Implementation of Reduced Precision Integer Epigenetic Networks in Hardware

Dr. Andrew Walter
Department of Computer Science
University of York
York, UK
andrew.walter@york.ac.uk

Dr Simon Bale
Department of Electronic Engineering
University of York
York, UK
simon.bale@york.ac.uk

Prof Andy Tyrrell
Department of Electronic Engineering
University of York
York, UK
andy.tyrrell@york.ac.uk

***Abstract*—This paper details the development of a resource efficient implementation of the Artificial Epigenetic Network (AEN) concept, based on reduced precision integer mathematics, and the translation of this implementation into hardware via a Field Programmable Gate Array (FPGA) to provide improvements in resource utilisation and execution speed while not sacrificing the unique benefits provided by the epigenetic mechanism. Validation of the implementation's performance on the inverted pendulum task is obtained and compared to that of previous AENs, as well as experiments to determine how far the precision of the network may be reduced while still maintaining an acceptable degree of performance.**

Keywords—Artificial Epigenetic Networks, Field Programable Gate Array, Reduced Precision Computation, Digital Hardware.

I. INTRODUCTION

As with many bio-inspired computing concepts, Artificial Epigenetic Networks (AENs), devised by Dr. Alex Turner, seeks to leverage the strengths of biological mechanisms such as adaptability and resilience to dynamic change [1] for engineering applications. However, the original AEN architecture is very computationally intensive to implement due to a number of factors such as its utilisation of floating-point mathematics and modelling of biologically accurate elements that are unnecessary and computationally inefficient [2]. This paper described a more resource efficient version of the AEN concept, based on reduced precision integer mathematics and the streamlining of the original biological model, while keeping the fundamental bio-inspired advantages of the original AEN. Furthermore, this paper also details a hardware implementation of this kind of AEN on a Field Programable Gate Array (FPGA), intended to bring additional resource utilisation improvements and further expand possible use cases.

II. ARTIFICAL EPIGENETIC NETWORKS

The AEN expands upon the concepts of Artificial Gene Regulatory Networks (AGRNs): a computational paradigm inspired by the mechanisms that control the expression or suppression of genes within a biological genome [3]. The first iteration, referred to as Artificial Epigenetic Regulatory Networks (AERNs), simply added a series of Boolean switches to the genes in a normal AGRN. When active, these switches suppressed the activity of their associated genes, altering the behaviour of the network, as illustrated in Fig 1.

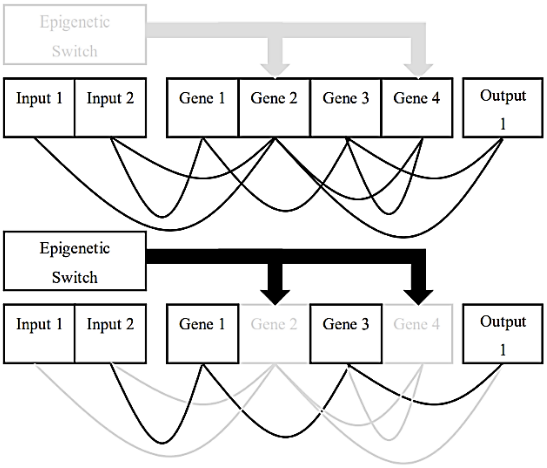

Fig. 1. An AERN with epigenetic switch inactive, top, and active, bottom. Note how the activation of the epigenetic mechanism, and hence suppression of the genes, dramatically changes the characteristics of the network.

Full AENs replace the Boolean switches with computational elements similar to the genes themselves, effectively allowing the network to alter its own structure in response to varying stimuli by suppressing the activity of different parts of itself. The original work carried out by Turner demonstrated that this topological self-modification allowed AENs to outperform their AGRN counterparts in various control tasks, specifically: navigating a Chirikov's standard map; controlling single and multiple coupled inverted pendulums; and the control of transfer orbits in gravitational systems [4].

A. *Shortcommings of the Original Networks*

As indicated in the introduction to this paper, the original AEN architecture is computationally intensive, which limits its potential applications and makes the process of creating a dedicated hardware implementation more difficult. This computational overhead comes from two factors: first, the use of high precision floating point mathematics as the basis for network; and second the modelling of various elements that, while biologically accurate, bring either no benefit to the network's computational abilities, or are an active detriment to its efficacy. To address these issues, a new version of the AEN was designed that streamlined the network model and reduced computational overhead by switching to an integer mathematics based approach. This switch also allowed for the easy employment of reduced precision in the network's calculations, bringing further improvements.

978-1-7281-9048-8/21/$31.00 ©2021 IEEE

III. Architectural Alterations

Starting with the changes to the architecture of the network, there are three elements to the original AEN model that are biological accurate but computationally unnecessary.

A. *Removal of Gene to Protein Transcription*

In biological systems, the genome stores information which is then used to create proteins that perform useful work. The original AENs replicated with mechanism, having a gene network that was then translated at each time step into a so-called protein network to actually perform computations, with the translation process being controlled by the epigenetic elements, referred to as Epigenetic Molecules or just Molecules. The pseudo-code below illustrates this process.

```
for number of epigenetic molecules
        Execute epigenetic molecule
        for number of genes
                if gene is suppressed by molecule
                        mark gene
clear the current protein network
for number of genes
        if gene is not marked
                Copy to the protein network
for number of genes in protein network
        Execute gene
        Copy state back to gene network
```

This not only introduces additional steps that must be performed each time the network operates, but it increases the storage needs by requiring two versions to the network in memory. The new version of the architecture presented here replaces this with a simple Boolean flag within each gene. When the epigenetic molecules update the activity of the genes, these flags are altered accordingly (TRUE, active, by default; FALSE, inactive, if set by a molecule). When the genes are executed, only those whose flags are set TRUE have their expressions updated. In addition, only active genes will have their outputs used as part of the weighted sum of other active genes. The pseudo-code below illustrates this new process.

```
set all genes flags to TRUE
for number of epigenetic molecules
        Execute epigenetic molecule
        for number of genes
                if gene is suppressed by molecule
                        set gene flag FALSE
for number of genes
        if gene flag is TRUE
                Execute gene
```

