Analog Computation and Learning in VLSI

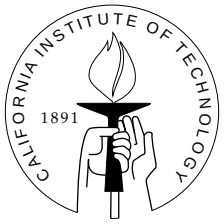
Thesis by

Vincent F. Koosh

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2001

(Submitted April 20, 2001)

Acknowledgements

First, I would like to thank my advisor, Rod Goodman, for providing me with the resources and tools necessary to produce this work. Next, I would like to thank the members of my thesis defense committee, Jehoshua Bruck, Christof Koch, Alain Martin, and Chris Diorio. I would also like to thank Petr Vogel for being on my candidacy committee. Thanks also goes out to some of my undergraduate professors, Andreas Andreou and Gert Cauwenberghs, for starting me off on this analog VLSI journey and for providing occasional guidance along the way. Many thanks go to the members of my lab and the Caltech VLSI community for fruitful conversations and useful insights. This includes Jeff Dickson, Bhusan Gupta, Samuel Tang, Theron Stanford, Reid Harrison, Alyssa Apsel, John Cortese, Art Zirger, Tom Olick, Bob Freeman, Cindy Daniell, Shih-Chii Liu, Brad Minch, Paul Hasler, Dave Babcock, Adam Hayes, William Agassounon, Phillip Tsao, Joseph Chen, Alcherio Martinoli, and many others. I thank Karin Knott for providing administrative support for our lab. I must also thank my parents, Rosalyn and Charles Soffar, for making all things possible for me, and I thank my brothers, Victor Koosh and Jonathan Soffar. Thanks also goes to Lavonne Martin for always lending a helpful hand and a kind ear. I also thank Grace Esteban for providing encouragement for many years. Furthermore, I would like to thank many of my other friends and colleagues, too numerous to name, for the many joyous times.

Abstract

Nature has evolved highly advanced systems capable of performing complex computations, adaptation, and learning using analog components. Although digital systems have significantly surpassed analog systems in terms of performing precise, high speed, mathematical computations, digital systems cannot outperform analog systems in terms of power. Furthermore, nature has evolved techniques to deal with imprecise analog components by using redundancy and massive connectivity. In this thesis, analog VLSI circuits are presented for performing arithmetic functions and for implementing neural networks. These circuits draw on the power of the analog building blocks to perform low power and parallel computations.

The arithmetic function circuits presented are based on MOS transistors operating in the subthreshold region with capacitive dividers as inputs to the gates. Because the inputs to the gates of the transistors are floating, digital switches are used to dynamically reset the charges on the floating gates to perform the computations. Circuits for performing squaring, square root, and multiplication/division are shown. A circuit that performs a vector normalization, based on cascading the preceding circuits, is shown to display the ease with which simpler circuits may be combined to obtain more complicated functions. Test results are shown for all of the circuits.

Two feedforward neural network implementations are also presented. The first uses analog synapses and neurons with a digital serial weight bus. The chip is trained in loop with the computer performing control and weight updates. By training with the chip in the loop, it is possible to learn around circuit offsets. The second neural network also uses a computer for the global control operations, but all of the local operations are performed on chip. The weights are implemented digitally, and counters are used to adjust them. A parallel perturbative weight update algorithm is used. The chip uses multiple, locally generated, pseudorandom bit streams to perturb all of the weights in parallel. If the perturbation causes the error function to decrease,

the weight change is kept, otherwise it is discarded. Test results are shown of both networks successfully learning digital functions such as AND and XOR.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Although many applications are well suited for digital solutions, analog circuit design continues to have an important place. The places where analog circuits are required and excel are the places where digital equivalents cannot live up to the task. For example, ultra high speed circuits, such as in wireless applications, require the use of analog design techniques. Another important area where digital circuits may not be sufficient is that of low power applications such as in biomedical, implantable devices or portable, handheld devices. Finally, the importance of analog circuitry in interfacing with the world will never be displaced because the world itself consists of analog signals.

In this thesis, several circuit designs are described that fit well into the analog arena. First, a set of circuits for performing basic analog calculations such as squaring, square rooting, multiplication and division are described. These circuits work with extremely low current levels and would fit well into applications requiring ultra low power designs. Next, several circuits are presented for implementing neural network architectures. Neural networks have proven useful in areas requiring man-machine interactions such as handwriting or speech recognition. Although these neural networks can be implemented with digital microprocessors, the large growth in portable devices with limited battery life increases the need of finding custom low power solutions. Furthermore, the area of operation of the neural network circuits can be modified from low power to high speed to meet the needs of the specific application. The inherent parallelism of neural networks allows a compact high speed solution in analog VLSI.

1.1 Overview

1.1.1 Analog Computation

In chapter 2, a brief review of analog computational circuits is given with an emphasis on translinear circuits. First, traditional translinear circuits are discussed which are implemented with devices showing an exponential characteristic, such as bipolar transistors. Next, MOS transistor versions of translinear circuits are discussed. In subthreshold MOS translinear circuits the exponential characteristic is exploited similarly to bipolar transistors; whereas, above threshold MOS translinear circuits use the square law characteristic and require a different design style. Also mentioned are translinear circuits that utilize networks of multiple, linear input elements. These form the basis of the circuits described in the next chapter.

In chapter 3, A class of analog CMOS circuits that can be used to perform many analog computational tasks is presented. The circuits utilize MOSFETs in their subthreshold region as well as capacitors and switches to produce the computations. A few basic current-mode building blocks that perform squaring, square root, and multiplication/division are shown. This should be sufficient to gain understanding of how to implement other power law circuits. These circuit building blocks are then combined into a more complicated circuit that normalizes a current by the square root of the sum of the squares (vector sum) of the currents. Each of these circuits have switches at the inputs of their floating gates which are used to dynamically set and restore the charges at the floating gates to proceed with the computation.

1.1.2 VLSI Neural Networks

In chapter 4, a brief review of VLSI neural networks is given. First, a discussion of the various learning rules which are appropriate for analog VLSI implementation are discussed. Next, some of the issues associated with analog VLSI networks is presented such as limited precision weights and weight storage issues. Also, a brief discussion of several VLSI neural network implementations is presented.

In chapter 5, a VLSI feedforward neural network is presented that makes use of digital weights and analog synaptic multipliers and neurons. The network is trained in a chip-in-loop fashion with a host computer implementing the training algorithm. The chip uses a serial digital weight bus implemented by a long shift register to input the weights. The inputs and outputs of the network are provided directly at pins on the chip. The training algorithm used is a parallel weight perturbation technique. Training results are shown for a 2 input, 1 output network trained with an AND function, and for a 2 input, 2 hidden layer, 1 output network trained with an XOR function.

In chapter 6, a VLSI neural network that uses a parallel perturbative weight update technique is presented. The network uses the same synapses and neurons as the previous network, but all of the local, parallel, weight update computations are performed on-chip. This includes the generation of random perturbations and counters for updating the digital words where the weights are stored. Training results are also shown for a 2 input, 1 output network trained with an AND function, and for a 2 input, 2 hidden layer, 1 output network trained with an XOR function.

Chapter 2 Analog Computation with Translinear Circuits

There are many forms of analog computation. For example, analog circuits can perform many types of signal filtering including lowpass, bandpass, and highpass filters. Operational amplifier circuits are commonly used for these tasks as well as for generic derivative, integrator, and gain circuits. The emphasis of this chapter, however, shall be on circuits with the ability to implement arithmetic functions such as signal multiplication, division, power law implementations, and trigonometric functions. A well known class of circuits exists to perform such tasks called translinear circuits. The basic idea behind using these types of circuits is that of using logarithms for converting multiplications and divisions into additions and subtractions. This type of operation was well know to users of slide rules in the past.

2.1 BJT Translinear Circuits

The earliest forms of these type of circuits were based on exploiting the exponential current-voltage relationships of diodes or bipolar junction transistors[1][2].

For a bipolar junction transistor, the collector current, I_C, as a function of its base-emitter voltage, V_{be}, is given as

$$I_C = I_S \exp\left(\frac{V_{be}}{U_t}\right)$$

where I_S is the saturation current, and $U_t = kT/q$ is the thermal voltage, which is approximately 26mV at room temperature. In using this formula, the base-emitter voltage is considered the input, and the collector current is the output which performs the exponentiation or antilog operation. It is also possible to think of the collector

current as the input and the base-emitter voltage as the output such as in a diode connected transistor, where the collector has a wire, or possibly a voltage buffer to remove base current offsets, connecting it to the base. In this case, the equation can be seen as follows:

$$V_{be} = U_t \ln\left(\frac{I_C}{I_S}\right)$$

From here it is possible to connect the transistors, using log and antilog techniques, to obtain various multiplication, division and power law circuits. However, it is not always trivial to obtain the best circuit configuration for a given application. It is also possible to obtain trigonometric functions by using other power law functional approximations[3].

2.2 Subthreshold MOS Translinear Circuits

Because of the similarities of the subthreshold MOS transistor equation and the bipolar transistor equation, it is sometimes possible to directly translate BJT translinear circuits into subthreshold MOS translinear circuits. However, care must be taken because of the nonidealities introduced in subthreshold MOS.

For a subthreshold MOS transistor, the drain current, I_d, as a function of its gate-source voltage, V_{gs}, is given as[30]

$$I_d = I_o \exp\left(\frac{\kappa V_{gs}}{U_t}\right)$$

where I_o is the preexponential current factor, and κ is a process dependent parameter which is normally around 0.7. Thus, it is clear that the subthreshold MOS transistor equation differs from the BJT equation with the introduction of the parameter κ.

Due to this extra parameter, direct copies of the functional form of BJT translinear circuits into subthreshold MOS translinear circuits do not always lead to the same

implemented arithmetic function. For example, for a particular circuit that performs a square rooting operation on a current using BJTs, when converted to the subthreshold MOS equivalent, it will actually raise the current to the $\frac{\kappa}{\kappa+1}$ power[30]. When $\kappa = 1$, this reduces to the appropriate $\frac{1}{2}$ power, but since normally $\kappa \approx 0.7$, the power becomes ≈ 0.41. Nevertheless, several types of subthreshold MOS circuits which display κ independence do exist[4][5]. These involve a few added restrictions on the types of translinear loop topologies allowed.

Subthreshold MOS translinear circuits are more limited in their dynamic range than BJT translinear circuits because of entrance into strong inversion. Furthermore, the characteristics slowly decay in the transition region from subthreshold to above threshold. However, due to the lack of a required base current, lower power operation may be possible with the subthreshold MOS circuits. Furthermore, easier integration with standard digital circuits in a CMOS process is possible.

2.3 Above Threshold MOS Translinear Circuits

The ability of above threshold MOS transistors to exhibit translinear behavior has also been demonstrated[6].

For an above threshold MOS transistor in saturation, the drain current, I_d, as a function of its gate-source voltage, V_{gs}, is given as

$$I_d = K\left(V_{gs} - V_t\right)^2$$

Because of the square law characteristic used, these circuits are sometimes called quadratic-translinear or MOS translinear to distinguish them from translinear circuits based on exponential characteristics. Several interesting circuits have been demonstrated such as those for performing squaring, multiplication, division, and vector sum computation[6][10][11]. However, because of the inflexibility of the square law compared to an exponential law, general function synthesis is usually not as straightforward as for exponential translinear circuits[7][9]. Although many of the circuits

rely upon transistors in saturation, some also take advantage of transistors biased in the triode or linear region[12].

These circuits are usually more restricted in their dynamic range than bipolar translinear circuits. They are limited at the low end by weak inversion and at the high end by mobility reduction[7]. Furthermore, the square-law current-voltage equation is usually not as good an approximation to the actual circuit behavior as for both bipolar circuits and MOS circuits that are deep within the subthreshold region. Also, for decreasingly small submicron channel lengths, the current-voltage equation begins to approximate a linear function more than a quadratic[8].

2.4 Translinear Circuits With Multiple Linear Input Elements

Another class of translinear circuits exists that involves the addition of a linear network of elements used in conjunction with exponential translinear devices. The linear elements are used to add, subtract and multiply variables by a constant. From the point of view of log-antilog mathematics, it is clear that multiplying values by a constant, n, can be transformed into raising a value to the n-th power. Thus, the addition of a linear network provides greater flexibility in function synthesis. When the exponential translinear devices used are bipolar transistors, the linear networks are usually defined by resistor networks[13][14], and when the exponential translinear devices used are subthreshold MOS transistors, the linear networks are usually defined by capacitor networks[15][16]. This class of circuits can be viewed as a generalization of the translinear principle.

Chapter 3 Analog Computation with Dynamic Subthreshold Translinear Circuits

A class of analog CMOS circuits has been presented which made use of MOS transistors operating in their subthreshold region[15][16]. These circuits use capacitively coupled inputs to the gate of the MOSFET in a capacitive voltage divider configuration. Since the gate has no DC path to ground it is floating and some means must be used to initially set the gate charge and, hence, voltage value. The gate voltage is initially set at the foundry by a process which puts different amounts of charge into the oxides. Thus, when one gets a chip back from the foundry, the gate voltages are somewhat random and must be equalized for proper circuit operation. One technique is to expose the circuit to ultraviolet light while grounding all of the pins. This has the effect of reducing the effective resistance of the oxide and allowing a conduction path. Although this technique ensures that all the gate charges are equalized, it does not always constrain the actual value of the gate voltage to a particular value. This technique works in certain cases[17], such as when the floating gate transistors are in a fully differential configuration, because the actual gate charge is not critical so long as the gate charges are equalized. If the floating gate circuits are operated over a sufficient length of time, stray charge may again begin to accumulate requiring another ultraviolet exposure. For the circuits presented here it is imperative that the initial voltages on the gates are set precisely and effectively to the same value near the circuit ground. Therefore, a dynamic restoration technique is utilized that makes it possible to operate the circuits indefinitely.

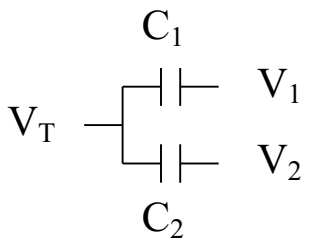


Figure 3.1: Capacitive voltage divider

3.1 Floating Gate Translinear Circuits

The analysis begins by describing the math that governs the implementation of these circuits. A more thorough analysis for circuit synthesis is given elsewhere[15][16]. A few simple circuit building blocks that should give the main idea of how to implement other designs will be presented.

For the capacitive voltage divider shown in fig. 3.1, if all of the voltages are initially set to zero volts and then V_1 and V_2 are applied, the voltage at the node V_T becomes

$$V_T = \frac{C_1 V_1 + C_2 V_2}{C_1 + C_2}$$

If $C_1 = C_2$ then this equation reduces to

$$V_T = \frac{V_1}{2} + \frac{V_2}{2}$$

The current-voltage relation for a MOSFET transistor operating in subthreshold and in saturation ($V_{ds} > 4U_t$, where $U_t = kT/q$) is given by[30]:

$$I_{ds} = I_o exp(\kappa V_{gs}/U_t)$$

Using the above equation for a transistor operating in subthreshold, as well as a capacitive voltage divider, the necessary equations of the computations desired can be produced.

In the following formulations, all of the transistors are assumed to be identical.

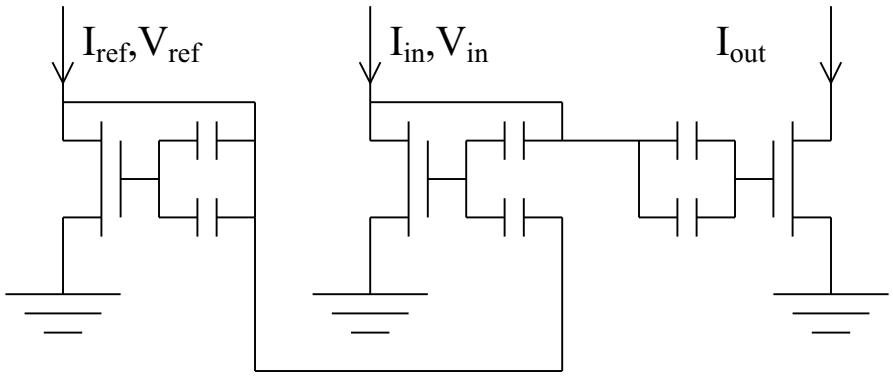


Figure 3.2: Squaring circuit

Also, all of the capacitors are of the same size.

3.1.1 Squaring Circuit

For the squaring circuit shown in figure 3.2, the I-V equations for each of the transistors can be written as follows.

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{out}}{I_o}\right) = V_{in}$$

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{ref}}{I_o}\right) = V_{ref}$$

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{in}}{I_o}\right) = \frac{V_{in}}{2} + \frac{V_{ref}}{2}$$

Then, these results can be combined to obtain the following:

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{in}}{I_o}\right) = \frac{U_t}{\kappa} \ln\left(\frac{I_{out}}{I_o}\right)^{\frac{1}{2}} + \frac{U_t}{\kappa} \ln\left(\frac{I_{ref}}{I_o}\right)^{\frac{1}{2}}$$

$$I_{out} = I_o \left(\frac{I_{in}/I_o}{(I_{ref}/I_o)^{\frac{1}{2}}}\right)^2 = \frac{I_{in}^2}{I_{ref}}$$

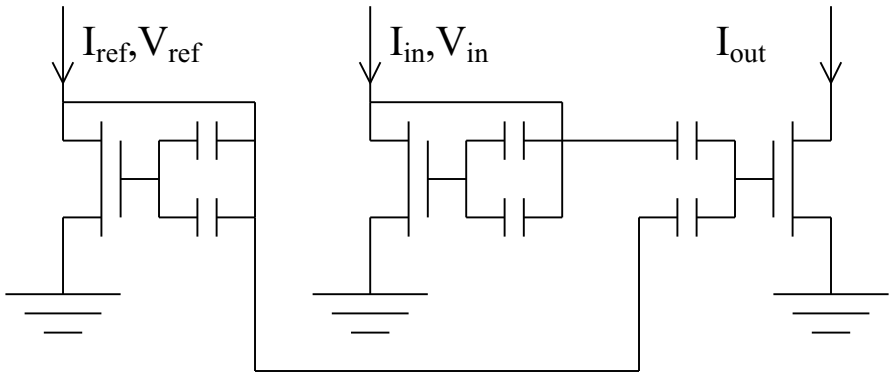


Figure 3.3: Square root circuit

3.1.2 Square Root Circuit

The equations for the square root circuit shown in figure 3.3 are shown similarly as follows:

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{ref}}{I_o}\right) = V_{ref}$$

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{in}}{I_o}\right) = V_{in}$$

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{out}}{I_o}\right) = \frac{V_{in}}{2} + \frac{V_{ref}}{2}$$

Combining the equations then gives

$$\frac{U_t}{\kappa} \ln\left(\frac{I_{out}}{I_o}\right) = \frac{U_t}{\kappa} \ln\left(\frac{I_{in}}{I_o}\right)^{\frac{1}{2}} + \frac{U_t}{\kappa} \ln\left(\frac{I_{ref}}{I_o}\right)^{\frac{1}{2}}$$

$$I_{out} = \sqrt{I_{ref} I_{in}}$$

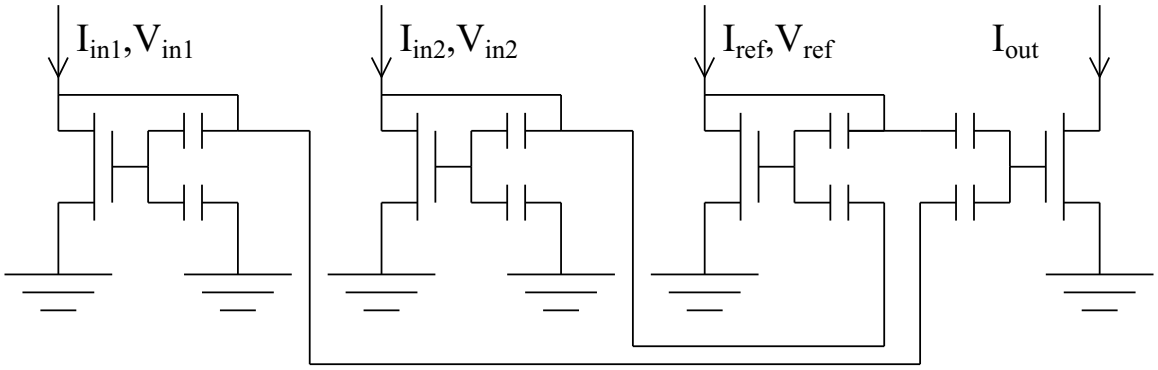


Figure 3.4: Multiplier/divider circuit

3.1.3 Multiplier/Divider Circuit

For the multiplier/divider circuit shown in figure 3.4, the calculations are again performed as follows:

$$\frac{U_t}{\kappa}\ln\left(\frac{I_{in1}}{I_o}\right) = \frac{V_{in1}}{2}$$

$$\frac{U_t}{\kappa}\ln\left(\frac{I_{in2}}{I_o}\right) = \frac{V_{in2}}{2}$$

$$\frac{U_t}{\kappa}\ln\left(\frac{I_{ref}}{I_o}\right) = \frac{V_{ref}}{2} + \frac{V_{in2}}{2}$$

$$\frac{V_{ref}}{2} = \frac{U_t}{\kappa}\ln\left(\frac{I_{ref}}{I_o}\right) - \frac{U_t}{\kappa}\ln\left(\frac{I_{in2}}{I_o}\right)$$

$$\frac{U_t}{\kappa}\ln\left(\frac{I_{out}}{I_o}\right) = \frac{V_{ref}}{2} + \frac{V_{in1}}{2}$$

$$\frac{U_t}{\kappa}\ln\left(\frac{I_{out}}{I_o}\right) = \frac{U_t}{\kappa}\left(\ln\left(\frac{I_{ref}}{I_o}\right) - \ln\left(\frac{I_{in2}}{I_o}\right) + \ln\left(\frac{I_{in1}}{I_o}\right)\right)$$

$$I_{out} = I_{ref}\frac{I_{in1}}{I_{in2}}$$

Although the input currents are labeled in such a way as to emphasize the dividing nature of this circuit, it is possible to rename the inputs such that the divisor is the reference current and the two currents in the numerator would be the multiplying inputs.

Note that the divider circuit output is only valid when I_{ref} is larger than I_{in2}. This is because the gate of the transistor with current I_{ref} is limited to the voltage $V_{fg_{ref}}$ at the gate by the current source driving I_{ref}. Since this gate is part of a capacitive voltage divider between V_{ref} and V_{in2}, when $V_{in2} > V_{fg_{ref}}$, the voltage at the node V_{ref} is zero and cannot go lower. Thus, no extra charge is coupled onto the output transistors. Thus, when $I_{in2} > I_{ref}$, $I_{out} \approx I_{in1}$.

Also, from the input/output relation, it would seem that I_{in1} and I_{ref} are interchangeable inputs. Unfortunately, based on the previous discussion, whenever $I_{in2} > I_{in1}$, $I_{out} \approx I_{ref}$. This would mean that no divisions with fractional outputs of $\frac{I_{in1}}{I_{in2}} < 1$ could be performed. When such a fractional output is desired, the use of the circuit in this configuration would not perform properly.

The preceding discussion indicates that the final simplified input/output relation of the circuit is slightly deceptive. Care must be taken to ensure that the circuit is operated in the proper operating region with appropriately sized inputs and reference currents. Furthermore, when doing the analysis for similar power law circuits the circuit designer must be aware of the charge limitations at the individual floating gates.

A more accurate input/output relation for the multiplier/divider circuit is thus given as follows:

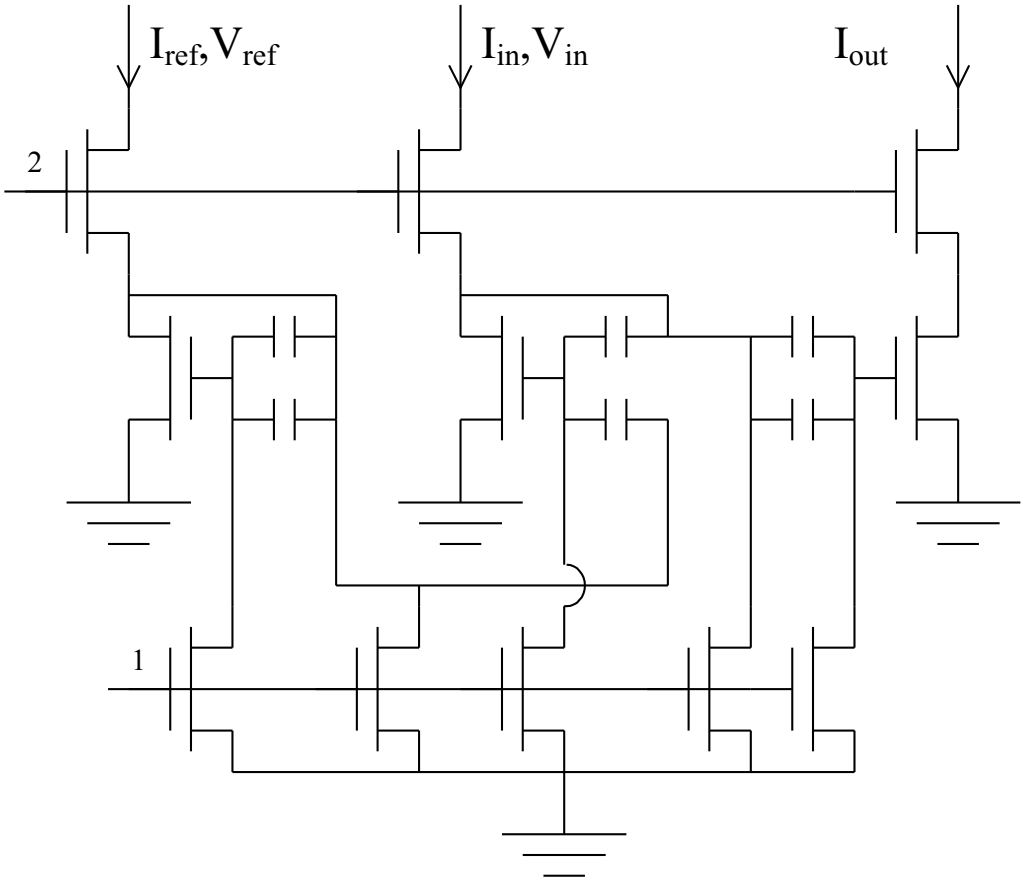


Figure 3.5: Dynamically restored squaring circuit

$$I_{out} = \begin{cases} I_{ref}\frac{I_{in1}}{I_{in2}} & \text{if} \quad I_{in2} < I_{ref} \\ I_{in1} & \text{if} \quad I_{in2} > I_{ref} \end{cases}$$

3.2 Dynamic Gate Charge Restoration

All of the above circuits assume that some means is available to initially set the gate charge level so that when all currents are set to zero, the gate voltages are also zero. One method of doing so which lends itself well to actual circuit implementation is that of using switches to dynamically set the charge during one phase of operation, and then to allow the circuit to perform computations during a second phase of operation.

