FPGA Implementation of Numerical P Systems
Zeyi Shang, Sergey Verlan, Gexiang Zhang, Haina Rong

To cite this version:

HAL Id: hal-04032275
https://hal.science/hal-04032275
Submitted on 25 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright
FPGA Implementation of Numerical P Systems

ZEYI SHANG\textsuperscript{1,2}, SERGEY VERLAN\textsuperscript{2}, GEXIANG ZHANG\textsuperscript{1,3*}

\textsuperscript{1} School of Electrical Engineering, Southwest Jiaotong University, Chengdu 611756, China
\textsuperscript{2} Univ Paris Est Creteil, LACL, F-94010 Creteil, France
\textsuperscript{3} College of Information Science and Technology, Chengdu University of Technology, Chengdu 610059, China

Numerical P Systems (NPS) is a variant of P systems using real-valued quantities. It was shown that it can successfully be applied for different real-world problems, in particular in the area of robotic control. In this paper we introduce an extension of NPS, called Generalized Numerical P Systems (GNPS) and we describe an efficient implementation of GNPS using FPGA hardware. This allows to build fast controller chips based on (G)NPS and interacting directly with the environment. Two test cases are presented describing the implementation results of Sobel image edge detection algorithm.

Key words: Membrane Computing, numerical P system, enzymatic numerical P system, generalized numerical P system, Field Programmable Gate Array (FPGA), hardware implementation.

1 INTRODUCTION

Membrane computing (MC) is a natural computation paradigm abstracted from structural and functional features of living cells \cite{29,24}. Computing models in MC are called \textit{P Systems}. There is an intense theoretic research in the area of MC focusing primarily on computability aspects, in particular on Turing-computability, see \cite{13} for an overview. There is a huge number of

* Corresponding author: Gexiang Zhang, E-mail: zhgxdlan@126.com
different variants of P systems, we cite some important variants below: cell-like P systems [29, 16], tissue/population-like P systems [19, 18], spiking neural P systems [13, 10, 14] and P systems with active membranes [30, 12, 36].

Because of the biological background of P systems, they are used as modeling frameworks for biological and ecological systems, see [5, 18, 20, 23] for more details. As for engineering application of P systems, it is a relatively new direction in its early stage comparing to biological/ecological modeling of P systems. Complex market interactions are modeled by population dynamic P systems in [34]. Several variants of spiking neural P systems are used in power system fault diagnosis [37, 32, 27, 45, 33]. Other applications can be found in [46, 43, 44].

Numerical P systems introduced in [31] is a variant of P systems very different from the standard model. Instead of objects and rewriting rules it features real-valued variables which are updated in discrete time steps using a set of equations. This particularity, long time unexplored, is very interesting for applications in the area of control theory, as usually a control is described using a set of differential equations, which in many cases can be translated to the NPS equation set. In order to be applicable to practical case studies NPS model was extended to Enzymatic Numerical P systems (ENPS) [25] that allowed much more complex behaviors in the corresponding equations. This in turn lead to the concept of membrane controllers [2] that are ENPS systems designed to act as controllers and running in some environment. As a test bed the control of differential wheeled robots [2] (motion, obstacle avoidance, wall following, robot following) was developed. In [39] a kinematic controller and a proportional-integral-derivative controller are designed for wheeled mobile robots. Another interesting application is found in [38] where an environment classifier and a novel multi-behaviors control approach are proposed to enhance the reactive navigation performance of autonomous mobile robots.

Membrane controllers mentioned above require several ingredients. The evolution of a robot in a real or simulated environment requires a program that reads/transmits the values of robot’ sensors (usually distance and speed), runs the simulation of the controller (given in ENPS form) for one or several steps and then updates/transmits the values of actuators (usually robot wheel motors). Before running the simulation this program should assign initial values for the membrane controller and after the simulation it should retrieve corresponding output values from it. The controller itself is simulated using a custom simulator [2, 7] or by Matlab code [39, 38]. In some cases [7, 38]
the experiments were carried out in real robot environments. To speed-up
the simulation in the case of complex controllers [40] the use of graphical
processing units (GPU) hardware architectures was proposed [11].

In this article we extend NPS to a new model that we call Generalized Nu-
merical P systems (GNPS). The main idea behind this extension is to provide
a theoretical background allowing us to build custom parallel hardware archi-
tectures using Field-Programmable Gate Array (FPGA) technology. We studied
what operations can be efficiently performed in hardware and restricted
the GNPS architecture to be a series of rules that imply that the dynamics of
the system is described using equations written in Presburger arithmetic. This
allows in turn a very efficient translation to a hardware description language
(HDL) used for FPGA circuit design, allowing to run the model at the clock
speed (10^8 steps/s) and using a low number of resources. To assist this trans-
lation we developed a compiler that translates GNPS to Verilog HDL. This
allows to simplify the design process and to rapidly develop real hardware
prototypes. Moreover, it turns out that there is a tight link between GNPS,
sequential circuits [35], Mealy/Moore automata [21, 22] and synchronous
programming languages like Esterel and Lustre.

We considered several test cases and in this paper we present two of them,
descrining implementation results of Sobel image edge detection algorithm
on Diligent Basis 3 FPGA board, based on Xilinx Artix 7 architecture.

