Computing with Truly Asynchronous Threshold Logic Networks

Pekka Orponen
Institute for Theoretical Computer Science
Technical University of Graz
A-8010 Graz, Austria*

March 21, 1995

Abstract

We present simulation mechanisms by which any network of threshold logic units with either symmetric or asymmetric interunit connections (i.e., a symmetric or asymmetric "Hopfield net") can be simulated on a network of the same type, but without any a priori constraints on the order of updates of the units. Together with earlier constructions, the results show that the truly asynchronous network model is computationally equivalent to the seemingly more powerful models with either ordered sequential or fully parallel updates.

1 Introduction

A somewhat unsatifying feature of many otherwise interesting constructions of recurrent threshold logic networks (or, more generally, automata networks) is their use of a global synchronizing mechanism. It is commonly assumed that either the computational units in the network update their states fully synchronously in parallel (e.g. [8, 10, 11, 16, 24]), or there is some a priori imposed sequential update order (e.g. [2, 25]), or some intermediate form of the two applies (e.g. [7]). Such global timing constraints are clearly not consonant with the otherwise distributed nature of the model, where the behavior of each unit in other respects depends only on locally

*On leave from the Department of Computer Science, University of Helsinki, Finland. E-mail: orponen@igi.tu-graz.ac.at

available information. On the other hand, experience has shown that programming such networks without any assumptions on synchronization is rather awkward.

For instance, Goles et al. constructed in [8, 11] symmetric threshold logic networks whose transient times under parallel updates are exponential in the number of units in the network. Tchuente [25] and, independently, Bruck and Goodman [2] then came up with a simple method to simulate parallel updates by updates that are performed in a cyclic sequential order, yielding as an immediate corollary the existence of symmetric networks with exponentially long transients under ordered sequential updates. Proving the existence of long transients under *unordered* sequential updates is quite a bit more complicated, however, and seems to have been worked out first by A. Haken in a manuscript [12] which, unfortunately, remains unpublished. (On the other hand, the result is now known to follow, albeit via a somewhat indirect route, also from the general theory of local search for optimization problems [23], and the explicit construction of [12] is reviewed in [3], and also below.)

As another example, in [16] general scheme was presented for simulating polynomial space (resp. polynomial time) bounded Turing machines by symmetric polynomial size nets (resp. polynomial size nets with polynomially bounded connection weights). The construction in [16] appears to rely quite heavily on the use of parallel updates, and has not so far been directly generalized to unordered sequential updates[1].

In this paper, we outline a general scheme whereby the computation of any network using cyclic sequential updates can be simulated on a slightly larger network where the update order is totally unconstrained. More precisely, we shall discuss two schemes, a simple one for asymmetric networks, and a more complicated one for symmetric networks. Applying the Tchuente/Bruck–Goodman construction [25, 2], these results imply that also parallel updates can always be simulated in a fully asynchronous manner. And via the construction of [16], we obtain as a corollary an efficient simulation of

[1]Two comments on related work are in place here. First, simulations of space-bounded machines by small asymmetric networks were designed already by Lepley and Miller in [15], both for parallel and, remarkably, for random sequential updates — although the latter scheme is correct only in a probabilistic sense. The interest in showing that the simulation can be done also on *symmetric* nets is their generally very lsimited convergence behavior, as discussed by many authors [4, 6, 10, 13, 21]. A second somewhat related result is the recent simulation by Siegelmann and Sontag [24] of arbitrary Turing machines, and even more general computations by *fixed-size* parallel asymmetric networks with *continuous-state* units.

space- or time-bounded Turing machines on symmetric asynchronous networks.

The essential component of our simulation schemes is building an internal sequencing mechanism into network. In the case of asymmetric networks, sequencing is achieved fairly easily with the addition of some intermediate units; in the case of symmetric networks the arrangement is more complicated, and is based on using Haken's [12] exponential-transient network as a "clock".

For general surveys of automata networks, see the books [5, 9]. Threshold logic networks have recently become (again) popular as discrete models of neural networks. Computational aspects of these models are discussed in, e.g., the survey papers [17, 18, 26], and in the books [14, 19, 20, 22].

2 Preliminaries

A *threshold logic network* (or a "binary recurrent neural network") consists of n *threshold logic units* (or "binary neurons"), each of which is at a given moment in either one of two *states* $x_i = 1$ or $x_i = 0$, also called the *on* and *off* states. A unit i receives at each moment of time as input information the sum of the states x_j of all units, weighted by some local coefficients, or *connection weights* w_{ij}. When a unit i is allowed to *update*, it changes its state according to the rule

$$x_i \leftarrow \operatorname{sgn}(\sum_{j=1}^{n} w_{ij}x_j - \theta_i),$$

where sgn is the discrete step, or "Heaviside" function

$$\operatorname{sgn}(t) = \begin{cases} 0 & \text{if } t < 0, \\ 1 & \text{if } t \geq 0, \end{cases}$$

and θ_i is a local *threshold* value.

