International Journal of Computer and Information Technology (ISSN: 2279 – 0764)
Volume 02– Issue 05, September 2013

Digital Equivalence of Biological Neural AND-gate, OR-gate and MIN-Gate

Nicoladie D. Tam
Department of Biological Sciences
University of North Texas
Denton, TX 76203 USA
Email: nicoladie.tam {at} unt.edu

Abstract— **This paper provides the mathematical derivation of the equivalent logic gates for spike code processing in neurons. It derives the computational equivalence between the binary code and spike code to illustrate the correspondence between binary and spike code logic operations theoretically. It identifies the similarities and differences between biological neural processing and binary logic gate processing. Using neural spike code for processing, a neuron can be generalized to process the equivalent of a multi-input *OR*-gate and a multi-input *AND*-gate by requiring a minimal number of *m* input spikes (from a set of all *n* inputs) before firing an output spike. A MIN_m-gate is introduced as the equivalent of the generalized multi-input AND_m-gate and multi-input OR_m-gate that require a minimum of *m* inputs to fire an output spike. The MIN_m-gate is an equivalent of a statistical voting system that processes input spikes with a threshold of a minimum of $m \geq n/2$ input spikes for a majority-rule system.**

Keywords- massively parallel processing, biological neural processing, digital logic gate operation, asynchronous processing

I. Introduction

This paper examines the computational equivalence between biological neural circuitries and digital logic so that the similarities and differences between them can be established. Although the similarities between artificial neural networks and digital logic circuitries have been explored [1-4], this paper focuses on addressing the computational equivalence of biological neurons using pulse-coded signals for processing rather than binary-coded signals. The unique computational characteristics using spike-coded signals (a special class of pulse-coded signals) are addressed in this paper, so that the computational equivalence between neural processing and binary processing can be identified. It will be shown below that if spike codes were used in the logic gates (instead of binary code), there exists an equivalent between a multi-input *AND*-gate and a multi-input *OR*-gate that can be replaced by a *MIN*-gate with a minimum threshold of *m* active inputs out of all total *n* inputs. Varying the threshold *m* will provide a majority-rule voting logic gate by a set of generalized *AND*-gates or *OR*-gates without requiring any custom-design complex voting VLSI digital logic circuitry [5-8]. Spike-coded signals can also be processed asynchronously without relying on any external clock pulse for synchronization.

II. Binary Code vs. Spike Code Encoding

In order to identify the computational differences between a spike-coded signal and binary signal for the subsequent mathematical derivation of the spike code processing logic, a brief review is provided here. The difference between binary-coded signals and spike-coded signals is that binary codes represent time-independent up/down states, whereas spike codes are time-dependent pulse-coded signals the represents a point process.

Pulse code uses both pulse width and pulse height to encode information [9]. That is, the time duration of the pulse does encode information (in addition to encode information by the amplitude of the pulse) [10]. In contrast, the time duration of the binary signals does not encode any additional information, since the binary information is encoded in the 0's and 1's only (up/down states without any dependence on how long the binary states last in time).

The spike code is a special type of binary-coded pulse code in which it takes on the value of 0's and 1's (as amplitude), except that the time duration of 1's is always fixed (a constant Δt), whereas the duration of 0's is a variable t. This time-dependent information encoding will result in producing some major differences between *how* signals are processed and *what* computations they can perform.

III. Characteristics of Spike Trains

The major difference is that spike code encodes the time occurrence of events (by using 1's to denote the time of occurrence of an event at time *t*, and 0's to denote absence of any event). Thus, spike-coded signal is a special class of pulse-coded signal with these distinct characteristics:

(1) The *occurrences* of an event and nonevent information are encoded by the binary code (fixed amplitude signals of 1's for encoding occurrence of events, and variable time duration of 0's for encoding non-occurrence of events, respectively);

(2) The timing information of the 1's (spikes) is encoded as the time of occurrence of an *event*;

(3) The timing information of the 0's (non-spikes) is encoded as *silence* (period of non-occurrence of events);

(4) The *pulse width* of the 0's (time duration of the 0's) encodes the *silence period* (without any intervening events);

(5) But the pulse width of the 1's does not encode any additional timing information, since a spike is always *fixed in duration* and amplitude (i.e., constant pulse width of Δt and pulse height of 1, representing the *occurrence* of an event only, but not the duration of the event);

(6) The spike can be considered as representing a *point* in time (with infinitesimal time Δt in duration);

(7) A series of spike code in time can be considered as a time-series of *point processes*;

(8) The time-series of spikes is considered as a *spike train*, representing the time of occurrence of spikes.