The squaring circuit in figure 3.5 is shown with switches now added to dynamically equalize the charge. A non-overlapping clock generator generates the two clock

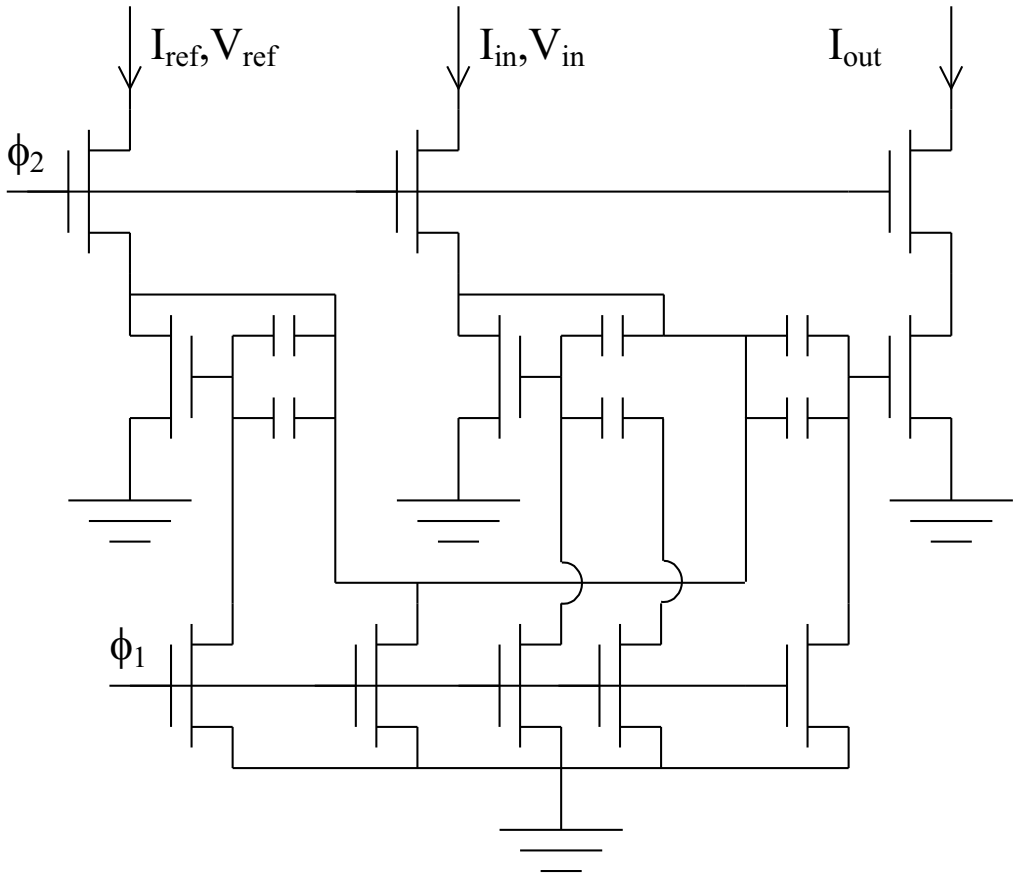


Figure 3.6: Dynamically restored square root circuit

signals. During the first phase of operation, ϕ_1 is high and ϕ_2 is low. This is the reset phase of operation. Thus, the input currents do not affect the circuit and all sides of the capacitors are discharged and the floating gates of the transistors are also grounded and discharged. This establishes an initial condition with no current through the transistors corresponding to zero gate voltage. Then, during the second phase of operation, ϕ_1 goes low and ϕ_2 goes high. This is the compute phase of operation. The transistor gates are allowed to float, and the input currents are reapplied. The circuit now exactly resembles the aforementioned floating gate squaring circuit. Thus, it is able to perform the necessary computation.

The square root (figure 3.6) and multiplier/divider circuit (figure 3.7) are also constructed in the same manner by adding switches connected to ϕ_2 at the drains of each of the transistors and switches connected to ϕ_1 at each of the floating gate and

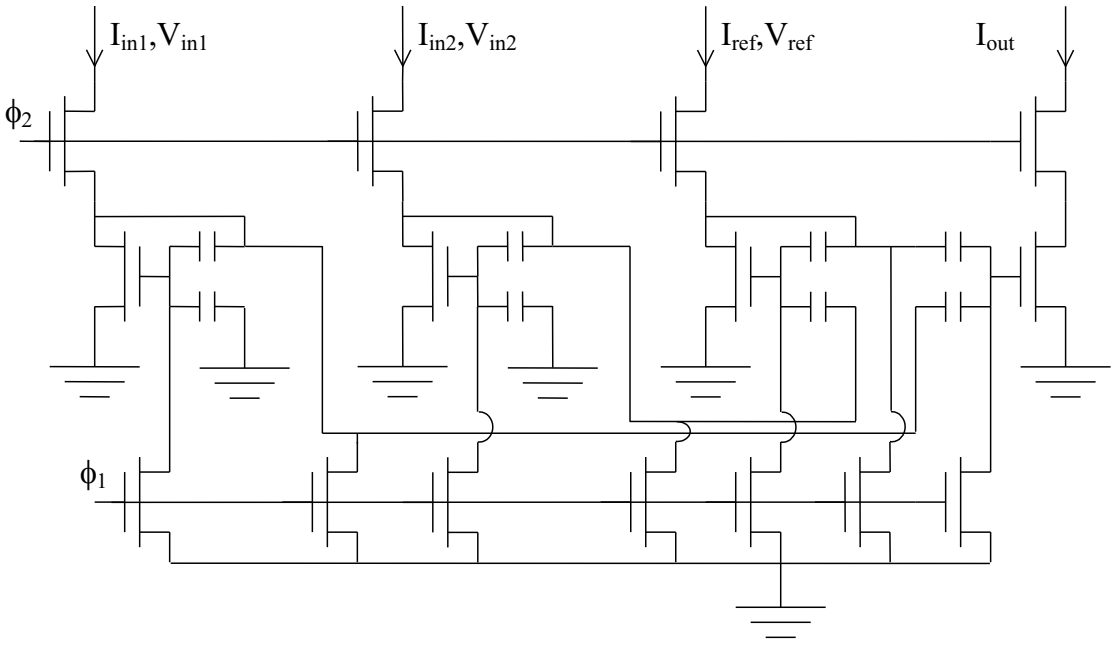


Figure 3.7: Dynamically restored multiplier/divider circuit

capacitor terminals.

3.3 Root Mean Square (Vector Sum) Normalization Circuit.

The above circuits can be used as building blocks and combined with current mirrors to perform a number of useful computations. For example, a normalization stage can be made which normalizes a current by the square root of the sum of the squares of other currents. Such a stage is useful in many signal processing tasks. This normalization stage is seen in figure 3.8. The reference current, I_{ref}, is mirrored to all of the reference inputs of the individual stages. The reference current is doubled with the 1:2 current mirror into the divider stage. This is necessary because the reference current must be larger than the largest current that will be divided by. Since the current that is being divided will be the square root of a sum of squares of two currents, when $I_{in1} = I_{in2} = I_{max}$, it is necessary to make sure that the reference

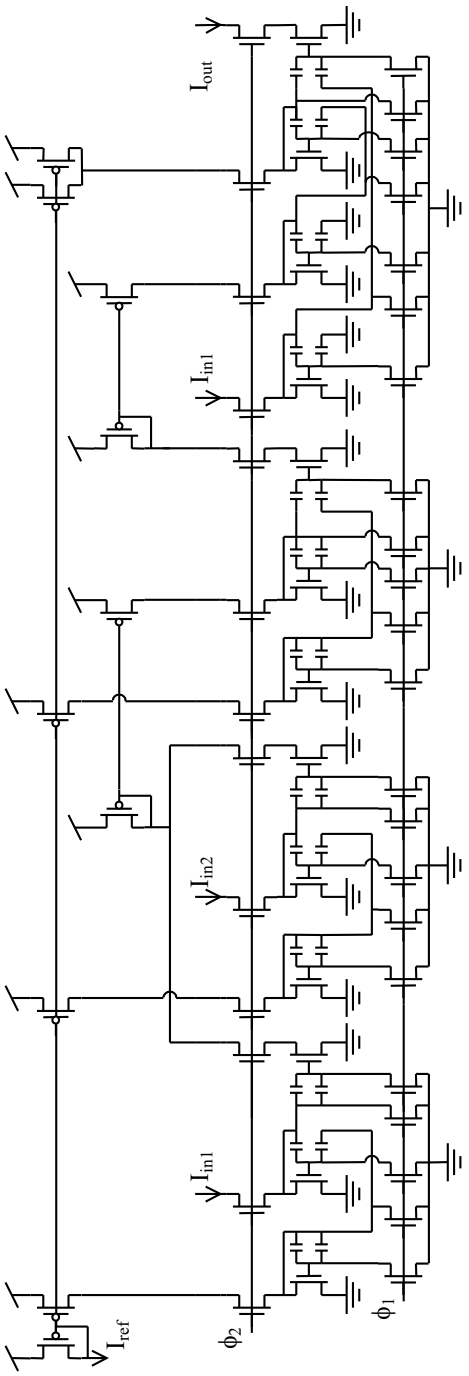


Figure 3.8: Root mean square (vector sum) normalization circuit

current for the divider section is greater than $\sqrt{2}I_{max}$. Using the 1:2 current mirror and setting $I_{ref} = I_{max}$, the reference current in the divider section will be $2I_{max}$, which is sufficient to enforce the condition.

The first two stages that read the input currents are the squaring circuit stages. The outputs of these stages are summed and then fed back into the square root stage with a current mirror. The input to the square root stage is thus given by

$$\frac{I_{in1}^2}{I_{ref}} + \frac{I_{in2}^2}{I_{ref}}$$

The square root stage then performs the computation

$$\sqrt{I_{ref}\left(\frac{I_{in1}^2}{I_{ref}} + \frac{I_{in2}^2}{I_{ref}}\right)} = \sqrt{I_{in1}^2 + I_{in2}^2}$$

The output of the square root stage is then fed into the multiplier/divider stage as the divisor current. The reference current for the divider stage is twice the reference current for the other stages as discussed before. The other input to the divider stage will be a mirrored copy of one of the input currents. Thus, the output of the divider stage is given by

$$2I_{ref}\frac{I_{in1}}{\sqrt{I_{in1}^2 + I_{in2}^2}}$$

The output can then be fed back through another 2:1 current mirror (not shown) to remove the factor of 2. Thus, the overall transfer function computed would be

$$I_{out} = I_{ref}\frac{I_{in1}}{\sqrt{I_{in1}^2 + I_{in2}^2}}$$

The addition of other divider stages can be used to normalize other input currents.

This circuit easily extends to more variables by adding more input squaring stages and connecting them all to the input of the current mirror that outputs to the square root stage.

3.4 Experimental Results

The above circuits were fabricated in a 1.2μm double poly CMOS process. All of the pfet transistors used for the mirrors were W=16.8μ, L=6μ. The nfet switches were all W=3.6μ, L=3.6μ. The floating gate nfets were all W=30μ, L=30μ. The capacitors were all 2.475 pF.

The data gathered from the current squaring circuit, square root circuit and multiplier/divider circuit is shown in figures 3.9, 3.10, and 3.11, respectively. Figure 3.12 shows the data from the vector sum normalization circuit.

The solid line in the figures represents the ideal fit. The circle markers represent the actual data points.

The circuits show good performance over several orders of magnitude in current range. At the high end of current, the circuit deviates from the ideal when one or more transistors leaves the subthreshold region. The subthreshold region for these transistors is below approximately 100nA. For example, in figure 3.10, an offset from the theoretical fit for the I_{ref} = 100nA case is seen. Since the reference transistor is leaving the subthreshold region, more gate voltage is necessary per amount of current. This is because in above threshold, the current goes as the square of the voltage; whereas, in subthreshold it follows the exponential characteristic. Since the diode connected reference transistor sets up a slightly higher gate voltage, more charge and, hence, more voltage is coupled onto the output transistor. This causes the output current to be slightly larger than expected.

Extra current range at the high end can be achieved by increasing the W/L of the transistors so that they remain subthreshold at higher current levels. The current W/L is 1, increasing W/L to 10 would change the subthreshold current range of these circuits to be below approximately 1μA and hence increase the high end of the dynamic range appropriately. Leakage currents limit the low end of the dynamic range.

Some of the anomalous points, such as those seen in figures 3.9 and 3.11, are believed to be caused by autoranging errors in the instrumentation used to collect

the data.

The divider circuit, as previously discussed, does not perform the division after $I_{in2} > I_{ref}$, and instead outputs I_{in1} as is seen in the figure.

The normalization circuit shows very good performance. This is because the reference current and the maximum input currents were chosen to keep the divider and all circuits within the proper subthreshold operating regions of the building block circuits. This is helped by the compressive nature of the function.

The reference current for the normalization circuit, I_{ref}, at the input of the reference current mirror array was set to 10nA. However, the ideal fit required a value of 14nA to be used. This was not seen in the other circuits, thus it is assumed that this is due to the Early effect of the current mirrors. In fact, the SPICE simulations of the circuit also predict the value of 14nA. Therefore, it is possible to use the SPICE simulations to change the mirror transistor ratios to obtain the desired output. Since this is merely a multiplicative effect, it is possible to simply scale the W/L of the final output mirror stage to correct it. Alternatively, it is possible to increase the length of the mirror transistors to reduce Early effect or to use a more complicated mirror structure such as a cascoded mirror.

3.5 Discussion

3.5.1 Early Effects

Although setting the floating gates near the circuit ground has been the goal of the dynamic technique discussed above, it is possible to allow the floating gate values to be reset to something other than the circuit ground. The main effect this has is to change the dynamic range. If the floating gates are reset to a value higher than ground, the transistors will enter the above threshold region with smaller input currents. In fact, in the circuits previously discussed, the switches will have a small voltage drop across them and the floating gates are actually set slightly above ground. It may also be possible to use a reference voltage smaller than ground in order to set

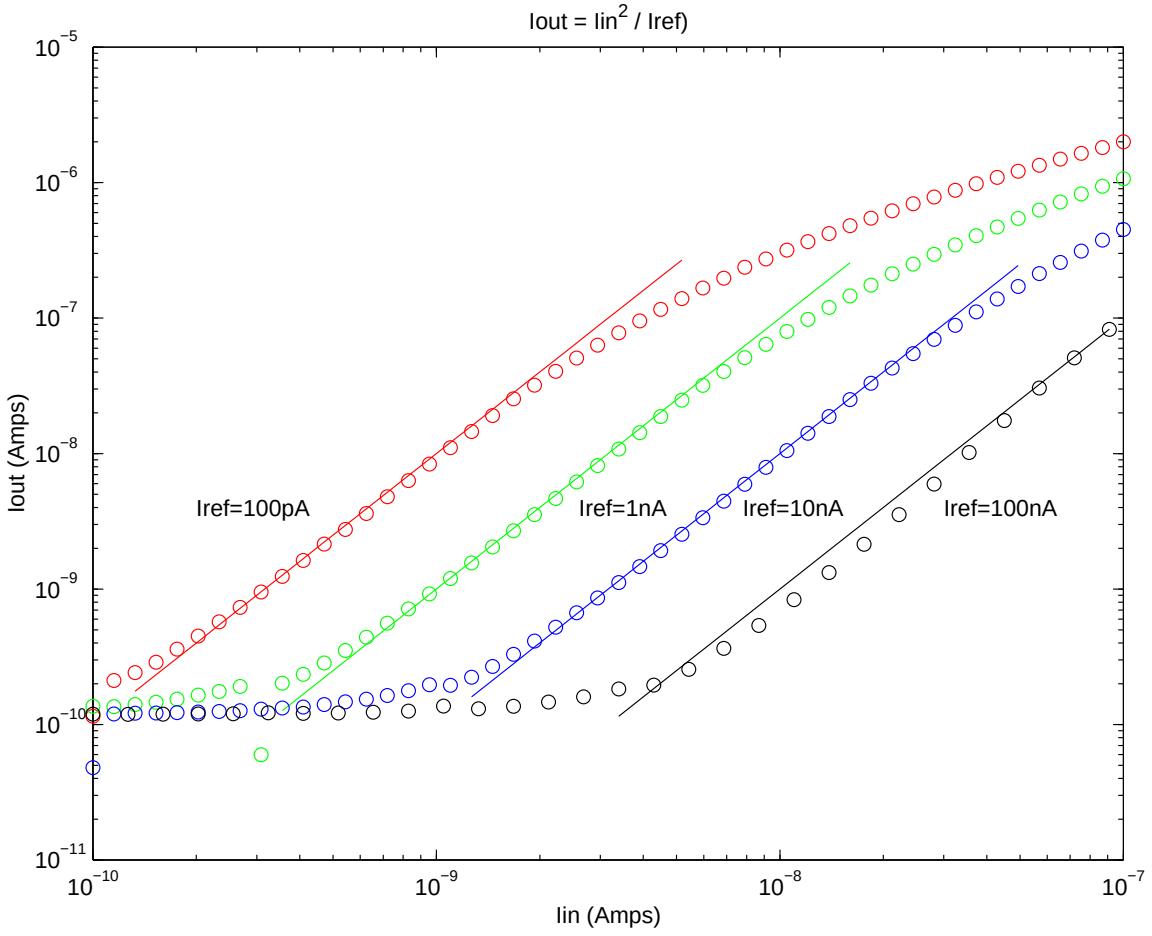


Figure 3.9: Squaring circuit results

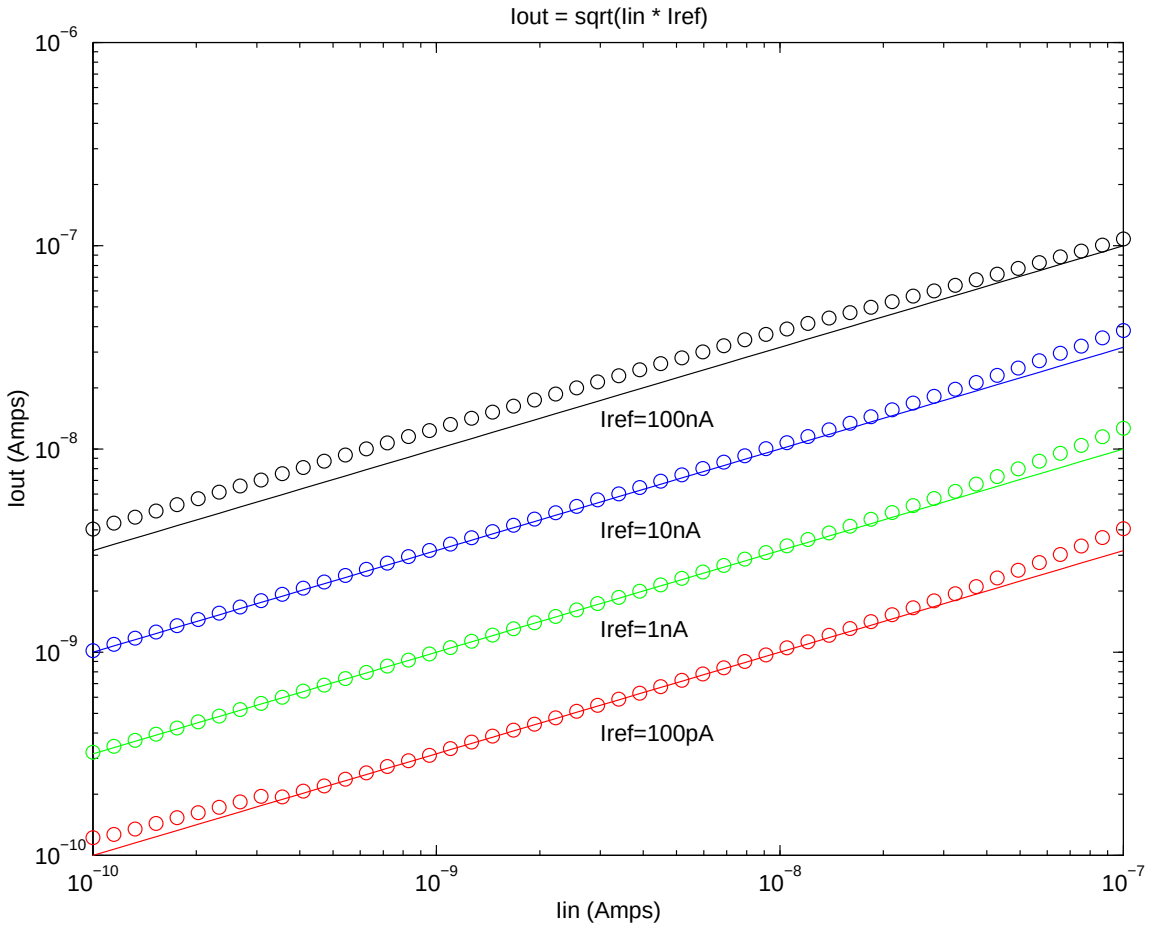


Figure 3.10: Square root circuit results

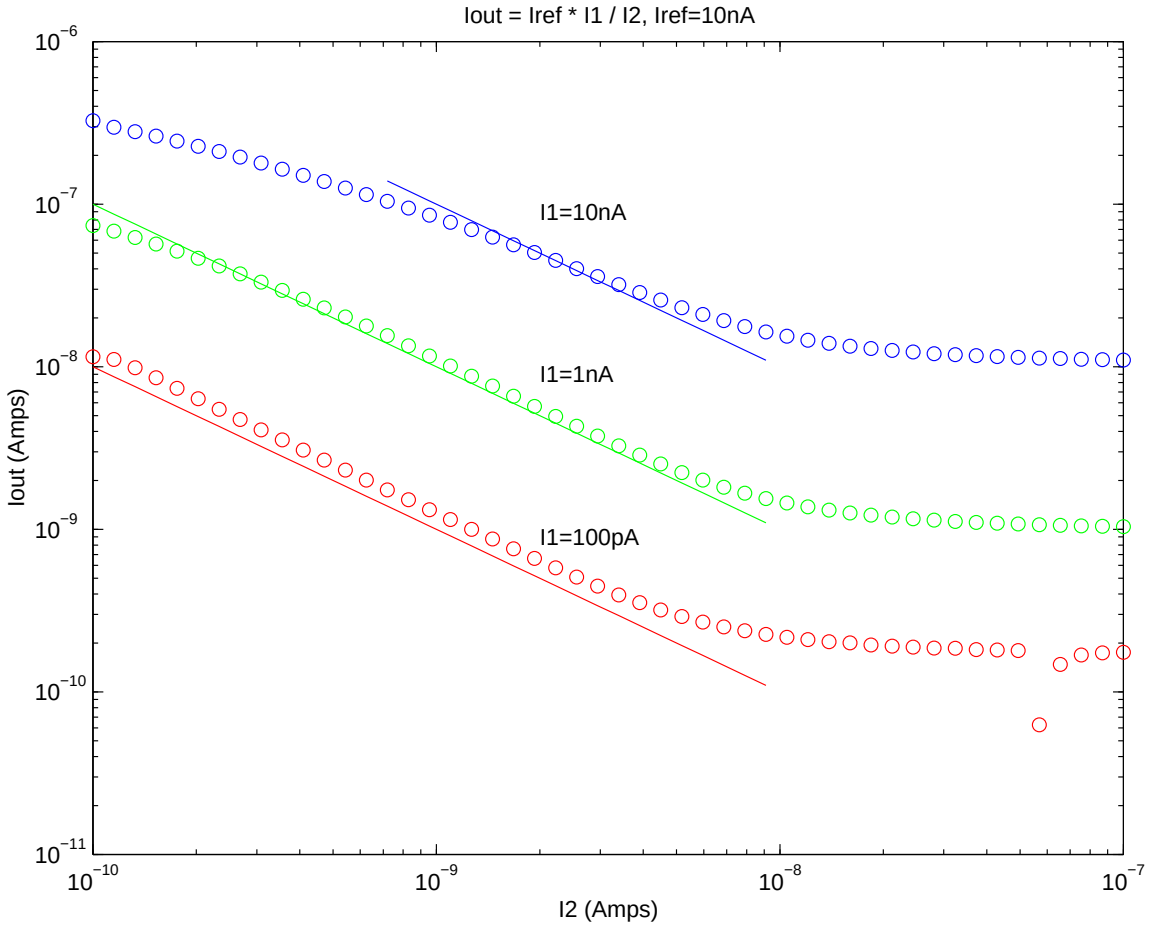


Figure 3.11: Multiplier/divider circuit results

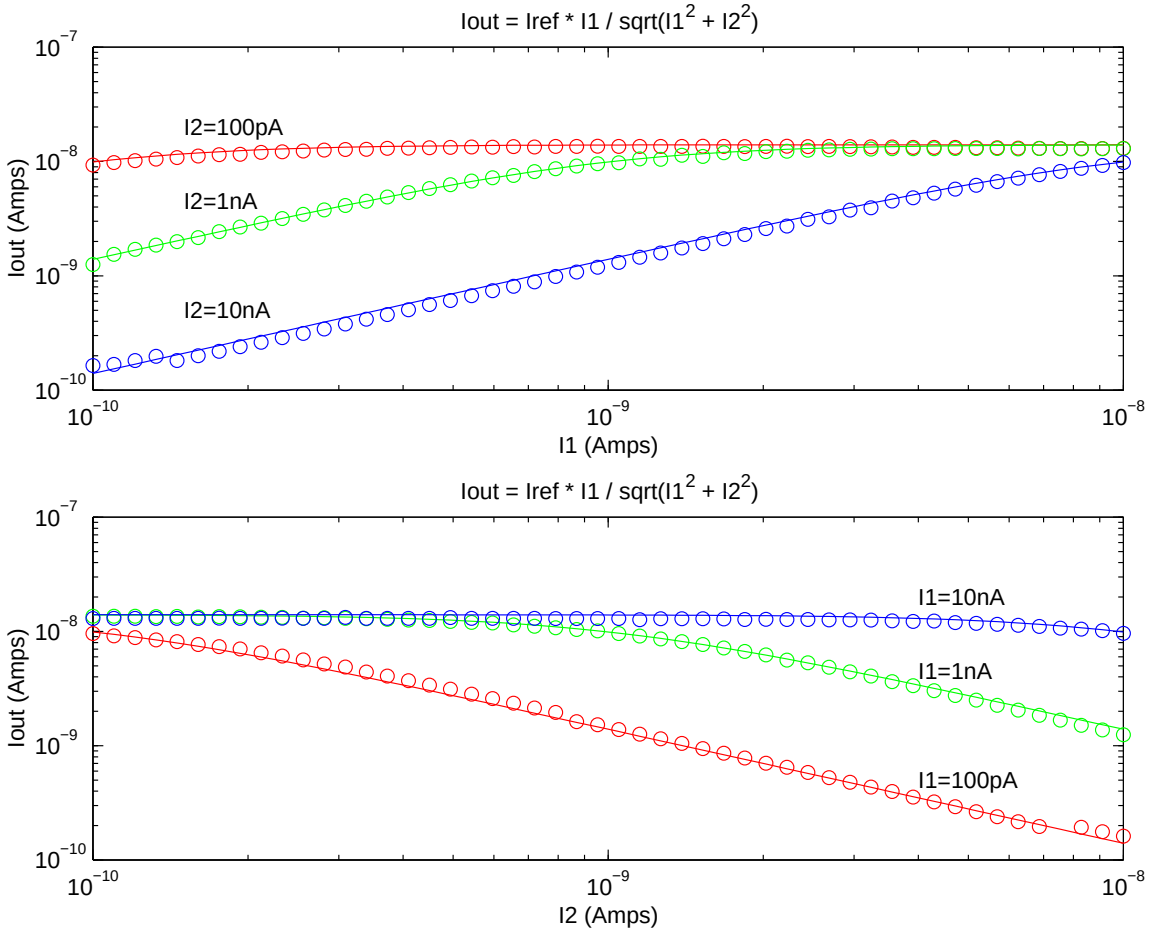


Figure 3.12: RMS normalization circuit results

the floating gates in such a way as to increase the dynamic range.