A network is *symmetric*, if $w_{ij} = w_{ji}$ for all i, j, and *asymmetric* if this assumption cannot be made. (Thus, symmetric nets are a special case of asymmetric ones). Symmetric threshold logic networks are often called "Hopfield nets" in reference to the paper [13], which was successful in drawing widespread attention to them.

The updates in a network may be arranged to occur either simultaneously in *parallel* for all the units, or *sequentially* one unit at a time. We shall say that a network is *asynchronous* if the update order of the units is not predetermined, and *synchronous* otherwise[2]. To simplify the presentation, we shall discuss the constructions below as if also in an asynchronous network only one unit would update at each time step. However, it is easy to check that the constructions do work also when arbitrary subsets of the units are updated in parallel. We shall also usually only consider the sequential update steps that actually change the state of the updated unit. Thus, e.g. "next update step" means the next step at which some updated unit rerally changes its value.

A global state, or *configuration* $x = (x_1, \ldots, x_n)$ of the network is *stable* if none of the units would change its state in an update. It is known that if all the diagonal weights w_{ii} are nonnegative, then a symmetric net will under any sequential update sequence eventually converge to a stable state [6, 13]. Under parallel updates a symmetric network, even one with negative diagonal elements, will always converge to either a stable state or to a cycle of two alternating states [11, 21]. In the article [4] upper bounds are derived on the number of update steps required for convergence, and all these results are reviewed, complete with proofs, in each of the surveys [3, 9, 14].

An elegant general device for simulating parallel updates by cyclically ordered sequential updates is the following technique of "doubling" the network [2, 25]. Given a network of n units with weights w_{ij} and thresholds θ_i, construct a new, bipartite network of $2n$ units with weights w'_{ij}, where $w'_{n+i,j} = w'_{i,n+j} = w_{ij}$ for $i, j = 1, \ldots, n$, and $w'_{ij} = 0$ otherwise. The thresholds in the new network are defined as $\theta'_i = \theta'_{n+i} = \theta_i$ for $i = 1, \ldots, n$. The idea of the construction is to have the nodes $i = 1, \ldots, n$ of the new network represent the original network at odd points of time, and the units $i = n + 1, \ldots, 2n$ at even points of time. Note that the only nonzero edges in the new network go from one side to the other. Parallel updates of the original network can now be simulated by updating the units in the doubled network sequentially in numerical order, since the updating of the units $1, \ldots, n$ does not interfere with their inputs from units $n+1, \ldots, 2n$, and vice versa. Each round of sequential updates in the doubled network corresponds to two parallel update steps in the original network.

As a special case, if the original network is symmetric, then so is the doubled network, and hence the sequential updates on the doubled network

[2]This use of the terms deviates from the custom in the automata networks literature, where the model of "truly" asynchronous networks is usually not considered, and so the term "synchronous" is used for our "parallel", and "asynchronous" for our "(ordered) sequential". However, for the purposes of this paper we need the more precise terminology.

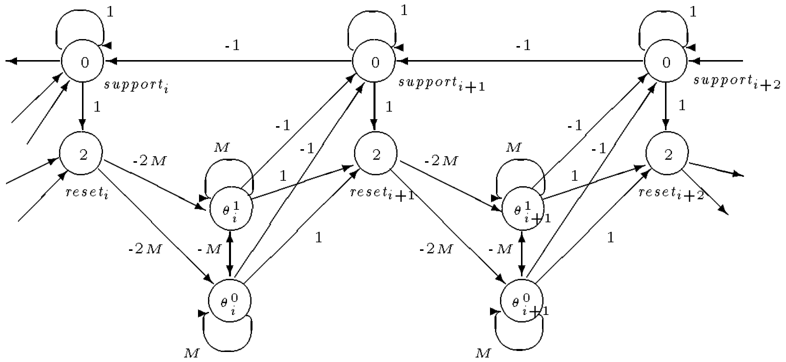

Figure 1: An asymmetric sequencing mechanism.

are guaranteed to converge to a stable state. It follows immediately that the original network converges under parallel updates to either a stable state or to a cycle of length two, depending on whether the two sides of the doubled network in its stable state are equal or not equal.

3 Asymmetric Nets

Let us consider first the simulation construction for asymmetric nets, as it is rather simpler than the one for symmetric nets. Thus, assume we are given an asymmetric network N of n units, whose fixed sequential update order we assume to be $x_1, x_2, \ldots, x_n, x_1, x_2, \ldots$. For simplicity, we also assume that he units in N should also have no self-connections, i.e. all the diagonal weights are zero[3]. We shall construct another asymmetric net N' of $4n$ units whose computations mimic those of N, but that is "self-sequencing", in the sense that at each moment of time there is at most one unit in N' whose state would change in an update.

