BINARY OSCILLATOR COMPUTING AND THRESHOLD LOGIC

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

By
Jay Berry (22577141)
Department of Computing and Mathematics

Contents

List of Tables

List of Figures

Abstract

This thesis is structured as a three pronged attack.

In Part I we explore mathematical models of biological neurons, resulting in appropriate focus on the Fitzhugh Nagumo model, and the Josephson Junction model. We reproduce computational simulations of both models for individual neurons, and combined networks of neurons for the Fitzhugh Nagumo model. This demonstrates the feasibility of instantiating a binary oscillator threshold effect in the real world. We give a relevant digest of leaps in computing power over the past century, and explain how Josephson Junction binary oscillator based computing may be the next big paradigm shift in conventional computing.

In Part II we explore threshold logic, creating basic ALUs out of networks of Fitzhugh Nagumo neurons as requested. We explain some of the advantages of threshold logic over digital logic, primarily in the reductions in complexities. Our creations give a taste of the new possibilities opened up by using threshold logic based components within computing. We create simple, novel algorithms for building extensible threshold logic circuitry, which inspires the development of more algorithms in future work for task specific hardware.

In Part III we explore the compatibility of threshold logic and digital logic, with the goal of easing the transition from present day transistor based technology to hypothetical Josephson Junction conventional computers. Our research consists primarily of synthetic threshold logic creations, providing highly portable work. We design various novel ALUs and algorithms with mathematical derivations, which result in the simplification of transistor based circuitry, justifying the switch-over.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number 57367.

Signed: [illegible]

Date: 2023/10/06

Acknowledgements

A big thank you to my supervisor Dr Stephen Lynch for overseeing this thesis.

Chapter 1

Introduction + Literature Survey

We present a brief history of the fundamental electronic logic components used over the past two centuries. The journey arrives at our responsibility to pursue the next generation of computing technology.

1.1 Vacuum Tubes

In the late 19th century Thomas Edison was trying to discover why lightbulbs discoloured only one one side of their glass over time, closest to the positive side of the filament. In some experiments involving specially made lightbulbs containing a third wire, separate from the filament, Edison discovered that a current would flow through the third wire if it were given a positive potential difference relative to the filament. He concluded that the filament acted as the source of the charge.

The high temperature to which the filament is heated provides some flowing electrons with enough energy to overcome the work function of the filament's material and break free. This is known as 'thermionic emission'. Thermionic emission had been discovered independently before Edison by Becquerel (1853) and Guthrie (1873), however Edison popularised the knowledge though related inventions. The interior of a lightbulb is a vacuum to prevent the filament from burning up. The emitted electrons are attracted towards the positive terminal of the lightbulb, and sometimes overshoot as they accelerate towards it, hitting the corresponding side of the lightbulb and discolouring it.

Following Edison's work, J. A. Fleming developed a device similar to Edison's three-terminal bulb in 1904. The two-terminal device consisted of a filament and a metal plate opposing each other within a bulb. This component acted as a diode, the first generation of the electrical diode (Fleming, 1904). Three years later Lee de Forest added another electrode to the diode, attached to a metal wire (known as the 'grid'), which varies how much current flows from the filament to the plate depending on the grid's polarity. This became known as the 'triode'. This invention was designed for signal amplification, a small change in the grid's voltage resulting in a large change in the potential difference between the plate and ground. In 1908 de Forest obtained a patent for his amplifier (de Forest, 1908), which would be later purchased by AT&T in 1913 when the applications of the triode's ability to amplify signals were realised. Mentions of AT&T, in particular AT&T Bell Labs, will occur several times again throughout this section, Bell Labs playing a key role in many 20th century technological inventions.

A new era of electronic inventions was opened with the popularisation of the triode and its amplification capabilities. AT&T implemented the triode into their telephone networks, which made possible very-long-distance phone calls, using triodes to boost the signal along the way. This lead to the first transcontinental phone call in 1915 from New York to San Francisco. Triode radios would replace fragile crystal radios, and in the 1920s radio broadcasting became feasible. In the research environment it was realised that vacuum tube triodes could replace electromechanical relays as the fundamental component in logic circuits, and early computers. The triode's output could be considered as a 0 or 1, controlled by input of the grid electrode. This is analogous to the transistor, which we will discuss soon. Triodes used more power than relays, but operated faster, and made no sound, compared to the clack-ety sound of relays, proving to be a better alternative. The first general purpose programmable computer, ENIAC, was created from vacuum tube triodes and diodes in 1945, and many more followed up until the mid 1960s.

1.2 The Transistor

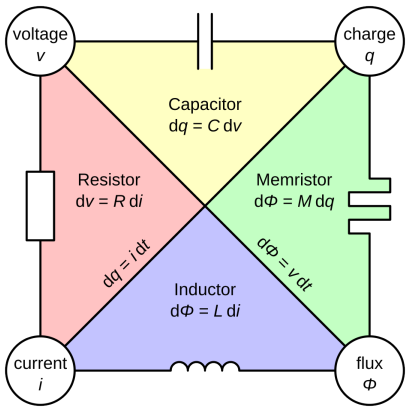
The triode reigned as the dominant component in consumer electronics such as the television and radio until the 1960s, when the transistor (in particular the MOSFET) would take over. This takeover was not overnight however. Work on transistors and semiconductors started as late as the 1920s, when Lilienfeld (1925) and Heil (1934) both patented the field-effect transistor. Although these patents existed, the former also expired before a working physical prototype could be created, due to manufacturing limitations at the time; the work remained theoretical for quite a while. It took until 1947, over a decade later, for William Shockley's team at Bell Labs to create a working transistor, and in 1948 Shockley created a working version of the now well known bipolar junction transistor. Later in the 1960s Mohamed Atalla and Dawon Kahng, also working at Bell Labs, created working models of the MOSFET transistor, over 30 years after Lilienfeld's work. The MOSFET is a faster, easier to manufacture device with lower power consumption than the bipolar junction transistor, and it is the type of transistor found most commonly in IC chips. It has been the undisputed king of electronic digital logic for decades. The MOSFET is the most manufactured item in history, with an estimated 2.9 sextillion (2.9×10^{21}) produced up to 2014 (Fudzilla, 2014).

To squeeze down transistor based chips to smaller footprints, and to fit more transistors within a given size die, chip producers such as Intel have invested much time and money into technologies such as 3D FinFET transistors, at the scale of 22nm in 2011 (Intel Newsroom, 2011). At even smaller levels Intel have had to account for the effects of quantum mechanics at the small scale of 5nm and below, due to quantum tunnelling of electrons from one track to another. This possibly hinted at an end to Moore's Law (AnandTech, 2019), or at least a slowing down of Moore's Law using this current generation component.

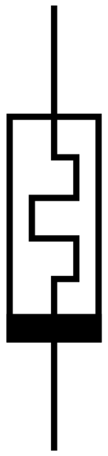
Vacuum tubes are incredibly large compared to how microscopically small MOSFET transistors can now be manufactured. Apple's M1 Ultra (2022) processors fit over 100 billion transistors on a single chip (Shankland, 2022), with a footprint smaller than the diameter of a single vacuum tube. Decades of engineering advancements in transistor density count, power reduction, and CPU architecture design have rendered vacuum tubes an obsolete relic of the past. So too, we now take on the task of investigating the components and technologies which may render transistors obsolete one day.

1.3 The Memristor

In 1971 Leon Chua published a paper describing a possible fourth fundamental electronic component, the 'memristor'. This component directly links together electric charge and magnetic flux (Chua, 1971). A portmanteau of 'memory resistor', the memristor demonstrates its memory abilities in the hysteresis exhibited in its modelling differential equations. We will later discuss how this hysteresis is exploited by Fang et al (2022) in their paper which demonstrates the use of a memristor in binary oscillator computing. It took until 2008 for HP labs to create the first working prototype of a memristor (Strukov, 2008)! This is a similar situation to the discovery of the transistor field effect, to the first working transistor; decades in apart in time. Just as the vacuum tube reigned supreme before transistors were made viable to manufacture, so too the transistor may only be dominant until binary oscillator computing is made feasible using memristors, or perhaps the final component we discuss, Josephson Junctions.


voltage
v
charge
q
Capacitor
$\mathrm{d}q = C\,\mathrm{d}v$
Resistor
$\mathrm{d}v = R\,\mathrm{d}i$
Memristor
$\mathrm{d}\Phi = M\,\mathrm{d}q$
$\mathrm{d}q = i\,\mathrm{d}t$
$\mathrm{d}\Phi = v\,\mathrm{d}t$
Inductor
$\mathrm{d}\Phi = L\,\mathrm{d}i$
current
i
flux
Φ

(a) The relations between the four fundamental components, memristor on the right. CC-BY-SA by Wikipedia user Parcly Taxel



(b) The circuit diagram symbol for a memristor. CC-BY-SA by Wikipedia user MovGP0

1.4 Josephson Junctions

In 1962 British Physicist Brian Josephson made predictions of the possibility of electron-pairs passing through a barrier placed between two superconducting materials. This may remind one of quantum tunneling, originally a hypothetical consequence of solving Schrödinger's equation with a thick potential barrier separating two regions. Indeed quantum tunneling occurs for simple, single particles, and Josephson's work predicts tunneling for electron pairs which can form at superconducting temperatures (Josephson, 1962). Both hypothetical predictions turn out to be correct in reality, and Josephson's work won the Nobel Prize in Physics eleven years after his publication. These superconductor-insulator-superconductor (SIS) constructions are known as 'Josephson Junctions'. We will discuss later how Josephson Junctions can be coupled together to create a component which can be used as a 'neuron' to exhibit a threshold effect for binary oscillator computing.

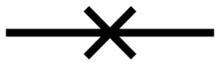


Figure 1.1: The circuit diagram symbol for a Josephson Junction.
CC-BY-SA by Wikipedia user Miraceti

1.5 Models for biological neurons

We now discuss some mathematical models of biological neurons. The evolution of models over the past century presents us with much inspiration, and we ultimately choose the Fitzhugh Nagumo and Josephson Junction models as our route into using threshold oscillatory effects found in neurons for binary oscillator computing.

1.5.1 Hodgkin-Huxley (1952)

Hodgkin and Huxley (1952) performed experiments on squid axons in order to formulate a model for neuron electrical activity. Their work produced a set of four differential equations, one of the components of which can be used to describe the potential difference at the exterior of an axon. This work is the direct result of experimentation with real biology, which grounds-in-reality further work using the HH model.

1.5.2 Fitzhugh-Nagumo (1961+1962)

Fitzhugh (1961), and later Nagumo et al (1962) provide a simplified version of the Hodgkin-Huxley model's mechanics.

Fitzhugh's work reduces down the HH equations from a system of differential equations of four unknowns to just two unknowns, providing "a simplified but central unifying concept for the theoretical study of axon physiology" (Fitzhugh, 1961:446). This is done to better understand the dynamics of the differential equations when all four unknowns are needed for an impulse in the HH model. Reducing to just two unknowns allows for phase plane analysis, which is easier for a human to inspect. There is not too much loss in biophysical meaningfulness within this reduction, which allows one to keep the intuitive parallels running between biology and electronics; Nagumo et al (1962) analogises these two unknowns to a neuron's membrane voltage, and sodium-potassium currents.

Fitzhugh's paper uses differential equations that are a generalisation of the van der Pol equations, published by Balthasar van der Pol in 1926. The circuit used in van der Pol's paper consists of a resistor, capacitor, coupled inductors, and a triode vacuum tube (van der Pol, 1926). The triode was of course used for signal amplification, with the circuit's output attached to the grid voltage. Bonhoeffer (1948) published another set of equations that model the current in passivated iron wires. Fitzhugh took inspiration from these two close cousins of equations, creating the 'Bonhoeffer-van der Pol' (BVP) model. The BVP equations provide access to an extended scope of dynamical systems that cannot not be expressed with just a VDP circuit (Fitzhugh, 1961:445).

Fitzhugh's BVP equations:

$$\dot{x} = c(y + x - x^3/3 + z) \tag{1.1}$$

$$\dot{y} = -(x - a + by)/c \tag{1.2}$$

with x and y the primary variables, and a, b, c, z constants.

Fitzhugh investigates the nullclines of the phase plane for the BVP equations. The two nullclines are given by the equations

$$y = -x + x^3/3 - z \tag{1.3}$$

$$y = (a - x)/b \tag{1.4}$$

The special case of $a = b = z = 0$ for equation 1.3 yields the nullcline of the original Van der Pol circuit (discarding the second nullcline which does not exist, multiplying both sides by b). Fitzhugh investigates the single intersection point P of these two equations, the 'resting state', which can be obtained by solving them as a set of simultaneous equations. Using the method of characteristics for analysis of PDEs, Fitzhugh makes several conclusions. Of interest to us is how "The location of the singular point P and hence its stability depends on z" (Fitzhugh, 1961:450).

When z is zero P acts as a stable stationary point. From a biological point of view this can be seen as an inactive / at-rest neuron. Next, what happens as z becomes more and more negative is considered. An unstable solution is produced for $0 < -z < 0.35$, that is the trajectory of any path close to P spirals in to P. This is colloquially known in bifurcation theory as a subcritical 'orbit trap'. An analogy to this failure to reach a stable orbit is to imagine one applying increasing efforts into trying to start a chainsaw!

At around $0.35 < -z < 0.4$ bifurcation is observed in trajectories around P, leading to stable orbits for $0.4 < -z < 1.4$. Plotting either of x or y against time produces an oscillatory output, which can be thought of as neuron activation. Fitzhugh describes this effect of achieving oscillations for a great enough $-z$ as a 'threshold phenomena', 'excitable and oscillatory behavior', which is reflected in many non-linear dynamical systems, not just BVP (Fitzhugh, 1961). We reproduce Fitzhugh's experimentation on z in Part I of this thesis.

The work of Nagumo et al (1962) expands on Fitzhugh's paper, investigating the transmission of pulse signals. The paper entails simulating the nerve axon of an animal, because it had been observed that biological axons are subject to much less signal degradation over distance compared to conventional electrical lines. The paper puts forth new equations which can be used to transmit output from a circuit, similar to a synapse acting as the junction between neurons in the brain. This gives us our first hints towards connecting neurons together, either with electrical circuitry, or at least with simulated mathematical modeling.

1.5.3 Fang et al's Memristive FHN (2022)

Fang et al (2022) fork from the FHN model, by modifying the BVP circuit. They swap the non-linear resistor component found in a voltage-controlled VDP circuit for a memristor. In their paper phase plane analysis is done, similar to in Fitzhugh's 1961 paper, showing a spiking neuron model when an input threshold is reached. The existence of limit cycles is proven by invoking the Poincaré–Bendixson theorem on the differential equations, and by using computational simulations to visualise them. This gives experimental evidence that binary oscillator computing could be feasible if interest and investment grows in mass-manufacturing of memristors.

1.5.4 Other biological neuron models

For completeness sake we mention a few other models for biological neurons. These show that there are more models out there that can be used as inspiration in the field of brain-inspired computing, whose differential equations could perhaps be sought out in synthetic materials for the purpose of harnessing them for conventional computing.

The 'Morris-Lecar' model by C. Morris and H. Lecar produces another simplification of the HH model to two differential equations by studying barnacle neurons (Morris, C. and Lecar, H., 1981). Hindmarsh and Rose (1984) produce a set of three differential equations by considering sodium and potassium currents in biological neurons, building off of the FHN model. Izhikevich (2003) creates a hybrid of the HH model with 'integrate-and-fire' style differential equations, which are easier to model computationally via explicit formulae instead of as derivatives.

With the scene set with mathematical models for neurons laid out, and the components of the memristor and Josephson Junction explained, we show in Part I how coupled components form electronic neurons, whose parameters we can investigate.

Part I

Binary Oscillators

Chapter 2

Josephson Junction Neurons

2.1 JJN Differential Equation

Chalkiadakis and Hizanidis (2022) provide us with a good starting point for our work using Josephson Junction neurons. In particular here we obtain the differential equations which will be of central importance to our simulations.

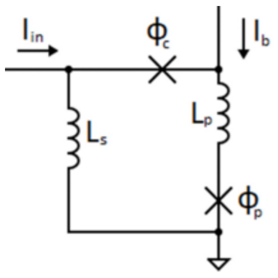


Figure 2.1: The circuit diagram of a JJN, modified from Chalkiadakis and Hizanidis (2022)

The coupled Josephson Junction neuron differential equations:

$$\ddot{\phi}_p + \Gamma\dot{\phi}_p + \sin(\phi_p) = -\lambda(\phi_p + \phi_c) + L_s i_{\text{in}} + (1 - L_p) i_{\text{b}} \tag{2.1}$$

$$\ddot{\phi}_c + \Gamma\dot{\phi}_c + \sin(\phi_c) = -\lambda(\phi_p + \phi_c) + L_s i_{\text{in}} - L_p i_{\text{b}} \tag{2.2}$$

L_s, L_p are the inductance values of the neuron's inductors. $\phi_p + \phi_c$ is the important quantity to us, which is analagous to the potential difference across a biological neuron's membrane, aka neuron activity.

We can rewrite these equations as a set of four first-order differential equations which will be easier to solve numerically

$$\dot{\phi}_p = \omega_p \tag{2.3}$$

$$\dot{\omega}_p = -\lambda(\phi_p + \phi_c) + L_s i_{\text{in}} + (1 - L_p) i_{\text{b}} - \Gamma\omega_p - \sin(\phi_p) \tag{2.4}$$

$$\dot{\phi}_c = \omega_c \tag{2.5}$$

$$\dot{\omega}_c = -\lambda(\phi_p + \phi_c) + L_s i_{\text{in}} - L_p i_{\text{b}} - \Gamma\omega_c - \sin(\phi_c) \tag{2.6}$$

2.2 Stability Region and Bifurcations

Chalkiadakis and Hizanidis explore the phase plane behaviour of the differential equations for varying the constants Γ and $\imath_{in}$ as seen in figure 2.2.