IV. Interpretation of 1's and 0's in Spike Trains

The 1's and 0's in spike code have very different meaning from the 1's and 0's in binary code. Since spike codes are encoding a time-series of events, the occurrence of an event can be considered as a point (in time) with an infinitesimal time Δt in duration. Therefore, spike code processing is essentially processing a time-series of point processes [11], which is different from the interpretation of the 1's and 0's of binary coded signals. The spike train encodes the time-series of spike occurrences in time [12] rather than a series of static 1's and 0's without any timing information embedded in the binary code. Most biological neurons use spike trains to encode information and to communicate instead of using binary signals without any timing information (cf. [13, 14]).

Thus, the major difference between spike and binary codes is that spike-coded signal processing essentially processes *event signals* specifically, whereas binary-coded signal processing can process *any* information encoded by the 1's and 0's. The information encoded by binary code is not limited only to events and nonevents. That is, binary code encodes 1's and 0's equally without any predetermined informational content (usually representing up/down states or on/off states), whereas spike code encodes 1's specifically as the *time of occurrence* of an event, while it encodes 0's as the *silence period* (the absence of any intervening events).

There is an asymmetry of the information encoded by 1's (as discrete events) and 0's (as silence period) in spike-coded signals. The duration Δt of the 1's (spikes) has no significant information, but the duration of 0's (silence) has significant information — encoding the silence period (with no intervening events). We will show below that even though spike code encodes very specific information content of event occurrences, it has the advantage of simplifying the circuitry in signal processing. It reduces the complexity of the computational load, if the time-dependent spike code logic were used, instead of the time-independent binary code logic.

V. Neural Code

Biological neurons essentially use both digital and analog signals for processing signals electrically. They use an equivalent analog circuitry (embedded in the membrane) to process the incoming signals, and then use the same circuitry to generate the digital signals to be transmitted to the next neurons. Higher animals use primarily digital signals for transmission and communication, while lower animals use analog signals. The digital signal is the spike-coded signal represented by the waveform of an action potential [15]. The mechanism for generating the electrical spike waveform is well-understood in what is known as the Hodgkin-Huxley equation [15]. The analog waveform of an action potential is represented by an analog pulse signal caused by a sudden change in the voltage across the neuron's membrane. The spike-coded signals used by neurons can be considered as hybrid signals encoding digitally for transmission and communication.

The spike codes are generated by action potentials (nerve impulses) in neurons. The spikes (1's) are often considered as the active output of a neuron (firing an event), while 0's (silence) are considered as an inactive output (no firing). The spike code of 1's is propagated along the axon (output of neuron), while maintaining the same pulse height electrically.

This ability to maintain the constant spike amplitude provides the digital signal needed for communication and processing. This constant spike amplitude is achieved by the regenerative process of action potential when it propagates over long distances without any signal decay. The regenerative process of action potentials is described mathematically by the Hodgkin-Huxley equation [15].

VI. Neural Processing of Artificial Neural Networks

The time-dependent spike code processing is unique to biological neurons, even though the computational equivalence of binary coded artificial neural networks has been extensively investigated and implemented in hardware [1-4]. The difference between artificial and biological neural networks is that artificial neurons often use binary code for processing, whereas biological neurons use spike code. Even though spiking neurons are used in some of the artificial neural networks, a burst of spike firings is used to represent the binary on-state, and the silent period is used to represent the off-state [16-18].

That is, a *burst* of spike firings (a sequence of 1's) is lumped together as a single *firing state* ("on" state) rather than treating individual single spike firing as an independent event. These bursts of spike firing are often treated as a single up-state (a single 1, not a sequence of 1's) while non-firing is treated as a single down-state (a single 0). Since the 1's and 0's are treated as on/off or up/down static "states," the time-dependence of spike firing is ignored. This reduces the spike code back to the time-independent binary code rather than retaining the time-dependence of event occurrence information uniquely encoded by a spike.

Thus, the neural processing in artificial neural network essentially processes binary state information (static state without any time dependence), rather than spike event information (dynamic state with time dependence on when these event occurred). In other words, digital processing in most artificial neural networks deals with state transitioning between up (1's) and down (0's) states rather than processing the spike train time-series.

In contrast, biological neural processing deals with point processing of events in the spike train. It uses silence periods to represent absence of any point events. Even though there are significance differences between the coding processes, we will introduce the equivalent processing between the binary code processing and spike train processing below.