The construction of N' from N is outlined in Figure 1. Each unit i in N is represented in N' by two units i^1 and i^0, whose activities complement each other so that $x_i^1 = 1$ and $x_i^0 = 0$ whenever $x_i = 1$, and $x_i^1 = 0$ and $x_i^0 = 1$ whenever $x_i = 0$. The advantage of such a twin-unit representation is that always exactly one of the twins will be on and the other off — except

[3]As N is asymmetric, this requirement is easy to satisfy by insrting an extra unit into any such connection. For symmetric nets the issue is somewhat more complicated; see below.

momentarily during update when they will both be off. The weight vectors and threshold values in N' are defined accordingly, so that if the update rule for unit i in network N is

$$x_i \leftarrow \mathrm{sgn}(\sum_{j=1}^{n} w_{ij} x_j - \theta_i),$$

then the update rules for the two corresponding units in N' are

$$\begin{aligned} x_i^1 &\leftarrow \mathrm{sgn}\left(\sum_{j=1}^{n} w_{ij} x_j^1 - \sum_{j=1}^{n} w_{ij} x_j^0 - (2\theta_i - \sum_{j=1}^{n} w_{ij})\right), \\ x_i^0 &\leftarrow \mathrm{sgn}\left(-\sum_{j=1}^{n} w_{ij} x_j^1 + \sum_{j=1}^{n} w_{ij} x_j^0 + (2\theta_i - \sum_{j=1}^{n} w_{ij})\right). \end{aligned}$$

To these rules must be added the effects of the control mechanism in N'. (Cf. Figure 1, which shows the twin pairs in N' corresponding to two consequently updated units i and $i+1$ in N, and their associated control units. The figure in fact shows *only* the control inputs to the units; regular interconnections have been left out to avoid cluttering the picture.)

Let M be an integer such that $M > \max\{\sum_{j=1}^{n} 2|w_{ij}| + 2|\theta_i| : i = 1, \ldots, n\}$. As can be seen in Figure 1, there is a self-connection of weight M at each $i^{0/1}$ unit, plus connections of weight $-M$ in both directions between any pair of twin units, plus connections of weight $-2M$ to each $i^{0/1}$ unit from a preceding *reset* control unit. The purpose of these heavy connections is to maintain the states of any twin pair of units until the associated *reset* unit becomes active and sets them both to state 0. Feeding into the *reset* units is another set of *support* control units.

Let us now see how computation proceeds in N'. The network is initialized according to the initial state of N, i.e. one unit in each twin pair is set to state 1 and the other to state 0, and all the *reset* and *support* units are initialized to state 0. It can be seen that this is a stable global state of the network. Assuming the update sequence of network N begins with unit i, the computation of N' is started by turning the unit $support_i$ on. The only possible (cf. Figure 1) consequent event in the network is for the unit $reset_i$ to turn on, which will in turn lead to both of the units in the i twin pair being turned off. Because now the i twins are no longer inhibiting the unit $support_{i+1}$, the latter is activated. (This, however, does not yet lead to the activation of $reset_{i+1}$, as that would require also positive support from either one of the i twins.)

Continuing, one can see that again the only possible consequent events are for first $support_i$, and then also $reset_i$ to turn off. At this point, finally, neither one of the i twins receives any input via any of the control connections, and so they are ready to accept input from the other twin pair units in the network. After one of the twins has been activated, the state of the pair freezes again (because of the M connections), but now the active member of the pair provides enough support for the unit $reset_{i+1}$ to be activated, and the cycle repeats at site $i+1$.

Without any further devices, this computation would obviously continue forever (we are naturally assuming that the sites are cyclically linked, so that site n activates site 1); thus, network N' it is not a faithful simulation of network N in the sense of preserving convergence. However, if N is performing some actual computation (as in, e.g. [16]), then one can usually have the activation of some specific unit h indicate the end of the computation. By leaving out the connection in N' from the positive member of the h twin pair to $reset_{h+1}$ one can make also N' converge immediately upon activation of h^1.

The following theorem summarizes the discussion:

Theorem 1 *Let N be an asymmetric network of n units with no self-connections, and with a cyclic sequential update order. Then there is another asymmetric network N' of $4n$ units, with no constraints on the update order, such that any computation performed by N in t changing update steps can also performed by N' in at most $6nt$ changing update steps.*

4 Symmetric Nets

Haken's asynchronous counter network

Let us then turn to the more complicated case of symmetric nets. We shall begin by reviewing Haken's [12] asynchronous exponential-transient network, as it is a central component of our construction.

The basic idea of Haken's network is to build a binary m-bit counter, with a convergence time of $\Omega(2^m)$, out of $O(m)$ units. The counter network is composed of smaller *xor* subnetworks, presented in Figure 2. It can be seen that the z unit of this device is eventually set to state 1 if the x and y units are maintained in opposite states, and to state 0 otherwise.

The counter is constructed by connecting m such *xor* devices in a sequence as illustrated in Figure 3. Each bit b_i, $i = 0, \ldots, m-1$, of the

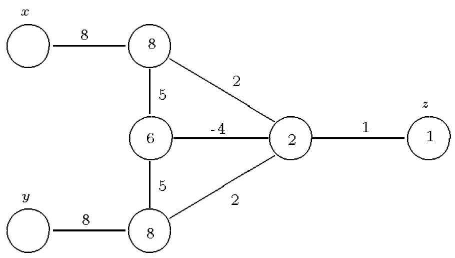

Figure 2: A symmetric device for computing the *xor* of states x and y.

