WORDT
NIET UITGELEEND

Nets In Space

Spatial Design of a Modular Neural Network by Reconfiguration of an FPGA

written by

Rudi Alberts

Rijksuniversiteit Groningen
Bibliotheek Wiskunde & Informatica
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

Groningen, 25 September 2002

WORDT
NIET UITGELEEND

in partial fulfilment of the requirements for the M.Sc. degree in Computing Science at the Rijksuniversiteit Groningen in Groningen (The Netherlands).

under supervision of

Prof.dr.ir. L. Spaanenburg

Rijksuniversiteit Groninge
Bibliotheek Wiskunde & Informatic
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

Abstract

The digital implementation of neural networks has never become really popular. The synapses seem too numerous to be physically shaped and therefore more easily handled in software. Further, their operation requires a multiplication, which is electrically easy but logically cumbersome. The idea of many simple nodes, that in combination produce a complicated function, seemed like a fairy tale.

But micro-electronic technology has changed this picture drastically. At the start of the silicon era, the lack of integration drove towards a temporal computing style, whereby many tasks were scheduled for the optimal use of just a few resources. But the level of integration rose faster than the design efficiency. This creates a „Productivity Gap": in current technology we have more resources available than we can optimally use.

It is suggested that in contrast to the past we can now utilize a spatial computing style, whereby few tasks are roaming over many resources. In a typical spatial device like a Field-Programmable Gate-Array (FPGA) we find configurable interconnect & logic, mixed with memory and arithmetic macros. Configuration blocks take the role of program segments and re-configuration schemes replace temporal scheduling. While adequate CAD tools are still lacking, the first challenge is to envision what this new computing style has to offer. This is best learned by experimentation. Hence, this thesis looks into the potential of modern FPGA devices to implement neural networks.

A neural network can be constructed from SRAM and multiplier macros, glued together by the Configurable Logic & Interconnect Blocks. As the implementation of a complete network, this has too much similarity with temporal computing; as the implementation of a single neuron we still have the classical size problems. Here, we investigate the modular neural network: many small networks that are dynamically configured into a virtual large one. We show that this concept is scalable, utilizes the resources efficiently and allows for a high-level behavioral abstraction during design. Hereby it illustrates a number of potential advantages of the spatial computing style.

Rijksuniversiteit Groninger
Bibliotheek Wiskunde & Informatic
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

Samenvatting

Digital neurale netwerken hebben zich nooit in een grote populariteit mogen verheugen. De vele synapsen impliceren een hoeveelheid verbindingen, waarvoor geen eenvoudige physische realisatie te denken is; software heeft daar minder moeite mee. Verder ligt aan de operatie van de enkele synapse een vermenigvuldiging ten grondslag, die analoog beter en efficiënter te realiseren is dan digitaal. Het oorspronkelijke idee achter het neurale netwerk van een grote hoeveelheid eenvoudige samenwerkende processoren lijkt daarmee gedoemd tot een sprookje.

Maar de micro-elektronica heeft inmiddels grote stappen vooruit gemaakt. De geringe integratie dichtheid leidde in den beginne bijna vanzelfsprekend tot een temporele ontwerpstijl, waarbij een grote hoeveelheid taken geordend werd voor een optimaal gebruik van de weinige rekendelen. Maar de integratie dichtheid groeide sterker dan het ontwerp vermogen. Dit gaf aanleiding tot een „Productiviteitsgat“: hedentendage zijn er meer rekendelen beschikbaar dan we optimaal kunnen gebruiken.

In de afgelopen jaren is het concept van een ruimtelijke ontwerpstijl gegroeid, waarbij een gering aantal taken zich verspreidt over een groot aantal rekendelen. Een typische bouwsteen zoals de Field-Programmable Gate-Array (FPGA) kent vrij configureerbare verbindingen en digitale bouwblokken, gemengd met geheugen en rekenmacro's. Configuratie blokken nemen daarbij de rol over van software programma's en configuratie schema's verzorgen de ordening van de taken. Een adequate CAD ondersteuning ontbreekt vooralsnog en het is nog een uitdaging om te concretiseren wat de nieuwe ontwerpstijl te bieden heeft. Kennelijk is er nog een experimenteerfase noodzakelijk. Vanuit deze optiek richt zich de scriptie op de mogelijke toepassing van FPGA elementen voor de constructie van neurale netwerken.

Een neuraal netwerk kan gebouwd worden met SRAM en vermenigvuldig macro's, samengesteld door configureerbare verbindingen en digitale bouwstenen. Voor de implementatie van een neuraal netwerk uit één stuk heeft dit te veel gemeen met de temporele methodiek; voor de implementatie van een enkele neuron houden we de aloude ruimtelijke problemen. Daarom bestuderen we hier modulaire neurale netwerken: een grote hoeveelheid kleine netwerken die dynamisch geconfigureerd worden tot een virtueel groot netwerk. We tonen dat dit concept schaalbaar is, zijn rekendelen optimaal gebruikt en openingen biedt voor een doorgroei naar een hoger abstractie niveau voor het ontwerpen. Daarmee illustreert het een aantal mogelijke voordelen van de ruimtelijke ontwerpstijl.

Contents

Chapter 1 The World of Neural Networks

Neural networks have attracted the interest of many people because of their potential relation with Nature. Where Nature seems capable to learn from scratch, an Artificial Neural Network (ANN) was supposed to be equally gifted. Unfortunately an ANN proves to be just as hard to nurture as any other man-made device. In this chapter we will illustrate what makes the ANN so difficult and why advances in modern micro-electronics may change this.

1.1 Basics of Operation

The type of neural network we are working with is the widely used multi-layer feed-forward network [1]. Typically, this network consists of a set of sensory units that constitute the *input layer*, one or more *hidden layers* of computation nodes, and an *output layer* of computation nodes. The input signal propagates through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as *multi-layer Perceptrons* (MLPs). An example of a feed forward network with one hidden layer is given in Figure 1. More hidden layers are possible.

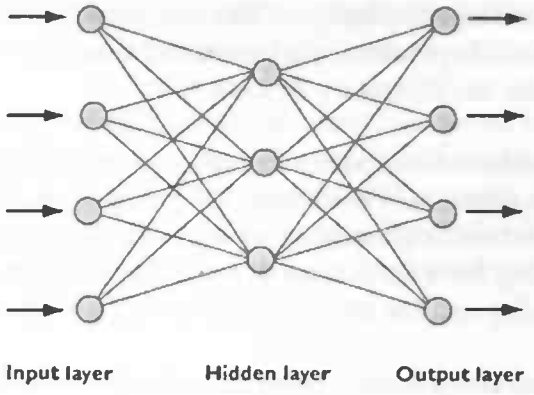

Figure 1 Feed-forward neural network

The input neurons of an MLP are just used for fan-out of the incoming signals, that is, they connect all incoming signals to all neurons in the first hidden layer. Computations take place in the hidden neurons and in the output neurons. The computation that takes place in these neurons consists of two steps. In the first step the value of the weights of all incoming synapses to that neuron will be multiplied by the value of the corresponding neurons (the neurons that feed these synapses). These values are summed up, including the bias value of the neuron. This results in the net internal activity level $v(n)$ of the neuron, where n is the iteration step. The value $v_j(n)$ for a specific neuron j is defined by:

$$v_j(n) = \sum^{p}_{i=0} w_{ji}(n)\, y_i(n)$$

where p is the total number of inputs (excluding bias) applied to neuron j, and $w_{ji}(n)$ is the synaptic weight connecting neuron i to neuron j, and $y_i(n)$ is the input signal of neuron j or, equivalently, the function signal appearing at the output of neuron i. The output value $y_j(n)$ of neuron j is computed by applying an activation function φ on the internal activity level of j. Thus,

$$y_j(n) = \varphi(v_j(n))$$

This activation or transfer function is often implemented by a sigmoid function. This is because it is a continuously differentiable nonlinear function. These properties are used during the learning process of the neural network (using back-propagation).

1.2 Temporal versus Spatial Computing

Computing technology can be superficially divided into software and hardware. From the early days of hardware, software has evolved as a means to personalize a platform after manufacture. A computer was a general-purpose device with an Instruction-Set Architecture (ISA), such that a software program expressed in such instructions can manipulate the hardware platform to operate as desired. The platform could be mass-fabricated, leading to a significant drop in cost when compared to special-purpose computers.

From the onset the computer was a single resource machine. It could handle one process at a time, though this process could be temporarily set aside (foreground / background). As the software complexity grew, more processes came into existence and it became necessary to put such processes in an efficient order. The complexity of this task grew and the system manager needed support to handle the dynamics of operation. This created the need for middleware that could alleviate the scheduling burden: the Operating System (OS).

Process scheduling on a limited set of resources stresses temporal aspects of computing. Given the set of hardware resources, the question is to allocate the processes such that execution time is minimized. Most of the task scheduling literature assumes the single resource, or at least that a myriad of processes are fighting for a limited set of resources. Such an assumption is true where the resources are complex and bulky and the processes are comparatively simple and small.

When von Neumann discussed computational processes, he was rather referring to the biological inspiration where many resources are willing to co-operate. From such a bio-inspiration he conferred that mostly communication will be a bottleneck. And he proved to be right, as the advances in micro-electronics pushed the computer industry towards faster and faster components, while the packaging technology remained almost unaltered. Already in the early Cray computers, most of the execution speed was spent on the transfer of information between components.

In the early eighties, this communication bottleneck resulted in the discussion on architectures that aim to keep the computation within the processor. A successful direction was to increase the amount of on-chip storage so that the demand on data from the external memory was limited. The room for the additional registers was created by simplifying the instruction set; hence the name "Reduced Instruction Set Computer". But in time, technology increased available chip size and the instructions became more complex again.

Table 1: The steps to reconfiguration.

	Computation	Communication
Processor (uP)	Hard	Hard
System on Silicon (SoS)	Soft	Hard
Network on Chip (NoC)	Soft	Soft

Further advances in micro-electronics did not only push the speed limits but also the complexity of the components. And suddenly one finds that the micro-electronic chip can house an entire complex system or a network of simpler ones. This renews the discussion on computing architectures. Where more resources are available on chip, communication can be kept on chip and a next increase in performance can be expected. This move towards "Networks on the Chip" clearly opens a number of new directions.

One way to look at this development path is shown in [2]. The conventional processor may have been programmable, but such only builds a selection of choices given by the architecture. It was only with the coming of the SoS technology that the hardware variety allowed for re-configuration, either in a hardware / software trade-off or by factual building a changed architecture. But the communication between the nodes will only become adaptive where entire networks are facilitated on the same silicon carrier.
This trend shows how we are coming from a world of restricted resources to a world of ample resources, if not even an overmass. This swap can also be the basis for spatial computing. In stead of sequentially developing the desired function over the available resource, it is envisaged that many functions can execute on the spatial variety of resources. Hence the functional goals can be stretched from extremely small to extremely fast in the same fabrication technology.

1.3 Experimental Scope

The question remains on what pillars spatial computing can be built. For a long time, n-dimensional transistor arrays were used to accommodate logic structures of standard cells. This links the electronical to the logical level of abstraction in a one-time personalization.

But this is insufficient for larger designs. When a large design is iteratively composed from small basic cells, small inefficiencies on the basis of the compositional hierarchy can easily grow into major overall losses. On the other hand, simply replicating a large optimized macro falls short every time when the macro is either too specialized or too general. In other words, Heisenberg seems to rule: either the design is optimal or it is efficient, but never both.

A lesson from the past is to embed optimized cores in a personalized area. This was primarily meant for last minute repairs and to glue the part into its environment. But software is required to program large parts as otherwise large parts are not required too often. But again the facilities for programming introduces inefficiencies.

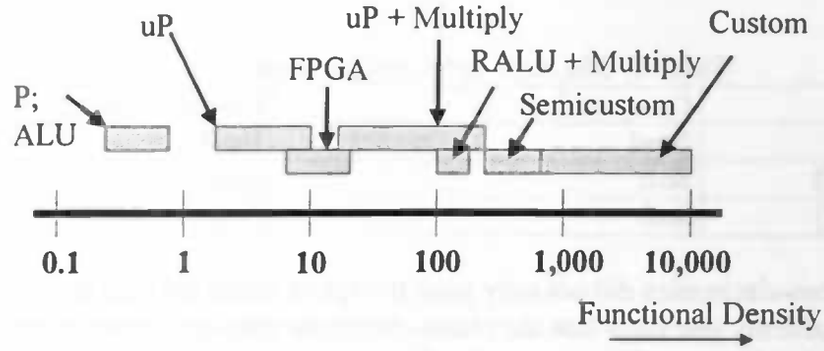

Figure 2 Multiplication Domain Comparison

The largest macro that can often be used is probably the multiplier. It has been subject to study for some decennia and a range of implementations is known. In [2] a number of implementation domains are compared in terms of functional density (multiply bit operations per unit of area). He concludes that this device covers a wide range of values and that re-configurable devices (RALU) with hardwired multipliers are close in density to dedicated parts.

But deHon's metric is not fully realistic. It rather stresses the presence of multipliers than giving an objective figure of merit. In order to get a better feeling for the issues involved, we will develop and compare different implementations of a multiplication-rich system. We like that system to be built from components optimized at widely differing abstraction levels, as designing with optimized multipliers might give a different result from optimizing the overall design.

1.4 Exploration Areas

In the course of the research on which this M. Sc. Thesis is based a number of different systems have been proposed to use as architectural glue. When looking at systems with multipliers, a variety to choose from exists because almost any rise in complexity will introduce multiplication or even higher level mathematical abstractions.

A first question is on the relevance of multiplier architectures. Given a realization technology, not all architectures are equally beneficial. Hence, where in theory some are faster than others, reality may paint a different picture. The ancient example of this effect is the speed-up of carry propagation by means of carry look-ahead logic. Though it is supposed to accelerate, the implied overhead will not always pay off directly and there is usually a minimal bitlength to be exceeded.

As we take field-programming as the product development style of the future, FPGA is the target technology. This design methodology based on hard-programmable parts has come a long way and will probably still have a long way to go. Currently the platform architecture is SRAM dominated, but it is to be expected that DRAM will be needed in the future to make further advances. Obviously any analysis will have to target on designs that can be realized now and will only become better in the future: timing locality is such a future-driven concern.

Current FPGA parts do not enforce local synchronous timing. Hence design portability is not ensured over future lithographical detail. Pipelining does not only aim to speed up a system by the

introduction of spatial and temporal parallelism, it is also a design property that supports local synchronicity and can be easily enforced. But there is no such a thing as a free lunch, and the question remains: what do we sacrifice by pipelining on an FPGA?

As the proof of the pudding is in the eating, the aim of our research is still: how do temporal and spatial decisions work out in the design of a larger system? This may be totally dependent on the nature of the system. A famous benchmark test is the verification of keys for encrypted messages. According to [2], the spatial implementation is a factor 20 faster than temporal software on a Pentium-III. Such a data cruncher is a clear source of inspiration.

In our case, we look at a modular neural network. Digital implementations have suffered from the sheer size of the multiplying adder. This caused the tendency for temporal solutions, that for this same reason had hardly any benefits over software implementations. Our main research question is whether better implementations are possible by exploiting the inherent parallelism of ANNs by the two outstanding characteristics of FPGAs. In one sentence:

> What is the most efficient ANN implementation on an FPGA using re-configuration and the many multiplier / SRAM macros?

Chapter 2 Tooling

In this chapter we give an overview of the tooling that has been used to perform the experiments that follow. VHDL is used as specification language, while ModelSim is applied for simulation. The open software market is searched for logic synthesis tools and the XILINX WebPack ends the suite.

2.1 Introduction to VHDL

VHDL is a language for describing digital electronic systems. It arose out of the United States government's Very High Speed Integrated Circuits (VHSIC) program. During this program it became clear that there was a need of a standard language for describing the structure and function of integrated circuits (IC's). The program was run over many years and therefore split into parts with individual milestones. As new contracts were given out for each subsequent program part, other people usually continued them. This gave new importance to design documentation, making it mandatory that it was written in a universally accepted, executable language. Hence the VHSIC Hardware Description Language (VHDL) was developed. Under the auspices of the Institute of Electrical and Electronic Engineers (IEEE) it was subsequently matured and in 1987 it adopted in the form of the IEEE Standard 1076, *Standard VHDL Language Reference Manual*. Like all IEEE standards, the VHDL standard is subject to reviews, at least every five years. These reviews have led the way to the revisions VHDL-93 and VHDL-2001 (the current version).

VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a system, how it is decomposed into subsystems and how those subsystems are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

As an example we take a 4 bits adder. The adder uses four full-adder components as given in Figure 3. This component has a carry in (Cin) and two bits x and y as inputs and a carry out (Cout) and s as outputs.

```
library ieee;
use ieee.std_logic_1164.all;

entity fulladd is
  port (    Cin, x, y : in  bit;
            s, Cout   : out bit );
end fulladd;

architecture fulladd_arch of fulladd is
begin
  s    <= x xor y xor Cin;
  Cout <= (x and y) or (Cin and x) or (Cin and y);
end fulladd_arch;
```

Figure 3 The full adder

The library std_logic_1164 is included so that standard components (for example logic gates) can be used. Then the external interface is given by a description of the ports. The implementation of

the entity is described in the architecture body. The input bits will be added and result in a sum (s) and carry out.