VII. Spike Code Representation

A spike train can be expressed mathematically by a Dirac delta, $\delta(t)$, function:

$$\delta(t) = \begin{cases} +\infty, & t = 0 \\ 0, & t \neq 0 \end{cases} \tag{1}$$

which is constrained to satisfy the identity:

$$\int_{-\infty}^{\infty} \delta(t)\,dt = 1 \tag{2}$$

Although the above Dirac delta function theoretically has an infinite spike amplitude at time 0 (the time of occurrence of the spike), it can be reduced to a unit-delta, $\delta'(t)$, function with a small finite time increment, Δt at time t:

$$\delta'(t) = \begin{cases} 1, & \text{for } 0 \leq t \leq \Delta t \\ 0, & \text{for } t < 0 \text{ or } t > \Delta t \end{cases} \tag{3}$$

The time-series of spike-coded signals $x(t)$ at any given time t is then given by:

$$x(t) = \left\{\delta(t_1) + \delta(t_2) + \ldots + \delta(t_n)\right\} \tag{4}$$

where t_i is the time of occurrence of a spike. This is essentially the digital signal needed for processing of signals in spike train computation.

VIII. Spike Train Processing without External Clock Pulse for Synchronization

The important distinction between spike code and binary code processing is that spike code processing is time sensitive. Processing occurs only during a small finite time window of Δt when the inputs arrive simultaneously at the same time window because the duration of 1's only lasts for Δt. Furthermore, since Δt is a constant, processing occurs only during the time window of Δt, without depending on any external clock pulse for processing. The clock pulse is self-contained in the spike code itself.

On the contrary, binary code processing is less time sensitive because the duration of 1's can last much longer than Δt, and processing is dependent on an external clock pulse for synchronization. Since binary code processing relies on an external clock pulse, the time window for processing can vary depending on the duration of the external clock pulse, providing for some flexibility of processing, unlike neural processing.

IX. Asynchronous Processing by Spike Trains

Spike train processing implicitly carries its clock pulse Δt as the 1's, it is self-contained in synchronization, which means it can process asynchronously with any time delay in the signal transmission (without relying on any extra transmission line for delivering the clock pulse). This self-contained clock pulse in spike code lends itself to asynchronous process for multi-input processing even with variable time delays, which is intrinsic to signal transmission. The processing is essentially self-timed asynchronously by its self-contained clock pulse in the spike.

X. Multi-Input Logic Gate Equivalents

Most biological neurons integrate thousands of synaptic spike inputs simultaneously by a neuron (neural processor) to produce a single output spike. In contrast, most digital logic gates process only two (or less) inputs to form one output. The most common digital processing of two inputs to form one output includes the following logic gates such as:

- *AND*-gate: for processing *a = b AND c*,
- *OR*-gate: for processing *a = b OR c*,
- *NAND*-gate: for processing *a = b NAND c*, and
- *NOR*-gate: for processing *a = b NOR c*.

An example of digital processing of one input to form a different output is:

- *NOT*-gate: for processing *a = NOT b*.

On the contrary, a single biological neuron often processes tens of thousands of inputs rather than two (or less) inputs, compared to most digital logic gates in computers. For example, a Purkinje neuron (in the cerebellum) processes more than 100,000 synaptic inputs connections simultaneously [19], which exemplifies the massively parallel operation performed by a single neuron. The equivalent logic gates for processing multiple inputs are:

- *AND*-gate: for processing $y = x_1$ *AND* x_2 *AND* … *AND* x_n
- *OR*-gate: for processing $y = x_1$ *OR* x_2 *OR* … *OR* x_n,

where x_i are the i-th inputs (for a total of n inputs) and $y(t)$ is the output (Fig. 1).

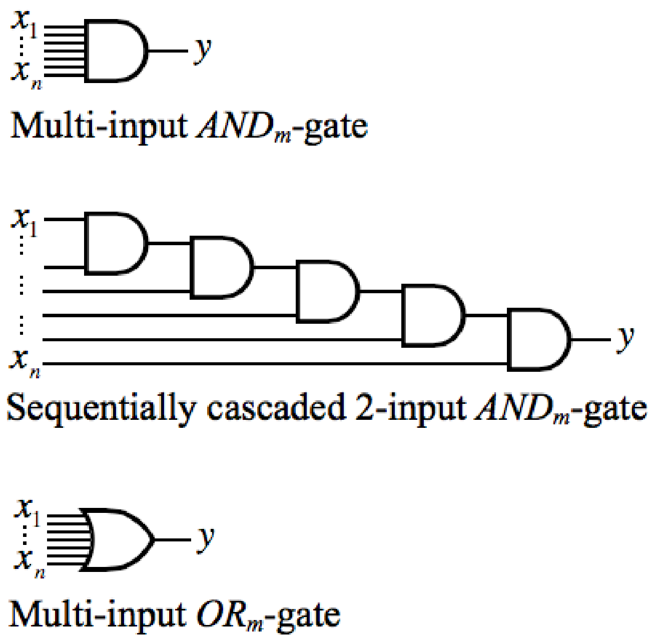

Figure 1. Multi-input logic gates for AND_m-gate and OR_m-gate that require a minimum of m active input spikes to produce an output spike. The multi-in

The logic processing can be generalized to include simultaneous multi-inputs to form a single output (see Table 1). All n-inputs can be operated on simultaneously, rather than sequentially. If n is large, this provides a massively parallel hardware implementation of the multi-input gates.