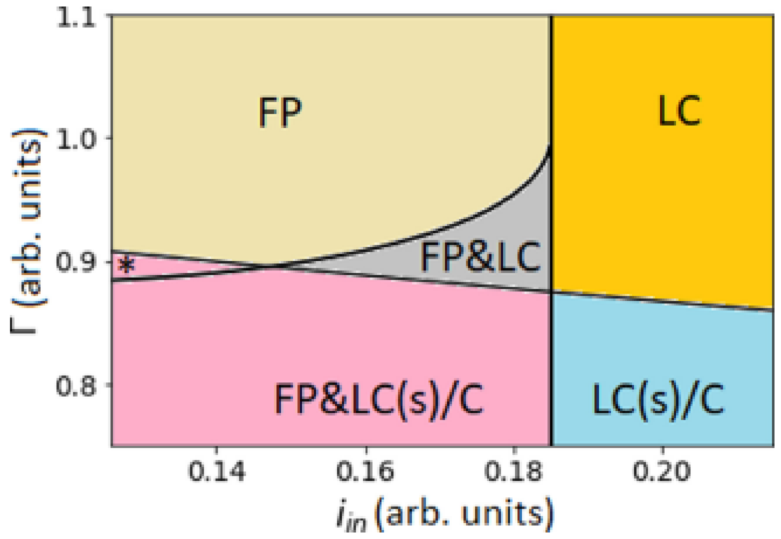
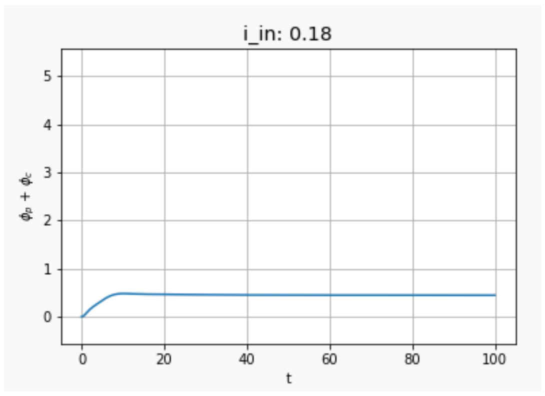


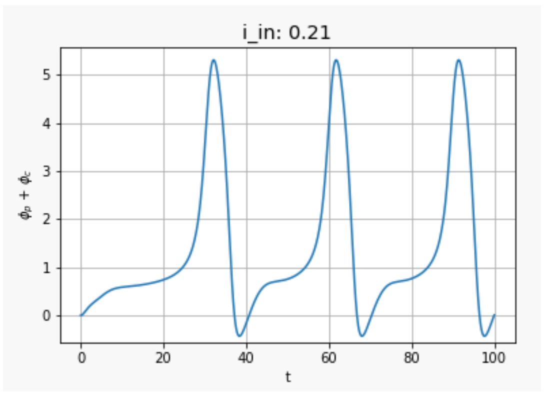
Figure 2.2: The regions of behaviour for a JJ neuron for $\lambda = 0.1$, $L_s = L_p = 0.5$, $i_{\text{b}} = 1.909$, modified from Chalkiadakis and Hizanidis (2022)

The grey region in figure 2.2 adapted from Chalkiadakis and Hizanidis (2022) is the ‘fixed point and limit cycle’ region, in which initial activity in a neuron dies out quickly (failure to start that chainsaw!). The orange limit cycle region to the right is the ideal place to represent active neuron activity. If we choose Γ correctly we can vary i_{in} in the region of ≈ 0.185 as our threshold value for switching on a neuron. We thus choose $\Gamma = 0.95$.

Using Python and `scipy`’s `odeint` we simulate the neuron for the given constants,



(a)



(b)

Figure 2.3: Output for `jj.ipynb`

and we indeed see the expected behaviour, with spiking output.

These pictures are of course better seen as animations instead of static.

In the future it may be possible to utilise Josephson Junction neurons as an alternative to the transistor. JJNs operate millions of times faster than human neurons, with very low power consumption. If only we had synapses to be able to connect JJNs together into networks! For the meantime we will consider using the FHN neuron model, as we know how to create synapses between FHN neurons.

Chapter 3

Fitzhugh Nagumo Neurons

Our equations for a coupled Fitzhugh Nagumo neuron are reduced to

$$\dot{u} = -u(u-\theta)(u-1) - v + i_{\text{in}} \tag{3.1}$$

$$\dot{v} = \epsilon(u - \gamma v) \tag{3.2}$$

for γ, ϵ, θ constants. Lynch (2023) gives appropriate values for these constants as $\gamma = 0.1$, $\epsilon = 0.1$, $\theta = 0.1$. We shall use these in Part II, joining neurons of FHN neurons together. Firstly however, we finish our work with Fitzhugh's equations.

We reproduce the phase plane diagrams that Fitzhugh (1961) details, using Python, with the addition of paths for solutions corresponding to different values of z.

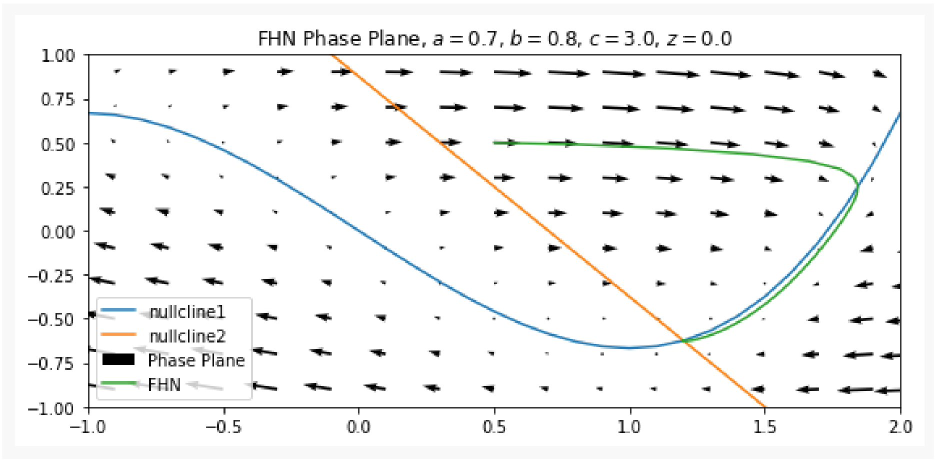


Figure 3.1: The green solution line corresponds to starting at $(0.5, 0.5)$, which traps to the nullcline intersection. Output of `test/libFHN/fhn-region.ipynb`

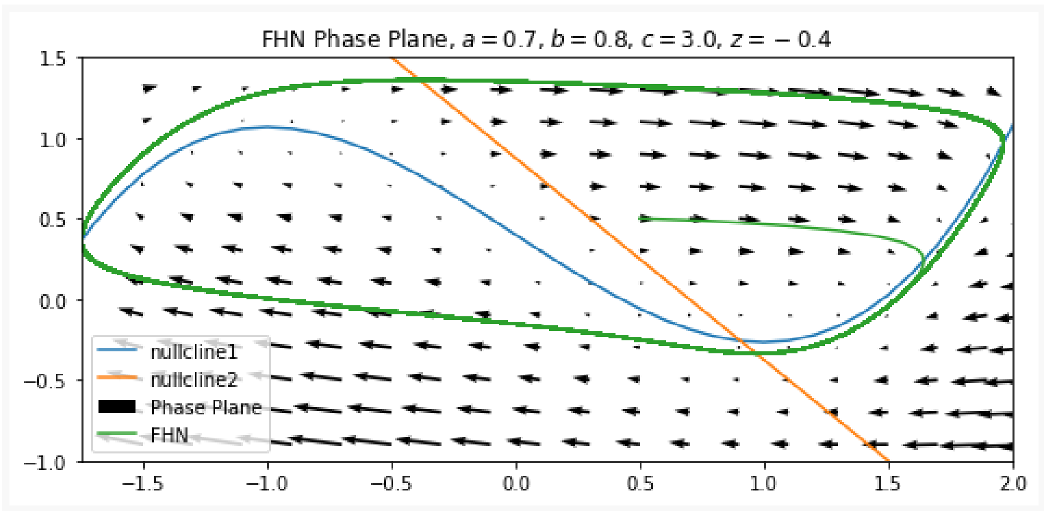
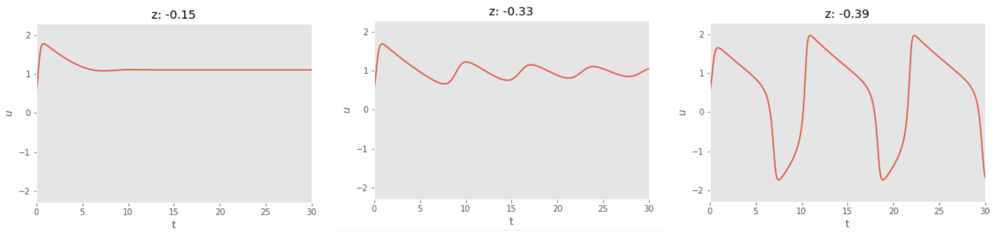


Figure 3.2: The green solution line shows a stable orbit for the sufficiently negative value of $z = -0.4$. Output of `test/libFHN/fhn-region.ipynb`



(a) $z = -0.15$, not-active (b) $z = -0.33$, subcritical (c) $z = -0.39$, active

Part II

Basic Threshold Logic

Chapter 4

Threshold Logic

We now discuss threshold logic, the crux of our neuron network creations.

We wish to mathematically replicate the quite discrete switching from off to on of a threshold oscillator. The way in which we construct our neuron networks is such that the output of a neuron is of the form $\varphi(\Sigma w_i x_i - b)$, where w_i are the weights of the inputs, and b is the offset 'bias', of which the total weighted input needs to overcome. φ is known as a 'threshold function' (also known as a 'transfer function'), as described below.

4.1 Threshold Functions

A threshold function's output makes the distinction between its input being classed as 'off' or 'on' very clear, by ideally taking only two values. A simple idealised threshold function is the Heaviside step function $H(x)$ described below.

$$H(x) = \begin{matrix} 0, & x \leq 0 \\ 1, & x > 0 \end{matrix}$$

We will denote a translated Heaviside step function $H(x - c)$ as $H_c(x)$. Here the value c is the 'threshold', or 'activation value'. We will make extensive use of our H_c functions in Part III, when dealing with idealised synthetic threshold logic. In reality however, such a discontinuous function unlikely to occur. Instead, a steep and continuous approximation to the step function is more feasible, and produces the same desired behaviour.

Borresen and Lynch (2012) use sigmoid threshold functions of the form

$$\sigma_c(x) = \frac{1}{1 + e^{-m(x-c)}} \tag{4.1}$$

when dealing with Fitzhugh-Nagumo binary oscillator threshold logic. For sufficiently large values of m, which make the transition from 0 to 1 sharp, σ_c is a good approximation to H_c. Likewise we use σ_c in our FHN oscillator networks.

It is worth noting also that σ_c is a good choice for computational efficiency. Alternative threshold functions such as $\tanh(x)$ require multiple hyperbolic calculations if code is not optimised correctly, whereas σ_c only requires the one exponential. Furthermore using the `numpy` Python package with x being an array allows the possibility for SIMD optimisations on sufficiently modern CPUs (NumPy, 2023).

On a related note $\sigma_c(x)$ is also useful in other areas of artificial intelligence, such as neural networks, where derivatives are required, as $\sigma_c(x)$'s derivative can be written in terms of itself. $\sigma_c'(x) = m\sigma_c(x)(1 - \sigma_c(x))$.

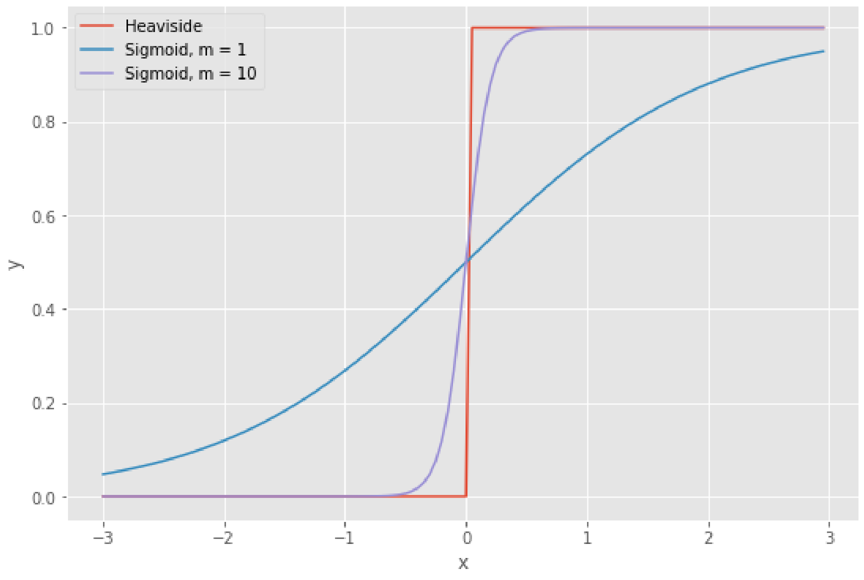


Figure 4.1: Various threshold functions

Chapter 5

The Half Adder

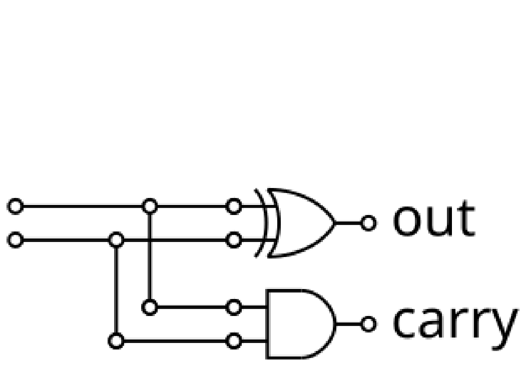
Arguably the most simple building block of arithmetic circuitry is the two bit input half adder. A half adder takes in two inputs I_1 and I_2 and produces output corresponding to truth table 5.1. This is used in binary addition, producing a two bit output: a 'sum' bit and a 'carry' bit.

I_1	I_2	out	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

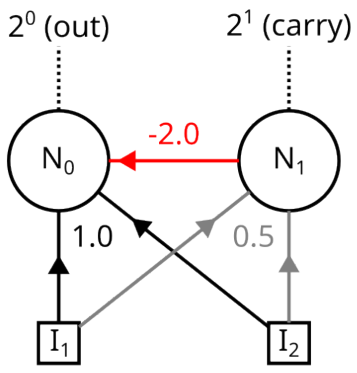
Table 5.1: Truth table of a half adder

One observes that the entries for 'out' correspond to XOR(I_1, I_2), and those of 'carry' are AND(I_1, I_2). Constructions of such gates using digital logic requires many transistors, whereas with networks of neurons using threshold logic we use only two neurons. For the remainder of part II, neurons in our neuron network diagrams have unit activation threshold, and unit output. Input weights are labeled with black and red colours corresponding to positive (excitatory) and negative (inhibitory) connections respectively. Figure 5.1(b) shows the construction of a half adder neuron network.

Using our Python library `libFHN` we can simulate a half adder using Fitzhugh Nagumo neurons. Observe in code excerpt 5.1 how pleasantly we can construct networks of neurons, specifying combinations of neurons as inputs with the `WeightedInput` class.



(a) Digital (boolean) logic gates



(b) Threshold logic neurons

Figure 5.1: Comparison of half adder constructions

```python
...

class HalfAdder(NeuronNetwork):
    def __init__(self) -> None:
        neuron_in_1 = Neuron(RangesConstInput([
            (500, 1000),
            (1500, 2000),
        ]))
        neuron_in_2 = Neuron(RangesConstInput([
            (1000, 2000),
        ]))
        neuron_carry = Neuron(WeightedInput(
            (neuron_in_1, 0.5),
            (neuron_in_2, 0.5),
        ))
        neuron_sum = Neuron(WeightedInput(
            (neuron_in_1, 1.0),
            (neuron_in_2, 1.0),
            (neuron_carry, -2.0),
        ))

        neurons = [neuron_in_1, neuron_in_2, neuron_sum,
    neuron_carry]

        super().__init__(neurons)
...
```

Listing 5.1: Code for libFHN test `half-adder.py`

Figure 5.2 shows the output voltages of the neurons in our half adder over time in arbitrary units. The spiky regions show the active (oscillating) neuron activity. O_s and O_c correspond correctly to the sum and carry bits for the two input neurons I_1 and I_2.

