FPNA, FPNN: from programmable fields to topologically simplified neural networks

Bernard Girau

FPNA, FPNN: from programmable fields to topologically simplified neural networks.

Bernard Girau

1 Introduction

Reconfigurable hardware devices such as FPGAs (field programmable gate arrays) are cheap, flexible, and they offer both digital hardware efficiency and simple software-like handling. Their main advantages for neural network implementations are : reprogrammable FPGAs allow prototyping, FPGAs may be used for embedded applications, FPGA-based implementations may be mapped onto new improved FPGAs (unlike the use of neuroprocessors, which rapidly become outdated). However, the 2D-topology of FPGAs does not allow to handle the *connection complexity* of standard neural network models. Moreover, FPGAs still implement a limited number of logic gates, whereas neural computations (multipliers, transfer functions) are *area-consuming* operators. Usual solutions ([3, 8, 11, 5, 4, 7, 2]) handle sequentialized computations with a FPGA used as a small neuroprocessor, or they implement very small low-precision neural networks without on-chip learning. Connectivity problems are not solved even by the use of several reconfigurable FPGAs with a bit-serial arithmetic ([6]), or by the use of small-area stochastic bitstream operators ([1]).

In [10], we propose an implementation method for multilayer perceptrons of any size on a single FPGA, with on-chip learning and adaptable precision. This work takes advantage of an area-saving on-line arithmetic that is well-adapted to neural computations. It also uses an original parallelization of the internal computations of each neuron. Yet, this implementation method barely exploits the parallelism induced by neural network architectures.

The work described in this report aims at developping *neural architectures* that are easy to *map onto FPGAs*, thanks to a *simplified topology* and an original *data exchange scheme*, without having any significant loss of approximation capability. It has been achieved thanks to the definition of a set of neural models called *Field Programmable Neural Arrays* (FPNA). FPNAs may lead to the definition of neural networks adapted to hardware topological constraints. Different such neural networks may be derived from a given FPNA. They are called *Field Programmed Neural Network* (FPNN). They reconcile the high connection density of neural architectures with the need of a limited interconnection scheme in hardware implementations.

This report proposes a brief overview of FPNA and FPNN definitions, computations, and implementations. A global study of FPNAs and FPNNs may be found in [9].

2 FPNAs, FPNNs

The distinction between FPNAs and FPNNs is mainly linked to implementation properties. A FPNA corresponds to a given set of neural resources that are organized according to specific neighborhood relations. A FPNN is a way to use this set of resources: it only induces local configuration choices. Therefore, the implementation needs of two different FPNNs are the same, as long as they are based on the same FPNA.

2.1 FPNAs

2.1.1 From FPGAs to FPNAs

The first aim of the FPNA concept is to develop neural structures that are easy to map onto digital hardware, thanks to a simplified and very flexible topology. Therefore, the structure of a FPNA derives from FPGA principles (complex functions realized by means of a set of simple programmable resources), while the nature and the relations of FPNA resources derive from the mathematical processing FPNAs have to perform.

These resources are communication links and computation elements. In a standard neural model, each communication link is a connection between the output of a neuron and an input of another neuron. The number of inputs of each neuron is its fan-in in the connection graph. On the contrary, communication links and neurons become *autonomous* in a FPNA: their dependencies are freely programmable.

2.1.2 FPNA resources

FPNAs are defined to compute partial convolutions for non-linear regression, as standard multilayer neural networks do. Nevertheless their architecture is simplified with respect to standard neural models.

A FPNA is intended to use programmable hardware principles to allow direct mappings of various neural networks onto digital hardware. Thus it is made of a programmable set of neural resources. Two kinds of autonomous FPNA resources naturally appear: neurons that apply standard neural functions to a set of input values on one hand, and communication links that behave as independent affine operators on the other hand.

These resources may be handled in different ways. The easiest scheme would allow to allocate any communication link to any neuron, with the help of an underlying programmable interconnection net (pruned standard neural networks, and weight sharing). Topological problems may still appear (such as high fan-ins). And weight sharing would induce few different $\vec{w}$ weight vectors. Therefore, the chosen scheme allows to connect any communication link to any *local resource*. The aim of locality is to reduce topological problems, whereas connected communication links result in more various weight vectors.