B. *Switching to Input Applied Weights*

Rather than the traditional approach of having each processing element (gene, neuron etc) have a different set of weights that are applied to their inputs, the original AENs employed a system where all connections drawn from a particular gene would have the same weight, regardless of which genes they were serving as inputs for, by applying the weights at the output of each gene. While this does reduce the complexity of the network, as well as the evolutionary processes, it is an uncommon approach that limits the potential functionality of the network; for example, two different genes, all other things being equal, cannot react to the actions of a third in different ways. With this in mind, a new multiple weight system was implemented, although it does come with a problem of its own: as the number of connections each gene possesses changes, both during the evolutionary process and execution, the number of weights also changes. Therefore each gene holds a number of weights equal to the maximum number of possible connections it could have (conveniently equal to the number genes plus the number of inputs). These weights are treated as being directly mapped to a particular possible connection, so when gene A uses gene B as an input, it uses weight B.

C. *Introduction of Dedicated Input/Output Elements*

Finally, instead of possessing dedicated input/output elements as part of the network architecture, the original AENs instead mapped inputs/outputs to genes within the network at run time. Each gene possesses an input and output number, which functioned as location values within two separate 1-dimentional regions. Each of these regions was then divided up into partitions, one for each input, or output, with any space left over being ignored. In a correctly functioning network, each Input/Output region would therefore have at least one gene within it, although this is not always the case as Fig 2 illistrates.

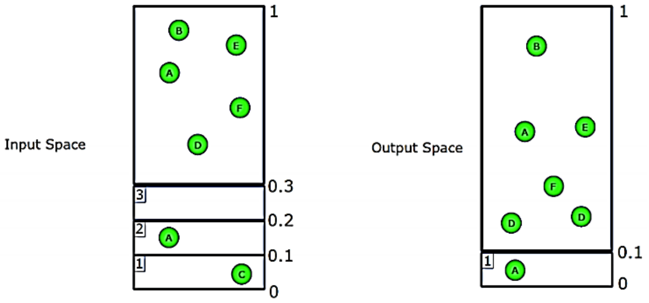

Fig. 2. The input and output spaces of an example network. This network has 7 genes, 3 inputs and 1 output. Note the difference in gene position between the two regions, as well as the fact the input 3 has no gene mapped to it at this time. Figure taken from [5].

With outputs, the output value of the first translated gene within the partition is used and fed out of the network. The method for inputs is a little more complex, as the external input value replaces the expression of the first translated gene within the partition. In effect, this injects the input value into the normal network space at the location of the replaced gene. This system has been replaced with a set of dedicated Input/Output elements, with inputs connecting to genes throughout the network as if they were other genes; and outputs taking the value of the first active gene they are connected to. This new system ensures there is no reduction in the network's capacity due to the re-tasking of genes as inputs. Additionally, it also allows for more of the network's functionality to be encapsulated with discreet units (the Input/Output elements), rather than relying on an overall algorithm, a feature that becomes more relevant during the translation to hardware.

IV. Translation to Interger Mathematics

Turning now to the switch from floating-point based networks to integer-based ones. The benefits of this are twofold: firstly, integer values, and their corresponding mathematical operations, are significantly easier to implement in digital hardware. Field Programmable Gate Array (FPGA) manufacturers do produce various IP cores intended to streamline the use of floating-point maths, such as the Xilinx LogiCORE Floating-Point Operator [6] however, using these increases the hardware footprint of the network, which goes against the ethos of reducing resource utilisation.

This leads to the second reason for switching to an integer network: ease of reducing the network's data width. In the same way that integer mathematics consume less resources than its floating-point counterpart, the smaller the bit width of the values used, the less resources that are required to utilise them; not just in terms of computational elements, like adders and multipliers, but also with more basic components such as registers, and even the connections between components.

Consider two networks, one with 64-bit values, the other 8 bits, implemented on a Xilinx series 7 FPGA. From Xilinx's documentation, each Configurable Logic Block (CLB) in a series 7 has: 8 6-input Look Up Tables; 16 1-bit Flip-Flops; 2 Carry Chains; 256 bits of Distributed RAM, for data storage; and 128 bits of Shift Registers [7]. If the networks required something as simple as two signals to undergo a bitwise AND operation, the 8-bit network would fit the required hardware with a single CLB. The 64-bit network on the other hand would not only need multiple CLBs, but also the additional complexity of the routing elements that connect the CLBs together. The same problem exists with memory: if the two networks each have, say, 4 inputs and 4 genes; then each gene requires 12 parameter values (identification, proximity, slope, offset and 8 weights). In this 8-bit network, this is a total of 96-bits of memory, once again able to fit within the resources provided by a single CLB. The 64-bit network requires 768-bits of memory, equal to the RAM of 3 CLBs.

A. Data Width Calculation

While some components of the network's algorithm can remain unchanged, as they function without difficulty when remapped from floating-point to integer, an issue arises with the activation function within the genes/epigenetic molecules.

$$y = \frac{1}{1 + e^{-sx-o}} \tag{1}$$

Equation 1 shows this activation function, a basic sigmoid, where y is the output of the gene; x is the weighted sum of inputs; and s and o are adjustment parameters referred to as the slope and offset respectively. With the floating-point networks, all the parameters, with the exception of the slope, are in the range -1.0 to +1.0 (the slope range is ±20.0) [4]. this means that for any given gene:

- A weighted input can never exceed ±1.0.
- The sum of weighted inputs can never exceed ±n, where n = maximum number of inputs a gene could have.
- The sigmoid exponent can never be exceed ±40.0n.