Logic Gates	Signal Processing of Multiple n-inputs
AND_n-gate	$y = x_1$ *AND* x_2 *AND* … *AND* x_n
OR_n-gate	$y = x_1$ *OR* x_2 *OR* … *OR* x_n

Table 1. Digital logic gate operation for signal processing of multiple n-inputs (variables $x_1, x_2, \ldots, x_n$) simultaneously to form a single output (variable y).

The multi-input time-independent n-input AND_n-gate can be represented in hardware by:

$$y = x_1 \; AND \; x_2 \; AND \; \ldots \; AND \; x_n \tag{5}$$

or by the *AND*() function in software:

$$y = AND_n(\mathrm{x}_1, x_2, \; \ldots \; x_n) \tag{6}$$

where *AND*() is a function operating on the multiple parameters $x_1, x_2, \ldots, x_n$ to produce a single output variable y. Note that this *AND*-gate binary code operation is time-independent, unlike the time-dependent processing in spike train processing of a time-series of $x(t)$.

Although the multi-parameter AND_n() function is implemented in software as a pseudo-parallel operation, it is often done by the software compiler to instantiate the *AND*-operations (multiple *AND*-ing operations) into a series of repeated hardware implementations (multiple cascaded *AND*-gate operations). This is accomplished by a sequential operation (instead of parallel operation) in a pipeline fashion, cascading a two-input *AND*-gate, i.e., *AND*(x_1, x_2) operation iteratively (see Fig. 1). The software *AND*-operation does not process all n inputs, i.e., $AND_n(x_1, x_2, \ldots, x_n)$, simultaneously nor in parallel. The difference between parallel multi-input AND_n-gate operation and sequential two-input *AND*-gate pipeline is shown in Fig. 1.

XI. Time-dependent Spike Train Processing

The time-dependent nature of the spike events is denoted by the time-series $x_i(t)$ and $y(t)$ as a function of time t. For time-dependent spike trains, the processing can be operated by the equivalent multi-input logic operator:

$$\begin{aligned} y(t) &= x_1(t)\, AND\, x_2(t)\, AND \ldots AND\, x_n(t) \\ &= AND_n\big(x_1(t),\, x_2(t),\, \ldots, x_n(t)\big) \\ &= \begin{cases} 1, \;\; \text{if} \sum_{i=1}^{i=n} x_i(t) \;\geq n, \;\; \text{for}\; 0 \leq t \leq \Delta t \\ 0, \;\; \text{if} \sum_{i=1}^{i=n} x_i(t) \;< n, \;\; \text{for}\; t < 0 \;\; \text{or} \;\; t > \Delta t \end{cases} \end{aligned} \tag{7}$$

where $x_i(t)$ is the i-th inputs (for a total of n inputs) and $y(t)$ is the output (Fig. 1). Note that we express the above AND_n-operator with an equivalent AND_n-function, which takes on a variable number of parameters $x_1, x_2, \ldots, x_n$ (the second part of Eq. 7). The last part of Eq. 7 indicates that, if the sum of *all* n inputs is greater than or equal to n, then the output is 1; otherwise, the output is 0. This means that *all* of its input has to fire a spike before the neuron will fire an output spike, satisfying the definition of AND_n-gate for all incoming spikes.

The multi-input digital logic AND_n() function is essentially an adder (*ADD*-gate) with a threshold of n set to produce a spike output of 1, only if the sum exceeds n; otherwise 0.

Similarly, the multi-input n-input time-independent conventional OR_n-gate can be represented in hardware by:

$$y = x_1 \; OR \; x_2 \; OR \; \ldots \; OR \; x_n \tag{8}$$

or by the *OR*() function in software:

$$y = OR_n(\mathrm{x}_1, x_2, \; \ldots \; x_n) \tag{9}$$

where *OR*() is a function operating on the multiple parameters $x_1, x_2, \ldots, x_n$ to produce a single output variable y.

For spike train processing, the multi-input time-dependent OR_n-gate can be expressed by setting the threshold to be 1 instead of n. The multi-input time-dependent OR_n-gate can be re-expressed by the time-dependent input and output functions, $x_i(t)$ and $y(t)$:

$$\begin{aligned} y(t) &= x_1(t)\, OR\, x_2(t)\, OR \ldots OR\, x_n(t) \\ &= OR_n\big(x_1(t),\, x_2(t),\, \ldots, x_n(t)\big) \\ &= \begin{cases} 1, \;\; \text{if} \sum_{i=1}^{i=n} x_i(t) \;\geq 1, \;\; \text{for}\; 0 \leq t \leq \Delta t \\ 0, \;\; \text{if} \sum_{i=1}^{i=n} x_i(t) \;< 1, \;\; \text{for}\; t < 0 \;\; \text{or} \;\; t > \Delta t \end{cases} \end{aligned} \tag{10}$$

The multi-input digital logic OR_n() function would fire a spike output if any *one* of the n inputs fires a spike at time t, which satisfies the definition of *OR*-operation for the incoming spikes. The last part of Eq. 10 indicates that, if the sum of all n inputs is greater than or equal to 1, then the output is 1; otherwise, the output is 0.