More precisely, the communication links connect the nodes of a directed graph, each node contains one neuron. The specificity of FPNAs is that relations between *any* of the local resources of each node may be freely set. A link may be connected or not to the local neuron *and to the other local links*. Direct connections between affine links appear, so that the FPNA may compute numerous composite affine transforms. These compositions create numerous *virtual neural connections*, so that different convolution terms may be obtained with a reduced number of connection weights.

2.1.3 Formal definition of FPNAs

A FPNA is defined by means of:

- a directed graph $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a finite set of nodes, and $\mathcal{E}$ is a set of directed edges ($\mathcal{E}$ can be seen as a subset of $\mathcal{N}^2$),
- a set of neurons (θ_n, i_n, f_n), for each n in $\mathcal{N}$: θ_n is a real threshold, i_n is an iteration operator (a function from $\mathbb{R}^2$ to $\mathbb{R}$), and f_n is a transfer function (from $\mathbb{R}$ to $\mathbb{R}$),
- a set of affine functions $x \mapsto W_n(p)x + T_n(p)$ for each (p, n) in $\mathcal{E}$.

For each node n, the set of the direct predecessors (resp. successors) of n is defined by $Pred(n) = \{p \in \mathcal{N} \mid (p,n) \in \mathcal{E}\}$ (resp. $Succ(n) = \{s \in \mathcal{N} \mid (n,s) \in \mathcal{E}\}$). The set of the input nodes is $\mathcal{N}_i = \{n \in \mathcal{N} \mid Pred(n) = \emptyset\}$.

2.1.4 Interpretation

Resources are associated with the nodes, whereas locality is defined by the edges. For each node $n \in \mathcal{N}$, there is one neuron resource and as many communication links as this node has got predecessors. Each communication link is associated with an affine operator. A neuron resource is defined by (θ_n, i_n, f_n), since it will handle any neuron computation as in a sequential program. Indeed, any standard neuron computation may be performed by means of a loop that updates a variable with respect to the neuron inputs, and a final computation that maps this variable to the neuron output. θ_n stands for the initialization value (see [9]). The iteration function i_n stands for the updating function inside the loop. The neuron output is finally computed with f_n.

2.2 FPNNs

A FPNN (field programmed neural network) is a FPNA where resources have been connected in a specific way.

2.2.1 Formal definition of FPNNs

A FPNN is specified by means of:

- a FPNA (that is the neural resources it can use),
- for each node n in $\mathcal{N} - \mathcal{N}_i$,
  - a positive integer a_n (number of iterations before a neuron applies its transfer function)
  - for each p in $Pred(n)$, a binary value $r_n(p)$ (set to 1 iff the link (p,n) and the neuron in n are connected),
  - for each s in $Succ(n)$, a binary value $S_n(s)$ (set to 1 iff the neuron in n and the link (n,s) are connected),
  - for each p in $Pred(n)$ and each s in $Succ(n)$, a binary value $R_n(p,s)$ (set to 1 iff the links (p,n) and (n,s) are connected),
- for each input node n in $\mathcal{N}_i$,
  - a positive integer c_n (number of global inputs sent to this node),
  - for each s in $Succ(n)$, a binary value $S_n(s)$ (see above).

2.2.2 Computing in a FPNN

Several computation methods have been defined for the FPNNs. Their common principle may be described as follows:

- All resources behave independently.
- A resource receives values. For each value,
  - the resource applies its local operator(s),

– the result is sent to all neighbouring resources to which it is locally connected (a neuron resources waits for a_n values before sending any result to its neighbours).

The main differences with the standard neural network computation are:

- A resource may or may not be connected to a neughbouring resource. It is set by the $r_n(p)$, $S_n(s)$ and $R_n(p,s)$ configuration values.
- A communication link may directly send values to other communication links.
- A resource (even a communication link) may handle several values during a single FPNN computation process.

2.2.3 Asynchronous sequential computation

This computation handles a list of tasks $\mathcal{L}$ that are processed according to a FIFO scheduling. Each task $[(p,n),x]$ corresponds to a value x sent on a communication link (p,n).

Initialization:

For each input node n in $\mathcal{N}_i$, c_n values $\left(x_n^{(i)}\right)_{i=1..c_n}$ are given (global inputs of the FPNN), and the corresponding tasks $[(n,s),x_n^{(i)}]$ are created for all s in $Succ(n)$ such that $S_n(s)=1$. The order of creation corresponds to a lexicographical order on (n,i,s) (with respect to the order of $\mathcal{N}$).

Sequential processing:

While $\mathcal{L}$ not empty

Let $[(p,n),x]$ be the first element in $\mathcal{L}$.