One problem with these circuits is the presence of a large Early effect. This effect comes from two sources. First, the standard Early effect comes from an increase in channel current due to channel length modulation[30]. This can be modeled by an extra linear term that is dependent on the drain-source voltage, V_{ds}.

$$I_{ds} = I_o e^{\left(\frac{\kappa V_{gs}}{U_t}\right)} \left(1 + \frac{V_{ds}}{V_o}\right)$$

The voltage V_o, called the Early voltage, depends on process parameters and is directly proportional to the channel length, L. Another aspect of the Early effect which is important for circuits with floating gates is due to the overlap capacitance of the gate to the drain/source region of the transistor[15]. In these circuits, the main overlap capacitance of interest is the floating gate to drain capacitance, C_{fg-d}. This also depends on process parameters and is directly proportional to the width, W. This parasitic capacitance acts as another input capacitor from the drain voltage, V_d. In this way, the drain voltage couples onto the floating gate by an amount $V_d \frac{C_{fg-d}}{C_T}$, where C_T is the total input capacitance. Since this is coupled directly onto gate, this has an exponential effect on the current level.

The method chosen to overcome these combined Early effects is to first make the transistors long. If L is sufficiently long, then the Early voltage, V_o, will be large and this will reduce the contribution of the linear Early effect. This also requires increasing the width, W, by the same amount as the length, L, to keep the same subthreshold current levels and not lose any dynamic range. However, this increase in W increases C_{fg-d} which worsens the exponential Early effect. Thus, it is necessary to further increase the size of the input capacitors to make the total input capacitance large compared to the floating gate to drain parasitic capacitance to reduce the impact of the exponential Early effect. In this way, it is possible to trade circuit size for accuracy.

It may be possible to use some of the switches as cascode transistors to reduce the Early effect in the output transistors of the various stages. The input and reference

transistors already have their drain voltages fixed due to the diode connection and would not benefit from this, but the output transistor drains are allowed to go near V_{dd}. This would involve not setting ϕ_2 all the way to V_{dd} during the computation and instead setting it to some lower cascode voltage. This may allow a reduction in the size of the transistors and capacitors. Furthermore, it is important to make the input capacitors large enough that other parasitic capacitances are very small compared to them.

3.5.2 Clock Effects

Unlike switched capacitor circuits that require a very high clock rate compared to the input frequencies, the clock rate for these circuits is determined solely by the leakage rate of the switch transistors. Thus, it is possible to make the clock as slow as 1 Hz or slower. The input can change faster than the clock rate and the output will be valid during most of the computation phase.

The output does require a short settling period due to the presence of glitches in the output current from charge injection by the switches. The amount of charge injection is determined by the area of the switches and is minimized with small switches. Also making the input capacitors large with respect to the size of the switches will reduce the effects of the glitches by making the injected charge less effective at inducing a voltage on the gate.

It may be possible to use a current mode filter or use two transistors in complementary phase at the output to compensate for the glitches[18].

Other techniques are also available which can improve matching characteristics and reduce the size of the circuits. One such technique would involve using a single transistor with a multiphase clock that can be used as a replacement for all the input and output transistors in the building blocks[18].

3.5.3 Speed, Accuracy Tradeoffs

It is necessary to increase the size of the input capacitors and reduce the size of the switches in order to improve the accuracy and decrease the effect of glitches. However, this also has an effect on the speeds at which the circuits may be operated and the settling times of the signals. Subthreshold circuits are already limited to low currents which in turn limits their speed of operation since it takes longer to charge up a capacitor or gate capacitance with small currents. Thus, the increases in input capacitance necessary to increase accuracy further limits the speed of operation. Furthermore, the addition of a switch along the current path reduces the speed of the circuits because of the effective on-resistance of the switch.

The switches feeding into the capacitor inputs can be viewed as a lowpass R-C filter at the inputs of the circuits. It is possible to make the switches larger in order to decrease the effective resistance, but this will increase the parasitic capacitance and require an increase in the size of the input capacitors to obtain the same accuracy. Thus, there is an inherent tradeoff between the speed of operation and the accuracy of the circuits.

3.6 Floating Gate Versus Dynamic Comparison

As previously discussed, it is possible to use these circuits in either the purely floating gate version or in the dynamic version. In this section, a brief comparison is made between the advantages and disadvantages of the two design styles.

Floating gate advantages

- Larger dynamic range - The switches of the dynamic version introduce leakage currents and offsets which reduce dynamic range.
- No glitches - The floating gate version does not suffer from the appearance of glitches on the output and, thus, does not have the same settling time requirements.

- Constant operation - The output is always valid.
- Better matching with smaller input capacitors - There are fewer parasitics because of the lack of switches, thus, it is possible to reduce the size of input capacitors and still have good matching.
- Less area - There are fewer transistors because of the lack of switches and also smaller input capacitors may be used.
- Less power - It is possible to go to smaller current values and, also, since there are fewer stacked transistors, smaller operating voltages are possible. Also, power is saved by not operating any switches.
- Faster - Since it is possible to use smaller input capacitors and there is no added switch resistance along the current path, it is possible to operate the circuits at higher speeds.

Dynamic advantages

- No UV radiation required - In many cases it is desirable not to introduce an extra post processing step in order to operate the circuits. Furthermore, in some instances, ultraviolet irradiation fails to place these circuits into a proper subthreshold operating range and instead places the floating gate in a state where the transistors are biased above threshold.
- Decreased packaging costs - Packaging costs are increased when it is necessary to accommodate a UV transparent window[19].
- Robust against long term drift - Transistor threshold drift is a known problem in CMOS circuits[20] and Flash EEPROMs[19]. Much of the threshold shift is attributed to hot electron injection. These electrons get stuck in trap states in the oxide, thereby causing the shift. Much effort is placed into reducing the number of oxide traps to minimize this effect. The electrons can then exit the oxide and normally contribute to a very small gate current. However, in floating

gate circuits, the electrons are not removed and remain as excess charge which leads to increased threshold drift. Furthermore, as process scales continue to decrease, direct gate tunneling effects will further exacerbate the problem. The dynamically restored circuits remove these excess charges during the reset phase of operation and provide greater immunity against long term drift.

Although the advantages that the pure floating gate circuits provide outnumber those of the dynamic circuits, the advantages afforded by the dynamic technique are of such significant importance that many circuit designers would choose to utilize the technique. In practice, a full consideration of each style's constraints with respect to the overall system design criteria would favor one or the other of the two design styles.

3.7 Conclusion

A set of circuits for analog circuit design that may be useful for analog computation circuits and neural network circuits has been presented. It is hoped that it is clear from the derivations how to obtain other power law circuits that may be necessary and how to combine them to perform useful complex calculations. The dynamic charge restoration technique is shown to be a useful implementation of this class of analog circuits. Furthermore, the dynamic charge restoration technique may be applied to other floating gate computational circuits that may otherwise require initial ultraviolet illumination or other methods to set the initial conditions.

Chapter 4 VLSI Neural Network Algorithms, Limitations, and Implementations

4.1 VLSI Neural Network Algorithms

4.1.1 Backpropagation

Early work in neural networks was hindered by the problem that single layer networks could only represent linearly separable functions[41]. Although it was known that multilayer networks could learn any boolean function and, hence, did not suffer from this limitation, the lack of a learning rule to program multilayered networks reduced interest in neural networks for some time[40]. The backpropagation algorithm[42], based on gradient descent, provided a viable means for programming multilayer networks and created a resurgence of interest in neural networks.

A typical sum-squared error function used for neural networks is defined as follows:

$$E(\vec{w}) = \frac{1}{2}\sum_i (T_i - O_i)^2$$

where T_i is the target output for input pattern i from the training set, and O_i is the respective network output. The network output is a composite function of the network weights and composed of the underlying neuron sigmoidal function, $s(.)$.

In backpropagation, weight updates, $\Delta w_{ij} \propto \frac{\partial E}{\partial w_{ij}}$, are calculated for each of the weights. The functional form of the update involves the weights themselves propagating the errors backwards to more internal nodes, which leads to the name backpropagation. Furthermore, the functional form of the weight update rules are composed of derivatives of the neuron sigmoidal function. Thus, for example, if

$s(x) = \tanh(ax)$, then the weight update rule would involve functions of the form $s'(x) = 2a\,(\tanh(ax))\,(1 - \tanh(ax))$. For software implementations of the neural network, the ease of computing this type of function allows an effective means to implement the backpropagation algorithm.

However, if the hardware implementation of the sigmoidal function tends to deviate from an ideal function such as the tanh function, it becomes necessary to compute the derivative of the function actually being implemented which may not have a simple functional form. Otherwise, unwanted errors would get backpropagated and accumulate. Furthermore, deviations in the multiplier circuits can add offsets which would also accumulate and significantly reduce the ability of the network to learn. For example, in one implementation, it was discovered that offsets of 1% degraded the learning success rate of the XOR problem by 50%[43]. In typical analog VLSI implementations, offsets much larger than 1% are quite common, which would lead to seriously degraded operation.

4.1.2 Neuron Perturbation

Because of the problems associated with analog VLSI backpropagation implementations, efforts have been made to find suitable learning algorithms which are independent of the model of the neuron[44]. For example, in the above example the neuron was modeled as a tanh function. In a model-free learning algorithm, the functional form of the neuron, as well as the functional form of its derivative, does not enter into the weight update rule. Furthermore, analog implementations of the derivative can prove difficult and implementations which avoid these difficulties are desirable. Most of the algorithms developed for analog VLSI implementation utilize stochastic techniques to approximate the gradients rather than to directly compute them.

One such implementation[45], called the Madaline Rule III (MR III), approximates a gradient by applying a perturbation to the input of each neuron and then measuring the difference in error, $\Delta E = E$(with perturbation)-E(without perturbation). This error is then used for the weight update rule.

$$\Delta w_{ij} = -\eta \frac{\Delta E}{\Delta \text{net}_i} x_j$$

where net_i is the weighted, summed input to the neuron i that is perturbed, and x_j is either a network input or a neuron output from a previous layer which feeds into neuron i.

This weight update rule requires access to every neuron input and every neuron output. Furthermore, multiplication/division circuitry would be required to implement the learning rule. However, no explicit derivative hardware is required, and no assumptions are made about the neuron model.

Another variant of this method, called the virtual targets method and which is not model-free, required the use of the neuron function derivatives in the weight update rule[47]. The method showed good robustness against noise, but studies of the effects of inaccurate derivative approximations were not done.

4.1.3 Serial Weight Perturbation

Another implementation which is more optimal for analog implementation involves serially perturbing the weights rather than perturbing the neurons[23]. The weight updates are based on a finite difference approximation to the gradient of the mean squared error with respect to a weight. The weight update rule is given as follows:

$$\Delta w_{ij} = \frac{E\left(w_{ij} + \text{pert}_{ij}\right) - E\left(w_{ij}\right)}{\text{pert}_{ij}}$$

This is the forward difference approximation to the gradient. For small enough perturbations, pert_{ij}, a reasonable approximation is achieved. Smaller perturbations provide better approximations to the gradient at the expense of very small steps in the gradient direction which then requires many extra iterations to find the minimum. Thus, a natural tradeoff between speed and accuracy exists.

The gradient approximation can also be improved with the central difference ap-

proximation:

$$\Delta w_{ij} = \frac{E\left(w_{ij} + \frac{\text{pert}_{ij}}{2}\right) - E\left(w_{ij} - \frac{\text{pert}_{ij}}{2}\right)}{\text{pert}_{ij}}$$

However, this requires more feedforward passes of the network to obtain the error terms. A usual implementation would use a weight update rule of the following form:

$$\Delta w_{ij} = -\frac{\eta}{\text{pert}}\left(E\left(w_{ij} + \text{pert}_{ij}\right) - E\left(w_{ij}\right)\right)$$

Since pert and η are both small constants, the above weight update rule merely involves changing the weights by the error difference scaled by a small constant.

It is clear that this implementation should require simpler circuitry than the neuron perturbation algorithm. The multiplication/division step is replaced by scaling with a constant. Also, there is no need to provide access to the computations of internal nodes. Studies have also shown that this algorithm performs better than neuron perturbation in networks with limited precision weights[46].

4.1.4 Summed Weight Neuron Perturbation

The summed weight neuron perturbation approach[24] combines elements from both the neuron perturbation technique and the serial weight perturbation technique. Since the serial weight perturbation technique applies perturbations to the weights one at a time, its computational complexity was shown to be of higher order than that of the neuron perturbation technique.

Applying a weight perturbation to each of the weights connected to a particular neuron is equivalent to perturbing the input of that neuron as is done in neuron perturbation. Thus, in summed weight neuron perturbation, weight perturbations are applied for each particular neuron, the error is then calculated and weight updates are made using the same weight update rule as for serial weight perturbation.

This allows for a reduction in the number of feedforward passes, yet still does not require access to the outputs of internal nodes of the network. However, this speed

up is at the cost of increased hardware to store weight perturbations. Normally, the weight perturbations are kept at a fixed small size, but the sign of the perturbation is random. Thus, the weight perturbation storage can be achieved with 1 bit per weight.

4.1.5 Parallel Weight Perturbation

The parallel weight perturbation techniques[21][25][48] extend the above methods by simultaneously applying perturbations to all weights. The perturbed error is then measured and compared to the unperturbed error. The same weight update rule is then used as in the serial weight perturbation method. Thus, all of the weights are updated in parallel.

If there are W total weights in a network, a serial weight perturbation approach gives a factor W reduction in speed over a straight gradient descent approach because the overall gradient is calculated based on W local approximations to the gradient. With parallel weight perturbation, on average, there is a factor $W^{\frac{1}{2}}$ reduction in speed over direct gradient descent[25]. This can be better understood if the parallel perturbative method is thought of in analogy to Brownian motion where there is considerable movement of the network through weight space, but with a general guiding force in the proper direction. Thus, a speedup of $W^{\frac{1}{2}}$ is achieved over serial weight perturbation. This increase in speed may not always be seen in practice because it depends upon certain conditions being met such as the perturbation size being sufficiently small.

It is also possible to improve the convergence properties of the network by averaging over multiple perturbations for each input pattern presentation[21]. This is feasible only if the speed of averaging over multiple perturbations is greater than the speed of applying input patterns. This usually holds true because the input patterns must be obtained externally; whereas, the multiple perturbations are applied locally.

Furthermore, convergence rates can be improved by using an annealing schedule where large perturbation sizes are used initially and then reduced. In this way, large

weight changes are discovered first and gradually decreased to get finer and finer resolution in approaching the minimum.

4.1.6 Chain Rule Perturbation

The chain rule perturbation method is a semi-parallel method that mixes ideas from both serial weight perturbation and neuron perturbation[26]. In a 1 hidden layer network, the output layer weights are first updated serially using the weight update formula from the serial weight perturbation method. Next, a hidden layer neuron output, n_i, is perturbed by amount npert_i. The effect on the error is measured and a partial gradient term, $\frac{\Delta E}{\text{npert}_i}$, is stored. This neuron output perturbation is repeated sequentially and a partial error gradient term stored for every neuron in the hidden layer. Then, each of the weights, w_{ij}, feeding into the hidden layer neurons are perturbed in parallel by an amount wpert_{ij}. The resultant change in hidden layer neuron outputs, Δn_i, from this parallel weight perturbation is measured and then used to calculate and store another partial gradient term, $\frac{\Delta n_i}{\text{wpert}_{ij}}$, for each weight/neuron pair. Finally, the hidden layer weights are updated in parallel according to the following rule:

$$\Delta w_{ij} = -\eta \frac{\Delta E}{\text{wpert}} = -\eta \frac{\Delta E}{\text{npert}_\text{i}} \frac{\Delta n_i}{\text{wpert}_{ij}}$$

Hence, the name of the method comes from the analog of the weight update rule to the chain rule in derivative calculus. For more hidden layers, the same hidden layer weight update procedure is repeated.

This method allows a speedup over serial weight perturbation by a factor approaching the number of hidden layer neurons at the expense of the extra hardware necessary to store the partial gradient terms. For example, in a network with i inputs, j hidden nodes, and k outputs, the chain rule perturbation method requires $k \cdot j + j + i$ perturbations; whereas, serial perturbation requires $k \cdot j + j \cdot i$ perturbations.

4.1.7 Feedforward Method

The feedforward method is a variant of serial weight perturbation that incorporates a direct search technique[49]. First, for a weight, w_{ij}, the direction of search is found by applying a fraction of the perturbation. For example if 0.1pert is applied and the error goes down, then the direction of search is positive; otherwise it is negative. Next, in the forward phase, perturbations of size pert continue to be applied in the direction of search until the error does not decrease or some predetermined maximum number of forward perturbations is reached. Then, in the backward phase, a step of size $\frac{\text{pert}}{2}$ in the reverse direction is taken. Successive backwards steps are taken with each step decreasing by another factor of 2 until the error again goes up or another predetermined maximum number of backward perturbations is reached. These steps are repeated for every weight.

This method is effectively a sophisticated annealing schedule for the serial weight perturbation method.

Many other variations and combinations of these stochastic learning rules are possible. Most of the stochastic algorithms and implementations concentrate on iterative approaches to weight updates as discussed above, but continuous-time implementations are also possible[50].

4.2 VLSI Neural Network Limitations

4.2.1 Number of Learnable Functions with Limited Precision Weights

Clearly in a neural network with a small, fixed number of bits per weight, the ability of the network to classify multiple functions is limited. Assume that each weight is comprised of b bits of information and that there are W total weights. The total number of bits in the network is given as $B = bW$. Since each bit can take on only 1 of 2 values, the total number of different functions that the network can implement is no more than 2^B. In computer simulations of neural networks where floating point

precision is used and b is on the order of 32 or 64, this is usually not a limitation to learning. However, if b is a small number like 5, then the ability of the network to learn a specific function may be compromised. Furthermore, 2^B is merely an upper bound. There are many different settings of the weights that will lead to the same function. For example, the hidden neuron outputs can be permuted since the final output of the network does not depend upon the order of the hidden layer neurons. Also, the signs of the outputs of the hidden layer neurons can also be flipped by adjusting the weights coming into that neuron and by also flipping the sign of the weight on the output of the neuron to ensure the same output. In one result, the total number of boolean functions which can be implemented is given approximately by $\frac{2^B}{N!2^N}$ for a fully connected network with N units in each layer[51].

4.2.2 Trainability with Limited Precision Weights

Several studies have been done on the effect of the number of bits per weight constraining the ability of a neural network to properly train[52][53][54]. Most of these are specific to a particular learning algorithm. In one study, the effects of the number of bits of precision on backpropagation were investigated[52]. The weight precision for a feedforward pass of the network, the weight precision for the backpropagation weight update calculation, and the output neuron quantization were all tested for a range of bits. The effect of the number of bits on the ability of the network to properly converge was determined. The empirical results showed that a weight precision of 6 bits was sufficient for feedforward operation. Also, neuron output quantization of 6 bits was sufficient. This implies that standard analog VLSI neurons should be sufficient for a VLSI neural network implementation. For the backpropagation weight update calculation, at least 12 bits were required to obtain reasonable convergence. This shows that a perturbative algorithm that relies only on feedforward operation can be effective in training with as few as half the number of bits required for a backpropagation implementation. Due to the increased hardware complexity necessary to achieve the extra number of bits, the perturbative approaches will be desirable in

analog VLSI implementations.

4.2.3 Analog Weight Storage

Due to the inherent size requirements to store and update digital weights with A/D and D/A converters, some analog VLSI implementations concentrate on reducing the space necessary for weight storage by using analog weights. Usually, this involves storing weights on a capacitor. Since the capacitors slowly discharge, it is necessary to implement some sort of scheme to continuously refresh the weight. One such scheme, which showed long term 8 bit precision, involves quantization and refresh of the weights by use of A/D/A converters which can be shared by multiple weights[60][61]. Other schemes involve periodic refresh by use of quantized levels defined by input ramp functions or by periodic repetition of input training patterns[59].

Since significant space and circuit complexity overhead exists for capacitive storage of weights, attempts have also been made at direct implementation of nonvolatile analog memory. In one such implementation, a dense array of synapses is achieved with only two transistors required per synapse[64]. The weights are stored permanently on a floating gate transistor which can be updated under UV illumination. A second drive transistor is used as a switch to sample the input drive voltage onto a drive electrode in order to adjust the floating gate voltage, and, hence, weight value. The weight multiplication with an input voltage is achieved by biasing the floating gate transistor into the triode region which produces an output current which is roughly proportionate to the product of its floating gate voltage, representing the weight, and its drain to source voltage, representing the input. The use of UV illumination also allowed a simple weight decay term to be introduced into the learning rule if necessary. Another approach which also implements a dense array of synapses based on floating gates utilized tunneling and injection of electrons onto the floating gate rather than UV illumination[28][29]. Using this technique, only one floating gate transistor per synapse was required. A learning rule was developed which balances the effects of the tunneling and injection. In another approach, a 3 transistor analog

memory cell was shown to have up to 14 bits of resolution[65]. Some of the weight update speeds obtainable using the tunneling and injection techniques may be limited because rapid writing of the floating gates requires large oxide currents which may damage the oxide. Nevertheless, these techniques can be combined with dynamic refresh techniques where weights and weight updates are quickly stored and learned on capacitors, but eventually written into nonvolatile analog memories[66].

4.3 VLSI Neural Network Implementations

Many neural network implementations in VLSI can be found in the literature. This includes purely digital approaches, purely analog approaches and hybrid analog-digital approaches. Although the digital implementations are more straightforward to build using standard digital design techniques, they often scale poorly with large networks that require many bits of precision to implement the backpropagation algorithms[57][58]. However, some size reductions can be realized by reducing the bit precision and using the stochastic learning techniques available[55][56]. Other techniques use time multiplexed multipliers to reduce the number of multipliers necessary.

An early attempt at implementation of an analog neural network involved making separate VLSI chip modules for the neurons, synapses and routing switches[62]. The modules could all be interconnected and controlled by a computer. The weights were digital and were controlled by a serial weight bus from the computer. Since the components were modular, it could be scaled to accommodate larger networks. However, each of the individual circuit blocks was quite large and hence was not well suited for dense integration on a single neural network chip. For example, the circuits for the neuron blocks consisted of three multistage operational amplifiers and three 100k resistors. The learning algorithms were based on output information from the neurons and no circuitry for local weight updates was provided.

4.3.1 Contrastive Neural Network Implementations

Some of the earliest VLSI implementations of neural networks were not based on the stochastic weight update algorithms, but on contrastive weight updates[67]. The basic idea behind a contrastive technique is to run the network in two phases. In the first phase, a teacher signal is sent, and the outputs of the network are clamped. In the second phase, the network outputs are unclamped and free to change. The weight updates are based on a contrast function that measures the difference between the clamped, teacher phase, and the unclamped, free phase.

One of the first fully integrated VLSI neural networks used one of these contrastive techniques and showed rapid training on the XOR function[33]. The chip incorporated a local learning rule, feedback connections and stochastic elements. The learning rule was based on a Boltzmann machine algorithm, which uses bidirectional, asymmetric weights[63]. Briefly, the algorithm works by checking correlations between the input and output neuron of a synapse. Correlations are checked between a teacher phase, where the output neurons are clamped to the correct output, and a student phase, where the outputs are unclamped. If the teacher and student phases share the same correlation, then no change is made to the weight. If the neurons are correlated in the teacher phase, but not the student phase, then the weight is incremented. If it is reversed, then the weight is decremented. The weights were stored as 5 bit digital words. On each pattern presentation, the network weights are perturbed with noise and the correlations counted. Although the system actually needs noise in order to learn, the weight update rule is not based on the previously mentioned stochastic weight update rules, but is an example of contrastive learning that requires randomness in order to learn. Analog noise sources were used to produce the random perturbations. Unfortunately, the gains required to amplify thermal noise caused the system to be unstable, and highly correlated oscillations were present between all of the noise sources. Thus, it was necessary to intermittently apply the noise sources in order to facilitate learning. The network showed good success with a 2:2:2:1 network learning XOR. The learning rate was shown to be over 100,000 times faster than that

achieved with simulations on a digital computer.

Another implementation showed a contrastive version of backpropagation[43]. The basic idea was to also use both a clamped phase and a free phase. The weight updates are calculated in both phases using the standard backpropagation weight update rules. The actual weight update was taken to be the difference between the clamped weight change and the free weight change. The use of the contrastive technique allowed a form of backpropagation despite circuit offsets. However, this implementation was not model free, and a derivative generator was also necessary in order to compute the weight updates. The weights were stored as analog voltages on capacitors with the eventual goal of using nonvolatile analog weights. Due to the size of the circuitry, only one layer of the network was integrated on a single chip. The multilayer network was obtained by cascading several chips together. Successful learning of the XOR function was shown.