Now we have the full-adder we can take four of these components and put them together to compose a 4 bits adder. We make a new entity adder4 as given in Figure 4. The entity has a carry and two bit-vectors of length 4 as inputs, one bit-vector of length 4 and a carry out bit as outputs. The entity uses three internal signals called c1, c2 and c3. These are used to connect the four full-adding components. In the declaration part of the architecture body the component is declared. Thereafter is it instantiated four times. Using *port map* statements, the ports of the instances are mapped onto signals and ports of the entity.

```
library ieee;
use ieee.std_logic_1164.all ;

entity adder4 is
      port (      ci    : in  bit;
                  a     : in  bit_vector(3 downto 0);
                  b     : in  bit_vector(3 downto 0);
                  sum   : out bit_vector(3 downto 0);
                  co    : out bit );
end adder4;

architecture structure of adder4 is
      signal c1, c2, c3 : bit;
      component fulladd
            port (      Cin, x, y   : in  bit;
                        s, Cout     : out bit );
      end component;
begin
      stage0: fulladd port map ( ci, a(0), b(0), sum(0), c1 );
      stage1: fulladd port map ( c1, a(1), b(1), sum(1), c2 );
      stage2: fulladd port map ( c2, a(2), b(2), sum(2), c3 );
      stage3: fulladd port map ( c3, a(3), b(3), sum(3), co );
end structure;
```

Figure 4 4 bits adder

At this moment the design is ready for simulation. We start our simulation program, add the two VHDL files and set the signals a, b and ci to some example values. After running the simulation we see that the sum and carry out signals got the right values (Figure 5).

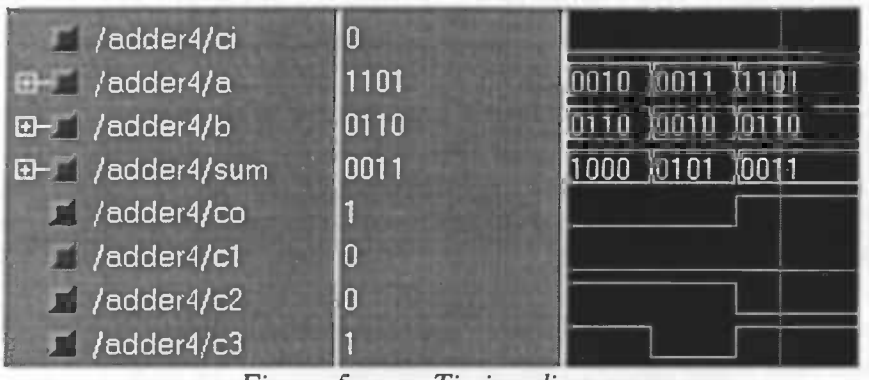

Figure 5 Timing diagram

2.2 What are CPLDs and FPGAs ?

The hardware on which we build is a Xilinx FPGA [3] [4]. This section explains what CPLDs and FPGAs are.

During the sixties there was discrete logic. Systems were built from lots of individual chips with a spaghetti-like maze of wiring between them. It was difficult to modify such a system after you built it. After a week or two it was difficult to remember what each of the chips was for.

Manufacturing such systems took a lot of time because each design change required that the wiring be redone which usually meant building a new printed circuit board. The chip makers solved this problem by placing an unconnected array of AND-OR gates in a single chip called a *programmable logic device* (PLD, Figure 6). The PLD contained an array of fuses that could be blown open or left closed to connect various inputs to each AND gate. You could program a PLD with a set of Boolean sum-of-product equations so it would perform the logic functions you needed in your system. Since the PLDs could be rewired internally, there was less of a need to change the printed circuit boards which held them.

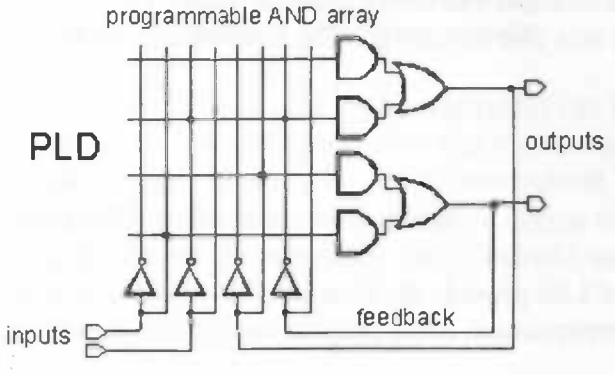

Figure 6 PLD Architecture

Simple PLDs could only handle up to 10-20 logic equations. If you had a large design, you had to break the design into parts and divide them over a set of PLDs. This was time consuming and you also had to interconnect the PLDs with wires. Problems then arose when you wanted to make changes to your design. The chip makers solved this problem by building much larger

programmable chips called *complex programmable logic devices* (CPLDs) and *field-programmable gate arrays* (FPGAs). With these, you could get a complete system onto a single chip.

A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.

An FPGA takes a different approach. It has a bunch of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented.

CPLD and FPGA manufacturers use a variety of methods to program the chips. The first method uses fuses or anti-fuses that are programmed by passing a large current through them. These chips are called *one-time programmable* (OTP) because you can't rewire them internally once the fuses are blown.

A second type of chip uses an EPROM or EEPROM that is underlying the programmable logic. Each bit in the memory determines whether the switch above it will be closed or opened, therefore, whether two logic elements will be connected or not. Older chips using EPROM can only be resetted with ultraviolet light. EEPROMs can be electrically reprogrammed in a few nanoseconds. Our chip uses this technique. A disadvantage is that the contents of the memory are lost when you switch off the device.

Finally, some manufacturers use static RAM or Flash bits to program the chips. CPLDs and FPGAs built using RAM/Flash switches can be reprogrammed without removing them from the circuit board. They are often said to be *in-circuit reconfigurable* or *in-circuit programmable*.

As you can see, figuring out which switches to open and close in order to create a logic circuit is quite difficult. That's why the chip manufacturers provide development software that takes a description of the logic design as input and then outputs a binary file which configures the switches in a CPLD or FPGA so that it acts like the design. The development software will be discussed in the next sections.

2.2.1 Basic building blocks

Xilinx user-programmable gate arrays include two major configurable elements: *configurable logic blocks* (CLBs) and *input/output blocks* (IOBs). IOB's provide the interface between the package pins and internal signal lines. CLBs provide the functional elements for constructing the user's logic. IOBs and CLBs are interconnected using programmable switch matrices (PSMs). This is shown in Figure 7.

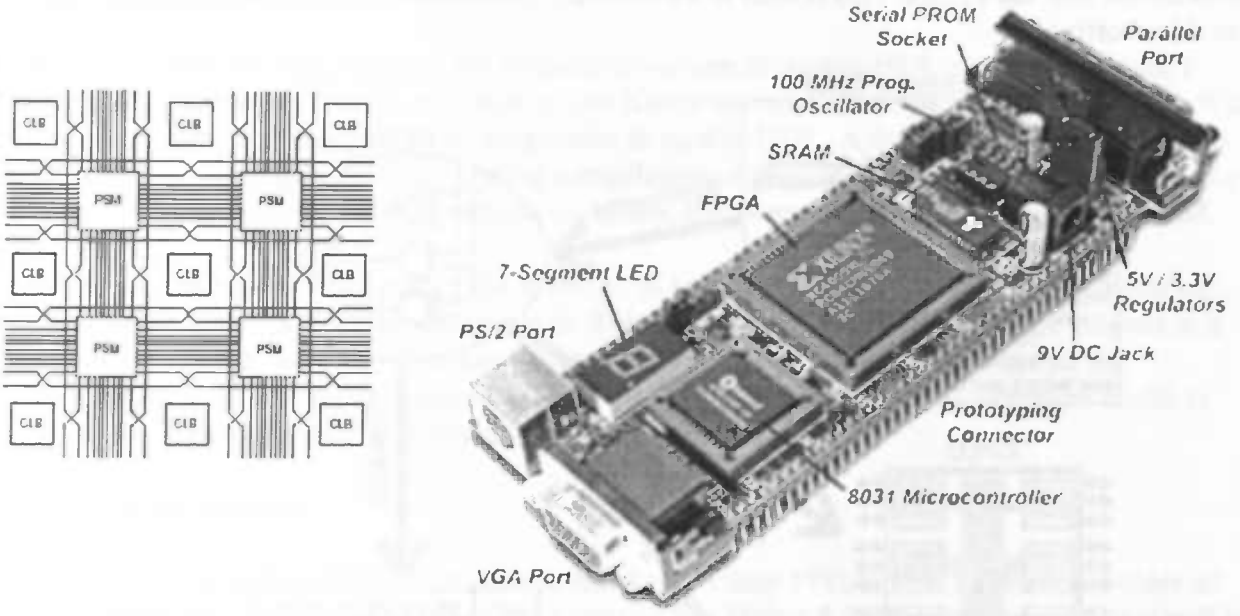

Figure 7 The Xilinx FPGA

Let's take a look inside a CLB. CLBs implement most of the logic in an FPGA. This logic is stored in three function generators. Each CLB contains two 4-input function generators and one 3-input function generator. Combinatorial logic functions can be stored in these function generators in the form of *look-up tables* (LUTs). A simple example of a 2-input look-up table for the XOR function:

```
i1 i2 out
 0  0  0
 0  1  1
 1  0  1
 1  1  0
```

The inputs of the 4-input function generators come from outside the CLB. All function generators have 1 output. The outputs of the two 4-input function generators can be fed to the 3-input function generator. Other inputs for this function generator come from outside.

Each CLB contains two storage elements (often used as *flip flops*) that can be used to hold the function generator outputs. However, the function generators and the storage elements can also be used independently. Inputs for the storage elements can come from outside directly and outputs of the function generators can directly drive an output of the CLB.

2.3 Development tools

During this project I have used several development tools. The used tools can be roughly divided into two groups. The first group consists of tools that can be used to develop your VHDL code. When the code is ready to test you start the logic synthesizer that transforms the VHDL into a net-list. A net-list is a description of the various logic gates in your design and how they are interconnected (see Figure 8). These net-lists will be simulated. The functionality can be checked in the timing diagrams of which an example is given in Figure 5. The tools in the other group can do the same thing. Additionally they provide an implementation program to map the logic gates and

interconnections into the FPGA. This results in a bit-stream that can be uploaded to the FPGA. We come to this shortly.

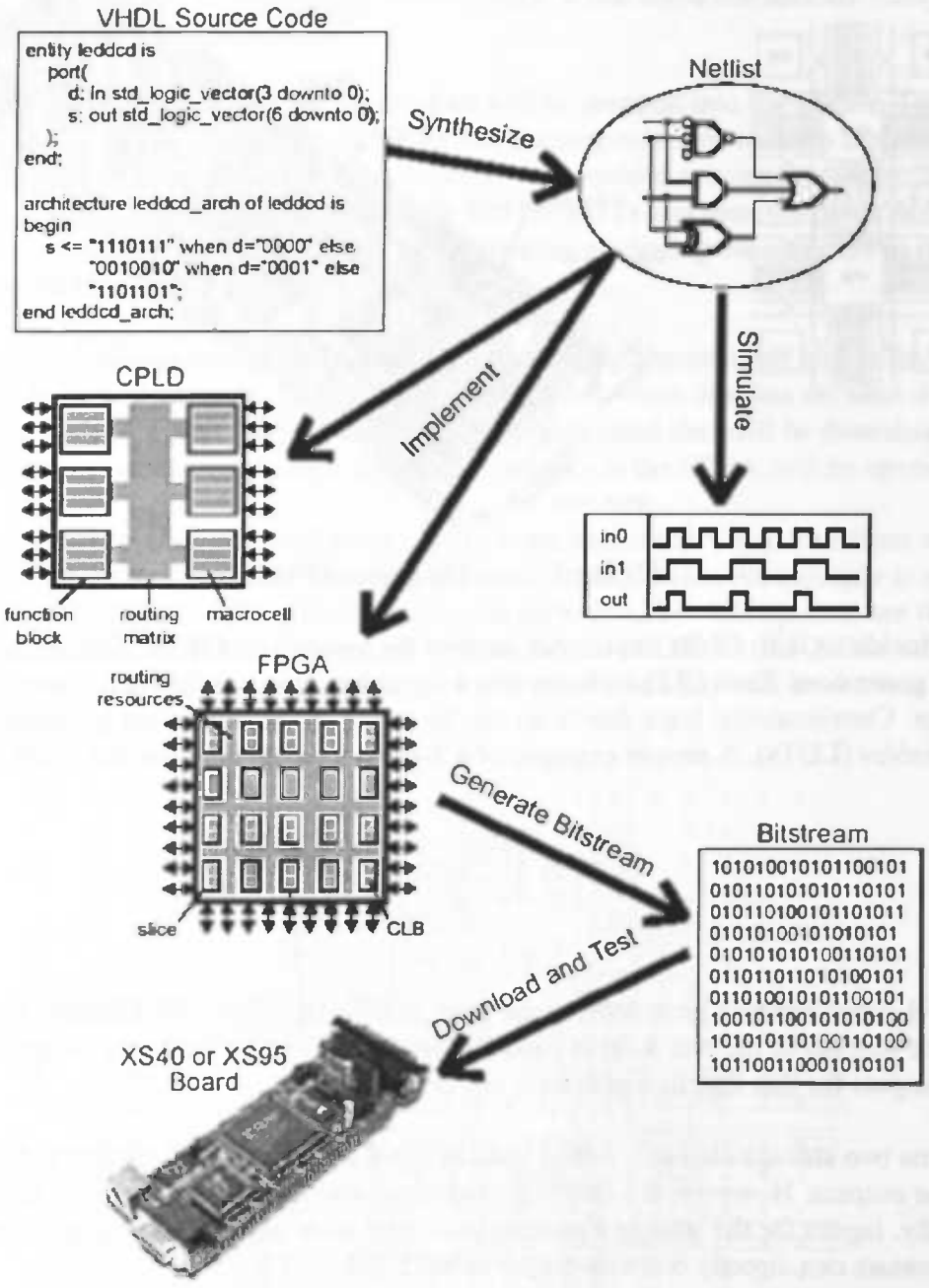

Figure 8 Steps in creating and testing an CPLD or FPGA-based design

2.3.1 Aldec Active-HDL

I have used an evaluation version of Aldec's Active-HDL 5.1 [5]. This software offers a good environment in which you can write your VHDL code. The simulation capabilities are good. You can assign stimulators to the signals easily and check the results in different formats. Two drawbacks are the limitations of the evaluation version, a time limit of 20 days and a maximum file size of 5 KB (for the VHDL files).

2.3.2 *Xilinx WebPACK and ModelSim*

After running into the limitations of the evaluation version of Active-HDL a couple of times, I fortunately found the WebPACK software on the Xilinx website, accompanied by ModelSim. With respect to functionality WebPACK is comparable to Active-HDL. A difference is that WebPACK targets the design on a specific chip. During compilation of the code the logic is optimized and an overview is given about the number of look-up tables, flip flops, input/output pads etc. required.

ModelSim can be started from within the WebPACK software. This is a widely used simulation tool. The usage of the tool is very comfortable. This is because all the actions can be entered in a command line. This saves a lot of mouse clicks, thus time. A further advantage is that the stimulation of the signals can be entered in a text file. So you don't need a lot of mouse clicks to assign values to signals every time you run a simulation.

2.3.3 *Xilinx Foundation*

Xilinx Foundation is the software that accompanied our Xilinx FPGA. This software provides all the tools necessary to design and implement a circuit. See Figure 8 for the various steps that will be taken during the design process.

Figure 9 shows the Foundation Project Manager. On the left you see the VHDL files that are in the current project. On the right the buttons for the various steps.