This paper is organized as follows: Section 2 introduces the definition of
NPS and ENPS, analyzing their relations with systems of difference equa-
tions, together with the definition of a normal form of (E)NPS. Section 3 ex-
tends the original definition of NPS to GNPS. Section 4 discusses the FPGA
implementation of two case studies in detail. Conclusions are drawn in Sec-
ction 5.

2 DEFINITIONS

2.1 Numerical P Systems

Numerical P systems were introduced in [31] as a model for the study of
economical processes. They have a tree-like structure and each compartment
contains a set of real-valued variables as well as evolution rules, called pro-
grams. They are formally defined as follows.

A numerical P system is the construct

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$
where \( m > 0 \) is the degree of the system, \( H \) is a set of labels, \( \mu \) is a membrane structure, \( \text{Var}_j, \text{Pr}_j \) and \( \text{Var}_j(0) \) are the set of variables, programs and initial values from compartment \( i, 1 \leq i \leq m \). By convention, we will label variables with two indices such that \( \text{Var}_j = \{x_{1j}, \ldots, x_{kj}\} \).

A program (rule) \( P_{li} \in \text{PR}_i \) has the following form

\[
P_{li} : F_{li}(x_{1i}, \ldots, x_{ki}) \rightarrow c_1v_1 + \cdots + c_nv_n
\]

where variables \( v_1, \ldots, v_n \) belong to membrane \( i \), or to the neighboring ones (the parent or the children of \( i \)).

The first part of the rule (function \( F \)) is called the production function, while the second part (at the right-hand-side of the arrow) is called the repartition protocol.

A rule is applied as follows \cite{31}. First the value of the production function is computed, based on current values of the variables. Second, each variable \( v_s, 1 \leq s \leq n_i \) from the repartition protocol part receives the fraction \( \frac{c_{ls}}{\sum_{r=1}^{n_k} c_{lr}} \) of the computed production function value. If several rules update the same variable, then the corresponding amounts are added. Finally, the value of a variable at the beginning of each new step is reset to 0 if this variable was used in a computation of some production function.

It is not very difficult to observe that a computation in a NPS corresponds to a discrete time series, where the value of a variable at some time step is a function of the values of several variables at the previous time step. More precisely the evolution of the system can be described by the following equations:

\[
x_{ji}(t) = \sum_{P_{ik} \text{ has } x_{ji} \text{ in rhs as } v_s} F_{ik}(x_{1k}(t), \ldots, x_{rk}(t)) \frac{c_{ls}}{\sum_{r=1}^{n_k} c_{lr}} + \bar{x}_{ji}(t) \quad (1)
\]

where \( \bar{x}_{ji}(t) = \begin{cases} x_{ji}(t) & \text{if } x_{ji} \text{ does not appear in any production function } F_{ik}, \\ 0 & \text{otherwise.} \end{cases} \)

**Example 2.1.** Consider the following NPS, also depicted in Figure 1, with two membranes nested as follows: \([1[2]2]1\). Let \( \text{Var}_1 = \{a, b, f\}, \text{Var}_2 = \{x, y\}, \text{Var}_1(0) = (0, 1, 3), \text{Var}_2(0) = (0, 1) \). The rules of the system are defined as follows:

\[
\begin{align*}
\text{Pr}_{11} : 4(a + b) & \rightarrow 1|a + 1|f + 2|x. \\
\text{Pr}_{12} : 3(x + y) & \rightarrow 1|b + 1|x + 1|y.
\end{align*}
\]
It is not difficult to observe that the corresponding system can be rewritten as following time series with initial conditions \(a(0) = 0, b(0) = 1, f(0) = 3, x(0) = 0, y(0) = 1\).

\[
\begin{aligned}
  a(t+1) &= a(t) + b(t) \\
  b(t+1) &= x(t) + y(t) \\
  f(t+1) &= f(t) + a(t) + b(t) \\
  x(t+1) &= x(t) + y(t) + 2(a(t) + b(t)) \\
  y(t+1) &= x(t) + y(t)
\end{aligned}
\]  

(2)

This can be also written in a matrix form as follows

\[
\begin{bmatrix}
  a \\
  b \\
  f \\
  x \\
  y
\end{bmatrix}
= \begin{bmatrix}
  1 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 1 \\
  1 & 1 & 1 & 0 & 0 \\
  2 & 2 & 0 & 1 & 1 \\
  0 & 0 & 0 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
  a \\
  b \\
  f \\
  x \\
  y
\end{bmatrix}
\]  

(3)

In many cases systems of recurrences can be solved analytically using standard methods. The analytical solution for system defined by Equation (2) is given below:

\[
\begin{aligned}
  a(t) &= 2 \times 3^{t-2}, \\
  b(t) &= 4 \times 3^{t-2}, \\
  f(t) &= 3^{t-1} + 3, \\
  x(t) &= 8 \times 3^{t-2}, \\
  y(t) &= 2 \times 3^{t-2} + 1, \quad t > 1
\end{aligned}
\]
We would like to notice that by definition it is allowed to have several rules in a membrane. In this case, at each step, one of them is chosen and applied non-deterministically. However, for practical considerations, most of the systems considered in the literature have only one rule per membrane.