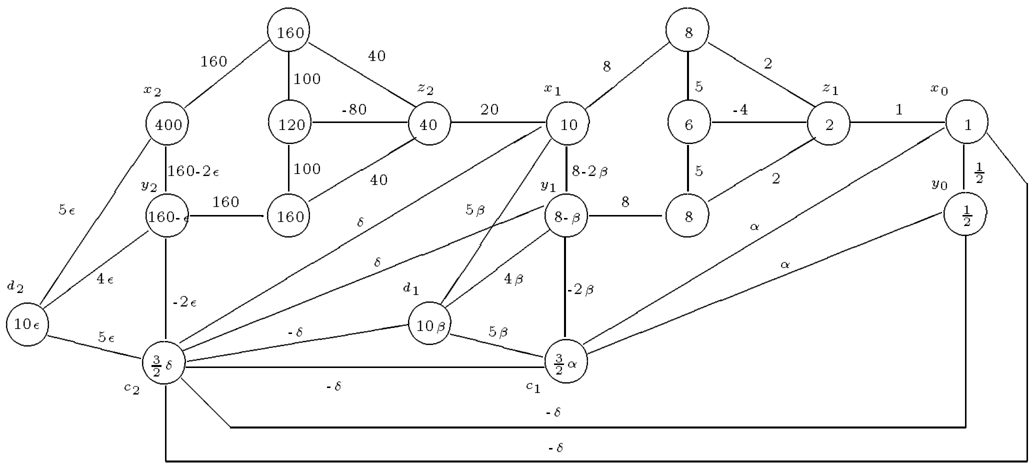

Figure 3: A symmetric three-bit counter network.

counter is represented by a pair of units x_i, y_i, with the interpretation that b_i has value 1 if $x_i = y_i$ (or equivalently $z_i = 0$, for $i \geq 1$). In the figure, the high-order units are to the left.

The counter is is initialized so that all the units in the top row (including all the x_i and z_i units) are in state 1, and all the other units are in state 0; this corresponds to all the bits b_i being initially 0. The first update step will then turn unit y_0 from state 0 to state 1, representing a flip of bit b_0 from 0 to 1.

The behavior of the network obeys the following rules: the state of each unit y_i gets incremented from 0 to 1 (i.e. bit b_i gets value 1), when $x_{i-1} = y_{i-1} = 1$ and $x_j = y_j = 0$ for all $j < i-1$. This change at y_i causes, via the *xor* line, x_{i-1} being reset back to 0, and each x_j, $j < i-1$, being set to 1 (i.e. all the lower order bits b_j, $j \leq i-1$, are reset from 1 to 0). Eventually, again, also y_0 gets value 1, and the counting resumes at the low order bits. Then when counting reaches back to bit b_{i-1}, this time with $x_{i-1} = 0$, $y_{i-1} = 1$, and $x_j = y_j = 0$ for each $j < i-1$, unit y_{i-1} gets decremented from 1 back to 0 (thus setting b_{i-1} to 1), and again each of the lower order x_j's gets set to 1 (corresponding to each b_i getting reset to 0).

To achieve this relatively complicated sequencing, each pair of counter units x_i, y_i, $i \geq 1$, has two associated control units c_i, d_i, all initially 0. These units are first activated when the first condition above, for turning unit y_i from 0 to 1, is achieved, and help y_i to make the state change. Node y_i will then maintain its new state supported by its right neighbor in the local *xor* structure until the control units are activated again, this time by the condition for resetting y_i to 0, which they again help y_i to do. (The difference to the first time is that now y_i gets no support from x_i, which has meanwhile been reset.)

The functioning of this control structure is perhaps most easily understood by simply simulating the behavior of the sample network in Figure 3. The weights $\alpha \gg \beta \gg \delta \gg \epsilon$ need to be chosen so that control information can only flow from right to left, i.e., that no combination of states to the left of some control unit can have an effect on the updates at that unit. Denoting $\alpha = \alpha_1$, $\delta = \alpha_2$, $\beta = \beta_1$, $\epsilon = \beta_2$ etc., Haken [12] suggests the values $\alpha_k = 1/(40m)^k$, $\beta_k = m/(40m)^{(k+1)}$ for an m-bit counter. The number of units in an m-bit counter can be seen to be $8m - 6$.

The symmetric simulation

Let then N be a symmetric network of n units that are updated sequentially in the order $x_1, x_2, \ldots, x_n, x_1, x_2, \ldots$. Again, we shall momentarily assumethat the units in N have no self-connections, and return to the problem of implementing these later[4]. Since each of the units in the networks can be in two alternative states, the network can compute for at most 2^n update cycles, i.e. $t = n2^n$ update rounds, before it either converges or repeats a configuration and goes into a cycle. (Actually, cycles are possible only in networks with negative weight self-connections, as otherwise the theorems in [6, 13] guarantee convergence to a stable state.) We shall use a counter network of $m = \lceil \log_2 n2^n \rceil + 2$ bits to construct an asynchronous network N' that simulates N for t steps. The counter network acts as an asynchronous "clock" used to sequence the updates in N'.

