Kyrre Glette Øyvind Tanum

January 21, 2004
TDT4720 Datamaskinkonstruksjon og -arkitektur, fordypningsemne
Subject teacher: Pauline Haddow
Teaching supervisor: Pauline Haddow

Abstract

This document report the autumn project of two fifth-year masterstudents at the Department of Computer Science and Information Technology, NTNU, Trondheim.

The agreed upon assignment was to build a flying machine that through evolution should learn to hover. Hovering means keeping a constant position in the air.

This project introduces a variety of different design techniques and tools, where most of them were almost new to the students. Both students had some knowledge about microcontrollers and Field Programmable Gate Arrays, but they were far from experienced. A project goal was to learn more about hardware programming and the use of microcontrollers.

To realise the flying machine some custom parts had to be constructed, and the students had to spend some time in a metal workshop.

The microcontroller is used to receive sensor input. The microcontroller converts the input values and sends them to the fpga. The microcontroller also generates signals for controlling a electromotor.

A virtual brain is implemented in hardware using a FPGA. The brain is realised building a neural net consisting of neurons, axons, dendrites and synapses. Neurons are like brain cells, where dendrite is the neuron input, axon is the neuron output and synapse is the connection between axons and dendrites.

A genetic algorithm controls the evolution of a population. Several methods, similar to nature's evolution methods, was implemented to manipulate the different individuals.

At the end of the project the Evocopter was able to hover, although it did not reach the wanted height. This was achieved using a genetic algorithm on a personal computer to control the evolution of a neural net, that was implemented in hardware.

Trondheim, January 21, 2004.

Øyvind Tanum

Kyrre Glette

Contents

Chapter 1

Introduction

This project has been named Evocopter, as we thought it was a suitable name for our evolving flying device. We have been free to choose the subject of the project ourselves, as long as it would be relevant for our field of study. After having decided to do something with robot control, we have been through many ideas before arriving on the Evocopter. We will first present the subject in terms of goals and motivation, before proceeding on to the background information necessary to follow our work. Then follows an in-detail description of our system and experiments. Finally we will sum up the results of the project.

1.1 Goals

In terms of results, the goal is to evolve a controller that keeps the evocopter at a certain height. This height will be fixed but could later on be specified as an input to the controller. The Evocopter should also be able to reach this position from a start position, ie. it should be able to perform a takeoff maneuver. Later this could be extended to for instance stabilizing rotors and free flight.

Science-wise, our goals are to explore evolution on a hardware design, combined with fitness evaluation in real world, and to explore ways of designing controllers in FPGAs, in particular neural networks.

1.2 Motivation

This section describes what inspired and motivated us to formulate the project the way we did.

1.2.1 General

Out motivations for this kind of project has been many: For a start, we wanted to do something that uses hardware and works in the real world, not just in a simulation. This is interesting for robot control, and it is really more motivating to have to work with something that is very concrete. Secondly, we wanted to learn more about hardware design, using VHDL and FPGAs. Neural networks are also one of our fields of interest, so this was seen as an opportunity to learn more about them. This could be nicely combined with evolution. We looked upon the challenge to do fitness evaluation in the real world as an interesting one, and something that should be possible to accomplish.

1.2.2 Motivation for the mechanical design

Evolution should be automated, it should not be necessary to supervise the evolutionary process, as this would risk taking much time. Therefore we decided to use a stationary robot that can return automatically to its initial position. A walking robot for instance could easily get stuck in some way.

As our motivation is to learn about neural networks and use an FPGA, we tried to choose a task that uses a simple reaction mechanism, as this is easier to obtain with a nn than more complex state-remembering behaviours. This steered us away from the locomotion tasks that we thought of in the beginning.

We have chosen not to concentrate on size and mobility of the system as this would increase the complexity of the project. A mobile system could be considered as a challenge for the future.

1.2.3 Inspirations

Although our inspirations have been many, there are two major sources of inspiration for our system:

- "Creation of a Learning, Flying Robot by Means of Evolution", Augustsson, Wolff and Nordin. [6] This paper is a report on evolving a controller for an ornithopter-like device. As this project also uses real-life evaluation, we were inspired by their test setup. They also use a sliding system to constrain the movement of their robot.
- D. Floreano and the ASL team's [1] various mobile robots. The results that have been achieved with these robots, both flying and ground-based, are interesting. Some of these controllers use spiking neural networks in hardware.

Chapter 2

Background

This chapter describes background information about the fields and techniques that are studied in this project.

2.1 Evolutionary algorithms

Evolutionary algorithms are inspired by evolution in nature. An evolutionary algorithm is, in short, a form of search algorithm. It can be seen as a parallel stochastic search. This kind of search is useful when the search space is large (there are many parameters to explore). It is good for avoiding getting stuck at local maxima in the search space. There are several different types of evolutionary algorithms. We have studied one of the most common, the genetic algorithm (GA).

2.1.1 Genetic algorithm overview

A genetic algorithm normally has the following form:

1. Create a new "population" of randomly generated "individuals" (solutions).
2. Evaluate the "fitness" (performance) of these solutions.
3. If a decent solution has been found, end the algorithm. Else, continue.
4. Select a new population based on the individuals in the old one, and their fitness.
5. Go to point 2.

The description of an individual is, like in nature, done through a genome. This genome is then used for building a phenotype that can be evaluated. The performance of this phenotype is the basis for how the genome, or genotype, is changed.

2.1.2 Selection methods

Common methods of selection are either fitness-proportionate selection or tournament selection. We have chosen to use fitness-proportionate selection. Here one whole population of individuals are evaluated and ranked based on their fitness. Then, based on this ranking, individuals for a new population are selected.

Fitness evaluation

Each individual/phenotype will have a fitness associated to it. This is determined through fitness evaluation, where one measures the individuals ability to perform as specified through a fitness function. The fitness function should be able to indicate gradually how good an individual is, in order for evolution to work.

Selection pressure

Selection pressure is a function that transforms the measured fitness value for an individual and converts it to a so-called expected value (expval). This is a function that is often applied to compensate for the fact that in early generations, fitness value variance is high, while later on as individuals become more fit, fitness value variance is low. The pressure function should give expvals that have a more constant variation.

Roulette wheel mechanism

Selection of individuals is commonly done through a "roulette wheel" selection mechanism. An individual's probability of selection (area on the roulette wheel, see figure 2.1) is based on its expval. Therefore, more fit individuals have a higher probability of being selected for reproduction.

2.1.3 Genome operators

During selection of a new population, elitism, crossover, mutation and the creation of new random individuals as well as other less common methods can be used.

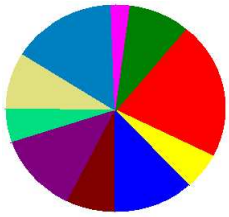

Figure 2.1: Illustration of a roulette wheel. Larger slices have higher probability to be selected.

Elitism

To guarantee that the best individual(s) survive(s) elitism is used. Elitism is basically that the best individuals are copied to the next generation. If elitism is not applied, one can experience that the best values oscillate, good solutions can be lost during evolution.

Crossover

Crossover is the reproduction of individuals phase. Like in nature the mother and father attributes are mixed together. For a genetic algoritm using bit-strings there are a couple of standard ways to perform the crossover:

- One-point crossover. One crossover point is selected randomly in the bit string. Then the part of the genome to the right or left of this point is exchanged with the other genome.
- Two-point crossover. Two crossover points are selected randomly in the bit string. See figure 2.2 for an illustration of a two-point crossover. The effect of this is that only a "slice" of one genome is exchanged with another. The process is meant to be less disruptive than one-point crossover.

There are other ways to perform crossover as well, including so-called uniform crossover and also knowledge-based crossovers which are optimized for a particular coding.

Mutation

Mutation is a way to add variation to the genome. Most common is to have a probability for each bit in the genome to mutate. Mutations could also be specific to a particular coding in order to make them more effective.

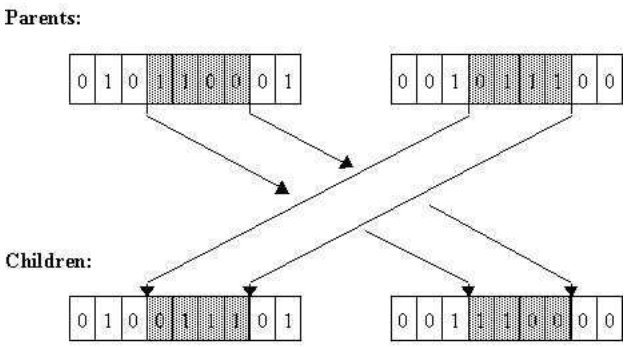

Figure 2.2: An illustrative picture of the crossover operation with two crossover points.

New individuals

As a variation on mutation, new individuals with random genomes can be injected into a population. This is done in order to introduce new "fresh blood" to the evolution. Whether this is efficient or not depends on the problem to be solved.

2.2 Evolution on hardware

We have decided to implement a digital design. An FPGA (Field Programmable Gate Array) is convenient to use as these are reconfigurable and the only practical way to test out the design in hardware without starting a chip manufacturing process. FPGAs can be reconfigured with a bit stream, either fully or partially. Partial reconfiguration could be a solution for evolving different controller solutions, but this method is somewhat complicated. Full reconfiguration is done by using a bitstream that comes from a synthesis tool. This stream is encrypted, and also tampering with this could possibly cause short-circuits in the FPGA. A synthesis process (generation of a bit stream from a high-level description) is quite slow. Another solution is to synthesize and implement the hardware design on the FPGA with all possible connections and functionalities, and then configure this new system (which could be seen as a new, virtual piece of hardware) using custom configuration logic implemented on the FPGA. This would require the use of another communications protocol, like a serial interface or something faster. This has the potential of becoming much faster than full or partial reconfiguration of the FPGA, which has also been shown in[8].