Therefore, if a network was implemented with 64-bit floating-point values, which have a range of ±1.7*10^{308}, a gene would require more than 4.25*10^{306} inputs for an overflow to occur. However, if the parameters of this floating-point network were directly mapped to a 64-bit integer network, then overflow could potentially occur when an input is multiplied by its weight. In order to prevent this, the integer range will be used to fix the ranges of all parameters, while increased bit widths will be calculated for the weighted inputs, the weighted sum and the sigmoid exponent.

$$P_{range} = \pm(2^{n-1}) - 1 \tag{2}$$

$$WI_{range} = \pm {P_{range}}^2 \tag{3}$$

$$Sum_{range} = \pm N(WI_{range}) \tag{4}$$

$$SigPow_{range} = \pm\left(20(P_{range})\right)(Sum_{range} \pm P_{range}) \tag{5}$$

Equations 2 through 5 detail the range calculation process. Starting with equation 2 which restates that the range for most network parameters (P_{range}) is defined by the range of values that can be represented by the bit width of the network (n), when using the two's complement representation (. Equation 3 states the range for a weighted input (WI_{range}) is the basic parameter range squared, and equation 4 that the weighted sum range (Sum_{range}) is that multiplied by the number of possible inputs (N). This is because the largest absolute value of the weighted sum of inputs would arise if the largest possible input and largest possible weight were present at all possible gene inputs, and as weights and inputs have the same range, this maximum weighted input range is the square of their range. Lastly, equation 5 details the range of the sigmoid power, which is the range of the weighted sum combined with the ranges for the slope and offset values of the sigmoid. Taking the example of a 4-bit network with 4 possible inputs for each gene: the parameters take the range ±7; the weighted inputs, ±49 (72); the weighted sum of inputs, ±196 (4*72); and the sigmoid exponent, ±28420 (140*((4*72) ±7)).

This results in the following specifications for the data widths of an integer network:

$$Param_{width} = n \tag{6}$$

$$Weighted\ Input_{width} = 2n \tag{7}$$

$$Sum_{width} = 3n \tag{8}$$

$$Sigmoid\ Power_{width} = 6n \tag{9}$$

Equation 6 restates that the network parameters are of the base data width n bits; 7 that the weighted inputs will need 2n bits, as a multiplication operation occurs which doubles to width requirement. Equation 8 states that the sum of inputs needs 3n bits, to allow for the range increase caused by the cumulative addition; and finally 9 states that the sigmoid power needs 6n bits, or 2(3n), as another multiplication occurs.

B. Sigmoid Function

The other alteration required for a switch to integer mathematics is to the sigmoid activation function itself, as any exponent of e outside the range ±1.0 will not work, and translating the integer to an acceptable value beforehand would still require the e exponent element to be computed, which brings with it significant hardware requirements (Xilinx's recommendation is to employ the Floating-Point Operator that was mentioned previously [6], which would very much defeat the purpose of the switch to integer). It is possible however, to construct an integer version of the sigmoid by using a Look Up Table (LUT). There is a design consideration with LUTs, which is the number of entries they hold. While a trivial function, like a two input AND gate, can be fully described with only 4 entries; something like the sigmoid function has the potential to be much too large to use this simplistic approach. For example, the 4-bit network used as an example in previously would need at 56841 entries in its lookup table in order to fully map each possible input to its outputs. Recalling the FPGA CLB specifications at the start of this section shows that while this wouldn't fit into the 256 bits of distributed RAM available, let alone accommodate the other parameters required.

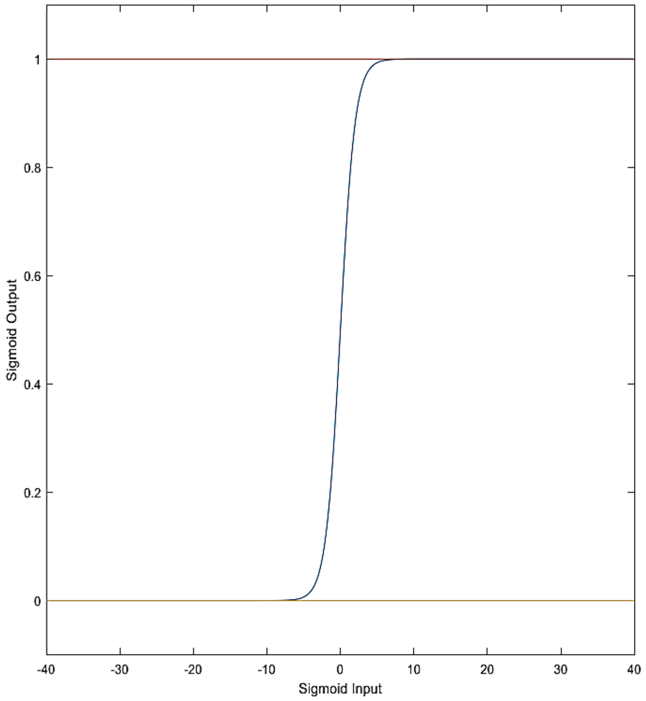

Fig. 3. Plot of the sigmoid activation function for a floating point network, with an input range of ±40. The upper and lower lines show where the sigmoid output levels off at 1.0 and 0.0 respectively.

Fortunately, the sigmoid function itself provides a simple method of reducing the LUT requirements significantly. Figure 3 is a plot of the sigmoid activation function for a floating point network, where the possible inputs to the sigmoid fall into the range ±40. Looking at this figure it is clear to see that a significant number of possible inputs result in outputs of ether 0.0 or 1.0, with the region of interest actually only being between -10.0 and +10.0. This means that when creating the integer LUTs, possible entries that correspond to outputs of 0 or $+2^{n-1}$ can be removed from the table can handled with trivial if statements. Returning again to the 4-bit network example, this method reduces the 56841 entry lookup table to only 4202 entries.

