Digital Implementation of a Logical Functor on a PLD

A. A. Vega-Ramírez*[1], J. L. Pérez-Silva[2]

[1] Faculty of Superior Studies Aragón (FES Aragón), UNAM
Avenida Rancho Seco, 57130
Netzahualcóyotl, Mexico
*alexandro_veg@yahoo.com
[2] Center for Applied Sciences and Technological Development
(CCADET), UNAM Circuito Exterior s/n,
Ciudad Universitaria, Coyoacán, 04510
Mexico City, Mexico

ABSTRACT
A digital circuit was implemented on a PLD to emulate the concept of logical functor physically, element from which all the Boolean connectors that exist can be generated. With this circuit, based on the Mc Cullock and Pitts neuron model and operating in agreement with the theory of the functor, several tests were carried out, starting from the application of values to the group of inputs that are considered as the threshold value that governs the behavior of the operator as well as to the group of inputs that are operated by the connector through this threshold. The obtained results present characteristics of behavior that refer to the above mentioned different Boolean operations. In these results, the utility of this functor is observed since it can be manipulated to solve diverse operation necessities, without requiring any connections or alterations to the existing system.

Keywords: Logic functor, time dependent logic, electronics neuron model.

RESUMEN
Se implementó un circuito digital, implantado en un PLD (Dispositivo Lógico Programable), para poder emular físicamente el concepto de functor lógico, elemento con el cual se pueden generar todos los conectivos booleanos que existen. Con este circuito, basado en el modelo de neurona Mc Cullock y Pitts, y operando de acuerdo con la teoría del functor, se realizaron varias pruebas, a partir de la aplicación de valores, tanto al grupo de entradas que se consideran como el valor de umbral, que rigen el comportamiento del operador, como al conjunto de entradas que son operadas por el conectivo, a través de dicho umbral. Los resultados obtenidos presentan características de comportamiento que refieren a las diferentes operaciones booleanas mencionadas. En estos resultados se observa la utilidad de este functor, ya que puede ser manipulado para resolver diversas necesidades de operación, sin requerir conexiones o alteraciones al sistema existente.

1. Introduction

Logic functors are connective elements which allow building a logic in an easy and generic way, similarly to how a Lego figure is built. Thus, the Theorist, with knowledge in logics, can develop these logic connectors and the application designer can compose them to design an application, and knowledge in logics is not required [1]. Query languages, such as lists of keywords in the case of search engines, or SQL in the case of databases, are examples of use of these applications. Logical formalism can be used to express queries and describe data, but also to define deduction relations between queries and answers [2]. Several information processing domains have logic-based components in which logic plays an important role, such as information systems with information retrieval or logic-based programming [3, 4]. This is the case of Logical Information Systems, in which logic is used to represent object descriptions and queries, to answer queries and to compute automatically a flexible navigation structure [5]. Querying and navigation in these systems is based on the ability to decide whether an object description is subsumed by a query or a navigation link [6, 7]. Several information processing domains have logic-based components: information retrieval [8, 9], diagnosis [10], programming [4, 11], and program analysis [12, 13, 14] are some examples, all of them logic-based. These components model an information processing domain in logic, and they also bring to the front solutions in which logic is the main engine. There is a proposal to model quality of service conditions, QoS, in logic and to make applications checking dynamic in such a way that the platform on which they run enforces the condition for a specified quality of service [15].

There are automatic systems that define logic functors as logic components. One component may be the propositional logic, another may be the interval logic, these logic functors, when composed, form new logics, a propositional logic on intervals [16]. Each logic functor has its own proof theory, which can be implemented as a theorem prover. If each proof theory and its theorem prover compose in a set of logic functors, it results in a composition of all the proof theories and their theorem provers of the component functors [16], and they implement a common interface. This makes it possible to construct generic applications that can be instantiated with a logic component. If there are customized logics built using logic functors, they can be embedded in an application that complies with this common interface [16, 17, 18]. Thus, logic functors specify off-the-shelf software components, the validation of the composition reduces to a form of type-checking, and their composition results in an automatic theorem prover. Logic functors can be easily assembled, and currently used in system-level programming, such as compilers, operating systems, file systems and information systems [16]. With this, a toolbox of logical components can be designed, and the user has only to compose these components.

The logic functors are the way of construction of logics from basic components. A construction of logics involves a definition of its syntax, its semantics via deduction rules, and its implementation as a theorem prover. A process is defined that goes from the description of a logic in terms of logic functors to an implementation of the logic [19]. The logic functor allows solving problems of digital design, reducing the drawback of developing a system where several basic-logic components are coupled, becoming more complex for the number of integrated components. These specific-purpose designs only solve one problem, based on Boolean logic; hence, if adding another variable or making a change is required, it is necessary to reconsider the problem and redesign it with the matching corrections, which causes to reconnect or to build the basic element-based circuit again. Because of this, the functor is established as the element from which all the Boolean logic connectors can be generated as peculiar cases [20]. If the functor is a result of a threshold logic from the neural model presented by Mc Cullock and Pitts, showing that these connectors are generated for a defined group of inputs starting from this element. If the function that governs the threshold is a stochastic one, a stochastic behavior of the functor can be presented, in such a way that the logical operators change according to the value of the inputs and of the threshold in time in a stochastic form [20]. Afterwards, with all this, a system can be developed which is based on one only logical element, the functor.

