Modular Petri Net Processor for Embedded Systems

Orlando Micolini[1], Emiliano N. Daniele, Luis O. Ventre

Laboratorio de Arquitectura de Computadoras (LAC) FCEFyN
Universidad Nacional de Córdoba

orlando.micolini@unc.edu.ar, endaniele@gmail.com,
luis.ventre@unc.edu.ar

Abstract. Reactive and concurrent embedded systems execute restricted algorithms depending on the requirements. It is possible to implement one of these hardware-software systems by using a Petri Net Processor. If logic and policy are decoupled from the system actions, then we can improve maintainability and system validation. To achieve this, the Petri Processor is integrated with other traditional processors, forming a heterogeneous multi-core processor, which allows to verify the system using Petri Net mathematical formalisms. In this article, a Modular Petri Processor Architecture is exposed, as well as the inclusion of programmable queues that enhance maintainability, module reusage and semantic extension.

Keywords: Petri Processor, Petri Net, FPGA, IP Core, Heterogeneous multi-core processor

1 Introduction

Heterogeneous multi-core processors include specific capabilities that are not available in homogeneous multi-core processors. Reactive and concurrent embedded systems need to comply with specific non-functional requirements[1]. These systems interact with their environment, from which they receive input events and to where they generate output reactions. The environment is the one imposing the rate at which the system needs to generate the reactions. During this interaction, the system should react as quickly as possible to satisfy the timing restrictions. This response time depends not only on the algorithm used, but also on the platform capabilities, which make them essential for estimating the overall response time of the final system [2].

The system proposed in this article is a heterogeneous system, with a Petri Processor (PP) and a General-Purpose Processor (GPP). The PP receives events, process them to calculate the next system state, while the GPP calculates and executes actions, thus decoupling logic from the actions of the system [3].

The proposed architecture implements an innovative way to relate hardware and system logic by using a synchronization monitor. Because of this, the software pre-

[1] Corresponding author

scind from taking care of the critical sections of the code, and the synchronization functions, since the PP is the one performing those tasks. Furthermore, there is a univocal relationship between the program that the PP executes and the Petri Net model used, which guarantees the same properties that the model verifies.

The requirements for Concurrent and Reactive Embedded Systems [4] are: precision, reliability and structural flexibility. To reach those requirements, the PP is programmed using the model directly, resolving the execution of the events, and the state of the system. The PP, then, processes and orders events based on the restrictions of the system.

Applications of this processor are not only tied to reactive embedded systems, since it can be used to solve parallel systems. Many problems have been solved by using Petri Nets [5, 6].

The architecture of previously implemented PP was monolithic [3, 4]. In the current article we propose and develop a modular PP that preserves the same advantages of the previous PP, while adding new features to it. Certain situations were taken into consideration, like the use of naming conventions when implementing internal circuitry as well as external interfaces, which helped to standardize the components.

1.1 Objectives

General Objective

Modularize each PP function to optimize system maintainability and add programmable event queues.

The first implementation is aimed at generating reusable and standardized hardware components. This allows to add, remove or replace components in an easy way, so the PP can be adjusted to fit small embedded systems by only instantiating the necessary modules for a given application.

The ability to program the event queues allows the PP to execute different non-autonomous Petri nets, which increases the semantic capabilities of the system.

2 Petri Nets

2.1 Ordinary Petri Nets (PN)

An Ordinary Petri Net (PN) [7] is a quintuple defined by PN = (P, T, I-, I+, M_0) where:

- $P = \{p_1, p_2, \dots, p_n\}$ is a finite non-empty set of places.
- $T = \{t_1, t_2, \dots, t_m\}$ is a finite non-empty set of transitions.
- Given that P and T form a bipartite graph, the following is true: $P \cap T = \{\emptyset\},\ P \cup T \neq \{\emptyset\}$
- I^-, I^+ are the incidence relationships between the places inputs and outputs, which relate Places and Transitions (I^-) or relate Transitions and Places (I^+). The Incidence Matrix is defined as: $I = I^+ - I^-$

- $M_0 = [m_0(p_1),\ m_0(p_2) \ldots, m_0(p_n)]$ is the initial marking vector of the PN, which represents the number of Tokens that each Place contains.

Taking the Incidence Matrix definition into account, a PN is then defined as a quadruple: PN = (P, T, I, M_0).

