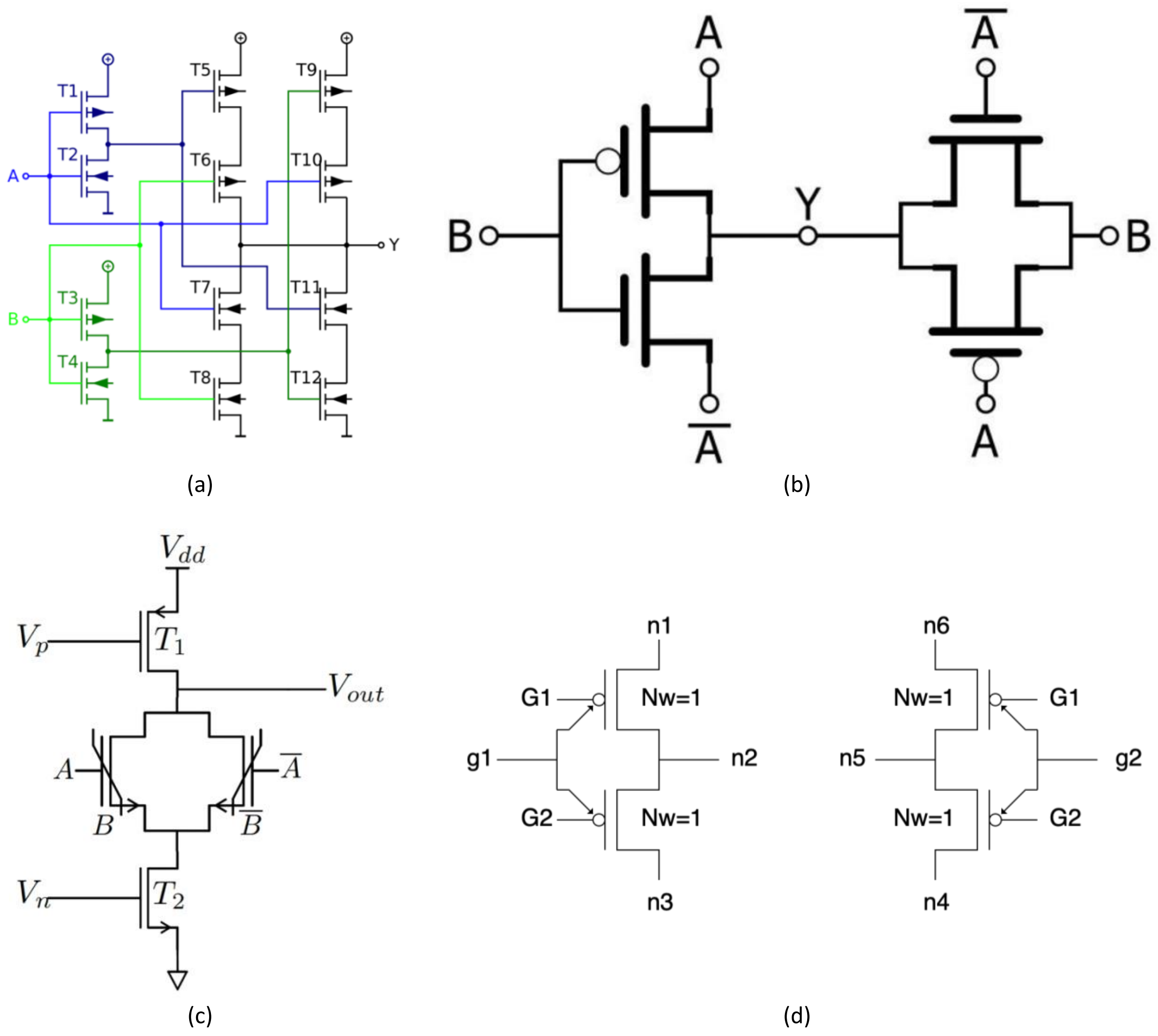

Figure 10 : Examples of logic design styles to be explored in view of N^2C^2 implementation. Static CMOS design style – XOR gate (a). PTL design style – XOR gate (b). Non-volatile design style – dynamic XOR gate (c). Ambipolar design style – reconfigurable tile (d).