2.3 Real-world evaluation

It is common that evolution projects are using some kind of simulator for evaluating the fitness of individuals. Often this is the easiest solution. Fitness can be evaluated quickly, and the simulator often simulates a more or less perfect conditions for the individuals. However, if solutions are taken from this evolution process and evaluated in a real world (for instance robot controllers that are tried out on real robots), they might not get a good fitness value anymore. This may be caused because the simulation could not model the real world accurately enough. Evaluating individuals in the real world, on the other hand, can be difficult or impractical. Evaluation can take long time for each individual. You cannot just reset the real world, so for example if the robot gets stuck you will have to help it back on track again. Stationary systems are easier, but there may still be mechanical difficulties, sensor noise etc. The reward is a system that works in the real world.

2.4 Neural networks

Neural networks, inspired by the neural structures in nature, are used for various tasks - pattern recognition/categorizing, control systems and more. Basically, these structures consist of neuron models that are connected in some kind of network with a certain topology.

Neuron models have the following in common: The neuron has several inputs (dendrites) and one output (axon). The dendrites are connected to other neurons' axons through so-called synapse connections. When a neuron receives enough stimulation through its dendrites, it will "fire" a signal through its axon. These neurons are then arranged in networks that have some inputs and one or more outputs.

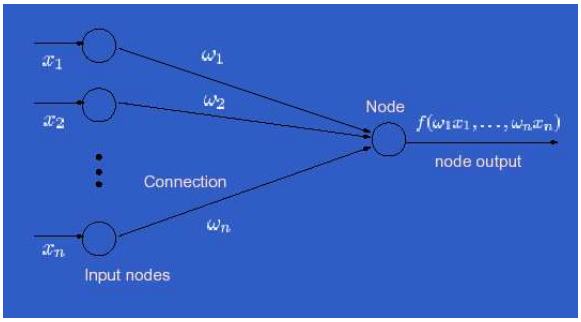

Figure 2.3: Illustration of a sigmoid neural net model. Each neuron has several inputs and a weight associated to each of them. A sigmoid transfer function defines the output. Figure taken from [10].

There are several different variations of neural networks, one very common model is a network with sigmoid neurons[1] arranged in a feed-forward structure (see figure 2.3). More complicated structures exist, with recurrent patterns and other neuron types.

A neural network can learn either by employing a learning algorithm (such as the popular back-propagation algorithm) or by being constantly reconfigured by an evolutionary algorithm.

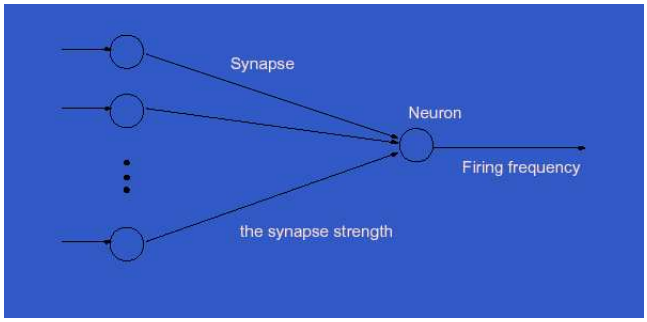

Figure 2.4: Illustration of a spiking neural net model. A neuron outputs a frequency of spikes, depending on the strength of its inputs. Figure taken from [10].

We have chosen to concentrate on the so-called spiking neural network model (see figure 2.4). This model is quite new, and uses temporally encoded spikes instead of continuous real values as outputs from the neurons. It is somewhat inspired of biological neurons which also send spike trains. These networks are suitable for hardware implementations since the spikes can be boolean - either on or off, and the internal membrane potential can be modelled without using multiplication circuits.

2.5 Mechanical Equipment

This section gives a brief introduction to the parts that was needed to realize this project. These parts would have been too difficult to build ourselves.

DC engine
The standard DC engine consists of a magnet, a long wire and two brushes. The wire is wound around a piece (this is called a coil from now on). The magnet generates a magnet field around this coil, and when a current is applied to the brushes the coil will rotate. This is the very basic way a DC engine is created. See figure 2.5.

[1]The sigmoid transfer function is usually on the form $f(s) = 1/(1 + e^{-s})$, where s is thesum of the inputs.

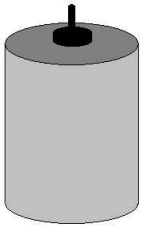

Figure 2.5: DC engine

Electronic Speed Controller:
The Electronic Speed Controller (ESC) controlls the engine throttle. A normal ESC contains a small microcontroller, some heavy-duty transistors (usually mounted in as an H-Bridge[2]) and some fast diodes for protection [3]. These transistors can handle huge currents, hence they are also very expensive. One effective way to control the engine throttle, without wasting power, is to use Pulse Width Modulation (PWM). PWM-signals is the input of the ESC, and most speed controllers receive a PWM-signals every 20ms, and the pulse width differs from 1 to 2ms. The ESC is connected to a power source. Using transistors mean that the ESC controls when power from the power source is transported to the DC engine.

Propeller:
The propeller generates lift to the system. Scientific definition of a propeller: "In the simplest terms, a propeller is an airfoil traveling in a circle with a positive angle of attack relative to the incoming air to produce thrust". Propeller performance is affected by several factors, among them are diameter relative to RPM (rounds/minute), and blade area relative to power absorption and pitch. The normal RC-propeller has two parameters, diameter and pitch and the are usually written diameter times pitch. Diameter is the measurement (usually inches) of the propeller from tip to tip and pitch is defined as the theoretical advancement of a propeller in one revolution (usually inches) and defines the speed and maneuverability characteristics of flying. It is obvious that larger propeller and higher pitch increases the load on the engine, and decreases the RPM. See figure 2.6 for a picture of a typical propeller. Pitch and RPM characteristics:

- High pitch provides good thrust at high speeds.
- Low pitch provides good thrust at low speeds.
- High RPM for high speeds.

[2]H-Bridge is a transistor-constuction that is able to change the direction of the current, thereby rotating the propeller both ways.

[3]When a transistor cuts the current transferred to the engine, a huge negative current is induced from the coil. The transistors need to be protected againt these currents or else they will be destoyed.

Figure 2.6: Propeller

See [13] for additional information about rc-propellers.

Propeller mounting:
A part for attaching the propeller onto the DC engine's drive pin. There are several designs, where some are squeezed to the drive pin, while others are attached with screws.

2.6 Height Measurement

A Height Measuring Sensor will be used to give the system fitness feedback. There are several ways to measure the height:

- Optical mouse. Using an optical mouse will accurately measure the movement of the helicopter, but there are some difficulties with how to attach the mouse and what kind of surface the optical sensor shall read from. An optical mouse could be interfaced through the PS2 or USB interface, and it would be easy to get a sample of the movement.
- Serial mouse. The early mouse design gives the possibility for easy serial read-out. The difficulties are similiar to the optical mouse or even more difficult.
- Infrared distance sensor. There are loads of products on the marked, but most of them are quite large. Sharps GP2Dxx(x) series are small, but the output is analog. The analog signals need to be converted to digital signals, hence the system needs an analog to digital converter (ADC). Sensor read-out and mounting are very easy, but these sensors cannot match the accuracy of mice.
- Air pressure sensor. Not investigated, because of lack of belief in accuracy.

Chapter 3

Mechanical System Design

In this section the mechanical design and the different parts are described. At the beginning of this project, different designs were reviewed. First vehicles moving on the surface were explored. Landmoving vehicles would get stuck, which means that the evolution needs to be supervised. They would also experience difficulties returning to point zero, leading to a more difficult fitness evaluation. Then flying vehicles were taken a look at. Gravity will automatically return the flying machine to the starting point, so height measure could be used as the fitness parameter. The next problem to be solved was what the flying machine should learn by evolution. It was decided that the Evocopter should learn to hover. Hovering only needs vertical movement making it easy to limit the movement. Using metal bars with a stopping mechanism on the top, secures unsupervised evolution.

3.1 Off-the-shelf Components

Engine: There are three obvious choices. A combustion engine, a large DC engine and a small DC engine. The combustion engine idea was quickly disregarded, due to exhaust problems since our model is for indoor usage only. Choosing one of the two other options is more difficult.

- The small engine was a Graupner 4:1 geared (model: 1718) engine just weighing 97 grams and, because it was geared, was able to drive a large propeller. It was indicated in the shop that this engine with a large propeller was able to lift approximately 500 grams, which was too little for our future expansions. The salesman was not sure about the lift and could not guarantee take-off with additional weight. The engine with gear costs approximately 700 nok.
- The large engine alternative was an AXI (figure 3.3) 4120-18 engine. The