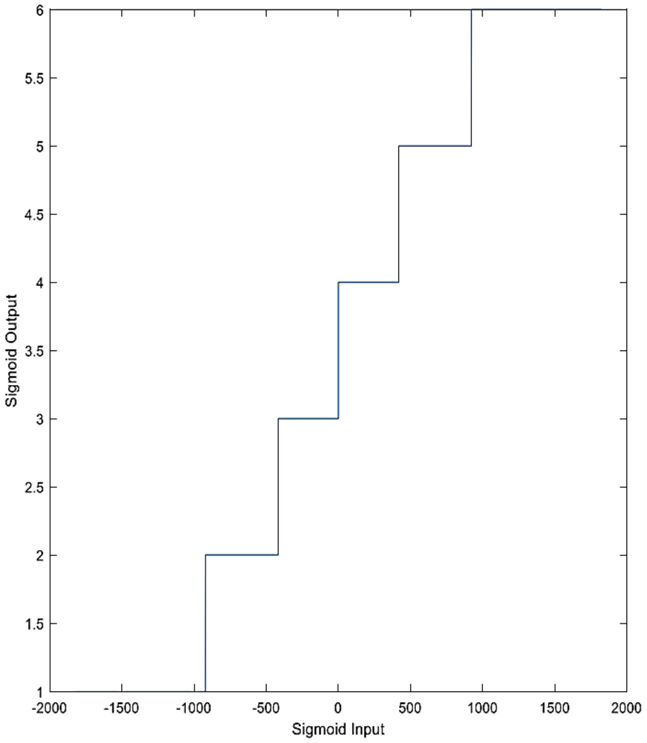

Fig. 4. Plot of the sigmoid activation function for a 4-bit network with maximum and minimum output values removed.

Figure 4 illustrates the "sigmoid" described by such a 4202 entry LUT, which shows another possible route to further reducing its size. Each possible output of the LUT corresponds to multiple input values, so instead of having one entry per input value, the LUT can be reduced to one per possible output value, with external logic to determine which entry an input corresponds to (input < A then output = entry A). Switching to this implementation allows the 4-bit network previously described to have a lookup table of only 6 elements. This equates to a LUT with a total of 24 bits, meaning it fits into the 256 bits of distributed RAM that the CLBs detailed in section 4.3.

V. Hardware Implementation

With an integer and lookup table based version of an Artificial Epigenetic Network (AEN) already created in software, the translation to hardware requires only that the existing architecture be rebuilt. This was done on a Xilinx Field Programmable Gate Array (FPGA) in Very high-speed integrated circuit Hardware Description Language (VHDL). There are two important properties that the completed hardware network must possess: paramarisability, meaning that properties such as the network's data width, or the number of network elements can be changed easily; and parallelisation, meaning that, unlike the sequential execution of the software networks, the hardware network will execute the epigenetic molecules, genes and outputs of the network all at once, greatly improving time performance.

With these requirements in mind, the hardware network consists of a number of discrete units that can be assembled as required to produce a complete network; along with a controller to handle parameter loading and connections to external systems. In this work, the controller consisted of an ARM Cortex A9 processor [8] with a custom generated AXI-Lite peripheral [9], which allowed the hardware networks various configuration and control signals, as well as its inputs and outputs, to be easily connected to a PC for experimentation purposes.

A. Input Elements

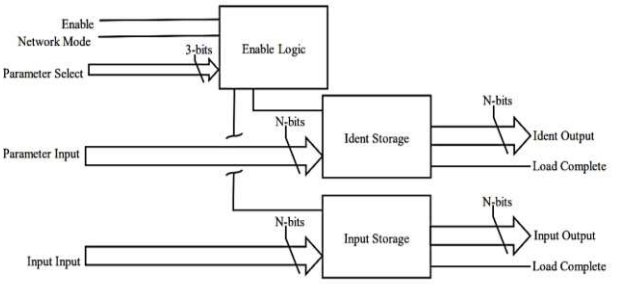

Fig. 5. Block diagram of an n-bit input element, with input value and parameter storage, as well as control logic.

The input elements, shown in figure 5, are the simplest part of the network. They consist of two storage elements, one for the actual input value; and the other for their Ident, effectively their location within the network and required for the network's interconnection system. Both these storage elements are simple n-bit synchronous registers (the CLK is not shown for simplicity), with a small amount of additional logic to ensure correct loading of data. Lastly is the enable logic component, which takes the external enable signal and carries it through to one of the two storage elements depending on the mode of the network and the value of the parameter selection bus. The network mode is a simple signal, with a value of 0 specifying that the network is to execute normal behaviour (taking in inputs and computing outputs); while a value of 1 corresponds to the network being in configuration mode, during which parameters, like the input elements identification, are loaded.

B. Gene Elements

The gene elements can be broken up into four components for ease of understanding: parameter storage, weighted sum calculation, LUT address calculation, and output generation.

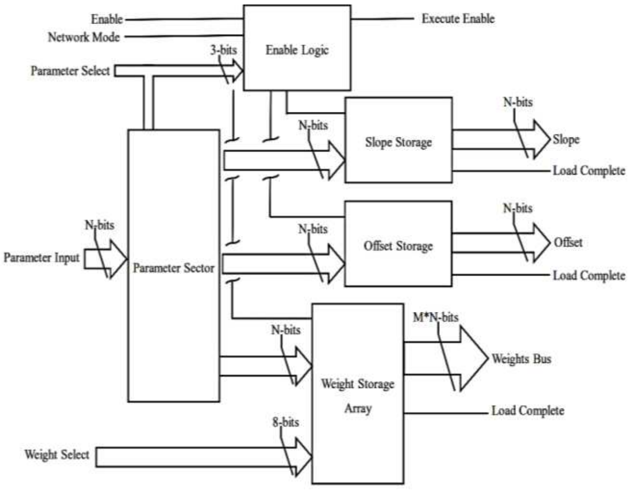

Fig. 6. Block diagram of the parameter storage section of an n-bit gene element with m possible inputs

The parameter storage section, shown in figure 6, is quite similar to the input units, but with a few key differences. First, the presence of more than one parameter storage element necessitates the use of a selection unit, which routes the parameter input through to the one of the storage elements, depending on the value of the parameter select input. This signal, which also now has an effect on the enable logic due to the multiple storage array enables, acts as an address, specifying which of the parameters is to be loaded. A similar function is performed by the weight select signal for the weight storage array. Given that each gene requires multiple weights, one for each possible connection it can form with both other genes and the input units, the parameter select alone is insufficient to address each storage element. Instead, the parameter select routes the parameter input and enable signals through to an array of storage elements, which have their own additional layer of logic mirroring those above. This layer uses the weight select signal to pass the parameter input and enable signals through to one of the storage elements. The weight select bus uses a simple numerical value, much like an addressable memory. While the number of weights, and thus the size of the weight storage array is parameterisable, the weight select bus is a fixed 8 bits, due to its requirement to connect to external components. This still allows for each gene to have a total of 256 possible inputs, which is more than sufficient.