1.1 Definition of the generalized operators

In accordance with the basic concepts of the generalized operator, using the case of excitatory entrances, Figure (1):

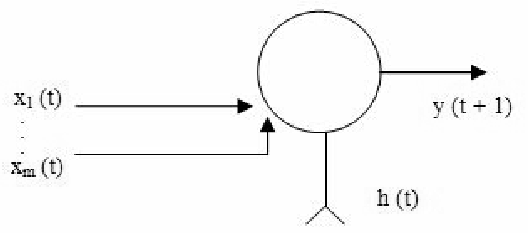

Figure 1. Diagram that shows the generalization of the event of a delay and multiple excitatory inputs.

Whose fire rule is

$$Dx(t) = \begin{cases} 1 \ if \ \sum_{i=1}^{m} \dot{x}_i(t) \geq h(t) \\ 0 \ if \ \sum_{i=1}^{m} \dot{x}_i(t) < h(t) \end{cases} \qquad (1)$$

Assuming the weight of all the entrances to be equal to one, then the generalized operator is at least *h* (threshold value) of *m* (inputs number) that will have a delay response if at least h of the excitatory inputs of the *m* is present. The symbol that represents it is

$$\prec^{m}_{h}$$

Two particular cases are observed:

"at least 1 of 2" is the temporal logical operator "Or".

$$\prec^{2}_{1} \equiv D\overset{t}{\vee}$$

and "at least 2 of 2" is the time dependent operator "And".

$$\prec^{2}_{2} \equiv D\overset{t}{\wedge}$$

For the case of only inhibitory inputs, as documented in [20], we have *n* inputs and a threshold *h(t),* as shown in Figure (2).

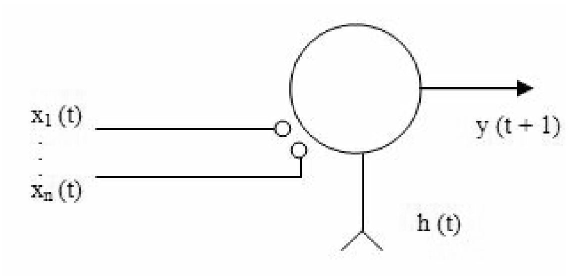

Figure 2. Diagram that shows an event with n inhibitory inputs and a threshold h(t).

And the fire rule will be

$$Dx(t) = \begin{cases} 1 \ if \ \sum_{i=1}^{m} \overset{\circ}{x}_i(t) \le h(t) \\ 0 \ if \ \sum_{i=1}^{m} \overset{\circ}{x}_i(t) > h(t) \end{cases} \qquad (2)$$

That is interpreted as when a lot of *h* (threshold value) of *n* (inputs number), that is the symmetrical operator of at least *h* of *m*. Its representation is

$$\prec^{n}_{h}$$

The case "when a lot 0 of 1", it is the time dependent negation operator "Not":

$$\succ^{1}_{0} \equiv D\overset{t}{\neg}$$

and "when a lot 0 of 2" is the time dependent logical operator "Nor".

$$\succ^{2}_{0} \equiv D\overset{t}{\neg\vee}$$

Operating both types of inputs, excitatory and inhibitory, we generalize the operator as shown in Figure (3):

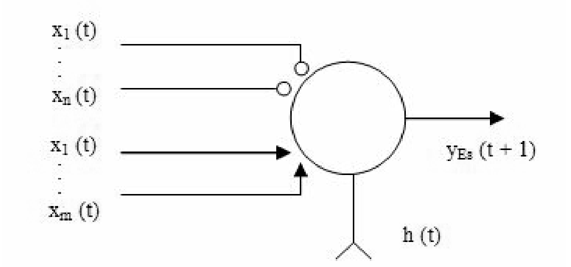

Figure 3. Generalization of the event of a delay and multiple inputs both excitatory and inhibitory.