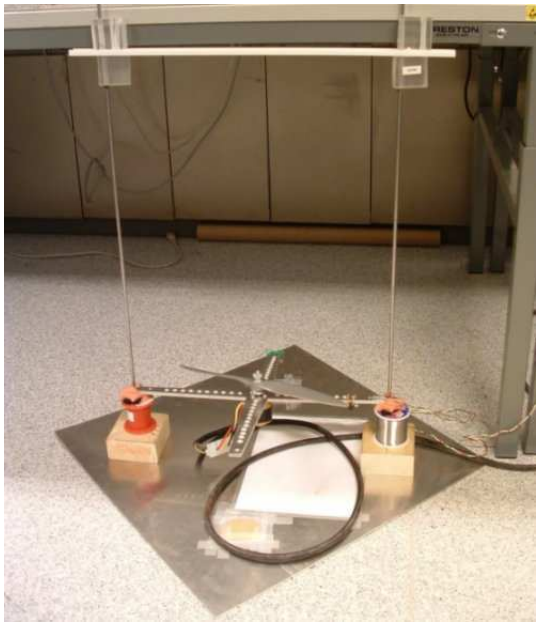

Figure 3.1: Mechanical overview, front view

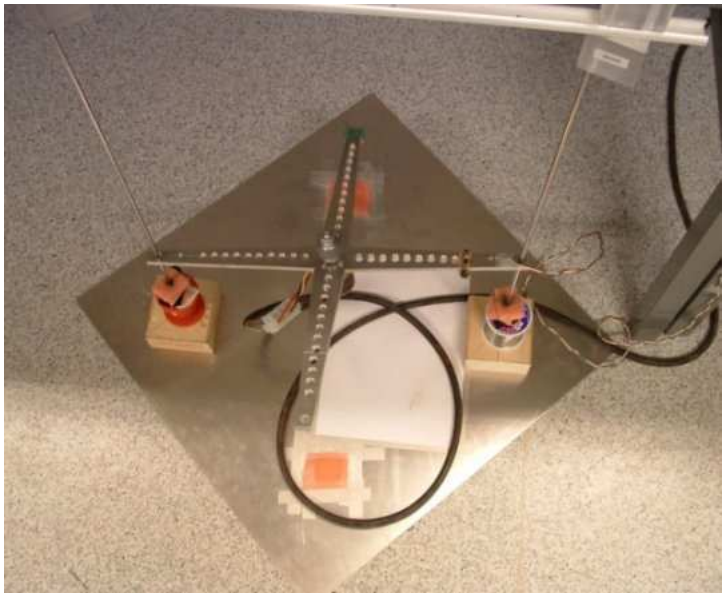

Figure 3.2: Mechanical overview, top view

specialist was sure that this engine would be able to lift over 2 kg, which is sufficient for our set-up and gives us the opportunity to have lots of future expansions. This engine is obviously more expensive than the smaller Graupner, and costs about 1600 nok. It also requires a more powerful Electronic Speed Controller (ESC).

Figure 3.3: The AXI/Model motors DC-engine

Electronic Speed Controller
The choice of ESC is based on the engine choice. Hence a more powerful engine requires a more powerful ESC and the cost of ESCs increase exponentially. A 300 nok ESC was enough to handle the Graupner, but the AXI motor requires a 1700 nok ESC (figure 3.4).

Figure 3.4: The AXI/Model motors ine

Propeller:
The choice of propeller is a science, but it is not the most expensive post in the budget. Since this is a very difficult task, the experts in the shop made this choice for us.

3.2 Custom Handmade Parts

3.2.1 Bottom plate

The whole design will be mounted on top of the bottom plate. See figure 3.5 for design. The bottomplate is $60*60cm^2$, and it was cut from a 2 mm thick aluminium plate.

Table 3.1: Off-the-shelf component choices

Component	Choice	Price
DC-engine	AXI/Model Motors 4120-18	1650NOK
Speed Controller	Model Motors 7524-3	1750NOK
Propeller	APC Propellers Composite 15"x8"	200NOK
Propeller attachment	Squeeze to drivepin	150NOK
Sum(NOK)		3750NOK

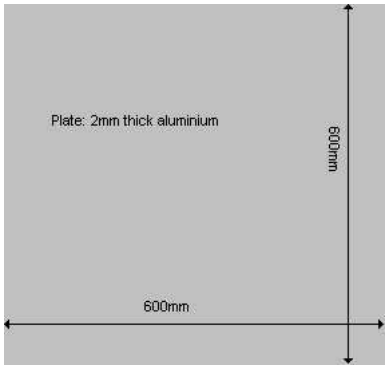

Figure 3.5: Bottom plate

3.2.2 Engine Mounting Plate

The engine needs to be mounted robustly. There are 4 screws mounting the engine onto this part. This part needs to be able to slide up and down the rods. The original custom design was like figure 3.6 and like the bottom plate cut from a 2mm thick aluminium plate. The first design was not robust enough to handle the forces, so it was strengthened with fibre glass tape. It seemed that the tape had done the trick, but after a lot of testing, the plate was once again bent. It was then decided that a new thicker design should be made. The second design (figure 3.7) was cut from a 6mm aluminium plate, which should be more than strong enough. This design was very stiff and strong, but needed almost full throttle to take-off. After consultance with a material-strength expert, it was decided to drill many 10mm holes placed in the middle of the 4 wings. The drilling proved to be a success, but the construction is still far from optimal. We tried to make some acryl bricks to stabilize the helicopter and improve the sliding, but these bricks didn't work as planned. They got stuck and would make the evolution very hard. The holes in the acryl should be drilled narrower than the sliding rods, and then drilled again with a special drill bit to make them the exact size. Since the sliding rods were not completely straight, this solution would not improve the construction at all. If straight rods were acquired, this method should be used. Check figure 3.8 to see

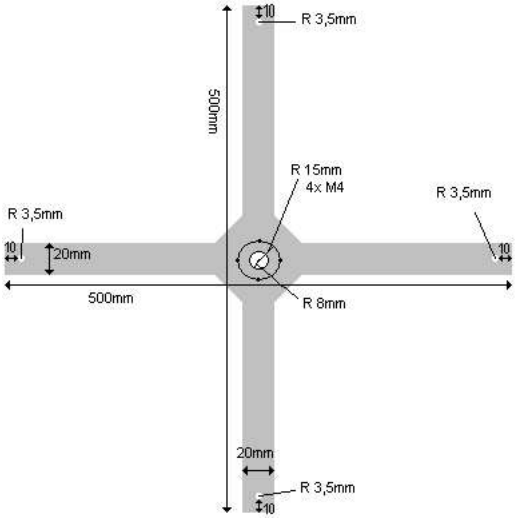

Figure 3.6: Engine mounting plate

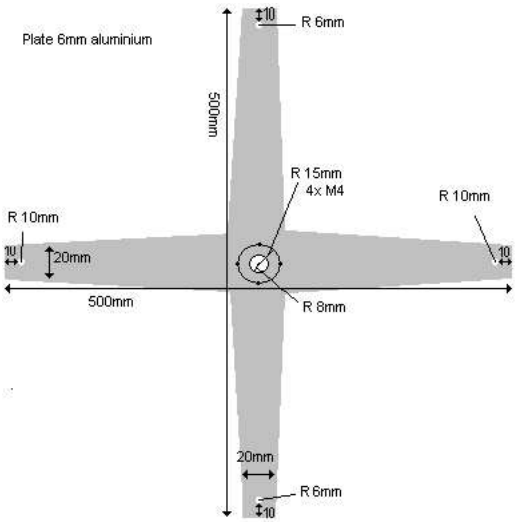

Figure 3.7: Engine mounting plate mk2

some band saw action.

Figure 3.8: Band saw action

3.2.3 Sliding rod

The sliding rods provide moveability just along the vertical axis. Other movement is not desired . See figure 3.9 for technical spesification of the rods. In addition to

Figure 3.9: Rod

these parts, some parts are necessary for the protection of the propeller. These parts should be as small as possible, because the more area covered around the propeller, the more the lift is decreased.

Chapter 4

Electronic System Design

This chapter looks at the electronic design. How the whole electronic system is designed, what components are used and how the they are connected. See figure 4.1 for an overview of the electric design. The distance sensor returns analog voltage values corresponding to the measured height. The sensor gets power from the Field Programmable Gate Array (FPGA) board, and the signal wire is connected to the microcontroller (MCU). The microcontroller contains an analog to digital converter (ADC), that converts the sensor input to a digital signal. This signal is sent to the neural net on the FPGA. The propeller's rotation velocity can be controlled manually by the buttons or by the neural net. The Pulse Width Modulation(PWM) Generator receives values either from the buttons or the neural net and generates signals that are accepted by the Electronic Speed Controller(ESC). These signals are so-called PWM signals. In addition to the neural net the FPGA contains an interface module that connects the FPGA to a personal computer(PC). The evolution of the neural net is controlled by the genetic algorithm located on a PC. The genetic algorithm produces individuals (neural net configurations), and these are sent using a UART to the FPGA interface module. The PC receives sensor values from the interface module, and these are used for calculating the fitness. The different components will now be looked at more comprehensively.

4.1 Distance Sensor

To measure the fitness of the neural network, a device for height measuring is needed. The distance sensor is a Sharp GP2D12 which is able to detect distances from 10-80 cm. The distance sensor has 3 pins: Voltage(around 5V), Ground and analog output. The analog output returns a value corresponding to the distance measured. There is no need for additional external control circuit. See figure 4.2 for the analog output value for a given distance to the reflective object. As seen on

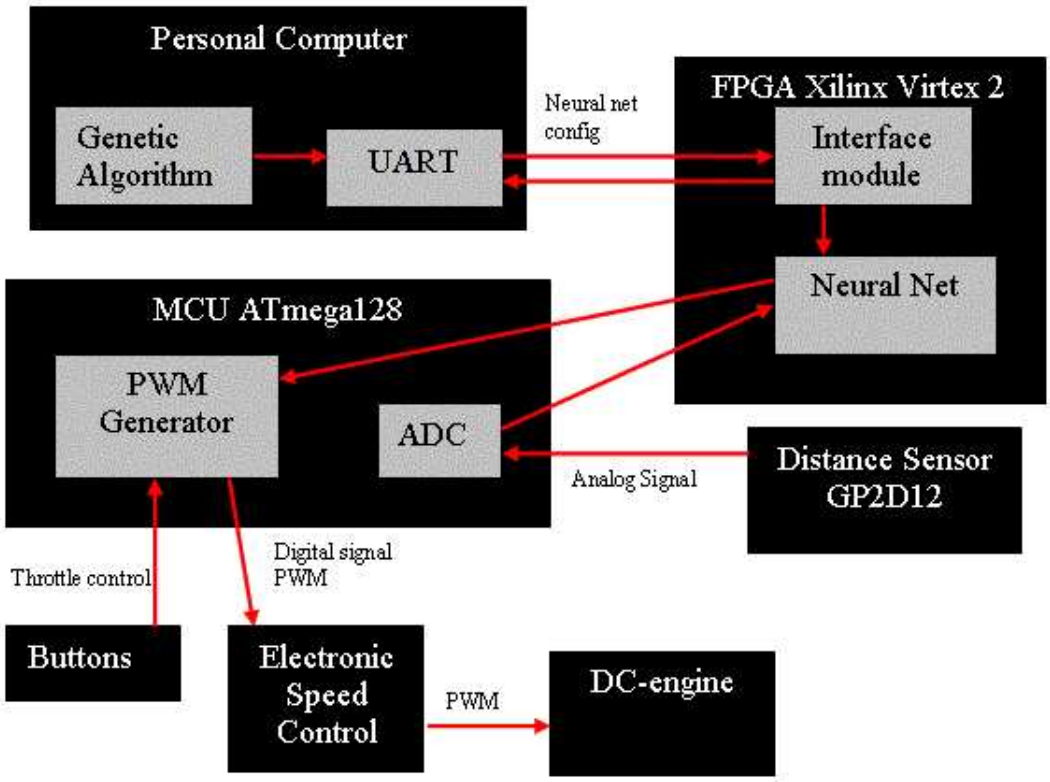

Figure 4.1: Electric overview

Table 4.1: Translation values exponential to linear

distance(cm)	sensor output	linear output
0		
10	2,60	2,6
20	1,40	2,23
30	1,00	1,86
40	0,80	1,49
50	0,60	1,12
60	0,50	0,75
70	0,45	0,38
80	0,40	0,01