4.3.2 Perturbative Neural Network Implementations

A low power chip implementing the summed weight neuron perturbation algorithm has been demonstrated[22]. The chip was designed for implantable applications which necessitated the need for extremely low power and small area. The weights were stored digitally and a serial weight bus was used to set the weights and weight updates. The chip was trained in loop with a computer applying the input patterns, controlling the serial weight bus and reading the output values for the error computation. To reduce power, subthreshold synapses and neurons were used. The synapses consisted of transconductance amplifiers with the weights encoded as a digital word which set the bias current. Neurons consisted of current to voltage converters using a diode connected transistor. They also included a few extra transistors to deal with common mode cancelation. A 10:6:3 network was trained to distinguish two cardiac arrythmia patterns. The chips showed a success rate with over 95% of patient data. With a 3V supply, only 200 nW was consumed. This is a good example of a situation where an analog VLSI neural network is very suitable as compared to a digital solution.

Another perturbative implementation was used for training a recurrent neural network on a continuous time trajectory[39]. A parallel perturbative method was used with analog volatile weights and local weight refresh circuitry. Two counter-propagating linear feedback shift registers, which were combined with a network of XORs, were used to digitally generate the random perturbations. The weight updates were computed locally. The global functions of error accumulation and comparison were performed off chip. Some of the weight refresh circuitry was also implemented off chip. The network was successfully trained to generate two quadrature phase sinusoidal outputs.

The chain rule perturbation algorithm has also been successfully implemented on chip[66]. The analog weights were stored dynamically and provisions were made on the chip for nonvolatile analog storage using tunneling and injection, but its use was not demonstrated. The weight updates were computed locally. Many of the global functions such as error computation were also performed on the chip. Only a few control signals from a computer were necessary for its operation. The size of the perturbations was externally settable. However, all perturbations seemed to be of the same sign, thus not requiring any specialized pseudorandom number generation. The network was shown to successfully train on XOR and other functions. However, the error function during learning showed some very peculiar behavior. Rather than gradually decreasing to a small value, the error actually increased steadily, then entered large oscillations between the high error and near zero error. After the oscillations, the error was close to zero and stayed there. The authors attribute this strange and erratic behavior to the fact that a large learning rate and a large perturbation size were used.

Chapter 5 VLSI Neural Network with Analog Multipliers and a Serial Digital Weight Bus

Training an analog neural network directly on a VLSI chip provides additional benefits over using a computer for the initial training and then downloading the weights. The analog hardware is prone to have offsets and device mismatches. By training with the chip in the loop, the neural network will also learn these offsets and adjust the weights appropriately to account for them. A VLSI neural network can be applied in many situations requiring fast, low power operation such as handwriting recognition for PDAs or pattern detection for implantable medical devices[22].

There are several issues that must be addressed to implement an analog VLSI neural network chip. First, an appropriate algorithm suitable for VLSI implementation must be found. Traditional error backpropagation approaches for neural network training require too many bits of floating point precision to implement efficiently in an analog VLSI chip. Techniques that are more suitable involve stochastic weight perturbation[21],[23],[24],[25],[26],[27], where a weight is perturbed in a random direction, its effect on the error is determined and the perturbation is kept if the error was reduced; otherwise, the old weight is restored. In this approach, the network observes the gradient rather than actually computing it.

Serial weight perturbation[23] involves perturbing each weight sequentially. This requires a number of iterations that is directly proportional to the number of weights. A significant speed-up can be obtained if all weights are perturbed randomly in parallel and then measuring the effect on the error and keeping them all if the error reduces. Both the parallel and serial methods can potentially benefit from the use of annealing the perturbation. Initially, large perturbations are applied to move the weights

quickly towards a minimum. Then, the perturbation sizes are occasionally decreased to achieve finer selection of the weights and a smaller error. In general, however, optimized gradient descent techniques converge more rapidly than the perturbative techniques.

Next, the issue of how to appropriately store the weights on-chip in a non-volatile manner must be addressed. If the weights are simply stored as charge on a capacitor, they will ultimately decay due to parasitic conductance paths. One method would be to use an analog memory cell [28],[29]. This would allow directly storing the analog voltage value. However, this technique requires using large voltages to obtain tunneling and/or injection through the gate oxide and is still being investigated. Another approach would be to use traditional digital storage with EEPROMs. This would then require having A/D/A (one A/D and one D/A) converters for the weights. A single A/D/A converter would only allow a serial weight perturbation scheme that would be slow. A parallel scheme, which would perturb all weights at once, would require one A/D/A per weight. This would be faster, but would require more area. One alternative would remove the A/D requirement by replacing it with a digital counter to adjust the weight values. This would then require one digital counter and one D/A per weight.

5.1 Synapse

A small synapse with one D/A per weight can be achieved by first making a binary weighted current source (Figure 5.1) and then feeding the binary weighted currents into diode connected transistors to encode them as voltages. These voltages are then fed to transistors on the synapse to convert them back to currents. Thus, many D/A converters are achieved with only one binary weighted array of transistors. It is clear that the linearity of the D/A will be poor because of matching errors between the current source array and synapses which may be located on opposite sides of the chip. This is not a concern because the network will be able to learn around these offsets.

The synapse[26],[22] is shown in figure 5.2. The synapse performs the weighting

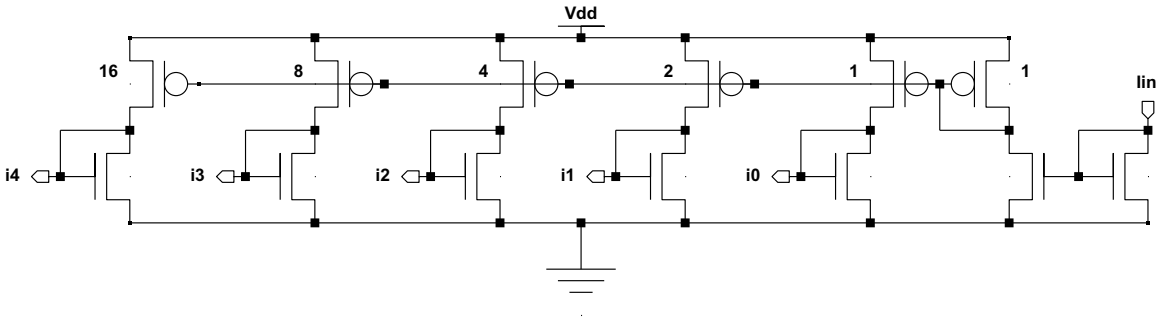


Figure 5.1: Binary weighted current source circuit

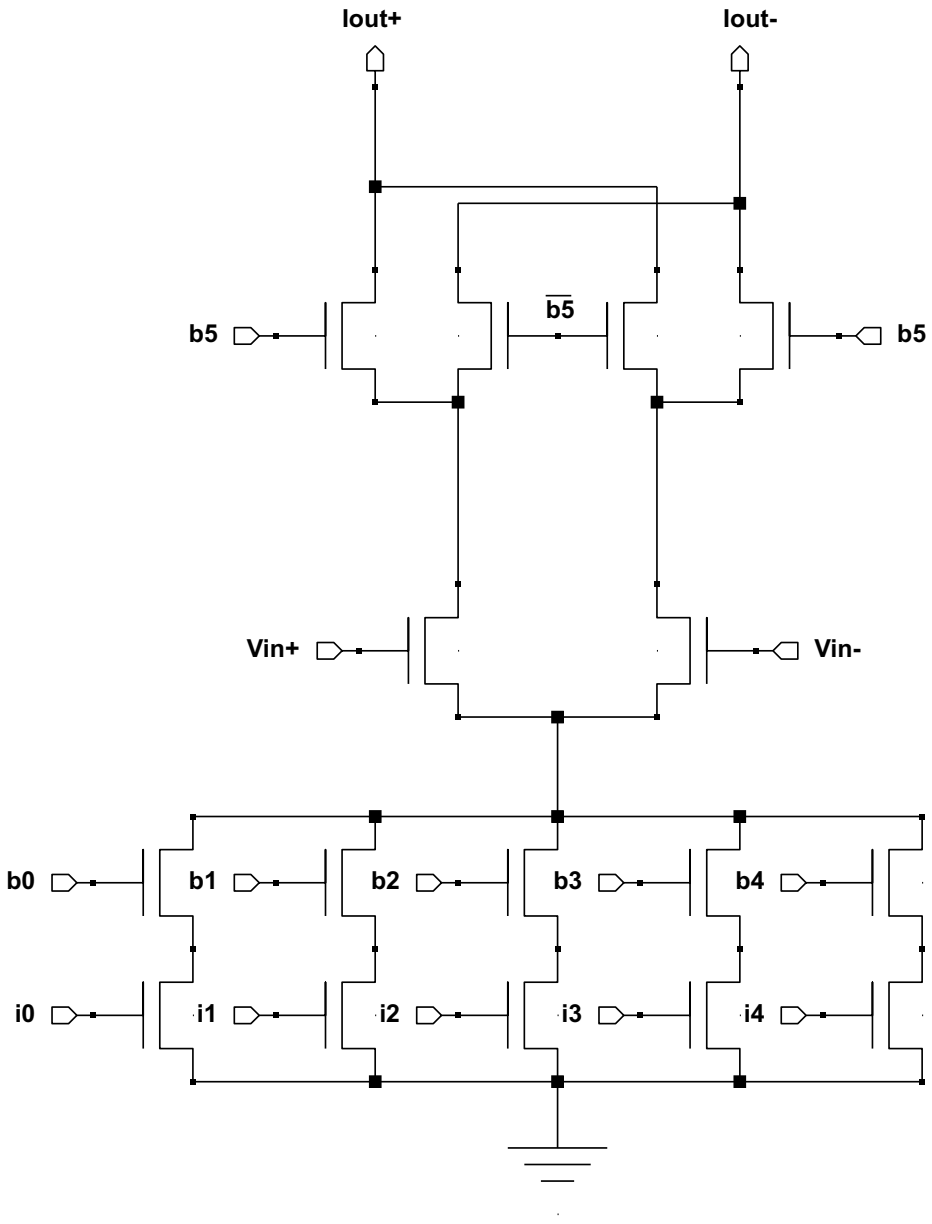


Figure 5.2: Synapse circuit

of the inputs by multiplying the input voltages by a weight stored in a digital word denoted by b0 through b5. The sign bit, b5, changes the direction of current to achieve the appropriate sign.

In the subthreshold region of operation, the transistor equation is given by[30]

$$I_d = I_{d0} e^{\kappa V_{gs}/U_t}$$

and the output of the synapse is given by[30],[22]

$$\Delta I_{out} = I_{out+} - I_{out-} = \begin{cases} +I_0 W \tanh\left(\frac{\kappa(V_{in+} - V_{in-})}{2U_t}\right) \\ b5 = 1 \\ -I_0 W \tanh\left(\frac{\kappa(V_{in+} - V_{in-})}{2U_t}\right) \\ b5 = 0 \end{cases}$$

where W is the weight of the synapse encoded by the digital word and I_0 is the least significant bit (LSB) current.

Thus, in the subthreshold linear region, the output is approximately given by

$$\Delta I_{out} \approx g_m \Delta V_{in} = \frac{\kappa I_0}{2U_t} W \Delta V_{in}$$

In the above threshold regime, the transistor equation in saturation is approximately given by

$$I_D \approx K(V_{gs} - V_t)^2$$

The synapse output is no longer described by a simple tanh function, but is nevertheless still sigmoidal with a wider "linear" range.

In the above threshold linear region, the output is approximately given by

$$\Delta I_{out} \approx g_m \Delta V_{in} = 2\sqrt{K I_0}\sqrt{W}\Delta V_{in}$$

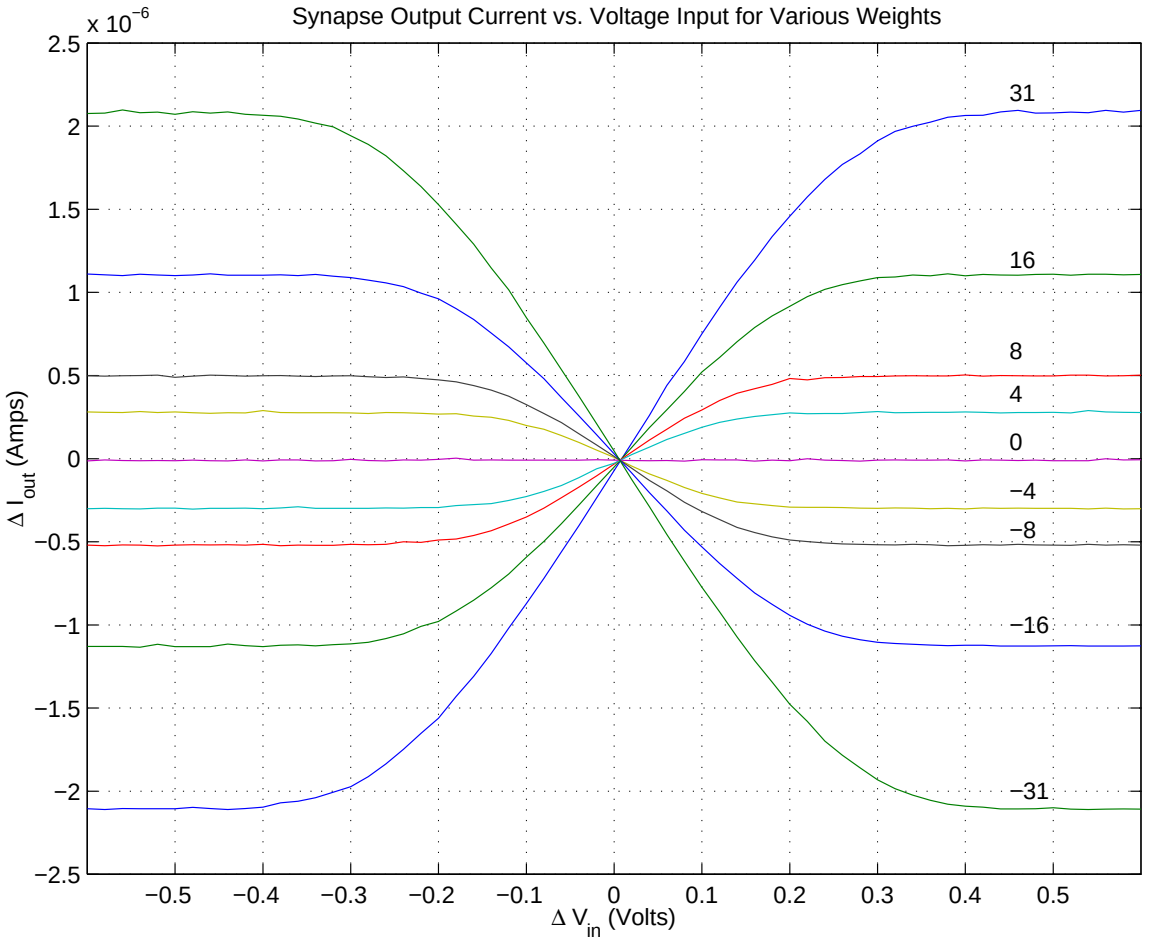


Figure 5.3: Synapse differential output current as a function of differential input voltage for various digital weight settings

It is clear that above threshold, the synapse is not doing a pure weighting of the input voltage. However, since the weights are learned on chip, they will be adjusted accordingly to the necessary value. Furthermore, it is possible that some synapses will operate below threshold while others above, depending on the choice of LSB current. Again, on-chip learning will be able to set the weights to account for these different modes of operation.

Figure 5.3 shows the differential output current of the synapse as a function of differential input voltage for various digital weight settings. The input current of the binary weighted current source was set to 100 nA. The output currents range from the subthreshold region for the smaller weights to the above threshold region for the

large weights. All of the curves show their sigmoidal characteristics. Furthermore, it is clear the width of the linear region increases as the current moves from subthreshold to above threshold. For the smaller weights, $W = 4$, the linear region spans only approximately 0.2V - 0.4V. For the largest weights, $W = 31$, the linear range has expanded to roughly 0.6V - 0.8V. As was discussed above, when the current range moves above threshold, the synapse does not perform a pure linear weighting. The largest synapse output current is not 3.1μA as would be expected from a linear weighting of 31 $\times$ 100nA, but a smaller number. Notice that the zero crossing of ΔI_{out} occurs slightly positive of $\Delta V_{in} = 0$. This is a circuit offset that is primarily due to slight W/L differences of the differential input pair of the synapse, and it is caused by minor fabrication variations.

5.2 Neuron

The synapse circuit outputs a differential current that will be summed in the neuron circuit shown in figure 5.4. The neuron circuit performs the summation from all of the input synapses. The neuron circuit then converts the currents back into a differential voltage feeding into the next layer of synapses. Since the outputs of the synapse will all have a common mode component, it is important for the neuron to have common mode cancelation[22]. Since one side of the differential current inputs may have a larger share of the common mode current, it is important to distribute this common mode to keep both differential currents within a reasonable operating range.

$$I_{in+} = I_{in+_d} + \frac{I_{in+_d} + I_{in-_d}}{2} = I_{in+_d} + I_{cm_d}$$

$$I_{in-} = I_{in-_d} + \frac{I_{in+_d} + I_{in-_d}}{2} = I_{in-_d} + I_{cm_d}$$

$$\Delta I = I_{in+} - I_{in-} = I_{in+_d} - I_{in-_d}$$

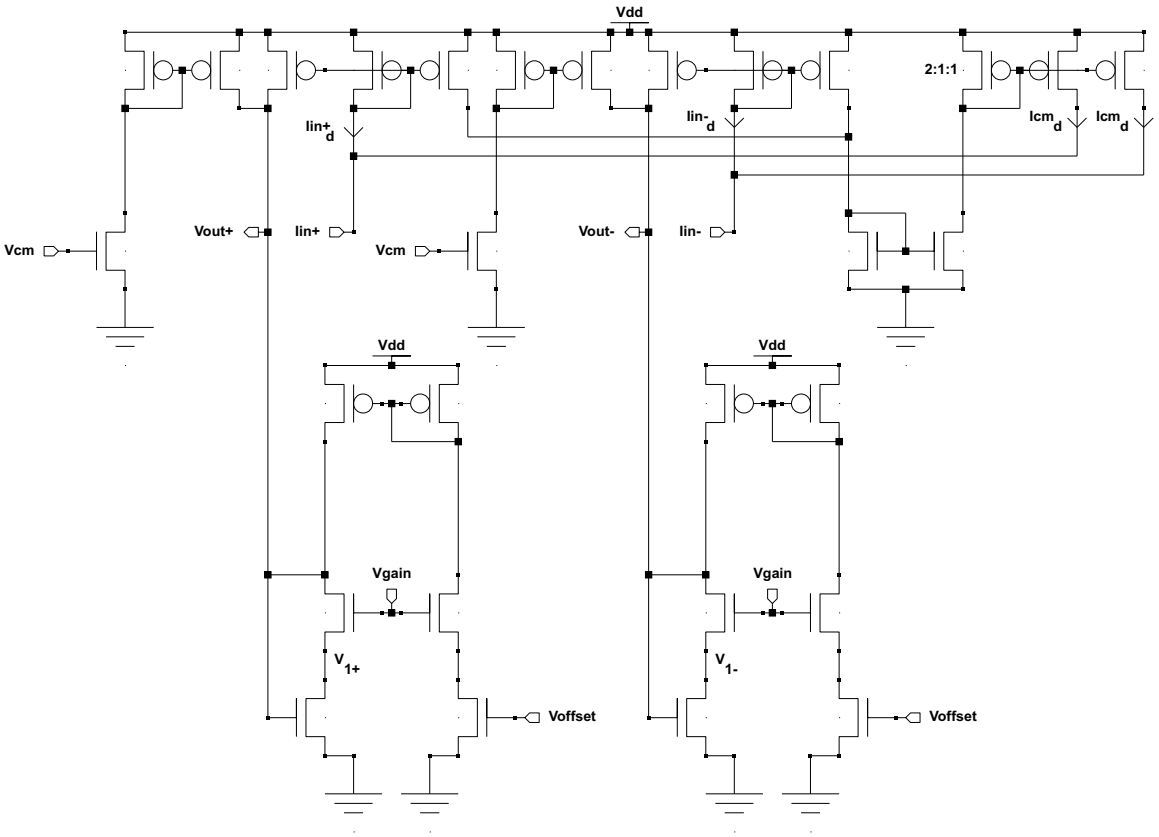


Figure 5.4: Neuron circuit

$$I_{cm} = \frac{I_{in+} + I_{in-}}{2} = \frac{I_{in+_d} + I_{in-_d} + 2I_{cm_d}}{2} = 2I_{cm_d}$$

$$\Rightarrow I_{in+_d} = I_{in+} - \frac{I_{cm}}{2} = \frac{\Delta I}{2} + \frac{I_{cm}}{2}$$

$$\Rightarrow I_{in-_d} = I_{in-} - \frac{I_{cm}}{2} = -\frac{\Delta I}{2} + \frac{I_{cm}}{2}$$

If the ΔI is of equal size or larger than I_{cm}, the transistor with I_{in-_d} may begin to cutoff and the above equations would not exactly hold; however, the current cutoff is graceful and should not normally affect performance. With the common mode signal properly equalized, the differential currents are then mirrored into the current to voltage transformation stage. This stage effectively takes the differential input currents and uses a transistor in the triode region to provide a differential output. This stage will usually be operating above threshold, because the V_{offset} and V_{cm} controls are used to ensure that the output voltages are approximately mid-rail. This is done by simply adding additional current to the diode connected transistor stack. Having the outputs mid-rail is important for proper biasing of the next stage of synapses. The above threshold transistor equation in the triode region is given by $I_d = 2K(V_{gs} - V_t - \frac{V_{ds}}{2})V_{ds} \approx 2K(V_{gs-}V_t)V_{ds}$ for small enough V_{ds}. If K_1 denotes the prefactor with W/L of the cascode transistor and K_2 denotes the same for the transistor with gate V_{out}, the voltage output of the neuron will then be given by

$$I_{in} = K_1(V_{gain} - V_t - V_1)^2 \approx K_2\,(2(V_{out} - V_t)V_1)$$

$$V_1 = V_{gain} - V_t - \sqrt{\frac{I_{in}}{K_1}}$$

$$I_{in} = 2K_2(V_{out} - V_t)\left(V_{gain} - V_t - \sqrt{\frac{I_{in}}{K_1}}\right)$$

$$V_{out} = \frac{I_{in}}{2K_2(V_{gain} - V_t) - 2\sqrt{\frac{K_2^2}{K_1} I_{in}}} + V_t$$

$$\text{if } \frac{W_1}{L_1} = \frac{W_2}{L_2} \text{ then } K_1 = K_2 = K,$$

$$\Rightarrow V_{out} = \frac{I_{in}}{2K(V_{gain} - V_t) - 2\sqrt{K I_{in}}} + V_t$$

$$\text{for small } I_{in,}\ R \approx \frac{1}{2K(V_{gain} - V_t)}$$

Thus, it is clear that V_{gain} can be used to adjust the effective gain of the stage.

Figure 5.5 shows how the neuron differential output voltage, ΔV_{out}, varies as a function of differential input current for several values of V_{gain}. The neuron shows fairly linear performance with a sharp bend on either side of the linear region. This sharp bend occurs when one of the two linearized, diode connected transistors with gate attached to V_{out} turns off.