**Enzymatic Numerical P systems**

Enzymatic numerical P systems (ENPS), introduced in [25] are an extension of NPS that adds a new type of rules (called enzymatic):

\[ P_{li} : F_{li}(x_{1i}, \ldots, x_{ki})(e \rightarrow) c_{1i}v_{1} + \cdots + c_{ni}v_{ni} \]

The application of this rule is conditioned to the verification of the minimality condition between the values of \( e \) and \( x_{1i}, \ldots, x_{ki} \). Unfortunately, there is no unique definition for this condition – several papers use different ones. Here is the list of most used conditions:

\[
\begin{align*}
\text{in [25]} & : e > \min(c(x_{1i}), \ldots, c(x_{ki})) \\
\text{in [25]} & : e > \min(x_{1i}, \ldots, x_{ki}) \\
\text{in [26]} & : e > \min(|x_{1i}|, \ldots, |x_{ki}|)
\end{align*}
\]

The first definition above uses the function \( c(x) \), which is the concentration of \( x \) in the biological sense. For example, for a production function \( 2x + y \) we obtain \( c(x) = x/2, c(y) = y \), see [25] for more details. As in the case of NPS, if there are several applicable rules, one of them is used non-deterministically. One can observe that ENPS also yield time series that use a conditional operator.

**Example 2.2.** Consider the following ENPS, also depicted in Figure 2, with three membranes nested as follows: \([1|2|3][3]|3\]. Let \( Var_1 = \{a, b\}, Var_2 = \{E\}, Var_3 = \{c\}, Var_1(0) = (0, 3), Var_2(0) = (2n), n > 0, Var_3(0) = 0 \). The set of rules is defined as follows:

\[
\begin{align*}
Pr_1 & : 2b + 1(E \rightarrow) 1|a, \\
Pr_2 & : E - 1 \rightarrow 1|E, \\
Pr_3 & : 2(c + 1)(E \rightarrow) 1|c + 1|b.
\end{align*}
\]

It is not difficult to observe that the corresponding system can be rewritten
as following time series.

\[
\begin{align*}
a(t + 1) &= \text{if } E(t) > b(t) \text{ then } 2b(t) + 1 \text{ else } a(t) \\
b(t + 1) &= c(t) + 1 \\
c(t + 1) &= c(t) + 1 \\
E(t + 1) &= E(t) - 1
\end{align*}
\]

Clearly, we obtain that \( a(t) = \sum_{i=0}^{t} 2i + 1 = t^2 \) (for \( t \leq n \)).

### 2.2 Binary normal forms for (E)NPS

Traditionally, NPS evolve in \( \min_1 \) mode (from each membrane a single program is selected and applied), see [9]. In [17] so called all-parallel mode is introduced for (E)NPS, when all applicable rules from a membrane are applied at the same step (we remark that contrary to traditional P systems there is no competition for symbols/variable values). The advantage of such a mode is that the computation becomes deterministic. Another advantage is that the system allows a very nice binary normal form.

**Definition 2.1.** A (E)NPS is said to be in the binary normal form if all rules
are of form

\[ P : F(x_1, \ldots, x_n) \rightarrow c|x_k + L|\lambda, \text{ or} \]

\[ P : F(x_1, \ldots, x_n)(E \rightarrow) c|x_k + L|\lambda, \]

for some \( n, k > 0; c, L \geq 0 \) and where \( \lambda \) is a special dummy variable.

It is relatively easy to see that any (E)NPS working in all-parallel mode can be transformed to an equivalent one in the binary normal form. To achieve this, any rule/program

\[ F(x_1, \ldots, x_{kj}) \rightarrow c_1|v_1 + \cdots + c_l|v_l \]

can be replaced by \( l \) rules that assign the same proportion of \( F \) to each variable \( x_l \)

\[ F(x_1, \ldots, x_{kj}) \rightarrow c_s|v_s + (K - c_s)|\lambda, \ 1 \leq s \leq l, \ K = \sum_{p=1}^{l} c_p. \]

We remark that when \( l = 1 \), then \( L = K - c_s = 0 \), so there is no variable \( \lambda \) introduced in the rule.

A similar transformation can be done with enzymatic programs.

Moreover, if we relax the condition that variables of the production function should be from the same membrane, then it is possible to obtain a stronger result. This allows to combine all rules related to a single variable into one rule by choosing appropriate coefficients. In this case several programs of type

\[ F_n(x_{1j}, \ldots, x_{kj}) \rightarrow c_n|x + z_n|\lambda \]

can be combined as

\[ \sum_{p=1}^{n} \frac{c_p}{c_p + z_p} F_p(x_{1j}, \ldots, x_{kj}) \rightarrow 1|x. \quad (4) \]

(E)NPS having only rules of the above type is said to be in unary normal form. It is easy to observe that Equation (4) corresponds to the following time series

\[ x(t + 1) = \sum_{p=1}^{n} \frac{c_p}{c_p + z_p} F_p(x_{1j}(t), \ldots, x_{kj}(t)) + \bar{x}(t). \quad (5) \]

which is exactly the Equation (1).
In this section we introduce a generalization of (E)NPS, called Generalized Numerical P System (GNPS), which has some interesting properties helpful for the hardware implementation of the model. As a starting point we take the notion of the membrane controller [2], hence from the beginning we assume to be interested not in the final result of the computation, but in the dynamics of the model. This naturally leads to the inclusion of the concept of dedicated input and output variables. The functioning of the system supposes that input variables are read-only and can be updated by an external entity at each step. The output variables are write-only and an external entity may use their values at each step. Such a definition allows to effectively build controllers based on GNPS, without using any additional tools or mechanisms to pass the values and start/stop the computation.