the graph 4.2 the sensor must be placed at least 10 cm from the reflective object to return accurate and correct results. Figure 4.2 shows that the output value of

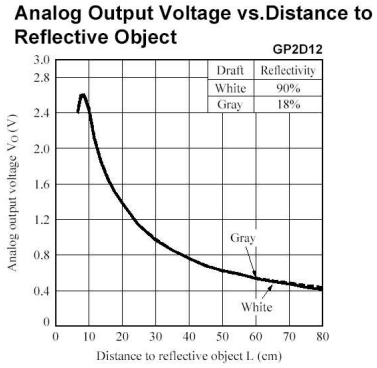

Figure 4.2: Analog output voltage vs distance to reflective object

the distance sensor is not linear. Therefore the output values of the sensor needs to be linearized. Table 4.1 shows corresponding linear values to the exponential values. The linearizer gets the sampled value from the ADC and compares it with the exiting exponential points. When a suitable point is found, the correct value is found by following the interpolated line between the two linear points. Figure 4.3 shows the resulting linear graph, which is approximately $out_{lin}(dis_{SAM}) = (-max_{VCC}/dis_{VCC=0}) * dis_{SAM} + max_{VCC}$, where max_{VCC} is the maximum value from the sensor (2,6V), $dis_{VCC=0}$ is the maximum distance measured, and dis_{SAM} is the translated measured signal [1].

[1] dis_{SAM} in interval 10-80cm

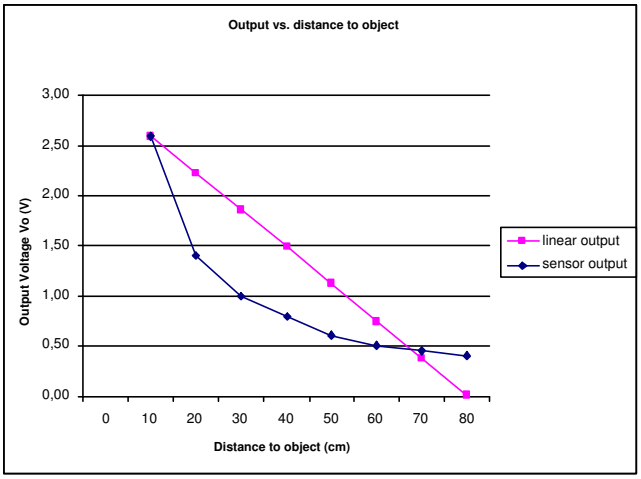

Figure 4.3: Translation of exponential to linear graph

4.2 Electronic Speed Controller

The ESC used is a Model Motors 7524-3 (Figure 3.4)[2]. The Speed Controller is programmable at start-up and minimum and maximum pulse width is defined. Minimum pulse width means minimum throttle and maximum pulse width means full throttle. The ESC receives pulses from the PWM-generator in the Atmel Atmega 128. These pulses arrive every 22,2ms and last from 1,15ms to 1,9ms. See figure 4.4 to see the correspondence between trasistor on time and throttle. Slimmer

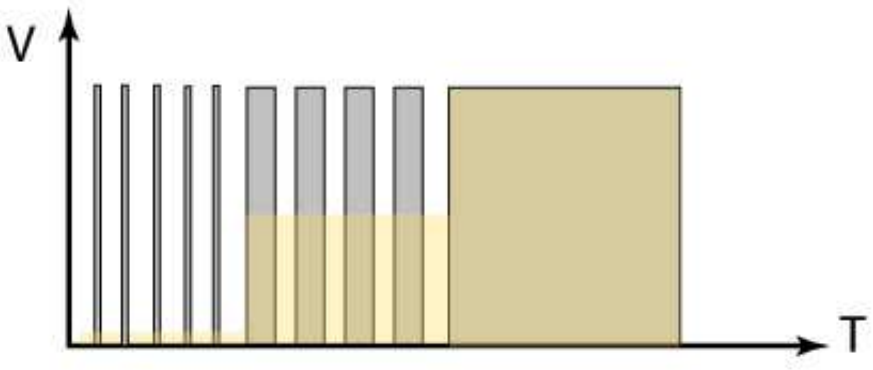

Figure 4.4: Transistor on time vs throttle

[2]The ESC is spesified for use with 7 to 24 batterycells, meaning that the operating voltage is from 8,4 to 28,8V. The ESC is able to deliver 75 A over time. See [12] for further details on the ESC.

pulse width gives less engine speed, and wider pulses mean higher throttle. The pulse width can also be manually controlled by the input button on the STK500 development card.

4.3 Microcontroller - Atmel ATmega128

The analog voltage signal from the distance sensor needs to be converted into a digital signal. The ATmega128 microcontroller (MCU) from Atmel provides a built-in Analog to Digital Converter (ADC) of the succesive approximation type.

4.3.1 Motor control

To send the right pulses to the ESC, correct timing is needed. A simple approach has been used: custom wait functions have been implemented that are suited to the resolution we need for the pulses. The functions are implemented as simple for loops with dummy operations. This gives good accuracy as long as the clock frequency of the microcontroller is not changed. See main.c under AVR code in the appendix for details.

4.3.2 Analog to Digital Converter

The built-in ADC has 10-bit resolution, which means that it is able to represent 1024 (2^{10}) discrete digital values corresponding to the analog input value. The conversion time of the ADC is also sufficiently low ($65 - 250\mu s$) and the ADC is connected to an 8-channel Analog Multiplexer which allows 8 single-ended voltage inputs to be read from Port F. This basically means that 8 distance sensors can be used in the future. The single-ended voltage inputs refer to 0V (GND). The ADC accept values from Ground to VCC (0-5V) and the voltage reference is adjustable. The ADC can also be run in different modes, namely free-running- and single conversion mode. Free-running-mode means that a new conversion starts when the previous one finishes, while the second mode is controlled by interrupts. The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion.

4.3.3 Pulse Width Modulation Generator

The voltage over the DC-engine is adjusted by Pulse Width Modulated-signals[3]. The rotation velocity of the engine is decided by amount of time the transistors

[3] PWM-signals are also used for light-dimmers.

is switched on. These transistors are located in the Electronic Speed Controller. PWM Generator sends pulses at a certain frequency to the ESC. The width of the pulses decides how long the transistors in the ESC will be switched on. See the chapter about the Electroninc Speed Controller for more details on the PWM frequency and width of the accepted signals.

4.3.4 Buttons

The input buttons on the Atmel STK500 card are used to manually control the signal sent to the ESC. The buttons have the following functions:

- Button 0: Engine off (Pulse width 1,15ms)
- Button 1: Low pulse width
- Button 2: Medium pulse width
- Button 3: Full throttle (Pulse width 1,9ms)

These buttons are only used in the testing phase.

4.3.5 LEDS

The Atmel STK500 card has eigth LEDs (light-emitting diodes). These LEDs shows the throttle output from the neural net. If LED 0 emits light the neural net outputs no throttle. The higher LED number the more throttle.

4.4 Field Programmable Gate Array - Xilinx Virtex 2

The FPGA used is a Xilinx Virtex 2 1000, containing 11520 logic cells, equivalent to one million gates. It is located on a Memec Design Virtex2 MB1000 development board. The board features 32MB DDR SRAM, a serial port, two 7-segment leds, one user led and a P160 expansion port. The FPGA contains a UART for serial communication and the neural net.

4.4.1 Interface module

A UART is implemented in the FPGA design. This enables RS232 serial communications with a PC.

4.4.2 Neural Network

The neural net basically consists of neurons and connections. The number of neurons must be decided before the FPGA is configured. There exists connections between every neuron, but which of these connections that are active is controlled by the bit string from the UART.

4.4.3 P160 Prototype module

A Memec prototyping module is connected to the development board using the P160 expansion port. It contains a prototype area with plenty of connectors. The 40-pin connector (J6) is used for communication with the microcontroller. Pins J6-31:38 (J6 connector, pins 31 to 38) are used for sensor input and pins J6-21:28 (J6 connector, pins 21 to 28) are the output byte from the neural net.

Chapter 5

Neural Net

This chapter looks at the neural net implementation.

5.1 Neural network design

This section takes a closer look at how the neural net was designed.

5.1.1 Neuron architecture

Each neuron has a set of inputs, called dendrites. Each dendrite is a source of either positive(exitatory) or negative(inhibitory) spikes. The neuron has one ouput, called the axon. The neuron can emit spikes through the axon, these are either positive or negative, depending on the configuration of the neuron.