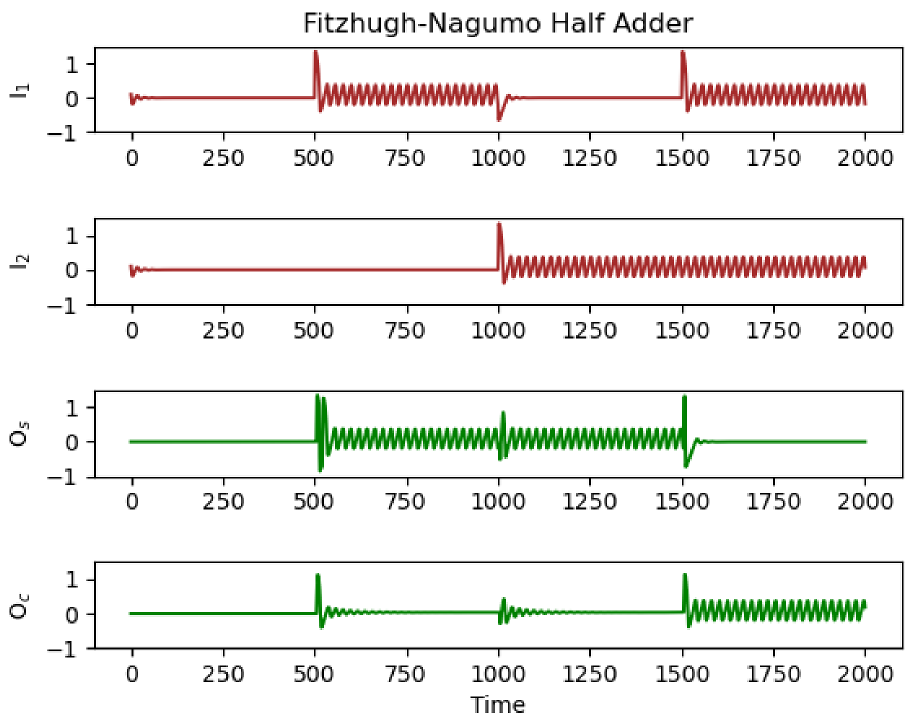


Figure 5.2: The output of `half-adder.py`

Chapter 6

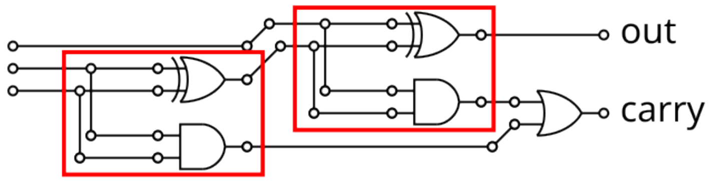
The Full Adder

The half adder is a nice component for adding two bits together, however in practicality we may wish for more computing power. Indeed we may wish to properly add together two n-bit binary numbers, which in digital logic requires an addition unit that can take in as input the carry bit from a previous addition unit, as well as outputting a carry bit. This addition unit is known as a 'full adder'. Truth table 6.1 gives the expected operation of the three-input two-output full adder. Rows are enumerated in a way such that they are grouped by total input, Σ_I, for the reader's ease.

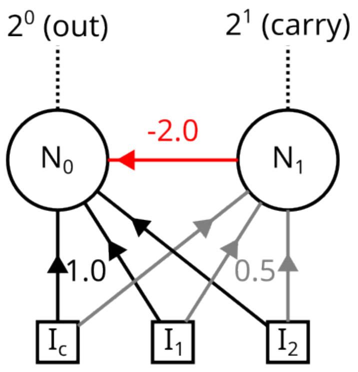
Σ_I	I_C	I_1	I_2	out	carry
0	0	0	0	0	0
1	1	0	0	1	0
	0	1	0	1	0
	0	0	1	1	0
2	1	1	0	0	1
	1	0	1	0	1
	0	1	1	0	1
3	1	1	1	1	1

Table 6.1: Truth table of a full adder

To construct such a circuit in digital logic, one typically employs two half-adders and an OR gate, as seen in 6.1(a). This explains the etymology of two 'half' adders making a 'full' adder! This of course requires over double the number of transistors than a half adder. Now observe the construction of a full adder using threshold logic in figure 6.1(b). Remarkably, we still only need two neurons. This is a prefect example of how threshold logic can reduce the component-count complexity of circuitry due to the high number of inter-neuron connections in threshold logic circuits. Further comparisons of the complexities of boolean logic vs threshold logic are deferred to part III of this thesis.



(a) Digital (boolean) logic gates. The two half adders are enclosed in red rectangles



(b) Threshold logic neurons

Figure 6.1: Comparison of full adder constructions

We again use `libFHN` to construct a simulation. In the output figure 6.2 observe both the sum and carry bit being on when all three inputs are on, effectively maxing out the range of numbers the adder can represent.

```
...

class FullAdder(NeuronNetwork):
    def __init__(self) -> None:
        neuron_in_1 = Neuron(RangesConstInput([
            (250, 500),
            (1000, 1500),
            (1750, 2000),
        ]))
        neuron_in_2 = Neuron(RangesConstInput([
            (500, 750),
            (1000, 1250),
            (1500, 2000),
        ]))
        neuron_in_3 = Neuron(RangesConstInput([
            (750, 1000),
            (1250, 2000),
        ]))
        neuron_carry = Neuron(WeightedInput(
            (neuron_in_1, 0.5),
            (neuron_in_2, 0.5),
            (neuron_in_3, 0.5),
        ))
        neuron_sum = Neuron(WeightedInput(
            (neuron_in_1, 1.0),
            (neuron_in_2, 1.0),
            (neuron_in_3, 1.0),
            (neuron_carry, -2.0),
        ))

        neurons = [neuron_in_1, neuron_in_2, neuron_in_3, neuron_sum
    , neuron_carry]

        super().__init__(neurons)
...
```

Listing 6.1: Code for libFHN test `full-adder.py`

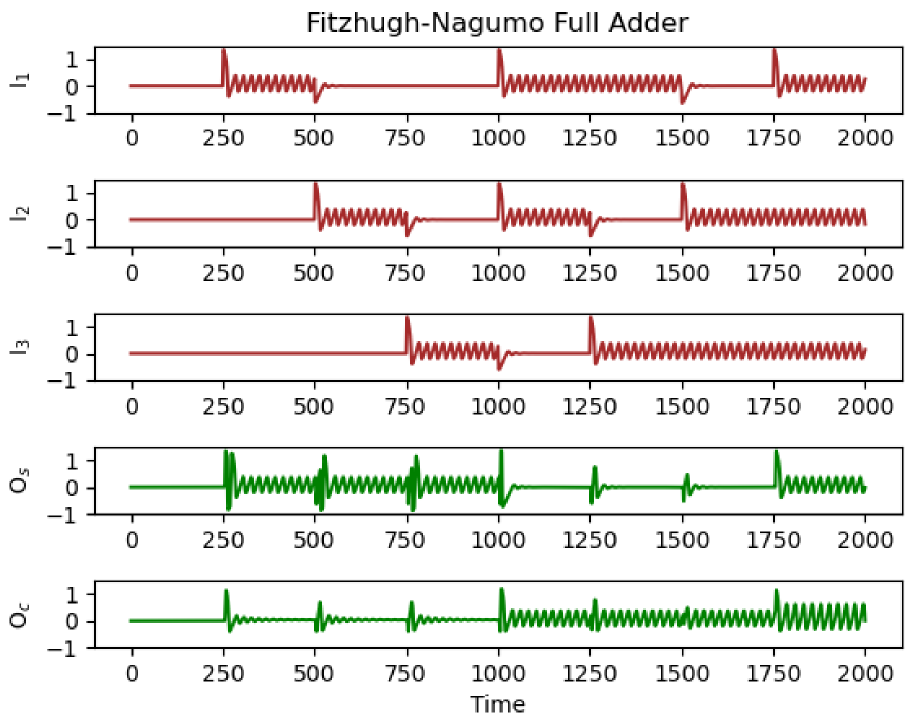


Figure 6.2: The output of `full-adder.py`

Chapter 7

A Generic Adder

Duan et al (2022) conclude that the “MFHN model can be investigated as a potential building block of the large-scale logic circuits”. We do such investigations by creating some more complicated circuits and ALUs in the rest of this paper. These are novel creations, inspired by existing computing, and our aspirations to reduce the complexity of circuitry corresponding to common operations such as addition and multiplication.

We can go further than a full adder and create a generic n-neuron $2^n - 1$-bit full adder. Borresen and Lynch (2012) postulate the existence of such an adder, and here we produce an algorithm for creating it.

This gives us an exponential increase in computation power (taking in $2^n - 1$ inputs) with only a linear increase in the number of components required (n neurons) (Borresen and Lynch, 2012).

We create a new algorithm for constructing a set of weights for the excitatory and inhibitory connections between the neurons within the adder.

7.0.1 The Algorithm

We state the set of weights used in the adder, and then prove our claims by induction on n, the number of neurons in the adder.

We index our neurons N_i over $i \in \{0, 1, \ldots, n-1\}$. Neuron N_i corresponds to the i-th output bit, of decimal value 2^i.

We take our neurons to be calibrated with unit activation threshold, and unit output.

For each neuron N_i connect all $2^n - 1$ inputs to N_i, each with excitatory connections of weight of $\frac{1}{2^i}$. (1)

For each neuron N_i connect the output of N_i to each neuron N_j for $j \in \{0, 1, \ldots, i-1\}$, each with inhibitory weight of 2^{i-j}. (2)

Proof

The case of $n = 2$, which we take to be our base case, is indeed the full adder which we covered in the previous chapter.

Now assume our claim is true for $n = k$. We will prove that our claim s true for $n = k + 1$. We split our proof down into cases for m, the number of active inputs. $m \in \{0, 1, \ldots, 2^{k+1} - 1\}$.

A k-neuron adder is embedded within the $k + 1$-neuron adder as the k neurons corresponding to the lowest k output bits. Hence for $m \in \{0, 1, \ldots, 2^k - 1\}$ our claim is true by the $n = k$ case, since the $k+1$-th neuron does not reach its activation threshold and is unused.

In the case $m = 2^k$, the N_k reaches its activation threshold. Each neuron N_j for $j \in \{0, 1, \ldots, k - 1\}$ has a weighted input of 2^{k-j} via (1). N_k inhibits N_j with a weight of 2^{k-j} via (2). Hence the effective input into N_j is zero, thus we do not need to consider any other inhibitory connections among the first k neurons. Overall this yields N_k alone on, corresponding to an output value of 2^k as expected. When the number of inputs into our adder is an exact power of two corresponding highest neuron 'resets' the lower neurons.

Finally we consider the case $m = 2^k + m'$ with $m' \in \{0, 1, \ldots, 2^k - 1\}$. The 'reset' performed by N_k reduces this case to the k-th output bit being on, plus the output from the first k neurons under m' active inputs, which is also m' from our assumption of $n = k$ yielding the correct result. Hence the overall output is $2^k + m'$ and we are done.

7.0.2 $2^n - n$ Input K-Bit SIMD Adder

As with our 2 neuron full-adder, we wish to chain our generic bit adders together in order to construct an ALU which takes in fixed size K-bit binary operands, say 64 bit numbers. We are able to do this by considering the lowest output bit of an n-neuron adder to correspond to that adder, with decimal value 2^n, and the other $n - 1$ output

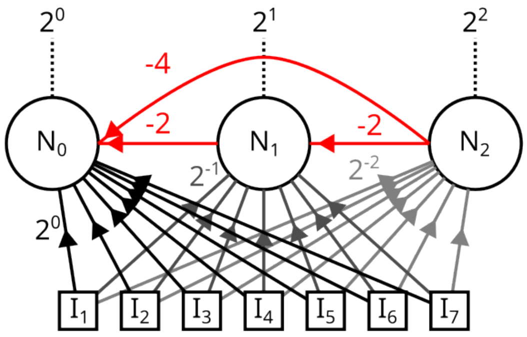


Figure 7.1: Neuron diagram of an example 3-neuron 7-bit adder

bits being carry bits for higher output bits. This generalises a bit full-adder based ripple-carry addition ALU.

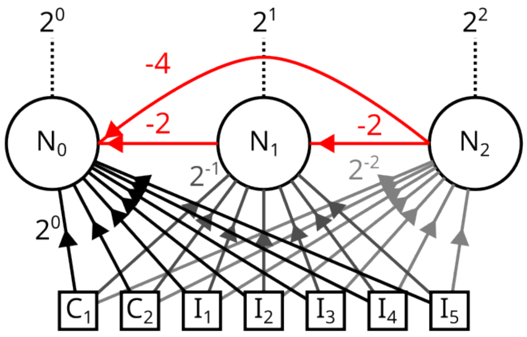


Figure 7.2: The 2 carry bits highlighted for a 3-neuron adder

Using n-neuron adders, each adder takes in $2^n - 1$ inputs, however $n - 1$ inputs are carry bit inputs as shown above. This yields $2^n - n$ 'real' inputs. The carry bit(s) of the highest adder in the ripple-carry addition ALU can be used by a CPU to detect overflow and set a CPU flag accordingly.

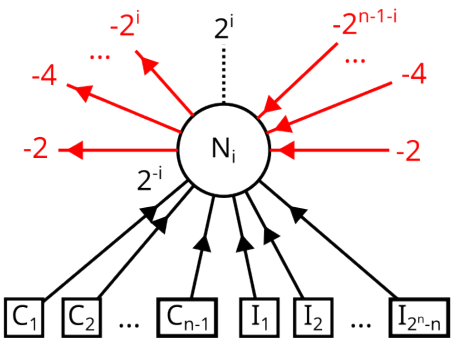


Figure 7.3: A typical neuron in our generic adder

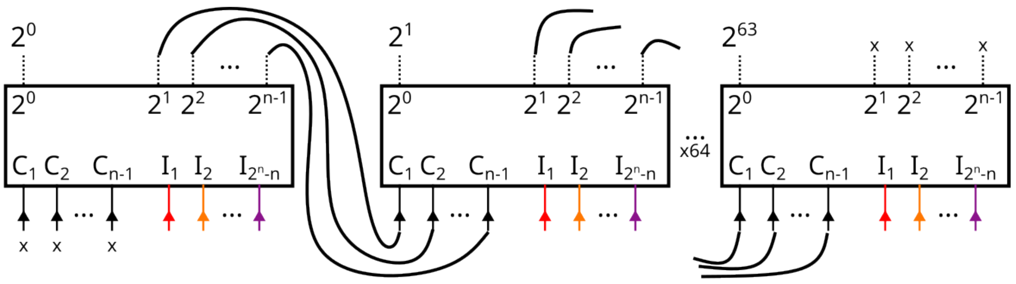


Figure 7.4: An example of 64 n-neuron adders chained together to allow $2^n - n$ 64-bit numbers to be summed

```
...

Sum of [
    (1, 1, 0, 0, 0, 0, 0, 0),
    (1, 1, 0, 1, 0, 0, 0, 0),
    (1, 0, 0, 0, 0, 0, 0, 0),
    (1, 0, 0, 1, 0, 0, 0, 0),
    (1, 0, 0, 0, 0, 0, 0, 0)
] = (1, 0, 0, 1, 1, 0, 0, 0) = 25

...
```

Listing 7.1: Output excerpt for `libThreholdLogic/generic-adder.py`

7.0.3 Complexity Comparison

To create a 2^n bit adder in transistors would require $\mathcal{O}(2^n)$ transistors, however our n neuron constructions can add $\mathcal{O}(2^n)$ bits; we have a logarithmic reduction in complexity, which is a very nontrivial feat to achieve.

7.0.4 Remarks

One may observe that the work we have achieved in this chapter is independent of the technology of Josephson Junction neurons or Fitzhugh Nagumo neurons, and is purely threshold-logic theoretic. This allows our algorithm to be transferred to other constructions which use threshold logic.

Part III

Advanced Threshold Logic - Transistor Logic Interop

Chapter 8

Transistor Logic Interoperability

Our work with threshold logic so far has been plentiful. We have demonstrated ways to reduce both time and spacial complexity of some example ALUs by switching to threshold logic from digital logic. Furthermore, the existence of Josephson Junction technology makes these advances feasible in the real world.

However, a logistical challenge one would face in implementing Josephson Junction technology into computers is interfacing with existing technology. Most, if not all, computing nowadays uses digital logic, typically transistor based logic circuitry. To avoid reinventing the wheel, (the entirety of modern computing), it is necessary to be able to connect together threshold logic-based computing with binary computing for arbitrary tasks.

In the previous chapters we have devised specialised ALUs, such as our SIMD adder. The derivation of the adder took careful strategising, and the algorithm for construction was specialised to that particular task. In the end we ended up with an ALU which takes binary input and gives binary output, but the method for devising the ALU was very specific.

We therefore consider two approaches for integrating JJ ALUs into existing computer architecture.

The first is to keep devising special purpose JJ binary ALUs as we have done so far and integrate them into hardware such as CISC CPUs and GPUs, and allow compiler technology to optimise for targeted hardware. In this way as generations of hardware come to pass, for example Intel creating new ‘generation’s of CPUs roughly annually, JJ ALUs can be integrated as they are discovered / invented / optimised, and existing CPU instructions can be microcoded to use the new additions, deprecating existing transistor circuitry. This is a Ship of Theseus approach for the gentle integration of our new technology. This avoids having to spend years, if not decades, creating from scratch a new CPU architecture, instruction set, standards, and specifications, which would take decades to compete with the titans of x86 and ARM (and possibly RISC-V).

The first approach presented aims to wrap JJN OTL technology around the digital / transistor world. Our second proposed approach takes the converse direction of aiming to embed digital logic within OTL, with OTL being a superset of compatibility. If we could create digital logic gates, such as AND, OR, NOT, XOR, which form the basis of any digital logic circuit, out of threshold-logic gates we would be in luck. This would allow us to port existing transistor based circuitry directly to JJN circuitry in a 1-to-1 fashion. With the exception of chips which require very precise timing constraints, silicon chips could be ported over effective immediately to superconducting equivalent JJN prototypes. Our additional SIMD ALUs and friends, which take advantage of the high connectivity of JJN networks could also be implemented within these from the start. In effect this second approach encompasses the first approach, and allows for optimisation of digital circuitry into more densely connected networks, possibly with lower component counts, as time progresses, as and when optimisations are found.