From the structural point of view we use a structural abstraction intermediate between a tree-based structure and a flattened system (more precisely a hypergraph structure), being the equivalent of the network of cells [9] in NPS. This allows to have the notion of the locality (useful for hardware implementation as it can trigger the use of neighbor cells), but does not impose the strong restriction of a tree structure — some examples of membrane controllers spend an enormous amount of time for the data propagation because of the imposed tree structure. Concretely, this allows production functions to contain variables defined in a different membrane and also the repartition protocol may involve variables from any combination of membranes.

The main difference of GNPS with respect to previous models is a new type of rules that generalize all previous ones. This comes from the observation that rules of ENPS are rather limited (and also have a poorly defined semantics). The ENPS simulator PeP [6] already proposed to use some simple arithmetic predicates to control the applicability of the rules. With GNPS we propose to go further and to use conditional rules of form (we separate by a semicolon local variables from the other ones):

\[
P(x_1, \ldots, x_k; E_1, \ldots, E_m); F(x_1, \ldots, x_k) \rightarrow c_1|v_1, \ldots, c_n|v_n, \quad (6)
\]

where \(P\) is predicate in Presburger arithmetic (we recall that this is the first-order theory of the natural numbers with addition, i.e. one can use comparisons, Boolean operations, additions, subtractions and constant multiplications in expressions). Moreover, in the basic variant of the definition we will restrict production functions \(F\) to be Presburger as well. However, in order to accommodate real-case scenarios we will allow the usage of a finite algebraic
signature (a set of functions that can be used in addition to the operations in
Presburger arithmetic) for both production functions and predicates.

**Example 3.1.** We can consider the following predicate for a rule

\[ P(x, y, z; E, F) = E > x \land (F > y \ast 2 + 3 \ast z). \]

If we consider an algebraic signature containing the ordinary multiplication
operation (\(\sigma = \{\times\}\)), then it would be possible to write the following predi-
cate

\[ P(x, y, z; E, F) = E > x \land ((F > y \ast 2 + 3 \ast z) \lor (E + F > x \times y + z)). \]

Finally, in order to obtain a deterministic evolution of the system, we as-
sume that GNPS works in all-parallel mode, i.e. all applicable rules are ap-
plied at each step. This allows to greatly simplify the design of the hardware
implementations.

Formally, we define a GNPS as the following tuple

\[ \Pi = (m, I, O, (Var_1, Var_1(0)), \ldots, (Var_m, Var_m(0)), Pr, \sigma), \]

where \(m, Var_1\) and \(Var_1(0)\) have the same meaning as in NPS (the number
of cells/membranes, the vectors of internal variables and their initial values).
The rules are no more specific to some membrane, so they are all collected in
the set \(Pr\). Each rule is of form (6). In the case of an always true predicate, it
can be omitted. Used variables in each rule induce a dependency hypergraph.
When this hypergraph is a tree, we may use a Venn diagram notation and place
rules in corresponding cells/membranes. The input (resp. output) variables
are given by the set \(I\) (resp. \(O\)). The algebraic signature \(\sigma\) contains the
list of additional functions used (with respect to the addition/subtraction and
constant multiplication). If \(\sigma = \emptyset\) then it may be omitted from the definition.

We recall that the evolution of the system is performed in all-parallel mode,
i.e. all applicable rules are applied in parallel at each step.

### 4 FPGA IMPLEMENTATION

According to the discussion in Section 2.2 and the fact that Presburger arith-
metic is recursive, any GNPS system can be rewritten as the following time
series (where \(X(t)\), \(Y(t)\) and \(Q(t)\) are the vectors of input, output and internal
variables, respectively, at time \(t\)):

\[ Q(t + 1) = F(Q(t), X(t)) \]  \hspace{1cm} (7)

\[ Y(t + 1) = G(Q(t), X(t)) \]  \hspace{1cm} (8)
These equations are the generalization (using real numbers instead of Boolean values) of equations used in switching algebra \[35\] for the definition of the concept of Mealy automaton \[21\], which together with Moore automaton \[22\] form the basis of modern synchronous circuit design. Since from the implementation point of view real numbers should be encoded using a fixed bit size, it appears that GNPS are very similar to vector Moore/Mealy machines. This in turn allows a straight implementation using hardware FPGA technology.

For implementation efficiency we considered following restrictions for GNPS:

- Real values are replaced by their approximation using a fixed-point binary representation.
- The production functions are linear.
- The predicates are Presburger-definable.

The above restrictions allow to relatively easy obtain the Mealy/Moore machine in form of Equation (7). The corresponding functions \(F\) and \(G\) are linear enriched with conditional statements as it is shown in Example 4.1.