1. *suppress this element in $\mathcal{L}$*
2. *$x' = W_n(p)x + T_n(p)$*
3. *for all $s \in Succ(n)$ such that $(R_n(p))(s)=1$, create $[(n,s),x']$ according to the order on s*
4. *if $r_n(p)=1$*
   - *increment c_n*
   - *update x_n : $x_n = i_n(x_n,x')$*
   - *if $c_n = a_n$*

   (a) *$y = f_n(x_n)$*

   (b) *$c_n = 0$*

   (c) *$x_n = \theta_n$*

   (d) *for all $s \in Succ(n)$ such that $S_n(s)=1$, create $[(n,s),y]$ according to the order on s*

If $r_n(p)=1$, the neuron in n is said to be receiving the value of task $[(p,n),x]$.

2.2.4 Asynchronous parallel computation

A request-acknowledge protocol must be handled. A request $req[(p,n),(n,s),x]$ corresponds to a value x sent by resource (p,n) to resource (n,s). Resource (n,n) stands for the neuron in n.

Initialization:

- For each input node n in $\mathcal{N}_i$, c_n values $\left(x_n^{(i)}\right)_{i=1..c_n}$ are given (global inputs of the FPNN), and the corresponding requests $req[(n,n),(n,s),x_n^{(i)}]$ are created for all s in $Succ(n)$ so that $S_n(s)=1$.
- Each node n in $\mathcal{N}-\mathcal{N}_i$ has got local variables c_n and x_n, initially set as $c_n=0$ and $x_n=\theta_n$.

Concurrent processing:

All resources in parallel sequentially handle all the requests that they receive. Resource (n_1,n_2) processes request $req[(n_0,n_1),(n_1,n_2),x]$ (chosen with a fair policy among the unprocessed requests already sent to (n_1,n_2)) as follows:

1. acknowledgement for (n_0,n_1)
2. if $n_1=n_2$ then
   - $c_{n_1}=c_{n_1}+1$
   - $x_{n_1}=i_{n_1}(x_{n_1},x)$
   - if $c_{n_1}=a_{n_1}$ then
     - for all s in $Succ(n_1)$ so that $S_{n_1}(s)=1$, create $req[(n_1,n_1),(n_1,s),f_{n_1}(x_{n_1})]$
     - wait acknowledgements
     - reset : $c_{n_1}=0 \quad x_{n_1}=\theta_{n_1}$

   else
   - for all $s\in Succ(n_2)$ so that $R_{n_2}(n_1,s)=1$, create $req[(n_1,n_2),(n_2,s),W_{n_2}(n_1)x+T_{n_2}(n_1)]$
   - if $r_{n_2}(n_1)=1$ then create $req[(n_1,n_2),(n_2,n_2),W_{n_2}(n_1)x+T_{n_2}(n_1)]$
   - wait acknowledgements

2.2.5 Some results

Standard neural models may be used as FPNAs (so that each neuron and its input and output connections become freely connectable). And any standard neural network can be exactly simulated by a FPNN based on the FPNA form of this neural network. For example, one can use the multilayer topology of a MLP to build a FPNA, and then set a derived FPNN that computes the same function as the original MLP[1]. FPNAs may also be built with a simple 2D topology, whereas derived FPNNs compute functions of far more complex neural networks.

Let $\mathcal{R}$ be a relation between the neural resources of a FPNN, defined by:

$$(n_0,n_1)\mathcal{R}(n_1,n_2) \text{ iff } \begin{cases} & n_2=n_1 \text{ and } r_{n_1}(n_0)=1 \\ or & R_{n_1}(n_0,n_2)=1 \\ or & n_0=n_1 \text{ and } S_{n_1}(n_2)=1 \end{cases}$$

A FPNN is feedforward iff the transitive closure of $\mathcal{R}$ is a partial order for the FPNA resources. Determinism conditions depend on neural functionalities and arities. All defined computation schemes lead to the same neuron output values with such FPNNs.

[1] To obtain this: $\forall n\in\mathcal{N}_i\ c_n=1 \quad \forall n\ i_n(x,y)=x+y,\ f_n(x)=\sigma(x),\ a_n=\#Pred(n)$
$\forall (p,n,s)\ T_n(p)=0,\ r_n(p)=1,\ S_n(s)=1,\ R_n(p,s)=0$

2.2.6 Recurrent FPNNs: synchronized computations

The above computation schemes can not satisfactorily handle recurrent FPNNs. Yet slight changes are required. In a synchronized FPNN computation, synchronisation barriers are used in order to separate the strict sending of neural output values from the simple direct value transmissions between connected communication links. It allows to define a correct recurrent computation, provided that there is no loop in the direct connections between the links.