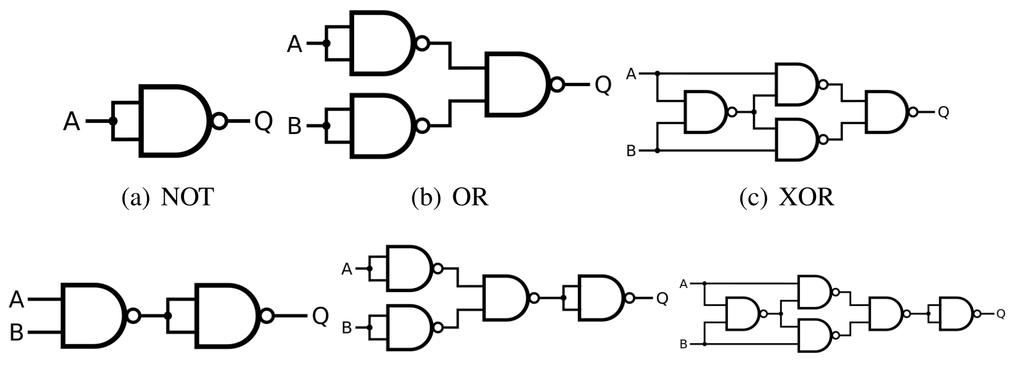
Pursuing the goal of the second approach, we thus proceed with the task of creating arbitrary logic gates from threshold logic neurons.

Firstly we acknowledge that the NAND gate of two inputs is a 'universal logic gate', that is NAND gates can be used in combinations to express any possible boolean truth table. This is also known as 'functional completeness'. Therefore if we are able to create a threshold logic gate which replicates a digital NAND gate, we know we can create anything we desire. Truth tables for base gates are given below.

A	B	NAND	AND	OR	NOR	XOR	XNOR	NOT(A)
0	0	1	0	0	1	0	1	1
0	1	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	0
1	1	0	1	1	0	0	1	0

Table 8.1: Digital logic gates truth tables

Constructions of base logic gates from NAND gates are also given on page 32. Public



(a) NOT (b) OR (c) XOR

(d) AND (e) NOR (f) XNOR

Figure 8.1: Common logic gates created from NAND gates

domain images credit of Wikipedia user ‘InductiveLoad’.

We successfully create a NAND gate using a single neuron as shown on page 32.

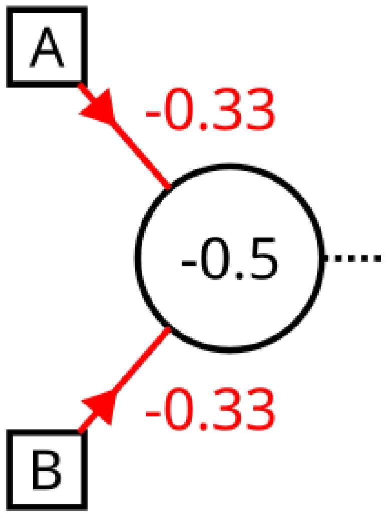
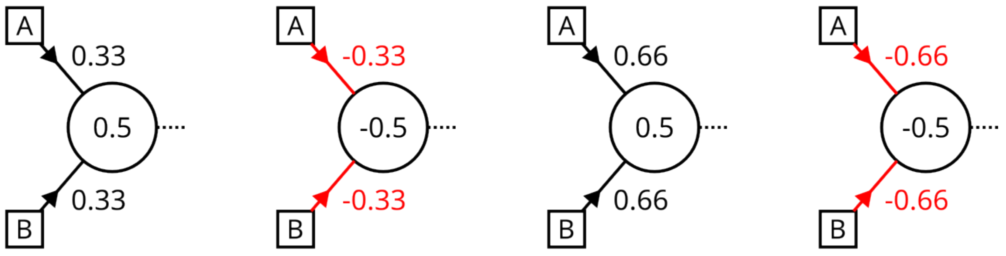


Figure 8.2: Threshold NAND Gate

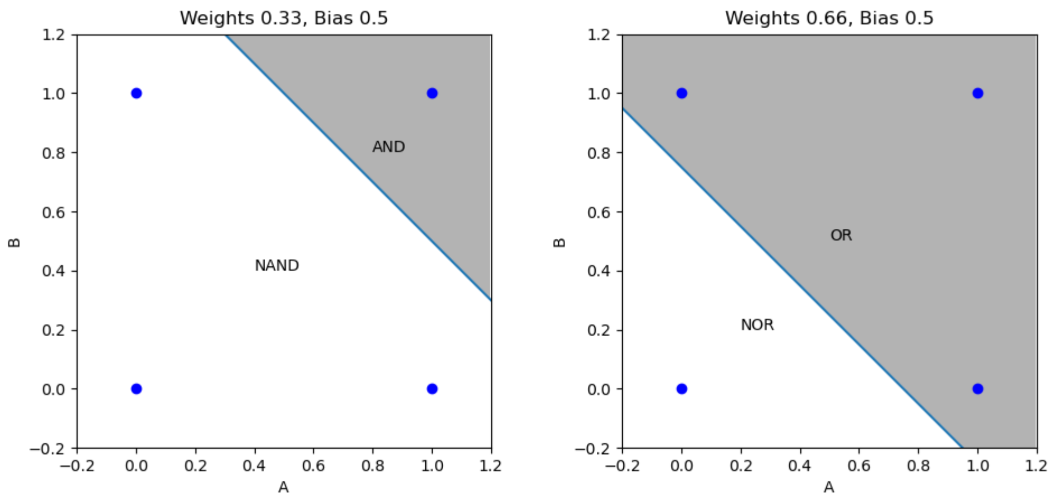
Most of the constructions of primitive logic gates using only NAND gates require several gates, and hence several neurons, however one may aspire to optimise this. Indeed we can optimise all of AND, OR, and NOR down to a single neuron! Through careful thinking we derive the weights and biases for the gates seen on page 33.

The pairing of AND with NAND, and OR with NOR as negations of each other is reflected in the negations of the neurons’ weights and bias. The construction of these four gates out of single neurons is a consequence of the linear-separability of the



(a) AND (b) NAND (c) OR (d) NOR

Figure 8.3: Common logic gates created from single neurons



(a) AND (b) OR

Figure 8.4: Linear separability of the plane for AND and OR gates

required inputs: for example the AND inequality $0.33x + 0.33y > 0.5$ divides the plane in two, and the NAND inequality $-0.33x - 0.33y > -0.5$ can be rearranged to $0.33x + 0.33y > 0.5$, the other side of the plane. This is shown in the following images.

The XOR and XNOR gates however are known to be not representable with a single neuron (also known as a perceptron) in a similar way (Minsky and Papert, 1969). A common way to create and XOR gate is as (NAND(A,B) AND OR(A,B)).

Porting this particular construction from digital logic to threshold logic directly would require three neurons. If we cast our minds back to Section II of this paper, we recall constructing a 2-bit half-adder, which contains within it an XOR gate, only requiring

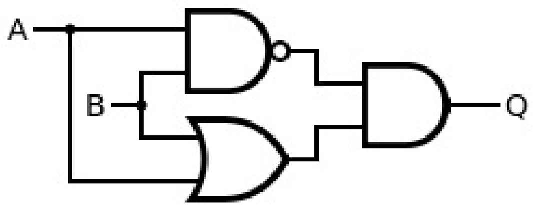
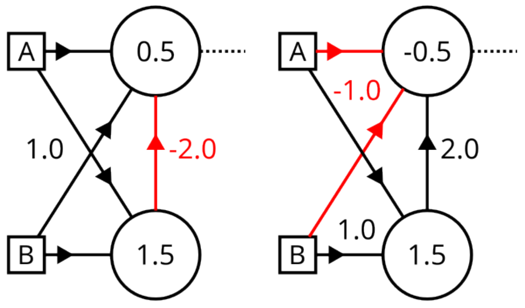


Figure 8.5: XOR gate constructed from NAND(A,B) AND OR(A,B)



(a) XOR (b) XNOR

Figure 8.6: XOR and XNOR gates created from two neurons

two neurons. This confirms again that it pays off to reconstruct gates using threshold logic / neurons directly since the component count can often be reduced due to the high-connectivity between the neurons in a gate. XOR and XNOR are shown in figure 8.6.

Why not transistor based threshold logic

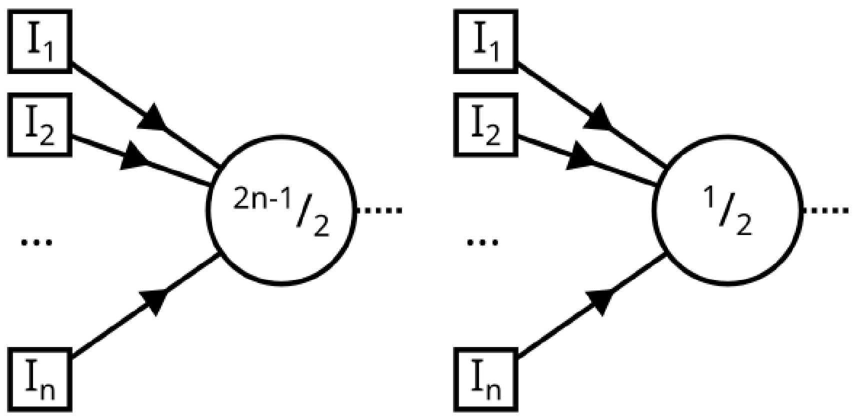
As a sanity check, one might ask ‘why not try to produce threshold logic using transistors?’ The answer it that for low currents the transistor’s output ‘forward voltage’ is proportional to the logarithm of the base current (Middleton and Valkenburg, 2002). Contrast this to the discreet bifurcation behaviour of threshold effect oscillators.

Chapter 9

Basic Multi-Input Gates

Our two-input gates constructed in the previous chapter successfully establish a compatibility layer from digital logic to threshold logic. Inspecting our formulae for the linear gates, we create extensible n-bit input versions of them, and compare the complexities of their digital equivalents.

Our formulae for AND and OR are of the form $\Sigma x_i - T > 0$, that is a single neuron with each input having uniform weight 1, and bias T, using threshold function H_0. For an AND gate T can be chosen as any value $n - 1 < T < n$, and for an OR gate $0 < T < 1$. The arithmetic means $\frac{2n-1}{2}$ and $\frac{1}{2}$ respectively are sensible choices.



(a) AND (b) OR

Figure 9.1: Multi-input AND and OR gates

For an intuition, one can take the plane-splitting images from earlier in the previous chapter and try generalising them to n dimensions; we are 'shaving off' either the $(0, 0, ..., 0)$ or $(1, 1, ..., 1)$ corner of an n dimensional unit hypercube with the plane $1 \cdot x - T = 0$ where 1 is a vector of all ones. This may be easiest to imagine in 2 and 3 dimensions.

The NAND and NOR flavours of gates clearly follow from reversing the plane inequality to $\Sigma x_i - T < 0$, which yields negated weights and biases.

Gate	Category	Digital	Threshold
AND	Component count	$\mathcal{O}(n)$; n transistors in series, or arranged in a tree for compactness	1 neuron; $\mathcal{O}(c)$
	Time	$\mathcal{O}(\log(n))$ in the case of a tree, propagation time	$\mathcal{O}(c)$
OR	Component count	$\mathcal{O}(n)$; n parallel transistors	1 neuron; $\mathcal{O}(c)$
	Time	$\mathcal{O}(c)$	$\mathcal{O}(c)$

Table 9.1: Digital-vs-Threshold complexity comparison

Threshold logic's use of input current being meaningful beyond being a binary 'on' or 'off' leads to orders of magnitude of reduction in complexities.

Chapter 10

Complex Multi-Input Gates

We previously mentioned ‘shaving off’ the $\vec{0}$ or $\vec{1}$ corner of an n-dimensional unit hypercube, partitioning the vector space with a plane, to obtain our multi-input AND and OR gates. What if we were to go even further to partition an arbitrary corner from the rest?

Doing so we would create two families of gates:

- Generalised AND (GAND): Permit as input only a single combination of 1s and 0s
- Generalised NAND (GNAND): Permit as input all combinations of 1s and 0s except for one

The purpose of these gates is to avoid the need of any inverter neurons (NOT gates) in our networks, which can simplify component counts.

These families highlight that AND and NOR are related, both being members of the GAND family, and NAND and OR are related, members of the GNAND family.

We proceed with the mathematics, which we have derived ourselves. We calculate the general form of our ’threshold plane’. The vector calculations are easiest when considering a hypercube of the form $\{-1, 1\}^n$. We will adjust to for in the hypercube $\{0, 1\}^n$ at the end.

We wish to separate a corner $\vec{v} \in \{-1, 1\}$ via a plane of the form $\vec{x} \cdot \vec{n} - b = 0$. We choose $\vec{n}$ to be $\vec{v}$ up to normalisation. The following inequalities must hold

$$\vec{x}\cdot\vec{v}-b>0 \quad \text{for } \vec{x}=\vec{v} \tag{10.1}$$
$$\vec{x}\cdot\vec{v}-b<0 \quad \text{for } \vec{x}\neq\vec{v} \tag{10.2}$$

From 10.1 we obtain $n-b>0$. In 10.2 we notice that $\vec{x}\cdot\vec{v}$ is the number of bits that are the same in $\vec{x}$ and $\vec{v}$, minus the number of bits that differ. This is maximised as $n-2$ when $\vec{x}$ and $\vec{v}$ differ by a single bit. Hence $n-2-b<0$. Putting both inequalities together we obtain $n>b>n-2$. A sufficient choice for b is $b=n-1$.

To scale a point in $\{0,1\}^n$ into $\{-1,1\}^n$ we use the function $f(\vec{x})=2\vec{x}-\vec{1}$. Our plane formula becomes

$$(2\vec{x}-\vec{1})\cdot(2\vec{v}-\vec{1})=n-1 \tag{10.3}$$
$$4\vec{x}\cdot\vec{v}-2\vec{1}\cdot\vec{v}-2\vec{x}\cdot\vec{1}+\vec{1}\cdot\vec{1}=n-1 \tag{10.4}$$
$$\vec{x}\cdot(4\vec{v}-2\vec{1})-2\vec{1}\cdot\vec{v}+n=n-1 \tag{10.5}$$

$\vec{1}\cdot\vec{v}$ corresponds to the number of non-zero entries of $\vec{v}$, which we will denote as k. This is the number of excitatory connections into the gate's neuron. Thus

$$\vec{x}\cdot(4\vec{v}-2\vec{1})-2k=-1 \tag{10.6}$$
$$\vec{x}\cdot(2\vec{v}-\vec{1})=\frac{2k-1}{2} \tag{10.7}$$

This is the form in which we keep our plane. The weights of the neuron in a GAND gate are therefore given as the tuple $2\vec{v}-\vec{1}$, with bias $\frac{2k-1}{2}$. This format agrees with the n-input gates from the previous chapter. Similarly the GNAND gates's weights and bias are $\vec{1}-2\vec{v}$ and $-\frac{2k-1}{2}$.

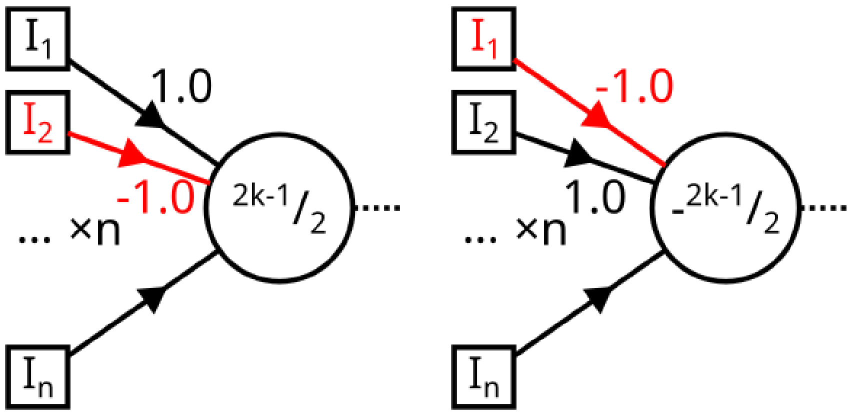


Figure 10.1: GAND and GNAND gates

Python Output

We test our gates in Python with the script `gand-and-gnand.py`:

```python
...

class GAND(Gate):
    def __init__(self, positives_map: Tuple[int]) -> None:
        self.neuron = Neuron(
            tuple((
                1.0 if positive else -1.0
                for positive in positives_map
            )),
            (2 * sum(positives_map) - 1) / 2,
            Heaviside(0)
        )

...
```

Listing 10.1: Code for libThresholdLogic.Gates::GAND

```
GAND of (1, 1, 0, 1)
(0, 0, 0, 0) 0
(0, 0, 0, 1) 0
(0, 0, 1, 0) 0
(0, 0, 1, 1) 0
(0, 1, 0, 0) 0
(0, 1, 0, 1) 0
(0, 1, 1, 0) 0
(0, 1, 1, 1) 0
(1, 0, 0, 0) 0
(1, 0, 0, 1) 0
(1, 0, 1, 0) 0
(1, 0, 1, 1) 0
(1, 1, 0, 0) 0
(1, 1, 0, 1) 1
(1, 1, 1, 0) 0
(1, 1, 1, 1) 0
```

Listing 10.2: Extract from the output of `gand-and-gnand.py`

```
GNAND of (1, 1, 0, 1)
(0, 0, 0, 0) 1
(0, 0, 0, 1) 1
(0, 0, 1, 0) 1
(0, 0, 1, 1) 1
(0, 1, 0, 0) 1
(0, 1, 0, 1) 1
(0, 1, 1, 0) 1
(0, 1, 1, 1) 1
(1, 0, 0, 0) 1
(1, 0, 0, 1) 1
(1, 0, 1, 0) 1
(1, 0, 1, 1) 1
(1, 1, 0, 0) 1
(1, 1, 0, 1) 0
(1, 1, 1, 0) 1
(1, 1, 1, 1) 1
```

Listing 10.3: Output (cont.)