That is, the multi-input digital logic OR_n() function is also an adder (*ADD*-gate) with a threshold set to 1 to produce a spike output; otherwise no spike is generated. That is, a minimum of *one* active input (a spike from its n inputs) is required to produce an output of 1 for an *OR*-gate operation, by definition. The OR_n-gate in Eq. 10 is the least restricted generalized *OR*-gate because it requires only a minimum of *one* spike (from its n inputs) for it to produce an output of 1 (a spike).

XII. Equivalence between OR_N-gate and a Minimum of 1-input for AND_1-gate

Most often in the real world neurons, it requires more than one spike to fire an output spike. That is, it requires a minimum of m active inputs for the generic *OR*-gate rather than a minimum of *one* active input in the conventional *OR*-gate. Extending this to a generalized OR_m-gate, it requires a minimum of m input spikes firing at time t before producing an output spike in the massively parallel signal processing of the OR_m function. (Note that we use the subscript m in the OR_m-gate here to denote the minimum of m inputs in its processing.) The multi-input OR_m-gate is given by (see Fig. 1):

$$
\begin{aligned}
y(t) &= x_1(t)\ OR_m\ x_2(t)\ OR_m\ \ldots OR_m\ x_n(t) \\
&= OR_m\big(x_1(t),\ x_2(t),\ \ldots, x_n(t)\big) \qquad (11) \\
&= \begin{cases} 1, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ \geq m, \ \text{for}\ 0 \leq t \leq \Delta t \\ 0, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ < m, \ \text{for}\ t < 0\ \text{or}\ t > \Delta t \end{cases}
\end{aligned}
$$

Similarly, the same interpretation can apply to the multi-input time-dependent *AND*$_n$-gate (for a minimum threshold of firing all *n* spikes to generating an output spike, which is already given by Eq. 7).

In most circumstances, neurons rarely require the strict condition that *all n* (tens of thousands) inputs need to be firing simultaneously in order to generate an output spike. The requirement is that only a majority *m* of the inputs are 1's (firing) rather than all *n* of them are 1's (firing). Thus, the condition in which a minimum of *m* spikes is required to produce an output spike is given by the generalized *AND*$_m$-gate (Fig. 1):

$$
\begin{aligned}
y(t) &= x_1(t)\ AND_m\ x_2(t)\ AND_m\ \ldots AND_m\ x_n(t) \\
&= AND_m\big(x_1(t),\ x_2(t),\ \ldots, x_n(t)\big) \qquad (12) \\
&= \begin{cases} 1, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ \geq m, \ \text{for}\ 0 \leq t \leq \Delta t \\ 0, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ < m, \ \text{for}\ t < 0\ \text{or}\ t > \Delta t \end{cases}
\end{aligned}
$$

XIII. Generalized Multi-Input Minimum Threshold Logic Gates (*Min*-Gate)

Comparing the last part of the equations Eq. 11 and 12, it is obvious that they are identical. This implies that the generalized *OR*$_m$-function (for a minimum threshold *m* inputs to fire) and *AND*$_m$-functions (for a minimum threshold *m* inputs to fire) are equivalent. More specifically, the conventional *OR*-gate is equivalent to the *OR*$_m$-function (Eq. 11), with the minimum threshold *m* set to 1 (Eq. 10). It requires a minimum of *one* input spike to fire an output spike (the equivalent of *OR*$_1$-gate).

Similarly, the conventional *AND*-gate is equivalent to the *AND*$_m$-function (Eq. 7), with the minimum threshold *m* set to *n* (Eq. 12). It requires a minimum of all *n* inputs to fire a spike before it will fire an output spike (the equivalent of *AND*$_n$-gate). This illustrates that multi-input *OR*$_1$-gate and *AND*$_n$-gate are the extreme ends of a continuum (with a threshold set at 1 and *n* minimum input spikes for *OR*$_1$-gate and *AND*$_n$-gate, respectively) in digital processing.