Where the fire rule will be

$$Dx(t) = \begin{cases} 1 \ if \ \sum_{i=1}^{m} \overset{\bullet}{x}_i(t) - \sum_{i=1}^{m} \overset{\circ}{x}_j(t) \ge h(t) \\ 0 \ if \ \sum_{i=1}^{m} \overset{\bullet}{x}_i(t) - \sum_{i=1}^{m} \overset{\circ}{x}_j(t) < h(t) \end{cases}$$

(3)

This is the fundamental operator and it is represented by the symbol:

$${}_{h}\updownarrow^{m}_{n}$$

That it is defined as: The operator fires if at least *m* (excitatory inputs) minus *n* (inhibitory inputs) is equal to *h* (threshold). Then, the operator

$${}_{h}\updownarrow^{m}_{n}$$

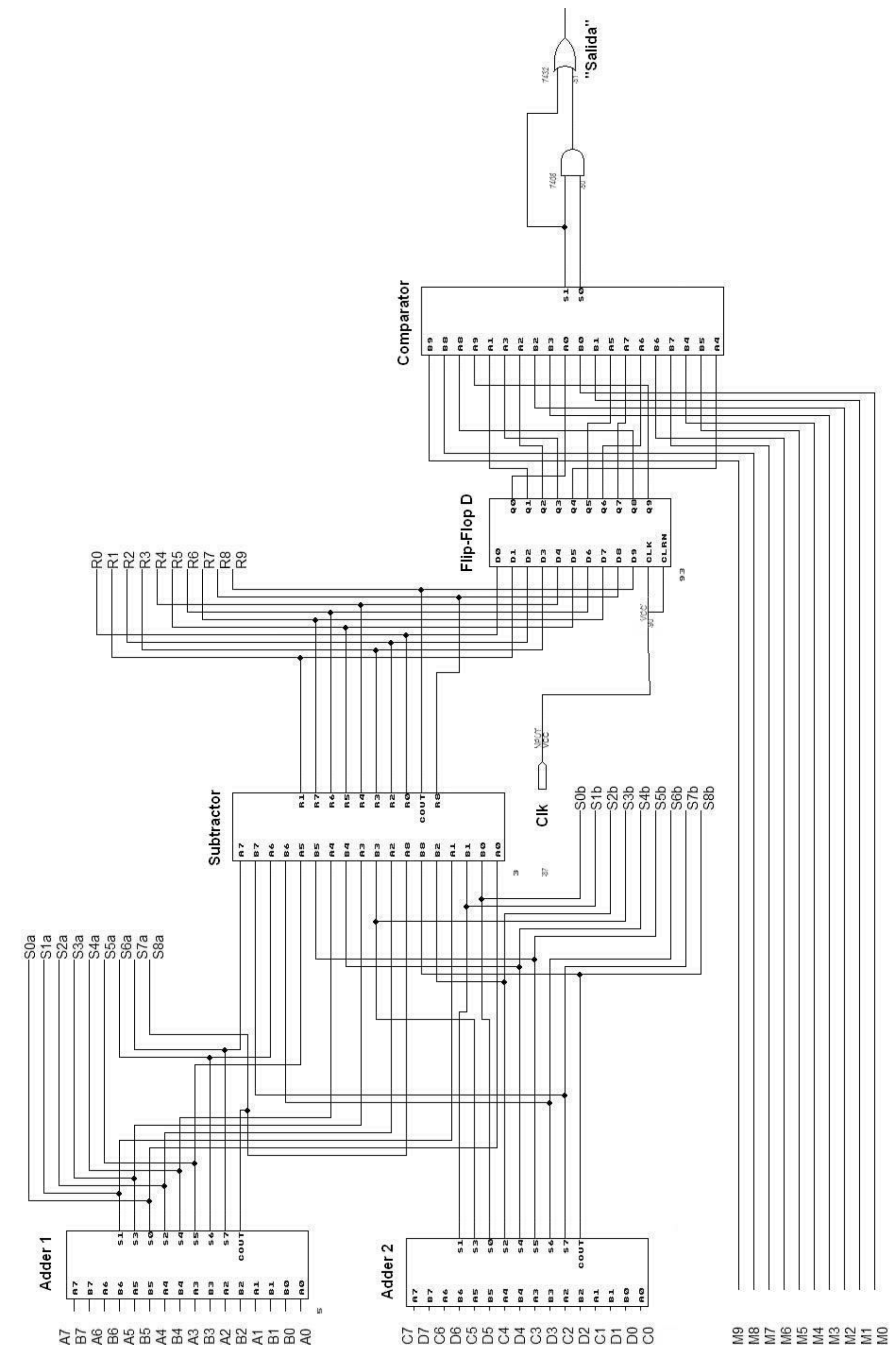
"Salida"
Comparator
Flip-Flop D
Subtractor
Clk
Adder 1
Adder 2
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9
S0a S1a S2a S3a S4a S5a S6a S7a S8a
S0b S1b S2b S3b S4b S5b S6b S7b S8b
A7 B7 A6 B6 A5 B5 A4 B4 A3 B3 A2 B2 A1 B1 B0 A0
C7 D7 C6 D6 C5 D5 C4 D4 C3 D3 C2 D2 C1 D1 D0 C0
M9 M8 M7 M6 M5 M4 M3 M2 M1 M0

Figure 4. Schematic circuit of the generalized time-dependent operator implanted in a PLD.

can be reduced to

$$\prec_h^m \quad \textit{for } n = 0$$

and to

$$\succ_h^n \quad \textit{for } m = 0.$$

Typical cases are:

$${}_0\updownarrow_1^1 \equiv D\overset{t}{\rightarrow},\ {}_0\succ_1^0 \equiv D\overset{t}{\neg}, \text{ and } {}_2\updownarrow_0^2 = \succ_2^2 \equiv D\overset{t}{\wedge}$$

Based on the above mentioned, the generalized operator allows a time dependent threshold to modify the logical operators.