This is a great feat to achieve with just a single neuron gate, and equips us well for the next chapters.

Chapter 11

Karnaugh Maps

As a prerequisite for the 2x2 bit multiplier circuitry which we put together in the next section, we first investigate the usefulness of Karnaugh Maps for simplifying boolean expressions.

A Karnaugh Map is constructed by mapping the truth table of a function

$$f(X_0, \dots, X_{n-1}) \to \{0, 1\}$$

to a $(w := \lceil \frac{n}{2} \rceil) \times (h := \lfloor \frac{n}{2} \rfloor)$ grid in a particular way; the rows and columns are enumerated by Gray Code ordinals, $g_n, n \in \mathbb{N}_0$. The entry in the Karnaugh Map at (x, y) is the truth table entry for $(X_0, \dots, X_w)$ having inputs equal to the binary bits of g_x and $(X_{w+1}, \dots, X_{n-1})$ having inputs equal to the binary bits of g_y. We will present an example soon, after we have first described Gray Codes.

11.1 Gray codes

Frank Gray was a 20th century Physicist, another scientist at Bell Labs. Gray (1953) patented a method of enumerating the first 2^N members of $\mathbb{N}_0$ for $N \geq 1$ in such a way that any two successive terms differ by only a single binary bit. As a bonus g_0 and g_{2^N-1} also only differ by a single bit (the property holds over the modulo boundary). A simple way to create the Gray code is recursively on N; we prove the existence of a code inductively.

Construction

For $N = 1$ define $g_0 = 0b0, g_1 = 0b1$. Our 1-bit-difference property holds.

Assuming there exists a Gray code g_n^N for $N = K$, extend the code in the following way:

$$
\begin{aligned}
g_n^{K+1} &= 0b0g_n^K && \text{for} \quad 0 \le n < 2^K && (11.1)\\
g_n^{K+1} &= 0b1g_{2^K-1-(n-2^K)}^K && \text{for} \quad 2^K \le n < 2^{K+1} && (11.2)\\
&= 0b1g_{2^{K+1}-1-n}^K && && (11.3)
\end{aligned}
$$

ie take a copy of g_n^K with a binary prefix of 0, and a mirror copy of g_n^K with a binary prefix of 1.

Within each of the two halves we have created, successive terms differ by only a single bit, since their new prefix is constant, and the last K digits are determined by g^K, which has the 1-bit-difference property by induction. At the boundary between the two halves, aka the ‘mirror line’, only the new prefix changes from 0 to 1, and likewise between $g_{2^{K+1}-1}^{K+1}$ and g_0^{K+1}. Hence the property holds, and our code is established.

Since the first half of g^{K+1} is created by adding the binary prefix 0 to g^K, their numerical values coincide, and the notation of g_n^N is superfluous; we can just use g_n.

Sample output of our Python function for generating Gray numbers is given here, testing the script `gray-example.py`:

```python
...

def gray_code(n: int) -> List[Tuple[int]]:
    """Returns a list of the first 2^N Gray numbers as big-bittian
    bit tuples"""
    ret = [(0,), (1,)]

    for _ in range(n - 1):
        ret = [(0, *row) for row in ret] +
              [(1, *row) for row in ret[::-1]]

    return ret

def gray_numbers(n: int) -> List[int]:
    """Returns a list of the first 2^N Gray numbers"""
    return [
        bit_tuple_be_to_int(row)
        for row in gray_code(n)
    ]

...

i_to_gray = gray_numbers(3)

for i, gray_i in enumerate(i_to_gray):
    print(f"gray({i}) = {gray_i} = {gray_i:03b}")
```

Listing 11.1: Code excerpt for `gray-example.py`

```
gray(0) = 0 = 000
gray(1) = 1 = 001
gray(2) = 3 = 011
gray(3) = 2 = 010
gray(4) = 6 = 110
gray(5) = 7 = 111
gray(6) = 5 = 101
gray(7) = 4 = 100
```

Listing 11.2: Output of `gray-example.py`

11.2 Usage of Karnaugh maps

We now come to describe the usefulness of Karnaugh maps, especially in the case of a 4-input truth table, which we will come across in the next chapter. As a preview, we consider the truth table of the ones digit of 2-bit multiplication, that is $y \times x \bmod 2$ where $y = 2Y_1 + Y_0$, $x = 2X_1 + X_0$.

Y_1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Y_0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
X_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
X_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Z	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1

Table 11.1: Truth table of the 1s bit of 2-bit multiplication

If we were to create an alternative table with regular indexing of (Y_1, Y_0) as its rows, and (X_1, X_0) as its columns, we would create the following:

$y \backslash x$	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	0	0
3	0	1	0	1

Table 11.2: Alternative table of the 1s bit of 2-bit multiplication

The result is a table with a topologically periodic 2x2 entry, which corresponds to the result of $y \times x$ mod 2 being equal to y mod 2 $\times$ x mod 2. A naive attempt at writing out the disjunctive normal form of this table is

$$X_0X_1'Y_0Y_1' + X_0X_1Y_0Y_1' + X_0X_1'Y_0Y_1 + X_0X_1Y_0Y_1 \tag{11.4}$$

which one may not immediately see how to simplify. Simplification of boolean expressions is not always trivial, especially in other cases where there is not a strong mathematical reasoning behind the truth table entries.

On the contrary if we were to create a table with rows and columns enumerated by Gray ordinals, a Karnaugh map, things will become clearer.

$y \backslash x$	0	1	3	2
0	0	0	0	0
1	0	1	1	0
3	0	1	1	0
2	0	0	0	0

Table 11.3: Karnaugh map of the 1s bit of 2-bit multiplication

Our Python test script `bit-multiplier.py` has red-green colour output to make things even clearer: one notices a 2x2 cluster in the Karnaugh map. These clusters correspond to a group of 'minterms' (1x1 entries, conjunctions of all inputs or their complements), which can be combined due to having common factors. The Gray code indexing ensures that any two minterms differ by a single factor, a primitive and its complement, which can be combined.

The first non-zero row's minterms combine to

$$X_0X_1'Y_0Y_1' + X_0X_1Y_0Y_1' = \tag{11.5}$$

$$(X_0Y_0Y_1')(X_1' + X_1) = X_0Y_0Y_1' \tag{11.6}$$

and the second non-zero row's minterms combine to

$$X_0X_1'Y_0Y_1 + X_0X_1Y_0Y_1 = \tag{11.7}$$
$$(X_0Y_0Y_1)(X_1' + X_1) = X_0Y_0Y_1 \tag{11.8}$$

which both combine to

$$X_0Y_0Y_1' + X_0Y_0Y_1 = \tag{11.9}$$
$$X_0Y_0(Y_1' + Y_1) = \underline{\underline{X_0Y_0}} \tag{11.10}$$

greatly simplifying the truth table's expression. The result is indeed consistent with our mathematical intuition that the last digit of binary multiplication is determined by the last digits of the operands.

4x4 Karnaugh maps allow any $2^n \times 2^m$ rectangle of minterms to be simplified in a similar way. The four possible expressions for rectangles of width 2, spanning columns (0,1), (1,3), (3,2), (2,0) correspond to $\neg X_1$, X_0, X_1, $\neg X_0$ respectively. ((2,0) is possible due to the toroidal topology of Karnaugh maps, which follows from g_0 and g_{2^N-1} differing by a single bit as established earlier). For a width 4 rectangle the expression is always 1. Hence any $2^n \times 2^m$ rectangle can be expressed as a conjunction of primitives or their negations, which we can express with a single GAND gate neuron we created in the previous chapter. Thus the entire truth table can be written as a simple disjunction of conjunctions, similar to disjunctive normal form.

With our knowledge of how to simplify down truth table boolean expressions, and hence the number of neurons needed for a threshold logic gate equivalent, we can tackle the 2x2 bit multiplier.

Chapter 12

2×2 Bit Multiplier Simplifications

Borresen and Lynch (2012) demonstrate an implementation of a 2x2 bit multiplier using threshold logic with Fitzhugh Nagumo oscillators to mimic conventional digital circuitry. This approach uses eight neurons, however with our optimisations from Section III we can bring this down to just six!

Figure 12.1 shows the typical implementation of a 2x2 bit multiplier. It uses four AND gates (1 neuron each), and two half-adders (2 neurons each). Observe the amount of redundancy in the last half-adder's logic, in both the XOR output to the 4s bit, and the AND output to the 8s bit.

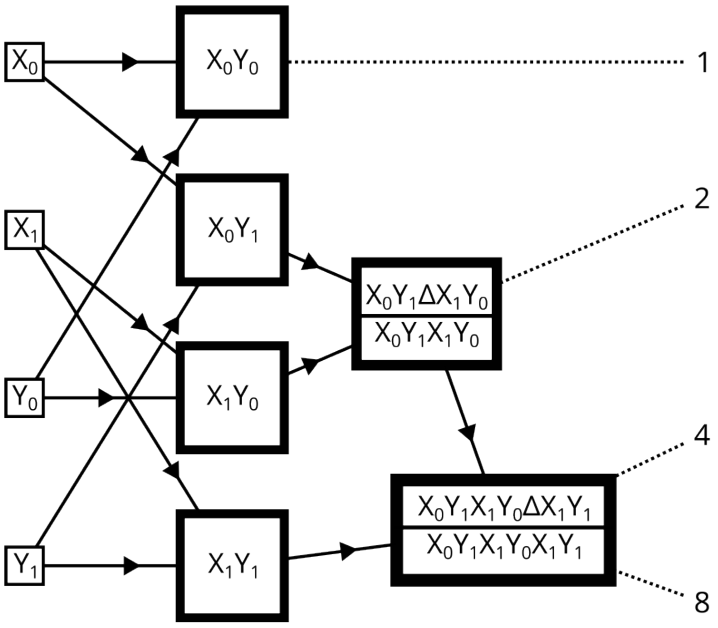


Figure 12.1: The original 2x2 Multiplier logic

To begin our attempts to optimise this circuit, we obtain the truth table for the multiplier's output bits, and plot Karnaugh maps for each of them. We do both of these programmatically with Python, in the script `bit-multiplier.py`

	Y_1	Y_0	X_1	X_0	Z_3	Z_2	Z_1	Z_0
$0 \times 0 = 0$	0	0	0	0	0	0	0	0
$0 \times 1 = 0$	0	0	0	1	0	0	0	0
$0 \times 2 = 0$	0	0	1	0	0	0	0	0
$0 \times 3 = 0$	0	0	1	1	0	0	0	0
$1 \times 0 = 0$	0	1	0	0	0	0	0	0
$1 \times 1 = 1$	0	1	0	1	0	0	0	1
$1 \times 2 = 2$	0	1	1	0	0	0	1	0
$1 \times 3 = 3$	0	1	1	1	0	0	1	1
$2 \times 0 = 0$	1	0	0	0	0	0	0	0
$2 \times 1 = 2$	1	0	0	1	0	0	1	0
$2 \times 2 = 4$	1	0	1	0	0	1	0	0
$2 \times 3 = 6$	1	0	1	1	0	1	1	0
$3 \times 0 = 0$	1	1	0	0	0	0	0	0
$3 \times 1 = 3$	1	1	0	1	0	0	1	1
$3 \times 2 = 6$	1	1	1	0	0	1	1	0
$3 \times 3 = 9$	1	1	1	1	1	0	0	1

Table 12.1: 2x2 bit multiplication truth table, with Z_i corresponding to the output bit of value 2^i

$y \backslash x$	0	1	3	2
0	🟥 0	🟥 0	🟥 0	🟥 0
1	🟥 0	🟩 1	🟩 1	🟥 0
3	🟥 0	🟩 1	🟩 1	🟥 0
2	🟥 0	🟥 0	🟥 0	🟥 0

Table 12.2: $Z_0 \quad (2^0 = 1)$

x y	0	1	3	2
0	0	0	0	0
1	0	0	1	1
3	0	1	0	1
2	0	1	1	0

Table 12.3: $Z_1 \quad (2^1 = 2)$

x y	0	1	3	2
0	0	0	0	0
1	0	0	0	0
3	0	0	0	1
2	0	0	1	1

Table 12.4: $Z_2 \quad (2^2 = 4)$

x y	0	1	3	2
0	0	0	0	0
1	0	0	0	0
3	0	0	1	0
2	0	0	0	0

Table 12.5: $Z_3 \quad (2^3 = 8)$

A sensible boolean expression for Z_0 is X_0Y_0. One could also consider tackling Z_3 next as $X_0Y_0X_1Y_1$, since its Karnaugh map can be written as an expression of a single minterm, however we cleverly inspect the other Karnaugh maps first.

Z_1's map is interesting. At first we may consider a disjunction of four conjunctions, namely $Y_0Y_1'X_1 + X_0X_1'Y_1 + X_0'X_1Y_0Y_1 + X_0X_1Y_0'Y_1$, which would require five neurons, however we can cleverly reduce this to just four neurons using an XOR expression $X_0Y_1 \Delta X_1Y_0$. Our XOR gate also gives us an AND gate for free, yielding $X_0Y_1X_1Y_0$, which corresponds to Z_3.

Finally Z_2 can be expressed with a single GAND gate of three inputs, $X_1Y_1(X_0Y_0)'$. We already have the X_0Y_0 term from Z_0's output. This is also an applicable demonstration of our n-input GAND gate creations, showing how a single neuron can simplify extensive transistor based logic gates.

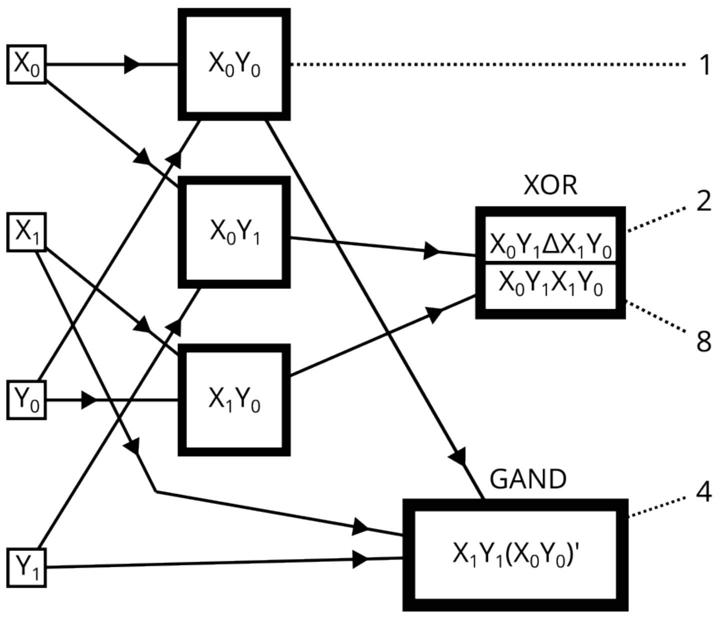


Figure 12.2: Our improved 2x2 multiplier, using six neurons

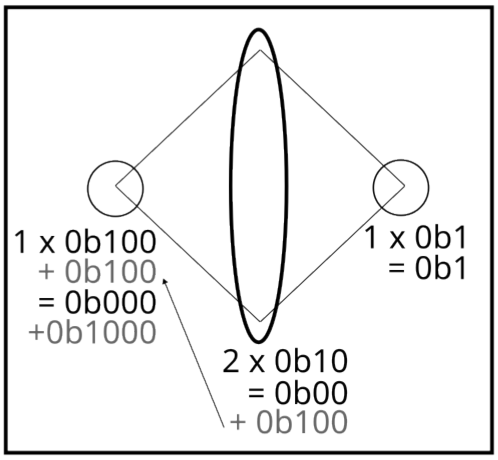
This method of simplification required careful analysis by a human to create it. It remains an open question for further work whether arbitrary circuitry can be simplified in an automated way. The simplifications were a combination of Karnaugh map analysis, which works well for a small multiplier of size 2x2, and employing n-input gates for $n > 2$, an enhancement of threshold logic over transistor based logic.

Chapter 13

$n \times n$ Bit Multiplier

Our final ALU with threshold logic is a generic $n \times n$ bit multiplier. This employs multiple of our generic adders, another good example of utilising threshold logic, and demonstrating scalability.

We wish to multiply the n-bit numbers x and y. We start off by considering the output $z := x \times y$ bit by bit. The value of bit k, denoted as z_k, will be determined by the value of $\Sigma_{i=0}^{k} x_i \times y_{k-i}$ plus any carry bits. This convolution of x and y is best seen with a binary version of Japanese multiplication.



0b11 x 0b11 = 0b1001

Figure 13.1: Binary Japanese Multiplication for the equation 3×3 showing the convolution of powers. Powers are smallest-to-largest right-to-left

13.1 An algorithm for the number of neurons for bit z_k of $x \times y$

We calculate how many neurons N_k we need in the generic adder for bit z_k. Without considering carry bits, the number of 1 terms possible in the sum contributing to z_k from $\Sigma_{i=0}^{k} x_i \times y_{k-i}$ is l_k where $l := [1, 2, \ldots, n-1, n, n-1, n-2, \ldots, 1]$.

We add to l progressively, looping over the number of elements of l. For each element l_i the number of carry bits / carry neurons out of z_i's adder is $\lfloor \log_2(l_i) \rfloor$. Thus we add 1 to each of the next $\lfloor \log_2(l_i) \rfloor$ elements of l, appending to the end of l if necessary.