III. PERFORMANCE METRICS

Specifications for the design space (i.e. performance metrics to be extracted from N^2C^2 hardware measurements and/or simulations) are detailed in this section. The design spaces will be populated with data in the form of Pareto Fronts and originating from circuit-level simulations using technological variants described in the previous section in D4.05b.

Table 5: Summary of performance metrics

Metric	Detail	Units	Comments
V_{dd}_nom	Nominal operating supply voltage	V	
V_{dd}_min	Min operating supply voltage	V	
T_{op}_nom	Nominal operating temperature	°C	
T_{op}_max	Max operating temperature	°C	
N_{ctrl}	Number of control inputs		
N_{in}/N_{out}	Number of data inputs / outputs		
T_{prog}	Programming time per function	s	Measured under nominal operating conditions
E_{prog}	Programming energy per function	J	Measured under nominal operating conditions
L_{ex}	Execution latency per function	s	Measured under nominal operating conditions
E_{ex}	Execution energy per function	J	Measured under nominal operating conditions
Thr_{ex}	Execution throughput per function	bits/s	Measured under nominal operating conditions
N_{cells}	Resource count (number of cells)		
Vol	Volume	μm^3	
Err_count	Reliability	errors / operation	Measured under nominal operating conditions and worst case operating conditions

4. Conclusion

This deliverable has described the first version of the virtual scalable Neural Network Compute Cube (N^2C^2). Its principal function is to carry out element-wise non-volatile matrix multiplication, accumulation and activation through a non-linear function. We have covered the architectural specifications and description of planned implementations of the N^2C^2 block, including technological variants, logic design styles and performance metrics.

As technology development and logic cell design is still in early stages, this version of D4.05 is intended to serve as a reference document. This information will be used mainly in WP4 (to focus logic cell design work towards a scaled down version of N^2C^2 in D4.4 scheduled for M30+2 as well as a second version of the virtual scalable N^2C^2 in D4.5b scheduled for M36+2). In particular, this will serve in logic cell library development in T4.1 (Logic cell design, optimization and validation) compatible with logic synthesis approaches:

- For baseline Static-CMOS architecture: fixed combinatorial logic blocks (classic CMOS / PTL design style), volatile memory (classic SRAM)
- For SRAM / NV-LUT logic in memory (results caching)
- For reconfigurable logic blocks (classic SRAM-based LUT / non-volatile LUT)
- For data-reconfigurable logic – non-volatile logic
- For ambipolar connection-based / NV-ambipolar 3D-nanofabric – high-expressivity logic blocks (XOR / tile / NV-tile)

It will also be used in WP5 to enable architectural exploration in D5.2 scheduled for M20+2 and M36+2.

As the FVLLMONTI project progresses, the content of this deliverable will also be updated to reflect opportunities and limitations that appear according to the state of technology and logic circuit development.

5. Appendix

I. ARCHITECTURE CODE

package.vhd

```
package n2c2_package is
       constant n: integer := 32;
       constant p: integer := 8;
 end n2c2_package;
```