In general, only a minimum of *m* spikes at time *t* is sufficed to satisfy the condition, producing the most generalized multi-input logic gate, without specifying whether it is an *AND*-gate or *OR*-gate. This theoretical formulation provides a generic description of the multi-input logic gate operation, independent of specific *AND*-gate or *OR*-gate requirement, by adding a minimum of *m* input spikes as the threshold to fire an output spike by a *MIN*$_m$-gate:

$$
\begin{aligned}
y(t) &= x_1(t)\ MIN_m\ x_2(t)\ MIN_m\ \ldots MIN_m\ x_n(t) \\
&= MIN_m\big(x_1(t),\ x_2(t),\ \ldots, x_n(t)\big) \qquad (13) \\
&= \begin{cases} 1, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ \geq m, \ \text{for}\ 0 \leq t \leq \Delta t \\ 0, & \text{if} \sum_{i=1}^{i=n} x_i(t)\ < m, \ \text{for}\ t < 0\ \text{or}\ t > \Delta t \end{cases}
\end{aligned}
$$

Most importantly, the generalized *OR*$_m$ and *AND*$_m$ functions are interchangeable, thus eliminating the needs for designing a specific multi-input hardware logic gate for the generalized *OR*$_m$-gate or the generalized *AND*$_m$-gate. Both *OR*$_m$-gate and *AND*$_m$-gate can be replaced by an equivalent logic operation, *MIN*$_m$-gate, i.e.,

$$OR_n\text{-gate} = AND_1\text{-gate} \qquad (14)$$

and

$$AND_m\text{-gate} = MIN_m\text{-gate} \qquad (15)$$

That is,

$$OR_m\text{-gate} = AND_m\text{-gate} = MIN_m\text{-gate}. \qquad (16)$$

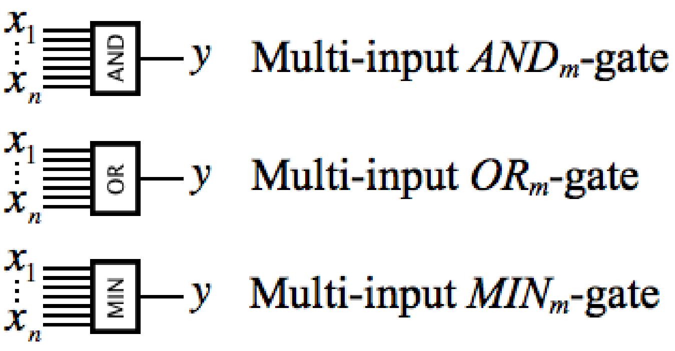
x_1
x_n
AND
OR
MIN
y
Multi-input *AND*$_m$-gate
Multi-input *OR*$_m$-gate
Multi-input *MIN*$_m$-gate

Figure 2. Multi-input logic equivalent gates for *AND*$_m$-gate, *OR*$_m$-gate and *MIN*$_m$-gate that require a minimum of *m* active input spikes to produce an output spike.

Fig. 2 shows the block diagrams for the equivalence of *AND*$_m$-gate, *OR*$_m$-gate, and *MIN*$_m$-gate. This provides a generic neural processer for a set of generalizable logic gates with a variable number of *m* (for either majority rule or minority rule logic operations).

XIV. Multi-Input Operation vs. Sequential Cascade 2-Input Operation

To illustrate the efficiency of multi-input operation for *AND*-gate, the simultaneous parallel operation of multi-input *AND*$_n$() function can be decomposed into a sequential cascaded 2-input *AND*-gate operation (see Fig. 1):

$$
\begin{aligned}
& AND\big(x_1(t),\ x_2(t),\ \ldots,\ x_n(t)\big) \\
&= \Big(\Big(\big(x_1(t)\ AND\ x_2(t)\big)\ AND\ x_3(t)\Big) \ldots\ AND\ x_n(t)\Big) \qquad (17)
\end{aligned}
$$

If there are a thousand inputs to be processed per neuron simultaneously, then implementing the computation using multiple cascaded 2-input *AND*-gates will lengthen the processing time by a thousand-fold.

For an AND_m-gate with a minimum *m*-input threshold, the logic gate design would be compounded by additional circuitry needed for testing the condition to see whether a minimum of *m* spikes is exceeded in each stage of the sequential cascade operation. This sequential cascade implementation of multi-input *AND*-gate is inefficient. It is computationally expensive in time and physically expensive in space.

XV. Majority/Minority Rule Voting System

The above neural processing is essentially a generalized vote counting system that processes input votes statistically:

- If $m \geq n/2$, it becomes a majority rule voting system.
- If $m < n/2$, it becomes a minority rule voting system.

That is, the requirement of a minimum of *m* spikes (from the *n* inputs) before firing an output spike by the neuron is essentially a vote counting system. As long as it has a majority of votes ($m \geq n/2$), independent of which input it is coming from, the processing (decision) of the neuron is to cast another vote of 1 as output to continue the process. Similarly, minority rule can be achieved by requiring less than half of the votes as the condition for forwarding the vote.