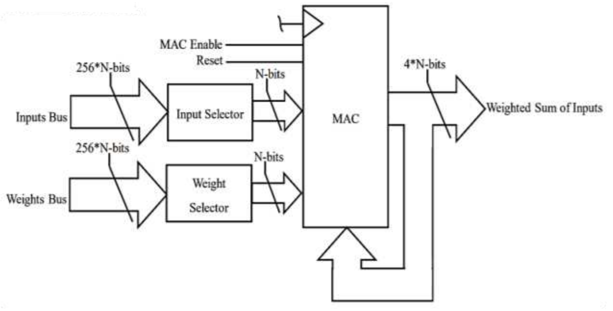

Fig. 7. Block diagram of the weighted sum calculation section of an n-bit gene element.

The weighted sum calculation section, shown in figure 7, primarily consists of a multiply-accumulate unit (MAC), and a pair of selectors which step though the array of input and weight values. Not shown is the controller for this section, which consists of a small finite state machine (FSM), simply runs the MAC for a number of cycles equal to the number of input/weight pairs, then sends an enable signal to the next stage of the gene's logic.

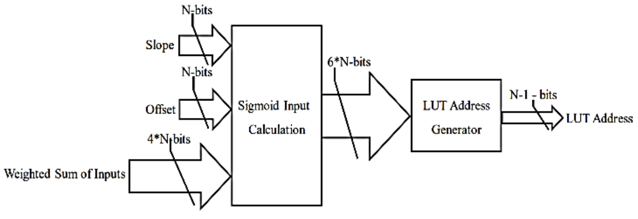

Fig. 8. Block diagram of the LUT address calculation section of an n-bit gene element.

The LUT address calculation section, shown in figure 8, consists of the logic to calculate the power value from the weighted sum, slope and offset values (the slope is multiplied by 20 at this point to reduce storage space), as shown in equation 1. The power value is then used to produce the address for the LUT, by applying the process detailed in section 3B. The resulting address is then passed to the final section.

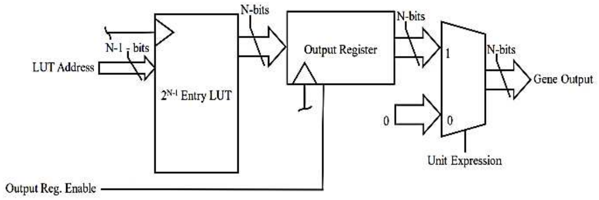

Fig. 9. Block diagram of the LUT and output storage for an n-bit gene element.

The LUT is a section of memory that stores the integer version of the sigmoid, like that shown in figure 4, though a different version is needed depending on the bit-width of the network. The output of the LUT is stored in the register, when enabled by the FSM. The final element is a multiplexer, which is controlled by the unit expression signal. When a gene is suppressed by an epigenetic molecule, this replaces its output with a value of 0, preventing it from having an impact on the network. Note that the enable signal for a suppressed gene is also suppressed, which prevents it from executing its calculation logic.

C. *Epigenetic Molecules*

As epigenetic molecules follow the same process for generating their output values from their inputs as genes, their hardware version is identical to that detailed in the previous section. There is one small difference, that being that epigenetic molecules do not have the output multiplexer or enable suppression logic.

D. *Gene/Molecule Wrappers*

It should be noted that the description in the previous two sections made no reference to the mechanism for connecting genes/molecules together. That is because this function is separated from the genes/molecules themselves and placed within a pair of wrappers.

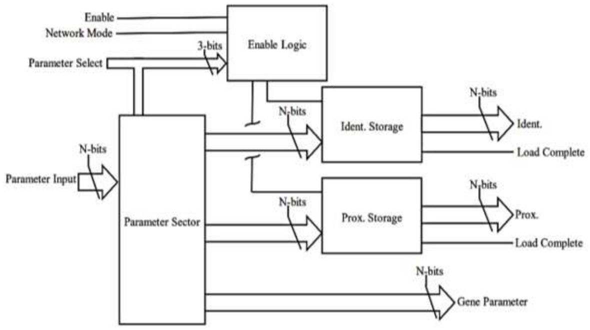

Fig. 10. Block diagram of the parameter storage section of an n-bit gene wrapper.

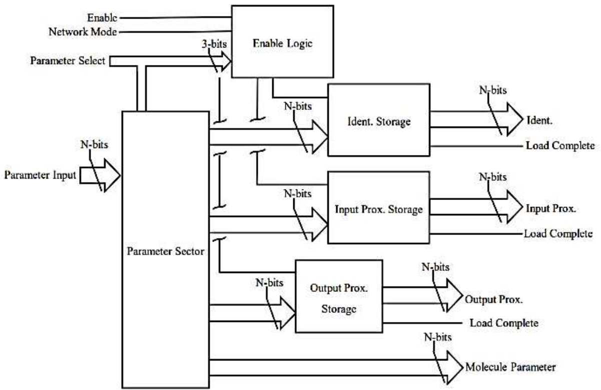

Fig. 11. Block diagram of the parameter storage section of an n-bit epigenetic molecule wrapper.

The parameter storage section of the gene and molecule wrappers, shown in figures 10 and 11 respectively, are comparable to their counterparts within the genes and molecules themselves, shown in figure 6. Other than storing different values, the only other difference of note is that the parameter selector includes the ability to pass a parameter value onwards, into the storage component of gene or molecule contained in the wrapper.

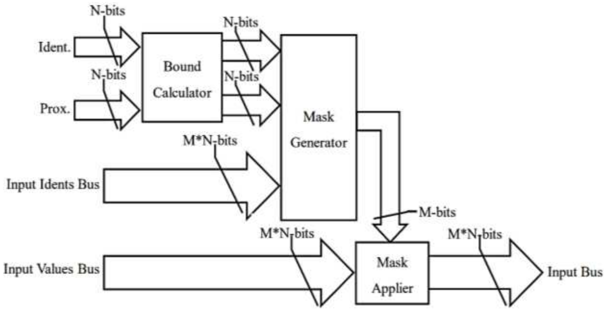

Fig. 12. Block diagram of the input selection section of an n-bit gene or epigenetic molecule wrapper.