2. Electronic implementation of the generalized operator

with base on the theory and operation principles described in this paper, and to prove that the operation of this element is verifiable using a physical device that operates in agreement with the mentioned position, a digital electronic circuit that represents the generalized operator was developed, built and implanted in a programmable logical device (pld). the schematic diagram that emulates it is shown in figure (4).

This basic circuit is a McCullock and Pitts neuron implemented with simple digital gates for the adders of excitatory and inhibitory inputs circuit, the comparator for the fire function, and for the delay circuit.

Using the circuit shown in Figure (4) (and as a threshold generator for the 8-bit input, a periodic descending ramp signal of 255 at 0 in its digital form), several cases were considered to be temporary generation. Both the threshold value and the performing responses of this generation are shown Figures (5), (6) and (7):

3. Results

The marked area in Figure (5) M(7..0) shows a descending ramp signal of 255 or FF at 0, changing each 2 milliseconds, as the threshold pattern. The quantities are in hexadecimal numbers. This threshold is compared with the result of the subtraction, R(9..0). A(7..0) and B(7..0) when added generate the result in S(8..0)a, while C(7..0) and D(7..0) when added generate the resulting quantity S(8..0)b. Both signals are subtracted to generate signal R(9..0) that is compared with the threshold signal. With this, the output “Salida” changes to 1, if M(7..0) = R(9..0), otherwise, the output is 0.

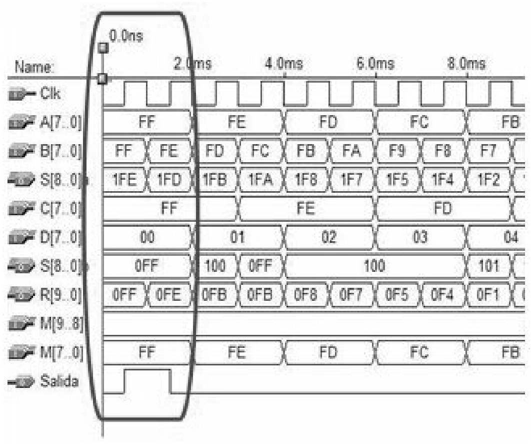

Figure 5. Partial graphic of the generalized response of the operator.

Then, “Salida” is 1 when M(7..0), FF, = R(9..0), 0FF; this is, at least FF of FF" (operator And) since the sum of the excitatory inputs, S(8..0)a = 1FE, is bigger than the sum of the inhibitory inputs, S(8..0)b = 0FF, there is a 1 at the output when the excitatory inputs are not inhibited, and the result of the subtraction is similar to the threshold. Afterwards, “Salida” is 0 when M(7..0), FF,> R(9..0), 0FE; it happens in the case when a lot of FF of FE" (operator Nor), this happens since the sum of the excitatory inputs, S(8..0)a = 1FD, is bigger than the sum of the inhibitory inputs, S(8..0)b = 0FF, there is a 0 when the excitatory ones are not inhibited, but the threshold is bigger than the result of the subtraction.

The marked area in Figure (6) shows how R(9..0) changes from 001 in hexadecimal notation to 3FE, also in hexadecimal notation; with this, the output takes the value of one, previous delay of 500 microseconds, since M(7..0) has a value of B2 in hexadecimal notation, having completed the condition, “Salida” is 1 when R(9..0) ≥ M(7..0), that is to say, when a lot B2 of 3FE" (operator Nor), this case is completed because the sum of the excitatory inputs, S(8..0)a, 117, is smaller than the sum of the inhibitory inputs S(8..0)b, 119.

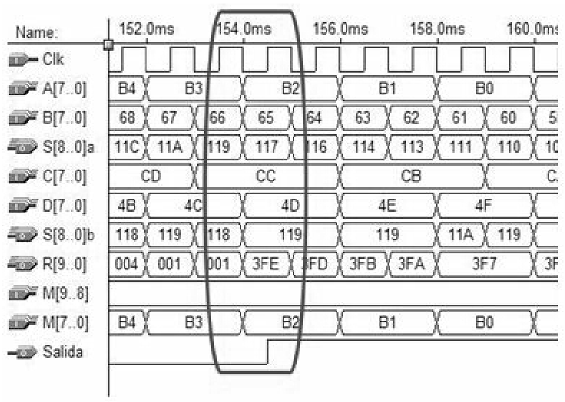

Figure 6. Details of the behavior of the generalized operator (continuation).