Figure 5.6 displays only the positive output, V_{out+} of the neuron. The diode connected transistor, with gate attached to V_{out+} turns off where the output goes flat on the left side of the curves. This corresponds to the left bend point in figure 5.5. Furthermore, the baseline output voltage corresponds roughly to V_{offset}. However, as V_{offset} increases in value, its ability to increase the baseline voltage is reduced because of the cascode transistor on its drain. At some point, especially for small values of V_{gain}, the V_{cm} transistor becomes necessary to provide additional offset. Overall, the neuron shows very good linear current to voltage conversion with separate gain and

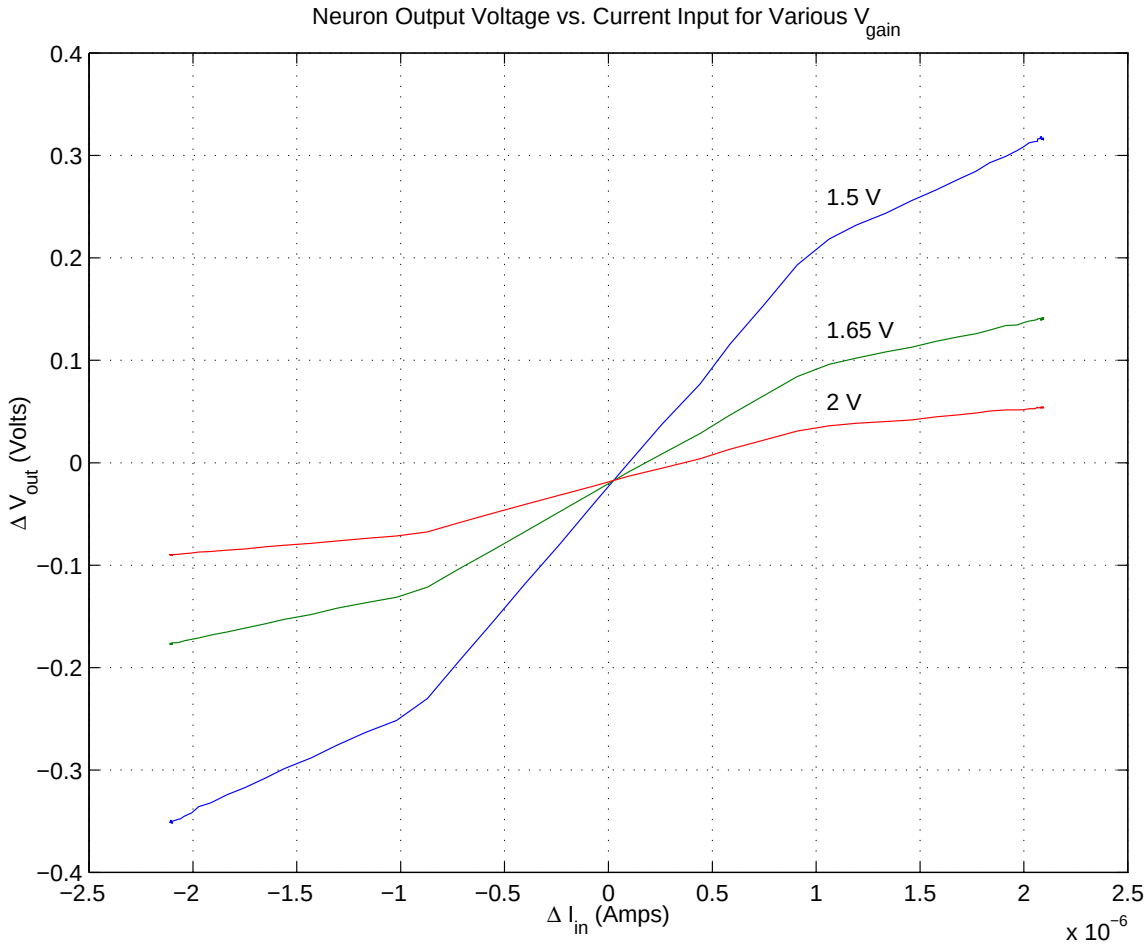


Figure 5.5: Neuron differential output voltage, ΔV_{out}, as a function of ΔI_{in}, for various values of V_{gain}

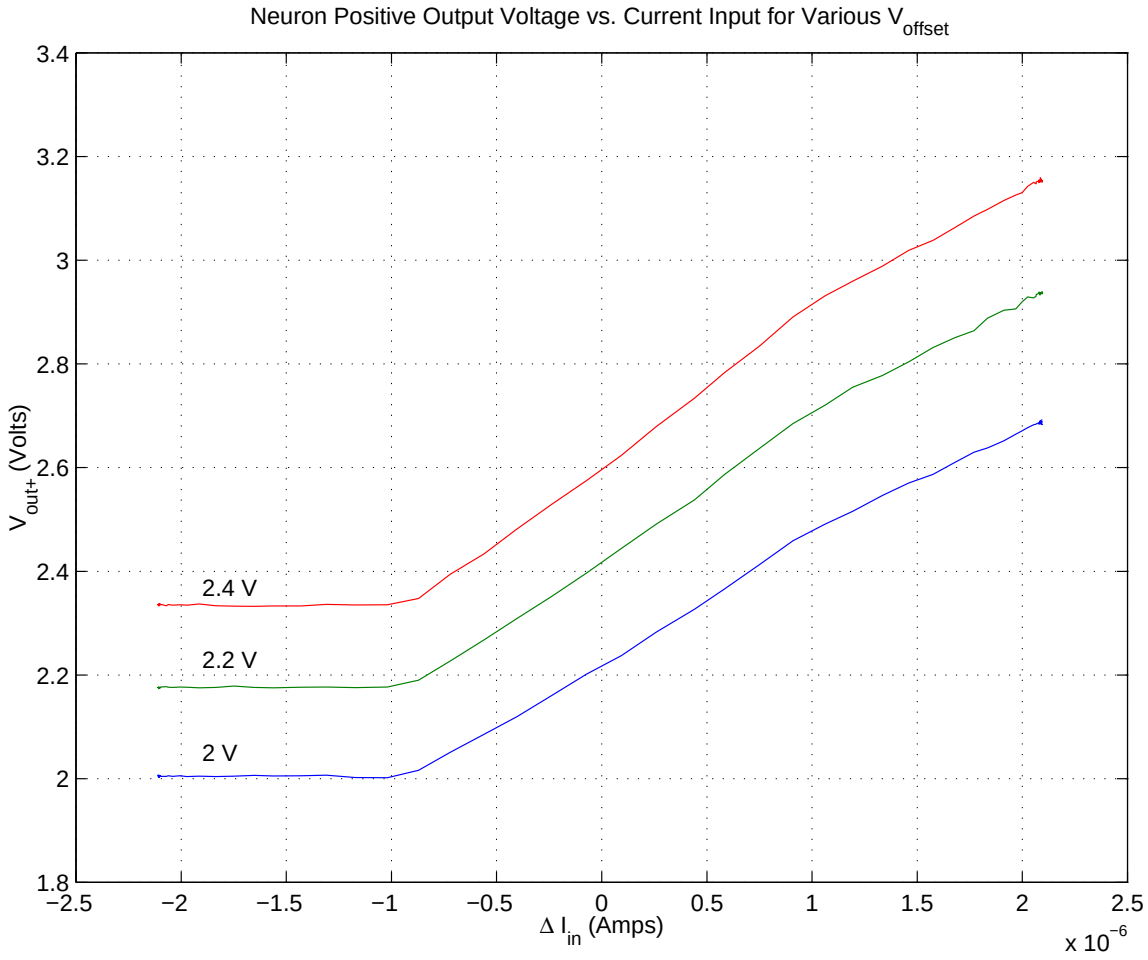


Figure 5.6: Neuron positive output, V_{out+}, as a function of ΔI_{in}, for various values of V_{offset}

offset controls.

5.3 Serial Weight Bus

A serial weight bus is used to apply weights to the synapses. The weight bus merely consists of a long shift register to cover all of the possible weight and threshold bits. Figure 5.7 shows the circuit schematic of the shift register used to implement the weight bus. The input to the serial weight bus comes from a host computer which implements the learning algorithm. The last output bit of the shift register, Q5, feeds into the data input, D, of the next shift register to form a long compound shift register.

The shift register storage cell is accomplished by using two staticizer or jamb latches[34] in series. The latch consists of two interlocked inverters. The weak inverter, labeled with a W in the figure, is made sufficiently weak such that a signal coming from the main inverter and passing through the pass gate is sufficiently strong to overpower the weak inverter and set the state. Once the pass gate is turned off, the weak inverter keeps the current bit state. The shift register is controlled by a dual phase nonoverlapping clock. During the read phase of operation, when ϕ_1 is high, the data from the previous register is read into the first latch. Then, during the write phase of operation, when ϕ_2 is high, the bit is written to the second latch for storage.

Figure 5.8 shows the nonoverlapping clock generator[35]. A single CLK input comes from an off chip clock. The clock generator ensures that the two clock phases do not overlap in order that no race conditions exist between reading and writing. The delay inverters are made weak by making the transistors long. They set the appropriate delay such that after one clock transitions from high to low, the next one does not transition from low to high until after the small delay. Since the clock lines go throughout the chip and drive long shift registers, it is necessary to make the buffer transistors with large W/L ratios to drive the large capacitances on the output.

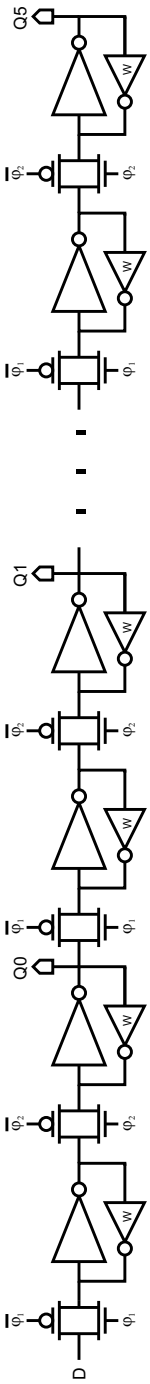


Figure 5.7: Serial weight bus shift register

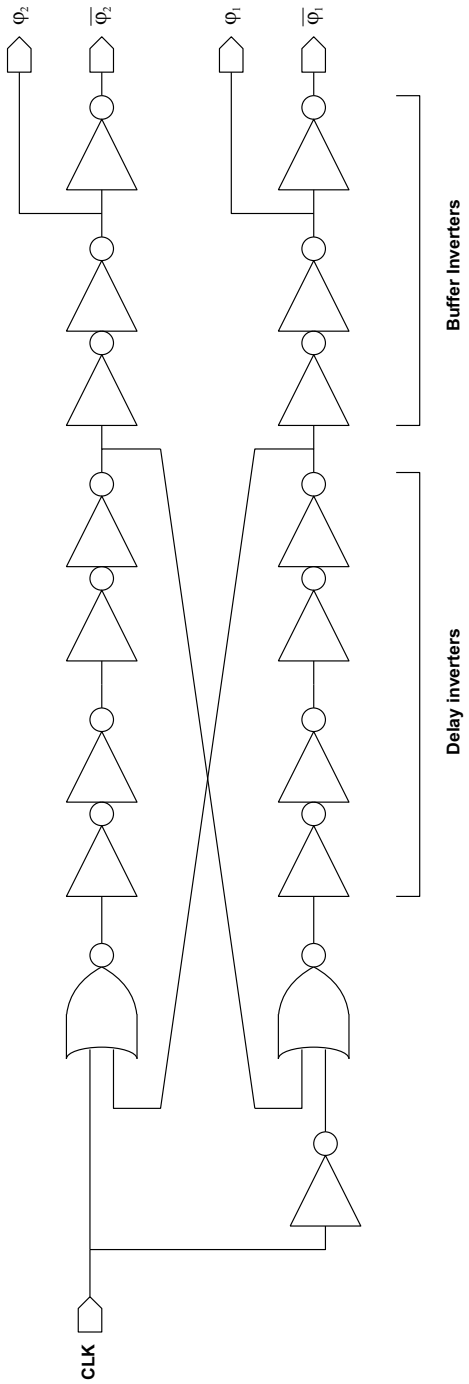


Figure 5.8: Nonoverlapping clock generator

5.4 Feedforward Network

Using the synapse and neuron circuit building blocks it is possible to construct a multilayer feed-forward neural network. Figure 5.9 shows a figure of a 2 input, 2 hidden unit, 1 output feedforward network for training of XOR.

Note that the nonlinear squashing function is actually performed in the next layer of synapse circuits rather than in the neuron as in a traditional neural network. However, this is equivalent as long as the inputs to the first layer are kept within the linear range of the synapses. For digital functions, the inputs need not be constrained as the synapses will pass roughly the same current regardless of whether the digital inputs are at the flat part of the synapse curve near the linear region or all the way at the end of the flat part of the curve. Furthermore, for non-differential digital signals, it is possible to simply tie the negative input to mid-rail and apply the standard digital signal to the positive synapse input. The biases, or thresholds, for each neuron are simply implemented as synapses tied to fixed bias voltages. The biases are learned in the same way as the weights.

Also, depending on the type of network outputs desired, additional circuitry may be needed for the final squashing function. For example, if a roughly linear output is desired, the differential output can be taken directly from the neuron outputs. In figure 5.9, a differential to single ended converter is shown on the output. The gain of this converter determines the size of the linear region for the final output. Normally, during training, a somewhat linear output with low gain is desired to have a reasonable slope to learn the function on. However, after training, it is possible to take the output after a dual inverter digital buffer to get a strong standard digital signal to send off-chip or to other sections of the chip.

Figure 5.10 shows a schematic of a differential to single ended converter and a digital buffer. The differential to single ended converter is based on the same transconductance amplifier design used for the synapse. The V_b knob is used to set the bias current of the amplifier. This bias current then controls the gain of the converter. The gain is also controlled by sizing of the transistors.

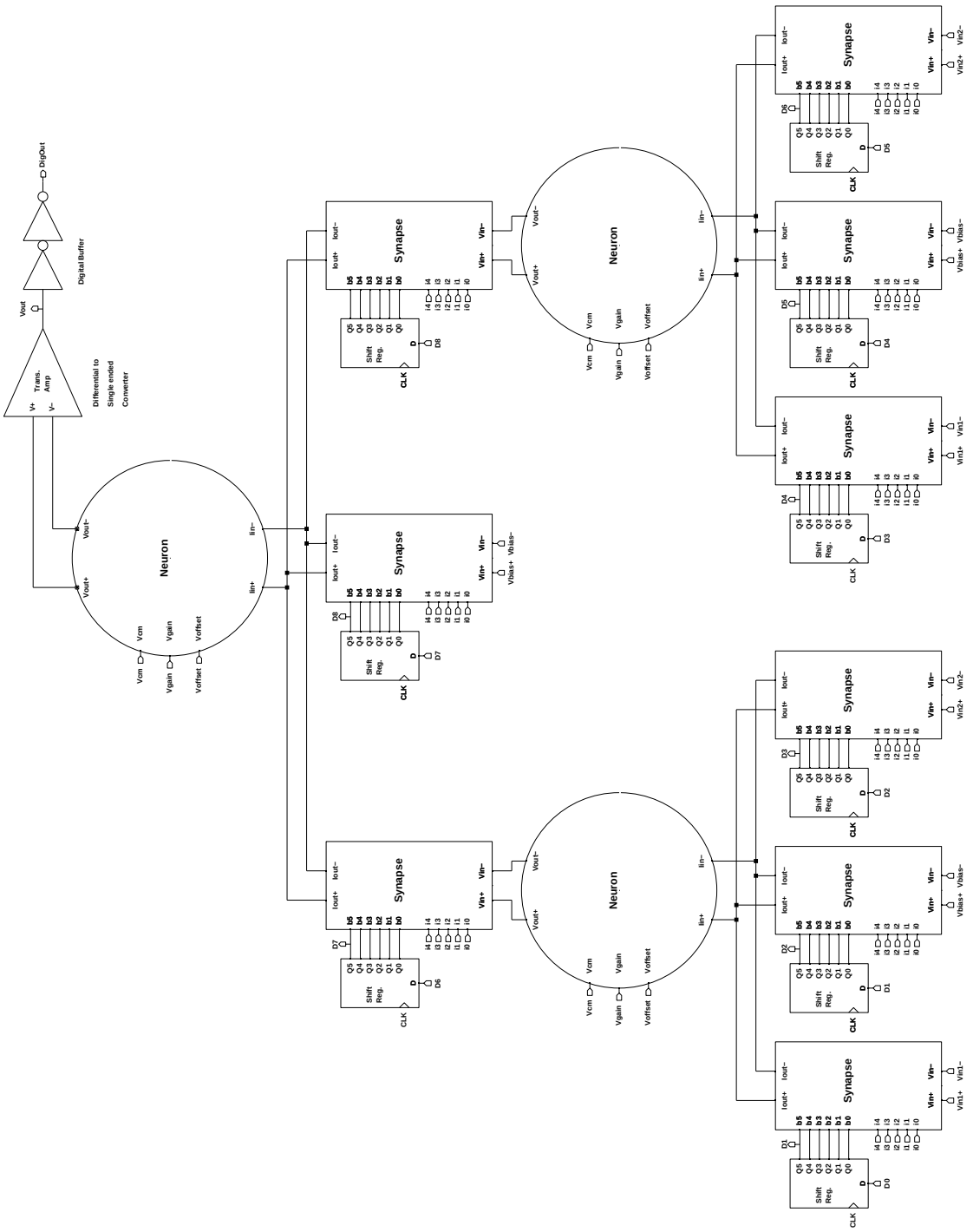


Figure 5.9: 2 input, 2 hidden unit, 1 output neural network

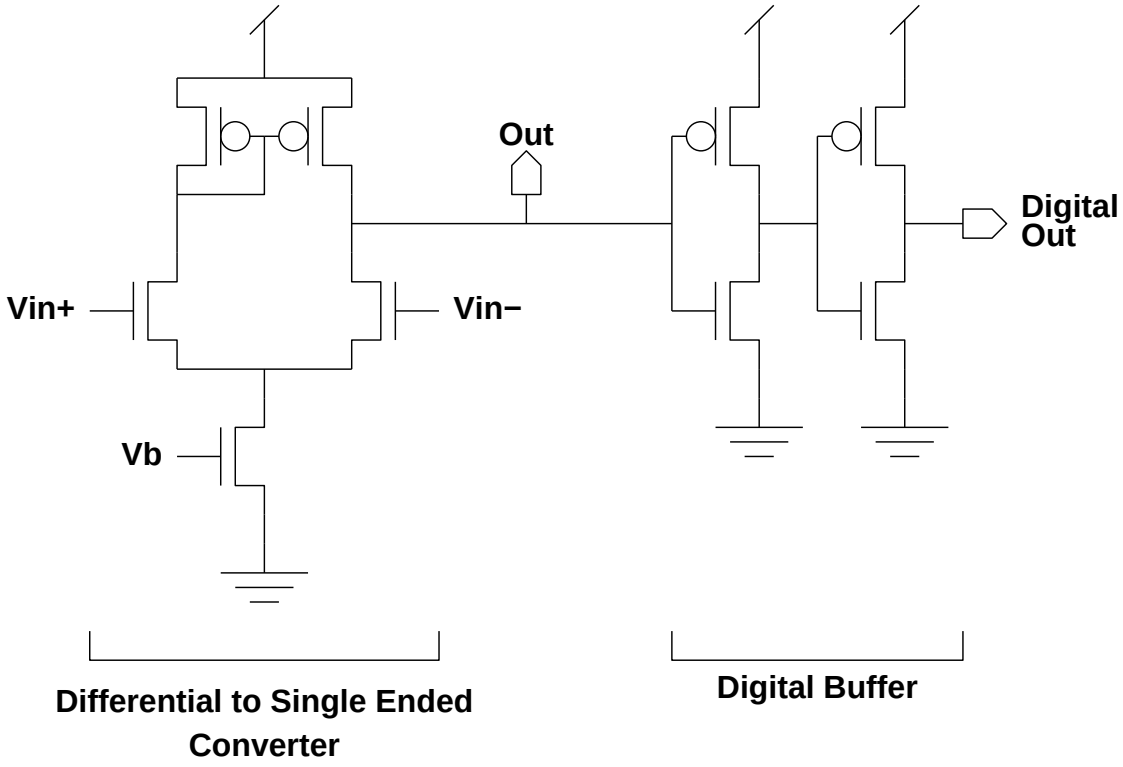


Figure 5.10: Differential to single ended converter and digital buffer

```
Initialize Weights;
Get Error;
while(Error > Error Goal);
   Perturb Weights;
   Get New Error;
   if (New Error < Error),
      Weights = New Weights;
      Error = New Error;
   else
      Restore Old Weights;
   end
end
```

Figure 5.11: Parallel Perturbative algorithm

5.5 Training Algorithm

The neural network is trained by using a parallel perturbative weight update rule[21]. The perturbative technique requires generating random weight increments to adjust the weights during each iteration. These random perturbations are then applied to all of the weights in parallel. In batch mode, all input training patterns are applied and the error is accumulated. This error is then checked to see if it was higher or lower than the unperturbed iteration. If the error is lower, the perturbations are kept, otherwise they are discarded. This process repeats until a sufficiently low error is achieved. An outline of the algorithm is given in figure 5.11. Since the weight updates are calculated offline, other suitable algorithms may also be used. For example, it is possible to apply an annealing schedule wherein large perturbations are initially applied and gradually reduced as the network settles.

5.6 Test Results

A chip implementing the above circuits was fabricated in a 1.2μm CMOS process. All synapse and neuron transistors were 3.6μm/3.6μm to keep the layout small. The unit size current source transistors were also 3.6μm/3.6μm. An LSB current of 100nA was

chosen for the current source. The above neural network circuits were trained with some simple digital functions such as 2 input AND and 2 input XOR. The results of some training runs are shown in figures 5.12-5.13. As can be seen from the figures, the network weights slowly converge to a correct solution. Since the training was done on digital functions, a differential to single ended converter was placed on the output of the final neuron. This was simply a 5 transistor transconductance amplifier. The error voltages were calculated as a total sum voltage error over all input patterns at the output of the transconductance amplifier. Since V_{dd} was 5V, the output would only easily move to within about 0.5V from V_{dd} because the transconductance amplifier had low gain. Thus, when the error gets to around 2V, it means that all of the outputs are within about 0.5V from their respective rail and functionally correct. A double inverter buffer can be placed at the final output to obtain good digital signals. At the beginning of each of the training runs, the error voltage starts around or over 10V indicating that at least 2 of the input patterns give an incorrect output.

Figure 5.12 shows the results from a 2 input, 1 output network learning an AND function. This network has only 2 synapses and 1 bias for a total of 3 weights. The weight values can go from -31 to +31 because of the 6 bit D/A converters used on the synapses.

Figure 5.13 shows the results of training a 2 input, 2 hidden unit, 1 output network with the XOR function. The weights are initialized as small random numbers. The weights slowly diverge and the error monotonically decreases until the function is learned. As with gradient techniques, occasional training runs resulted in the network getting stuck in a local minimum and the error would not go all the way down.

An ideal neural network with weights appropriately chosen to implement the XOR function is shown in figure 5.14. The inputs for the network and neuron outputs are chosen to be (-1,+1), as opposed to (0,1). This choice is made because the actual synapses which implement the squashing function give maximum outputs of +/- I_{out}. The neurons are assumed to be hard thresholds. The function computed by each of the neurons is given by

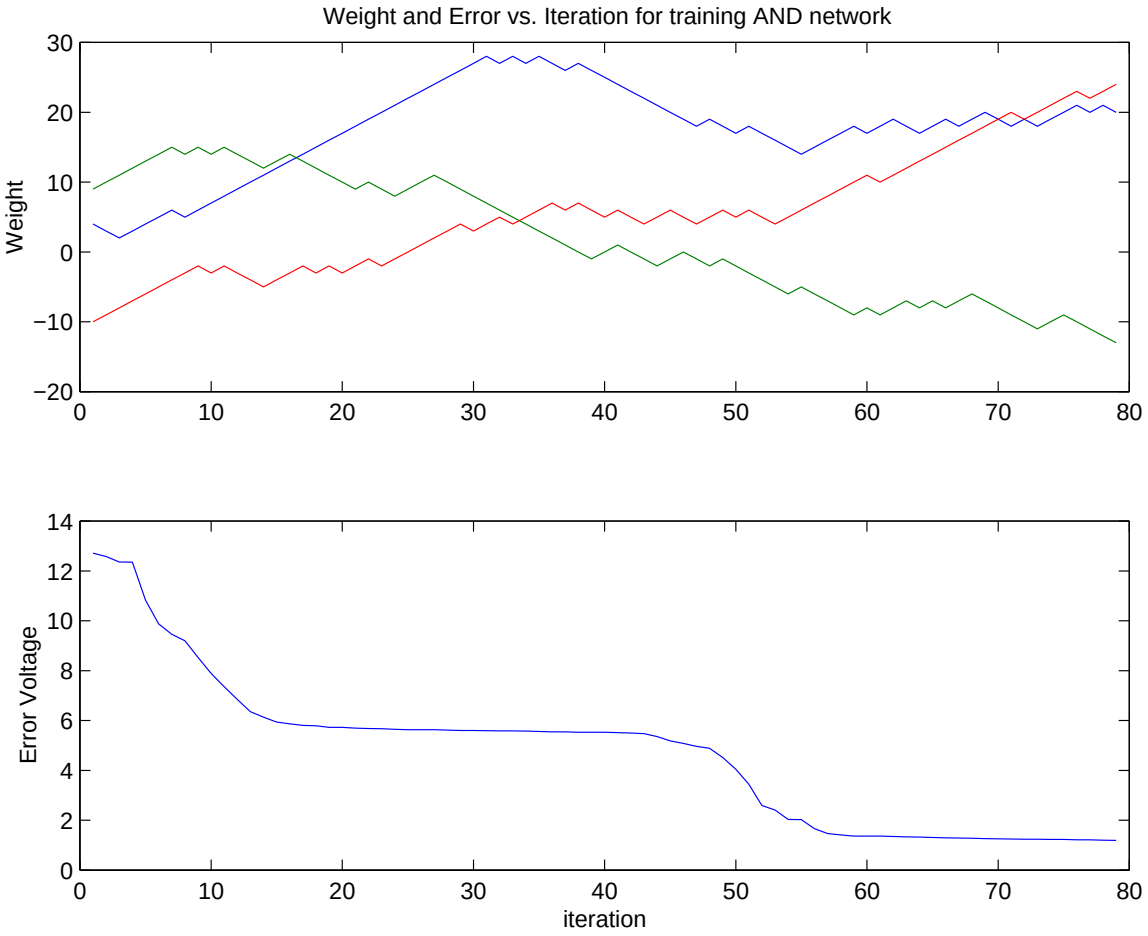


Figure 5.12: Training of a 2:1 network with AND function

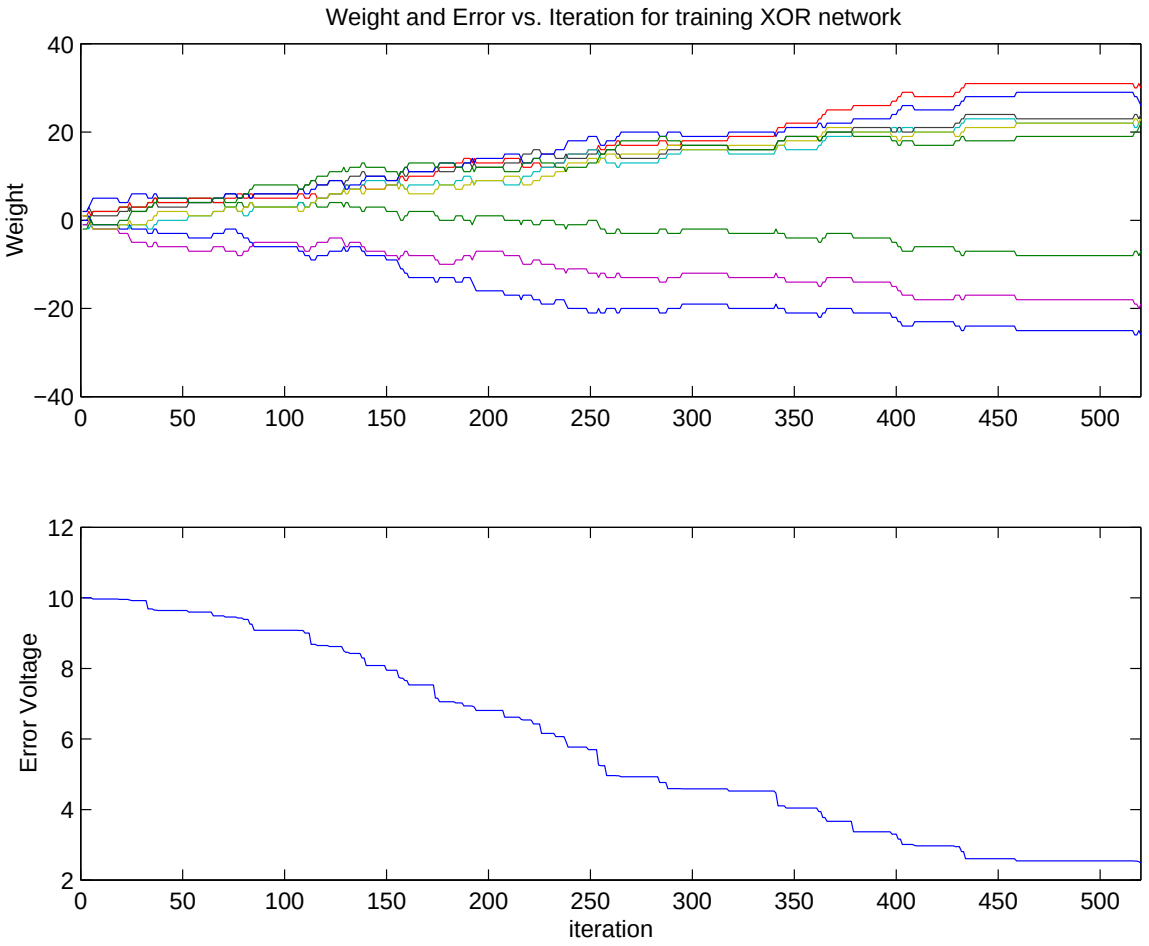


Figure 5.13: Training of a 2:2:1 network with XOR function starting with small random weights

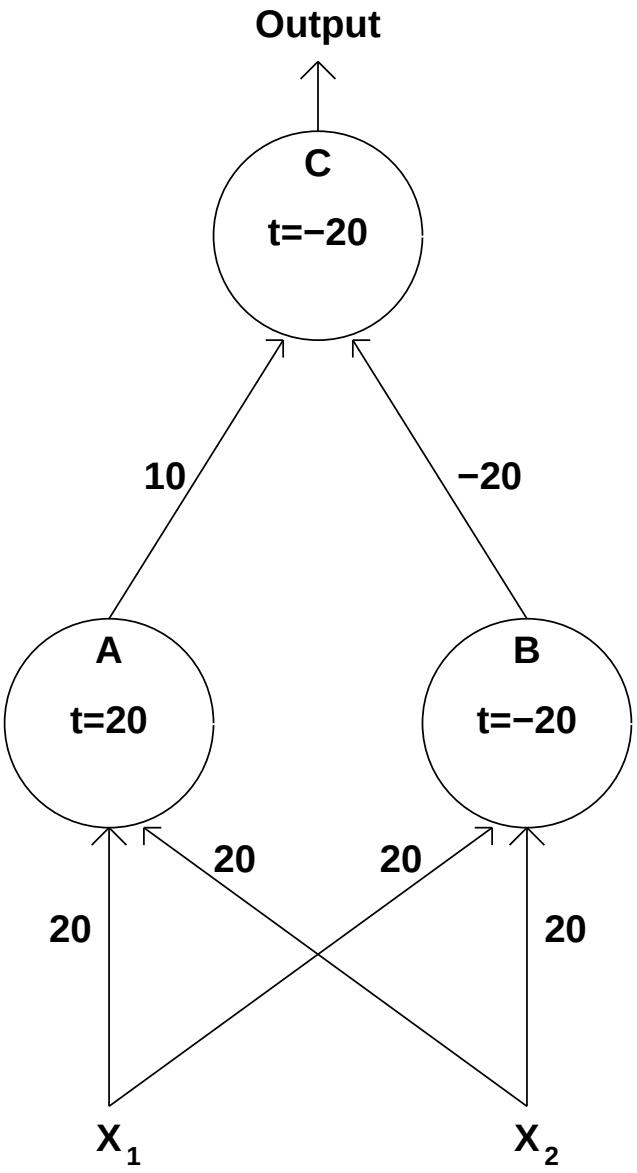


Figure 5.14: Network with "ideal" weights chosen for implementing XOR

X_1	X_2	A	B	C
-1	-1	$-60 \Rightarrow -1$	$-60 \Rightarrow -1$	$-10 \Rightarrow -1$
-1	+1	$+20 \Rightarrow +1$	$-20 \Rightarrow -1$	$+10 \Rightarrow +1$
+1	-1	$+20 \Rightarrow +1$	$-20 \Rightarrow -1$	$+10 \Rightarrow +1$
+1	+1	$+60 \Rightarrow +1$	$+20 \Rightarrow +1$	$-30 \Rightarrow -1$

Table 5.1: Computations for network with "ideal" weights chosen for implementing XOR

$$Out = \begin{cases} +1, & \text{if } ((\sum_i W_i X_i) + t) \geq 0 \\ -1, & \text{if } ((\sum_i W_i X_i) + t) < 0 \end{cases}$$

Table 5.1 shows the inputs, intermediate computations, and outputs of the idealized network. It is clear that the XOR function is properly implemented. The weights were chosen to be within the range of possible weight values, -31 to +31, of the actual network. This ideal network would perform perfectly well with smaller weights. For example, all of the input weights could be set to 1 as opposed to 20, and the A and B thresholds would then be set to 1 and -1 respectively, without any change in the network function. However, weights of large absolute value within the possible range were chosen (20 as opposed to 1), because small weights would be within the linear region of the synapses. Also, small weights such as +/- 1 might tend to get flipped due to circuit offsets. A SPICE simulation was done on the actual circuit using these ideal weights and the outputs were seen to be the correct XOR outputs.