In order to enable fast reconfiguration of the neural network, each neuron is designed to be totipotent[1]. This means that in hardware, each neuron is connected to all inputs, all other neurons and has all possible functionalities that a neuron can have(in our case it can be both an emitter of positive spikes and an emitter of negative spikes). The neuron can then be subject to a "soft" reconfiguration via a configuration input. This can be done through custom data interface and no reconfiguration of the actual FPGA is needed.

Spike model

Different spike models have been studied [3],[2],[1]. Our model is heavily inspired by [1] and is very simple. It is kept simple partly because of hardware considerations (no multiplications or exponential calculations), partly because of simplicity

of implementation but also because it has been proven earlier that this simple model can work[5].

The model is based on discrete time steps. The neuron functionality is based on a so-called mebrane potential, inspired by the membrane potential observed in natural neural cells. The neuron can be seen as a state machine, see figure 5.1.

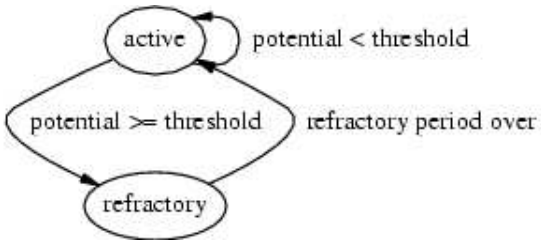

Figure 5.1: Neuron states.

Active state

- When the neuron is in the active state, it adds all of its inputs to the internal membrane potential. The potential is reset if it becomes lower than the resting value. See figure 5.2.
- If the membrane potential is above a threshold, the neuron fires a spike. Then the neuron enters the refractory period and the potential is reset to its resting value.
- When no firings occur, potential leakage is applied. The membrane potential is decreased by a small value.

Refractory state

- When the neuron is in the refractory state, incoming spikes do not affect its membrane potential. The neuron cannot fire. See figure 5.2.
- After a short time, the neuron re-enters the active state.

In our experiments, each incoming spike from other neurons counts +/- 2 potential units. Incoming spikes from sensors count +6 potential units. The threshold used is +8. Leakage for each time step is -1.

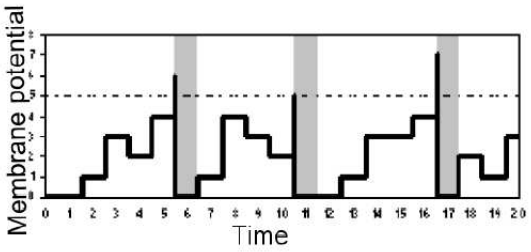

Figure 5.2: Membrane potential over time. Grey columns indicate refractory periods. figure taken from [1]

5.1.2 Network architecture

In our implementation, each neuron can be connected to every other neuron in the network. There are no specific layers or feed-forward structure. This structure also allows for recurrent topologies.[1] See figure 5.3 for an example of a network topology. As the number of neurons get larger, other connection methods should be

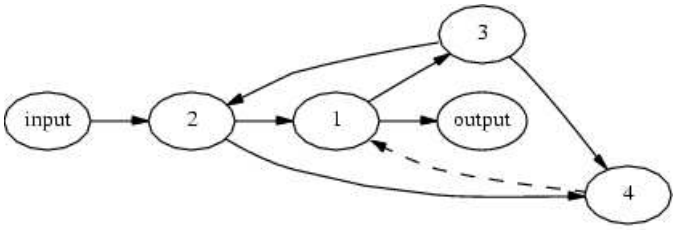

Figure 5.3: Simple example neural net. Recurrent connections are allowed. The dashed line represents an inhibitory connection.

considered. The number of wires used for connections will get very large. As each neuron now requires $N_{neurons} + 2$ input connections, this will at maximum require $N^2 + 2N$ connection wires so this is not very scalable. Two possible solutions have been considered, which could be looked into in future work. The first and most obvious solution is to restrict the connection capabilities of each neuron. By only letting a neuron connect to a certain sized neighbourhood of say, 8 or 10 neurons, it would be possible to do with much less connection wires and also a smaller configuration input for each neuron. However, this could require more neurons to achieve desired functionality. Another solution which could be considered is a kind of event-based solution. In this design each neuron would have an address and spike events would be sent out on one or more common "spike buses." This

[1] No delay units are introduced though, as often seen in recurrent sigmoid NNs.

could exploit the fact that spikes are sent at irregular intervals and most dendrites and axons are more "off" than "on." However, one could risk reducing the level of parallellism.

5.2 Neural network hardware implementation

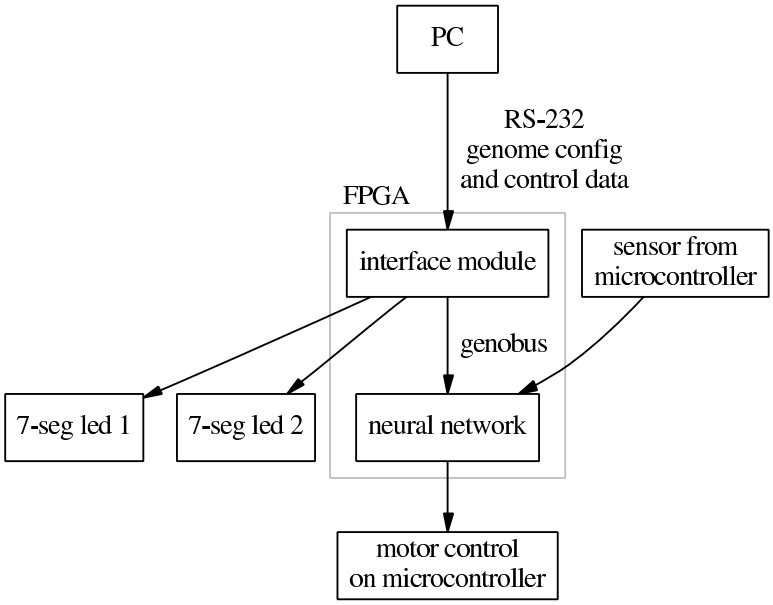

Figure 5.4: Elements in the FPGA design and their connections with the rest of the system.

The neural network has been implemented in VHDL code, with the Virtex2-1000 FPGA as the target device. The source code can be found in the appendix, section VHDL. See figure 5.4 for an overview of the hardware design.

The FPGA receives genome data and control signals through the RS-232 cable from the PC. The interface module contains a UART and is responsible for this communication, but passes the data on to the neural network through the internal Genobus. There are also two 7-segment LEDs connected for debug output. The neural network receives a sensor signal directly from the microcontroller, and sends its output value out directly to the microcontroller.

Table 5.1: Interface registers

address	register name	read/write
0x00	aux	r/w
0x01	control	r/w
0x02	sensor	r

5.2.1 Communications interface

The communications interface consists of a UART for RS-232 serial communications, a register for sensor data and some control logic. It is connected to the internal neural net via the Genobus, a simple one-way data bus. An overview of the registers addressable in the interface module can be seen in table 5.2.1. However, the aux and control registers are unused at the moment. It is planned that the interface module in the future should be able to handle burst modes through the RS-232 connection, for faster configuration. The sensor register is read-only, and contains the value of the sensor input to the fpga. This is read by the PC and used for fitness evaluation. Address values are 7-bit, the MSB of the address byte controls wether a read or write operation is to be performed. By setting the MSB of the address byte to 1, a read operation is requested. A byte of data will be returned to the PC through the RS-232 connection.

RS-232 Serial interface

A UART core was found on OpenCores [11]. This core has been used and modified slightly. At the moment a speed of 2400 baud is used, but this can be increased when higher transfer rates are required.

Genobus

The genobus is a simple 16 bit wide, one-way data bus. 8 bits are used for data, 8 bits are used for address. One extra wire is used for signalling that data is available on the bus. The data is only present for one clock cycle. This requires the listening device to be ready. The design will have to be extended if readback from the neural network is to be implemented.

5.2.2 Neural network

The neural net module contains the neurons, their configuration and logic for configuring them. It is connected to the outside world via the Genobus. (Signals with

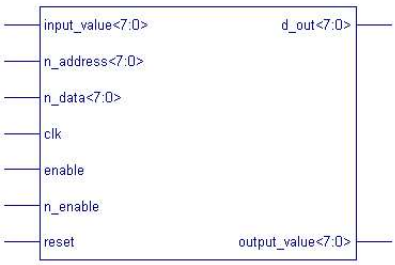

Figure 5.5: Schematic symbol of the neural network module. Signals with prefix n_ belongs to the Genobus. input_value is the sensor input and output_value is the activity output. d_out is a debug signal.

Table 5.2: Neural net registers

address	**register name**
0x10	control
0x11	init config
0x12	conf stream
0x13	end config

prefix n_ on figure 5.5.) The sensor input and the activity output are directly connected. 8 neuron modules are instantiated, where each neuron is connected to all other neurons' axon outputs. Internal registers hold the configuration values for each neuron. Before activation, the net will have to be configured through the Genobus. As a simple solution the genome is divided into bytes of data on the PC and sent on the bus, then, after arrival in the neural net module, shifted in to the configuration registers.

There are some control registers that are available through the genobus. See table 5.2.2 for an overview. Registers 0x11-0x13 are used for configuration. First write any value to address 0x11, then configure the net by sending all bytes in the genome to address 0x12. Finally, send any value to 0x13. The control register commands the network behaviour. See table 5.2.2 for an overview. The soft reset commands the network to reset neuron potentials etc., but not the configuration values.

The neural network module also contains an instance of a spike generator module, as well as an activity analyzer. The spike generator input is connected to the sensor input value, while the activity analyzer output value is connected to the activity output. Any neuron can connect to the spike generator by its configuration input. The activity analyzer is hardwired to connect to the output of neuron 0.

Table 5.3: Control register values

register value	function
0x00	stop network
0x01	start network
0x02	soft reset network

5.2.3 Neuron module

Figure 5.6: Schematic symbol of a neuron module.