At a given moment, the task or request list (synchronized sequential or synchronized parallel computations) is processed as above, except for any value sent by a neuron (resource (n, n)). The corresponding task or request is put in a second list, without being processed. When the main list is emptied, a new synchronisation barrier is raised, and all waiting tasks or requests in the second list are set active in the main list, which processing starts again.

A precise description of these computation schemes, and of the specific properties of recurrent FPNNs is not the subject of this report. A next one will focus on such properties for both feedforward and recurrent FPNNs : halt, deadlock, determinism, computation equivalences, etc.

2.3 An example

FPNNs allow to obtain complex neural network behaviours with simple 2D topologies. Such FPNNs have been studied for the standard parity problem. They show great properties, despite the above results. The search for optimal two-hidden layer shortcut perceptrons in [12] has led to solve the d-dimensional parity problem[2] with only $\sqrt{d}(2+o(1))$ neurons. Shortcut connections and second hidden layer are essential in this work. This neural network uses $d(\sqrt{d}+1+o(1))$ weights. For all d, a FPNN with the same number of neurons, but only $\mathcal{O}(\frac{15}{2}\sqrt{d})$ weights *exactly* performs the same computation. Figure 1 shows the optimal shortcut network of [12] for the 11-dimensional parity problem. Figure 2 shows the equivalent FPNN.

3 FPGA implementation

Several implementation methods have been defined. Some have been optimized for particular FPNNs. This report proposes a very general implementation method that directly derives from the asynchronous parallel computation scheme, so as to express more clearly how time is inherent to FPNN computations.

3.1 Communication links

Figure 3 shows a possible implementation of a communication link. All flip-flops use asynchronous reset active on '1'.

`SELECT` :

> This block receives all request signals that may be sent by the predecessors of the communication link according to the FPNA topology. It also receives a signal "libre" (free), set to '1' when the communication link handles no request.
>
> Signal "debut" (start) is an output set to 1 during one clock cycle when a request processing begins. Signal "acq" is an acknowledge signal which is routed towards the resource that sent the current request. Signal "sel`(1..lp)`" codes the number of the selected request signal. It controls the input mux and the acknowledge dmux.

[2] The number of non zero values among the d coordinates must be classified as odd or even.