Figure 5.15 shows the 2:2:1 network trained with XOR, but with the initial weights chosen as the mathematically correct weights for the ideal synapses and neurons. Although the ideal weights should, both in theory and based on simulations, start off with correct outputs, the offsets and mismatches of the circuit cause the outputs to be incorrect. However, since the weights start near where they should be, the error goes down rapidly to the correct solution. This is an example of how a more complicated network could be trained on computer first to obtain good initial weights and then the training could be completed with the chip in the loop. Also, for more complicated

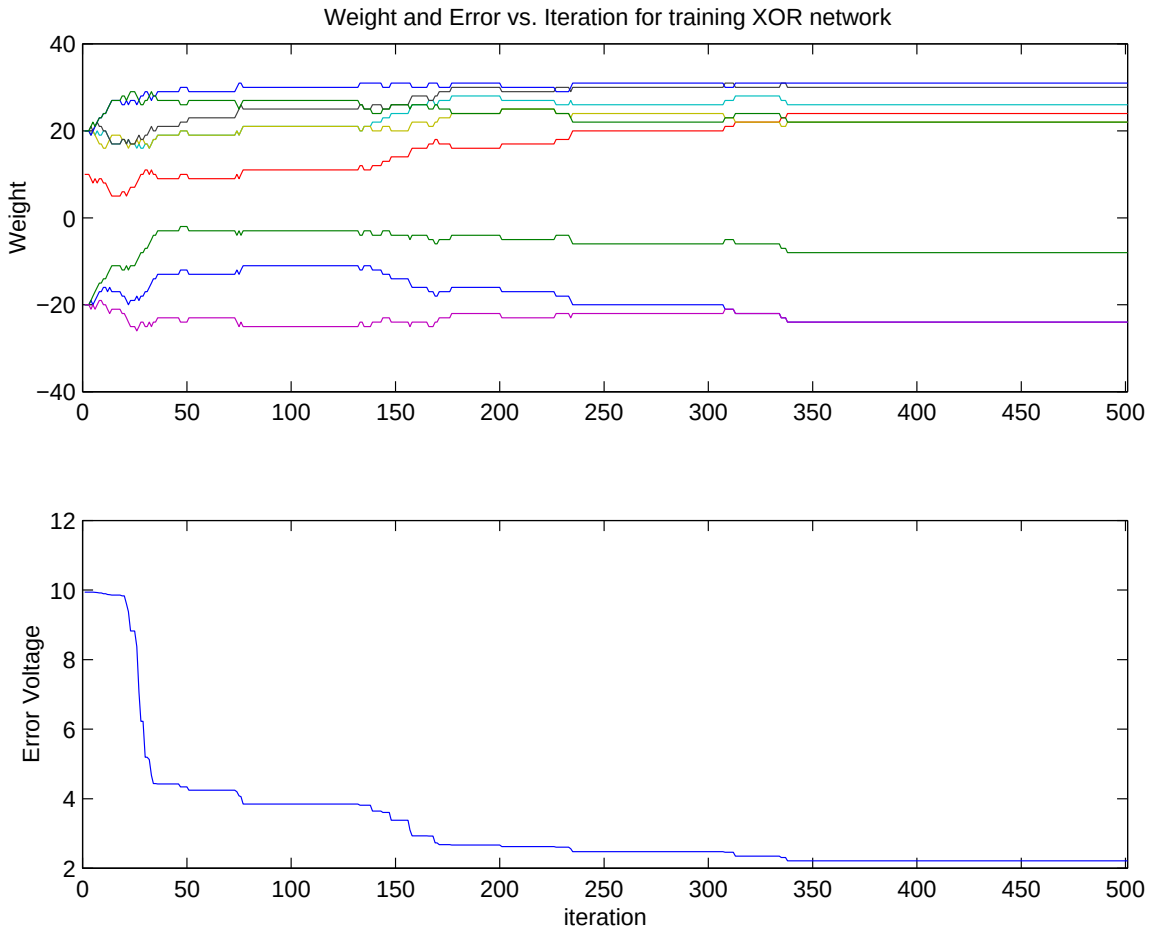


Figure 5.15: Training of a 2:2:1 network with XOR function starting with "ideal" weights

networks, using a more sophisticated model of the synapses and neurons that more closely approximates the actual circuit implementation would be advantageous for computer pretraining.

5.7 Conclusions

A VLSI implementation of a neural network has been demonstrated. Digital weights are used to provide stable weight storage. Analog multipliers are used because full digital multipliers would occupy considerable space for large networks. Although the functions learned were digital, the network is able to accept analog inputs and provide analog outputs for learning other functions. A parallel perturbation technique was used to train the network successfully on the 2-input AND and XOR functions.

Chapter 6 A Parallel Perturbative VLSI Neural Network

A fully parallel perturbative algorithm cannot truly be realized with a serial weight bus, because the act of changing the weights is performed by a serial operation. Thus, it is desirable to add circuitry to allow for parallel weight updates.

First, a method for applying random perturbation is necessary. The randomness is necessary because it defines the direction of search for finding the gradient. Since the gradient is not actually calculated, but observed, it is necessary to search for the downward gradient. It is possible to use a technique which does a nonrandom search. However, since no prior information about the error surface is known, in the worst case a nonrandom technique would spend much more time investigating upward gradients which the network would not follow.

A conceptually simple technique of generating random perturbations would be to amplify the thermal noise of a diode or resistor (figure 6.1). Unfortunately, the extremely large value of gain required for the amplifier makes the amplifier susceptible to crosstalk. Any noise generated from neighboring circuits would also get amplified. Since some of this noise may come from clocked digital sections, the noise would become very regular, and would likely lead to oscillations rather than the uncorrelated noise sources that are desired.

Such a scheme was attempted with separate thermal noise generators for each neuron[33]. The gain required for the amplifier was nearly 1 million and highly correlated oscillations of a few MHz were observed among all the noise generators. Therefore, another technique is required.

Instead, the random weight increments can be generated digitally with linear feedback shift registers that produce a long pseudorandom sequence. These random bits are used as inputs to a counter that stores and updates the weights. The counter

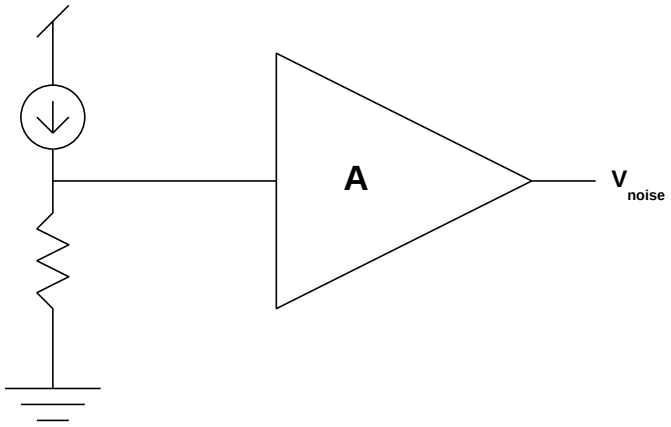


Figure 6.1: Amplified thermal noise of a resistor

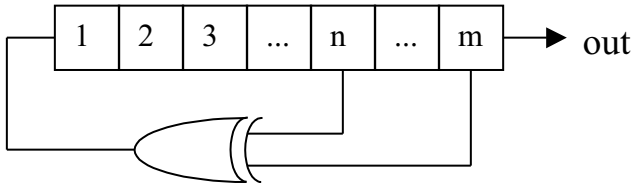


Figure 6.2: 2-tap, m-bit linear feedback shift register

outputs go directly to the D/A converter inputs of the synapses. If the weight updates led to a reduction in error, the update is kept. Otherwise, an inverter block is activated which inverts the counter inputs coming from the linear feedback shift registers. This has the effect of restoring the original weights. A block diagram of the full neural network circuit function is provided in figure 6.8.

6.1 Linear Feedback Shift Registers

The use of linear feedback shift registers for generating pseudorandom sequences has been known for some time[31]. A simple 2 tap linear feedback shift register is shown in figure 6.2. A shift register with m bits is used. In the figure shown, an XOR is performed from bit locations m and n and its output is fed back to the input of the shift register. In general, however, the XOR can be replaced with a multiple input parity function from any number of input taps. If one of the feedback taps is not

1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	1
1	1	0	0
0	1	1	0
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
1	0	0	0

Table 6.1: LFSR with m=4, n=3 yielding 1 maximal length sequence

1	0	0	0
0	1	0	0
1	0	1	0
0	1	0	1
0	0	1	0
0	0	0	1
1	0	0	0

1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
1	0	0	1
1	1	0	0

0	1	1	0
1	0	1	1
1	1	0	1
0	1	1	0

Table 6.2: LFSR with m=4, n=2, yielding 3 possible subsequences

taken from the final bit, m, but instead the largest feedback tap number is from some earlier bit, l, then this is equivalent to an l-bit linear feedback shift register and the output is merely delayed by the extra $m - l$ shift register bits.

A maximal length sequence from such a linear feedback shift register would be of length $2^M - 1$ and includes all 2^M possible patterns of 0's and 1's with the exception of the all 0's sequence. The all 0's sequence is considered a dead state, because $\mathrm{XOR}(0, 0) = 0$ and the sequence never changes. It is important to note that not all feedback tap selections will lead to maximal length sequences.

Table 6.1 shows an example of a maximal length sequence by choosing feedback taps of positions 4 and 3 from a 4 bit LFSR. It is clear that as long as the shift register

does not begin in the all 0's state, it will progress through all $2^M - 1 = 15$ sequences and then repeat. Table 6.2 shows an example where non-maximal length sequences are possible by utilizing feedback taps 4 and 2 from a 4 bit LFSR. In this situation it is possible to enter one of three possible subsequences. The initial state will determine which subsequence is entered. There is no overlap of the subsequences so it is not possible to transition from one to another. In this situation, the subsequence lengths are of size 6, 6, and 3. The sum of the sizes of the subsequences equal the size of a maximal length sequence.

It is desirable to use the maximal length sequences because they allow the longest possible pseudorandom sequence for the given complexity of the LFSR. Thus, it is important to choose the correct feedback taps. Tables have been tabulated which describe the lengths of possible sequences and subsequences based on LFSR lengths and feedback tap locations[31].

LFSRs of certain lengths cannot achieve maximal length sequences with only 2 feedback tap locations. For example, for $m = 8$, it is required to have an XOR(4,5,6) to get a length 255 sequence, and, for $m = 16$, it is required to have XOR(4,13,15) to obtain the length 65536 sequence[32]. Table 6.3 provides a list of LFSRs that can obtain maximal length sequences with only 2 feedback taps[32].

It is clear that it is possible to make very long pseudorandom sequences with moderately sized shift registers and a single XOR gate.

In applications where analog noise is desired, it is possible to low pass filter the output of the digital pseudorandom noise sequence. This is done by using a filter corner frequency which is much less than the clock frequency of the shift register.

6.2 Multiple Pseudorandom Noise Generators

From the previous discussion it is clear that linear feedback shift registers are a useful technique for generating pseudorandom noise. However, a parallel perturbative neural network requires as many uncorrelated noise sources as there are weights. Unfortunately, an LFSR only provides one such noise source. It is not possible to use

m	n	Length
3	2	7
4	3	15
5	3	31
6	5	63
7	6	127
9	5	511
10	7	1023
11	9	2047
15	14	32767
17	14	131071
18	11	262143
20	17	1048575
21	19	2097151
22	21	4194303
23	18	8388607
25	22	33554431
28	25	268435455
29	27	536870911
31	28	2147483647
33	20	8589934591
35	33	34359738367
36	25	68719476735
39	35	549755813887

Figure 6.3: Maximal length LFSRs with only 2 feedback taps

the different taps of a single LFSR as separate noise sources because these taps are merely the same noise pattern offset in time and thus highly correlated. One solution would be to use one LFSR with different feedback taps and/or initial states for every noise source required. For large networks with long training times, this approach becomes prohibitively expensive in terms of area and possibly power required to implement the scheme. Because of this, several approaches have been taken to resolve the problem, many of which utilize cellular automata.

In one approach[36], they build from a standard LFSR with the addition of an XOR network with inputs coming from the taps of an LFSR and with outputs providing the multiple noise sources. First, the analysis begins with an LFSR consisting of shift register cells numbering a_0 to a_{N-1}, with the cell marked a_0 receiving the feedback. Each of the N units in the LFSR except for the a_0 unit is given by

$$a_i^{t+1} = a_i^t$$

The first unit is described by

$$a_0^t = \bigoplus_{i=0}^{N-1} c_i a_i^t$$

where $\oplus$ represents the modulo 2 summation and the c_i are the boolean feedback coefficients where a coefficient of 1 represents a feedback tap at that cell.

The outputs are given by

$$g^t = \bigoplus_{i=0}^{N-1} d_i a_i^t$$

A detailed discussion is given as to the appropriate choice of the c_i and d_i coefficients to ensure good noise properties.

In another approach a standard boolean cellular automaton is used[37][38]. The boolean cellular automaton is comprised of N unit cells, a_i, wherein their state at time $t+1$ depends only on the states of all of the unit cells at time t.

$$a_i^{t+1} = f\left(a_0^t, a_1^t, ..., a_{N-1}^t\right)$$

An analysis was performed to choose a cellular automaton with good independence between output bits. The following transition rule was chosen:

$$a_i^{t+1} = (a_{i+1}^t + a_i^n) \oplus a_{i-1}^t \oplus a_{i-1}^t$$

Furthermore, every 4th bit is chosen as an output to ensure independent, unbiased, bits.

6.3 Multiple Pseudorandom Bit Stream Circuit

Another simplified approach utilizes two counterpropagating LFSRs with an XOR network to combine outputs from different taps to obtain uncorrelated noise[39]. For example, figure 6.4 shows two LFSRs with $m = 7$, $n = 6$ and $m = 6$, $n = 5$. Since the two LFSRs are counterpropagating, the length of the output sequences obtained from the XOR network are $(2^6 - 1)(2^7 - 1) = 8001$ bits[31]. In the figure, the bits are combined to provide 9 pseudorandom bit sequences. It is possible to obtain more channels and larger sequence lengths with the use of larger LFSRs. This was the scheme that was ultimately implemented in hardware.

6.4 Up/down Counter Weight Storage Elements

The weights in the network are represented directly as the bits of an up/down digital counter. The output bits of the counter feed directly into the digital input word weight bits of the synapse circuit. Updating the weights becomes a simple matter of incrementing or decrementing the counter to the desired value. The counter used is a standard synchronous up/down counter using full adders and registers (figure 6.5)[34].

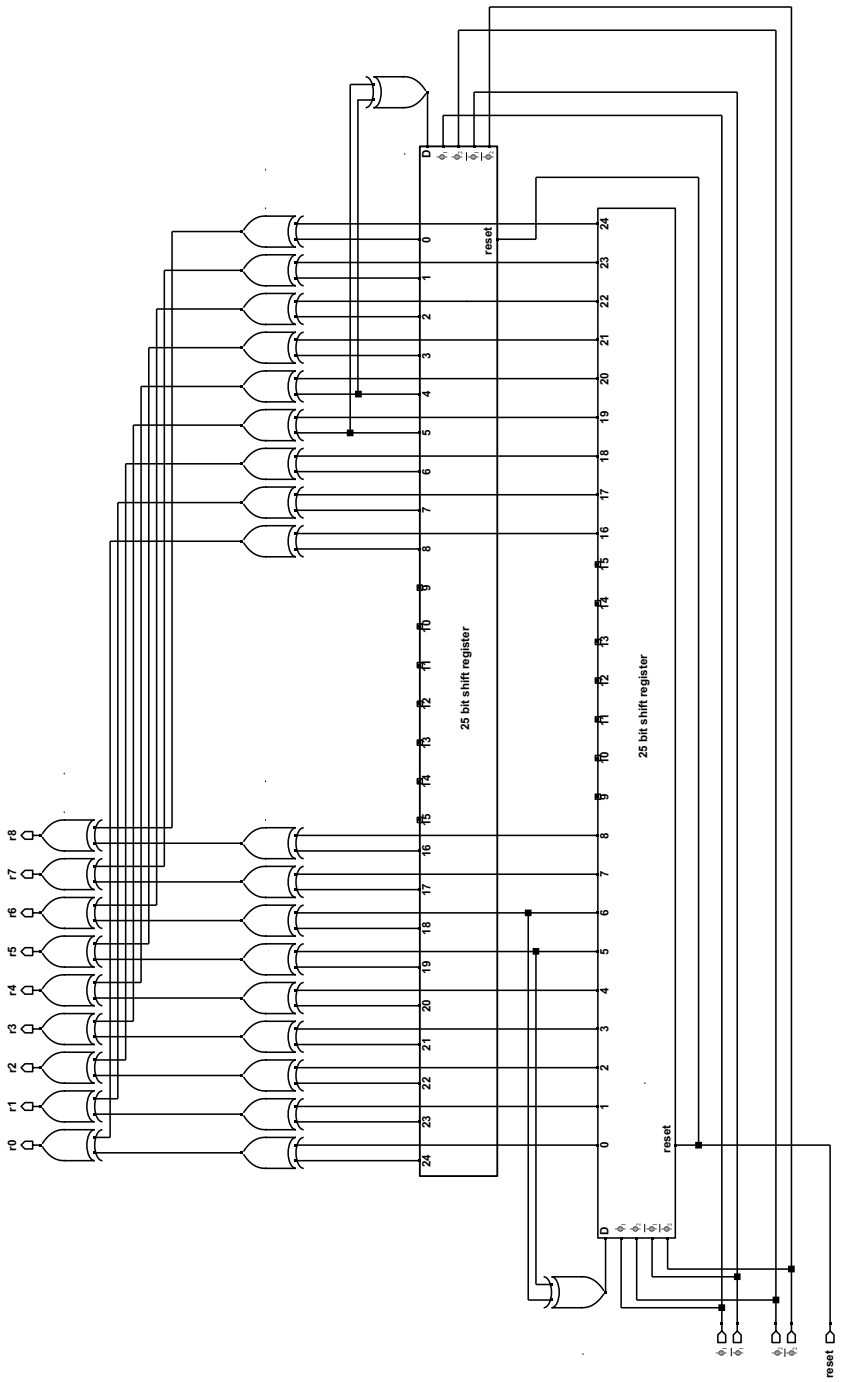


Figure 6.4: Dual counterpropagating LFSR multiple random bit generator

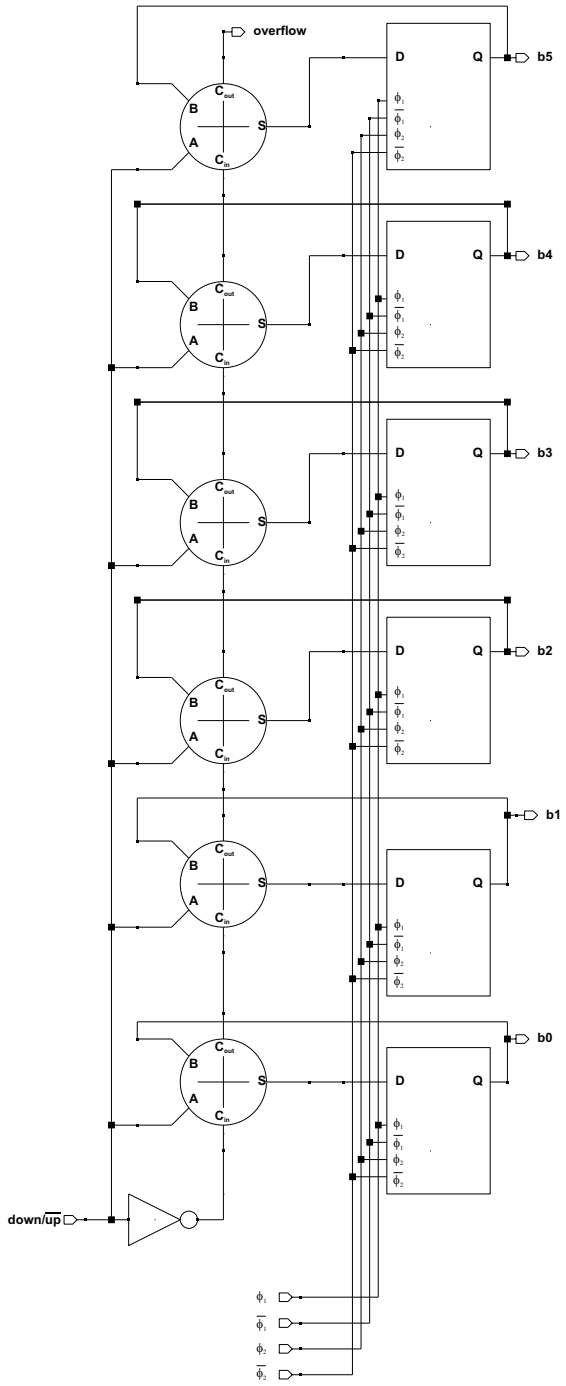


Figure 6.5: Synchronous up/down counter

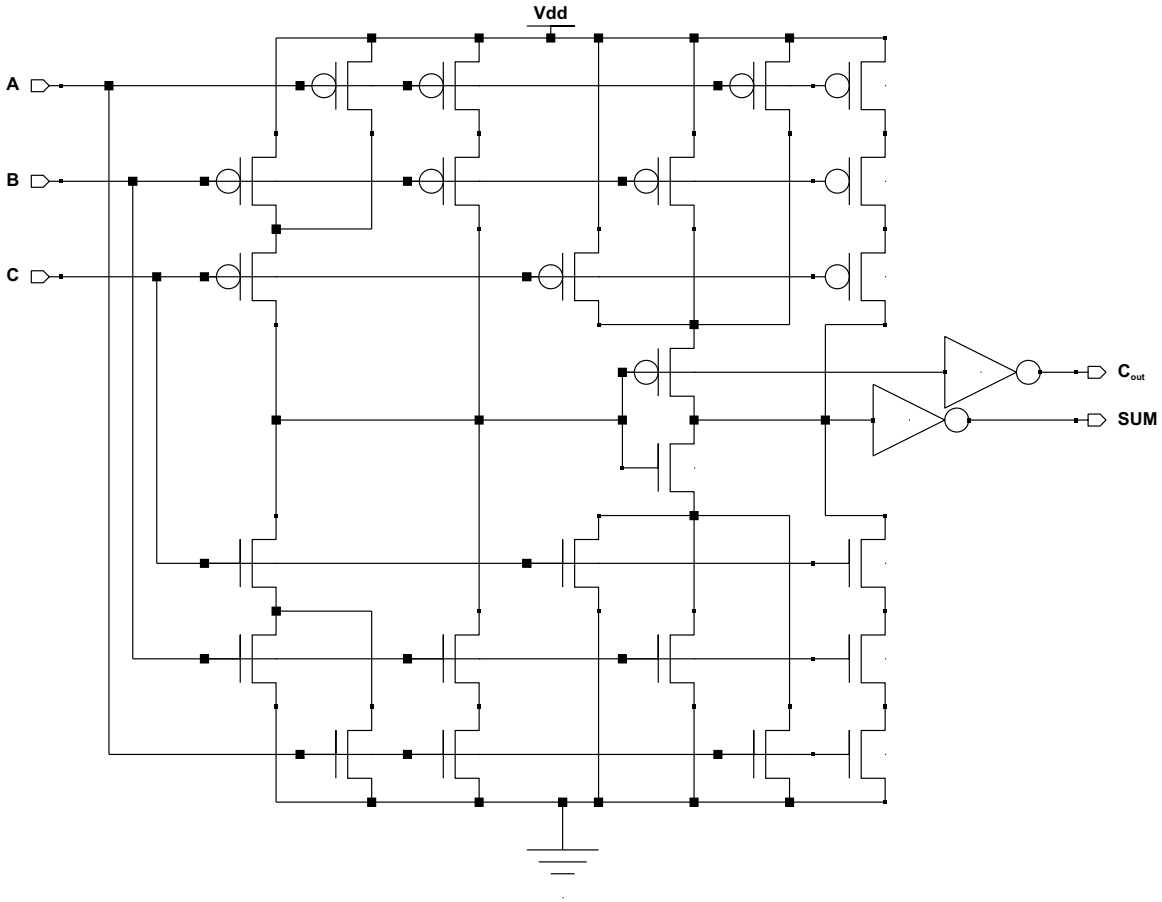


Figure 6.6: Full adder for up/down counter

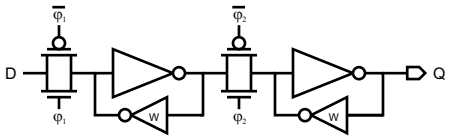


Figure 6.7: Counter static register

The full adder is shown in figure 6.6[34]. Since every weight requires 1 counter, and every counter requires 1 full adder for each bit, the adders take up a considerable amount of room. Thus, the adder is optimized for space instead of for speed. Most of the adder transistors are made close to minimum size.