Each neuron takes $N_{neurons} + N_{inputs}$ dendrites as input. Its output is an axon. See figure 5.6. The dendrites consist of two wires each, one for the spike and one for the sign of the spike. The config input consists of $N_{neurons} + N_{inputs} + 1$ bits. The MSB tells which sign the axon spikes will have. The following N_{inputs} bits are the dendrite weights for the sensor input signals. The resting bits are dendrite weights for signals from other neurons.

For one time step of the spiking neuron model, all the incoming spikes will have to be added to the potential. This is solved by dividing this time step into $N_{neurons}+1$ smaller time steps. These are timed by the clock signal and will therefore be refered to as clock cycles, while the neuron model time step will be referred to as a neuron cycle. For each clock cycle one of the incoming dendrites is processed. If the dendrite signal is asserted the membrane potential will be increased or decreased by 2. At the last clock cycle, the potential is compared to the threshold value. If it is above the threshold, the neuron asserts its axon and enters the refractory state. If the threshold is not reached, leakage is applied by decreasing the membrane potential. Finally, if the potential should be below the resting value, it is reset. While in the refractory state, the axon spike is kept asserted. This state also lasts for one neuron cycle. See figure 5.7.

FPGA Area matters

So far the neuron code has not been optimized for size. Effort has only been made to implement the functionality. As can be seen in the synthesis report for one

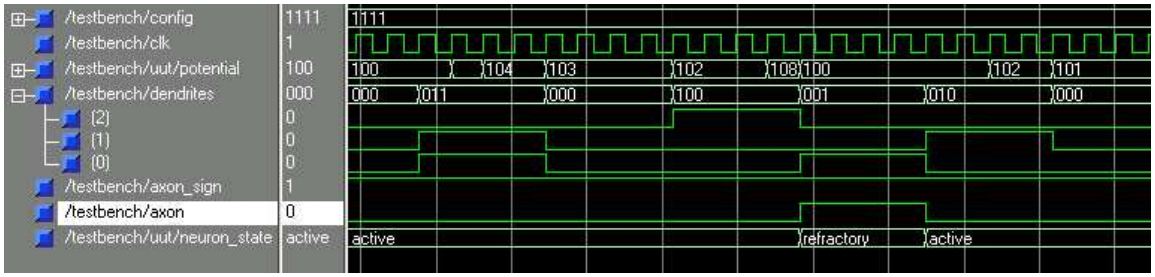

Figure 5.7: Simulation of a neuron. Dendrites 0 and 1 come from other neurons, dendrite 2 is from a sensor input and therefore weighs more. The threshold in this example is 106. Notice how the neuron enters a refractory state after firing and rests unaffected by incoming spikes.

neuron:

```
Number of Slices:                         63  out of   5120     1%
Number of Slice Flip Flops:               18  out of  10240     0%
Number of 4 input LUTs:                  116  out of  10240     1%
```

This would intuitively indicate that it could be possible to scale the design up to 100 neurons. More effort will probably be applied by the synthesis program to optimize area if the design gets large. One would also need to consider the extra resources used by the interface module. The biggest problem is that as the number of neurons increase, the number of connections per neuron will increase by the square of the number of neurons. So in order to get good scalability one would neet to implement another connection scheme, like discussed in 5.1.2.

5.2.4 Spike generator

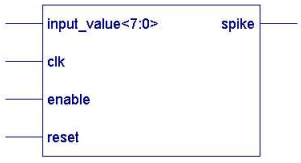

Figure 5.8: Schematic symbol of the spike generator module.

The spike generator takes an 8-bit binary encoded input value (the sensor input value in this case) and generates a train of spikes as output. See figure 5.8. The spikes are always positive, ie. they have no sign. It is implemented by the use of

a 9-bit counter, where the input value is added for each neuron cycle. When the MSB bit rises, a spike is emitted. The counter is not cleared. This can be seen as a 1.8 fixed point counter where a spike is emitted when a value of 1 is reached. This gives a train of spikes with a frequency corresponding to the input value. See figure 5.9.

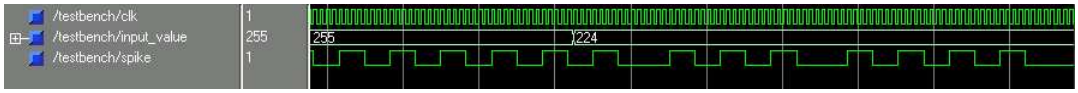

Figure 5.9: Simulation of the spike generator. Notice how the frequency of spikes decreases when the input value is decreased.

5.2.5 Activity Analyzer

Figure 5.10: Schematic symbol of the activity analyzer module.

The activity analyzer can be seen as an inversed spike generator. It takes a spike train input and outputs a binary encoded value. See figure 5.10. The higher the spike input frequency, the higher the output. It is implemented by counting the number of spikes received over a given period of time. The output resolution is therefore dependent of the length of this time interval. We used an output resolution of 8 bits. This requires 512 neuron cycles for a measure, since spikes can at maximum be fired every other neuron cycle (due to the refractory period).

This is no problem for our design running at 24MHz and using 8 neurons. One neuron cycle takes 8+1=9 clock cycles. The number of clock cycles required for a measure is then 512*9=4608. A cycle has a period of 42ns so the whole measure only takes 0.2ms. This scales linearly with the number of nodes in the network and therefore quite a few more nodes can be added while maitnaining a good refresh rate.

Chapter 6

Evolution

6.1 Genetic Algorithm

Our implementation of the GA is straightforward. Fitness-proportionate selection is used. The genome is directly encoded, no development is used. Standard genome operators are used. The probabilities and quantities for each of the following features are adjustable.

6.1.1 Fitness Evaluation

The fitness evaluator measures how the current neural net configuration performs. The number of measurements and the time between each measurement is decided before start-up. The fitness is calculated the following way:

$$fitness = \sum_{i=0}^{Measures}(GoalValue - height(i))^2/(Measureperiod * Measures).$$

The fitness function calculates the difference between the goal value and the measure height for each measure. The difference is squared to increase the sensitivity, providing better fitness for configurations that hovers than for ustable ones. Divinding by the product of the time between each measure and the number of measures (measure period times number of measures is the total time for a sample). Divinding by this product means that the fitness function always returns values in the same interval when measure period and number of measures. The use of difference from goal value means that the lower fitness value the better. The use of dynamic fitness evaluation was also considered. This would break the evaluation quite early for individuals who would not manage to take off. Others, who had a tendency to hover, would get extended evaluation time. This would reduce evaluation time and therefore speed up evolution.

6.1.2 Selection pressure

No advanced selection pressure functions are used. There are two types that can be chosen as evaluation parameters:

- No pressure. $Expval_{ind} = fitness_{max,glob} - fitness_{ind}$. This computes the expected value from the fitness simply by taking the global maximum fitness value and subtracting the individual's fitness. This gives no pressure, only a way to convert fitness values to expected values, which start at zero for worst. High value of expval is good.
- Simple pressure. $Expval_{ind} = fitness_{max,pop} - fitness_{ind}$. This function is slightly more advanced, always taking into account the curerent generation's worst fitness and adjusting so that the worst individual in one generation always gets an expected value of zero.

6.1.3 Genome operators

During selection of a new population, elitism, crossover, mutation and the creation of new random individuals are used. Elitism is straightforward, just a copy of the best individual(s) to the next generation. No mutation is applied to the elite individuals.

Crossover

Crossover is done directly on the bit string genome. Two crossover methods are available: One-point and two-point. In addition another variation can be applied to crossover: The crossover point(s) selected are only on the boundaries of single neurons in the genome, for instance crossover points are only selected for every 10 bits. By default, crossover points are selected anywhere in the genome. Crossover produces two children, if there is enough space in the destination population. Otherwise, only one child is generated. Mutation is then applied to the children.

Copy

The individuals who are not selected for crossover are copied to the new generation. Mutation is applied.

Mutation

Mutation is done on the genome by using a per-bit probability of a bit flip.

Parameter	Value
Neurons	8
Population size	30
Measure time	7000 ms
Pressure method	No pressure
Crossover prob.	0.9
Crossover type	Two-point
Mutation prob.	0.01
Elite size	1
New individuals	1
Power	IMPO PSU @ 20V

Table 6.1: GA parameters for the first run.

New individuals

A number of new random individuals are added to the population.

6.2 Evolution runs

After the system was completed, it was time to test it through evolution experiments. The results of these experiments and some discussion is presented in this section.

6.2.1 First run

We started out with the Grefenstette parameters[7] (see table 6.1) which are popular for evolution runs where fitness evaluation takes msuch time. We also used the DC power supply, set to 20V. After having started out okay, in the second generation, all individuals seemed to be unable to take off. The power supply did not deliver enough current and therefore further evolution was impossible. We tried to turn off the power supply for a while, and restart. This worked somewhat, sometimes we were able to finish one more generation. But we had to find a better solution.

6.2.2 Second run

This run has been very inconsistent, with changes in power supply and evolution parameters. The goal has been to try out different parameters to find out what to use for later runs. At generation 0, we started out with parameters as shown in table 6.2. As can be seen on figure 6.1, the evolution process did not produce

Parameter	Value
Neurons	8
Population size	30
Measure time	7000 ms
Pressure method	Simple
Crossover prob.	0.9
Crossover type	Two-point
Mutation prob.	0.01
Elite size	1
New individuals	1
Power	24V Lead acid accum.

Table 6.2: Starting GA parameters for the second run.