Finally we work out how many neurons we need for l_k. An N-bit adder can support $l_k \leq 2^{N_k} - 1$ inputs, hence we need $N_k \geq \log_2 l_k + 1$. $N_k = \lceil \log_2(l_k + 1) \rceil$ suffices.

13.2 Example

Find the number of neurons for each bit of 3×3 multiplication:

Start with $l = [1, 2, 3, 2, 1]$. Adding on the carry bits leads to

$$\begin{aligned}
&[1, 2, 3, 2, 1] \\
\lfloor \log_2(l_0 = 1) \rfloor = 0 &\rightarrow [1, 2, 3, 2, 1] \\
\lfloor \log_2(l_1 = 2) \rfloor = 1 &\rightarrow [1, 2, 4, 2, 1] \\
\lfloor \log_2(l_2 = 4) \rfloor = 2 &\rightarrow [1, 2, 4, 3, 2] \\
\lfloor \log_2(l_3 = 3) \rfloor = 1 &\rightarrow [1, 2, 4, 3, 3] \\
\lfloor \log_2(l_4 = 3) \rfloor = 1 &\rightarrow [1, 2, 4, 3, 3, 1] \\
\lfloor \log_2(l_5 = 1) \rfloor = 0 &\rightarrow [1, 2, 4, 3, 3, 1]
\end{aligned}$$

Then taking $\lceil \log_2(l + 1) \rceil$ = $\lceil \log_2([2, 3, 5, 4, 4, 2]) \rceil$ yields $[1, 2, 3, 2, 2, 1]$ as the values for N_k.

13.3 Doing the multiplication

Multiplication is thus done via

$$z_k, \{c_{k,i}\}_{0<i<N_k} = \text{ADD}_{N_k}(\{\text{AND}(x_i, y_{k-i})\}_{i=0}^{k} + \{c_{k-i,i}\}_{i=1}^{k}) \tag{13.1}$$

where ADD_{N_k} is our N_k neuron $2^{N_k} - 1$ bit adder and $c_{j,i}$ is the i-th output bit of adder j.

13.4 Testing the multiplier

Our Python test script `generic-multiplier.py` verifies programatically that our codebase file `GenericBitMultiplier.py`'s `GenericBitMultiplier` class gives the correct result for multiplying all 8×8-bit inputs. This of course can be changed to verify any $n \times n$-bit input. We also choose an arbitrary 5×5-bit multiplication operation, 31×17, just to see our multiplier in action, to which our test script correctly outputs

```
31 * 17 = 527
```

Listing 13.1: Output of `generic-multiplier.py`

13.5 Complexity

For multiplying n bit numbers we only require $\mathcal{O}(n^2)$ AND gates from the convolutions, and $\mathcal{O}(\log(n))$ adders of $O(n)$ neurons each, giving a total complexity of $\mathcal{O}(n^2 + n\log(n)) = \mathcal{O}(n^2)$. This is on par with transistor based multiplication, however big-$\mathcal{O}$ notation is not everything; our component count for the z_k additions implements our extensible generic adders, whose complexity is much lower than transistor based addition.

13.6 Future Work

It remains for future work whether arbitrarily scaled ALUs such as n-bit multiplication and n-bit addition are the way forwards, or whether specialised optimisations suitable for the job like in the previous chapter are more efficient, whilst more time consuming to put together.

Part IV

Conclusions

Chapter 14

Summary & Conclusions

We set out on our three pronged attack as laid out in the abstract and we have conquered all fields:

Part I

$>$ reproduced the Fitzhugh Nagumo model neuron threshold effect oscillator

$>$ reproduced the Josephson Junction model neuron threshold effect oscillator

Part II

$>$ Reproduced a FHN half adder successfully

$>$ Reproduced a FHN full adder successfully

$>$ Created a generic n-neuron extensible adder, detailing a novel synthetic threshold logic algorithm corresponding to its creation

$>$ Highlighted the computational potential complexity advantages of threshold logic over digital logic, in particular the reduction in component count e.g. in our n-bit adder

Part III

> established a compatibility layer between digital logic and threshold logic
> demonstrated that all digital logic gates can be constructed from threshold neuron networks
> provided excellent simplifications in common gates (AND, OR, NOT, etc.)
> laid out detailed formulae for creating multi-input versions of common gates with synthetic threshold logic
> created novel generalised single-neuron gates (GAND and GNAND) to remove the need for inverter neurons, with a corresponding mathematical walk-through
> put forwards simplifications to common circuitry such as the 2×2 multiplier, reducing from 8 neurons to 6
> conjectured optimisation techniques by considering the overlap between the Karnaugh maps of output bits of multi-output ALUs
> derived all the mathematical formulae for synthetic threshold logic creations ourselves

Our three Python codebases `libFHN`, `libJJ`, `libThresholdLogic` successfully demonstrate our creations, and provide useful reusable classes for neurons, logic gates, and more. There are provided test scripts. We did not end up needing to use recurrent neural networks, as our initial terms-of-reference set out.

14.1 Future work

It may have been nice to simulate a FHN set-reset latch, however with the plentiful amount of threshold logic creations we have achieved enough in our thesis, and finite timespan, we have done very well. The codebase for `libFHN` ended up very nicely declarative, a style in which `libThresholdLogic` could be rewritten for more extensibility in the future. Investigation of how generic neuron networks hold up under Gaussian noise could also be investigated in the future. Lynch (June 2014) already considers this in detail for the set-reset flip-flop, and therefore we prioritise creating our novel ALUs over reproducing existing work, in an already extensive thesis.

14.2 Self Reflection

I think this project has gone very well, with Parts I, II, and III producing increasingly more fruitful results. The work started out with reproducing existing results, and then flourished into useful creations, which naturally flowed one to the next. I have gained a deeper appreciation for the time it takes for research to become instantiated in the real world. Reading through papers was time consuming, but gave one a richer environment in which to take inspiration for our work. It should be clear from our 'Part' structure that we have allocated time in corresponding chunks to give a report that covers all parts with respectful weighting. Our codebase is large, bloating out the appendix of the thesis, however all code was useful and produced results for the paper.

Overall I am very content with my work!

Chapter 15

References

AnandTech (2019) ‘Intel’s Manufacturing Roadmap from 2019 to 2029’
https://www.anandtech.com/show/15217/intels-manufacturing-roadmap-from-2019-to-2029

Archimedes Lab ’The Japanese Multiplication Method’ [Online][Accessed on 16th September 2023]
https://www.archimedes-lab.org/Maths2_Multiplication.html

Becquerel, E. (1853) ‘Researches on the electrical conductivity of gases at high temperatures’ *Philosophical Magazine* 4(6), 456-457

Bonhoeffer, K. F., (1948) ‘Activation of passive iron as a model for the excitation of nerve’ *Journal of General Physiology* DOI: 10.1085/jgp.32.1.69

Borresen, J., Lynch, S. (2012) ‘Oscillatory Threshold Logic’ *ResearchGate* [Online] DOI: 10.1371/journal.pone.0048498

Borresen, J., Kit, L., Lynch, S. (2013) ‘Josephson Junction Binary Oscillator Computing’ *ResearchGate* DOI: 10.1109/ISEC.2013.6604275

Chalkiadakis, D., Hizanidis, J. (2022) ‘Dynamical properties of neuromorphic Josephson junctions’ *Physical Review* E 106, 044206 DOI: 10.1103/PhysRevE.106.044206

Chua, L. (1971) ‘Memristor - The missing circuit element’ *IEEE Transactions on Circuit Theory* 18(5) 507–519

Dai, P, Chakoumakos B.C., Sun G.F., Wong K.W., Xin Y., Lu D.F. (1995) ‘Synthesis and neutron powder diffraction study of the superconductor HgBa2Ca2Cu3O8 + δ’

243(3-4) 201-206 DOI: https://doi.org/10.1016/0921-4534(94)02461-8

DeForest, L. (1908) ‘Demodulation of amplitude-modulated oscillations by means of non-linear elements having more than two poles of discharge tubes’ United States Patent 879,532

Fang, X., Duan, S., Wang, L. (2022) ‘Memristive FHN spiking neuron model and brain-inspired threshold logic computing’ *Elsevier Neurocomputing* DOI: https://doi.org/10.1016/j.neucom.2022.08.056

Fitzhugh, R. (1961) ‘Impulses and Physiological States in Theoretical Models of Nerve Membrane’ *Biophysical Journal* 1(6), 445-466 DOI: 10.1016/s0006-3495(61)86902-6

Fleming, J. A. ‘Instrument for converting alternating electric currents into continuous currents’ (1904). United States Patent 803,684

Fudzilla (2014) ‘Forbes works out how many transistors have been made’ [Online][Accessed September 2023]
https://www.fudzilla.com/news/pc-hardware/34833-forbes-works-out-how-many-transistors-have-been-made

The Guardian (2023) ‘There’s no room-temperature superconductor yet, but the quest continues’
https://www.theguardian.com/science/2023/sep/02/room-temperature-superconductor-south-korea-lk-99-nuclear-fusion-maglev

Gray, F; Horton, J. W. (1927) ‘The Production and Utilisation of Television Signals’ *Bell System Technical Journal* 6(4), 579-584

Gray, F (1953) ‘Pulse code communication’. United States Patent 2,632,058

Guthrie, F. (1873) ‘On a new relation between Heat and Electricity’ *Proceedings of the Royal Society of London* (21)

Heil, O. (1934) ‘Improvements in or relating to Electrical Amplifiers and other Control Arrangements and Devices’ United Kingdom Parent 439,457

Hindmarsh, J. L., Rose, R. M. (1984) ‘A model of neuronal bursting using three coupled first order differential equations’ *Proceedings of the Royal Society of London* 221(1222) 87–102

Hodgkin, A., Huxley, A. (1952). ‘A quantitative description of membrane current and its application to conduction and excitation in nerve' *The Journal of Physiology* 117(4) 500–44

Intel Newsroom (2011) 'Intel's Revolutionary 22nm Transistor technology' https://newsroom.intel.com/press-kits/intel-22nm-3-d-tri-gate-transistor-technology/

Josephson, B. D. (1962) 'Possible new effects in superconductive tunnelling' *Phys. Lett.* 1(7) 251-253

Karnaugh, M (1953) 'The Map Method for Synthesis of Combinational Logic Circuits' *Transactions of the American Institute of Electrical Engineers* 72(5), 593-599

Lilienfeld, J. E. (1925) 'Method and apparatus for controlling electric currents' United States Patent 1,745,175

Lynch, S. (June 2014) 'Binary Oscillator Computing' *ResearchGate* DOI: 10.1007/978-3-319-06820-6_20

Lynch, S. (October 2014) 'Brain Inspired Computing' *ResearchGate*

Lynch, S. (2023) *Python for Scientific Computing and Artificial Intelligence* 1st ed., Oxon: CRC Press, ISBN: 978-1-003-28581-6

Middleton, W., Valkenburg, M. (2002) *Reference Data for Engineers* 9th ed., Newnes Press, ISBN: 978-0-7506-7291-7

Minsky, M, Papert, S. A. (1969) 'Perceptrons: An Introduction to Computational Geometry' *The MIT Press*

Morris, C., Lecar, H. (1981) 'Voltage Oscillations in the barnacle giant muscle fiber', Biophysical Journal 35(1) 193–213

Nagumo, J., Arimoto, S., Yoshizawa, S. (1962) 'An Active Pulse Transmission Line Simulating Nerve Axon' *Proceedings of the IRE* 50(10), 2061-2070 DOI: 10.1109/JRPROC.1962.288235

NumPy. (2023) *CPU build options* [Online][Accessed on 27th August 2023] https://numpy.org/doc/stable/reference/simd/build-options.html#on-x86

Nolt, J., Rohatyn, D., Varzi, A. (1998) 'Schaum's outline of theory and problems of logic' (2nd ed.), New York: McGraw-Hill, ISBN 978-0-07-046649-4

Puers R., Baldi L., Van de Voorde M., E. van Nooten S. (2017) 'Nanoelectronics: Materials, Devices, Applications' (2)

Shankland S. (2022) ‘Apple’s M1 Ultra Shows the Future of Computer Chips’ [Online][Accessed September 2023]
https://www.cnet.com/tech/computing/why-the-m1-ultra-is-a-big-deal-apple-shows-the-future-of-chips/

Strukov, D., Snider, G.; Stewart, D.; Williams, R. (2008) ‘The missing memristor found’ *Nature* 453(7191) 80–83

Van der Pol, B. (1926) ‘On Relaxation-Oscillations’ *The London, Edinburgh, and Dublin Phidophical Magazine and Journal of Science* (7) 978-992

Appendix A

libThresholdLogic

```
from typing import Union
from .Neuron import Neuron
from .TransferFunctions import Heaviside
class Gate:
    def __init__(self, n_inputs: int = 2) -> None:
        self.neuron: Neuron = NotImplemented
        raise NotImplementedError
    def __call__(self, *inputs: Union[float, int]) -> float:
        return self.neuron(inputs)
class AND(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        self.neuron = Neuron(
            tuple((1.0 for _ in range(n_inputs))),
            (2 * n_inputs - 1) / 2,
            Heaviside(0)
        )
class NAND(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        self.neuron = Neuron(
            tuple((-1.0 for _ in range(n_inputs))),
            -(2 * n_inputs - 1) / 2,
            Heaviside(0)
        )
class OR(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        self.neuron = Neuron(
            tuple((1.0 for _ in range(n_inputs))),
            1 / 2,
            Heaviside(0)
        )
class NOR(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        self.neuron = Neuron(
            tuple((-1.0 for _ in range(n_inputs))),
            -1 / 2,
            Heaviside(0)
        )
class XOR(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        assert n_inputs == 2
        self.neuron1 = Neuron(
            (1.0, 1.0, -2.0),
            0.5,
            Heaviside(0)
        )
        self.neuron2 = Neuron(
            (1.0, 1.0),
            1.5,
            Heaviside(0)
        )
    def __call__(self, *inputs: Union[float, int], return_carry_bit:
     bool = False) -> float:
        neuron2_eval = self.neuron2(inputs)
        neuron1_eval = self.neuron1((*inputs, neuron2_eval))
        if return_carry_bit:
            return (neuron1_eval, neuron2_eval)
        else:
            return neuron1_eval
class XNOR(Gate):
    def __init__(self, n_inputs: int = 2) -> None:
        assert n_inputs == 2
        self.neuron1 = Neuron(
            (-1.0, -1.0, 2.0),
            -0.5,
            Heaviside(0)
        )
        self.neuron2 = Neuron(
            (1.0, 1.0),
            1.5,
            Heaviside(0)
        )
    def __call__(self, *inputs: Union[float, int]) -> float:
        neuron2_eval = self.neuron2(inputs)
        neuron1_eval = self.neuron1((*inputs, neuron2_eval))
        return neuron1_eval
class GAND(Gate):
    def __init__(self, *positives_map: int) -> None:
        self.neuron = Neuron(
            tuple((1.0 if positive else -1.0 for positive in
    positives_map)),
            (2 * sum(positives_map) - 1) / 2,
            Heaviside(0)
        )
class GNAND(Gate):
    def __init__(self, *positives_map: int) -> None:
        self.neuron = Neuron(
            tuple((-1.0 if positive else 1.0 for positive in
    positives_map)),
            -(2 * sum(positives_map) - 1) / 2,
            Heaviside(0)
        )
```

Listing A.1: `src/libThresholdLogic/Gates.py`

```
from typing import Tuple, Union
from .Neuron import Neuron
from .TransferFunctions import Heaviside
class GenericBitAdder:
    """N neurons with (2 ** N) - 1 (excitatory) inputs
    ie a (2 ** N) - 1 bit adder"""
    def __init__(self, n_neurons: int) -> None:
        self.n_neurons = n_neurons
        self.n_inputs = (2 ** self.n_neurons) - 1
        epsilon = 1 / 2 ** (2 * n_neurons)
        self.neurons = [
            Neuron(
                tuple(1 / (2 ** n) + epsilon for _ in range(self.
    n_inputs)) + # excitatory
                tuple(-(2 ** (m - n)) for m in range(n + 1,
    n_neurons)),      # inhibitory
                1.0,
                Heaviside(0),
            )
```

```python
            for n in range(n_neurons)
        ]
    def __call__(self, *inputs: Union[float, int]) -> Tuple[int]:
        """Returns little bittian"""
        assert len(inputs) == self.n_inputs
        ret = [0 for _ in range(self.n_neurons)]
        for neuron_index, neuron in reversed(list(enumerate(self.
    neurons))):
            neuron_inputs = inputs + tuple(ret[neuron_index + 1:])
            ret[neuron_index] = neuron(neuron_inputs)
        return tuple(ret)
```