Remark 4.1. In the basic case we consider an empty signature \(\sigma\) as this allows a straight translation to Verilog. For more complex computations, corresponding functions should be implemented additionally as Verilog modules. This can induce a delay, as in many cases it is not possible to compute corresponding functions in one clock step.

Remark 4.2. In the case of fixed-point encoding, it is possible to easily implement the multiplication operation working in one time step. This can be done either directly (by using multiplication code dependent on the width of the encoding), or using a special component of FPGA called DSP slice that allows to perform multiplication operations in one step (up to 48-bit width).

Remark 4.3. Contrary to multiplication, it is not easy to implement the division operation in one time step. However, the division by a constant \(c\) can be seen as the multiplication by \(c^{-1}\).

Example 4.1. Consider the system depicted on Fig. 3. It has two input, two output and two internal variables. The system computes the average value of its inputs and also indicates if this value changed by more than 0.1% on the previous step. We recall that all rules are executed in parallel.

The set of equations corresponding to this system is the following: \(a(0) = \)
It can be directly transformed to Verilog as follows (we assume a fixed point encoding of real numbers over 32 bits and using 12 bits for the fractional part).

In the below listing the fixed-point (constant) multiplication is performed by the function \_mult (recall that 2048 is 0.5 in the chosen fixed-point encoding).

```
module A #( parameter WIDTH = 32 , parameter BPPOS = 12)
( output [WIDTH: 0] out1 , output [WIDTH: 0] out2 ,
input [WIDTH: 0] in1 , input [WIDTH: 0] in2 ,
input clk );
    reg [WIDTH: 0] a = 0 ;
    reg [WIDTH: 0] b = 0 ;
    reg [WIDTH: 0] c = 0 ;
    reg [WIDTH: 0] d = 0 ;
always @(posedge clk) begin
    a <= _mult (in1+in2 , 2048); 
    b <= a ;
    out1 <= a-b ;
    out2 <= a-b < 4 && a-b >= -4 ? 0 : 4096 ;
end
endmodule
```

The synthesis of this circuit using Vivado tools uses 87 logic cells.

It can be seen that the translation is rather straightforward. A compiler `FPNtoVerilog` was developed in order to assist in this translation. As input it takes the GNPS model in form of Equations (7) and (5) and produces.
as output behavioral Verilog code implementing the corresponding Mealy-/Moore automaton.

The compiler performs the following steps:

1. Parse the input file.
2. Identify input and output symbols.
3. Flatten the obtained system.
4. Perform constant propagation.
5. Convert all constants to fixed-point real number representation.
6. Write Verilog output.

These steps are performed using standard compiling techniques. The last step is straightforward as a sequential Boolean (switching) function/circuit can be directly translated to Verilog. As a result a file containing the synthesizable (in FPGA) Verilog module whose code simulates each step of the GNPS at each clock tick is generated. Two case studies are designed as target models and their FPGA implementation process is detailed to elucidate how a GNPS can be implemented in a FPGA. Since the algorithm we implement is making use of the square root function, we consider that the signature of the system is $\sigma = \{\sqrt{\cdot}\}$.

Our target development board is Digilent BASYS 3 equipped with a Xilinx Artix-7 XC7A35T-1CPG236C FPGA as core component. The FPGA developing environment is Xilinx Vivado 2019.1 and Verilog is used as HDL. A Dell Latitude outfitted with an Intel Core i7-7820HQ and 16 GB RAM is the host computer.

### 4.1 Case study 1

The GNPS model (called GNPS1 for simplicity) of case study 1 is illustrated in Figure 4. It corresponds to Sobel image edge detection algorithm. GNPS1 only has a skin membrane, without any inner membranes. A program is applicable if its conditional rule can be met. Variable $e$ is assigned a big enough value so that the 4 programs can be executed at the same time. We employ fixed-point number format to represent real values. Specifically, every variable is assigned a 20-bit register in which the first bit designates the sign bit, the following 8-bit denoting integer part and the rest of 11-bit presenting fraction part of a real number. We also use the signature $\sigma = \{x^2, \sqrt{\cdot}\}$. 

13
FIGURE 4

 GNPS model for case study 1. It implements the core computations of Sobel image edge detection algorithm. GNPS1 has a skin membrane containing 4 programs and 19 real-value variables. The predicate for each program is taken to a separate line before it. All the programs execute in parallel if their conditional rules are satisfied. Input variables are \( a_1, a_2, a_3, a_4 \) (highlighted in red). Output variables are \( b_1, b_2, b_3, b_4 \) (highlighted in blue). Others are intermediate variables.
After inputting GNPS1 to our FPNtoVerilog compiler, the output is a behavioral model that specifies the behavior of GNPS1. We also provided the implementations of the functions from the signature σ (square root and the square function). Next, this model is translated to register transfer level (RTL) using Vivado tools. The upper-most level of GNPS1 schematic generated by Vivado is depicted in Figure 5. Before proceeding to the next processing step, RTL model should be evaluated to verify that it does possess required functions/behaviors. A test file named testbench is designed to perform such verification work. In the testbench, a model should be instantiated at first. Then input data/impulse to its input ports of the model and analyze outcomes represented in the form of timing waveforms. It is notable that outcomes are simulated by software (Vivado for our case, this simulation process is called Behavioral Simulation in Vivado.) instead of FPGA. In this research, two case study models are constructed as sequential circuits, namely clock is involved as a metronome to synchronize operations. Rising edge of clock is the trigger signal, i.e., operations can only be carried out after a rising edge and variables hold their values until next rising edge. The period of clock is set to 10 ns in the test bench.