2.2 Synchronized Petri Nets or Non-autonomous Petri Nets.

This type of PN introduce events modelling to the equation and they are an enhancement of the Ordinary PN [3, 8].

Non-autonomous PNs are used for modelling systems where the external discrete events synchronize the firing of the transitions. These events are tied to the transitions. In Fig. 1 the transition is synchronized with the event E^3, and the firing is produced when the following requirements are met:

- The transition is enabled
- The associated event occurs

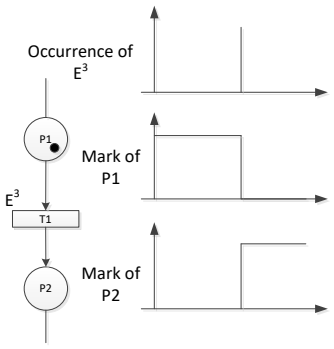

Fig. 1. Transition associated to an event.

Changes outside of the system trigger the external events (this includes changes in time) while internal events are changes within the system itself. Synchronized Petri Nets can be then defined as a triple:

PNsync = (PN, E, sync) where:

PN is a marked PN, *E* is a set of external events and *sync* is the function that relates Transitions T with $E \cup \{e\}$, where $\{e\}$ is the null event. All firings are atomic and instantaneous.

2.3 Perennial and Non-perennial events.

There are three types of events that can be associated with a Non-autonomous PN: perennial events, non-perennial events and automatic or null events [3].

Perennial events are those that, when they are triggered, they stay requesting a firing until the firing and synchronization conditions are met (the result will be the firing of the associated transition). These events are kept in the input queue until the associated Transition is fired.

Non-perennial events will fire a Transition, only if the transition was enabled before the event arrives; if the transition is not enabled, the event is discarded. In the PP implementation, there is a period of tolerance of two clock cycles for the Transitions to be enabled until the event is discarded.

Null or **Automatic** events {e} are associated with the automatic Transitions. These special types of Transitions are described in the next section.

2.4 Automatic Transitions

There is a type of internal event that is always generated and available, named $\{e\}$. In a Synchronized PN, one or more transitions can be associated with the Automatic Event {e}. This means that those Transitions can be fired automatically as soon as they are enabled, since the automatic event is "always happening".

2.5 Conflicts among transitions

We can find conflicts among transitions in a synchronized PN when:

- Two or more transitions are enabled
- Their associated events occur simultaneously
- If one of the enabled transitions is fired, then some of the other transitions become disabled

Mathematically, the conflicts are defined as:

$$|\{t_s\}| > 1 \wedge |\{E^s\}| > 1 \text{ y } t_i, t_j \in \{t_s\}$$

Where:

- $\{t_s\}$ is a set of enabled transitions
- $\{E^s\}$ is a set of events associated to the transitions $\{t_s\}$
- $t_i \; y \; t_j$ are transitions in conflict. They share at least an entry point with another place. If one of the transitions is fired, the other transition will be disabled. All these conflicts can be solved by a priority policy [6].

2.6 Relationship between events and the Incidence Matrix

If the semantics of a transition fire in a non-autonomous PN are analyzed, we find that the Incidence Matrix is the conjunctive evaluation of the columns (transitions) and the rows (places). This means that, if we have an Incidence Matrix of $m \times n$ dimension, n combinations of m logic variables are evaluated, so if we consider that every transition has an event associated, then we could write an expression like the following:

$$s_i = \left(\bigwedge_{h=0}^{m-1} (M(p_h) \geq i_{hi}) \; and \; E^i \right),$$

Where i_{hi} are the elements of the Matrix I and $M(p_h)$ is the marking value of the place h. The result s_i holds the elements of the vector of enabled transitions.

The PP executes this equation and is the foundation of its direct programmability since we can consider the Matrix and the Events, the equivalent of the program. This architecture executes an extended non-autonomous PN, and taking its semantic capability into account, gives us a Touring Machine [9, 10].

3 Architecture of the PP

The PP developed and presented in this article, reveals changes in the architecture implemented in [11]. The big difference resides in the programmable queues, the modularization of the hardware and the specific optimizations of each module.

The main building blocks of the PP are: the core, the queues, the priority module and the interfaces to connect to external devices.

3.1 Core

The main responsibility of the core module is to keep the internal state of the PN that the processor executes. Fig. 2 displays the overall architecture of the processor, where the modules that form the core are marked with (*).

The core is composed of the following components: the Incidence Matrix, the Marking Vector and the Fire Requests Vector.