N2C2.vhd

```
library work;
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;




---------------------------------------------------

entity N2C2 is

port(   A, B, X, W:    in std_logic_vector(n - 1 downto 0);
        clk :   in std_logic;
        rst :   in std_logic;
        Cin :   in std_logic;
        Neuron: in std_logic;
        BW :    in std_logic_vector (1 downto 0);
        Mode :  in std_logic_vector (2 downto 0);
        Cout :  out std_logic;
        Y:      out std_logic_vector(n -1 downto 0)
);

end N2C2;

----------------------------------------------------

architecture behv of N2C2 is

---- Component declaration

component acc_reg is

port(   D:      in std_logic_vector(2*n + p -1 downto 0);
        clk:    in std_logic;
        rst:    in std_logic;
        Q:      out std_logic_vector(2*n + p -1  downto 0)
);
end component;


component multiplier is

port(   A, B:   in std_logic_vector(n-1 downto 0);
        BW:     in std_logic_vector (1 downto 0);
        Z:      out std_logic_vector(2*n-1 downto 0)
);

end component;

component adder is
```

```
port(   A, B:   in std_logic_vector(2*n + p -1 downto 0);
        Cin :   in std_logic;
        Cout :  out std_logic;
        Z:      out std_logic_vector(2*n + p -1 downto 0)
);

end component;

---- Internal signals

signal out_multiplier :                std_logic_vector (2*n -1 downto 0);
signal in_adder_A, in_adder_B:         std_logic_vector (2*n + p -1 downto 0);
signal out_adder:              std_logic_vector (2*n + p -1 downto 0);
signal in_acc_reg:             std_logic_vector (2*n + p -1 downto 0);
signal out_acc_reg:            std_logic_vector (2*n + p -1 downto 0);
signal out_truncation:                 std_logic_vector (n-1 downto 0);



begin

        -- Components port maps

        MULT: multiplier port map (X, W, BW, out_multiplier);
        ADD: adder port map (in_adder_A, in_adder_B, Cin, Cout, out_adder);
        ACC: acc_reg port map (in_acc_reg, clk, rst, out_acc_reg);


        -- processes

        mux_mul_to_add : process (Mode(1), B, out_multiplier)
        begin
                if (Mode(1) = '1') then
                        in_adder_B <= ( (2*n + p -1 downto 2*n => '0') & out_multiplier);
                else
                        in_adder_B <= ( (2*n + p -1 downto n => '0') & B);
                end if;
        end process;


        mux_MAC_to_add : process (Mode(0), A, out_acc_reg)
        begin
                if (Mode(0) = '1') then -- Accumulation
                        in_adder_A <=  out_acc_reg;
                else
                        in_adder_A <= ( (2*n + p -1 downto n => '0') & A);
                end if;
        end process;

        mux_truncation : process (Mode(0), Mode(2),   out_acc_reg, out_adder, out_multiplier)
        begin
                if (Mode(0) = '1' and Mode(2) = '1') then -- Accumulation
                        out_truncation <=  out_acc_reg(n-1 downto 0);
                else
                        if (Mode(0) = '0' and Mode(2) = '0') then
                                out_truncation <= out_adder(n-1 downto 0);

                        elsif (Mode(0) = '1' and Mode(2) = '0') then

                                out_truncation <= out_multiplier(n-1 downto 0);

                        end if;

                end if;
        end process;

        -- mux_neuron : to be implemented
        in_acc_reg <= out_adder;

        Y <= out_truncation;

end behv;

---------------------------------------------------
```

II. COMPONENT CODE

multiplier.vhd

```
library work;
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;




---------------------------------------------------

entity multiplier is

port(   A, B:   in std_logic_vector(n-1 downto 0);
        BW:     in std_logic_vector (1 downto 0);
        Z:      out std_logic_vector(2*n-1 downto 0)
);

end multiplier;

----------------------------------------------------

architecture behv of multiplier is


begin

    process(A, B, BW)
    begin
        case BW is
                when "00" =>   Z <= A*B; -- full precision
                when "01" => -- half precision
                        Z(2*n-1 downto n) <= A(n-1 downto n/2) * B(n-1 downto n/2);
                        Z(n-1 downto 0) <= A(n/2-1 downto 0) * B(n/2-1 downto 0);
                when "10" => -- quarter precision
                        Z(2*n-1 downto 3*n/2) <= A(n-1 downto 3*n/4) * B(n-1 downto 3*n/4);
                        Z(3*n/2 - 1  downto n) <= A(3*n/4 - 1  downto n/2) * B(3*n/4 - 1 downto n/2);
                        Z(n - 1  downto n/2) <= A(n/2 - 1  downto n/4) * B(n/2 - 1 downto n/4);
                        Z(n/2 - 1  downto 0) <= A(n/4 - 1  downto 0) * B(n/4 - 1 downto 0);
                when "11" => -- octave precision
                        Z(2*n-1 downto 7*n/4) <= A(n-1 downto 7*n/8) * B(n-1 downto 7*n/8);
                        Z(7*n/4 - 1  downto 6*n/4) <= A(7*n/8 - 1  downto 6*n/8) * B(7*n/8 - 1 downto
6*n/8);
                        Z(6*n/4 - 1  downto 5*n/4) <= A(6*n/8 - 1  downto 5*n/8) * B(6*n/8 - 1 downto
5*n/8);
                        Z(5*n/4 - 1  downto n) <= A(5*n/8 - 1  downto 4*n/8) * B(5*n/8 - 1 downto
4*n/8);
                        Z(n - 1  downto 3*n/4) <= A(4*n/8 - 1  downto 3*n/8) * B(4*n/8 - 1 downto
3*n/8);
                        Z(3*n/4 - 1  downto 2*n/4) <= A(3*n/8 - 1  downto 2*n/8) * B(3*n/8 - 1 downto
2*n/8);
                        Z(2*n/4 - 1  downto n/4) <= A(2*n/8 - 1  downto 1*n/8) * B(2*n/8 - 1 downto
1*n/8);
                        Z(n/4 - 1  downto 0) <= A(1*n/8 - 1  downto 0) * B(1*n/8 - 1 downto 0);

                when others => Z <= A*B;
        end case;

    end process;



end behv;

---------------------------------------------------
```