Behavioral simulation omits any gate delays and data path delays, which means that results are output instantaneously, at the same time of trigger edge for sequential circuits [3] and the change instant of signals for combinational circuits. GNPS1’s behavioral simulation timing waveform is shown in Figure 6 from which we can see that b₁, b₃, b₄ get their outcomes after the first rising edge of clock. While the result of b₂ arises after the second rising edge, for the sum of two operands should be computed in the first period of clock and during that time the initial value for variable to be radicated is zero. Here we resort to corresponding Vivado IP core to compute the square root [42]. Apparently, behavioral simulation suggests that four programs of GNPS1 execute simultaneously and output variables emit results 20 ns later. As stated above, behavioral simulation neglects any delays so the waveform cannot reflect the real timing situation. Post implementation timing simulation which can only be conducted after implementation operation can provide more reliable timing waveform.

As the platform for design & validation of prototype circuits [4], the essential task of FPGA implementation is to obtain circuits. Synthesis procedure bridges the gap between RTL models and circuits. Models can be synthesized after behavioral simulation if it behaves as expected behaviors. There are two important tasks should be done after synthesizing target model: setting constraints and debug cores. Constraints include timing constraints and physical
FIGURE 5
Block diagram of GNPS1 RTL model. After the design of a model, the corresponding schematic can be drawn automatically by Vivado. The schematic characterizes the same functions/behaviors as RTL model representing by HDL.
FIGURE 6
Behavioral simulation of GNPS1. The first rising edge of clock is at 5 ns. \( b_2 \) gets its result at 15 ns and other three variables get their results at 5 ns since all the delays are neglected. This is not the reality but a simplification for assessing the behaviors of target model.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk 1</td>
<td>6.7061953125</td>
</tr>
<tr>
<td>a1</td>
<td>2.43013578125</td>
</tr>
<tr>
<td>a2</td>
<td>0.93994140625</td>
</tr>
<tr>
<td>a3</td>
<td>-3.0602421075</td>
</tr>
<tr>
<td>b1</td>
<td>14.1089765625</td>
</tr>
<tr>
<td>b2</td>
<td>0.0</td>
</tr>
<tr>
<td>b3</td>
<td>0.0</td>
</tr>
<tr>
<td>b4</td>
<td>3.9240040625</td>
</tr>
</tbody>
</table>

In timing constraints, the period of the clock and input/output delays are set, while physical constraints specify I/O configurations, mapping ports of model to pins of FPGA. The clock period is set 10 ns in the two case studies. To save pins, only \( b_3 \) and \( b_4 \) are set as output ports for GNPS1. All the constraints are written in constraint file (.xdc) in the format of industry standard Synopsys Design Constraints (SDC) [41]. The variables to be debugged in hardware debug procedure can be marked and set after synthesis.

The subsequent procedure of Synthesis is Implementation, which performs plan & route of the synthesized circuits and other vital operations such as power and hardware resource consumption analysis, real timing analysis. As mentioned above, behavior simulation ignores any delays so the timing situation cannot be evaluated from its waveform. Nevertheless, gate delays and data path delays of a model are taken into account after implementation so the timing of a design can be revealed by post implementation timing simulation, as shown in Figure 7. According to design timing summary provide by Vivado, the worst negative slack (WNS) is 3.97 ns, worst hold slack (WHS) 0.058 ns and worst pulse width slack (WPWS) 3.75 ns.

For the sake of comparing the computing speed of FPGA hardened GNPS and software simulation of GNPS, speedup is defined as the ratio of elapsed time of two methods. A software called PeP which dose not have a GUI dedi-
FIGURE 7
Post implementation timing simulation of GNPS1. Port $b_1$, $b_3$ and $b_4$ obtain their steady output value after the eleventh rising edge of clock, indicating it costs 110 ns to get results. For $b_1$, its steady output value emerges after sixteenth rising edge, costing 160 ns to compute outcome.

FIGURE 8
Software simulation of GNPS1. It is assumed that GNPS1 evolves one step to stop. Here we can see that there is no one-to-one correspondence between a clock cycle and a GNPS step. For complex arithmetic computations, one step of GNPS requires more than one clock cycle.

Emulates to emulate (E)NPS [6]. For the sake of simulating other types P systems, one can resort to P-Lingua [28]. GNPS1 is transformed to its ENPS counterpart and emulated by PeP, which outputs results and elapsed time to compute the results, as shown in Figure 8. Then the speedup of FPGA hardened GNPS1 is calculated in Equation (9). The maximum error of output variables is computed in Equation (10).

\[
\frac{5.651 \times 10^6}{160} = 3.5319 \times 10^4
\]  
(9)

\[
\left| \frac{3.92 - 3.92480469}{3.92} \right| \times 100\% \approx 1.225765 \times 10^{-3}
\]  
(10)

The estimated power consumption is reported after implementation, shown in Figure 9(a). Because the function of GNPS1 and GNPS2 is not complicate, the dynamic power merely shares 13% to 14% of total power and clock power makes up more than 70% of dynamic power.
(a) Power consumption of GNPS1 is 0.078 w.