As in the asymmetric case, the network N' always runs its full course even if N converges fast. However, the simulation can again be made more faithful if N contains some specific unit h to indicate the termination of the computation. One can then force also N' to converge immediately upon the activation of h by leaving out the appropriate positive connection from N'. (To be precise, this would be the connection from unit h^1 to unit $done_h$, as described below.) Also, if one knows that N is going to converge in t update steps, one can use a counter network of only $\lceil \log_2 nt \rceil + 2$ bits for N'.

The details of the construction of N' are illustrated in Figure 4. As in the asymmetric simulation, each unit i of N is duplicated in N', and to each pair of twin units i^1, i^0 is attached a system of four control units, labeled $equal_i$, $reset_i$, $primed_i$, and $done_i$. The weights and the thresholds of the $i^{0/1}$ units are set exactly as in the asymmetric simulation, except for the obvious modifications required by the differences in the control structure.

Let again M be an integer such that $M > \max\{\sum_{j=1}^{n} 2|w_{ij}| + 2|\theta_i| : i = 1, \ldots, n\}$. Similarly to the asymmetric simulation, there is for the purpose of controlling the update times of the $i^{0/1}$ units a self-connection of weight M at each $i^{0/1}$ unit, plus a connection of weight $-M$ between any pair of twin units, plus a connection of weight $-2M$ between any $i^{0/1}$ unit and a preceding $reset_i$ unit. As can be seen in the figure, there are also other control connections, whose purpose will become clear momentarily.

The clock subnetwork controls the sequencing mechanism via the $equal_i$

[4] As an important special case, note that networks that are obtained via the doubling construction never have self-connections. Thus, for asynchronous sequential simulation of parallel updates one does not need to consider this complication.

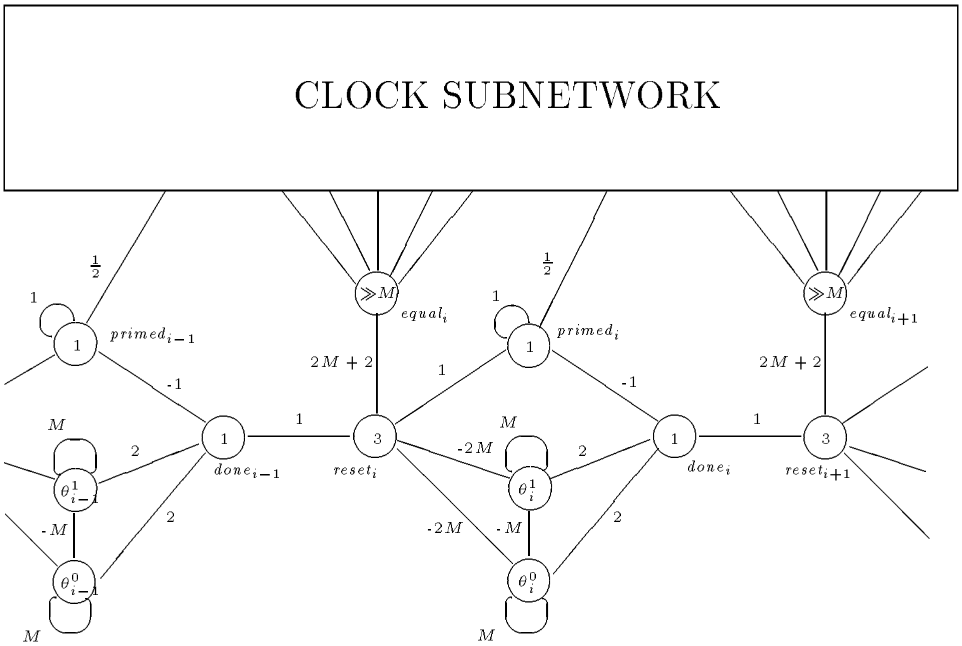

Figure 4: A symmetric sequencing mechanism.

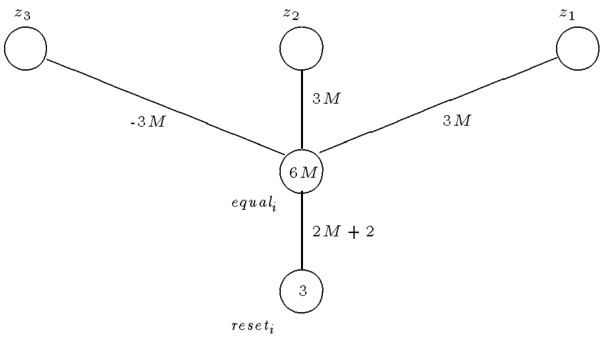

Figure 5: A network testing for $b_3 b_2 = 10$ and $z_1 = 1$.