The inhibitory inputs inhibit the excitatory ones; because of this, there is 1 at the output when the excitatory inputs are inhibited, and the result of the subtraction is bigger than the threshold. The work frequency of the circuit is of 1 kHz, reason why every time that a pulse ascent flank appears in this period, the data enters to the comparator. With the result that a delay of 500 μS is noted, this happens because there is a wait for the flank to enter the hexadecimal 3FE data to the comparator.

The oval in Figure (7) shows that R(9..0) changes its value, from hexadecimal value 357 to hexadecimal value 054, the one when being compared with M(7..0), 7F makes that output “Salida”; again, previous delay of 500 μS goes from 1 to 0.

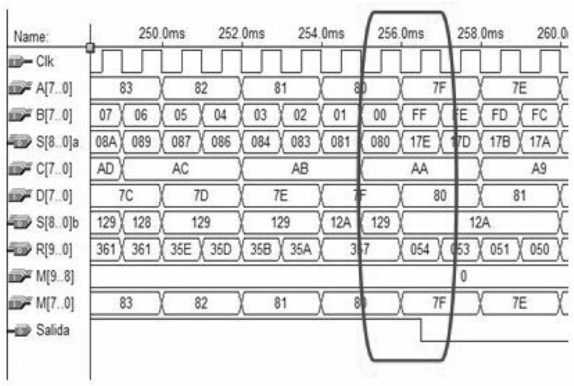

Figure 7. Partial view of the reponse of the generalized operator (continuation).

Observe how “Salida” changes when the pulse flank ascends, that is to say, when the comparator takes the value of R(9..0) to operate it with M(7..0). Now the condition is completed, output “Salida” is 0 when M(7..0) > R(9..0). It happens when a lot 7F of 054" (Nor operator), this happens since the sum of the excitatory inputs, S(8..0)a = 17E, is bigger than the sum of the inhibitory inputs, S(8..0)b = 12A; there is a 0 when the excitatory inputs are not inhibited but the threshold is bigger than the result of the subtraction.

As can be seen in Figures (5), (6) and (7), starting from the data variation in the inputs of the designed device and of the applied threshold, several events appear that emulate the different connectors of the Boolean logic. Since the threshold value changes with certain frequency, the operator does not remain in a single Boolean connector but rather it varies in function of the applied data. Then we can see that several configurations allow different handlings of the logical operators, and these depend on the purposes of their employment. The obtained results when manipulating several data with the operator developed in this work, show its usefulness as universal connector. In this paper, it is presented as an operator with characteristics of behavior that respond to the theoretical bases and to the operation principle of the logical functor described in the Introduction. Figure (8) shows the symbolic form in which this generalized operator can be represented, Figure (4) shows the operator implanted in a PLD whose fire form is

$$Salida\ (t+1) = \begin{cases} 1\ if\ \sum_{i=A}^{B} \overset{\bullet}{i}(t) - \sum_{j=C}^{D} \overset{\circ}{j}(t) \geq M(9..0) \\ 0\ if\ \sum_{i=A}^{B} \overset{\bullet}{i}\ (t) - \sum_{j=C}^{D} \overset{\circ}{j}(t) < M(9..0) \end{cases} \tag{4}$$

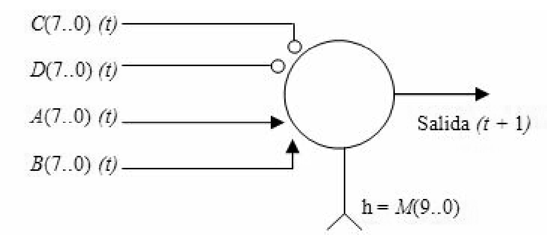

Figure 8. Symbolic form of the generalized operator in Figure (4), M(9 ..0); it is a binary value of the threshold.

A simple example in which this generalized operator can be applied is shown next. The circuit of Figure (9) will operate only when bigger or similar to 1011, 11 binary, combinations exist. That is what indicates the Boolean expression that is shown, F = A(B + CD).

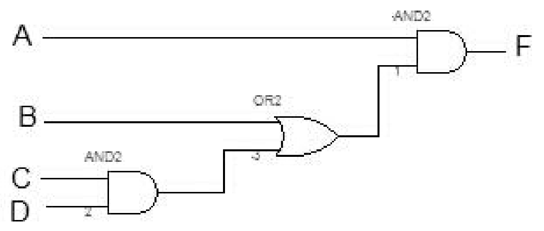

Figure 9. Logical circuit that responds to he Boolean expression F = A(B + CD).

Nevertheless, it happens that the condition has changed. Now, the circuit should operate when the combinations are bigger or similar to 1101, 13 binary, that is to say, F = AB(D + C).

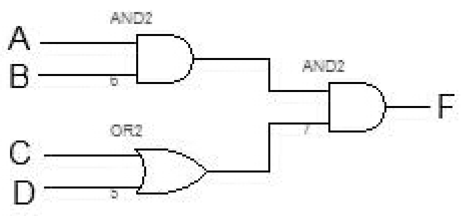

Figure 10. Logical circuit readapted to cover the conditions of F = AB(D + C).