(b) Power consumption of GNPS2 is 0.08 w.

FIGURE 9
Total power consumption is the sum of device static power and dynamic power. Power consumption of the two cases are nearly the same, although GNPS1 works in all parallel and GNPS2 works in sequential.

After implementation of a model, its Place & Route planning is also performed. Vivado provides powerful view check feature so the place and route can be zoomed in to see each path clearly. Hardware resource cost for GNPS1 is listed in Table 1.

If design objectives are satisfied after implementation, one can proceed to generate bitstream which contains design specifications and program device that download bitstream to FPGA to carry out physical plan & route. The real computing results of FPGA cannot be observed straightforwardly, but requires a particular procedure called hardware debug. The variables to be debugged should be marked in Verilog codes or marked in the net list. We mark $b_3$ and $b_4$ in Verilog codes as debug signals. After programing device, the integrated logic analyzer window open automatically. Debug signals should be added into the window manually, then run debug to exhibit values computed by FPGA, shown in Figure 10.

4.2 Case study 2
In practical applications such as image processing and robot path planning, computation process comprises several sequential procedures. In each procedure, multiple functions may be performed in all parallel mode, like the way GPNS1 works. In this subsection, we modify GNPS1 from all parallel
<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice</td>
<td>680</td>
<td>8150</td>
<td>8.34</td>
</tr>
<tr>
<td>LUT</td>
<td>1309</td>
<td>20800</td>
<td>6.29</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>159</td>
<td>9600</td>
<td>1.66</td>
</tr>
<tr>
<td>FF</td>
<td>2126</td>
<td>41600</td>
<td>5.11</td>
</tr>
<tr>
<td>BRAM</td>
<td>1.5</td>
<td>50</td>
<td>3.00</td>
</tr>
<tr>
<td>IO</td>
<td>41</td>
<td>106</td>
<td>38.68</td>
</tr>
<tr>
<td>BUFG</td>
<td>2</td>
<td>32</td>
<td>6.25</td>
</tr>
</tbody>
</table>

**TABLE 1**

Hardware resources utilization of GNPS1

**FIGURE 10**

Hardware debug of GNPS1. Input variables cannot be debugged so there is no clock signal. Values are represented in hexadecimal, 01f66 is 8038 in decimal. \(8038 \div 2^{11} = 3.9248046875\), which is the value of \(b_4\). 005eb is 1515 in decimal, \(1515 \div 2^{11} = 0.73974609375\), which is the value of \(b_3\).
to sequential mode, resulting in GNPS2, depicted in Figure 11. Conditional rule of membrane 1 is met at beginning, so programs $Pr_{11}$ and $Pr_{21}$ take place simultaneously. Other conditional rules are not met for the initial value of conditional variables are zeros. After $Pr_{21}$ modifying the value of $e_2$ from 0 to 3, programs in membrane 2 are triggered to execute. So does membrane 3 and 4. It is worth to note that $Pr_{21}$ and $Pr_{22}$ consume $e_1$ and $e_2$ so each program can only execute once. In short, a train-type ignition method is used to control the execution sequence of programs.

Input GNPS2 to FPNtoVerilog obtaining the RTL model. The block diagram of GNPS2 RTL model is illustrated in Figure 12. Edge detection is used as the trigger signal to impel the next membrane to execute. By this way, the train-type ignition is realized. According to Figure 7, reflecting actual timing condition of each program, program $Pr_{11}$, $Pr_{13}$, $Pr_{14}$ cost 11 clock cycles to compute while $Pr_{12}$ needs 16 clock cycles. The testbench designed for GNPS2 conforms to these delays and the behavioral simulation is shown in Figure 13 in which variable $cout1$ is the edge detection signal that can detect the rising edge of $cont$. Be ware that $b_i$ ($i = 1, 2, 3, 4$) gets value instantly for the neglect of delays in behavioral simulation.

Synthesize GNPS2 and open the synthesized model, complete constraints design and debug core set as that of GNPS1, then implement GNPS2. Run post implementation timing simulation to check the timing situation, as shown in Figure 14. PeP simulation of GNPS2 shown in Figure 15, so the speedup of FPGA implementation is computed in Equation 11. Power consumption of GPNS2 is given in Figure 9(b).

$$\frac{9.306 \times 10^6}{480} \approx 1.9388 \times 10^4$$ (11)

Hardware consumption for GNPS2 is summarized in Table 2. It can be seen that GNPS2 utilizes a little bit more resources than GNPS1 for its more complicate logic. At last, perform hardware debug to verify that FPGA hardened GNPS2 obtained correct outcomes, shown in Figure 16.