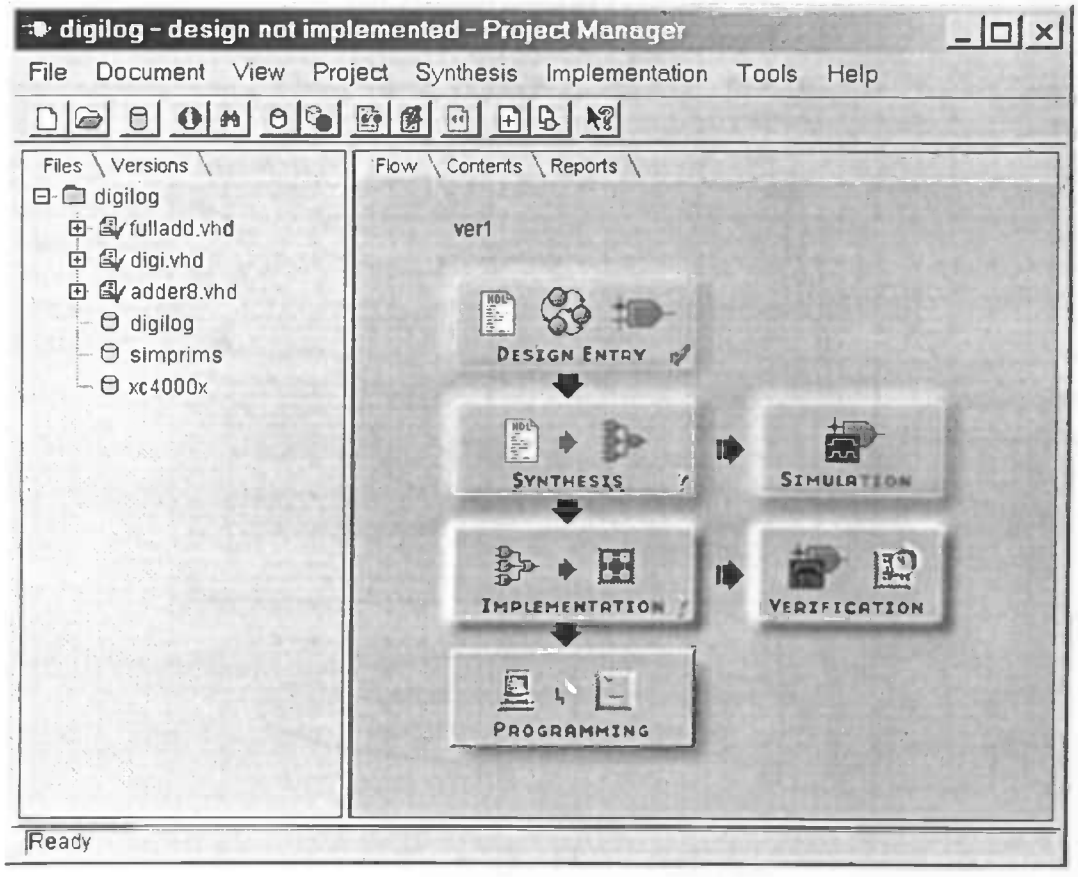

Figure 9 The Foundation Project Manager

Firstly the VHDL code will be entered into the VHDL editor. The created VHDL files can be added to a project and all files will be analysed. When all code is OK the design can be *synthesized.* During this process the code is transformed into net-lists. When these net-lists are ready they can be simulated to check the functionality.

By pressing the *Implementation* button the net-lists will be mapped into the FPGA. The configurable logic blocks in the FPGA will be further decomposed into look-up tables that perform logic operations. The CLBs and LUTs are interwoven with various routing resources. The mapping tool collects your net-list gates into groups that fit into the LUTs and then the place & route tool assigns the gate collections to specific CLBs while opening or closing the switches in the routing matrices to connect the gates together.

Once the implementation phase has been completed, the resulted system can be verified by pressing the *Verification* button. Usually the functionality will be good when this was the case during the simulation of the net-list. In the final part a bit-stream is generated that will be downloaded to the FPGA. This bit-stream determines which electronic switches in the FPGA will be opened or closed.

When the design has been implemented correctly, you can start the FPGA editor from within Foundation. This program shows how the logic is placed on the FPGA. An example is shown in Figure 10. The small squares in the grid, that consume up to 20x20 = 400 squares, are the CLBs. The IOBs are on the border. You can see wires running from the IOBs to the CLBs in the middle. When you double-click a CLB in the program the contents of that CLB will be shown.

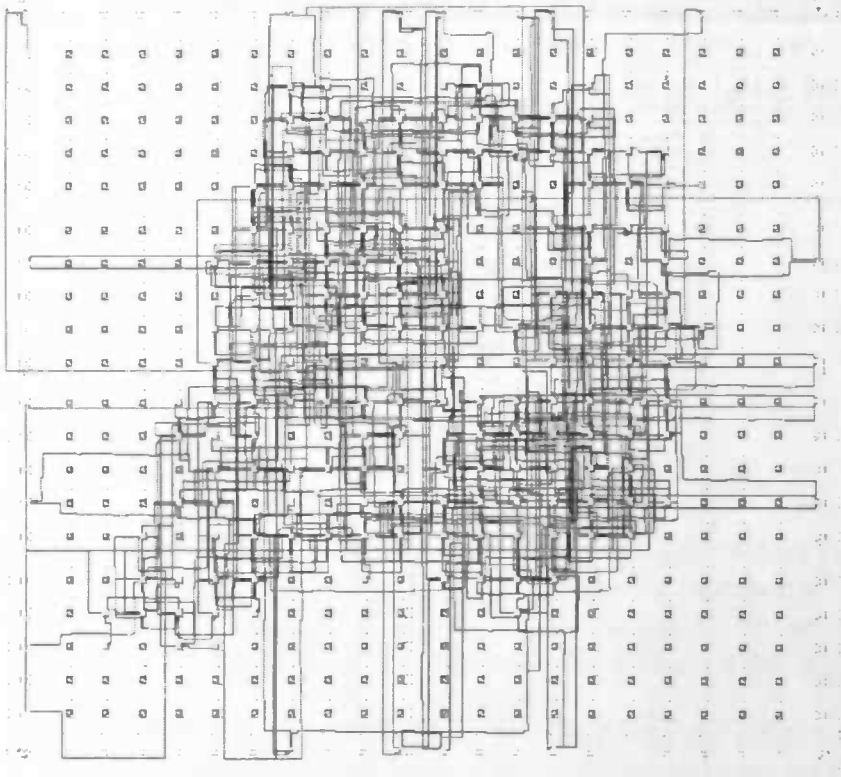

Figure 10 Example FPGA Placement

2.4 *Logical and Physical Synthesis*

The objective of this project is to put as many neural calculations into a given chip area as possible. To help us minimizing the chip area needed to perform the required functions, it is important to examine the capabilities of third-party tools for minimizing the logic and place and route the design into the FPGA.

A typical CAD flow that is used for these purposes is given in Figure 11. First, the SIS synthesis package is used to perform technology-independent logic optimization of each circuit. That is, SIS attempts to simplify the logic and remove redundant circuitry. Next, each circuit is technology-mapped into 4-LUTs and flip flops by FlowMap. FlowMap takes a description of a circuit in terms of basic gates and implements it using only 4-LUTs and flip flops. Then, T-VPack is used to pack LUTs and flip flops together into larger logic blocks, and finally VPR is used to place and route the circuit.

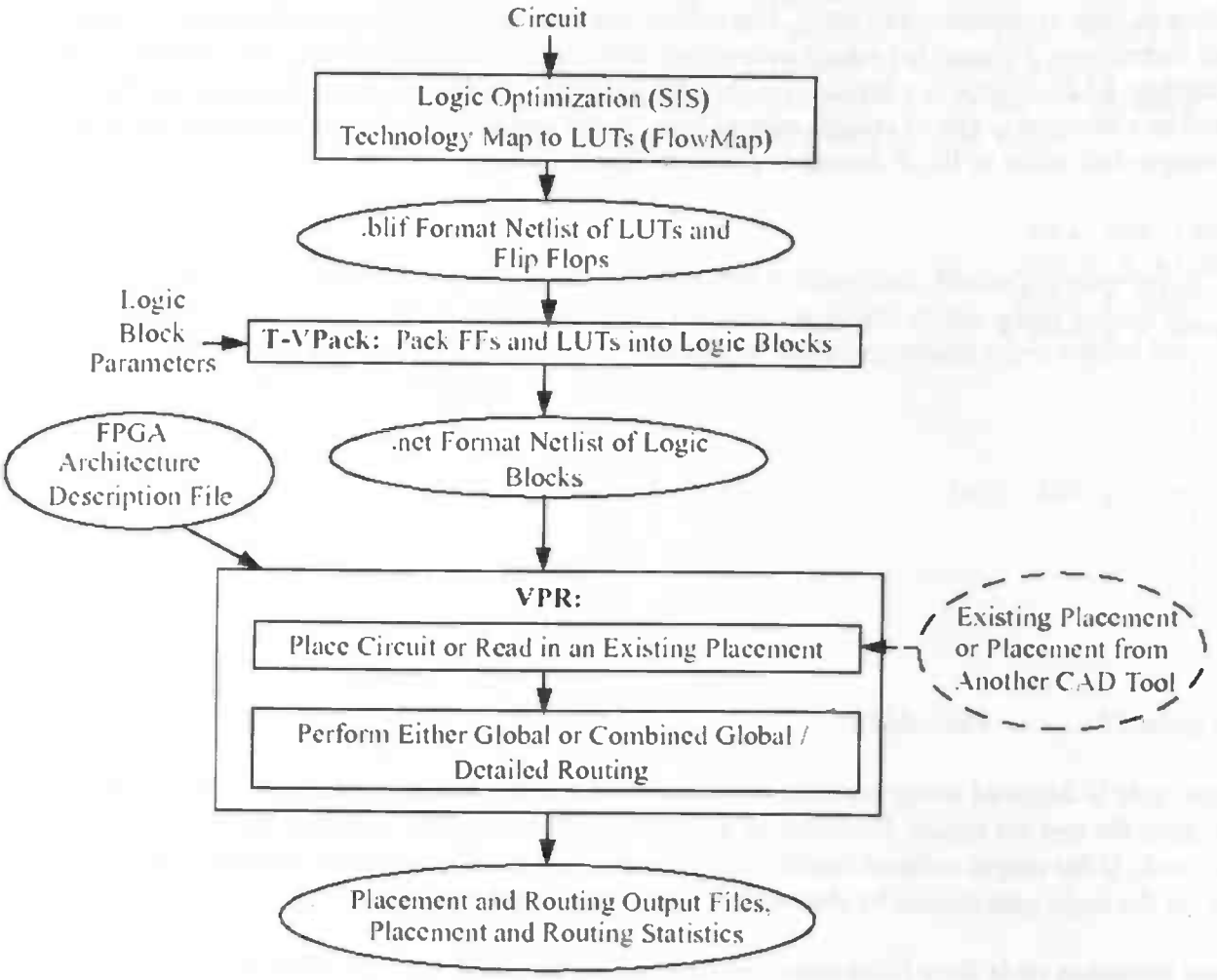

Figure 11 CAD flow

2.4.1 SIS

SIS [6] is an interactive tool for synthesis and optimization of sequential circuits. Given a state transition table, a signal transition graph or a logic-level description of a sequential circuit, it produces an optimized netlist of the circuit. SIS supports a design methodology that allows the designer to search a larger solution space than was previously possible. The synthesis of sequential circuits often proceeds like the synthesis of combinatorial circuits: they are divided into purely combinational blocks and registers. Combinational optimization techniques are applied to the combinational logic blocks, which are later reconnected to the registers to form the complete circuit. This limits the optimization by fixing the optimizing logic only within combinational blocks without exploiting signal dependencies across register boundaries.

SIS employs state-of-the-art synthesis and optimization techniques, using many algorithms. For synchronous systems, these include methods for state assignment, state minimization, testing, retiming, technology mapping, verification, timing analysis, and optimization across register boundaries. The two most common input forms for SIS are a netlist of gates and a finite-state machine in state-transition-table form. The netlist description is given in extended BLIF (Berkeley Logic Interchange Format) [6] which consists of interconnected single-output combinational gates and latches. BLIF describes a logic-level hierarchical circuit in textual form. A circuit can be viewed as a directed graph of combinational logic nodes and sequential logic elements. An example of a simple full-adder in BLIF format is given in Figure 12.

```
.model fulladd
.inputs x y cin
.outputs s cout
.names x y cin s
010 1
100 1
001 1
111 1
.names x y cin cout
110 1
011 1
101 1
111 1
.end
```

Figure 12 Fulladd.blif

A logic-gate is declared using the *.names* statement. The last signal after *.names* is the output of the logic gate, the rest are inputs. Elements of a row are ANDed together, and then all rows are ORed. As a result, if the output column contains only 1's, the first n columns can be viewed as the truth table for the logic gate named by that output. Don't cares can be expressed using '-'.

A state transition table for a finite-state machine can be specified with the KISS format [6]. Each state is symbolic; the transition table indicates the next symbolic state and output bit-vector given a current state and input bit-vector. Figure 13 shows an example of an AND gate in KISS format.

```
.i 2
.o 1
00 state1 state2 0
01 state1 state2 0
10 state1 state2 0
11 state1 state2 1
```

Figure 13 Fulladd.kiss2

SIS operates in text mode. It provides a lot of commands to read input files and apply the mentioned algorithms. The output can be written to a file.

2.4.2 T-VPack

T-VPack [7] takes as input a technology-mapped netlist of look-up tables and flip flops in .blif format, and outputs a .net format netlist composed of more complex logic blocks. The logic block to be targeted is selected via command-line options. The simplest logic block T-VPack can target consists of a LUT and a FF (flip flop). A default LUT size of 4 is assumed by T-VPack. Other LUT sizes can be specified using a command-line option. T-VPack is capable of targeting a more complex form of logic block. You can specify logic blocks consisting of N LUTs and N FFs, along with local interconnect that allows the N cluster outputs to be routed back to LUT inputs.

2.4.3 VPR

VPR (Versatile Place and Route) [8] is an FPGA placement and routing tool. *Placement* consists of choosing a position for each logic block within the FPGA so that the length of the wires needed to interconnect the circuitry is minimized, while *routing* consists of choosing which wires within the FPGA will be used to make each connection.

Chapter 3 Multiplication

An important component of the implementation of a neural network in hardware is the multiplier. There are a lot of ways to do that, varying in size and speed. We consider the conventional series-parallel multiplier, the Booth multiplier and the Modified Booth multiplier. We do not take into consideration the full-parallel nor the full-series multiplier. This is because the former multiplier is far too large for our requirements while the latter is too slow. Of the multipliers studied here, we make four different versions: normal, shift, 2-operands, 2-operands+shift.

3.1 Conventional series-parallel

3.1.1 Normal

The series-parallel multiplier works in the way in which we perform a multiplication with pen and paper. For example:

```
                  0 1 1 1 0 1 0      58 * 15 = 870
                  0 0 0 1 1 1 1  *
                  0 1 1 1 0 1 0       1 * 58 = 58
                0 1 1 1 0 1 0         2 * 58 = 116
              0 1 1 1 0 1 0           4 * 58 = 232
            0 1 1 1 0 1 0             8 * 58 = 464   +
          0 0 0 0 0 0 0                          870
        0 0 0 0 0 0 0
      0 0 0 0 0 0 0                 +
-----------------------
0 0 0 0 1 1 0 1 1 0 0 1 1 0
```

The first bitstring is the controlled one, the second bitstring is the controlling bitstring. We walk through this bitstring from right to left. Everytime we find a `'1'` a shifted version of the first bitstring is added to the result. For the rightmost `'1'` in this example we add `0 1 1 1 0 1 0` (1 * 58), for the next `'1'` we add `0 1 1 1 0 1 0 0` (2 * 58) and so on.

Let's call the controlling bitstring A and the other B. These bitstrings can then be represented as the polynomials:

$$A = a_{n-1} * 2^{n-1} + \ldots + a_0 * 2^0 \text{ and } B = b_{n-1} * 2^{n-1} + \ldots + b_0 * 2^0$$

in which $a_{n-1} \ldots a_0$ are single bits composing the bitstring A.
Using this representation the preceding multiplication can be written as:

$$A * B = \Sigma (a_j * 2^j * B) \text{ for } 0 \leq j < n.$$

3.1.2 Shift

Instead of adding the zero bitstrings in case the current bit of the controlling bitstring is `'0'`, we can use a shift operation to shift to the next `'1'` in the controlling bitstring whenever a `'0'` is encountered. The multiplication will be accelerated because less additions have to be performed.

```
                  0 1 1 1 0 1 0
                  0 0 0 1 1 1 1   *
                  ---------------
                  0 1 1 1 0 1 0
                0 1 1 1 0 1 0
              0 1 1 1 0 1 0
            0 1 1 1 0 1 0         +
-----------------------------------
0 0 0 0 1 1 0 1 1 0 0 1 1 0
```

This can be expressed in the following formula:

$$A * B = \Sigma (2^j * B) \text{ for } a_j \neq 0 \text{ and } 0 \leq j < n.$$

3.1.3 2-Operands

We can perform the series/parallel multiplication on two operands simultaneously. This is a symmetric method in which no controlling bitstring is chosen. The bitstrings are walked through from left to right. This can be expressed in the following recursive formula:

$$A * B = a_j * b_j * 2^{2j} + a_j * 2^j * B_r + b_j * 2^j * A_r + A_r * B_r$$

in which A_r means A_{rest}, that is A after the first bit has been stripped off.
We illustrate this with the following example.

```
                                0 1 1 1 0 1 0
                                0 0 0 1 1 1 1   *
0 * 0 * 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    a6 * b6 * 2^12
                        0 * 0 0 1 1 1 1 1 1 1    a6 * 2^6 * Br
                        0 * 1 1 1 0 1 0 1 1 0    b6 * 2^6 * Ar
    1 * 0 * 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    a5 * b5 * 2^10
          1 0 0 0 0 0 0 * 0 1 1 1 1 1 1 1 1 0    a5 * 2^5 * Br
                        0 * 1 1 0 1 0 1 1 1 0    b5 * 2^5 * Ar
        1 * 0 * 1 0 0 0 0 0 0 0 0 0 0 0 0 0      ...
              1 0 0 0 0 * 1 1 1 1 1 1 1 1 0
                    0 * 1 0 1 0 1 1 1 1 1 0
            1 * 1 * 1 0 0 0 0 0 0 0 0 0 0 0
                  1 0 0 0 * 1 1 1 1 1 1 1 0
                  1 0 0 0 * 0 1 0 1 1 1 1 0
                0 * 1 * 1 0 0 0 0 1 1 1 1 0
                          0 * 1 1 1 1 1 1 0
                      1 0 0 * 1 0 1 1 1 1 0
                    1 * 1 * 1 0 0 1 1 1 1 0
                          1 0 * 1 1 1 1 1 0
                          1 0 * 0 1 1 1 1 0
                        0 * 1 * 1 1 1 1 1 0  +
-----------------------------------------------
0 0 0 0 1 1 0 1 1 0 0 1 1 0
```

3.1.4 2-Operands + shift

Again this can be accelerated by not performing the additions with zeroes. In this case both operands are searched for the most significant bit in each step. This leads to the original Digilog formulation [9]. The formula is as follows:

$$A * B = 2^j * 2^k + 2^j * B_r + 2^k * A_r + A_r * B_r$$

in which j is the index of the most significant bit (highest '1' bit) of A and k is the index of the most significant bit of B. The multiplication is ready when one of the operands becomes zero. The calculation will then run as:

```
                0 1 1 1 0 1 0
                0 0 0 1 1 1 1   *
          1 0 0 0 0 0 0 0 0       2^5 * 2^3
      1 0 0 0 0 0 * 1 1 1         2^5 * Br
      1 0 0 0 * 1 1 0 1 0         2^3 * Ar
            1 0 0 0 0 0 0         2^4 * 2^2
          1 0 0 0 0 * 1 1         2^4 * Br
          1 0 0 * 1 0 1 0         2^2 * Ar
                1 0 0 0 0         ...
              1 0 0 0 * 1
                1 0 * 1 0
                    1 0   +
-----------------------------
0 0 0 0 1 1 0 1 1 0 0 1 1 0
```

An advantage of these types of multiplication where we run through the operands from left to right is that the calculation can be stopped at a certain moment while the result has already approached the final result (in case we completed the calculation). This is because we start with the most significant part of the operands and work towards the least significant part.