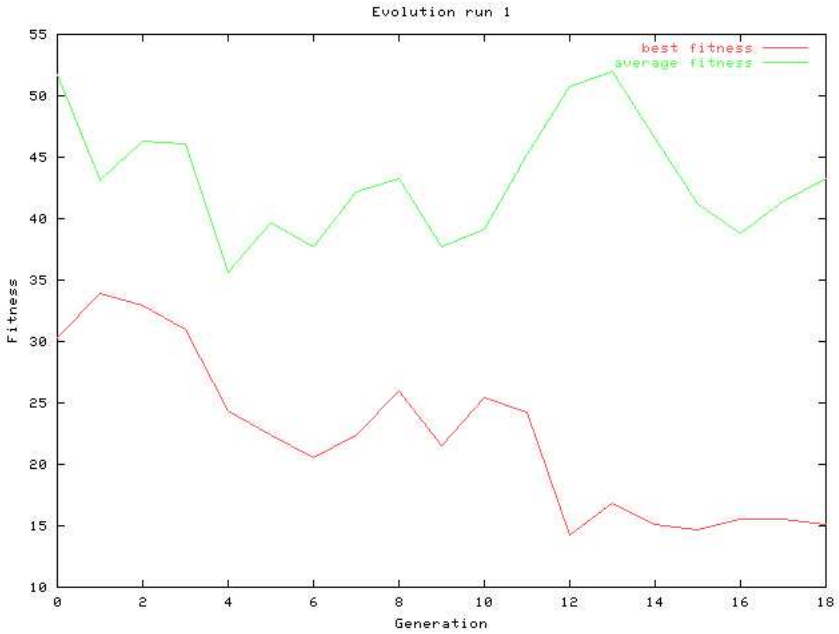

Figure 6.1: Second evolution run.

Parameter	Value
Neurons	8
Population size	30
Measure time	10000 ms
Pressure method	Simple
Crossover prob.	0.8
Crossover type	Two-point
Mutation prob.	0.02
Elite size	2
New individuals	1
Power	24V Lead acid accum. plus PSU

Table 6.3: Starting GA parameters for the third run.

a very stable improvement on the fitness curves. Although the best fitness at the last generation improves from the fitness at the beginning, it oscillates quite much during the generations, even if elitism is used. This is due to the fact that the output voltage from the accumulators sank gradually during the evolution run. A certain output from the neural net will not give the same amount of boost when the supply voltage varies. The max speed limit in the microcontroller had to be adjusted up to compensate for this. But fitness had already dropped so it took some time to get a good fitness again. An elite size of 2 was introduced, and we also tried to introduce more new individuals. This can be one of the causes for little improvement on population average fitness. Mutation rate was also slightly increased, to 0.02, while crossover rate was reduced to 0.08.

6.2.3 Third run

This experiment was more stable, although we did change some parameters during evolution here as well. We started with parameters as in table 6.3. The main changes in this experiment were a new kind of power supply and some changes in fitness evaluation. This run proved to be more successful, mainly due to the new and improved power supply method of combining the two previous ones. See 6.2.4 for more details and a discussion. We also discovered that an individual's fitness value was very unstable, the same individual could have a completely different fitness in the next generation. We therefore introduced, from generation 11, an addition to the fitness evaluation: Fitness is tested two times and averaged for each individual, if it is better than a certain value in the first place. Poor individuals did not submit this treatment in order to save time. The goal height for the device was also slightly lowered, and evaluation time was risen to 10 seconds to see if the individuals could get more stable. From generation 11 new individuals were no longer introduced in the population. As seen from the graph 6.2, population aver-

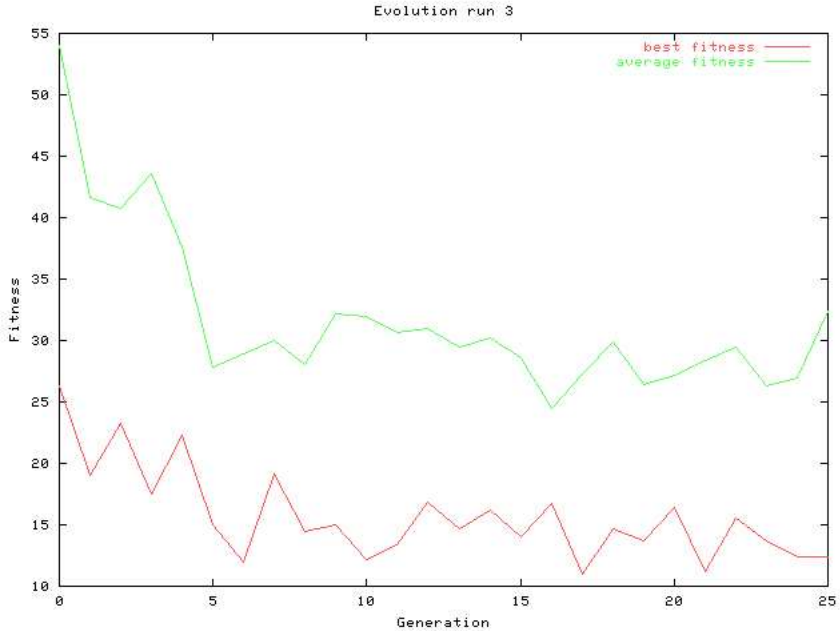

Figure 6.2: Third evolution run.

age fitness is steadily decreasing. The best fitness also decreases, but even if elitism is used, it seems to oscillate quite a bit. This clearly indicates that evaluation of an individual is an unstable process. Probably factors such as power supply, friction, displacement of cables, sensor noise and heating contribute to this uncertainity.

A possible solution could be to introduce even longer evaluation times, and many tests for each individual, preferably not in sequence. Also, many individuals seem to be "jumping" up and down, while still getting a relatively good fitness value. One could consider to introduce an even stronger penalty for such action in the fitness function, even if it is already covered to a certain extent. We could not see any tendencies for improvement after generation 25.

6.2.4 Power difficulties

The evolution runs were very difficult to accomplish due to bad power supply for the engine. The DC power supply (see 7.2.4) could not deliver enough current for the engine during the takeoff phase. As a result of this, the supplied voltage fell and the Evocopter rose more slowly. Even if evolution could have worked with that problem, a more serious problem was that the power supply seemed to go warm after a few evaluations and could not deliver more current than 5A. Takeoff was then impossible and the population fitness sank drastically.

Later, Lead acid accumulators were tried. These could deliver enough current, but the voltage sank gradually during evolution. It started out with almost 26V and sank to 23V after some generations. This made the population worse and worse fit. We hardcoded a max speed limit in the microcontroller so that the net could not order the engine to fly away and damage the setup. This had to be quite low when the voltage supplied was at 26V, but it was proboably too low when the engine only had 23V available. Since this causes many changes in the environment, during the evolution runs, it is difficult to come to a convergence. Evolution will spend time adjusting to the new environment.

The next solution consisted of a combination of the two previous ones. The Accumulators and the power supply were connected in parallell, and the power supply was set to 24V. This way, the accumulators were helping the power supply with delivering enough current when needed, all while the power supply was constantly charging the accumulators. This proved to be a more effective solution. The voltage did not drop more than one or two volts during the heaviest power surges. The next step would be to try to introduce some large capacitors to help cope with supplying current during those surges.

Chapter 7

Tools

This chapter takes a look at the tools used to complete this project, and is divided into software tools and hardware used.

7.1 Software tools

This section gives a brief description of the commercial software used and what we used it for.

7.1.1 Xilinx ISE 5.0 and iMPACT

Xilinx' integrated development environment. ISE was used for managing and synthesizing the hardware design, launching Modelsim and adjusting various parameters. Xilinx iMPACT was used for configuring the FPGA.

7.1.2 Mentor graphics Modelsim SE 5.6

Modelsim was used for logic simulation. It is launched directly from ISE, but some custom .fdo scripts were used to simplify the simulation process.

7.1.3 Atmel AVR studio 4.0 and AVR-GCC

The AVR-GCC compiler was used for compiling the microcontroller C code. Atmel's integrated development environment was used for downloading the program to the microcontroller and to adjust different voltage parameters.

7.1.4 Microsoft Visual C++ 6.0

Microsoft's C++ compiler and integrated development environment. Used for making the GA and control programs on PC.

7.2 Hardware used

This section gives a brief description of the hardware used.

7.2.1 Atmel STK500 + STK501

The STK500 development kit from Atmel was used for our microcontroller setup. The STK501 expansion module was added in order to fit the Atmega128 microcontroller. Programming was done by using a serial cable connected to the PC.

7.2.2 Memec MB1000 + P160 + MultiLinx

The Memec MicroBlaze development kit was used for the FPGA setup. It contains a Xilinx Virtex2-1000 FPGA. The P160 prototyping expansion module was added to make connectors available. Configuration of the fpga was done using a MultiLinx USB cable.

7.2.3 Oscilloscope / Logic Analyzer - Agilent

The logic analyzer is a powerful tool for analyzing digital signals, and was very useful when timing the pwm-signals and checking bus-activity. It also has the functionality of a standard oscilloscope.

7.2.4 Power supply - Impo 11.60

The power supply has adjustable voltage and current. The voltage is adjustable from 0 to 50 volt and the current range from 0 to 10 ampere. The power supply had problems delivering 10 ampere at 20V over time, and could not cope with the dc-engine's enormous peak currents.

7.2.5 Battery Cells

The Evocopter set-up was tried with two battery cells, which is normally used on a motorcycle. One battery was specified to hold 12V and lasting 18Ah. The batteries were connected in series to raise the voltage to 24V.

Chapter 8

Discussion

The following chapter takes a closer look at the mistakes made, the current project status, suggested improvements and what can be done in the future. In the end we give an overall conclusion.

8.1 Design flaws

Fortunately this subchapter is not the biggest one. Every part of the system works to satisfaction, but no single part is optimal and there are huge rooms for improvements. The mechanics are the biggest problem. The sliding up and down the rods is not friction free. This means that the copter does not slide smoothly up/down and it makes the evolution harder. It has a tendency to oscillate and fails to hover smoothly. The Evocopter is not stable sideways either, which leads to the sensor not giving the actual height. This happens because the forces caused by the propeller makes the Evocopter lean to one side, and the sensor is not pointing straight down.