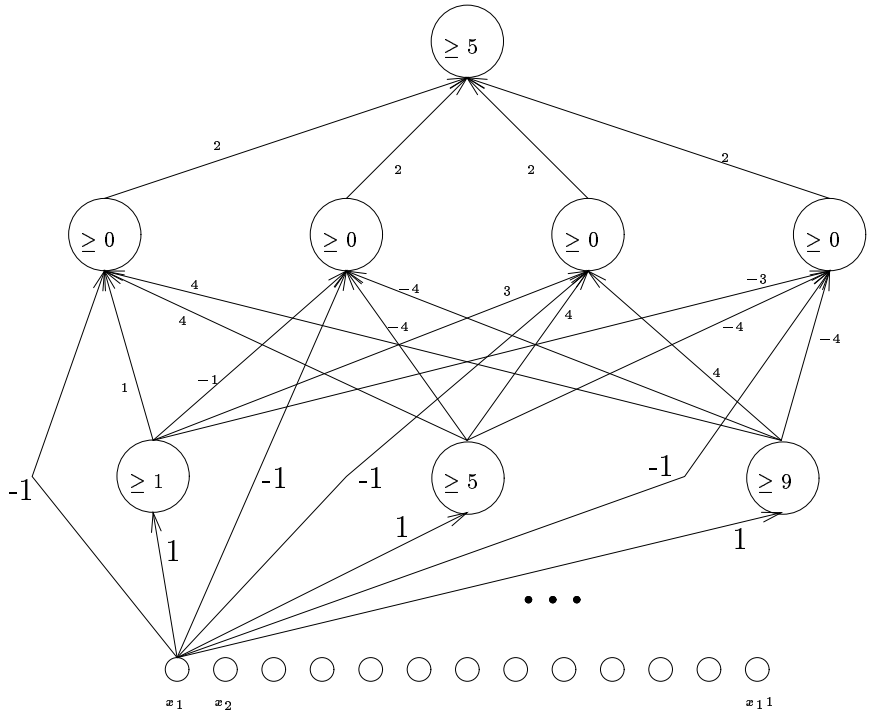

Figure 1: 3-layer shortcut perceptron for the 11-dimensional parity problem

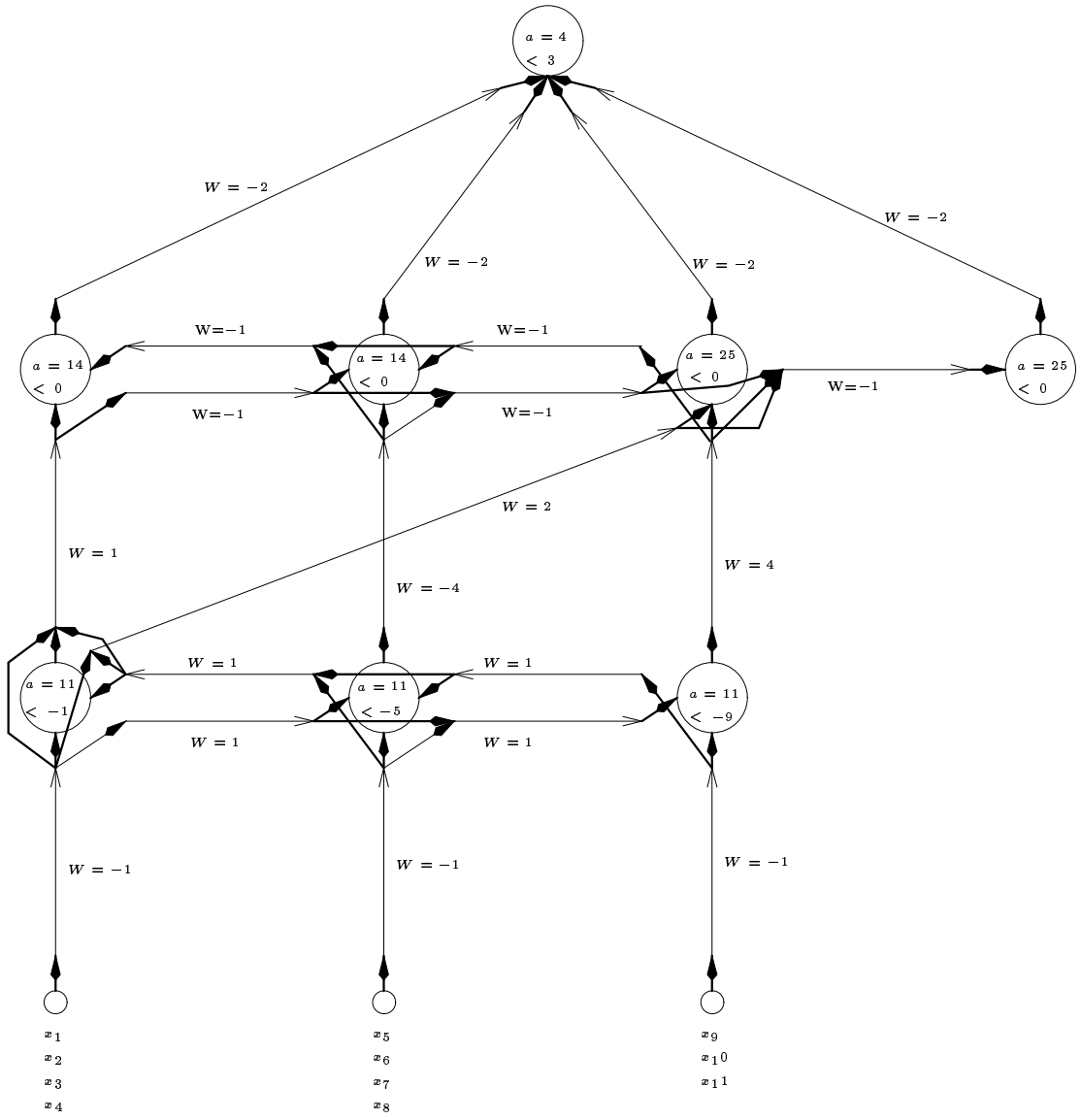

Figure 2: Equivalent FPNN

r3 r2 r1 r0 / d1 d0	0000	0001	0011	0010	0110	0111	0101	0100
00	00	00	01	01	01	01	10	10
01	00	00	00	01	10	10	10	10
11	00	00	00	01	01	00	00	10
10	00	00	00	01	01	00	00	10

r3 r2 r1 r0 / d1 d0	1100	1101	1111	1110	1010	1011	1001	1000
00	10	10	01	01	01	01	11	11
01	10	10	10	10	11	11	11	11
11	10	00	00	01	01	00	00	11
10	11	11	11	11	11	11	11	11

Table 1: Truth table of PRIO (value of `s1 s0`)

The architecture of a SELECT block for 4 predecessors and with a rotating priority policy is shown on figure 4. Table 1 is the truth table of the PRIO block (priority).