3.2 Booth

A widely used multiplication method is Modified Booth. Before we take a look at that we consider the standard Booth method [10]. This method is based on a recoding of the operands and instead of only additions it uses additions and subtractions. It works on bitstrings that are in two's complement representation. With $p_j = a_{j-1} - a_j$ we can rewrite the number representation of A as

$$A = p_{n-1} * 2^{n-1} + \ldots + p_0 * 2^0 \text{ with } a_{-1} = 0 \text{ and } p \in \{-1, 0, 1\}.$$

The recoding mechanism is given in Table 2.

Table 2 Booth algorithm

a_i	a_{i-1}	p_i	Operation
0	0	0	+0
0	1	1	-B
1	0	-1	+B
1	1	0	-0

As an example we give two bitstrings and their Booth encoding.

```
 0  0  1  0      0  1  1  1  1
 0  1 -1  0      1  0  0  0 -1
```

We see that 2 is the same as -2 + 4 and 15 is the same as -1 + 16.

3.2.1 Normal

The normal Booth multiplication is described in the formula

$$A * B = \Sigma (p_j * 2^j * B) \text{ for } 0 \leq j < n$$

with A as the recoded controlling number. For the example in the previous section we get:

```
                                0  1  1  1  0  1  0
                                0  0  1  0  0  0 -1   *
 1  1  1  1  1  1  1  1  0  0  0  1  1  0
                   0  0  0  0  0  0  0
                0  0  0  0  0  0  0
             0  0  0  0  0  0  0
          0  1  1  1  0  1  0
       0  0  0  0  0  0  0
    0  0  0  0  0  0  0                             +
 0  0  0  0  1  1  0  1  1  0  0  1  1  0
```

3.2.2 Shift

Skipping the additions of zeroes gives us the following formula:

$$A * B = \Sigma (2^j * B) \text{ for } p_j \neq 0 \text{ and } 0 \leq j < n.$$

```
                                0  1  1  1  0  1  0
                                0  0  1  0  0  0 -1   *
 1  1  1  1  1  1  1  1  0  0  0  1  1  0
          0  1  1  1  0  1  0                        +
 0  0  0  0  1  1  0  1  1  0  0  1  1  0
```

3.2.3 2-Operands

The Booth multiplication method can also be performed on two operands. In this case we encode both of the operands and then walk through their operands from left to right simultaneously. The formula will be like this:

$$A * B = p_j * q_j * 2^{2j} + p_j * 2^j * B_r + q_j * 2^j * A_r + A_r * B_r.$$

Following this method and omitting the terms that are multiplied by zero the example will be:

```
                                1  0  0 -1  1 -1  0
                                0  0  1  0  0  0 -1   *
                             ------------------------
1  0  0  0  0  0  0  0          *  0  1  0  0  0 -1
               1  0  0  0  0  0  * -1  1 -1  0
                     -1  0  0  0  0  *  0  0 -1
                         1  0  0  0  *  0 -1
                            -1  0  0  *  0 -1
                            -1  0  *  0 -1   +
                             ------------------------
0  1  1  0  1  1  0  0  1  1  0
```

3.2.4 2-Operands + shift

Instead of walking through both operands step by step we now again search for the most significant bit of each operand every step. The resulting multiplier looks like the original Digilog multiplier. The formula becomes

$$A * B = p_j * 2^j * q_k * 2^k + p_j * 2^j * B_r + q_k * 2^k * A_r + A_r * B_r.$$

where j is the index of the most significant bit of A and k is the index of the most significant bit of B. The example will in this case looks like:

```
                                1  0  0 -1  1 -1  0
                                0  0  1  0  0  0 -1   *
                             ------------------------
1  0  0  0  0  0  0  0  0  0  0
         1  0  0  0  0  0  0  0  0  * -1
      1  0  0  0  0  0  * -1  1 -1  0
                           1  0  0  0
                        -1  *  1 -1  0   +
                             ------------------------
0  1  1  0  1  1  0  0  1  1  0
```

3.3 Modified Booth

The normal Booth encoding is based on overlapping pairs of two bits in the bitstring. Modified Booth however is based on overlapping groups of three bits (triplets). With $p_{2j} = a_{2j-1} + a_{2j} - 2\,a_{2j+1}$ we can rewrite the number representation of A as

$$A = p_{2n} * 2^{2n} + \ldots + p_0 * 2^0 \text{ with } a_{-1} = 0 \text{ and } p \in \{-2, -1, 0, 1, 2\}.$$

The recoding mechanism is given in Table 3.

Table 3 Modified Booth algorithm

a_{j+1}	a_j	a_{j-1}	p_j	Operation
0	0	0	0	+0
0	0	1	+1	+B
0	1	0	+1	+B
0	1	1	+2	+2B
1	0	0	-2	-2B
1	0	1	-1	-B
1	1	0	-1	-B
1	1	1	0	-0

For instance, the modified booth recoding of the bitstring `1011010` is `1 0 2 0 -1 0 -2`. We see that 64 + 2*16 - 4 - 2*1 = 90 = 1011010. We find this coding by placing a 0 at the end of the bitstring and then taking the triplets from right to left with one overlapping bit for every pair of triplets. Then we look up the according values of p_j in the table.

3.3.1 Normal

The formula for the Modified Booth multiplication is

$$A * B = \Sigma (p_j * 2^j * B) \text{ for } 0 \leq j < n \text{ and } j \text{ even and } p \in \{-2, -1, 0, 1, 2\}.$$

To demonstrate the multiplication we use a different example than before. We calculate 10111 * 0110 = 23 * 6 = 138. The encoding of 0110 is 0 0 2 0 -2.

```
                    1  0  1  1  1
                    0  0  2  0 -2   *
              ------------------------
1  1  1  1  0   1  0  0  1  0
      1  0  1   1  1  0
   0  0  0  0   0                    +
-------------------------------------
0  0  1  0  0   0  1  0  1  0
```

3.3.2 Shift

Omitting the zeroes leads to the formula

$$A * B = \Sigma (p_j * 2^j * B) \text{ for } 0 \leq j < n \text{ and } p_j \neq 0 \text{ and } j \text{ even and } p \in \{-2, -1, 0, 1, 2\}.$$

The example now looks like:

```
                    1  0  1  1  1
                    0  0  2  0 -2   *
              ------------------------
1  1  1  1  0   1  0  0  1  0
      1  0  1   1  1  0             +
-------------------------------------
0  0  1  0  0   0  1  0  1  0
```

3.3.3 2-Operands

Like before it is possible to work on 2 operands simultaneously. In this case the formula becomes

$$A * B = p_j * q_j * 2^{2j} + p_j * 2^j * B_r + q_j * 2^j * A_r + A_r * B_r.$$

Both operands are walked through from left to right step by step. We give the example omitting the values in which a multiplication by zero occurs.

```
                          1  0  2  0 -1
                          0  0  2  0 -2   *
                       ------------------
   1  0  0  0   0   *   2  0 -2          96
   2  *  2  *   1   0   0  0  0          64
                2   0   0  * -2         -16
                2   0   0  * -1          -8
                       -1  * -2   +      2  +
                       ------------------
0  0  1  0  0   0  1   0  1  0          138
```

3.3.4 2-Operands + shift

In this final case we shift through the operands from left to right skipping zeroes. The formula is as follows:

$$A * B = p_j * 2^j * q_k * 2^k + p_j * 2^j * B_r + q_k * 2^k * A_r + A_r * B_r.$$

The example is given by

```
                          1  0  2  0 -1
                          0  0  2  0 -2   *
                       ------------------
1  *  2  *  1  0   0   0  0  0  0          128
            1  0   0   0  0  * -2          -32
            2  0   0   *  2  0 -1           56
            2  * -2   *  1  0  0          -16
                       -2  * -1   +      2  +
                       ------------------
0  0  1  0  0   0  1   0  1  0          138
```

3.4 *Modes of multiplication*

Before we draw any conclusions, lets review the modes of multiplication: seriel, serias/parallel and fully parallel.

3.4.1 Serial

The easiest way to compute the formula we can perform each multiplication one after the other and then add all the results together. Better performance is achieved by adding the result of the multiplication to a temporary value, which holds the sum so far (Figure 14).

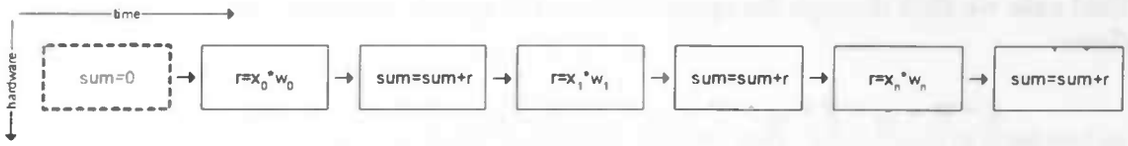

Figure 14 Serial computation

3.4.2 Parallel

However, because the multiplications are totally independent of each other it is possible to compute them at the same time. This is the parallel approach to the problem. But there are two problems with the parallel method. First of all, the summation of all the results isn't independent completely, so it is parallelizable to a lesser extent. Secondly more hardware is needed for parallel computation than for serial. In Figure 15 it takes 8 multiplication units and 4 summation units to have the final answer in 4 steps. This is of course much faster than the 16 steps it takes with the serial approach (for this number of synapses).

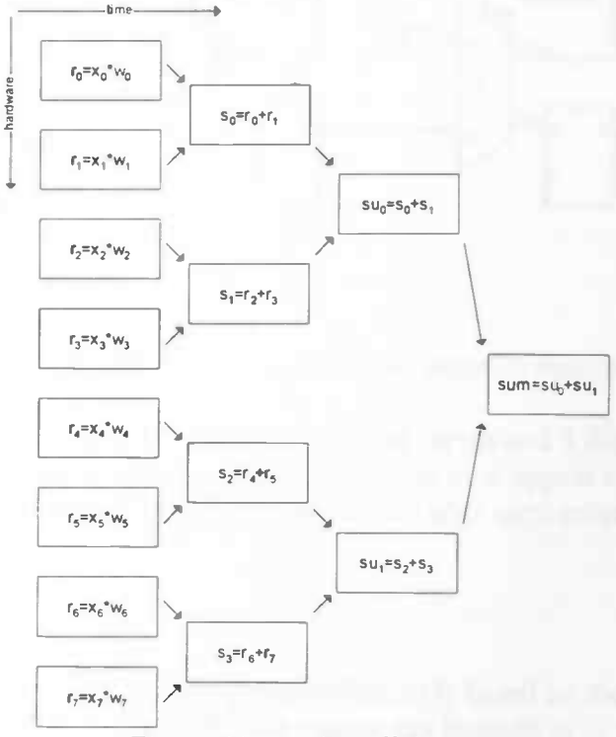

Figure 15 Parallel computation

3.4.3 *Pipeline computation*

In order to find a compromise between the low cost of the serial solution and the high speed of the parallel version a pipeline is a good option. It is both cheap and speedy, but it costs more effort to
develop. One has to split the whole calculation in the smallest parts possible; then it should be possible to use one computational unit of each kind, but have them occupied as much as possible.

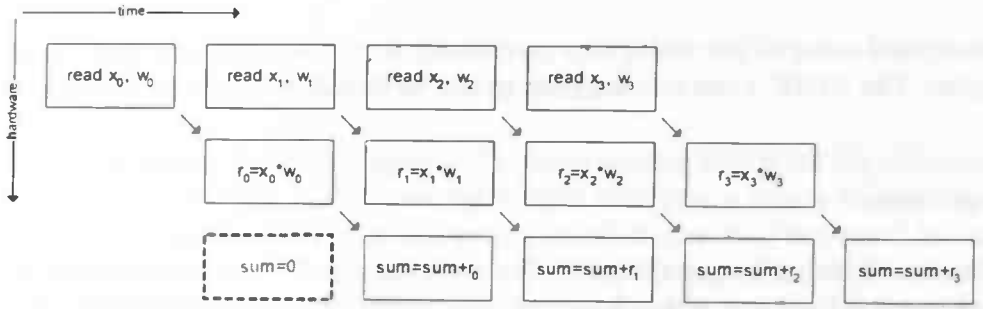

Figure 16 Pipeline computation

To optimize a pipeline, timing is critical. We have to determine how many clock cycles each step of the pipeline needs. If the multiplication takes twice as long as the summation, for example, it is useful to have a second multiplication unit in the pipeline. For this optimization the memory access has to be studied as well. We want to make sure to have the data ready in time for computation. In Figure 16 memory access is shown as 'read x, w', but in reality these are 2 operations which can't be performed simultaneously when using a single-port memory. However, if the memory can supply the data fast enough it can be directly fed to the multiplication unit, without the need for an intermediate register. The use of registers isn't shown either, but you'll want to try and use as little registers as possible.

3.4.4 *Remarks*

We have seen three types of multipliers. The series-parallel multiplier is the smallest. It multiplies two unsigned numbers. One result bit is produced in every clock cycle. So, if the operands are N bits long, the multiplication will take 2N clock cycles.

The Booth multiplier is based on the series-parallel multiplier. However it is bigger because the Booth encoding has to be performed. An important difference with the series-parallel multiplier is that this multiplier works on two's complement bitstrings. As with the series-parallel multiplier one output bit is produced every clock cycle.

The Modified Booth multiplier is about 33 percent larger than the Booth multiplier. However, this multiplier runs twice as fast. This is because every clock cycle two output bits are produced. Modified Booth also works on two's complement bitstrings.

Most of our interest goes to the series-parallel multiplier because it is very small and to the Digilog multiplier because it's calculation can be truncated while the intermediate result has

already approached the final result. This kind of truncation can be very useful in a situation in which you can take advantage of starting the next calculation while omitting the least significant part of the current multiplication. Further the Digilog multiplier is pretty fast. The number of clock cycles needed is the number of ones of the operand that has the least number of ones.

3.5 Implementation of the multipliers

We have implemented some of the multipliers mentioned above. We start with the series-parallel multiplier. The VHDL code that we refer to can be found in Appendix A.

3.5.1 Series-parallel

The implementation of the series-parallel multiplier is quite easy. The circuit is depicted in Figure 17. There are 4 full-adders, 4 flip flops that contain the carry's and 4 flip flops that contain the result. Every clock cycle the complete bit string B and one of the bits of the controlling operand A are presented to the circuit. The AND gate lets through the value of B when Aj='1'. After the addition the carry's are stored in the upper flip flops and the result bits in the lower flip flops, through which they proceed to the right. Every clock cycle one result bit (Pj) appears at the bottom on the right.