The second part of the gene and molecules wrappers is the input selector, shown in figure 12. This is the same for both kinds of wrapper and uses a system of location and ranges to determine if a particular network element should be an input to the wrapped component. The Ident value, also appearing in input elements, is the location within the network; while the Proximity is the "distance" that an element should look for other elements to connect to. Note that a molecule's' wrapper has two different proximities, one for inputs and the other for outputs. The output proximity will be addressed later. These two values are used along with the Idents of the other network elements to create a mask, which specifies whether or not a given element is "close enough" to the wrapped element to be an input. This mask is then applied to the bus of possible inputs from all network elements. Any from units outside the wrapper's range are zeroed, while those in range passed through to the gene or molecule within the wrapper to serve as its inputs.

E. *Gene Expression Controller*

Each molecule has an associated gene expression controller, which functions similarly to the input selection section shown in figure 12, though with a few key differences. Instead of input proximity, it uses the wrapper's output proximity; while the Ident is replaced with the output of the epigenetic molecule; and instead of possible inputs, the Idents taken in by the mask generator are those of the network's gene elements. The resulting mask now specifies whether a given gene is in range of a given molecule, and hence should have its activity supressed.

F. *Output Elements*

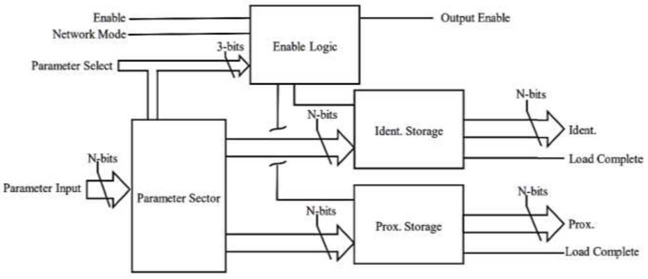

Fig. 13. Block diagram of the storage section of an n-bit output element.

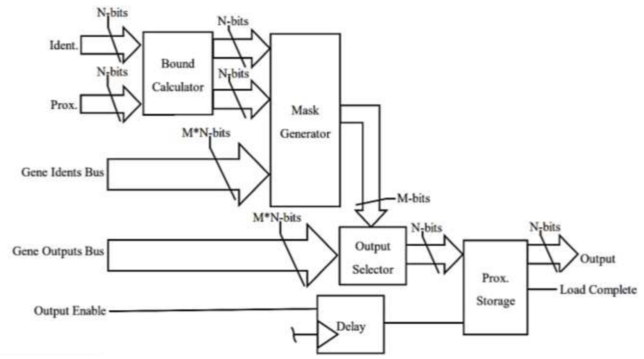

Fig. 14. Block diagram of the output value selection section of an n-bit output element.

The final network components are the output elements, which consist of a storage section, shown in figure 13, and an output value selection section, shown in figure 14. The storage section is comparable to those of other elements, while the output value selector functions akin to the input value selector shown in figure 12. The only differences are that, like the expression controller in the previous section, only gene elements are considered; and instead of multiple values being chosen by the mask, only one is. This mask chosen value is then passed to the output storage register. The last element to note is the delay, which holds back the register's enable signal to ensure that the output selection process is complete.

VI. Precision Reduction

With a network now available to perform experiments with, it is possible to look at the effects of reducing the bit width of the networks. More specifically, the goal is to identify how low the bit width can be made without it negatively impacting the performance of the network.

A. Experiment Design

As it is already a task that has been proven to be solvable by the existing AENs, this experiment will use the inverted pendulum, specifically the implementation designed by Hamann et al [10], illustrated in Fig. 15. The Hamann inverted pendulum, intended as a benchmark for robotic control research, models the behaviour of an inverted pendulum on a cart that is able to move in 1 dimension. The model is implemented using the Runge-Kutta method of the 3rd order [11], while feedback on the status of the pendulum and cart are provided by simulated sensors that only monitor part of the model, instead of providing absolute measurements of factors like pendulum angle and velocity. In addition, the outputs of these sensors (as well as the control signals for the model's actuators) are low resolution, mapping all values to the range [0, 127].

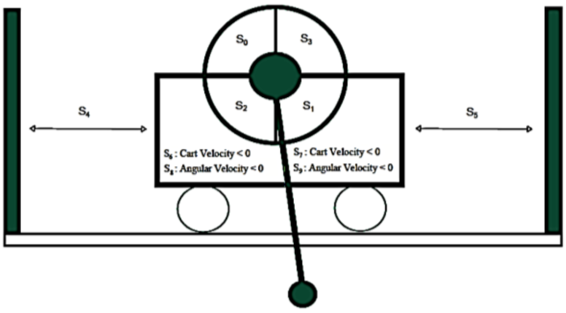
S_0
S_3
S_2
S_1
S_4
S_5
S_6 : Cart Velocity < 0
S_8 : Angular Velocity < 0
S_7 : Cart Velocity < 0
S_9 : Angular Velocity < 0

Fig. 15. A cart and pendulum from the Hamann model, illustrating the "positions" of the various simulated sensors, taken from [10].

The fitness function for the Hamann model returns a value is proportional to the number of simulation time steps that the pendulum spent in the upright position, as is described in equation 10.

$$Fitness = \sum_{t=0}^{t_{max}} \frac{|\theta(t) - \pi|}{t_{max}\pi} \quad (10)$$

t is the current time step, t_{max} is the maximum run time of the simulation, and $\theta(t)$ is the angle off vertical at the given time step. A fitness higher than 0.75 is considered to denote a run in which the pendulum is being maintained in the upright position satisfactorily.

Design and optimization of the epigenetic network was done with hardware in-loop utilising a simple genetic algorithm (GA). The GA employed rank-based selection with elitism of the top 12.5% (1/8th) of the population, uniform crossover and random replacement mutation. The other experimental parameters are given in table 1.