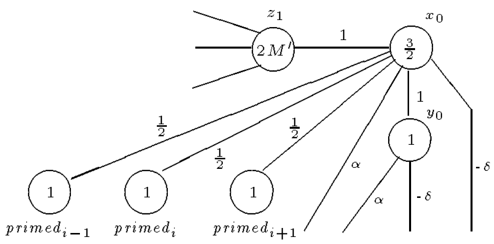

Figure 6: A latch mechanism for controlling the clock.

control units as follows. Each $equal_i$ unit is connected to the $\lceil \log_2 n \rceil + 1$ low-order z_j units in the clock, from $j = 1$ to $j = \lceil \log_2 n \rceil + 1$, in such a manner that $equal_i$ can be activated only when the binary number encoded by the clock bits $b_{\lceil \log_2 n \rceil + 1} \ldots b_2$ equals i, and in addition $z_1 = 1$. A subnetwork for implementing the equality test for 2+1 bits and $i = 2 = 10_2$ is illustrated in Figure 5. (In interpreting the figure, keep in mind that the states of the z_j units in the clock correspond to *complements* of the bits b_j.)

Note how the weights and thresholds in the equality testing network are chosen sufficiently large so that the connection from the $reset_i$ unit does not affect the test. In fact, also the weights and thresholds in the clock network *itself* must be made so large that the test networks do not have an effect on its behavior. Multiplying all the "large" weights in the clock network by $M' = 3nM$ suffices for this; the small control weights α_j, β_j and the units x_0, y_0 are not affected, as can be affirmed by a brief look at Figure 3.

The lowest-order bits in the clock are used as a latch that connects advancing the clock to progress in the computation, in the following manner (cf. Figure 6). When both of the units x_0 and y_0 are off, counting on the clock cannot proceed until both unit z_1 and one of the units $primed_i$ are on. (We are assuming at the moment that only one of the $primed_i$ unit can be on at each time. This will be seen to be true in the following.) After this activation condition has been reached, counting can proceed unhampered — and both x_0 and y_0 will be maintained in state 1, assuming $primed_i$ stays on — until eventually unit z_1 gets turned off. Then further progress requires that also $primed_i$ is turned off. Assuming that subsequently this happens, and assuming the condition $primed_i = 0$ is then maintained, eventually both x_0 and y_0 will be turned off, and counting continues from the $x_0 = y_0 = 0$ configuration until it is again halted in the configuration $x_0 = y_0 = 0$, $z_1 = 1$, and $primed_i = 0$ for all i.

Let us then see how this coroutine-like controlling of the update sequence by the clock, and the clock by the update sequence, helps make the asynchronous simulation work. If the update sequence for the simulated network N begins with unit i, the asynchronous network N' is initialized so that the unit pairs j^1, j^0 correspond to the initial state of N, all the $equal_j$, $reset_j$, and $primed_j$ units are off, the $done_j$ units are on, and the clock encodes the value i (i.e., the clock network looks as if it had been started from the all-0's initial state, and let run freely for $4i$ cycles). In particular, we are assuming that unit z_1 is on, and the x_0 and y_0 units are off.

It can be seen that the only possible changing update in this configuration is to change the state of unit $equal_i$ from 0 to 1. This will then lead first to the turning on of unit $reset_i$, and then to the turning off of whichever member of the twin pair i^1, i^0 initially was on. (Note that all the time there is always just one possibility for the next changing update step. In particular, the configuration of the clock subnetwork stays stable.) Now there is no longer enough support to maintain the $done_i$ unit in state 1, and so it will be turned off. This, on the other hand, then gives the $primed_i$ unit the opportunity to turn on, after which the update events move inside the clock subnetwork.

The activation of the $primed_i$ unit releases the latch mechanism, and the clock moves until it is again caught by the latch, this time in the configuration $z_1 = 0$, $x_0 = y_0 = 1$, $primed_i = 1$. When unit z_1 was set to 0, however, unit $equal_i$ lost its support, and will thus subsequently be turned off. (Note, on the other hand, that unit $equal_{i+1}$ is not yet supported: this requires another revolution of the clock.)

After $equal_i$ turns off, unit $reset_i$ loses its support and will also be turned off. Now the twin units i^1, i^0 are no longer receiving any control input, and are free to update their states based on the states of the other twin pairs in the network — and in fact must do so in order for the computation to continue. After one of the twins has been activated again, the state of the pair freezes, but the active member of the pair provides enough support for the unit $done_i$ to turn on. This, on the other hand, leads to the turning off of unit $primed_i$, and to the consequent release of the latch mechanism in the clock. The clock then revolves to its next blocked position, with $z_1 = 1$, $x_0 = y_0 = 0$, and support for unit $equal_{i+1}$, after which the update cycle repeats at site $i + 1$.

We summarize the above discussion in a theorem:

Theorem 2 *Let N be a symmetric network of n units with no self-connections,* *and with a cyclic sequential update order. Then for any time bound t there is another symmetric network N' of* $8\log_2 t + 6n + 8\log_2 n + 10$ *units, with no constraints on the update order, such that any computation performed by N in t changing update steps can also performed by N' in at most* $38(n-1)t$ *changing update steps.*

Proof. The construction of N' from N has been presented above. To obtain the size bound on N', note that the clock requires $8(\lceil\log_2 nt\rceil + 2) - 6$ units, and the rest of the sequencing/simulation network another 6n units, for a total of $8\log_2 t + 6n + 8\log_2 n + 9$ units.