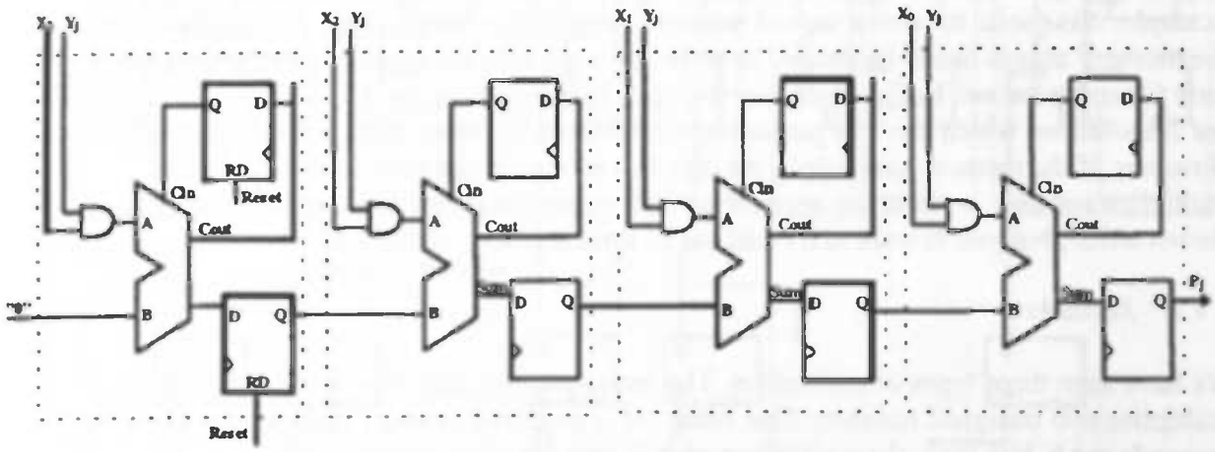

Figure 17 4x4 bit series-parallel multiplier

In our implementation we added 2 x 4 flip flops for the inputs and 8 flip flops for the output. By connecting the flip flops in which A is stored, the bits of A appear one by one. Multiple multiplications can be performed by this circuit directly after each other. Every 8 clock cycles a result is ready.

3.5.2 Digilog

As mentioned in section 3.1.4 the Digilog multiplication is based on the following formula $A * B = 2^j * 2^k + 2^j * B_r + 2^k * A_r + A_r * B_r$. We change the formula in $A * B = 2^j * B + 2^k * A_r$

+ A_r * B_r that has two additions instead of three. Roughly three actions have to be taken. First of all we need to split A into 2^j and A_r (since j is the index of the most significant bit of A and A_r is A with the most significant bit set to zero). The same should be done for B. Then the multiplications by the powers of 2 have to be performed. This is done by shifting the values to the left using a barrel shifter. Last of all the results have to be added. These actions will be taken by three components:

- slbmsbnz
- barrel
- adder16

The VHDL code is listed in Appendix A.

The component slbmsbnz has three outputs. Slb ('strip leading bit') gives the bitstring with the most significant bit set to zero. Msb ('most significant bit') gives a binary value of the most significant bit of the input, which is needed by the barrel shifter. Nz ('not zero') indicates whether the bitstring is zero. This is used for determining the end of the calculation. To search for the most significant bit and to set this bit to zero we use the circuitry, shown in Figure 18.

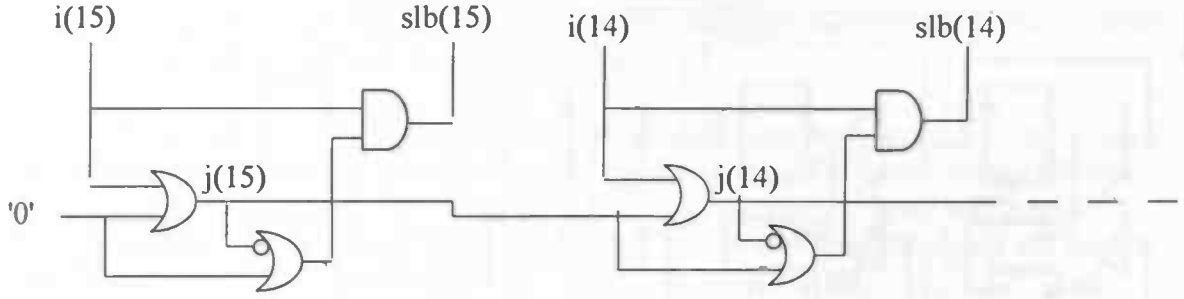

Figure 18 Strip leading bit circuitry

The input is i(15) ... i(0). The output is given in slb(15) ... slb(0). Suppose zeroes appear at i first. The inverter then outputs a '1' to the AND gate, so slb will get the value of i. When the first '1' arrives at i, the inverter will output a '0' and the lowermost input of the lowermost OR gate will still be '0', so the AND gate will have it's lower input set to zero. This will suppress this '1'. When thereafter other '1'-s appear at i, the lowermost input of the lowermost OR gate will always be '1', so the lowermost input of the AND gate will always be '1', so i is passed to slb.

In this circuitry j(15) ... j(0) will have all zeroes as long as the input string consists of zeroes, and from the most significant bit of the input string it has all ones. So when the input string is `00010100` j will be `00011111`. We use a combination of OR and XOR gates on j to find the binary value of the most significant bit.

As said a barrel shifter is used to multiply by the powers of 2. The 16 bit barrel shifter works in four stages. A binary value (msb) of 4 bits determines how much the shifter will shift. If bit 0 is '1', he shifts 1 place, otherwise 0 places. If bit 1 is '1', he shifts 2 places, otherwise 0 places. If bit 2 is '1', he shifts 4 places, otherwise 0 places. If bit 3 is '1', he shifts 8 places, otherwise 0 places. In total he can shift between 0 and 15 places.

The last component is a normal 16 bit adder. This adder doesn't have a carry in and carry out, because we know that these carry's won't be used. This is because we start with zero and the result of a 8 bit multiplication can never be longer than 16 bits. Appendix A, section A.2 lists the VHDL code of this multiplier. Because this multiplier is quite large, in the next chapter we will change this design into a pipelined one.

3.5.3 Booth

In an article about Booth multipliers [10] we found a circuit that implements a 4 x 4 bit Booth multiplier. This circuit is given in Figure 19. The upper part of the circuit looks just like the series-parallel multiplier. At the bottom some circuitry is added that calculates the Booth encoding. In case an addition has to be performed, the Booth encoding gives the input bitstring for addition. In case a subtraction has to be performed, the Booth encoding gives the inverse of the input bitstring. In that case the two flip flops at the bottom right provide a '1' that is also added to the result. This is because a negative number in two's complement notation is found by inverting it's positive value and increasing it with 1.

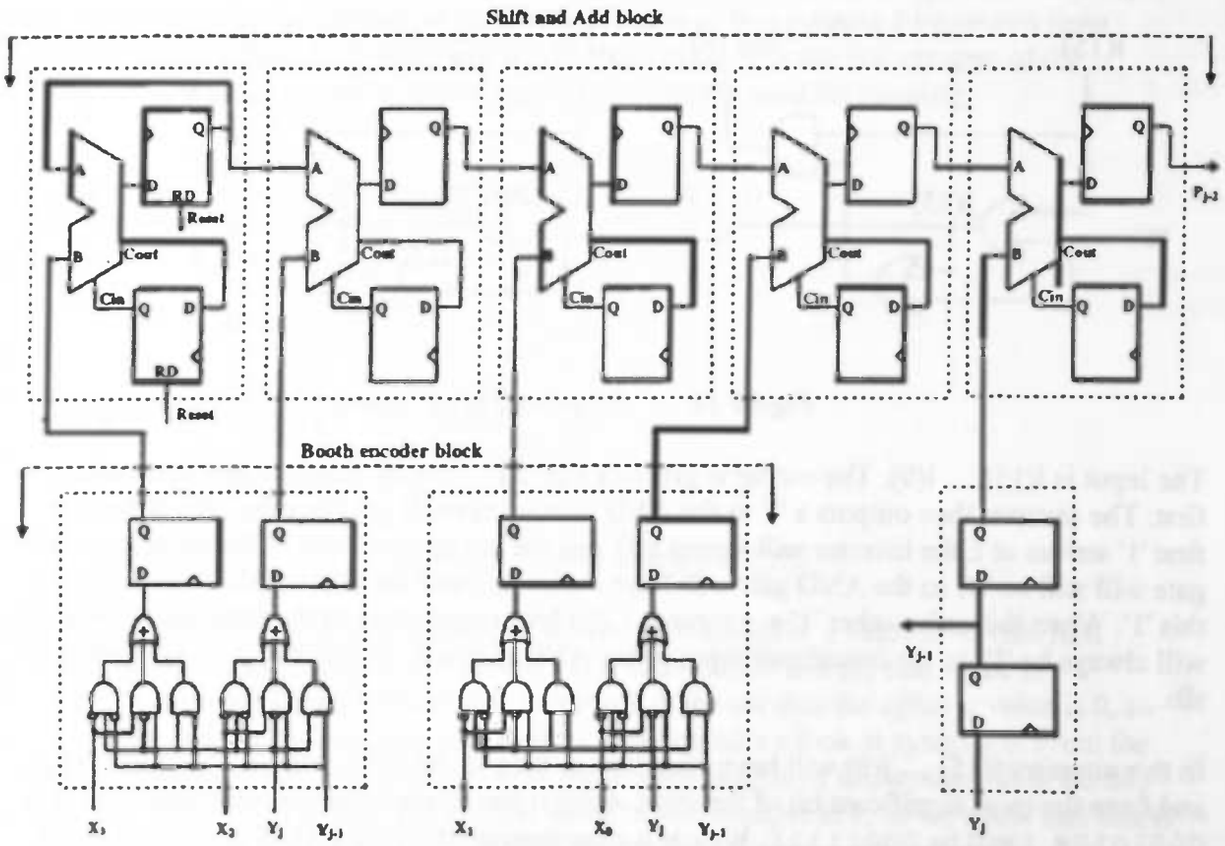

Figure 19 4x4 bit Booth multiplier

Chapter 4 Neural Network Implementation

We will limit our attention to feed-forward neural networks. For our experiments such networks are trained elsewhere and the resulting file is available in the Simple Neural Format (SNF). Eventually a tighter integration of the neural and digital world should allow for a direct generation of the VHDL NoC description.

4.1 The SNF system

The SNF system is a collection of programs with a shared file system, that was originally meant for the remote monitoring of small intelligent devices (Figure 20). Stimuli can be either provided or generated to be stored for training, validation and test purposes in *.lrn files. Which signals are to be presented at which nodes of the network is shown in the *.pck files. This notation also supports the external use of delay lines.

New or existing networks can use such files for (post) learning and versions may be saved in *_save.snf files. Larger networks can be created from empty or trained existing ones. Iterative parallel and series assemblies are supported. As post training of assembled networks may have influence on the original weight settings within a module, one may also retrieve the post-trained module from the assembly and look for the differences.

Small programs are built around the file store to support such elementary actions as train, verify, test, assemble and dis-assemble. The file format is primitive and ASCII based, which allows for an easy addition to the available tool set. General-purpose routines such as signal and database handling are available from a library. Typical examples of such newer additions are the generators for network descriptions in VHDL and in C.

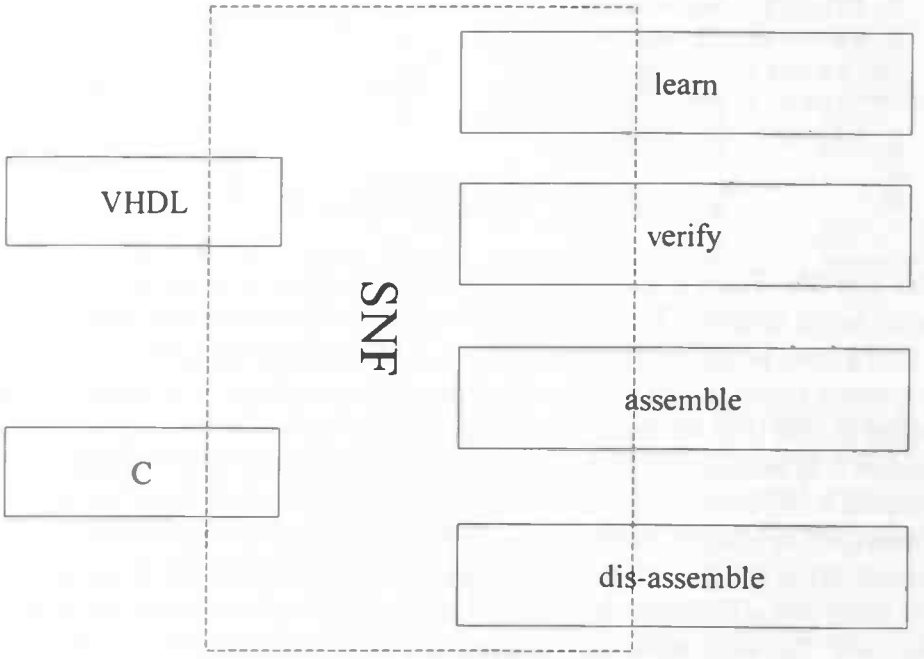

Figure 20 The SNF system architecture

4.2 *The SNF file format*

We use the SNF system for calculating the values of the neurons in the network. The SNF system leaves the neurons in order so that it is possible to calculate the value of all neurons sequentially from begin to end. We start with the top neuron in the first hidden layer and then go down. After all neuron values of this layer have been calculated we proceed to the top neuron of the second hidden layer. In this way the neuron values of all hidden layers plus the output layer are calculated.

The SNF system creates neural network description files called *.snf* files. An example *.snf* file is given in Figure 21. This file describes a complete neural network. The character 'N' indicates a neuron, the character 'S' indicates a synapse. All neurons are numbered consecutively, starting at 0. So this network contains neurons 0 till 4, inclusive. The synapses are numbered 0 till 5, inclusive. The values after the 'N' are for the *value*, *bias*, *old_bias*, *error* and *offset_s* respectively. *Value* gives the neuron value of the neuron. In this case all neuron values are 0.000000. The bias value of each neuron is given in the next colomn. Thereafter the *old_bias* and *error* are given. These are used for learning. The last value, *offset_s*, gives the number of the first incoming synapse of the neuron. For input neurons, this value is -1. The values after 'S' are for the *offset_n*, *lasts*, *weight* and *old_weight* respectively. *Offset_n* gives the number of the feeding neuron of that synapse (the neuron from which this synapse originates). *Lasts* indicates whether this synapse is the last synapse of the current neuron. *Weight* is the weight of the synapse, *old_weight* is used for learning.

```
N     0.000000   0.654037   0.654037   0.000000   -1
N     0.000000   -0.568264  -0.568264  0.000000   0
N     0.000000   0.050154   0.050154   0.000000   1
N     0.000000   0.732548   0.732548   0.000000   2
N     0.000000   -0.530047  -0.530047  0.000000   3
S     0     1    0.606643   0.606643
S     0     1    0.481455   0.481455
S     0     1    0.665364   0.665364
S     1     0    -0.864217  -0.864217
S     2     0    -0.155977  -0.155977
S     3     1    0.380069   0.380069
```

Figure 21 An example SNF file

Let's take a look at this *.snf* file. There is only one neuron with an *offset_s* value of -1 (the first neuron). This is the only input neuron. The neuron value is 0.000000 and the bias value is 0.654037. When we take a look at the next neuron, neuron 1, we see that the *offset_s* value is 0, so synapse 0 is the first synapse running into this neuron. Now we take a look at synapse 0. From the *offset_n* value of synapse 0, that is 0, we know that the feeding neuron of this synapse is 0. So this synapse runs from neuron 0 to neuron 1. The *lasts* value of this synapse is 1, so we know that this is the last synapse for neuron 1. Now we proceed with neuron 2. In the same manner we see that synapse 1 runs from neuron 0 to neuron 2 and synapse 2 runs from neuron 0 to neuron 3. The first synapse of the last neuron (4) is synapse 3. The *lasts* value for this synapse is 0, so the next synapse will run to neuron 4 as well. Since the *lasts* value for this synapse is 0 as well, synapse 5 will also be connected with neuron 4. The *lasts* value of synapse 5 is 1 so this is the last synapse running into neuron 4. All together we have 1 input neuron, 3 hidden neurons and 1 output neuron, all fully connected.

For the implementation of neural networks in hardware we use these *.snf* files. I wrote a C program that reads an *.snf* file and transforms it into parts of VHDL files, containing the declarations of the neurons and synapses. The rest (static parts) of the VHDL code is appended to it. On chip finally this SNF method is used to gather the right neuron values and corresponding synapse weights or biases which then will be multiplied and summed up.

4.3 Behaviour

In section 4.2 we have seen that for the calculation of the output of a neuron a number of multiplications has to be performed which have to be summed up. What we actually have to do is to start with the first hidden neuron and multiply its bias by -1. Then we should walk through all the synapses of that neuron and multiply their weights by the corresponding neuron values of the neurons feeding those synapses. The results should be summed up and finally a transfer function will generate the output of the current neuron. Then we proceed to the next neuron. We will perform these actions according to the scheme given in Figure 22.