Even though VLSI voting digital circuitry had been implemented in hardware [5, 6, 8] and in bridging fault voting circuitry [7], such circuitry could have been simplified by a generalized multi-input *AND*-gate or *OR*-gate using spike code instead of binary code. Although there are many other neural logic circuitries implemented in hardware, most of them use binary logic rather than spike logic for processing. Examples of these binary logic implementations are numerous, which include neural threshold logic circuitry [20], neural floating gates [21-24], neural AND/OR gates [25], neural Boolean logic satisfiability model [26], comparison neural gate [27], neural circuit MNOS [28], and neutromatrix [29], with the exception of the electronic spiking network [30, 31].

XVI. Statistical Processing by Neurons

To extend the equivalent computations processed by spiking neurons, neurons can be shown to implicitly process signals statistically rather than deterministically. Statistical processing is performed by the vote counting process, without specifying which of its inputs is firing a spike, as long as a majority $m \geq n/2$ input has reached. Most significantly, it can even be used as a minority rule voting system if $m < n/2$ is satisfied statistically.

This illustrates the statistical nature of neural processing, which is a stochastic system; unlike most modern computers, which are deterministic system. The difference is that neural processing does not require absolute deterministic origin of where the spikes are coming from to fire its own output. The stochastic nature of neural processing lends itself in fault tolerance, which means that even if some of the inputs are unreliable, as long as other inputs can compensate for it, the statistical processing continues.

XVII. Example Computational Circuitry

Finally, as an example, the versatility of such generalized multi-input processing can be realized in the real world computation of the mathematical cross-correlation function [32]. It has be shown that, in a time-delayed neural network (TDNN) with successive delays of Δt to form multi-input for a neural network, it can compute the cross-correlation function of two sets of input streams by a hardwired digital neural circuitry [32]. This provides an example computation by a set of custom designed neural circuitry to perform high throughput signal processing in a massively parallel configuration. Other examples of computation can be designed to take advantage of the multi-input processing by these neural processors for integrating signals once the theoretical operations of these principles are implemented in hardware.

XVIII. Conclusions

Multi-input neural spike code processing can accomplish a set of equivalent digital signal processing of *OR*- and *AND*-gates. It provides the most generalized signal integration by allowing for a minimum of *m* spikes firing at time *t* to generate an output spike that encapsulates both *OR*-gate and *AND*-gate at the extreme ends of a minimum of *one* or all *n* active inputs set as threshold for *OR*-gate and *AND*-gate, respectively. Most importantly, self-contained spike-coded signals can eliminate the reliance on any external clock pulse for processing. This allows self-timed asynchronous massively parallel processing, accounting for transmission delay, without needing any extra transmission line (carrying the clock pulse signal) for every processor (logic gate) to perform its function. The generalized OR_m-function or AND_m-function provides a generic statistical voting system for majority rule or minority rule logic operations of the MIN_m-function. It performs statistical processing implicitly. This provides a theoretical description of the equivalence between digital processing (in computers) and multi-input neural spike code processing (in biological neurons). Massively parallel processing with 100,000 simultaneous inputs is typical in biological neurons, whereby similar large-scale implementations of multi-input processing can be achieved by these simple generalizable OR_m-gate, AND_m-gate and MIN_m-gates.

Acknowledgment

I would like to thank Ms. Krista Smith for the helpful suggestions, and for proofreading the manuscript.

References

[1] J. Davis, "An Introduction to Neural Networks," *Journal of Cognitive Neuroscience*, vol. 8, pp. 383-383, 1996/10/01 1996.

[2] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks, 1989. IJCNN., International Joint Conference on*, 1989, pp. 593-605 vol.1.

[3] D. Psaltis, A. Sideris, and A. A. Yamamura, "A multilayered neural network controller," *Control Systems Magazine, IEEE*, vol. 8, pp. 17-21, 1988.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*: Prentice Hall PTR, 1994.

[5] M. Radu, D. Pitica, and C. Posteuca, "Reliability and failure analysis of voting circuits in hardware redundant design," in *Electronic Materials and Packaging, 2000. (EMAP 2000). International Symposium on*, 2000, pp. 421-423.

[6] G. Yang, W. N. N. Hung, X. Song, and M. Perkowski, "Majority-based reversible logic gates," *Theoretical Computer Science*, vol. 334, pp. 259-274, 4/15/ 2005.

[7] S. D. Millman and J. M. Acken, "Special applications of the voting model for bridging faults," *Solid-State Circuits, IEEE Journal of,* vol. 29, pp. 263-270, 1994.

[8] D. B. Carlton, N. C. Emley, E. Tuchfeld, and J. Bokor, "Simulation Studies of Nanomagnet-Based Logic Architecture," *Nano Letters,* vol. 8, pp. 4173-4178, 2008/12/10 2008.