TABLE I. Parameters for Bit Width Experiments

Parameter	Value
Population Size	64
Number of Generations	2048
Number of Repeats	50
Crossover Rate	0.5
Mutation Rate	0.05
Number of Genes	20
Number of Epigenetic Molecules	3

This process will be repeated for all data widths between 4 and 32 bits. Ideally, this range would go up to 52 bits, so as to be equivalent to the mantissa width of a 64 bit floating-point value, but this limitation is imposed by the 32 bit width of the AXI-Lite bus [9]. Any data width lower than 4 is considered too small to be worth testing, as due to the network's interconnection mechanism previous discussed, it would result in all the elements of the network being connected together in one large mass.

B. Results

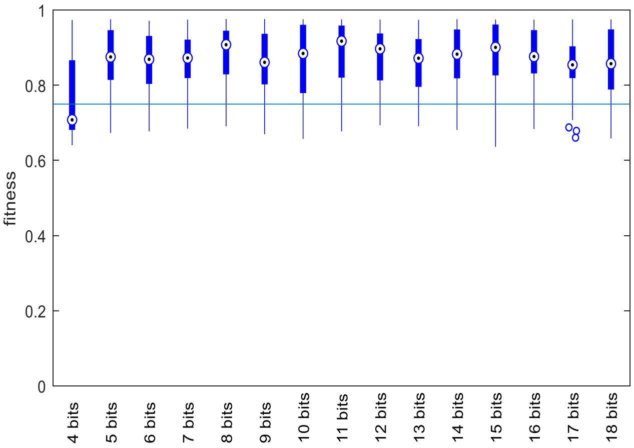

Fig. 16. Fitnesses of the best integer epigenetic networks from each repeat, with data widths in the 4- to 18-bit range. The blue line at 0.75 denotes the fitness at which the networks are able to maintain the pendulum in the upright equilibrium position.

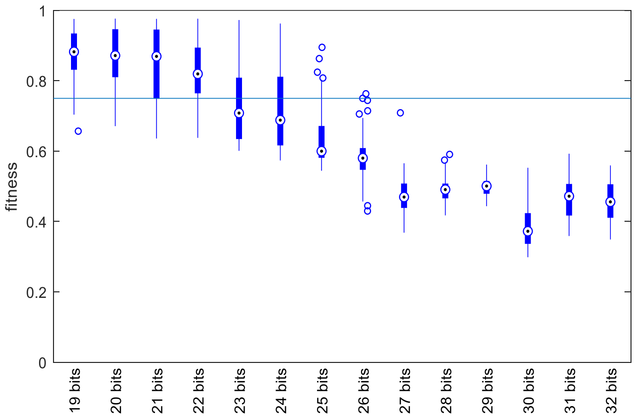

Fig. 17. Fitnesses of the best integer epigenetic networks from each repeat, with data widths in the 19- to 32-bit range.

The results of the data width evolutions are shown in figures 16 and 17. 4-bit networks are able to exceed the 0.75 fitness threshold, but most fall short, with a median fitness of 0.71. Networks with data widths between 5 and 20-bits all exhibit similar results to each another, with most networks achieving a fitness greater than 0.75. However, unexpected behaviour occurs with the networks with data widths beyond this point. Given that the floating-point networks have a data width of 64 bits, it should be expected that the fitness values would slowly increase as the data widths got closer to this value, but instead a decline begins at 21-bits. By 23-bits the median value is below 0.75, and by 27-bits even outliers are unable to exceed this fitness.

Figure 18 clears up this mystery however, showing the changing maximum fitness of an 8-bit and 32-bit network over 4000 generations. This suggests that the increase in data width comes with an increase in the search space of possible network configurations, and thus that given sufficient generations the higher bit width networks will reach fitness values eqivalent to those that the smaller width networks. More importantly however, the result of these experiments give a clear picture: reducing the width of the integer networks even to as low as 5-bits can still produce viable controllers.

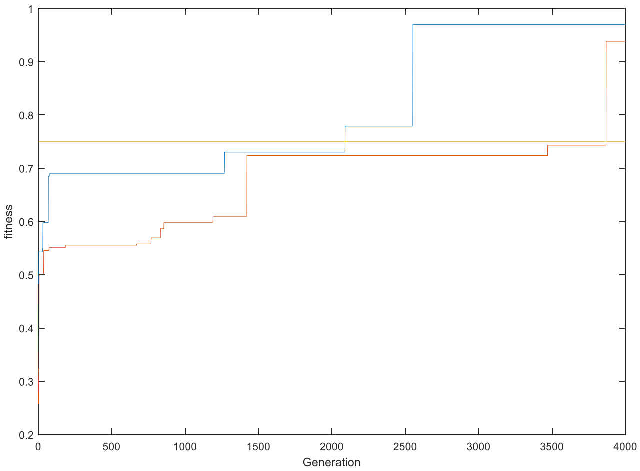

Fig. 18. Time evolution plot, showing fitness over 4000 generations: the blue line is an 8-bit network; the orange a 32-bit network; and the yellow line is the 0.75 fitness threshold.

VII. Comparison with Original Networks

The final step is to compare the networks described in this paper with their original counterparts. Looking at the performance of the inverted pendulum task, a number of networks were evolved using the same parameters as Turner's original experiments [4], while using integer hardware architecture described in this paper. Table 2 shows the parameters of this experiment.

TABLE II. Parameters for Bit Width Experiments

Parameter	Value
Population Size	500
Number of Generations	200
Number of Repeats	50
Crossover Rate	0.5
Mutation Rate	0.05
Bit Width	8
Number of Genes	12
Number of Epigenetic Molecules	3

Table 3 contains the results of these evolutions, compared against their 64-bit floating-point networks counterparts.

TABLE III. Fitness comparisions

Property	8-bit Networks	64-bit Networks
Minimum Fitness	0.83	0.95
25th Percentile Fitness	0.89	0.96
Median Fitness	0.91	0.97
75th Percentile Fitness	0.95	0.98
Interquartile Range	0.06	0.02
Maximum Fitness	0.98	0.98

While the integer networks fall slightly behind their floating-point counterparts in most of these metrics, they are able to match them in maximum fitness, 0.98 in both cases. Given this fact it is reasonable to assert that, although statistically speaking there is a slight performance reduction; an 8-bit, integer hardware epigenetic network is able to achieve a similar performance as a 64-bit, floating point software version.