Listing A.2: `src/libThresholdLogic/GenericBitAdder.py`

```python
from typing import List, Tuple, Union
import math
from .Gates import Gate, AND
from .GenericBitAdder import GenericBitAdder
def n_neurons(n: int):
    """Returns a list of the number of neurons needed in our generic
     bit adders for `n` by `n` bit multiplication"""
    numbers = [t + 1 for t in range(n)] + [n - 1 - t for t in range(
    n - 1)] + [0, 0]
    for idx in range(len(numbers)):
        if not numbers[idx]:
            break
        n_carry_bits = math.floor(math.log2(numbers[idx]))
        for j in range(n_carry_bits):
            numbers[idx + 1 + j] += 1
    ret = [math.ceil(math.log2(i+1)) for i in numbers]
    try:
        trailing_zero = ret.index(0)
    except ValueError:
        pass
    else:
        ret = ret[:trailing_zero]
    return ret
def convolution_indices(n: int):
    """Give a list of lists of pairs of (x,y) that sum to [0, 1,
    ..., 2n-2] for 0 <= x,y < n"""
    return [
        [(u, t - u) for u in range(t + 1)]
        for t in range(n)
    ] + [[(t + u, n - 1 - u) for u in range(n - t)] for t in range(1, n)]
class GenericBitMultiplier(Gate):
    def __init__(self, n_bit: int) -> None:
        """Perform multiplication for two n_bit operands"""
        self.n_bit = n_bit
        adder_sizes = n_neurons(n_bit)
        self.adders = [GenericBitAdder(t) for t in adder_sizes]
    def __call__(self, x: Tuple[Union[float, int]], y: Tuple[Union[float, int]]) -> float:
        """x and y in little-bittian"""
        assert len(x) == self.n_bit
        assert len(y) == self.n_bit
        x_y_convolution = [
            tuple(
                AND()(x[x_idx], y[y_idx])
                for (x_idx, y_idx) in convol_indices
            )
            for convol_indices in convolution_indices(self.n_bit)
        ] # inputs from x and y
        outputs: List[Tuple[int]] = [] # outputs from adders, including carry bits
        for adder_idx, adder in enumerate(self.adders):
            inputs = tuple()
            try:
                inputs += x_y_convolution[adder_idx]
            except IndexError:
                # the final adders may consist purely of carry bits, no x_y bits
                pass
            for carry_adder_idx, carry_adder_output in enumerate(outputs):
                try:
                    inputs += (carry_adder_output[adder_idx - carry_adder_idx],)
                except IndexError:
                    # adder number ‘carry_adder_idx‘ doesn’t have a carry bit for this adder
                    pass
            inputs += tuple(
                0 for _ in range(adder.n_inputs - len(inputs))
            ) # pad inputs with 0s, tied to ground in the circuitry
            outputs.append(adder(*inputs))
        # first bit in every tuple in `outputs` corresponds to that adder's non-carry (real) output
        return tuple((output[0] for output in outputs))
```

Listing A.3: `src/libThresholdLogic/GenericBitMultiplier.py`

```python
from typing import List, Tuple, Union
import itertools
from .GenericBitAdder import GenericBitAdder
from .util import int_to_bit_tuple_le, bit_tuple_le_to_int
class GenericNumberAdder:
    def __init__(self, n_bit_input: int, n_neurons: int) -> None:
        """Adds `(2 ** n_neurons) - n_neurons` of `n_bit_input`-bit numbers"""
        self.n_bit_input = n_bit_input
        self.n_neurons = n_neurons
        self.adders = [GenericBitAdder(n_neurons) for _ in range(n_bit_input)]
        # we could get away with just 1 in this codebase, but we keep n_bit_input number of them
        # for being analogous to real hardware
    def __call__(self, inputs: List[Tuple[Union[float, int]]]) -> Tuple[int]:
        """Takes in a list of little-bittian tuples, returns little-bittian"""
        if len(inputs) != (2 ** self.n_neurons) - self.n_neurons:
            raise ValueError
        if not all((len(input_) == self.n_bit_input for input_ in inputs)):
            raise ValueError
        output_bits: List[int] = []
        carry_bits = [
            tuple((0 for _ in range(self.n_neurons - 1))) # each adder yields n_neurons - 1 carry bits
            for _ in range(self.n_neurons - 1) # n_neurons - 1 previous adders
        ]
        for nth_bit, nth_bit_adder in enumerate(self.adders):
            nth_bit_adder_evaluated = nth_bit_adder(*tuple(itertools.chain(
                (
                    adder_carry_bits[(self.n_neurons - 1) - 1 -
    nth_carry_bit]
                    for nth_carry_bit, adder_carry_bits in enumerate
    (carry_bits)
                ),
                (input_[nth_bit] for input_ in inputs)
            )))
            output_bits.append(nth_bit_adder_evaluated[0])
            carry_bits.pop(0)
            carry_bits.append(nth_bit_adder_evaluated[1:])
        return tuple(output_bits)
    def add_ints(self, inputs: List[int]):
        return bit_tuple_le_to_int(self([int_to_bit_tuple_le(input_,
     self.n_bit_input) for input_ in inputs]))
```

Listing A.4: `src/libThresholdLogic/GenericNumberAdder.py`

```
from typing import Tuple
from .TransferFunctions import TransferFunction
class Neuron:
    def __init__(
        self,
        weights: Tuple[float],
        bias: float,
        transfer_function: TransferFunction
    ) -> None:
        self.weights = weights
        self.bias = bias
        self.transfer_function = transfer_function
    def __call__(self, inputs: Tuple[float]) -> float:
        return self.transfer_function(sum((i * w for (i, w) in zip(
    inputs, self.weights)))- self.bias)
```

Listing A.5: `src/libThresholdLogic/Neuron.py`

```
class TransferFunction:
    def __call__(self, x: float) -> float:
        raise NotImplementedError
class Heaviside(TransferFunction):
    def __init__(self, threshold: float) -> None:
        self.threshold = threshold
    def __call__(self, x: float) -> float:
        if x > self.threshold:
            return 1
        else:
            return 0
```

Listing A.6: `src/libThresholdLogic/TransferFunctions.py`

```
from .Gates import Gate, AND, NAND, OR, NOR, XOR, XNOR, GAND, GNAND
from .GenericBitAdder import GenericBitAdder
from .GenericBitMultiplier import GenericBitMultiplier
from .GenericNumberAdder import GenericNumberAdder
from .Neuron import Neuron
from .TransferFunctions import Heaviside
from .util import int_to_bit_tuple_le, int_to_bit_tuple_be,
    bit_tuple_le_to_int, bit_tuple_be_to_int, gray_code, gray_numbers
__all__ = [
    "Gate", "AND", "NAND", "OR", "NOR", "XOR", "XNOR", "GAND", "
    GNAND", "GenericBitAdder", "GenericBitMultiplier", "
    GenericNumberAdder", "Neuron", "Heaviside", "int_to_bit_tuple_le"
    , "int_to_bit_tuple_be", "bit_tuple_le_to_int", "
    bit_tuple_be_to_int", "gray_code", "gray_numbers"
]
```

Listing A.7: `src/libThresholdLogic/__init__.py`

```
from typing import List, Tuple
def int_to_bit_tuple_le(x: int, n_bits: int) -> Tuple[int]:
    # little bittian
    return tuple((1 if x & (1 << bit_n) else 0 for bit_n in range(
    n_bits)))
def int_to_bit_tuple_be(x: int, n_bits: int) -> Tuple[int]:
    # big bittian
    return int_to_bit_tuple_le(x, n_bits)[::-1]
def bit_tuple_le_to_int(t: Tuple[int]) -> int:
    # little bittian
    return sum((2 ** idx if bit else 0 for (idx, bit) in enumerate(t
    )))
def bit_tuple_be_to_int(t: Tuple[int]) -> int:
    # big bittian
    return bit_tuple_le_to_int(t[::-1])
def gray_code(n: int) -> List[Tuple[int]]:
    """Returns a list of the first 2^N Gray numbers as big-bittian
    bit tuples"""
    ret = [(0,), (1,)]
    for _ in range(n - 1):
        ret = [(0, *row) for row in ret] + [(1, *row) for row in ret
    [::-1]]
    return ret
def gray_numbers(n: int) -> List[int]:
    """Returns a list of the first 2^N Gray numbers"""
    return [bit_tuple_be_to_int(row) for row in gray_code(n)]
```

Listing A.8: `src/libThresholdLogic/util.py`

```
#!/usr/bin/env python3
from typing import Callable, Tuple, Union
import itertools
from libThresholdLogic import Gate, AND, XOR, GAND,
    bit_tuple_be_to_int, int_to_bit_tuple_be, gray_numbers
def truth_table_entry_2x2_bit_multiply(bits: Tuple[int]):
        y = bit_tuple_be_to_int(bits[2:])
        x = bit_tuple_be_to_int(bits[:2])
        return int_to_bit_tuple_be(x * y, 4)
def truth_table(n_bits: int, entry_fn: Callable[[Tuple[int]], Tuple[
    int]]):
    ret = []
    for i in range(2 ** n_bits):
        bits = int_to_bit_tuple_be(i, n_bits)
        ret.append((*bits, *entry_fn(bits)))
    return ret
class BitMultiplier2x2(Gate):
    def __init__(self) -> None:
        self._1s_and = AND()
        self._2s_and1 = AND()
        self._2s_and2 = AND()
        self._2s_xor = XOR()
        self._4s_gand = GAND(1, 1, 0)
    def __call__(self, *inputs: Union[float, int]) -> float:
        (y1, y0, x1, x0) = inputs
        _1s = self._1s_and(x0, y0)
        _2s, _8s = self._2s_xor(self._2s_and1(x1, y0), self._2s_and2
    (x0, y1), return_carry_bit = True)
        _4s = self._4s_gand(x1, y1, _1s)
        return (_8s, _4s, _2s, _1s)

class Table:
    def __init__(self, data, column_labels, row_labels) -> None:
        self.data = data
```

```python
        self.column_labels = column_labels
        self.row_labels = row_labels
        assert len(self.row_labels) == len(self.data)
        assert len(self.column_labels) == len(self.data[0])
    def __format__(self, format_spec = "") -> str:
        ret = ""
        for row_label, row in itertools.chain(((" ", self.column_labels),), zip(self.row_labels, self.data)):
            ret += " ".join((f"{col:{format_spec}}" for col in itertools.chain((row_label,), row))) + "\n"
        ret = ret[:-1] # remove trailing '\n'
        return ret
    def __str__(self) -> str:
        return self.__format__()
CSI_BG_DEFAULT  = "\x1b[49m"
CSI_BG_BR_RED   = "\x1b[101m"
CSI_BG_BR_GREEN = "\x1b[102m"
def main():
    i_to_gray = gray_numbers(2)
    for i, gray_i in enumerate(i_to_gray):
        print(f"gray({i}) = {gray_i} = {gray_i:02b}")
    print("2x2 bit multiplication truth table")
    truth_table_entries = truth_table(4, truth_table_entry_2x2_bit_multiply)
    t = Table(
        truth_table_entries,
        ["Y_1", "Y_0", "X_1", "X_0", "Z_3", "Z_2", "Z_1", "Z_0"],
        [f"{y}*{x}={y*x}" for (y, x) in (
            (bit_tuple_be_to_int(bits[0:2]), bit_tuple_be_to_int(bits[2:4]))
            for bits in truth_table_entries
        )]
    )
    print(f"{t:>5}")
    N_COLS, N_ROWS = 4, 4
    for bit_number, output_bit_column_index in ((t, 8 - 1 - t) for t in range(4)):
        print(f"Karnaugh Map for Z_{bit_number}; 2 ** {bit_number} = {2 ** bit_number}")
        t = Table([[f"{CSI_BG_BR_GREEN if bit else CSI_BG_BR_RED}{bit}{CSI_BG_DEFAULT}" for bit in (
                        truth_table_entries[entry_index][output_bit_column_index]
                        for entry_index in
                        (N_COLS * y_gray + x_gray for x_gray in (i_to_gray[x] for x in range(N_COLS))))]
                for y_gray in (i_to_gray[y] for y in range(N_ROWS))], i_to_gray, i_to_gray
        )
        print(t)
    print("2x2 Bit Multiplier output")
    multiplier = BitMultiplier2x2()
    multiplier_truth_table = []
    for i in range(2 ** 4):
        bits = int_to_bit_tuple_be(i, 4)
        multiplier_truth_table.append((*bits, *multiplier(*bits)))
        print(multiplier_truth_table[-1])
    print(multiplier_truth_table == truth_table_entries)
if __name__ == "__main__":
    main()
```

Listing A.9: `test/libThresholdLogic/bit-multiplier.py`

```
#!/usr/bin/env python3
from typing import Tuple
from libThresholdLogic import int_to_bit_tuple_be, GAND, GNAND
def test_tuple(t: Tuple[int]):
    for gate_class in [GAND, GNAND]:
        print(gate_class.__name__, "of", t)
        gate = gate_class(*t)
        for i in range(2 ** len(t)):
            bits = int_to_bit_tuple_be(i, len(t))
            print(bits, gate(*bits))
def main():
    for inputs in [(1, 1, 0, 1), (1, 0, 1),]:
        test_tuple(inputs)
if __name__ == "__main__":
    main()
```

Listing A.10: `test/libThresholdLogic/gand-and-gnand.py`

```
#!/usr/bin/env python3
from libThresholdLogic import Gate, NAND, int_to_bit_tuple_be
class NANDBasedNOT(Gate):
    def __init__(self) -> None:
        self.nand1 = NAND()
    def __call__(self, x: float) -> float:
        return self.nand1(x, x)
class NANDBasedAND(Gate):
    def __init__(self) -> None:
        self.nand1 = NAND()
        self.not1 = NANDBasedNOT()
    def __call__(self, x: float, y: float) -> float:
        return self.not1(self.nand1(x, y))
class NANDBasedOR(Gate):
    def __init__(self) -> None:
        self.nand1 = NAND()
        self.not1 = NANDBasedNOT()
        self.not2 = NANDBasedNOT()
    def __call__(self, x: float, y: float) -> float:
        return self.nand1(self.not1(x),self.not2(y),)
class NANDBasedNOR(Gate):
    def __init__(self) -> None:
        self.or1 = NANDBasedOR()
        self.not1 = NANDBasedNOT()
    def __call__(self, x: float, y: float) -> float:
        return self.not1(self.or1(x, y))
class NANDBasedXOR(Gate):
    def __init__(self) -> None:
        self.nand1 = NAND()
        self.nand2 = NAND()
        self.nand3 = NAND()
        self.nand4 = NAND()
    def __call__(self, x: float, y: float) -> float:
        nand_1_out = self.nand1(x, y)
        return self.nand4(self.nand2(x, nand_1_out),self.nand2(y,
   nand_1_out),)
class NANDBasedXNOR(Gate):
    def __init__(self) -> None:
        self.xor1 = NANDBasedXOR()
        self.not1 = NANDBasedNOT()
    def __call__(self, x: float, y: float) -> float:
        return self.not1(self.xor1(x, y))
def main():
    for gate_class in [NAND,NANDBasedAND,NANDBasedOR,NANDBasedNOR,
   NANDBasedXOR,NANDBasedXNOR,]:
        print(gate_class.__name__)
        gate = gate_class()
        for i in range(2 ** 2):
            bits = int_to_bit_tuple_be(i, 2)
            print(bits, gate(*bits))
    for gate_class in [NANDBasedNOT]:
        print(gate_class.__name__)
        gate = gate_class()
        for i in range(2 ** 1):
            bits = int_to_bit_tuple_be(i, 1)
            print(bits, gate(*bits))
if __name__ == "__main__":
    main()
```

Listing A.11: `test/libThresholdLogic/gate-nand.py`

```
#!/usr/bin/env python3
from libThresholdLogic import AND, NAND, OR, NOR, XOR, XNOR,
   int_to_bit_tuple_be
def main():
    n_inputs = 2
    for gate_class in [AND, NAND, OR, NOR, XOR, XNOR]:
        print(gate_class.__name__)
        gate = gate_class(n_inputs = n_inputs)
        for i in range(2 ** n_inputs):
            bits = int_to_bit_tuple_be(i, n_inputs)
            print(bits, gate(*bits))
if __name__ == "__main__":
    main()
```

Listing A.12: `test/libThresholdLogic/gates.py`

```
#!/usr/bin/env python3
from libThresholdLogic import bit_tuple_le_to_int, GenericBitAdder,
   GenericNumberAdder
def test_generic_bit_adder() -> None:
    for N in range(2, 5):
        bit_adder = GenericBitAdder(N)
        for k in range(2 ** N):
            inputs = tuple([1 for _ in range(k)] + [0 for _ in range
   (2 ** N - 1 - k)])
            result = bit_adder(*inputs)
            result_int = bit_tuple_le_to_int(result)
            print(inputs, result, "=", result_int)
            assert result_int == sum(inputs)
```

```python
def test_generic_number_adder() -> None:
    number_adder = GenericNumberAdder(8, 3)
    inputs = [(1, 1, 0, 0, 0, 0, 0, 0),(1, 1, 0, 1, 0, 0, 0, 0),(1, 0, 0, 0, 0, 0, 0, 0),(1, 0, 0, 1, 0, 0, 0, 0),(1, 0, 0, 0, 0, 0, 0, 0),]
    result = number_adder(inputs)
    result_int = bit_tuple_le_to_int(result)
    print(f"Sum of {inputs} = {result} = {result_int}")
    decimal_inputs = [32, 3, 1, 2, 3]
    result = number_adder.add_ints(decimal_inputs)
    print(f"Sum of {decimal_inputs} = {result}")
    assert result == sum(decimal_inputs)
def main() -> None:
    test_generic_bit_adder()
    test_generic_number_adder()
if __name__ == "__main__":
    main()
```

Listing A.13: `test/libThresholdLogic/generic-adder.py`

```python
#!/usr/bin/env python3
import itertools
from libThresholdLogic import bit_tuple_le_to_int, int_to_bit_tuple_le, GenericBitMultiplier
def test_full_n_bits(N: int):
    multiplier = GenericBitMultiplier(N)
    for x, y in itertools.product(range(2 ** N), range(2 ** N)):
        assert bit_tuple_le_to_int(multiplier(int_to_bit_tuple_le(x, N),int_to_bit_tuple_le(y, N),)) == x * y
def main():
    test_full_n_bits(8) # test all 8-bit by 8-bit multiplications
    N = 5
    multiplier = GenericBitMultiplier(N)
    x, y = (31, 17)
    inputs = tuple(int_to_bit_tuple_le(i, N) for i in (x, y))
    result = multiplier(*inputs)
    result_int = bit_tuple_le_to_int(result)
    assert x * y == result_int
    print(f"{x} * {y} = {result_int}")
if __name__ == "__main__":
    main()
```