[9] W. M. Goodall, "Telephony by pulse code modulation," *The Bell System Technical Journal,* vol. 26, pp. 395-409, July 1947.

[10] H. S. Black and J. O. Edson, "Pulse Code Modulation," *American Institute of Electrical Engineers, Transactions of the,* vol. 66, pp. 895-899, 1947.

[11] H. Wold, "On stationary point processes and Markov chains," *Scandinavian Actuarial Journal,* vol. 1948, pp. 229-240, 1948/01/01 1948.

[12] W. Truccolo, U. T. Eden, M. R. Fellows, J. P. Donoghue, and E. N. Brown, "A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects," *J Neurophysiol,* vol. 93, pp. 1074-89, Feb 2005.

[13] S. Homma, T. Musha, Y. Nakajima, and Y. Okamoto, "Estimation of the rising phase of EPSP analyzed by computer simulation of the coding process," *Neurosci Res,* vol. 1, pp. 53-65, Feb 1984.

[14] M. A. Fitzurka and D. C. Tam, "A joint interspike interval difference stochastic spike train analysis: detecting local trends in the temporal firing patterns of single neurons," *Biol Cybern,* vol. 80, pp. 309-26, May 1999.

[15] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J Physiol,* vol. 117, pp. 500-44, Aug 1952.

[16] J. J. Hopfield and D. W. Tank, ""Neural" computation of decisions in optimization problems," *Biol Cybern,* vol. 52, pp. 141-52, 1985.

[17] J. J. Hopfield, D. I. Feinstein, and R. G. Palmer, "'Unlearning' has a stabilizing effect in collective memories," *Nature,* vol. 304, pp. 158-9, Jul 14-20 1983.

[18] D. W. Tank and J. J. Hopfield, "Neural computation by concentrating information in time," *Proc Natl Acad Sci U S A,* vol. 84, pp. 1896-900, Apr 1987.

[19] D. Marr, "A theory of cerebellar cortex," *J Physiol,* vol. 202, pp. 437-70, Jun 1969.

[20] R. Lashevsky, K. Takaara, and M. Souma, "Neuron MOSFET as a way to design a threshold gates with the threshold and input weights alterable in real time," in *Circuits and Systems, 1998. IEEE APCCAS 1998. The 1998 IEEE Asia-Pacific Conference on,* 1998, pp. 263-266.

[21] M. Holler, T. Simon, H. Castro, and R. Benson, "An electrically trainable artificial neural network (ETANN) with 10240 'floating gate' synapses," in *Neural Networks, 1989. IJCNN., International Joint Conference on,* 1989, pp. 191-196 vol.2.

[22] B. W. Lee, B. J. Sheu, and H. Yang, "Analog floating-gate synapses for general-purpose VLSI neural computation," *Circuits and Systems, IEEE Transactions on,* vol. 38, pp. 654-658, 1991.

[23] O. Fujita and Y. Amemiya, "A floating-gate analog memory device for neural networks," *Electron Devices, IEEE Transactions on,* vol. 40, pp. 2029-2035, 1993.

[24] P. Hasler and T. S. Lande, "Overview of floating-gate devices, circuits, and systems," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on,* vol. 48, pp. 1-3, 2001.

[25] H. K. Lam and F. H. F. Leung, "Design and Training for Combinational Neural-Logic Systems," *Industrial Electronics, IEEE Transactions on,* vol. 54, pp. 612-619, 2007.

[26] S. Chakradhar, V. Agrawal, and M. Bushnell, "Neural net and Boolean satisfiability models of logic circuits," *Design & Test of Computers, IEEE,* vol. 7, pp. 54-57, 1990.

[27] D. A. Durfee and F. S. Shoucair, "Comparison of floating gate neural network memory cells in standard VLSI CMOS technology," *Neural Networks, IEEE Transactions on,* vol. 3, pp. 347-353, 1992.

[28] J. P. Sage, K. Thompson, and R. S. Withers, "An artificial neural network integrated circuit based on MNOS/CCD principles," *AIP Conference Proceedings,* vol. 151, pp. 381-385, 5 June 2008.

[29] R. Melzack, "From the gate to the neuromatrix," *Pain,* vol. 82, Supplement 1, pp. S121-S126, 8// 1999.

[30] A. van Schaik, "Building blocks for electronic spiking neural networks," *Neural Netw,* vol. 14, pp. 617-28, Jul-Sep 2001.

[31] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, *et al.*, "Simulation of networks of spiking neurons: a review of tools and strategies," *J Comput Neurosci,* vol. 23, pp. 349-98, Dec 2007.

[32] D. Tam, "Theoretical Analysis of Cross-Correlation of Time-Series Signals Computed by a Time-Delayed Hebbian Associative Learning Neural Network," *The Open Cybernetics & Systemics Journal,* vol. 1, pp. 1-4, 2007.