This implies a redesign, or readaptation of the logical elements, in such a way that this condition can be executed. As observed in Figure (10), this has required to change connections and to move connectors. The use of a generalized operator has the advantage of avoiding to alter or to restructure a circuit.

Figure (11) shows the configured generalized operator in such a way that, thanks to the weights that can be assigned to the inputs, generates a certain value that can be compared with the threshold. Due to this, the threshold has fixed in 11, therefore, the output will be activated when the sum of all the inputs is equal or bigger than this value; otherwise, it will remain in zero.

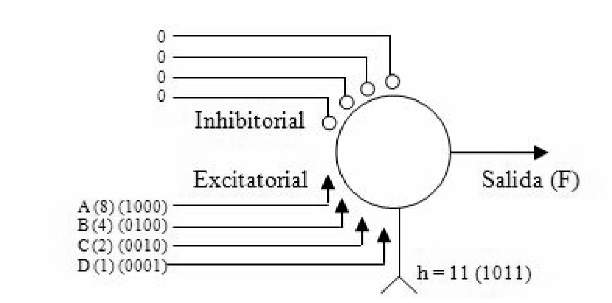

Figure 11. Appropriate generalized operator to operate the Boolean expression F = A(B + CD).

Under the consideration that the conditions changed, the adjustment in the generalized operator is minimum, Figure (12), because only the threshold value has been varied. This is, the output will be activated when the sum of the inputs is equal or bigger than 13. If it does not happen in this way, the output will remain inactive.

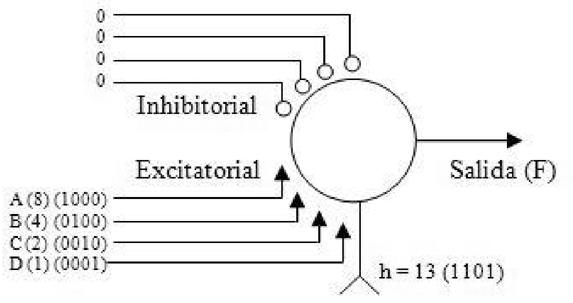

Figure 12. Generalized operator that responds to the Boolean expression F = AB(D + C).

4. Conclusions

The generalized operators implanted in programmable logical devices (PLDs) can be applied in the solution of problems or in systems that operate with binary components without the need of using too many of these simple components since their internal structure accommodates to the fundamental operators of the logic, depending on the function that is applied to the threshold. The small example shows the utility of implanting a generalized operator in a PLD because of the easiness that this operator has and for its notorious configurability to be adapted to the problem. Besides that it doesn't need any additional devices, because as it has been seen in the analysis, everything is contemplated and integrated in its design.

On the other hand, the obtained results using this digital generalized operator show the utility in its installation as universal connector, with a notable point: this operator has operating readiness with groups of binary data. If it is necessary to operate with continuous signals, this seems to be a problem to solve. This operator, based on digital devices and implanted in the programmable logical device, can operate the process because its operation, using only binary data, can be increased with the use of analog-digital and digital-analog converters devices that allow the conditioning from the continuous signals to groups of binary data, and viceversa, which can be operated by this digital operator. Figures (13) and (14) show graphic results of the digital generalized operator behavior when continuous signals are applied. In this case, the whole model integrated in the PLD has an external addition, these analog-digital and digital-analog converters.

Figure (13) shows two input signals, Vin1 and Vin2, both changing on two level values, 0 and 1. These signals are operated by the digital operator, particularly, they are added; Vsum shows the addition result: it is a three level result. In this case, the threshold is set to 1, and Vsum can get three levels: 0, 1 or 2, hence, when Vsum is bigger or equal to the threshold, 1, Vout changes to 1, otherwise, the value remains 0; which means, if Vsum ≥ threshold, then Vout = 1, otherwise, Vout = 0. This behavior agrees with "at least 1 of 2", and it is the logical operator "Or". Therefore, Figure (13) shows an “Or” operator behavior.

Figure (14) shows something similar to Figure (13): two input signals, Vin1 and Vin2, both changing on two level values, 0 and 1. These signals are also operated (added) by the digital operator, and Vsum shows the addition’s three-level result. This time, the threshold is set to 2. As before, Vsum can get three levels, 0, 1 or 2, thus, when Vsum is bigger or equal to the threshold, now in value 2, Vout changes to 1, otherwise, it remains in value 0. It means, if Vsum ≥ threshold, then Vout = 1, otherwise, Vout = 0. At this time, the behavior agrees with "at least 2 of 2", and it is the logical operator "And". Thus, Figure (14) shows an “And” operator behavior.