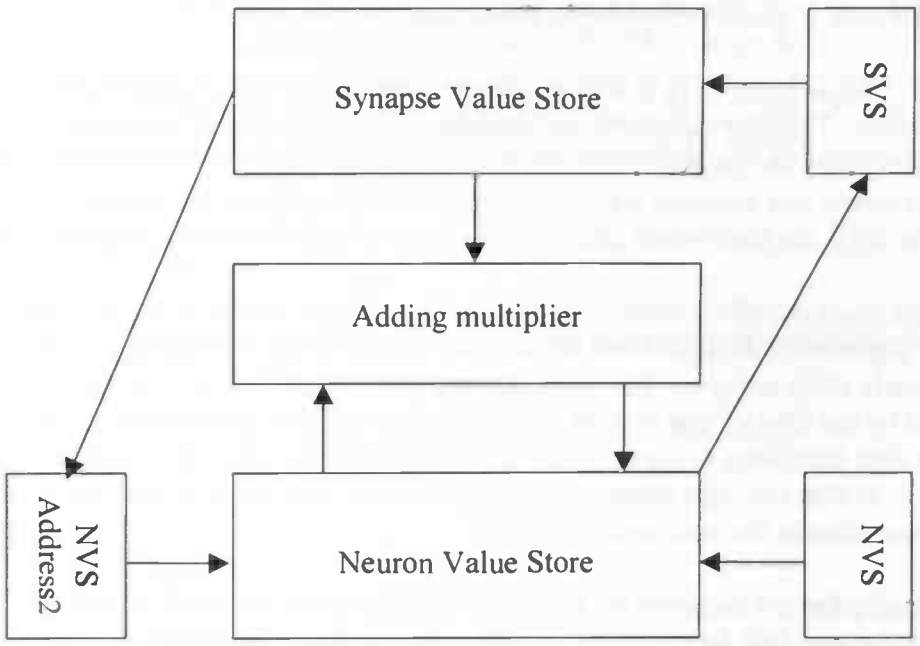

Figure 22 Storage of parameters

The information of the *.snf* file is stored in the Neuron Value Store and the Synapse Value Store. We need three address counters to point to the right positions in these stores while running the calculation of the neural network. The first address counter, NVS Address1, contains the index of the neuron we are currently working on. For each neuron we have to walk through all incoming synapses. SVS Address will contain the indices of the synapses. The weight of each synapse has to be multiplied by the output signal of the corresponding feeding neuron. The index of this feeding neuron is stored in NVS Address2. As you can see there is a delay in the system. The weights from the Synapse Value Store should be multiplied by the neuron values from the Neuron Value Store, but NVS Address2 that points to the neuron values can just be read from the Synapse Value Store after SVS Address has been set to the right value.

4.4 *Value Stores*

Instead of using a Neuron Value Store and a Synapse Value Store we split the Neuron Value Store in a Neuron Value Store and a Bias Value Store. Table 4 indicates where all values can be found.

Table 4: Value Stores

Neuron Value Store	Bias Value Store	Synapse Value Store
neuron value	bias + sign offsets	synweight + sign offsetn last

We made four addresses that point to these value stores: nvsadd, bvsadd, svsadd and svsadd2. So there are two addresses that point to the Synapse Value Store. This is necessary because of the delay in the design, as described in section 4.3. The weights appear one clock cycle before the neuron values by which they have to be multiplied. To solve this problem we build a delay into the system so that the weights and the neuron values can be put into the multiplier simultaneously. The VHDL code of the neural network is given in Appendix B, section 1.

In an earlier paper by Diepenhorst [20], it was suggested to solve the above problem by a different filling of the value stores. The first weight in the synapse value store is to be the bias; the actual weight of the synapse comes on the next location and is therefore read at the same time as the neuron output. We rejected this solution for the simple reason that this would require a transformation on the SNF file that would place the synapse information in an illogical order.

The values of the addresses nvsadd, svsadd, svsadd2 and bvsadd are stored in the flip flops that can be written using the signals d3, d2, d2del and d1 and can be read using the signals q3, q2, q2del and q1 respectively. Initially d2 is set to -1. This indicates that we start with a new neuron and this -1 should be multiplied by the bias of that neuron. After the bias has been multiplied, d2 will get the value of offsets (the first incoming synapse) and will be increased by 1 for all synapses. In d3 offsetn will be stored, so that the right neuron value can be read. The value of last will be checked to take care of the transition to the next neuron.

The outputs of the multiplier are summed up by the 16 bits adder and when all synapses of a neuron have been processed the sum will be put on the signal called 'activity' (internal activity of the neuron). The transfer function will be applied to this activity signal and the resulting neuronvalue will be stored in the Neuron Value Store.

4.5 *Transfer function*

In neural networks a sigmoid function is often used as transfer function. This function looks like $1/1+e^{-g*v}$, where g is the gradient (slope) of the function and v is the internal activity level of the neuron. An example for g=1 is given in Figure 23.

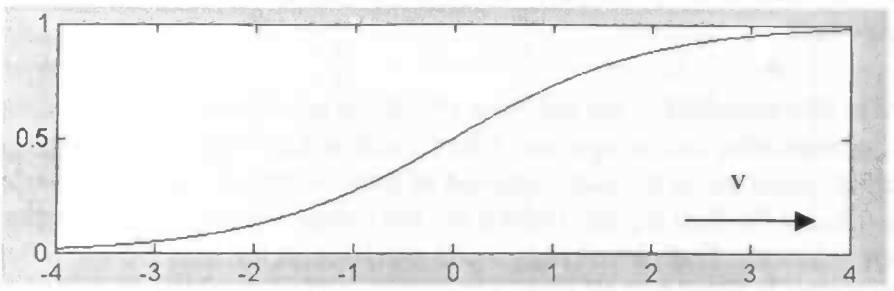

Figure 23 Sigmoid with g = 1

The transfer function can be implemented in two ways. In the first way we make use of a look-up table. We simply put a number of *v* values and their corresponding function values into a table. Previous research has shown that a table with 20 inputs performs reasonably well. The value of *v* is rounded off to the nearest value in the table and the corresponding function value is given. Of course you can put into the table any function you like.

A disadvantage of a look-up table is that all table entries have to be stored on the chip and that takes space. Therefore in this second approach we calculate the function value instead of putting it into a table. By having just an adder and a multiplier it is impossible to calculate the fraction given in the formula. Since we already have the multiplier, we will approximate the sigmoid using squares. We found out that a sigmoid with a gradient of 4 can be approximated by the following squares:

$$y = -(x-1)^2 + 1 \quad for\ x \in [0, 1]$$
$$y = (x+1)^2 - 1 \quad for\ x \in [-1, 0]$$

This is illustrated in Figure 24, in which the blue line is the sigmoid and the yellow line are the squares.

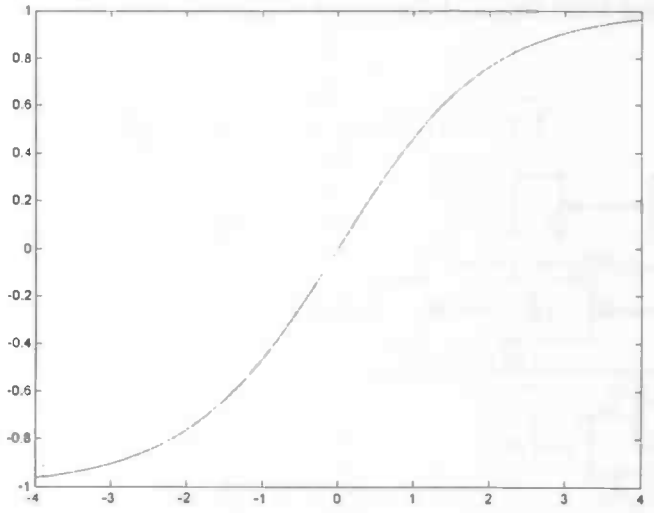

Figure 24 Sigmoid approximation

4.6 *Werner diagrams*

The Digilog multiplier as mentioned in the previous chapter is quite large. Therefore we will put the components of the multiplier into a pipeline. When constructing a pipeline the design is divided into parts that preferably need about the same amount of time. When the system is started, the first part of the system works on the first inputs. During the next stage the second part works on the result of the first part, while the first part already starts working on the next inputs. After a latency period the complete system is filled and all parts are working. The advantage is that now every stage a result becomes available, so it just takes the time needed for one part of the pipeline to get each new result.

Pipelined designs can very well be graphically depicted using so called Werner diagrams [11]. The Werner diagram is a schematic convention for clock cycle true models of synchronous systems. All clocked elements are lined up onto a set of vertical lines, and combinational blocks are drawn in between those lines. The main computational flow through the diagram is from left to right in step with the clock, complemented by appropriate feedback paths as needed. The clock-cycle sequence can be unrolled to better visualize the computation in terms of allocation and scheduling of operations.

As an example consider the circuit given in Figure 25. Figure 25 A shows a conventional schematic drawing. We just left out the clock signal that runs to every flip flop. We now draw all flip flops on top of each other to emphasize the clock-controlled data transport mechanism (Figure 25 B) and draw a vertical line through all the flip flops. This vertical line can be considered as a replacement of the clock signal. In the last stage as shown in Figure 25 C the clock sequence has been unrolled to better reflect the computational flow. From this figure you can clearly see what computation takes place during each stage of the pipeline.

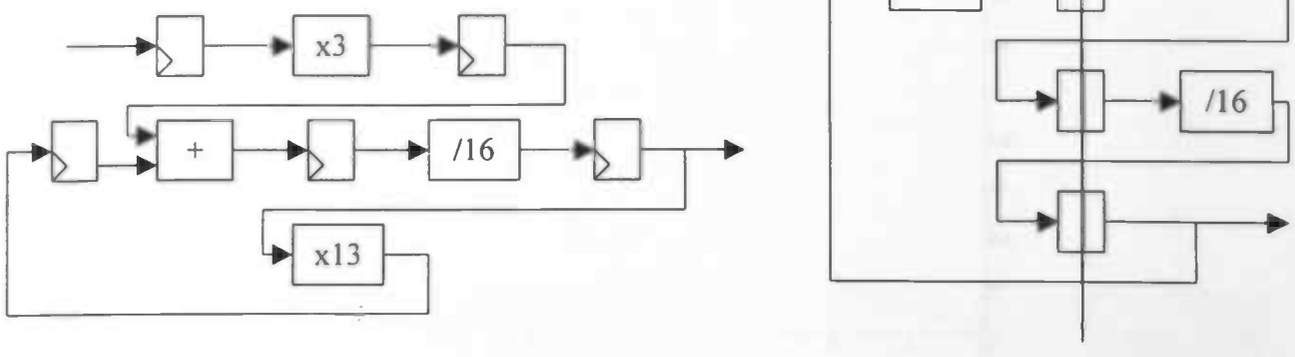

Figure 25 Werner diagram. (A) conventional schematic (B) Flip flops vertically aligned

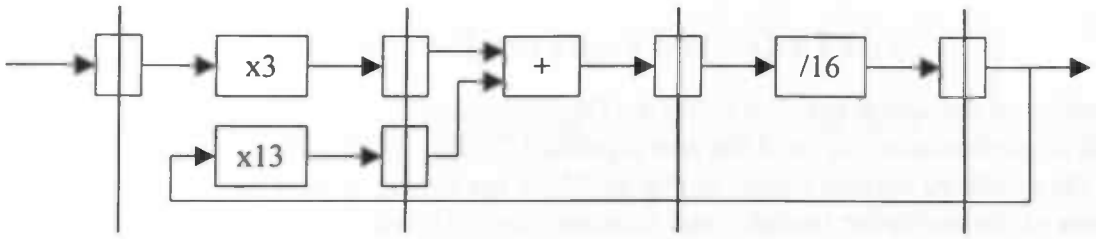

Figure 25 (C) A four-line Werner diagram where the clock sequence has been unrolled

Instead of using a Werner diagram on a structural description, you can also apply it on a behavioral description. That is what we do for the pipeline of the Digilog multiplier. We would like to make a system in which we have only one instance of each of the three components mentioned in the previous chapter (slbmsbnz, barrel and adder16). The pipeline is given as a Werner diagram in Figure 26. We can see that the components are used for the A and B operands alternately. At some times in the pipeline 'R' should be set to zero (when the previous multipication is ready). It can be nicely seen that at all moments all hardware components are in use. The VHDL code of this multiplier is given in Appendix A, section A.4.

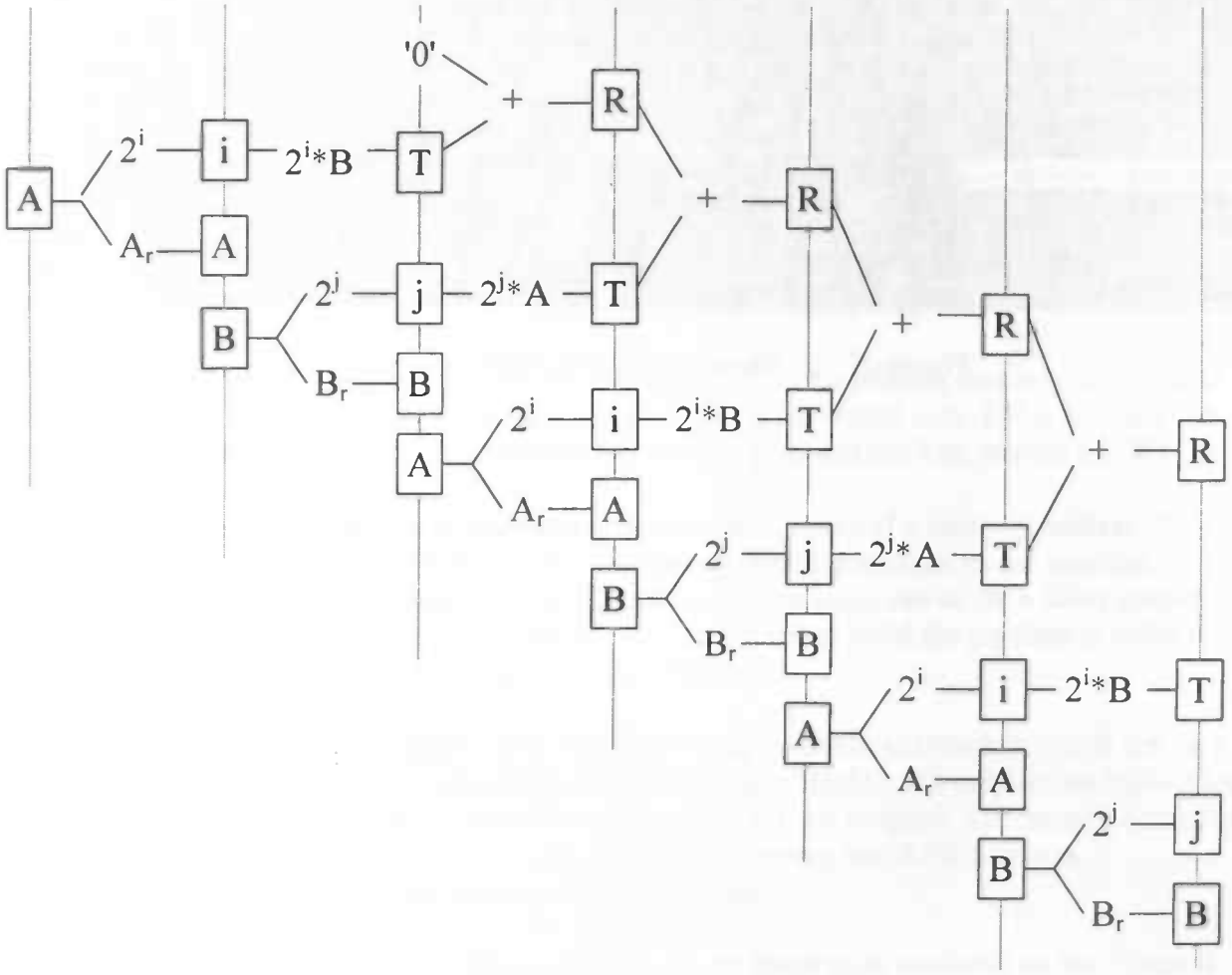

Figure 26 Werner diagram for behavioral description of the Digilog multiplier

4.7 *Results*

The implementation of the neural network using the Digilog multiplier is given in Appendix B. Note that in this implementation we used the non-pipelined Digilog multiplier, though this one can be replaced by the pipelined version easily. In Figure 27 we see the simulation results. a1 and a2 contain the inputs of the multiplier (weights and neuronvalues). The result of the summed up multiplications is given in the signal z3. One can nicely see how many clock cycles the Digilog multiplier needs for each multiplication and that when the multiplication is ready (indicated by the mrdy signal) the following values appear at a1 and a2.