The responsibilities of each component are:

- **Incidence Matrix I**: stores the Incidence Matrix of the PN. Its dimension is equal to |T|x|P|. All values stored are integers.
- **Marking Vector**: Stores the vector M of the PN, that is, the state of the PN. All stored values are positive integers.

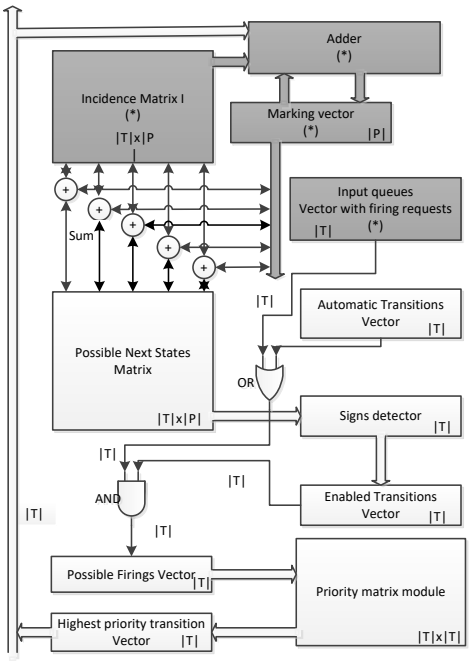

Fig. 2. PP's architecture

- **Input queues**: their interfaces expose a vector to the PP, where each position represents the connection with each transition. Each position of the vector can be equal to 1 if there are events in the correspondent queue, or equal to 0 if there are no events queued.
- **Automatic Transitions Vector:** each position of this vector represents each transition. If a transition is configured as automatic, then its correspondent position in this vector will be equal to 1.
- **Possible Next States Matrix**: corresponds to the sum of each column of the Matrix I with the Marking Vector. Each i-th column of this matrix has the state that would be reached if the i-th transition were triggered.
- **Signs detector**: it is a vector where each position corresponds to a column of the Possible Next States Matrix. If any of the values in a column is negative, then the position in its vector will be equal to 1, meaning that the firing of this column would reach an unreachable marking (negative values mean that we would try to fire a disabled transition).
- **Enabled Transitions Vector**: it is the negated output of the signs vector.

- **Possible Firings Vector**: stores the possible firings of the PN, which is calculated from the Enabled Transitions Vector and the Requested Firings Vector (Input queues *or* Automatic transitions).

3.2 Processor Operation

As well as the previous PP [3], this new version of the PP also executes a PN in two clock cycles. Single server semantics has been adopted for this implementation.
Cycle 1 – Calculations. In this cycle, the PP calculates the transition that will be fired. To achieve this, the marking vector is added to every column in the incidence matrix, so we can obtain the signs of the possible next state that would be reached with every possible firing. This is stored in a new matrix where the column i holds the signs of the next markings.
The output of the Possible Next States Signs Matrix is the input of the Sign detector module. This module performs an '*or*' operation with all values stored in each column. If any of those values are negative, it means that the next state will not be reachable (the correspondent transition is not enabled).
The enabled transitions vector stores a value of 1 in those positions where the columns of the Possible Next States Matrix did not have negative values.
To calculate the transitions that can be fired, the information of two other vectors is needed: The requested firing vector (the output of the input queues) and the automatic transitions vector. The first one represents which transitions were requested to be fired by sending specific instructions to the processor (external event). The second one represents those transitions that do not require explicit firing events, because they are associated with the null event {e}.
Finally, to determine which transition the processor will fire, the Possible Firings Vector is inputted to the Priority Module. The output of this module is the id of the enabled and requested transition with the highest priority. The vector will hold a single value of 1 in the position that corresponds to the selected transition. This transition will be fired in the next cycle.
Cycle 2 – Update. In this cycle the firing is performed and the marking vector is updated. To achieve this, the selected transition vector works as a column selector. The adder (which can be found at the top of Fig. 2) carries on the sum of the marking vector with the selected column of the Matrix I. The result of the sum is stored back into the marking vector.
Besides, if applicable, the queues are updated; the correspondent output queue counter is incremented and the correspondent input queue counter is decremented.

3.3 Queues

Both the input and output queues of the PP have been redesigned to make them configurable.

There is one instance of an input queue and one instance of an output queue for every transition that the PP has. Each of those instances contains a saturated counter

(only positive integers). Each counter is equipped with a max detector (for the overflow signal) and a zero detector (this is used for marking the queue as empty).