adder.vhd

```
library work;
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;




---------------------------------------------------

entity adder is

port(   A, B:   in std_logic_vector(2*n + p -1 downto 0);
        Cin :   in std_logic;
        Cout :  out std_logic;
        Z:      out std_logic_vector(2*n + p -1 downto 0)
);

end adder;

----------------------------------------------------

architecture behv of adder is


signal tmp: std_logic_vector (2*n + p downto 0);

begin

    process(A, B, Cin)
    begin
        tmp <= ('0'&A) + ('0'&B)  + ((2*n +p downto 1  => '0')&Cin);
    end process;

    Cout <= tmp(2*n+p);
    Z <= tmp (2*n +p -1 downto 0);

end behv;

---------------------------------------------------
```

register.vhd

```
library work;
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;




---------------------------------------------------

entity acc_reg is

port(   D:      in std_logic_vector(2*n + p -1 downto 0);
        clk:    in std_logic;
        rst:    in std_logic;
        Q:      out std_logic_vector(2*n + p -1  downto 0)
);
end acc_reg;
```

```
----------------------------------------------------

architecture behv of acc_reg is

    signal Q_tmp: std_logic_vector(2*n + p - 1 downto 0);

begin

    process(D, clk, rst)
    begin

       if rst = '1' then
            Q_tmp <= (Q_tmp'range => '0');
       elsif (clk='1' and clk'event) then
               Q_tmp <= D;
       end if;

    end process;

    Q <= Q_tmp;

end behv;

---------------------------------------------------
```

III. TESTBENCH CODE

N2C2_tb.vhd

```
library work;
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;




---------------------------------------------------

entity N2C2_tb is
end N2C2_tb;

----------------------------------------------------

architecture behv of N2C2_tb is

---- Component declaration


component N2C2 is

port(   A, B, X, W:    in std_logic_vector(n - 1 downto 0);
        clk :  in std_logic;
        rst :  in std_logic;
        Cin :  in std_logic;
        Neuron: in std_logic;
        BW :   in std_logic_vector (1 downto 0);
        Mode : in std_logic_vector (2 downto 0);
        Cout : out std_logic;
        Y:     out std_logic_vector(n -1 downto 0)
);

end component;



---- Internal signals
```

```
signal clk, rst, Cin, Cout, Neuron:         std_logic;

signal BW :             std_logic_vector (1 downto 0);
signal Mode :           std_logic_vector (2 downto 0);
signal sA, sB, sX, sW, sY:          std_logic_vector (n-1 downto 0);



begin

        -- Components port maps

        N2C2_1: N2C2  port map (sA, sB, sX, sW, clk, rst, Cin, Neuron, BW, Mode, Cout, sY);


        -- processes

        Workload : process
        begin
                rst <= '1';
                wait for 1 ns;
                rst <= '0';
                -- 32 bit
                BW <= "00";
                Neuron <= '0';
                Mode <= "011";  -- multiplication
                Cin <= '0';
                sA <=  x"00000000";
                sB <=  x"00000000";
                sX <= x"AAAAAAAA";
                sW <= x"11111111";
                wait for 10 ns;
                -- 4 bit
                BW <= "11";
                wait for 10 ns;

                -- MAC 32 bits
                rst <= '1';
                wait for 1 ns;
                rst <= '0';
                Mode <= "111";
                BW <= "00";
                sX <= x"0000000A";
                sW <= x"0000000A";
                wait for 40 ns;




        end process;

        Clock : process
        begin
                clk <= '0';
                wait for 5 ns;
                clk <= '1';
                wait for 5 ns;

        end process;


end behv;

---------------------------------------------------
```