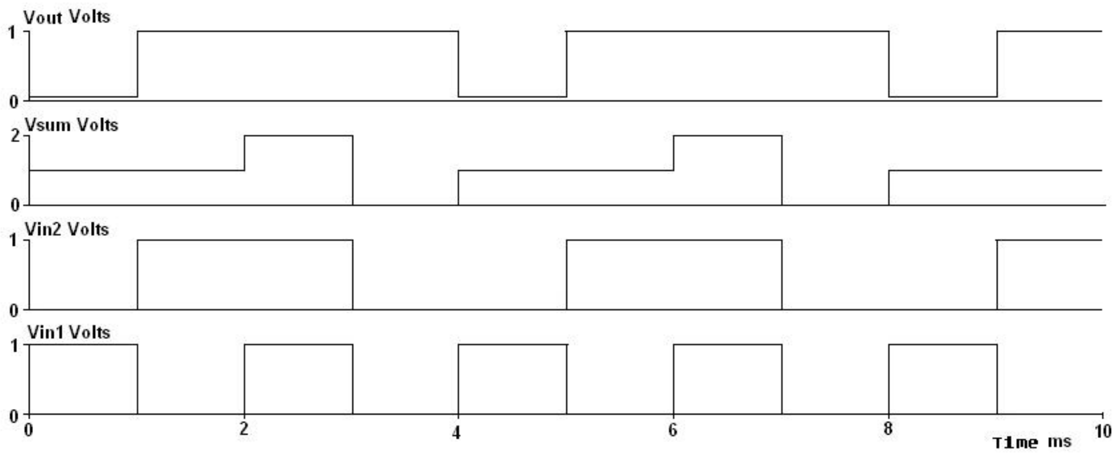

Figure 13. Digital generalized operator with analog signals. “Or” operator.

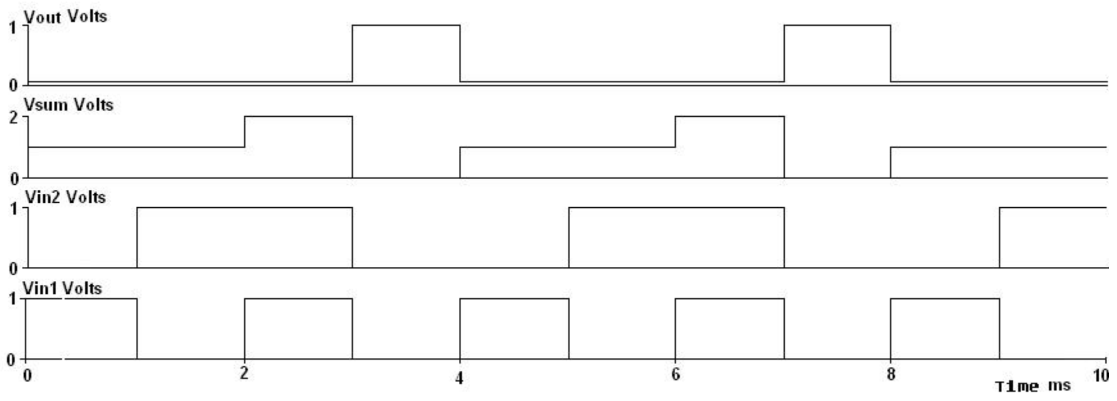

Figure 14. Digital generalized operator with analog signals. “And” operator.

As can be seen, Figures (13) and (14) describe and reveal the generalized operator’s performance and its theoretical bases, now operating with analog signals. With these results, the option of the generalized operator based on digital components is viable to be used as universal Boolean connector in applications where it is required to operate under this binary logic, with possibility to operate analog signals, using additional devices, the mentioned converters.

This PLD-based operator has another very important advantage that worths mentioning: the possibility of increasing the number of inputs related to the process to be operated. This is because PLDs are high integration devices with also a high number of input pins, hundreds per instance, which allows expanding its scope to any application. Moreover, its internal structure can make changes quickly and easily, hence, the changes are made inside the PLD with the help of the device programmer. This programmer can program the device with a schematic option, as done in this case, or with hardware description language, such as Verilog or VHDL, another advantage to mention.

Because of its high scale of integration, delays are very small, 7.5 ns, thus, this PLD-based operator can operate signals with low percentage of data loss. This is a very important advantage when selecting a PLD as a device to work with. Operation frequencies are related to the signals involved in the processes to operate and to the converters to be used, therefore, the PLD-based operator can have a good performance, like that seen in this article.

If operation with analog signals involves a very high number of converters and cost becomes greater, an implementation of a digital generalized operator can be developed with a microcontroller-based design. This design allows us to convert analog signals in groups of binary data with just one device, the own microcontroller, with a built-in analog-digital device, and operate it as a universal connector showing the same utility in its installation as the PLD-based operator. However, a very remarkable problem is the conversion time, because this microcontroller has just one converter for four or even eight analog signals; hence, these conversion times are delays in the operation. Therefore, the operation frequency is decremented because of the sequential process of the microcontroller. With binary data, a microcontroller-based operator cannot increase the number of inputs to the process to operate because the number of pins and ports to work is fixed. And with analog signals, only 8 signals can be operated - it is the highest number of analog pins in a microcontroller.