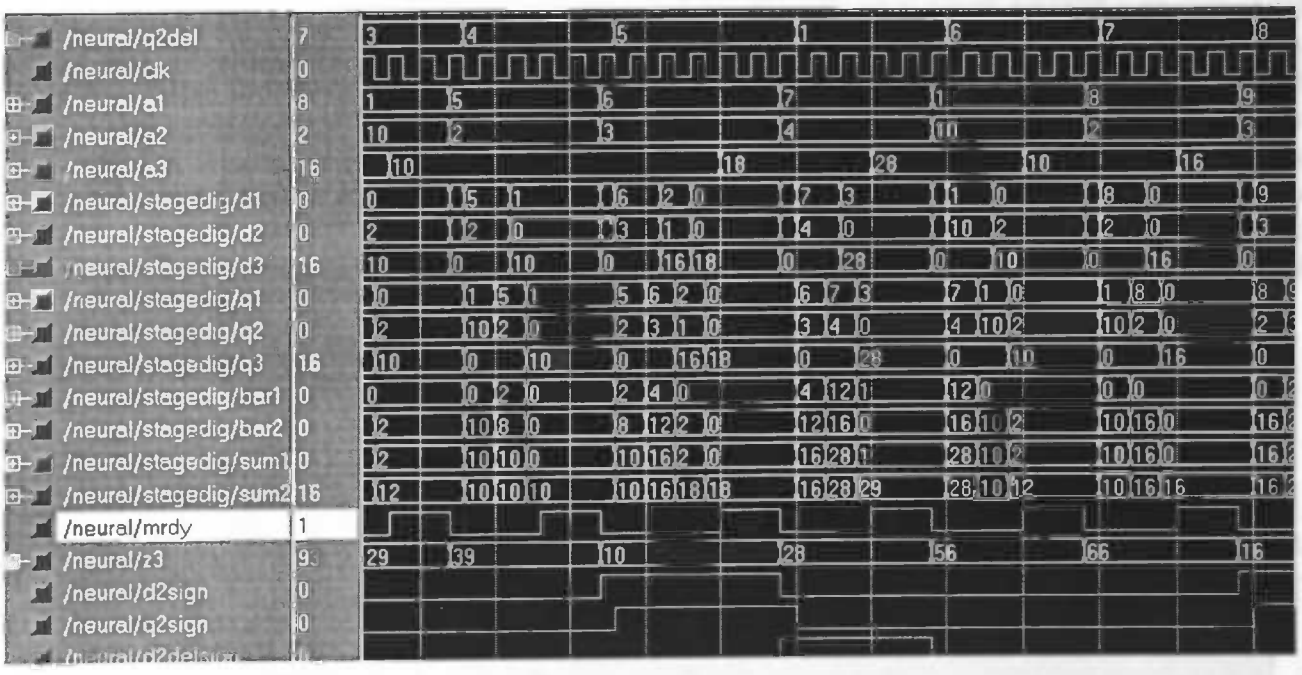

Figure 27 Simulated timing diagram for a neuron

Chapter 5 Neural Network on the Virtex II FPGA

In this chapter we describe how we have implemented a modular neural network on the Virtex II FPGA using its RAM and multiplier macro's.

5.1 FPGA – Virtex II

The Xilinx FPGA's we use have memory called Block SelectRAM. It is Static RAM (SRAM), built using flip-flops, so it doesn't need to be refreshed like Dynamic RAM (DRAM). The SelectRAM is written to on the rising clock-edge. It is also write-through, meaning the input just written appears on the output almost immediately.

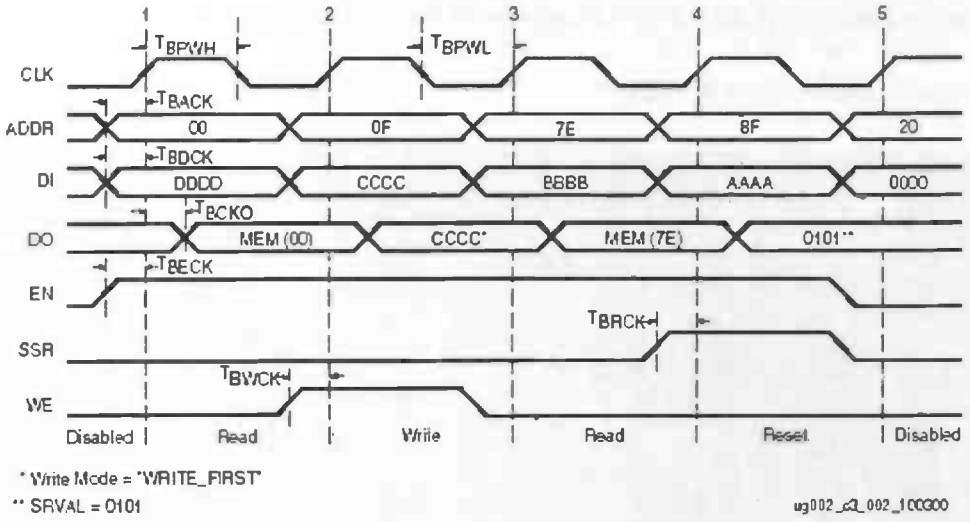

Figure 28 Block SelectRAM timing diagram

Figure 28 has the following signals: CLK is the clock, ADDR is the address that is written to and which is visible on the output pins, DI is the input data, DO is the output data, EN is enable (with enable set to zero the memory only has the last output on its pins and can't be written to), WE is write enable.
As you can see it takes at most one clock cycle to output the contents of a memory address. So we can safely say the memory is very fast and won't cause any timing problems in our pipeline. However, the lines connecting the memory with the computation logic can cause a delay caused by the distance the signal has to travel across the FPGA. We'll have to build the pipeline in order to simulate the time it takes to deliver the data from the memory.

The Virtex II has 4-32 fixed multipliers on board (depending on the board model), which are very fast. Faster multipliers are possible using the CLB's (core logic blocks), but only for multiplication by a constant; so this is only useful for filter design and less for our purpose. The multiplier blocks are actually 18 by 18 bits, but we use only part of it for multiplying two 8-bit numbers. Multiplication of two 8-bit numbers results in a 16-bit product.

For the summation following the multiplication there are no fixed units available on the Virtex II series. This means it has to be constructed using CLB's. This, however, doesn't make a difference

for the VHDL code, because the tool recognizes VHDL multiplier and memory implementations and maps them to the dedicated hardware automatically.

5.2 Implementation

To make an effective pipeline we have to analyze the timing specifics of the individual stages. We have to take a look how long it takes to retrieve information from the memory, to multiply the numbers, to add two numbers and to store the result in memory. Below, we provide the various timing diagrams. These timings do not specify the delay from the input of the specific component to the output of the component. In order to generate a timing diagram of the components as they would have been mapped on the chip, we had to connect the inputs and outputs to pins, as these were the only signals we could view in the Xilinx Model Simulator (probably due to a starters license). As a result, the timings specify the delay between the inputs of the component at the time of the rising clock edge, to output pins associated with the outputs of the component.

5.2.1 Memory timing (post place & route)

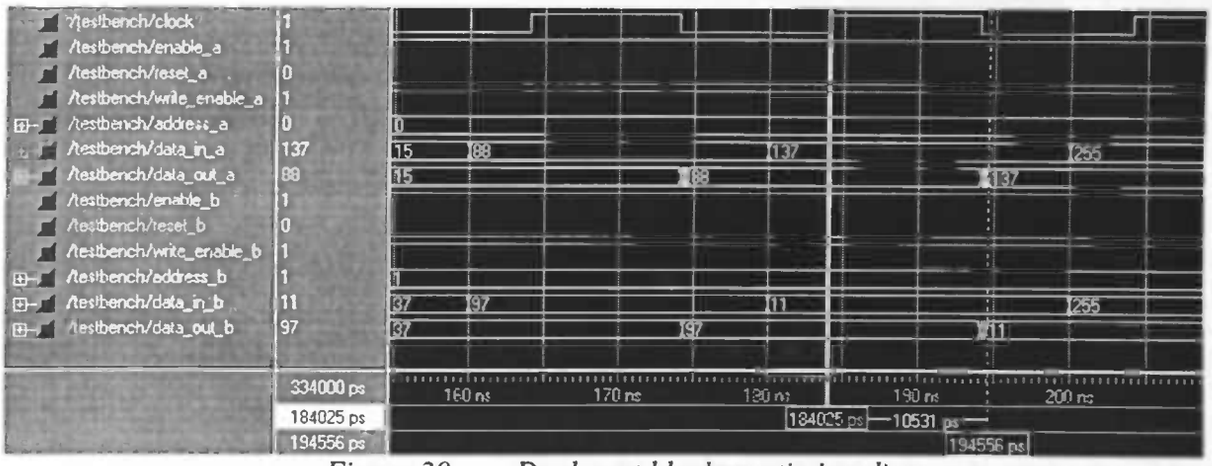

Figure 29 Dual-port blockram timing diagram

As we can see in Figure 29 the dual-port blockram is write-through: the value written to a certain memory location appears on the output just as fast as it would when requesting data stored at that location. The delay to an output pin is approximately 10 to 11ns. Note that when using dual-port blockram, two addresses can be accessed simultaneously. These addresses cannot be the same however.

5.2.2 Multiplier timing (post place & route)

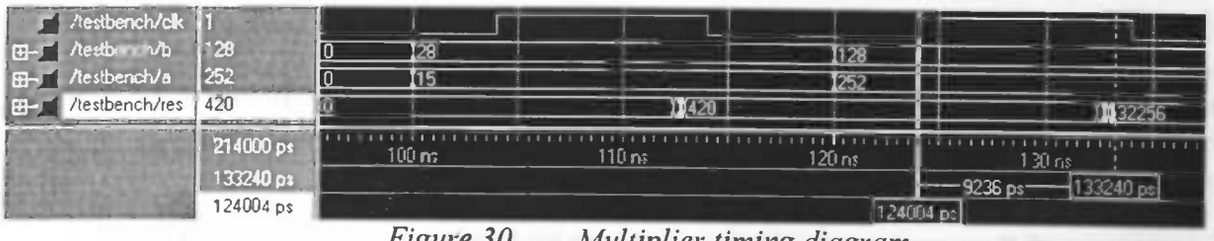

Figure 30 Multiplier timing diagram

As we can see in Figure 30 the rising-edge triggered multiplier has a delay off approximately 9 nanoseconds.

5.2.3 Adder timing (post place & route)

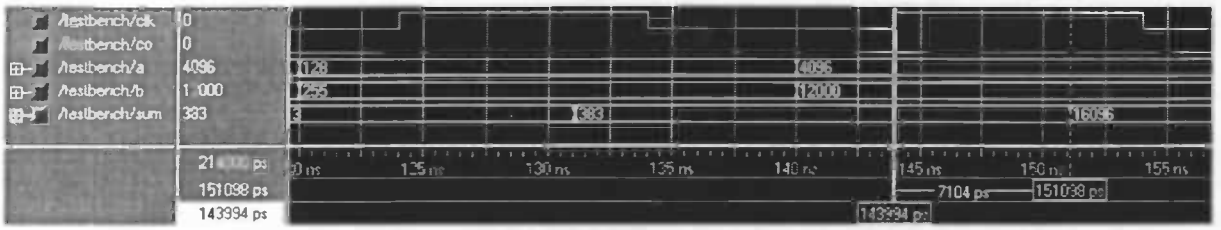

Figure 31 Adder timing diagram

Similarly, from Figure 31 we see that the adder is a little bit faster then the memory and multiplier: it has a delay of approximately 7 ns. As all components have a delay of the same order, it is not necessary to have a pipeline containing multiple units of one type. We can adjust the clock speed so that each action fits inside one clock-cycle. This way, the clock functions as a trigger for the next stage of the pipeline and there is no need for additional registers or memory access for storing intermediate values.

5.2.4 The multiplying adder (post place & route)

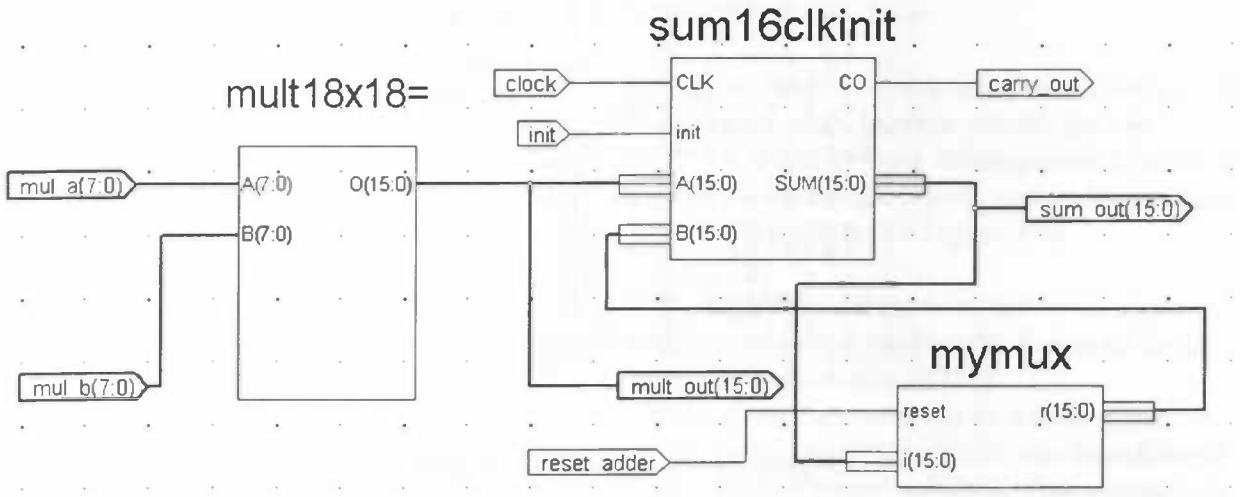

Figure 32 Multiplying adder

Figure 32 gives the schematic of the multiplying adder. The multiplier gets two 8 bit inputs and produces an 16 bit output. This output goes to the adder. The output of the adder is fed into the multiplexer mymux. Via this multiplexer the output of the adder goes back to the input of the adder, except when reset_adder is set to '1'.

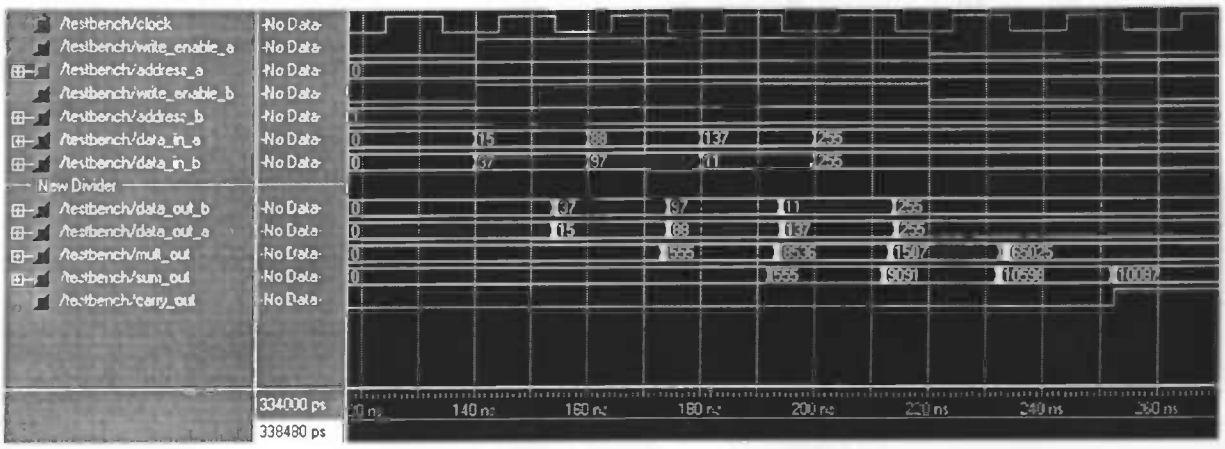

Figure 33 Pipeline timing diagram

Figure 33 visualizes the several pipeline stages: the multiplier output (mult_out) is the multiplication of its inputs (data_out_a and data_out_b), the adder output is the sum of its input (mult_out) and the previous output (sum_out). When combining a RAM macro and a multiplier macro together with some CLB's we can construct a neural network module that nicely fits into a square part of the Virtex II chip (Figure 34).

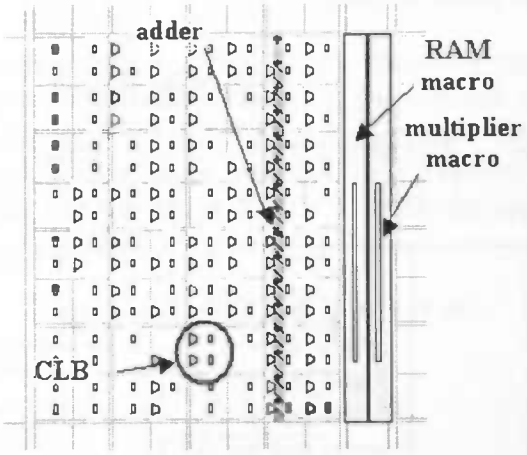

Figure 34 Module floor plan

5.3 Modular Neural Network

Modular neural networks or more commonly called multi-nets are combinations of several neural networks [12]. They are of growing interest, as the implied feature redundancy is believed to make the overall net more accurate than the parts. Moreover, multi-nets can be easier to understand and to modify.

The monolithic neural network is a combination of neurons with a characteristic transfer function. This function can be explicitly imposed or locally created from a small sub-network. For instance, a sub-network of neurons with linear transfer can behave as a single neuron with a sigmoid transfer. In this sense, the monolithic neural network is already a modular one in disguise and one may

therefore expect that the learning problems will be the same. From the observation that, with growing problem size, the monolithic network has increasing difficulty to learn with sufficient quality [13], one may expect not better from a multi-net.

In both cases, the learning process suffers from the entropy in the example set. This can only be resolved (a) by data preprocessing, (b) by inclusion of pre-knowledge or (c) by domain structuring. Such can be achieved by using modular networks [14]. The claim that more than 80% of the development time for monolithic networks is spent on the data preprocessing underlines this observation [15].

Hierarchical networks go one step further in compositional sense. Each node in a neural network may be again a neural network [16], or a specialized function. This means that a neural network implements the evaluation function of the node, thereby preserving the weights on its inputs from the upper layer. An example is the fuzzyfication of singleton input variables, who would otherwise cause a classification problem.