Input queue

The input queue stores the firing requests for the configured PN transitions. In Fig. 3 (a), the external interfaces are shown. There are three modes of operation.

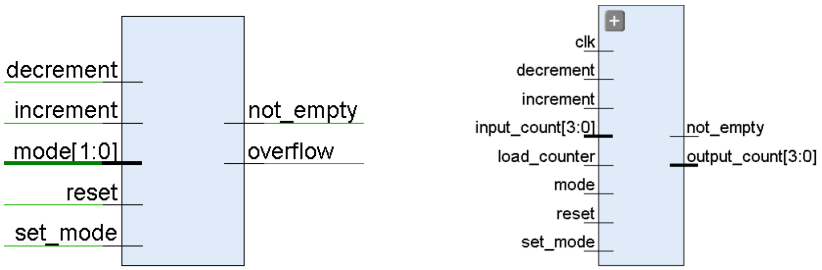

Fig. 3. (a) Input queue, (b) Output queue

Mode 1 – Normal Mode (Perennial event, default mode): The binary counter is incremented whenever a new firing request is issued (this is achieved via a specific instruction to the PP), and it is decremented on each fire performed on cycle 2. Since this is counting perennial events, the counter will keep its count until the increment or decrement signals of the module modifies it. In this case, the events queued are kept until the associated transition is fired. This is the only mode previous PP queues had.

Mode 2 – Automatic transition: In this operation mode the signal **not_empty** is kept high, see Fig. 3 (a). This can be interpreted as a transition that is always requested to be fired, that is, an automatic transition.

Mode 3 – Non-perennial event: All events that are counted are non-perennial. After a certain amount of time (two clock cycles of the PP), the counter will be reset, and thus the associated transition could not be fired after that time.

Output queue

As seen on Fig. 3 (b), output queues store the amount of firings performed. If a transition is fired, the output queue associated with it will increment its counter. When the queue is read (with a specific instruction) the counter will be decremented. The output queues have two different operation modes.

Mode 1 – Reporting mode: In this mode, the internal counter is incremented when the associated transition is fired and decremented when the counter is read (to check if a transition was fired). If the interruptions are enabled in the GPP, an interruption will be generated when the counter is different than zero.

Mode 2 – Non reporting mode: In this mode, the internal counter is not used and the signal **not_empty** is always equal to zero.

Priorities and conflicts among transitions

The PP cannot detect a conflicting state, so it treats transitions as if they were all in conflict. The Possible firing vector of the PP represents all those transitions.

The PP fires only one transition per execution cycle (single server model). This solves the conflicts issue and makes the system deterministic; the priority module makes the decision about which transition to fire. This is implemented as a binary matrix that establishes the necessary relationships to determine the highest priority transition. This module is also configurable during execution time.

Microblaze MCS Interface

Given that the PP is associated with a GPP [3], it is necessary to connect both processors in order for the system to perform the required system actions. The GPP that was chosen for this integration is the softcore Microblaze MCS [12], because of its low impact on the system resources and because it is well supported by the development tools from Xilinx.

The software architecture is very simple and it is made of one main program and two drivers. The first driver exposes a communication interface that we use for connecting to the PP (**pp_driver**) and the second one (**external_comm**) initializes and controls the UART module to send instructions and info through a serial connection.

4 Results

Several application cases were executed to evaluate the PP performance. It has been used successfully for controlling production lines, like the ones presented by Naiqi Wu y MengChu Zhou in [13]. The comparisons have been conducted by taking into consideration the results obtained in [3], which were very similar.

4.1 FPGA resources

To determine the amount of resources used and to compare them with other implementations, a PP was instantiated multiple times, varying the number of elements of the vectors and matrices. Multiple synthesis of the core were performed as well, and data lengths of 4 and 8 bits were used.

Fig. 4 shows the amount of FPGA resources used for each synthesized configuration. Registers are related to the number of flip-flops consumed, while LUTs, are directly related to the hardware architecture of the FPGA Atlys, which was the platform we used for implementing the PP.

In Fig. 4 an exponential use of resources is shown a long as we increase the number of elements in the vectors and matrices.

A very important discovery was that 4 bit configurations use approximately the same amount of registers than 8 bit configurations, even when an 8 bit configuration has fewer elements in the vectors and matrices (and in some cases, the number of registers decrease). For example, configuration 32x32x4 consumes fewer resources than configuration 24x24x8.