Finally, the last tests were performed using an autonomous execution of the system without output and using distributed read-only memory data storage for the input. Under this setup the speed of 100Mhz (corresponding to the system clock) was achieved. This means that a GNPS model can be simulated at a speed of $10^8$ steps per second. We would like to remark that in real-use cases the reaction speed will be dependent on the input/output delay. It is pointed out that the input/output circuits are not system-specific and can be reused for different simulations. However, at the present state they
FIGURE 11
GNPS model for case study 2 is numbered as GNPS2. The equations inside are the core computations of Sobel image edge detection algorithm. GNPS2 has 5 membranes and evolves 4 steps to reach halt condition. Programs in each membrane compute concurrently while each membrane execute serially.
FIGURE 12
Block diagram of GNPS2 RTL model. Each membrane is modeled in Verilog basic functional unit, module. The bug icons indicate variables to be debugged in Hardware Debug procedure.

FIGURE 13
Behavioral simulation of GNPS2. At a rising edge of clock, if $cout_i = 1$, execute programs in the next membrane. Three edge detection signal appears at 105 ns, 265 ns and 375 ns respectively and lasts 10 ns.
(a) Variable $b_1$ gets its value in the 11th cycle, while $b_2$ obtains its value in the 25th cycle.

(b) Variable $b_3$ gets its value in the 38th cycle, while $b_4$ obtains its value in the 49th cycle.

FIGURE 14
Post implementation timing simulation of GNPS2. The real timing of $b_2$ and $b_3$ is a little different than expected.

FIGURE 15
Software simulation of GNPS2. It is assumed that GNPS1 evolves four steps to stop. CPU of the host computer costs 0.009306 s to get results.


<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice</td>
<td>744</td>
<td>8150</td>
<td>9.13</td>
</tr>
<tr>
<td>LUT</td>
<td>1327</td>
<td>20800</td>
<td>6.38</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>156</td>
<td>9600</td>
<td>1.63</td>
</tr>
<tr>
<td>FF</td>
<td>2150</td>
<td>41600</td>
<td>5.17</td>
</tr>
<tr>
<td>BRAM</td>
<td>1.50</td>
<td>50</td>
<td>3.00</td>
</tr>
<tr>
<td>IO</td>
<td>41</td>
<td>106</td>
<td>38.68</td>
</tr>
<tr>
<td>BUFG</td>
<td>2</td>
<td>32</td>
<td>6.25</td>
</tr>
</tbody>
</table>

TABLE 2
Hardware resources utilization of GNPS2

FIGURE 16
Hardware debug of GNPS2. Values are represented in hexadecimal, 070cd is 28877 in decimal. $28877 \div 2^{11} = 14.10009765625$, which is the value of $b_1$. 17c5 is 6085 in decimal, $6085 \div 2^{11} = 2.97119140625$, which is the value of $b_2$. 

(a) Hardware debug of $b_1$ (connect to $b_{11}$) and $b_2$ (connect to $b_{12}$).

(b) Hardware debug of $b_3$ and $b_4$. 
need to be integrated manually in the final hardware design. At this moment, the development of FPNtoVerilog continues in order to integrate the automatic generation of input/output modules. This will allow a generation of a hardware circuit directly from the GNPS specification, without any user intervention.

5 CONCLUSIONS

In this work we provide an efficient custom FPGA-based hardware architecture design for the implementation of generalized numerical P systems. This allows a high-speed simulation of the corresponding system as well as a direct on-chip handling of input and output data. This architecture has a solid theoretical basis based on GNPS systems and consumes a low number of resources. However, at the moment a manual intervention during the design process is still necessary and we concentrate our future effort on the complete authorization of the design. In perspective, this will allow to create custom chips performing a control function and handling their input/output in an automated manner, without having any hardware programming knowledge.

Another important point is to test the obtained chips by using them to directly perform the robot control. Input/output from serial ports, leds and switches on the BASYS 3 board were tested to build data transmission channels. We have made several preliminary investigations showing the possibility to acquire sensor data and to send control signals to the Pioneer 3 DX wheeled robot from the GNPS controller. A further development would need to design the corresponding interface and accompanying functions in the compiler. We have also shown that the underlying model is extremely close to well known circuit design abstractions based on Mealy/Moore automata. This can give a new research direction investigating the links between GNPS and synchronous programming languages.

The method for FPGA implementation of GNPS is proposed and verified. Results show that it is feasible to implement GNPS on FPGA to exploit their parallelism to speedup computations. Comparing to software simulation of GNPS, the speedup achieved is an order of $10^4$. Obviously, the hardware architecture of GNPS is a parallel one. However, the fact that programs and membranes worked as processing units to handle tasks inspires us that a FPGA hardened GNPS seems to be a heterogeneous multicore processor for different programs and membranes working in parallel have different functions. As a consequence, this heterogeneous architecture defined by P systems differing from current architectures can be applied in computation intensive...
fields such as image/video processing, robot path planning, big data, etc.

**ACKNOWLEDGMENTS**

This work is supported by the National Natural Science Foundation of China (61972324, 61672437, 61702428), by Beijing Advanced Innovation Center for Intelligent Robots and Systems (2019IRS14), Artificial Intelligence Key Laboratory of Sichuan Province (2019RYJ06) and the Sichuan Science and Technology Program (2018GZDZX0043, 2018GZ0185, 2018GZ0086).

**REFERENCES**