`MULT_ADD :`

This block receives the value stored in register X (value of the selected request). It also receives $W_n(p)$ and $T_n(p)$.

It outputs signal "pret" (ready) which is set to '1' when signal "`y(1..nb)`" contains the output value of the communication link (affine transform of X).

`libre :`

When register X stores the selected input value, a set of flip-flops stores the request signals that will have to be sent to the successors of the communication link. These flip-flops are reset when the corresponding acknowledgements have been received. Signal "libre" is reset when all expected acknowledgements have been received. Yet the effective output request signals stay low as long as "pret" is not active.

The chronogram of figure 5 gives a possible example of successive requests processing by a communication link (p, n), with 3 preceding resources and 3 successors, and with $r_n(p) = 1$, $R_n(p) =' 010'$. Block MULT_ADD is assumed to compute its result within 4 clock cycles (semi-parallel multiplier).

When all successors are free (i.e. are able to send immediate acknowledgements), a request processing requires only 5 clock cycles, which means that the specificity of FPNN computation only costs one clock cycle. The blocks required to handle this protocol only use 11 CLBs (configurable logic blocks) on a Xilinx XC4000 FPGA, whereas the affine transform requires at least 100 CLBs for 16-bit precision (even with an area-saving semi-parallel multiplier).

3.2 Neurons

The main changes with respect to a communication link are limited to the arithmetic operator. (cf figure 6). Operators ITERE and SORTIE (output) are used instead of MULT_ADD. The computation of SORTIE only occurs when a_n values have been processed by ITERE.

A linear feedback shift register is required for arity handling, so that the FPNN asynchronous protocol requires 12 CLBs for neurons (if $a_n \leq 15$) instead of 11 for a communication link.