8.2 Status

This section looks at how well the Evocopter fulfilled its task. The overall goal of this project was to build a flying machine, which should learn to fly by the use of evolution. A digital brain (neural net) has been realised in the FPGA and software has been written to control the evolution.

8.2.1 Mechanical

Parts for a basic constuction is built and mounted. The Evocopter is able to fly up and down the sliding rods, while restricted from flying freely. The mechanical construction, although far from optimal, has the proper functionality.

8.2.2 Genetic Algorithm

The genetic algorithm has been implemented successfully on a personal computer using the c++ programming language. The fitness evaluation function has been implemented and tested. The evaluator seems to be working properly. The necessary genome manipulators (elitism, crossover, mutation and new individuals) to construct the next generation of individuals have been realised. They have been implemented successfully, but their parameters need to be fine-tuned, as evolution has not yet produced extremely good individuals.

8.2.3 Neural Net

The neural net has been realised on a Field Programmable Gate Array (FPGA). A spike generator to generate spikes, a given number of neurons and the connections between them (synapses, M:M) have been implemented. One of the neurons is the output of the neural net. The net would benefit from tuning its parameters and structure.

8.2.4 Evocopter Design Incremental Checklist

The incremental milestone fulfillment is summarized in table 8.2.4. As seen in table 8.2.4 all milestones have been reached, but lines marked '*' are not fully reached at the time of writing.

8.3 Suggested improvements

The following improvements are suggested:

Mechanical design

- The friction needs to be reduced. This could be improved by getting very straight rods and smaller holes.
- A propeller protection shed should be added, to ensure safety during evolution and operation.

Table 8.1: Incremental checklist

MILESTONE	FULFILLMENT
Reading the sensor values	OK
Generate PWM signals	OK
Manual control of the propeller using PWMgenerator	OK
Convert sensor output to digital values(ADC)	OK
Linearize the sensor output	OK
Send linearized sensor output to FPGA	OK
Generate spikes corresponding to sensor output	OK
Using GA to change the population	OK*
Sending neural net configurations to the FPGA	OK
Using the neural net output to generate PWM pulses	OK
Evocopter take-off	OK
Evocopter hovering	OK*

Neural network

- At the moment, the network only has one spike source, which comes from the sensor. This makes it too sensitive to the height. Another spike source could probably increase the abilities of the network. This could come from another sensor (for example one pointing upwards), the motor speed or maybe just a constant spike source.
- Increasing the number of neurons should also be considered, possibly in conjunction with an analysis of topologies of evolved networks. This could lead to more advanced functionality and, for our case, improving the throttle control.
- The membrane potential threshold for firing a spike is at the moment fixed. If this could be evolved, neurons would be more flexible and we could probably do with fewer neurons.

Genetic algorithm

- The fitness function could be improved by being better at detecting hovering, no matter the height. Many individuals have had almost perfect hovering, but not at the desired height, thus not getting a high fitness value. As long as it is above the starting point, it is acceptable. This could be implemented by taking into account the derivative of the height.
- The fitness evaluation of an individual is highly unstable. Many evaluations should be applied for each individual and averaged before it is possible to say something certain about its fitness.

- The GA parameters should be optmized, but it would need several evolution runs to figure out optimal parameters. For instance, the population size has now been chosen from the so-called Greffenstette settings, but maybe a smaller population would give quicker evolution for our slowly-evaluating system. Maybe crossover has a too high probability now, and this might cause disruptions in the genomes. Crossover points should be investigated more thoroughly.

8.4 Future expansions

There are a couple of very tempting expansions:

- Increase number of sensors. The ATmega128 microcontroller provides the possibility to connect 8 sensors to ADC.
- Increase the number of fitness parameters. The added sensors could be used as additional parameters.
- Increase the degrees of freedom and adding additional propellers to control the movement. This could be developed in a stepwise way, incrementally increasing the moveability to reach the ultimate goal. The ultimate goal is obviously free-flying mode.
- Building a suitable power supply. This can easily be achieved with a transformer, a rectifier and a variAC [1].

8.5 Final Conclusion

As shown in the checklist, most of the milestones have been reached, which is beyond our expectations. The hovering goal has been almost fulfilled, although some improvements will have to be made to have perfect hovering. Real world fitness evaluation has been employed through our mechanical design. We have found a solution for doing evolution in hardware, by implementing a spiking neural network on an FPGA. We have learned much about VHDL coding, digital design and evolutionary algorithms. It has been very inspiring to work on a concrete hardware project, even though it has been time demanding. We have built a good basis platform suitable for further expansions and experiments.

[1] a device to adjust the alternating current (AC)

Chapter 9

Acknowledgements

In this chapter credit is given to the people who have been extra helpful.

- The mechanical workshop crew at Perleporten[1] for their assistance and providing us all the necessary parts.
- Elkraft for providing a power supply for the setup.
- Staff and students on 3rd floor in it-bygget for eagerly giving feedback, advice and other useful inputs.
- Morten Loftet and friends at Hobbyloftet (located in Stjørdalen) for providing their expertise in choosing the right rc equipment.
- Martin and Marius for lending us their batteries.

Thanks!

[1] Institutt for produktutvikling og materialer

Chapter 10

Glossary

Atmel An innovative high-tech company. Its Trondheim based division makes the world's most prominent microcontroller.

AVR A microcontroller produced by Atmel.

DC-engine A small electromotor.

ESC Electronic Speed Controller.

FPGA A Field Programmable Gate Array where the logic network can be programmed into the device after its manufacture. An FPGA consists of an array of logic elements, either gates or lookup table RAMs, flip-flops and programmable interconnect wiring.

ISP In System Programming allows you to program a chip on the PCB. Before the components had JTAG support, this was the only way to load programs.

JTAG Joint Test Action Group (IEEE 1149.1) is a standard specifying how to control and monitor the pins of compliant devices on a printed circuit board.

LED Light Emitting Diode is a type of diode that emits light when current passes through it. Depending on the material used the colour can be visible or infrared. LEDs have many uses, visible LEDs are used as indicator lights on all sorts of electronic devices and in moving-message panels, while infrared LEDs are the heart of remote control devices.

LSB Least Significant Bit.

Microcontroller (MCU) A device with the capability to execute programs.

MSB Most Significant Bit.

RC Remote Controlled

SPROM Serial Programmable Read Only Memory is a kind of ROM that can be written serially using a PROM programmer.

UART Universal Asynchronous Receiver/Transmitter is an integrated circuit used for serial communications, containing a transmitter (parallel-to-serial converter) and a receiver (serial-to-parallel converter), each clocked separately. The parallel side of a UART is usually connected to the bus of a computer. When the computer writes a byte to the UART's transmit data register (TDR), the UART will start to transmit it on the serial line. The UART's status register contains a flag bit which the computer can read to see if the UART is ready to transmit another byte. Another status register bit says whether the UART has received a byte from the serial line, in which case the computer should read it from the receive data register (RDR). If another byte is received before the previous one is read, the UART will signal an "overrun" error via another status bit.

Bibliography

[1] Roggen, D., Hofmann, S., Thoma, Y. and Floreano, D. (2003) Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot. Lohn, J. and Zebulum, R. and Steincamp, J. and Keymeulen, D. and Stoica, A. and Ferguson, M. I. (eds.) , 2003 NASA/DoD Conference on Evolvable Hardware, pages 189-198.

[2] Andrés Upegui, Carlos Andrés PEÑA-REYES, and Eduardo Sanchez. "A functional spiking neuron hardware oriented model". In Proceedings of the International Work-conference on Artificial and Natural Neural Networks IWANN2003, Lecture Notes in Computer Science 2686, pages 136-143, Springer, 2003.

[3] Floreano, D. and Mattiussi, C. (2001) Evolution of Spiking Neural Controllers for Autonomous Vision-based Robots. In T. Gomi (ed.), Evolutionary Robotics IV, Berlin: Springer-Verlag.

[4] Selene Maya, Rocio Reynoso, Cesar Torres, Miguel Arias-Estrada, "Compact Spiking Neural Network Implementation in FPGA". Field Programmable Logic Conference (FPL'2000). Villach, Austria. Aug. 28-30, 2000, pp. 270-276

[5] Floreano, D., Schoeni, N., Caprari, G. and Blynel, J. (2002) Evolutionary Bits'n'Spikes. In R. K. Standish, M. A. Beadau and H. A. Abbass, editors. Artificial Life VIII: Proceedings of the Eight International Conference on Artificial Life, MIT Press.

[6] Augustsson, Wolff, Nordin (2002) Creation of a Learning, Flying Robot by Means of Evolution. GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference.

[7] Grefenstette, J.J. "Optimization of Control Parameters for Genetic Algorithms," IEEE Trans. Systems, Man, and Cybernetics, Vol. SMC-16, No. 1, Jan./Feb. 1986, pp. 122-128.

[8] Djupdal, Asbjørn. "Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter," Master thesis, NTNU, June 2003.

[9] Ellingsen, Erling A., Glette, K. "MNFIT-378 Prosjekt 1: Genetisk Programmering." Project report, NTNU, March 2003.

[10] Thomas Philip Runarsson: "Lecture notes on Neural Networks", Applied Mathematics and Computer Science, Science Institute, University of Iceland. cerium.raunvis.hi.is/~tpr/courseware/rl/slides/annintro.pdf Accessed 28.11.2003.

[11] OpenCores project: Serial UART, http://www.opencores.org/projects/miniuart2/, Accessed 21.11.2003.

[12] Model Motors electronic speed controller data sheet. http://www.modelmotors.cz/index.php?id=en&nc=produkty_vypis&u Accessed 17.11.2003.

[13] Propeller tips from Dubai RC Hobbies. http://www.dubairchobbies.org/propeller_tips.htm Accessed 24.11.2003