The register cell (figure 6.7) used in the counter is simply a 1 bit version of the serial weight bus shown in figure 5.7. This register cell is both small and static, since the output must remain valid for standard feedforward operation after learning is complete.

6.5 Parallel Perturbative Feedforward Network

Figure 6.8 shows a block diagram of the parallel perturbative neural network circuit operation. The synapse, binary weighted encoder and neuron circuits are the same as those used for the serial weight bus neural network. However, instead of the synapses interfacing with a serial weight bus, counters with their respective registers are used to store and update the weights.

6.6 Inverter Block

The counter up/down inputs originate in the pseudorandom bit generator and pass through an inverter block (figure 6.9). The inverter block is essentially composed of pass gates and inverters. If the invert signal is low, the bit passes unchanged. If the invert bit is high, then the inversion of the pseudorandom bit gets passed. Control of the inverter block is what allows weight updates to either be kept or discarded.

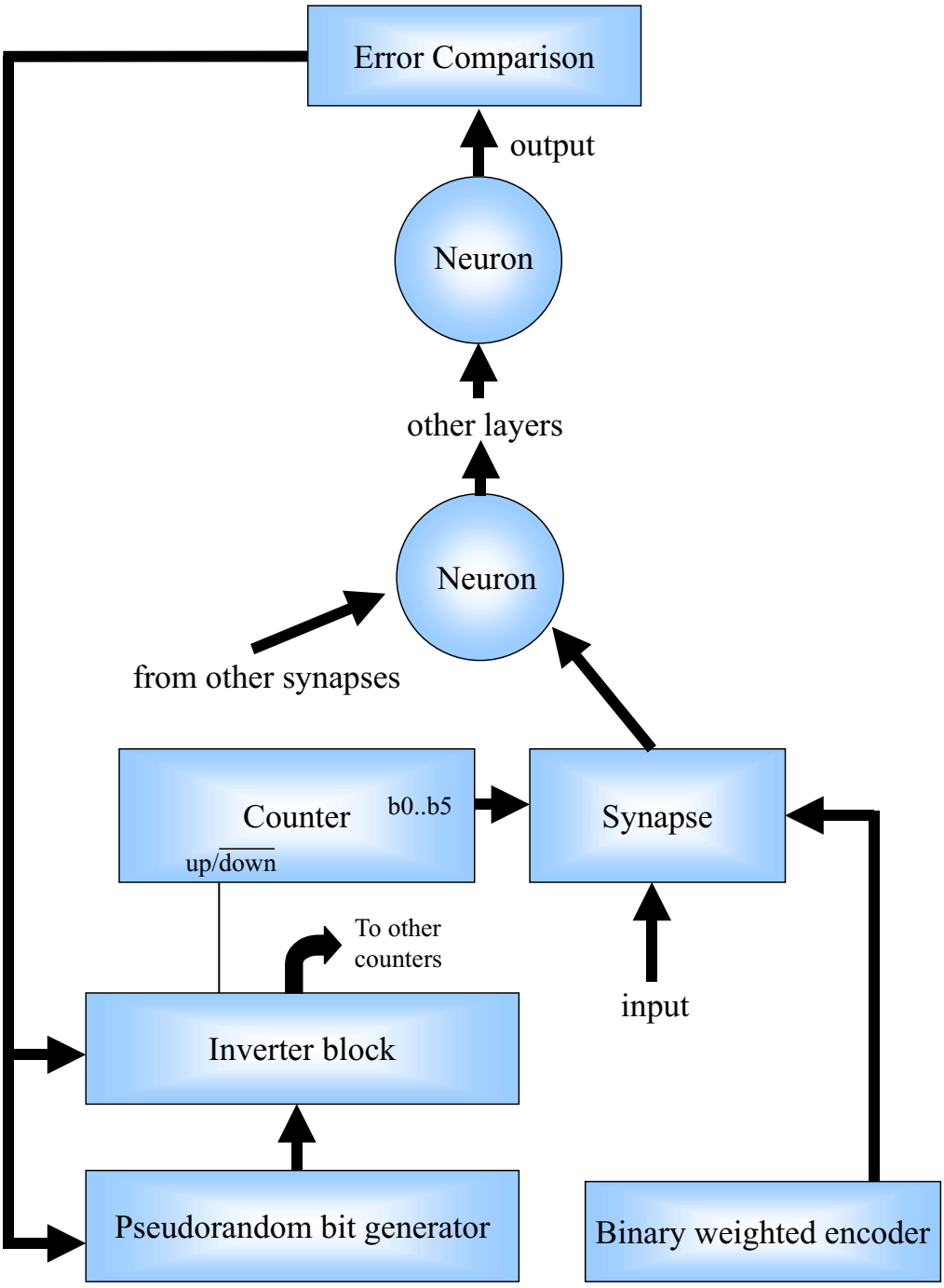


Figure 6.8: Block diagram of parallel perturbative neural network circuit

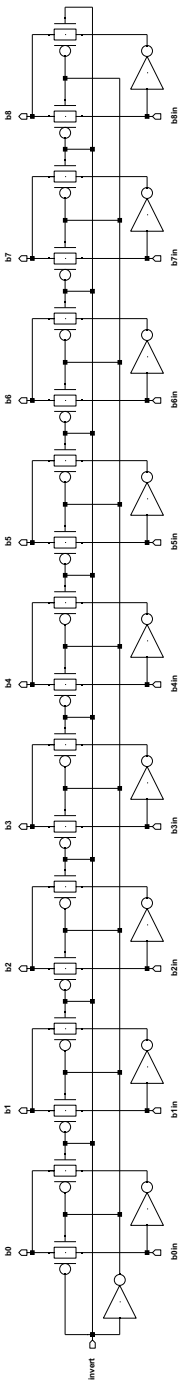


Figure 6.9: Inverter block

6.7 Training Algorithm

The algorithm implemented by the network is a parallel perturbative method[21][25]. The basic idea of the algorithm is to perform a modified gradient descent search of the error surface without calculating derivatives or using explicit knowledge of the functional form of the neurons. The way that this is done is by applying a set of small perturbations on the weights and measuring the effect on the network error. In a typical perturbative algorithm, after perturbations are applied to the weights, a weight update rule of the following form is used:

$$\Delta \overrightarrow{w} = -\eta \frac{E\left(\overrightarrow{w} + \overrightarrow{\text{pert}}\right) - E\left(\overrightarrow{w}\right)}{\text{pert}}$$

where $\overrightarrow{w}$ is a weight vector of all the weights in the network, $\overrightarrow{\text{pert}}$ is a perturbation vector applied to the weights which is normally of fixed size, pert, but with random signs, and η is a small parameter used to set the learning rate. Thus, the weight update rule can be seen as based on an approximate derivative of the error functions with respect to the weights.

$$\frac{\partial E}{\partial \overrightarrow{w}} \approx \frac{\Delta E}{\Delta \overrightarrow{w}} = \frac{E\left(\overrightarrow{w} + \overrightarrow{\text{pert}}\right) - E\left(\overrightarrow{w}\right)}{\left(\overrightarrow{w} + \text{pert}\right) - \overrightarrow{w}} = \frac{E\left(\overrightarrow{w} + \overrightarrow{\text{pert}}\right) - E\left(\overrightarrow{w}\right)}{\text{pert}}$$

However, this type of weight update rule requires weight changes which are proportional to the difference in error terms. A simpler, but functionally equivalent, weight update rule can be achieved as follows. First, the network error, $E\left(\overrightarrow{w}\right)$, is measured with the current weight vector, $\overrightarrow{w}$. Next, a perturbation vector, $\overrightarrow{\text{pert}}$, of fixed magnitude, but random sign is applied to the weight vector yielding a new weight vector, $\overrightarrow{w} + \overrightarrow{\text{pert}}$. Afterwards, the effect on the error, $\Delta E = E\left(\overrightarrow{w} + \overrightarrow{\text{pert}}\right) - E\left(\overrightarrow{w}\right)$, is measured. If the error decreases then the perturbations are kept and the next iteration is performed. If the error increases, the original weight vector is restored. Thus, the weight update rule is of the following form:

$$\overrightarrow{w}_{t+1} = \begin{cases} \overrightarrow{w}_t + \overrightarrow{\text{pert}}_t, & \text{if } E\left(\overrightarrow{w}_t + \overrightarrow{\text{pert}}_t\right) < E\left(\overrightarrow{w}_t\right) \\ \overrightarrow{w}_t, & \text{if } E\left(\overrightarrow{w}_t + \overrightarrow{\text{pert}}_t\right) > E\left(\overrightarrow{w}_t\right) \end{cases}$$

The use of this rule may require more iterations since it does not perform an actual weight change every iteration and since the weight updates are not scaled with the resulting changes in error. Nevertheless, it significantly simplifies the weight update circuitry. For either update rule, some means must be available to apply the weight perturbations; however, this rule does not require additional circuitry to change the weight values proportionately with the error difference, and, instead, relies on the same circuitry for the weight update as for applying the random perturbations. Some extra circuitry is required to remove the perturbations when the error doesn't decrease, but this merely involves inverting the signs of all of the perturbations and reapplying in order to cancel out the initial perturbation.

A pseudocode version of the algorithm is presented in figure 6.10.

6.8 Error Comparison

The error comparison section is responsible for calculating the error of the current iteration and interfacing with a control section to implement the algorithm. Both sections could be performed off-chip by using a computer for chip-in-loop training, as was chosen for the current implementation. This allows flexibility in performing the global functions necessary for implementing the training algorithm, while the local functions are performed on-chip. However, the control section could be implemented on chip as a standard digital section such as a finite state machine. Also, there are several alternatives for implementing the error comparison on chip. First, the error comparison could simply consist of analog to digital converters which would then pass the digital information to the control block. Another approach would be to have the error comparison section compare the analog errors directly and then output digital control signals.

```
Initialize Weights;
Get Error;
while(Error > Error Goal);
   Perturb Weights;
   Get New Error;
   if (New Error < Error),
      Error = New Error;
   else
      Unperturb Weights;
   end
end

function Perturb Weights;
   Set Invert Bit Low;
   Clock Pseudorandom bit generator;
   Clock counters;

function Unperturb Weights;
   Set Invert Bit High;
   Clock counters;
```

Figure 6.10: Pseudocode for parallel perturbative learning network

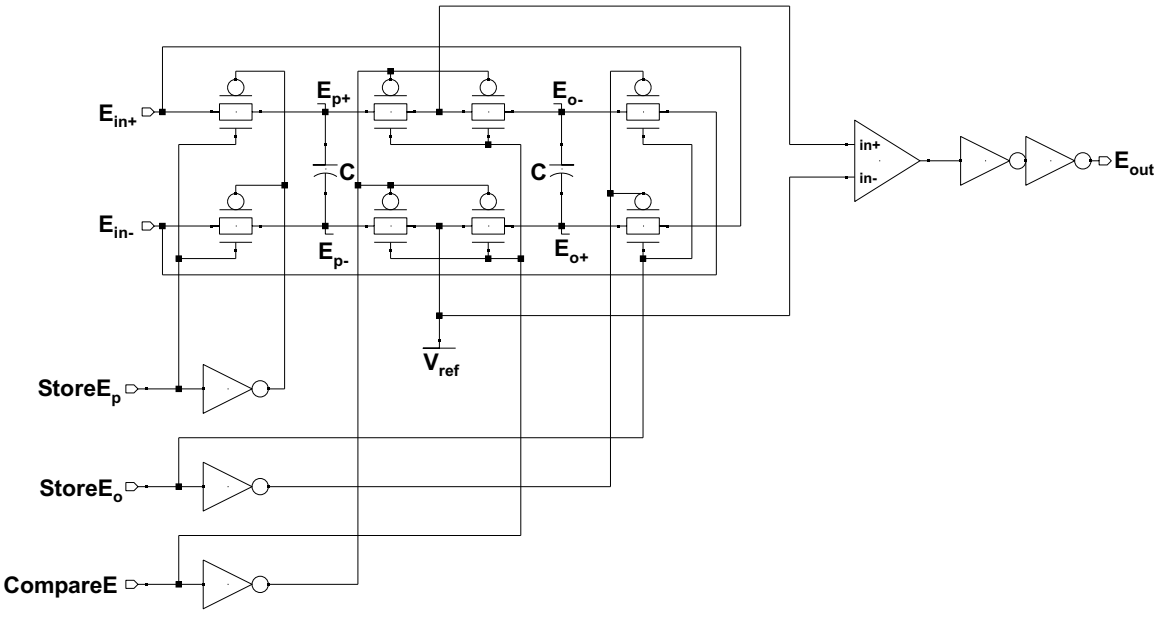
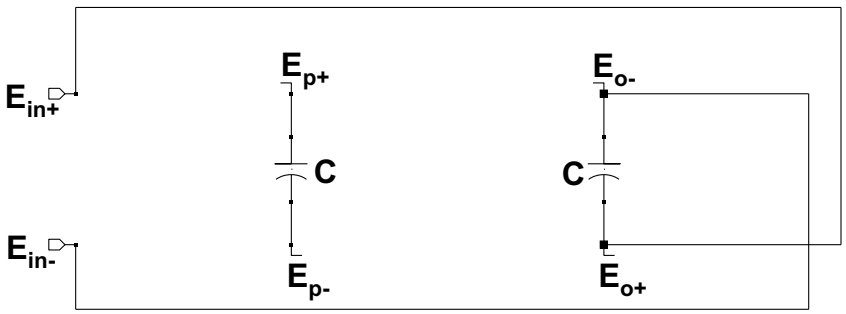
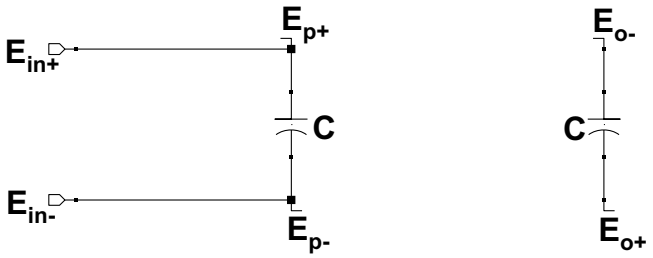


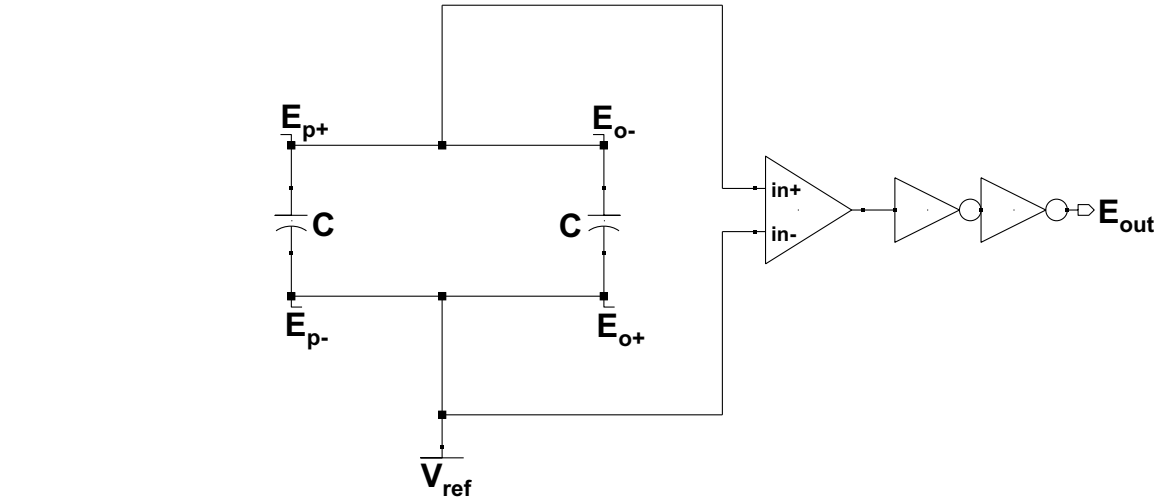
Figure 6.11: Error comparison section



a) StoreE_o high



b) StoreE_p high



c) CompareE high

Figure 6.12: Error comparison operation

A sample error comparison section is shown in figure 6.11. The section uses digital control signals to pass error voltages onto capacitor terminals and then combines the capacitors in such a way to subtract the error voltages. This error difference is then compared with a reference voltage to determine if the error decreased or increased. The output consists of a digital signal which shows the appropriate error change.

The error input terminals, E_{in+} and E_{in-}, are connected, in sequence, to two storage capacitors which are then compared. $StoreE_o$ is asserted when the unperturbed, initial error is stored. $StoreE_p$ is asserted in order to store the perturbed error after the weight perturbations have been applied. $CompareE$ is asserted to compute the error difference, and E_{out} shows the sign of the difference computation.

Figure 6.12 shows the operation of the error comparison section in greater detail. In figure 6.12a, $StoreE_o$ is high and the other signals are low. The capacitor where E_p is stored is left floating, while the inputs E_{in+} and E_{in-} are connected to E_{o+} and E_{o-} respectively. This stores the unperturbed error voltages on the capacitor. Next, in figure 6.12b, $StoreE_p$ is held high while the other signals are low. In the same manner as before, the inputs E_{in+} and E_{in-} are connected to E_{p+} and E_{p-} respectively. This stores the perturbed error voltages on the capacitor. Note that the polarities of the two storage capacitors are drawn reversed. This will facilitate the subtraction operation in the final step. In figure 6.12c, $CompareE$ is asserted and the other input signals are low. The two storage capacitors are connected together and their charges combine. The total charge, Q_T, across the parallel capacitors is given as follows:

$$Q_T = Q_+ - Q_- = C\left(E_{p+} + E_{o-}\right) - C\left(E_{p-} + E_{o+}\right) = C\left(\left(E_{p+} - E_{p-}\right) - \left(E_{o+} - E_{o-}\right)\right)$$

If both capacitors are of size C, the parallel capacitance will be of size $2C$. The voltage across the parallel capacitors can be obtained from $Q_T = 2CV_T$.

Thus,

$$V_T = \frac{Q_T}{2C} = \frac{1}{2}\left(\left(E_{p+} - E_{p-}\right) - \left(E_{o+} - E_{o-}\right)\right) = \frac{1}{2}\left(E_p - E_o\right)$$

Since these voltages are floating and must be referenced to some other voltage, the combined capacitor is placed onto a voltage, V_{ref}, which is usually chosen to be approximately half the supply voltage to allow for the greatest dynamic range.

Also, during this step, this error difference feeds into a high gain amplifier to perform the comparison. The high gain amplifier is shown as a differential transconductance amplifier which feeds into a double inverter buffer to output a strong digital signal. The final output, E_{out}, is given as follows:

$$E_{out} = \begin{cases} 1, & \text{if } E_p > E_o \\ 0, & \text{if } E_p < E_o \end{cases}$$

where the 1 represents a digital high signal, and the 0 represents a digital low signal.

This output can then be sent to a control section which will use it to determine the next step in the algorithm.

The error inputs which are computed by a difference of the actual output and a target output can be computed with a similar analog switched capacitor technique. However, it might also be desirable to insert an absolute value computation stage. Simple circuits for obtaining voltage absolute values can be found elsewhere[30].

Furthermore, this sample error computation stage can easily be expanded to accommodate a multiple output neural network.

6.9 Test Results

In this section several sample training run results from the chip are shown. A chip implementing the parallel perturbative neural network was fabricated in a 1.2μm CMOS process. All synapse and neuron transistors were 3.6μm/3.6μm to keep the layout small. The unit size current source transistors were also 3.6μm/3.6μm. An LSB current of 100nA was chosen for the current source. The above neural network circuits were trained with some simple digital functions such as 2 input AND and 2 input XOR. The results of some training runs are shown in figures 6.13-6.15. As can be

seen from the figures, the network weights slowly converge to a correct solution. The error voltages were taken directly from the voltage output of the output neuron. The error voltages were calculated as a total sum voltage error over all input patterns. The actual output voltage error is arbitrary and depends on the circuit parameters. What is important is that the error goes down. The differential to single-ended converter and a double inverter buffer can be placed at the final output to obtain good digital signals. Also shown is a digital error signal that shows the number of input patterns where the network gives the incorrect answer for the output. The network analog output voltage for each pattern is also displayed.

The actual weight values were not made accessible and, thus, are not shown; however, another implementation might also add a serial weight bus to read the weights and to set initial weight values. This would also be useful when pretraining with a computer to initialize the weights in a good location.

Figure 5.12 shows the results from a 2 input, 1 output network learning an AND function. This network has only 2 synapses and 1 bias for a total of 3 weights. The network starts with 2 incorrect outputs which is to be expected with initial random weights. Since the AND function is a very simple function to learn, after relatively few iterations, all of the outputs are digitally correct. However, the network continues to train and moves the weight vector in order to better match the training outputs.

Figures 6.14 and 6.15 show the results of training a 2 input, 2 hidden unit, 1 output network with the XOR function. In both figures, although the error voltage is monotonically decreasing, the digital error occasionally increases. This is because the network weights occasionally transition through a region of reduced analog output error that is used for training, but which actually increases the digital error. This seems to be occasionally necessary for the network to ultimately reach a reduced digital error. Both figures also show that the function is essentially learned after only several hundred iterations. In figure 6.15, the network is allowed to continue learning. After being stuck in a local minimum for quite some time, the network finally finds another location where it is able to go slightly closer towards the minimum.