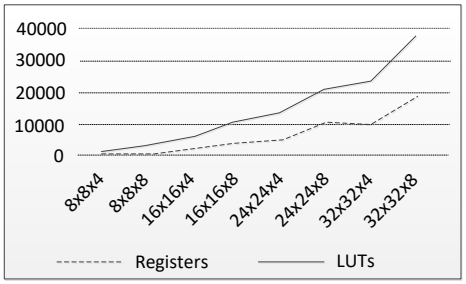

Fig. 4. Graph of resource consumption

This means that we can process bigger PNs, if we can restrict the data length for a given application.
Due to the optimization of the IP Core, the resources impact of using Microblaze, is considered constant (12.72% of all LUTs available and 21.09% of registers). These resources are the same for all the PP architectures.

4.2 Frequency analysis

To perform a frequency analysis, we followed the same approach as explained in the previous section. This shows the maximum frequency that each implementation could reach. The data length has a negligible effect in the max frequency reached. For example, if we compare configurations 8x8x4 and 8x8x8, the difference in frequency is just 2 MHz (80.073 MHz and 78.422 MHz respectively). However, if we increase the number of elements of the matrices and vectors, that is, if we increase the size of the PN, then the frequency is significantly lower. For example, between configurations 8x8x8 and 16x16x8 there is approximately a 23 MHz difference (78.422 MHz and 55.32 MHz respectively). The boundaries of frequency are mostly due to the model of FPGA used.

5 Conclusions

In the current project, the PP was extended, modularized so it is an enhanced version of the one presented in [3]. An interconnected system was achieved, in replace of the monolithic version of previous implementations. The module design and the inclusion of programmable queues did not impact negatively on the resources required.
The programmability of the queues, which now support different modes of operation and types of events, is a huge step forward. Besides, the maintainability of the PP has been simplified, so it is easier to add new features in the future, like the possibility of executing Hierarchical PNs, temporal PNs, etc. The control stage of the processor is the only module that needs to be modified for that.

The results obtained from the optimizations implemented in the hardware level show that the PP is suitable for Embedded Systems that require up to 32 conditions (transitions), 32 logic variables (places) and 32 simultaneous events.
The addition of a serial communication module allowed to have a testable, programmable and configurable PP. All these can be now performed from any PC terminal.
The modular implementation of the PP implies a step forward for maintainability, scalability and the future auto-configuration of the PP. Furthermore, the programmable queues have increased its semantic capability.

References

1. Munir, A., A. Gordon-Ross, and S. Ranka, *Modeling and Optimization of Parallel and Distributed Embedded Systems*, ed. W.-I. Press2016.
2. Gamatié, A., *Designing embedded systems with the Signal programming language: synchronous, reactive specification*2009: Springer Science & Business Media.
3. Micolini, O., *PhD thesis Arquitectura asimétrica multicore con procesador de Petri*, 2015, Facultad de Informática: La Plata, Argentina.
4. Bainomugisha, E., et al., *A survey on reactive programming.* ACM Computing Surveys (CSUR), 2013. **45**(4): p. 52.
5. Moutinho, F. and L. Gomes, *Distributed Embedded Controller Development with Petri Nets: Application to Globally-Asynchronous Locally-Synchronous Systems*. Vol. 150. 2015: pp.43-67 Springer.
6. Haustermann, M. *Applications of Petri Nets*. 2017 [cited 2017; Available from: https://www.informatik.uni-hamburg.de/TGI/PetriNets/applications/.
7. Diaz, M., *Petri Nets Fundamental Models, Verification and Applications*2009, NJ USA: John Wiley & Sons, Inc.
8. David, R. and H. Alla, *Discrete, continuous, and hybrid Petri nets*2010, Springer Science & Business Media.
9. Hopcroft, J., R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation* Prentice Hall, 2006.
10. Popova-Zeugmann, L., *Time and Petri Nets.* Springer, 2013.
11. Micolini, O., et al. *Procesador de Petri para la Sincronización de Sistemas Multi-Core Homogéneos*. in *CASECongreso Argentino de Sistemas Embebidos*. 2012.
12. Xilinx, I., *Microblaze processor reference guide.* reference manual, 2011. **23**.
13. Naiqi Wu, M.Z., *System Modeling and Control with Resource-Oriented Petri Nets*, ed. C. Press2010, Boca Raton, FL.