To conclude, we consider it is good option to implement a Boolean connective in a PLD-based generalized operator because of the advantages shown in this article.

References

[1] André, E. Handling Theories in Logic Functors for Recomposing Description Logics. Institut de Recherche en Informatique et Systems Aléatoires, Rennes, France, 2007. 56 pp.

[2] Van Rijsbergen, C. J. A new theoretical framework for information retrieval. SIGIR Forum, 21(1-2):23–29, 1987.

[3] Lloyd, J.W. Foundations of Logic Programming. Symbolic computation — Artificial Intelligence. Springer, Berlin, 1987.

[4] Marriott, K. and Stuckey, P. J. Programming with Constraints: An Introduction. The MIT Press, 1998.

[5] Ferré, S. and Ridoux, O. An introduction to logical information systems. Information processing & Management, 40(3):383–419, 2004.

[6] S. Ferré, O. Ridoux. Introduction to Logical Information Systems. Institut National de Recherche en Informatique et en Automatique, Rennes, France. Rapport de recherche no. 4540, Septembre, 2002. 25 pp.

[7] Calvanese, D. Lenzerini, M. and Nardi, D. Description logics for conceptual data modeling. In Logics for Databases and Information Systems, pages 229–263, 1998.

[8] Salton, G. and McGill, M. J. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[9] Van Rijsbergen, C.J. Crestani, F. and Lalmas, M. editors. Information Retrieval: Uncertainty and Logics. Advanced models for the representation and retrieval of information. Kluwer Academic Publishing, Dordrecht, NL, 1998.

[10] Poole, D. Representing knowledge for logic-based diagnosis. In Int. Conf. Fifth Generation Computer Systems, pages 1282-1290. Springer, 1988.

[11] Lloyd, J. W. Foundations of Logic Programming. Symbolic computation - Artificial Intelligence. Springer, Berlin, 1987.

[12] Sagiv, M. Francez, N. Rodeh, M. and Wilhelm, R. A logic-based approach to program flow analysis. Acta Informatica, 35 (6): 457-504, June, 1998.

[13] Armstrong, T. Marriott, K. Schachte, P. and Søndergaard, H. Two classes of boolean functions for dependency analysis. Science of Computer Programming, 31: 3-45, 1998.

[14] Codish, M. Søndergaard, H. and Stuckey, P. J. Sharing and groundness dependencies in logic programs. ACM TOPLAS, 21(5): 948-976, 1999.

[15] Issarny, V. and Bidan, Ch. Aster: A framework for sound customization of distributed runtime systems. In 16th Int. Conf. Distributed Computing Systems, 1996.

[16] S. Ferré, O. Ridoux. Logic Functors: A Framework for Developing Embeddable Customized Logics. Institut de Recherche en Informatique et Systems Aléatoires, Rennes, France. Publication interne no. 1459, Mai 2002, 32 pp.

[17] Ferré, S. and Ridoux, O. A logical generalization of formal concept analysis. In G. Mineau and B. Ganter, editors, Int. Conf. Conceptual Structures, LNCS 1867, pages 371-384. Springer, 2000.

[18] Ferré, S. and Ridoux, O. Searching for objects and properties with logical concept analysis. In H. S. Delugach and G. Stumme, editors, Int. Conf. Conceptual Structures, LNCS 2120, pages 187-201. Springer, 2001.

[19] S. Ferré, O. Ridoux. The Principles of a Toolbox for the Implementation of Customized Logics. Institut de Recherche en Informatique et Systems Aléatoires, Rennes, France. 2000, pp 104-111.

[20] J. L. Pérez S., F. Lara, A. Miranda, A. Garcés. A. Herrera, M. Bañuelos, J. Castillo, S. Quintana, A. Padrón. El Functor Lógico Como Elemento Básico De Una Lógica Temporal Estocástica. Memorias del Congreso Nacional de Instrumentación SOMI XVI, Querétaro, Querétaro, México, 2001.

Authors´ Biographies

Alejandro Antonio VEGA-RAMÍREZ

He received both the B. S. and M. S. degrees in electrical mechanic engineering from Universidad Nacional Autónoma de México (National Autonomous University of Mexico), UNAM, in 2001 and 2006, respectively. He is currently an electrical engineering PhD. student at Centro de Ciencias Aplicadas y Desarrollo Tecnológico (Center for Applied Sciences and Technological Development) of the UNAM. Since 1999, he has been with Facultad de Estudios Superiores Aragón (Faculty of Higher Education Aragón), FES Aragón, at first as professor assistant and later as professor (2001). His research interests lay on digital electronics, digital systems implementation, digital systems design, and programmable logic devices for artificial neurons and neural networks, in control and pattern recognition processes.