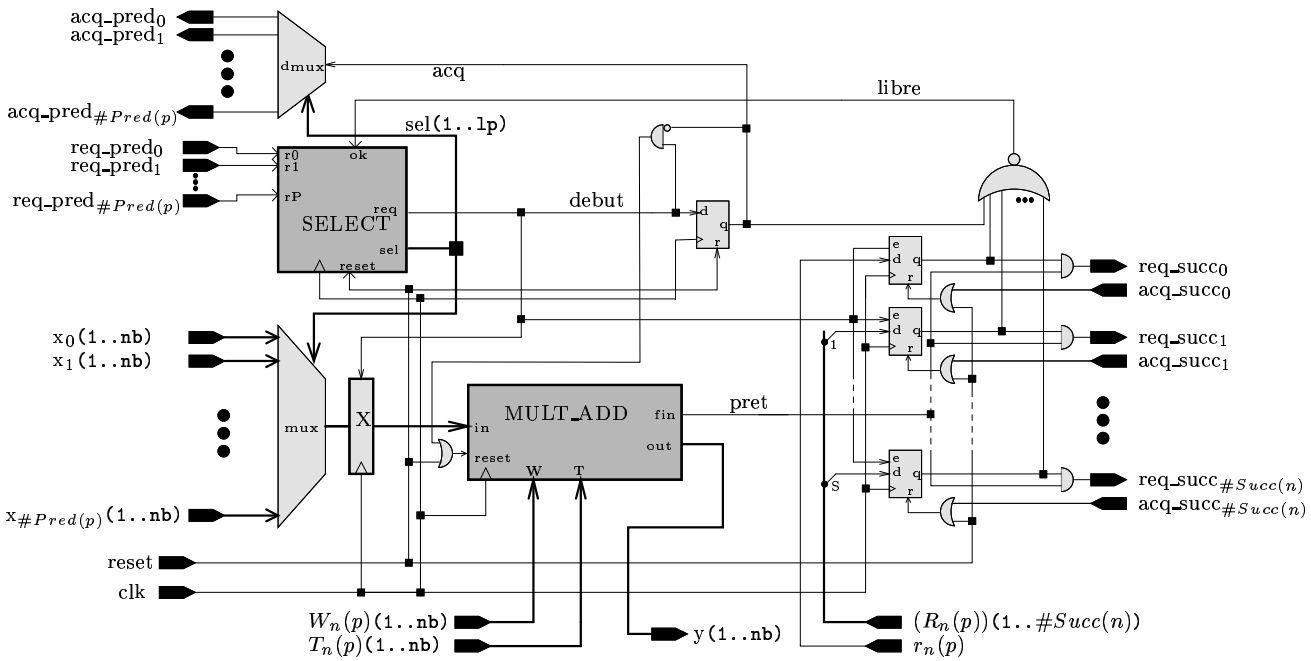

Figure 3: Communication link architecture

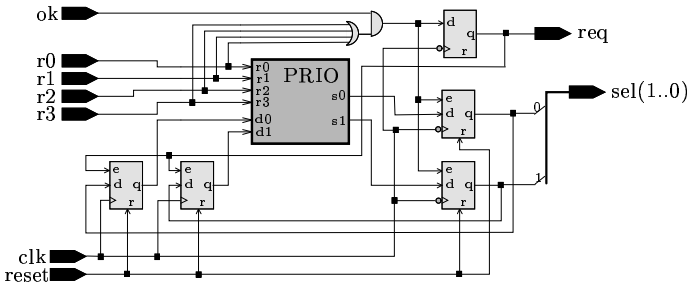

Figure 4: Request signal selection

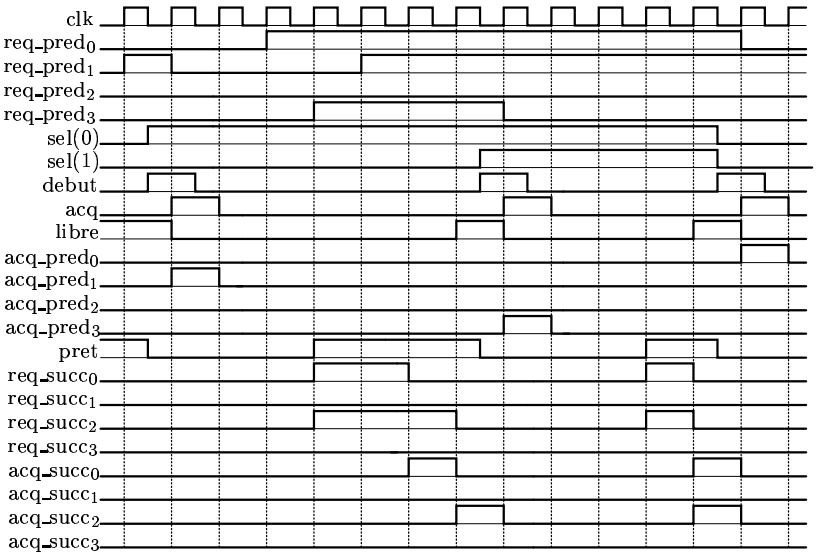

Figure 5: Chronogram of a communication link implementation

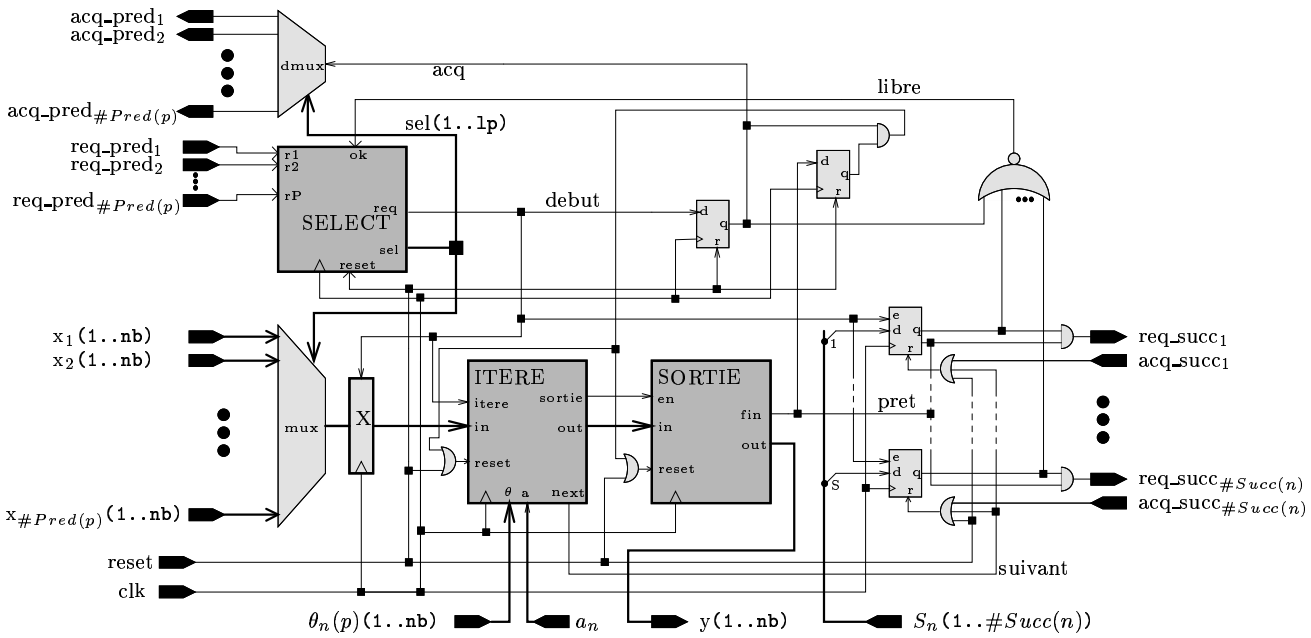

Figure 6: General architecture for a neuron resource

4 Conclusion

Neural network hardware implementations have to reconcile simple hardware topologies with often complex neural architectures. FPNAs have been defined for that. Their computation scheme creates numerous virtual neural connections by means of a limited set of communication links, whatever the device, the arithmetic, and the neural structure.

This report is a first introduction to FPNAs and FPNNs. The definitions and some computation schemes are given. FPNN topologies may be so simple that a modular description of their implementation is sufficient (there will not be any problem of dimension or fan-in and fan-out). Their concrete use has proved that they allow to have the computation power of standard neural models with a reduced set of neural resources easy to map directly only digital hardware. Next reports will focus on :

- less general FPNN implementations
- theoretical properties of the different computation schemes
- applications, performances

References

[1] S.L. Bade and B.L. Hutchings. FPGA-based stochastic neural networks - implementation. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 189–198, 1994.

[2] J.-L. Beuchat. Conception d'un neuroprocesseur reconfigurable proposant des algorithmes d'apprentissage et d'élagage: une première étude. In *Proc. NSI Neurosciences et Sciences de l'Ingénieur*, 1998.