VIII. Resource Utalisation

Having demonstrated that a reduced precision integer hardware network is able to obtain comparable functional performance to its floating-point predecessor, it is now necessary to validate the argument for making the transition. Table 4 shows the resource utilisation of the various elements of both a 32-bit network and an 8-bit network. Utilisation is given as a percentage of the available resources on the FPGA used in the experiments thus far, a Xilinx ZynQ-7000 [8].

TABLE IV. RESOURCE UTALISATION BY ELEMENT TYPE

Element	Resource	32-bit network		8-bit network	
		No.	Percent	No.	Percent
Input	LUTs	607	1.14%	272	0.51%
	Registers	40	0.04%	18	0.02%
	DSP Slices	0	0.00%	0	0.00%
	F7 Mux.	0	0.00%	0	0.00%
	F8 Mux.	0	0.00%	0	0.00%
	BRAM	0	0.00%	0	0.00%
Gene	LUTs	2643	4.97%	1105	2.08%
	Registers	379	0.36%	278	0.26%
	DSP Slices	16	7.27%	2	0.91%
	F7 Mux.	368	1.38%	62	0.23%
	F8 Mux.	116	0.87%	18	0.14%
	BRAM	32	22.86%	0.5	0.36%
Mole.	LUTs	2485	4.67%	965	1.81%
	Registers	386	0.36%	284	0.27%
	DSP Slices	16	7.27%	2	0.91%
	F7 Mux.	368	1.38%	20	0.08%
	F8 Mux.	116	0.87%	0	0.00%
	BRAM	32	22.86%	0.5	0.36%
Output	LUTs	377	0.71%	145	0.27%
	Registers	48	0.05%	24	0.02%
	DSP Slices	0	0.00%	0	0.00%
	F7 Mux.	0	0.00%	0	0.00%
	F8 Mux.	0	0.00%	0	0.00%
	BRAM	0	0.00%	0	0.00%

The two largest differences are in the DSP and BRAM utilization; although this is not surprising, as the DSP resources are partly responsible for mathematical operations, along with the LUTs; while the BRAM resources hold the sigmoid lookup tables. In both these cases, increasing the data width brings increased demands. Outside these two instances, the other resources are also used to a greater extent by the 32-bit network, with the LUT and Multiplexer utilisation almost doubling. This clearly illustrates the advantage of reducing the bit width of the network when it comes to reducing silicon resources. And while there isn't a hardware floating point network to compare with as well, the fact that one would need an even greater data width, as well as specialised mathematical elements, shows the benefits of making the switch.

IX. EXECUTION TIMES

Another axis of comparison, and one where a floating-point network can be looked at, is execution time. The hardware network is designed to allow for parallel execution, whereas the both the integer and floating-point software versions are forced to execute each network element in turn.

TABLE V. AVERAGE EXECUTION TIMES

Network Type	8-bit integer hardware network	8-bit integer software network	64-bit integer software network	64-bit floating point software network
Execution Time	3 µS	26 µS	29 µS	542 µS

Table 5 shows these averaged execution time measurements, which encapsulates molecule execution; gene activation state updates; gene execution; and output execution. Input execution is not included as their execution timings are primarily dependent on factors external to the network. The hardware network is an order of magnitude faster than its software counterparts, which is almost certainly down to the parallelisation. The two integer software networks have comparable execution times, indicating that the bit width has little impact on execution times. However, the floating-point network is dramatically slower than even the 64-bit integer network, something that demonstrates the additional computational complexity brought on by floating point maths.

X. CONCLUSION AND FUTURE WORK

Drawing this work to a close, and taking all the results presented though out into consideration, it is reasonable to assert that: by transitioning to a dedicated, integer-based hardware architecture, it is possible to significantly reduce the resource requirements of an epigenetic network without making significant performance sacrifices.

To move forward in this area, it will be necessary to investigate possible applications where the properties of the AEN architecture can bring benefits, but there is a need to keep computation resource usage to a minimum. The author's own opinion is that such a use case can within the field of robotics, as the multi-facetted nature of robotic control problems would likely benefit from AEN's ability to divide the aspects of a task across the different sections of a suitably designed network.

ACKNOWLEDGMENT

Work in this paper was previously submitted for the award of PhD in Electronic Engineering at the University of York by Andrew Walter. Funding for this work was received from the EPSRC & RAEng.

REFERENCES

[1] A. P. Turner, M. A. Trefzer, M. A. Lones and A. M. Tyrrell, "Evolving Efficient Solutions to Complex Problems Using the Artificial Epigenetic Network," *Information Processing in Cells and Tissues,* vol. 9303, pp. 153-165, 2015.

[2] A. Walter, *Hardware Implementation of Epigenetic Networks,* York: University of York, 2019.

[3] S. Cussat-Blanc, K. Harrington and W. Banzhaf, "Artificial Gene Regulatory Networks - A Review," *Artificial Life,* vol. 24, no. 4, pp. 296 - 328, 2018.

[4] A. P. Turner, *The Artifical Epigenetic Network,* York: University of York, 2013.

[5] A. P. Turner, M. A. Lones, L. A. Fuente, S. Stepney, L. S. D. Caves and A. Tyrrell, "The Artifical Epigenetic Network," in *IEEE International Conference on Evolvable Systems*, Singapore, 2013.

[6] Xilinx, "LogiCORE IP: Floating-Point Operator," Xilinx, San Jose, California, 2014.

[7] Xilinx, "7 Series FPGAs Configurable Logic Block User Guide," Xilinx, San Jose, California, 2016.

[8] Xilinx, "Zynq-7000 SoC Data Sheet," Xilinx, San Diego, California, 2018.

[9] Xilinx, "Vivado Design Suite - AXI Reference Guide," Xilinx, San Diego, California , 2017.

[10] H. Hamann, T. Schmickl and K. Crailsheim, "Coupled Inverted Pendulums: A Benchmark for Evolving Decentral Controllers in Modular Robotics," in *Genetic and Evolutionary Computation Conference*, Dublin, 2011.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, Numerical Recipes in C++, Cambridge: Cambridge University Press, 2002.