Some of the analog output values occasionally show a large jump from one iter-

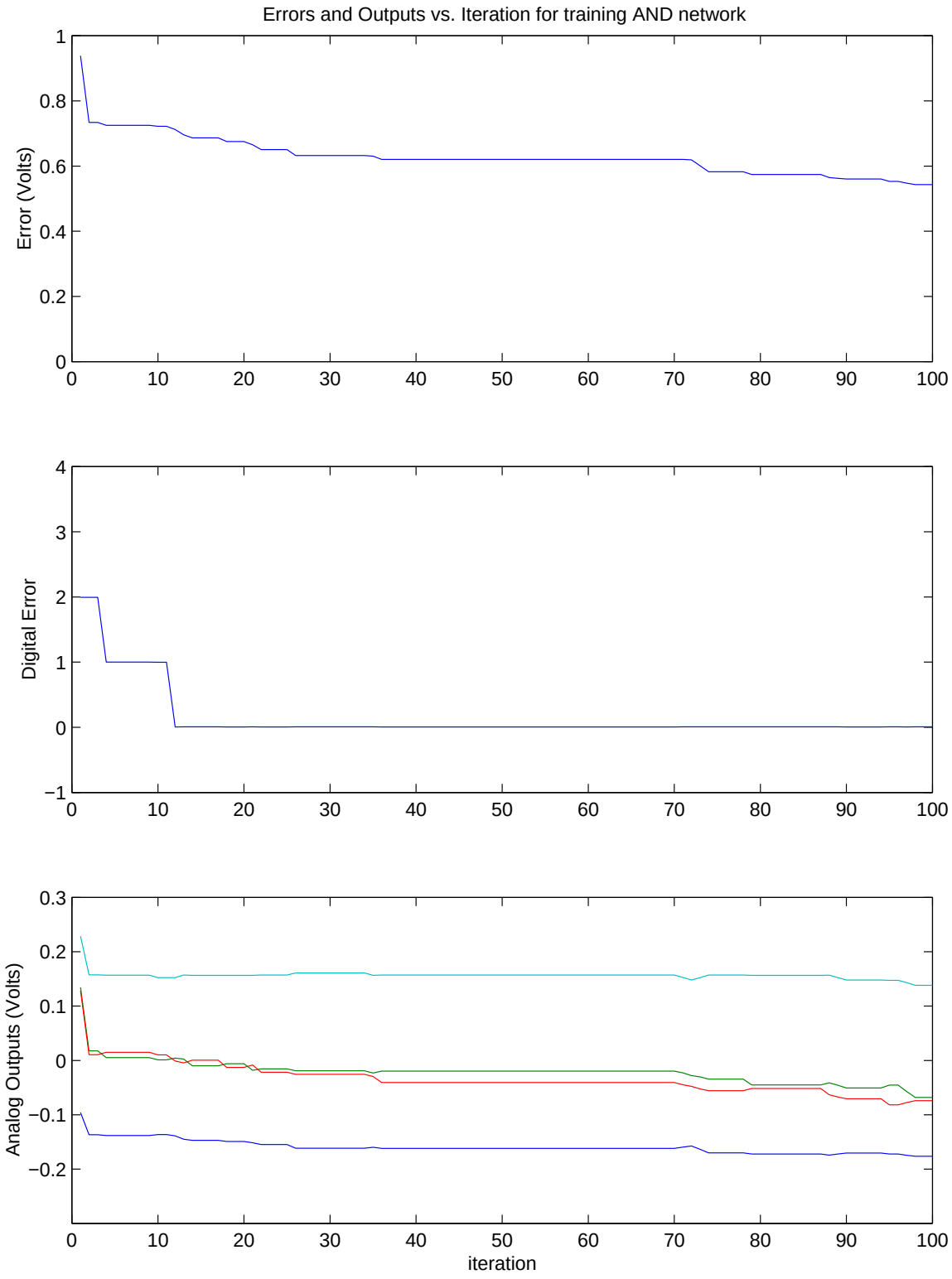


Figure 6.13: Training of a 2:1 network with the AND function

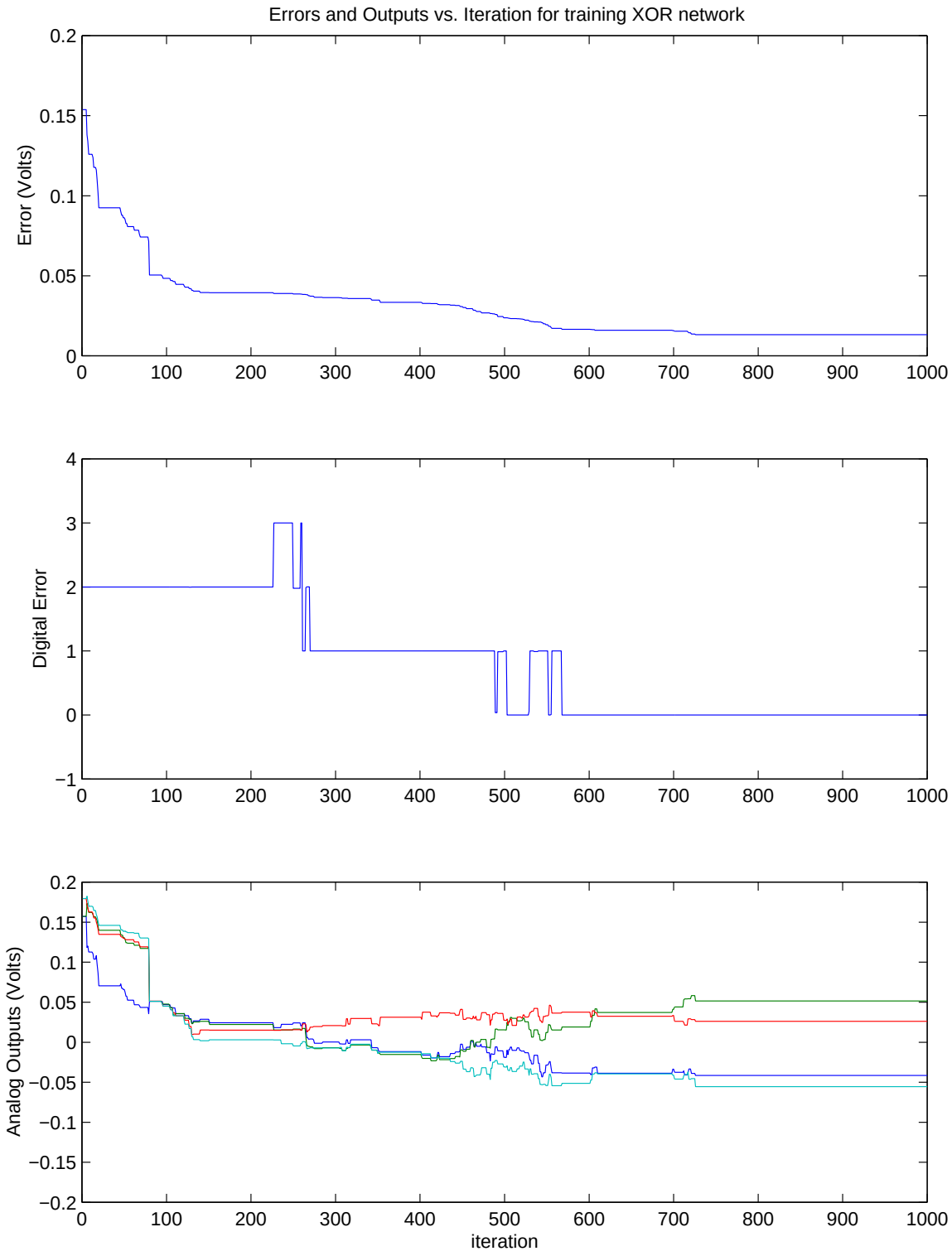


Figure 6.14: Training of a 2:2:1 network with the XOR function

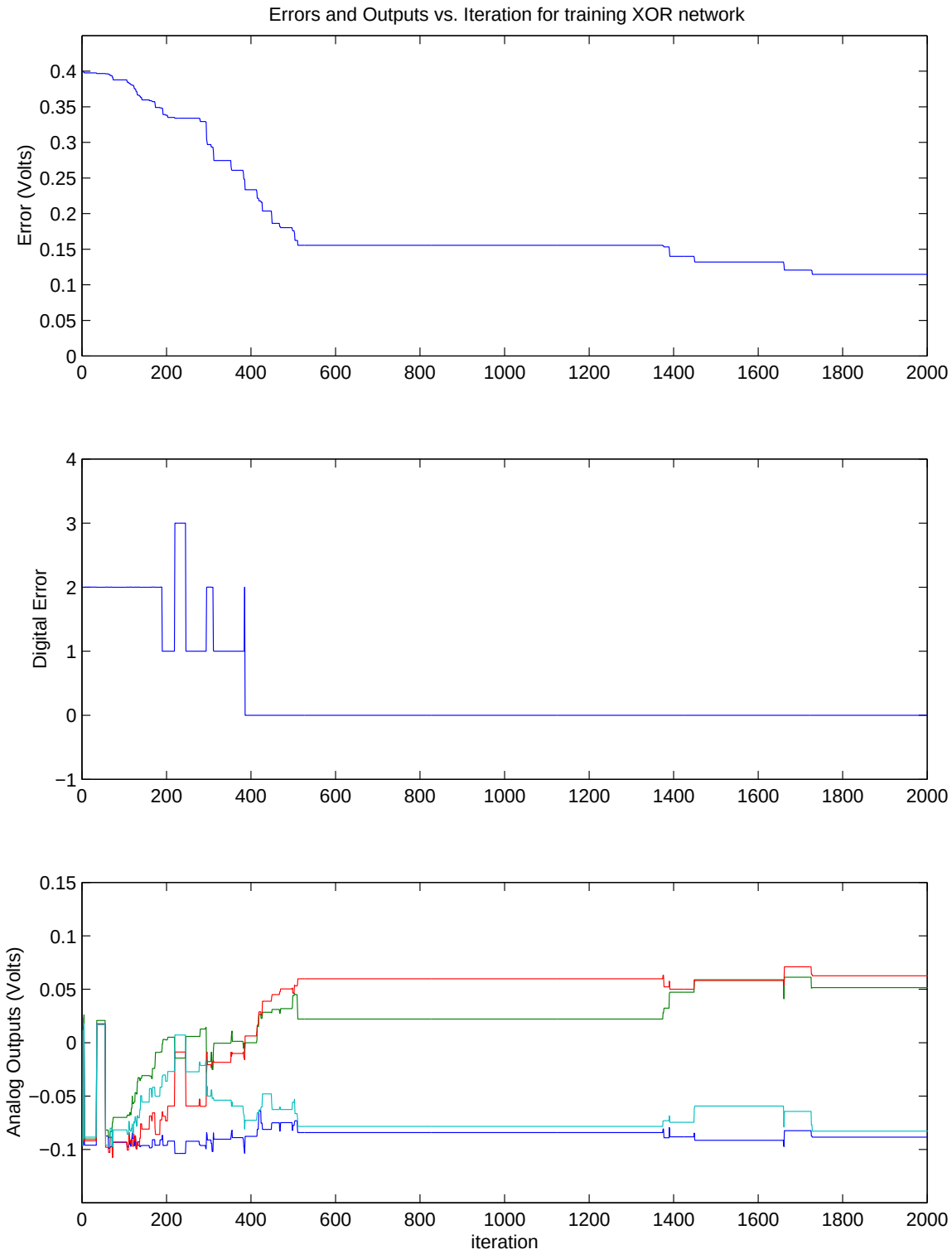


Figure 6.15: Training of a 2:2:1 network with the XOR function with more iterations

ation to another. This occurs when a weight value which is at maximum magnitude overflows and resets to zero. The weights are merely stored in counters, and no special circuitry was added to deal with these overflow conditions. It would require a simple logic block to ensure that if a weight is at maximum magnitude and was incremented, that it would not overflow and reset to zero. However, this circuitry would need to be added to every weight counter and would be an unnecessary increase in size. However, these overflow conditions should normally not be a problem. Since the algorithm only accepts weight changes that decrease the error, if an overflow and reset on a weight is undesirable, the weight reset will simply be discarded. In fact, the weight reset may occasionally be useful for breaking out of local minima, where a weight value has been pushed up against an edge which leads to a local minima, but a sign flip or significant weight reduction is necessary to reach the global minimum. In this type of situation, the network will be unwilling to increase the error as necessary to obtain the global minimum.

6.10 Conclusions

A VLSI neural network implementing a parallel perturbative algorithm has been demonstrated. Digital weights are used to provide stable weight storage. They are stored and updated via up/down counters and registers. Analog multipliers are used to decrease the space that a full parallel array of digital multipliers would occupy in a large network. The network was successfully trained on the 2-input AND and XOR functions. Although the functions learned were digital, the network is able to accept analog inputs and provide analog outputs for learning other functions.

The chip was trained using a computer in a chip-in-loop fashion. All of the local weight update computations and random noise perturbations were generated on chip. The computer performed the global operations including applying the inputs, measuring the outputs, calculating the error function and applying algorithm control signals. This chip-in-loop configuration is the most likely to be used if the chip were used as a fast, low power component in a larger digital system. For example, in a

handheld PDA, it may be desirable to use a low power analog neural network to assist in the handwriting recognition task with the microprocessor performing the global functions.

Chapter 7 Conclusions

7.1 Contributions

7.1.1 Dynamic Subthreshold MOS Translinear Circuits

The main contribution of this section involved the transformation of a set of floating gate circuits useful for analog computation into an equivalent set of circuits using dynamic techniques. This makes the circuits more robust, and easier to work with since they do not require using ultraviolet illumination to initialize the circuits. Furthermore, the simplicity with which these circuits can be combined and cascaded to implement more complicated functions was demonstrated.

7.1.2 VLSI Neural Networks

The design of the neuron circuit, although similar to another neuron design[22], is new. The main distinction is a change in the current to voltage conversion element used. The previous element was a diode connected transistor. This element proved to have too large of a gain, and, in particular, learning XOR was difficult with such high neuron gain. Thus, a diode connected cascoded transistor with linearized gain and a gain control stage was used instead.

The parallel perturbative training algorithm with simplified weight update rule was used in the serial weight bus based neural network. Although many different algorithms could have been used, such as serial weight perturbation, this was done with an eye toward the eventual implementation of the fully parallel perturbative implementation. Excellent training results were shown for training of a 2 input, 1 output network on the AND function and for training a 2 input, 2 hidden layer, 1 output network on the XOR function. Furthermore, an XOR network was trained which was initialized with “ideal” weights. This was done in order to show the effects

of circuit offsets in making the outputs initially incorrect, but also the ability to significantly decrease the training time required by using good initial weights.

The overall system design and implementation of the fully parallel perturbative neural network with local computation of the simplified weight update rule was also demonstrated. Although many of the subcircuits and techniques were borrowed from previous designs, the overall system architecture is novel. Excellent training results were again shown for a 2 input, 1 output network trained on the AND function and for a 2 input, 2 hidden layer, 1 output network trained on the XOR function.

Thus, the ability to learn of a neural network using analog components for implementation of the synapses and neurons and 6 bit digital weights has been successfully demonstrated. The choice of 6 bits for the digital weights was chosen to demonstrate that learning was possible with limited bit precision. The circuits can easily be extended to use 8 bit weights. Using more than 8 bits may not be desirable since the analog circuitry itself may not have significant precision to take advantage of the extra bits per weight. Significant strides can be taken to improve the matching characteristics of the analog circuits, but then the inherent benefits of using a compact, parallel, analog implementation may be lost.

The size of the neuron cell in dimensionless units was 300λ x 96λ, the synapse was 80λ x 150λ, and the weight counter/register was 340λ x 380λ. In the 1.2μm process used to make the test chips, λ was equal to 0.6μm. In a modern process, such as a 0.35μm process, it would be possible to make a network with over 100 neurons and over 10,000 weights in a 1cm x 1cm chip.

7.2 Future Directions

The serial weight bus neural network and parallel perturbative neural network can be combined. By adding a serial weight bus to the parallel perturbative neural network, it would also make it possible to initialize the neural network with weights obtained from computer pretraining. In a typical design cycle, significant computer simulation would be carried out prior to circuit fabrication. Part of this simulation may involve detailed

training runs of the network using either idealized models or full mixed-signal training simulation with the actual circuit. With the serial weight bus, it would then make it possible to initialize the network with the results of this simulation. Additionally, for certain applications such as handwriting recognition, it may be desirable to train one chip with a standard set of inputs, then read out those weights and copy them to other chips. The next steps would also involve building larger networks for the purposes of performing specific tasks.

Bibliography

[1] B. Gilbert, "Translinear Circuits: A Proposed Classification", *Electronics Letters*, vol. 11, no. 1, pp. 14-16, 1975.

[2] B. Gilbert, "Current-Mode Circuits From A Translinear Viewpoint: A Tutorial", *Analogue IC Design: The Current-Mode Approach*, C. Toumazou, F. J. Ledgey, and D. G. Haigh, Eds., London: Peter Peregrinus Ltd., pp. 11-91, 1990.

[3] B. Gilbert, "A Monolithic Microsystem for Analog Synthesis of Trigonometric Functions and Their Inverses", *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 6, pp. 1179-1191, 1982.

[4] A. G. Andreou and K. A. Boahen, "Translinear Circuits in Subthreshold MOS", *Analog Integrated Circuits and Signal Processing*, vol. 9, pp. 141-166, 1996.

[5] T. Serrano-Gotarredona, B. Linares-Barranco, and A. G. Andreou, "A General Translinear Principle for Subthreshold MOS Transistors", *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 46, no. 5, May 1999.

[6] K. Bult and H. Wallinga, "A Class of Analog CMOS Circuits Based on the Square-Law Characteristic of an MOS Transistor in Saturation", *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 3, June 1987.

[7] E. Seevinck and R. J. Wiegerink, "Generalized Translinear Circuit Principle", *IEEE Journal of Solid-State Circuits*, vol. 26, no. 8, Aug. 1991.

[8] P. R. Gray and R. G. Meyer, *Analysis and Design of Analog Integrated Circuits*, 3rd Ed., New York: John Wiley and Sons, 1993.

[9] R. J. Wiegerink, "Computer Aided Analysis and Design of MOS Translinear Circuits Operating in Strong Inversion", *Analog Integrated Circuits and Signal Processing*, vol. 9, pp. 181-187, 1996.

[10] W. Gai, H. Chen and E. Seevinck, "Quadratic-Translinear CMOS Multiplier-Divider Circuit", *Electronics Letters*, vol. 33, no. 10, May 1997.

[11] S. I. Liu and C. C. Chang, "A CMOS Square-Law Vector Summation Circuit", *IEEE Transactions on Circuits and Systems - II: Analog and Digital Signal Processing*, vol. 43, no. 7, July 1996.

[12] C. Dualibe, M. Verleysen and P. Jespers, "Two-Quadrant CMOS Analogue Divider", *Electronics Letters*, vol. 34, no. 12, June 1998.

[13] X. Arreguit, E. A. Vittoz, and M. Merz, "Precision Compressor Gain Controller in CMOS Technology", *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 3, pp. 442-445, 1987.

[14] E. A. Vittoz, "Analog VLSI Signal Processing: Why, Where, and How?" *Analog Integrated Circuits and Signal Processing*, vol. 6, no. 1, pp. 27-44, July 1994.

[15] B. A. Minch, C. Diorio, P. Hasler, and C. A. Mead, "Translinear Circuits using Subthreshold floating-gate MOS transistors", *Analog Integrated Circuits and Signal Processing*, Vol. 9, No. 2, 1996, pp. 167-179.

[16] B. A. Minch, "Analysis, Synthesis, and Implementation of Networks of Multiple-Input Translinear Elements", Ph.D. Thesis, California Institute of Technology, 1997.

[17] K. W. Yang and A. G. Andreou, "A Multiple-Input Differential-Amplifier Based on Charge Sharing on a Floating-Gate MOSFET", *Analog Integrated Circuits and Signal Processing*, vol. 6, no. 3, 1994, pp. 167-179.

[18] E. Vittoz, "Dynamic Analog Techniques", *Design of Analog-Digital VLSI Circuits for Telecommunications and Signal Processing*, Ed. Franca J., Tsividis Y., Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[19] S. Aritome, R. Shirota, G. Hemink, T. Endoh, and F. Masuoka, "Reliability Issues of Flash Memory Cells", *Proceedings of the IEEE*, vol. 81, no. 5, pp. 776-788, May 1993,

[20] R. W. Keyes, *The Physics of VLSI Systems*, New York: Addison-Wesley, 1987.

[21] J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, and D. Lippe, "A Parallel Gradient Descent Method for Learning in Analog VLSI Neural Networks", *Advances in Neural Information Processing Systems*, San Mateo, CA: Morgan Kaufman Publishers, vol. 5, pp. 836-844, 1993.

[22] R. Coggins, M. Jabri, B. Flower, and S. Pickard, "A Hybrid Analog and Digital VLSI Neural Network for Intracardiac Morphology Classification", *IEEE Journal of Solid-State Circuits*, vol. 30, no. 5, pp. 542-550, May 1995.

[23] M. Jabri and B. Flower, "Weight Perturbation: An Optimal Architecture and Learning Technique for Analog VLSI Feedforward and Recurrent Multilayer Networks", *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 154-157, Jan. 1992.

[24] B. Flower and M. Jabri, "Summed Weight Neuron Perturbation: An O(N) Improvement over Weight Perturbation", *Advances in Neural Information Processing Systems*, San Mateo, CA: Morgan Kaufman Publishers, vol. 5, pp. 212-219, 1993.

[25] G. Cauwenberghs, "A Fast Stochastic Error-Descent Algorithm for Supervised Learning and Optimization", *Advances in Neural Information Processing Systems*, San Mateo, CA: Morgan Kaufman Publishers, vol. 5, pp. 244-251, 1993.

[26] P. W. Hollis and J. J. Paulos, "A Neural Network Learning Algorithm Tailored for VLSI Implementation", *IEEE Transactions on Neural Networks*, vol. 5, no. 5, pp. 784-791, Sept. 1994.

[27] G. Cauwenberghs, "Analog VLSI Stochastic Perturbative Learning Architectures", *Analog Integrated Circuits and Signal Processing*, 13, 195-209, 1997.

[28] C. Diorio, P. Hasler, B. A. Minch, and C. A. Mead, "A Single-Transistor Silicon Synapse", *IEEE Transactions on Electron Devices*, vol. 43, no. 11, pp. 1972-1980, Nov. 1996.

[29] C. Diorio, P. Hasler, B. A. Minch, and C. A. Mead, "A Complementary Pair of Four-Terminal Silicon Synapses", *Analog Integrated Circuits and Signal Processing*, vol. 13, no. 1-2, pp. 153-166, May-June 1997.

[30] C. Mead, *Analog VLSI and Neural Systems*, New York: Addison-Wesley, 1989.

[31] S. W. Golomb, *Shift Register Sequences*, Revised Ed., Laguna Hills, CA: Aegean Park Press, 1982.

[32] P. Horowitz, and W. Hill, *The Art of Electronics*, Second Ed., Cambridge, UK: Cambridge University Press, pp. 655-664, 1989.

[33] J. Alspector, B. Gupta, and R. B. Allen, "Performance of a stochastic learning microchip" in *Advances in Neural Information Processing Systems 1*, D. S. Touretzky, Ed., San Mateo, CA: Morgan Kaufman, 1989, pp. 748-760

[34] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Second Ed., Reading, Mass.: Addison-Wesley, 1993.

[35] R. Gregorian and G. C. Temes, *Analog MOS Integrated Circuits for Signal Processing*, New York: John Wiley & Sons, 1986.

[36] J. Alspector, J. W. Gannett, S. Haber, M. B. Parker, and R. Chu, "A VLSI-Efficient Technique for Generating Multiple Uncorrelated Noise Source and Its

Application to Stochastic Neural Networks", *IEEE Transactions on Circuits and Systems*, vol. 38, no. 1, pp. 109-123, Jan. 1991.

[37] S. Wolfram, *Theory and Applications of Cellular Automata*, World Scientific Publishing Co. Pte. Ltd., 1986.

[38] A. Dupret, E. Belhaire, and P. Garda, "Scalable Array of Gaussian White Noise Sources for Analogue VLSI Implementation", *Electronics Letters*, vol. 31, no. 17, pp. 1457-1458, Aug. 1995.

[39] G. Cauwenberghs, "An Analog VLSI Recurrent Neural Network Learning a Continuous-Time Trajectory", *IEEE Transactions on Neural Networks*, vol. 7, no. 2, pp. 346-361, March 1996.

[40] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation*, New York: Addison-Wesley, 1991.

[41] M. L. Minsky and S. A. Papert, *Perceptrons*, Cambridge: MIT Press, 1969.

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors", *Nature*, vol. 323, pp. 533-536, Oct. 9, 1986.

[43] T. Morie and Y. Amemiya, "An All-Analog Expandable Neural Network LSI with On-Chip Backpropagation Learning", *IEEE Journal of Solid-State Circuits*, vol. 29, no. 9, pp. 1086-1093, Sept. 1994.

[44] A. Dembo and T. Kailath, "Model-Free Distributed Learning", *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 58-70, March 1990.

[45] B. Widrow and M. A. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation", *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415-1442, Sept. 1990.

[46] Y. Xie and M. Jabri, "Analysis of the effects of quantization in multilayer neural networks using a statistical model", *IEEE Transactions on Neural Networks*, vol. 3, no. 2, pp. 334-338, 1992.

[47] A. F. Murray, "Multilayer Perceptron Learning Optimized for On-Chip Implementation: A Noise-Robust System", *Neural Computation*, no. 4, pp. 367-381, 1992.

[48] Y. Maeda, J. Hirano, and Y. Kanata, "A Learning Rule of Neural Networks via Simultaneous Perturbation and Its Hardware Implementation", *Neural Networks*, vol. 8, no. 2, pp. 251-259, 1995.

[49] V. Petridis and K. Paraschidis, "On the Properties of the Feedforward Method: A Simple Training Law for On-Chip Learning", *IEEE Transactions on Neural Networks*, vol. 6, no. 6, Nov. 1995.

[50] D. B. Kirk, D. Kerns, K. Fleischer, and A. H. Barr, "Analog VLSI Implementation of Multi-dimensional Gradient Descent", *Advances in Neural Information Processing Systems*, San Mateo, CA: Morgan Kaufman Publishers, vol. 5, pp. 789-796, 1993.

[51] J. S. Denker and B. S. Wittner, "Network Generality, Training Required, and Precision Required", *Neural Information Processing Systems*, D. Z. Anderson Ed., New York: American Institute of Physics, pp. 219-222, 1988.

[52] P. W. Hollis, J. S. Harper, and J. J. Paulos, "The Effects of Precision Constraints in a Backpropagation Learning Network", *Neural Computation*, vol. 2, pp. 363-373, 1990.

[53] M. Stevenson, R. Winter, and B. Widrow, "Sensitivity of Feedforward Neural Networks to Weight Errors", *IEEE Transactions on Neural Networks*, vol. 1, no. 1, March 1990.

[54] A. J. Montalvo, P. W. Hollis, and J. J. Paulos, "On-Chip Learning in the Analog Domain with Limited Precision Circuits", *International Joint Conference on Neural Networks*, vol. 1, pp. 196-201, June 1992.

[55] T. G. Clarkson, C. K. Ng, and Y. Guan, "The pRAM: An Adaptive VLSI Chip", *IEEE Transactions on Neural Networks*, vol. 4, no. 3, May 1993.

[56] A. Torralba, F. Colodro, E. Ibáñez, and L. G. Franquelo, "Two Digital Circuits for a Fully Parallel Stochastic Neural Network", *IEEE Transactions on Neural Networks*, vol. 6, no. 5, Sept. 1995.

[57] H. Card, "Digital VLSI Backpropagation Networks", *Can. J. Elect. & Comp. Eng.*, vol. 20, no. 1, 1995.

[58] H. C. Card, S. Kamarsu, and D. K. McNeill, "Competitive Learning and Vector Quantization in Digital VLSI Systems", *Neurocomputing*, vol. 18, pp. 195-227, 1998.

[59] H. C. Card, C. R. Schneider, and R. S. Schneider, "Learning Capacitive Weights in Analog CMOS Neural Networks", *Journal of VLSI Signal Processing*, vol. 8, pp. 209-225, 1994.

[60] G. Cauwenberghs and A. Yariv, "Fault-Tolerant Dynamic Multilevel Storage in Analog VLSI", *IEEE Transactions of Circuits and Systems - II: Analog and Digital Signal Processing*, vol. 41, no. 12, Dec. 1994.

[61] G. Cauwenberghs, "A Micropower CMOS Algorithmic A/D/A Converter", *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 42, no. 11, Nov. 1995.

[62] P. Mueller, J. Van Der Spiegel, D. Blackman, T. Chiu, T. Clare, C. Donham, T. P. Hsieh, and M. Loinaz, "Design and Fabrication of VLSI Components for a General Purpose Analog Neural Computer", *Analog VLSI Implementation of Neural Systems*, C. Mead and M. Ismail, Eds., Norwell, Mass.: Kluwer, pp. 135-169, 1989.

[63] R. B. Allen and J. Alspector, "Learning of Stable States in Stochastic Asymmetric Networks", *IEEE Transactions on Neural Networks*, vol. 1, no. 2, June 1990.

[64] G. Cauwenberghs, C. F. Neugebauer, and A. Yariv, "Analysis and Verification of an Analog VLSI Incremental Outer-Product Learning System", *IEEE Transactions on Neural Networks*, vol. 3, no. 3, May 1992.

[65] C. Diorio, S. Mahajan, P. Hasler, B. Minch, and C. Mead, "A High-Resolution Nonvolatile Analog Memory Cell", *Proceedings of the 1995 IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 2233-2236, 1995.

[66] A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos, "An Analog VLSI Neural Network with On-Chip Perturbation Learning", *IEEE Journal of Solid-State Circuits*, vol. 32, no. 4, April 1997.

[67] P. Baldi and F. Pineda, "Contrastive Learning and Neural Oscillations", *Neural Computation*, vol. 3, no. 4, pp. 526-545, Winter 1991.