For the time bound, it easy to see that each simulation step requires 10 updates outside of the clock network. Analyzing the time used by the clock requires a little more care, but one can see that each flip of an *xor* structure from 0 to 1 or 1 to 0 requires 6 updates, and there will be at most $4T$ such flips altogether in a simulation of length T. (Of these, $2T$ flips come from the lowest-order *xor* alone, which makes 2 flips per each simulated step; the remaining $2T$ accounts for all the higher-order *xor*'s, amortized over the length of the computation.) To this must be added the activity of the x_0 and y_0 units, which both change state twice per each simulated step. Summing up, this makes for $38T$ update steps per T simulated steps. In the worst case where the actual changing update order of N is $x_n, x_{n-1}, \ldots, x_1, x_n, \ldots$, the length of the simulation can be $T = (n-1)t$. □

Networks with self-connections

Let us then discuss briefly the problem of simulating units with nonzero self-connections. A positive self-connection of weight w at a unit i presents no difficulty, as this can be replaced by a unit i' , to be updated immediately after unit i in the sequential order, with connection of weight w to unit i (and no other connections), and with threshold $w/2$. Negative self-connections are rather more difficult to handle, and we have to resort to the sequencing mechanism of the asynchronous simulation to deal with them. (Note that no local transform of the synchronous network can remove negative self-connections, because this would entail changing the convergence behavior of the network.)

Let then i be a unit in the synchronous network with a negative self-connection of weight $-w$. The basic idea is to replace unit i with the sub-network presented in Figure 7, where the units i and i' are scheduled to be cleared and updated during the *same* simulated update step, and unit

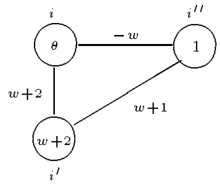

Figure 7: Device outline for simulating a negative self-connection.

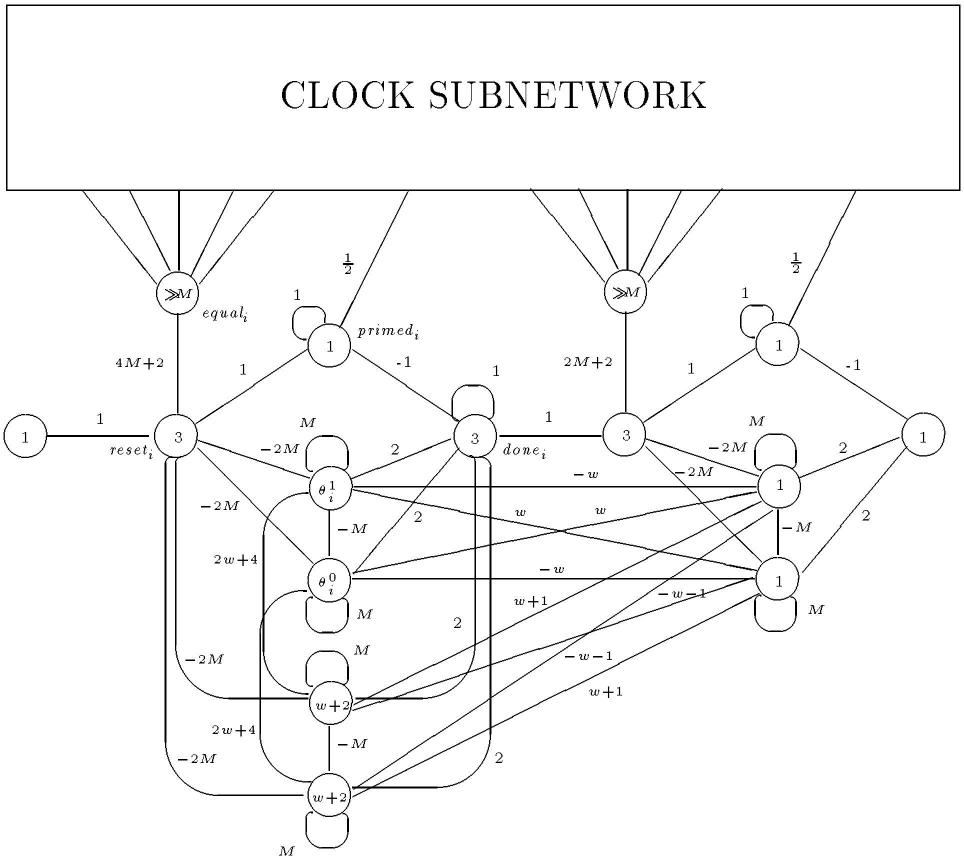

Figure 8: Full device for simulating a negative self-connection.

i'' during the immediately following simulated step. The full asynchronous construction is presented in Figure 8. It can be seen that in this modification of the sequencing mechanism from Figure 4, the unit $done_i$ can only turn off when both of the twin pairs corresponding to units i and i' have been reset, and conversely it can only turn back on when it again gets input from each of the pairs. Also, it can be seen that after the i and i' pairs have been reset, the i' pair will stay in the off state until one of the i units has been set to state 1. In particular, this means that the i' units have no effect on the setting of the i units.