N2C2_MM_tb.vhd

```
library work;
```

```
use work.n2c2_package.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;



---------------------------------------------------

entity N2C2_MM_tb is
end N2C2_MM_tb;

----------------------------------------------------

architecture behv of N2C2_MM_tb is

---- Component declaration


component N2C2 is

port(   A, B, X, W:    in std_logic_vector(n - 1 downto 0);
        clk :   in std_logic;
        rst :   in std_logic;
        Cin :   in std_logic;
        Neuron: in std_logic;
        BW :    in std_logic_vector (1 downto 0);
        Mode :  in std_logic_vector (2 downto 0);
        Cout :  out std_logic;
        Y:      out std_logic_vector(n -1 downto 0)
);

end component;



---- Internal signals


signal clk, rst, Cin, Cout, Neuron:         std_logic;

signal BW :           std_logic_vector (1 downto 0);
signal Mode_1, Mode_2, Mode_3, Mode_4 :      std_logic_vector (2 downto 0);
signal sA11, sA12, sA31, sA32, sB12, sB22, sB13, sB23, sY1, sY3, sY2, sY4, dummy: std_logic_vector
(n-1 downto 0);



begin

       -- Components port maps

       N2C2_1: N2C2  port map (dummy, dummy, sA11, sB12, clk, rst, Cin, Neuron, BW, Mode_1, Cout,
sY1);
       N2C2_2: N2C2  port map (dummy, sY1, sA12, sB22, clk, rst, Cin, Neuron, BW, Mode_2, Cout,
sY2);
       N2C2_3: N2C2  port map (dummy, dummy, sA31, sB13, clk, rst, Cin, Neuron, BW, Mode_3, Cout,
sY3);
       N2C2_4: N2C2  port map (dummy, sY3, sA32, sB23, clk, rst, Cin, Neuron, BW, Mode_4, Cout,
sY4);


       -- processes

       Workload : process
       begin
              rst <= '1';
              wait for 1 ns;
              rst <= '0';
              -- 32 bit
              BW <= "00";
              Neuron <= '0';
              Mode_1 <= "011";  -- multiplication
              Mode_2 <= "011";  -- multiplication
```

```
               Mode_3 <= "011";  -- multiplication
               Mode_4 <= "011";  -- multiplication
               Cin <= '0';
               dummy <=  x"00000000";
               sA11 <= x"00000002";
               sB12 <= x"00000003";
               sA12 <= x"00000004";
               sB22 <= x"00000003";  -- C12 = A11*B12 + A12*B22 = 2*3 + 4*3 = 6 + 12 = 18  (dec)


               sA31 <= x"00000005";
               sB13 <= x"00000004";
               sA32 <= x"00000001";
               sB23 <= x"0000000A";  -- C33 = A31*B13 + A32*B23 = 5*4 + 1*10 = 20 + 10 = 30 (dec)

               wait for 10 ns;
               Mode_2 <= "101";  -- multiplication
               Mode_4 <= "101";  -- multiplication


               wait for 10 ns;




       end process;

       Clock : process
       begin
               clk <= '0';
               wait for 5 ns;
               clk <= '1';
               wait for 5 ns;

       end process;


end behv;

---------------------------------------------------
```