[3] N.M. Botros and M. Abdul-Aziz. Hardware implementation of an artificial neural network. In *Proc. ICNN*, volume 3, pages 1252–1257, 1993.

[4] V.F. Cimpu. Hardware FPGA implementation of a neural network. In *Proc. Int. Conf. Technical Informatics*, volume 2, pages 57–68, 1996.

[5] J. Cloutier, E. Cosatto, S. Pigeon, F. Boyer, and P. Simard. VIP: an FPGA-based processor for image processing and neural networks. In *Proc. MicroNeuro*, pages 330–336, 1996.

[6] J.G. Eldredge and B.L. Hutchings. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. In *Proceedings of the IEEE World Conference on Computational Intelligence*, 1994.

[7] F. Elie. *Conception et réalisation d'un système utilisant des réseaux de neurones pour l'identification et la caractérisation, à bord de satellites, de signaux transitoires de type sifflement.* PhD thesis, LPCE, Université d'Orléans, 1997.

[8] A. Ferrucci. *ACME: a FPGA implementation of a self-adapting and scalable connectionist network.* PhD thesis, University of California, 1994.

[9] B. Girau. *Du parallélisme des modèles connexionnistes à leur implantation parallèle.* PhD thesis, ENS Lyon, 1999.

[10] B. Girau and A. Tisserand. On-line arithmetic based reprogrammable hardware implementation of multilayer perceptron back-propagation. In *Fifth international conference on Microelectronics for Neural Networks and Fuzzy Systems - MicroNeuro'96*, pages 168–175. IEEE Computer Society Press, 1996.

[11] V. Salapura, M. Gschwind, and O. Maisch berger. A fast FPGA implementation of a general purpose neuron. In *Proc. FPL*, 1994.

[12] K. Siu, V. Roychowdhury, and T. Kailath. Depth-size tradeoffs for neural computation. *IEEE Trans. on Computers*, 40(12):1402–1412, 1991.