5 Conclusion

We have presented constructions whereby computations performed by either asymmetric or symmetric threshold logic networks with either parallel or ordered sequential updates can be simulated on totally asynchronous networks of the same type. The simulations are not faithful to the convergence behavior of the original networks: terminating in time proportional to that required by the original network requires an explicit termination signal. In most cases, however, such a signal can be readily obtained. The existence of asynchronous simulations that reproduce also this aspect of the simulated networks' behavior remains an open problem.

References

[1] Anderson, J. A., Rosenfeld, E. (eds.) *Neurocomputing: Foundations of Research.* The MIT Press, Cambridge, MA, 1988.

[2] Bruck, J., Goodman, J. W. A generalized convergence theorem for neural networks. *IEEE Trans. Information Theory 34* (1988), 1089–1092.

[3] Floréen, P., Orponen, P. Complexity Issues in Discrete Hopfield Networks. Report A–1994–4, Dept. of Computer Science, University of Helsinki, September 1994. To appear in [20].

[4] Fogelman, F., Goles, E., Weisbuch, G. Transient length in sequential iterations of threshold functions. *Discr. Appl. Math. 6* (1983), 95–98.

[5] Fogelman Soulié, F., Robert, Y., Tchuente, M. *Automata Networks in Computer Science: Theory and Applications.* Manchester University Press, 1987.

[6] Goles, E. Fixed point behavior of threshold functions on a finite set. *SIAM J. Alg. Discr. Methods 3* (1982), 529–531.

[7] Goles Chacc, E., Fogelman Soulié, F., Pellegrin, D. Decreasing energy functions as a tool for studying threshold networks. *Discr. Appl. Math. 12* (1985), 261–277.

[8] Goles, E., Martínez, S. Exponential transient classes of symmetric neural networks for synchronous and sequential updating. *Complex Systems 3* (1989), 589–597.

[9] Goles, E., Martínez, S. *Neural and Automata Networks.* Kluwer Academic, Dordrecht, 1990.

[10] Goles, E., Olivos, J. Comportement périodique des fonctions à seuil binaires et applications. *Discr. Appl. Math. 3* (1981), 93–105.

[11] Goles, E., Olivos, J. The convergence of symmetric threshold automata. *Info. and Control 51* (1981), 98–104.

[12] Haken, A. Connectionist networks that need exponential time to converge. Unpublished manuscript, Dept. of Computer Science, University of Toronto, January 1989. 10 pp.

[13] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA 79* (1982), 2554–2558. Reprinted in [1], pp. 460–464.

[14] Kamp, Y., Hasler, M. *Recursive Neural Networks for Associative Memory.* John Wiley & Sons, Chichester, 1990.

[15] Lepley, M., Miller, G. Computational power for networks of threshold devices in an asynchronous environment. Unpublished manuscript, Dept. of Mathematics, Massachusetts Inst. of Technology, 1983.

[16] Orponen, P. On the computational power of discrete Hopfield nets. In: *Proc. 20th International Colloquium on Automata, Languages, and Programming (Lund, Sweden, July 1993).* Lecture Notes in Computer Science 700. Springer-Verlag, Berlin, 1993. Pp. 215–226.

[17] Orponen, P. Computational complexity of neural networks: A survey. *Nordic Journal of Computing 1* (1994), 94–110.

[18] Parberry, I. A primer on the complexity theory of neural networks. In *Formal Techniques in Artificial Intelligence: A Sourcebook* (ed. R. B. Banerji). Elsevier, Amsterdam, 1990. Pp. 217–268.

[19] Parberry, I. *Circuit Complexity and Neural Networks.* The MIT Press, Cambridge, MA, 1994.

[20] Parberry, I. (ed.) *The Computational and Learning Complexity of Neural Networks: Advanced Topics.* The MIT Press, to appear.

[21] Poljak, S., M. Sůra, M. On periodical behaviour in societes with symmetric influences. *Combinatorica 3* (1983), 119–121.

[22] Roychowdhury, V., Siu, K.-Y., Orlitsky, A. (eds.) *Theoretical Advances in Neural Computation and Learning.* Kluwer Academic, Boston, MA, 1994.

[23] Schäffer, A. A., Yannakakis, M. Simple local search problems that are hard to solve. *SIAM J. Computing 20* (1991), 56–87.

[24] Siegelmann, H. T., Sontag, E. D. Analog computation via neural networks. *Theoretical Computer Science 131* (1994), 331–360.

[25] Tchuente, M. Sequential simulation of parallel iterations and applications. *Theoretical Computer Science 48* (1986), 135–144.

[26] Wiedermann, J. Complexity issues in discrete neurocomputing. In *Aspects and Prospects of Theoretical Computer Science.* Lecture Notes in Computer Science 464. Springer-Verlag, Berlin, 1990. Pp. 480–491.