Listing A.14: `test/libThresholdLogic/generic-multiplier.py`

```python
#!/usr/bin/env python3
from libThresholdLogic import gray_numbers
def main():
    i_to_gray = gray_numbers(3)
    for i, gray_i in enumerate(i_to_gray):
        print(f"gray({i}) = {gray_i} = {gray_i:03b}")
if __name__ == "__main__":
    main()
```

Listing A.15: `test/libThresholdLogic/gray-exaple.py`

Appendix B

libJJ

```python
class CurrentBase:
    def __call__(self, t: float) -> float:
        raise NotImplementedError
```

Listing B.1: `src/libJJ/Current/Base.py`

```python
from .Base import CurrentBase
class ConstCurrent(CurrentBase):
    def __init__(self, const: float) -> None:
        self.const = const
    def __call__(self, t: float) -> float:
        return self.const
```

Listing B.2: `src/libJJ/Current/Const.py`

Listing B.3: `src/libJJ/Current/__init__.py`

```python
from scipy.integrate import odeint
import numpy as np
from .Current.Base import CurrentBase
class JJCouple:
    """Our Neuron class"""
    def __init__(self, Gamma: float, lambda_: float, L_s: float, L_p
   : float, i_in_factory: CurrentBase, i_b_factory:  CurrentBase,
    ) -> None:
        self.Gamma = Gamma
        self.lambda_ = lambda_
        self.L_s = L_s
        self.L_p = L_p
        self.i_in_factory = i_in_factory
        self.i_b_factory  = i_b_factory
    def ode(self, y, t: float):
        i_in = self.i_in_factory(t)
        i_b = self.i_b_factory(t)
        phi_p, omega_p, phi_c, omega_c = y
        dy_dt = [omega_p, -self.lambda_ * (phi_p + phi_c) + self.L_s * i_in + (1 - self.L_p) * i_b - self.Gamma * omega_p - np.sin(phi_p), omega_c, -self.lambda_ * (phi_p + phi_c) + self.L_s * i_in - self.L_p * i_b - self.Gamma * omega_c - np.sin(phi_c)]
        return dy_dt
    def solve(self, phi_p0: float, omega_p0: float, phi_c0: float, omega_c0: float, t):
        y0 = (phi_p0, omega_p0, phi_c0, omega_c0)
        solution = odeint(self.ode, y0, t)
        (phi_p, omega_p, phi_c, omega_c) = tuple((solution[:, i] for i in range(4)))
        return (phi_p, omega_p, phi_c, omega_c)
```

Listing B.4: `src/libJJ/JJCouple.py`

```python
"""Josephson Junctions"""
from .JJCouple import JJCouple
from .Current.Const import ConstCurrent
__all__ = ["JJCouple","ConstCurrent",]
```

Listing B.5: `src/libJJ/__init__.py`

```python
import numpy as np
import libJJ
values = []
for i_in in np.arange(0.0, 0.3, 0.01):
    params = {
        "Gamma": 0.95, "lambda": 0.1, "L_s": 0.5, "L_p": 0.5, "i_in_factory": libJJ.ConstCurrent(i_in), "i_b_factory": libJJ.ConstCurrent(1.909),
        "initial": {"phi_p": 0.0, "omega_p": 0.0, "phi_c": 0.0, "omega_c": 0.0},
        "results": {"phi_p": None, "omega_p": None, "phi_c": None, "omega_c": None}
    }
    couple = libJJ.JJCouple(params["Gamma"], params["lambda"], params["L_s"], params["L_p"], params["i_in_factory"], params["i_b_factory"])
    t = np.linspace(0, 100, 1001)
    y0 = (params["initial"]["phi_p"], params["initial"]["omega_p"],
   params["initial"]["phi_c"], params["initial"]["omega_c"])
    (params["results"]["phi_p"], params["results"]["omega_p"],
   params["results"]["phi_c"], params["results"]["omega_c"]) =
   couple.solve(*y0, t)
    values.append(params)
import matplotlib.pyplot as plt
from matplotlib import style
fig = plt.figure()
ax = fig.add_subplot(111)
ax.grid()
style.use("ggplot")
ymin = min((value["results"]["phi_p"] + value["results"]["phi_c"]).
   min() for value in values)
ymax = max((value["results"]["phi_p"] + value["results"]["phi_c"]).
   max() for value in values)
ax.set_ylim(ymin - 0.1, ymax + 0.1)
value = values[0]
line, = ax.plot(t, np.zeros(t.shape))
def animate(i: int):
    value = values[i]
    line.set_data(t, value["results"]["phi_p"] + value["results"]["
   phi_c"])
    i_in = value["i_in_factory"].const
    ax.set_title(f"i_in: {i_in:.2f}")
plt.xlabel("t")
plt.ylabel("$\\phi_p$ + $\\phi_c$")
plt.close()
from matplotlib import animation
from IPython.display import HTML
anim = animation.FuncAnimation(fig,animate,frames = len(values),
   interval = 50,blit = False,repeat_delay = 500)
HTML(anim.to_jshtml())
```

Listing B.6: test/libJJ/jj.ipynb

Appendix C

libFHN

```python
"""
Factories which are called in Neuron::_ode to give the
input currents into a neuron, depending on time, and possibly other
Neurons' states
"""
from __future__ import annotations
from typing import List, Tuple
import numpy as np
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from .Neuron import Neuron
class BaseInput:
    def __call__(self, neuron_to_X, t: float) -> float:
        """
        `neuron_to_X` is a dictionary from neuron instances to their
 state,
        that is their 'u' and 'v' values. This gives each neuron a
 global
        view of the NeuronNetwork if needed when solving the ode
 steps
        """
        raise NotImplementedError
class ConstInput(BaseInput):
    """Return a time-independent constant"""
    def __init__(self, value: float) -> None:
        self.value = value
    def __call__(self, neuron_to_X, t: float) -> float:
        return self.value
```

```python
class RangesInput(BaseInput):
    """Specify the input for ranges of time"""
    def __init__(self, ranges: List[Tuple[float, float, float]]) -> None:
        self.ranges = ranges
    def __call__(self, neuron_to_X, t: float) -> float:
        try:
            return next((value for (start, end, value) in self.ranges if start <= t <= end))
        except StopIteration:
            if t < self.ranges[0][0]: # before first range's start
                return self.ranges[0][2] # return value
            else:
                return self.ranges[-1][2] # return last range's value
class RangesConstInput(BaseInput):
    """Specify the ranges of time for a constant input"""
    def __init__(self, ranges: List[Tuple[float, float]], value_high: float = 1.0, value_low: float = 0.0) -> None:
        self.ranges = ranges
        self.value_high = value_high
        self.value_low = value_low
    def __call__(self, neuron_to_X, t: float) -> float:
        if any((start <= t <= end for (start, end) in self.ranges)):
            return self.value_high
        else:
            return self.value_low
class WeightedInput(BaseInput):
    """An input of weighted values after a transfer function is applied"""
    def __init__(self, *neurons_and_weights: Tuple[Neuron, float]) -> None:
        self.neurons_and_weights = neurons_and_weights
    def __call__(self, neuron_to_X, t: float) -> float:
        return sum((weight * self.transfer(neuron_to_X[neuron][1]) for neuron, weight in self.neurons_and_weights))
        # neuron_to_X[neuron] is the input `X := (u, v)`
    def transfer(self, v: float, m: float = 100.0, c: float = 60.0) -> float:
        """Sigmoid transfer function"""
        return 1 / (1 + np.exp(-(m * v - c)))
```

Listing C.1: `src/libFHN/Inputs.py`

```python
from __future__ import annotations
from typing import Tuple
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from .Inputs import BaseInput
class Neuron:
    """Our FHN neuron class"""
    def __init__(self, i_in_factory: BaseInput, gamma: float = 0.1,
   theta: float = 0.1, epsilon: float = 0.1,
    ) -> None:
        self.i_in_factory = i_in_factory
        self.gamma = gamma
        self.theta = theta
        self.epsilon = epsilon

    def _ode(self, neuron_to_X, t: float) -> Tuple[float, float]:
        """The FHN differential equation is evaluated here"""
        gamma = self.gamma
        theta = self.theta
        epsilon = self.epsilon
        u, v = neuron_to_X[self]
        i_in = self.i_in_factory(neuron_to_X, t) * 0.8
        dy_dt = (-u * (u - theta) * (u - 1) - v + i_in, epsilon * (u
    - gamma * v))
        return dy_dt
```

Listing C.2: `src/libFHN/Neuron.py`

```python
from typing import List, Tuple
from scipy.integrate import odeint
from .Neuron import Neuron
class NeuronNetwork:
    """A network of FHN neurons which can be evaluated to see their
   interactions"""
    def __init__(self, neurons: List[Neuron]) -> None:
        self.neurons = neurons
    def _ode(self, X, t: float) -> Tuple[float]:
        """The function which is passed to `odeint`. This splits the
    input vector X into the components
        which get delegated to their corresponding Neuron, to solve
   the DEs for all neurons at once
        """
        neuron_to_X = {neuron: X[neuron_idx * 2 : (neuron_idx + 1) * 2] for neuron_idx, neuron in enumerate(self.neurons)}
        ret = tuple()
        for neuron in neuron_to_X:
            ret += neuron._ode(neuron_to_X, t)
        return ret
    def solve(self, y0, t):
        return odeint(self._ode, y0, t)
```

Listing C.3: `src/libFHN/NeuronNetwork.py`

```
"""Fitzhugh-Nagumo Neuron Networks"""
from .Inputs import BaseInput, ConstInput, RangesInput, RangesConstInput, WeightedInput
from .Neuron import Neuron
from .NeuronNetwork import NeuronNetwork
__all__ = ["BaseInput", "ConstInput", "RangesInput", "RangesConstInput", "WeightedInput", "Neuron", "NeuronNetwork",]
```

Listing C.4: `src/libFHN/__init__.py`

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt
from libFHN import NeuronNetwork, Neuron, RangesConstInput, WeightedInput
class FullAdder(NeuronNetwork):
    def __init__(self) -> None:
        neuron_in_1 = Neuron(RangesConstInput([(250, 500),(1000, 1500),(1750, 2000),]))
        neuron_in_2 = Neuron(RangesConstInput([(500, 750),(1000, 1250),(1500, 2000),]))
        neuron_in_3 = Neuron(RangesConstInput([(750, 1000),(1250, 2000),]))
        neuron_carry = Neuron(WeightedInput((neuron_in_1, 0.5),(neuron_in_2, 0.5),(neuron_in_3, 0.5),))
        neuron_sum = Neuron(WeightedInput((neuron_in_1, 1.0),(neuron_in_2, 1.0),(neuron_in_3, 1.0),(neuron_carry, -2.0),))
        neurons = [neuron_in_1, neuron_in_2, neuron_in_3, neuron_sum, neuron_carry]
        super().__init__(neurons)
def main():
    adder = FullAdder()
    t = np.arange(0.0, 2000.0, 0.1)
```

```python
    soln = adder.solve([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0], t)
    u1, v1, u2, v2, u3, v3, u4, v4, u5, v5 = soln.T
    plt.subplots_adjust(hspace = 1)
    plt.figure(1)
    for idx, (res, label, col) in enumerate([(u1, "I$_1$", "brown"),(u2, "I$_2$", "brown"),(u3, "I$_3$", "brown"),(u4, "O$_{s}$", "green"),(u5, "O$_{c}$", "green"),
    ], start = 1):
        plt.subplot(5, 1, idx)
        plt.plot(t, res, col)
        plt.ylim(-1, 1.5)
        plt.ylabel(label)
        if idx == 1:
            plt.title("Fitzhugh-Nagumo Full Adder")
    plt.xlabel("Time")
    plt.show()
if __name__ == "__main__":
    main()
```

Listing C.5: `test/full-adder.py`

```python
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt
from libFHN import NeuronNetwork, Neuron, RangesConstInput, WeightedInput
class HalfAdder(NeuronNetwork):
    def __init__(self) -> None:
        neuron_in_1 = Neuron(RangesConstInput([(500, 1000),(1500, 2000),]))
        neuron_in_2 = Neuron(RangesConstInput([(1000, 2000),]))
        neuron_carry = Neuron(WeightedInput((neuron_in_1, 0.5),(neuron_in_2, 0.5),))
        neuron_sum = Neuron(WeightedInput((neuron_in_1, 1.0),(neuron_in_2, 1.0),(neuron_carry, -2.0),))
        neurons = [neuron_in_1, neuron_in_2, neuron_sum, neuron_carry]
        super().__init__(neurons)

def main():
    adder = HalfAdder()
    t = np.arange(0.0, 2000.0, 0.1)
    soln = adder.solve([0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0], t)
    u1, v1, u2, v2, u3, v3, u4, v4 = soln.T
    plt.subplots_adjust(hspace = 1)
    plt.figure(1)
    for idx, (res, label, col) in enumerate([(u1, "I$_1$", "brown")
    ,(u2, "I$_2$", "brown"),(u3, "O$_{s}$", "green"),(u4, "O$_{c}$",
    "green"),
    ], start = 1):
        plt.subplot(4, 1, idx)
        plt.plot(t, res, col)
        plt.ylim(-1, 1.5)
        plt.ylabel(label)
        if idx == 1:
            plt.title("Fitzhugh-Nagumo Half Adder")
    plt.xlabel("Time")
    plt.show()
if __name__ == "__main__":
    main()
```

Listing C.6: `test/half-adder.py`

```
from typing import Tuple
import numpy as np
import matplotlib.pyplot as plt
class FHNNeuron:
    def __init__(self,a: float,b: float,c: float,z: float,) -> None:
        self.a = a
        self.b = b
        self.c = c
        self.z = z
    def ode(self, X, t: float) -> Tuple[float, float]:
        x, y = X
        dy_dt = (self.c * (y + x - (x ** 3) / 3 + self.z),-(x - self
    .a + self.b * y) / self.c)
        return dy_dt
def nullcline1(x, a, b, c, z):
    return -x + (x ** 3) / 3 - z
def nullcline2(x, a, b, c, z):
    return (a - x) / b
from scipy.integrate import odeint
import itertools
def plot_phase_plane(a, b, c, z, xmin, xmax, ymin, ymax):
    fhn_neuron = FHNNeuron(a, b, c, z)
    fig, ax = plt.subplots()
    fig.set_figwidth(9)
    X = np.arange(-3.0, 3.0, 0.1)
    Y = nullcline1(X, a, b, c, z)
    ax.plot(X, Y, label = "nullcline1")
    Y = nullcline2(X, a, b, c, z)
    ax.plot(X, Y, label = "nullcline2")
    X = np.arange(-1.5, 2.5, 0.2)
    Y = np.arange(-1.5, 1.5, 0.2)
    res = np.empty((len(X) * len(Y), 4))
    for index, (y, x) in enumerate(itertools.product(Y, X)):
        coords = fhn_neuron.ode((x, y), None)
        res[index] = (x, y, coords[0], coords[1])
    x = list(r[0] for r in res)
    y = list(r[1] for r in res)
    u = list(r[2] for r in res)
    v = list(r[3] for r in res)
    q = ax.quiver(x, y, u, v, label = "Phase Plane")
    t = np.arange(0.0, 2000.0, 0.1)
    soln = odeint(fhn_neuron.ode, [0.5, 0.5], t)
    x = [s[0] for s in soln]
    y = [s[1] for s in soln]
    ax.plot(x, y, label = "FHN")
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(ymin, ymax)
    ax.set_title(f"FHN Phase Plane, $a={a}$, $b={b}$, $c={c}$, $z={z
   }$")
    ax.legend(loc="lower left")
plot_phase_plane(0.7, 0.8, 3.0, 0.0, -1.0, 2.0, -1.0, 1.0)
plot_phase_plane(0.7, 0.8, 3.0, -0.4, -1.75, 2.0, -1.0, 1.5)
a, b, c, z = 0.7, 0.8, 3.0, 0.4
t = np.arange(0.0, 2000.0, 0.1)
solns = []
def gen_solns(num_solns: int):
    for i in range(num_solns):
        z = -0.03 * i
        a, b, c, z = 0.7, 0.8, 3.0, z
        fhn_neuron = FHNNeuron(a, b, c, z)
        soln = odeint(fhn_neuron.ode, [0.5, 0.5], t)
        x = [s[0] for s in soln]
        y = [s[1] for s in soln]
        solns.append((x, y, z))
```

```python
NUM_FRAMES = 50
gen_solns(NUM_FRAMES)
from matplotlib import style
fig, ax = plt.subplots()
ax.grid(visible = 1)
style.use("ggplot")
soln_pts, = ax.plot([], [], label = "FHN")
fig.set_figwidth(9)
def animate(i: int):
    (x, y, z) = solns[i]
    soln_pts.set_data(x, y)
    ax.set_title(f"FHN Phase Plane, $a={a}$, $b={b}$, $c={c}$, $z={z
    :.3f}$")
ax.set_xlim(-2.0, 2.0)
ax.set_ylim(-1.0, 3.0)
animate(0)
ax.legend(loc = "lower left")
plt.close()
from matplotlib import animation
from IPython.display import HTML
anim = animation.FuncAnimation(fig,animate,frames = NUM_FRAMES,
    interval = 50,blit = False,repeat_delay = 500)
HTML(anim.to_jshtml())
```

Listing C.7: `test/fhn-region.ipynb`