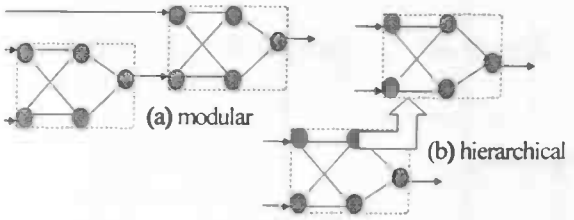

Figure 35 Some neural network types.

A modular network can be interpreted as a multi-layer hierarchical network where ultimately on the highest level the weights are constant and equal to one (Figure 35a). In other words, the top modular composition has lost its exclusive neural outlook and has become heterogeneous in nature by allowing for components of any fabric. By the expansion to multiple layers, and adding weights on the connections between networks, hierarchy is enabled as depicted in Figure 35b.

What is similar to both types of networks is that both hierarchical and modular networks apply functional specialization, although in a different form. Specialization enables the fusion of existing knowledge into the neural network, as was shown for modular networks in [17].
A typical example is shown in Figure 36. The knowledge about the operation of a float-glass furnace was collected as a set of small rule blocks. Subsequently each rule block was transformed into a neural module and then the modules were combined into a neural network. This example is later used for a spatial implementation.

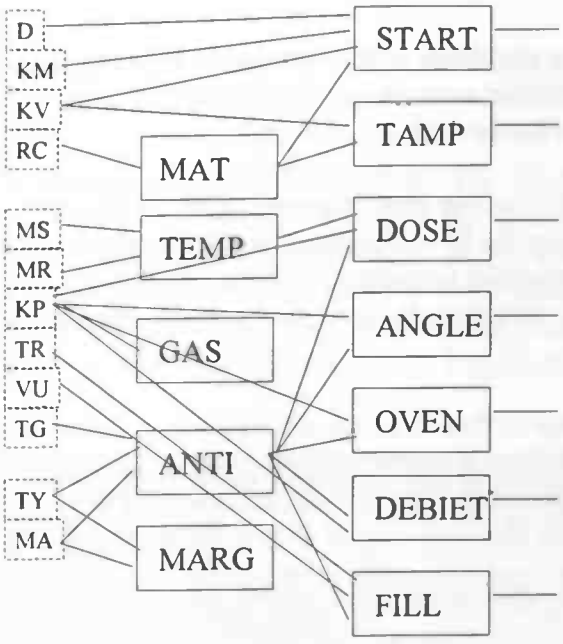

Figure 36 Rule set for a float-glass furnace.

5.4 Spatial neural computing

The temporal design of a module contains already all the necessary ingredients for neural data processing. Scaling can easily be achieved by increasing the network representation within the SRAM, but this will soon lead to unwielding long execution times. The alternative is the replication of the elementary module over the chip. In the past this was no option but with the coming of ground-breaking FPGA devices like the Xilinx Virtex-II family the option becomes very real.

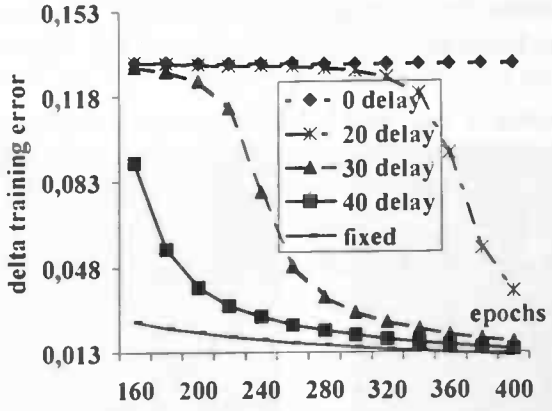

Figure 37 Impact of delayed module activation.

The added advantage of spatial computing is the opportunity to learn the modules of a neural network almost in parallel. This was first noted in [18] in the analysis of the unlearning potential of neural composition. When a network is assembled from trained and empty modules, it frequently happens that the inserted knowledge is swept away during the first epochs and the network continues as if nothing had been there.
The remedy has proven to be a time-ordering of the activation time of the individual modules. A simple example is given in Figure 37. The original circuit was next to impossible to learn, but

already small delays between the activation of the modules brings learning time back to acceptable properties, wherein the overall network is trained in just slightly more time than a single module. As all inputs and outputs of the modules are handled by using the SRAM, passing information between modules views the SRAM as a blackboard. It is not necessary to signal new events by semaphores as a neural network is robust enough to allow for the occasional mixture of old and new values [19].

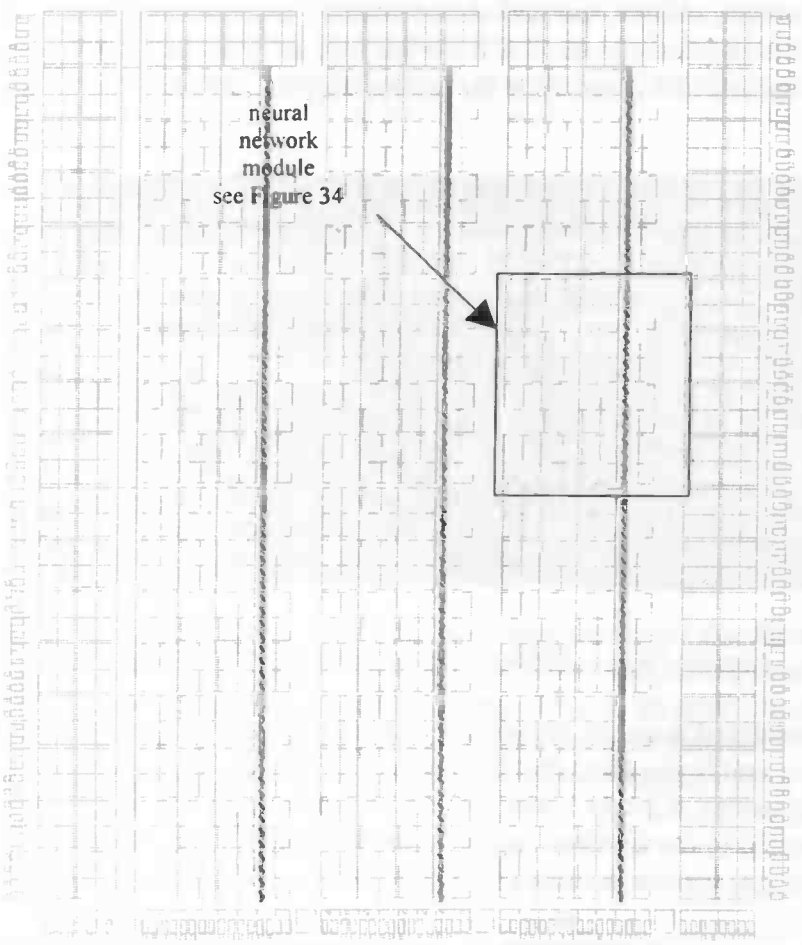

Figure 38 Floor plan of a spatial neural network

Such considerations make for a compact arrangement by mere concatenation of the modules. Figure 38 shows the floor plan of such a spatially unrolled neural network, executing the knowledge of Figure 36 using 12 modules. The indicated module is the one given in Figure 34. This mudule occupies 83 % of the available flip flops and 57 % of the available LUT's.
The only real variable is the RAM usage, which in turn relates to the network size. This leads to using the average speed per line of RAM code (loc) as Figure of Merit. For our design this leads to 40 ns/loc. For a temporal design this number would be a constant, but for the spatial design the number of modules in the longest path divides the number. This leaves of course the latency of the design. When compared to a temporal software design on a Pentium-III, the acceleration is by a factor 20, as earlier reported in [2].

5.5 RAM usage and results

Our neural network module uses one dual-port RAM block. The one port reads and writes words of 8 bits, the other port reads and writes words of 16 bits. The first one is used to read and write neuron values. The second one is used to read a bitstring of 16 bits containing a lot of information. Namely, the synapse weight, offsetn, last and next address. Next address points to the address of the next synapse. We put the addresses in the RAM because we have a lot of RAM compared to logic in CLB's, so now we don't need a counter for the addresses. Figure shows the timing diagram of a network.

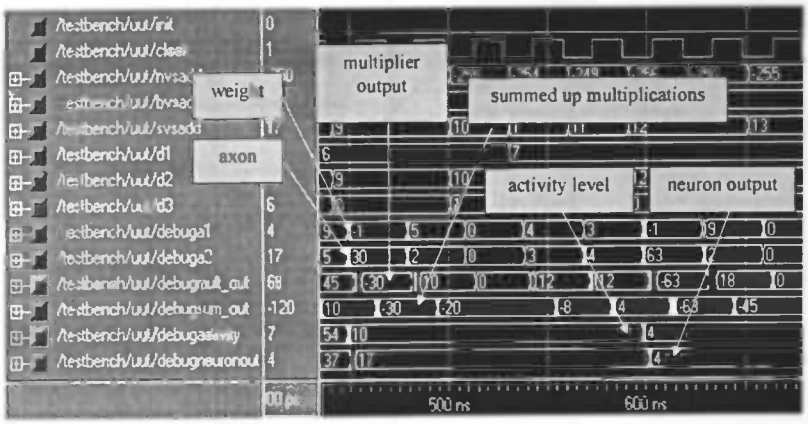

Figure 39 Timing diagram of a neural network module

In Figure we see the results of a neural network consisting of two modules. In the left part we see the two modules (upper and lower part) that do the same. The bias is 11 and is multiplied by -1. There are three weight values, 4,2 and 1, and three axon values, 2, 3 and 4. The two outputs of the first module are connected to the first two inputs of the second module. In the right part (something later during the simulation) we can see that the output values of the first module (both 4) are used in the second module.

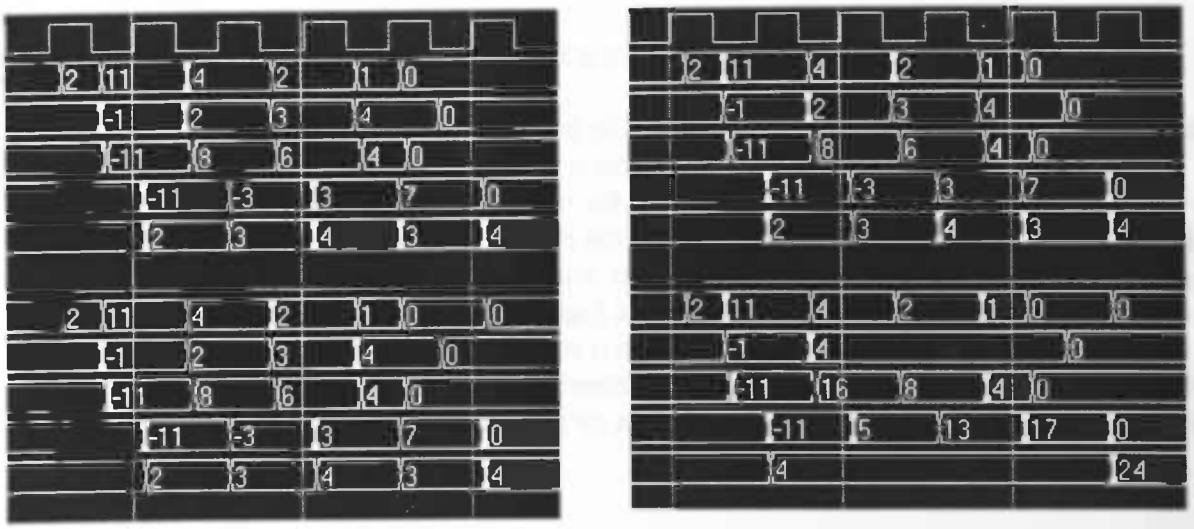

Figure 40 Timing diagram of two modules

Chapter 6 Conclusions

In the past the digital implementation of neural networks didn't get a lot of attention. The synapses seemed too numerous to be physically shaped and therefore more easily handled in software. Further, their operation requires a multiplication, which is electrically easy but logically cumbersome.

Micro-electronic technology has changed. In stead of scheduling multiple tasks to be performed on limited resources, more and more resources became available and the implementation approach changed from a temporal style to a spatial style. Recent FPGA (Field Programmable Gate Array) devices are well suited for a spatial design in which many computational nodes are running in parallel. We have studied these devices to see which possibilities they offer for the implementation of a digital neural network.

The most important and largest component of a neural network is the multiplier. We compared a number of multipliers and showed which multiplier you can use best depending on your speed and area requirements. The different implementations nicely show the dependency between speed and area.

The latest families of FPGA devices turned out to be quite appropriate for the design of modular neural networks. Using the RAM and multiplier macro's we succesfully constructed a neural network module of which a number can be placed on a single chip in an interconnected way. Reconfiguration of the FPGA can be used to change the neural network modules while the other modules are running. Reconfiguration allows the network to grow over the size of the FPGA.

Chapter 7 References

[1] S. Haykin, "Neural Networks – A Comprehensive Foundation", Prentice Hall International Inc., 1994.

[2] A. de Hon, "Reconfigurable Architectures for General-Purpose Computing, AI Techn. Rpt 1586 (MIT, Cambridge) 1996.

[3] Xilinx website, http://www.xilinx.com

[4] Xess website, http://www.xess.com

[5] Aldec website, http://www.aldec.com

[6] E.M. Sentovich et al, "SIS: A System for Sequential Circuit Analysis," Tech. Report No. UCB/ERL M92/41, University of California, Berkeley, 1992.

[7] V. Betz, "VPR and T-VPack User's Manual (Version 4.30)," March 27, 2000. (Available for download from http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html).

[8] Vaughn Betz and Jonathan Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", International Workshop on Field Programmable Logic and Applications, 1997, pp.213-222.

[9] R. van Drunen, L. Spaanenburg, P. Lucassen, J.A.G. Nijhuis and J.T. Udding, "Arithmetic for relative Accuracy", Submitted to Computer Arithmetic '95 (Bath, UK)

[10] Abdelkrim Kamel Oudjida , High Speed and Very Compact Two's Complement Serial/Parallel Multipliers Using Xilinx's FPGA, CDTA/Microelectronics Laboratory, ALGERIA

[11] Werner diagrams - Bengt Werner et al. Werner Diagrams—Visual Aid for Design of Synchronous Systems. Technical report, Department of Computer Engineering, Lund University, Sweden, August 1992.

[12] A. Sharkey, "Multi-Net Systems", in "Combining Artificial Neural Nets" Ed. A Sharkey, Springer-Verlag, London, 1999, ISBN 1-85233-004-X.

[13] W.G. Macready, A.G. Siapas, and S.A. Kauffman, "Criticality and parallelism in combinatorial optimization", Science, Vol. 271, pp. 56-59 (1996).

[14] T. Caelli, L.Guan and W.Wan, "Modularity in Neural Computing", Proceedings of the IEEE 87, No.9 (September 1999), pp. 1497-1518.

[15] B. Schuermann, "Applications and Perspectives of Artificial Neural Networks", VDI Berichte, Vol. 1526, pp. 1-14 (2000).

[16] A.J.W.M. ten Berg and L. Spaanenburg, "Considerations of the Compositionality of Neural Networks", Proceedings ECCTD, vol. III (Helsinki, September 2001) pp. 405-408.

[17] L. Spaanenburg, "Over multiple rule-blocks to modular nets", Proceedings 23rd Euromicro (Budapest, 1997) pp. 698 – 705.

[18] R.S. Venema and L. Spaanenburg, "Learning feed-forward multi-nets", ICANNGA'01 (Prague, 2001) pp. 102 - 105.

[19] J.A.G. Nijhuis et al., "Delay-insensitive learning in a feed-forward neural network", Proceedings INNC'90, Paris (France) July 1990.

[20] M. Diepenhorst et al. (1996) Using the GREMLIN for digital FIR networks, Proceedings MicroNeuro'96 (Lausanne) pp. 341 - 346.

Acknowledgement

I would like to thank everybody who has contributed to this thesis in one way or another. Special thanks go to my supervisor Prof.dr.ir. L. Spaanenburg. Also I would like to thank my family for food and support.

Appendix A Components

A.1 Series-parallel multiplier

sp8.vhd

```
--------------------------------------------------------------------------------
--  8 bit series-parallel multiplier
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity sp8 is
      port (      clk         : in  bit;
                  reset       : in  bit;
                  x           : in  bit_vector(7 downto 0);
                  y           : in  bit_vector(7 downto 0);
                  p           : out bit_vector(15 downto 0);
                  ready       : out bit );
end sp8 ;

architecture structure of sp8 is
      component spblock is
            port (      clk         : in  bit;
                        reset       : in  bit;
                        x           : in  bit;
                        y           : in  bit;
                        b           : in  bit;
                        p           : out bit );
      end component;
      component dff8 is
            port (      clk         : in bit;
                        data        : in bit_vector(7 downto 0);
                        q           : out bit_vector(7 downto 0) );
      end component;
      component dff16 is
            port (      clk         : in bit;
                        data        : in bit_vector(15 downto 0);
                        q           : out bit_vector(15 downto 0) );
      end component;
      signal p1,p2,p3,p4,p5,p6,p7,p8,nul,first: bit;
      signal dr,qr: bit_vector(15 downto 0);
      signal dcount,qcount: bit_vector(15 downto 0);
      signal dy,qy,dx,qx: bit_vector(7 downto 0);
begin
      stage1: spblock port map (clk, reset, qx(7), qy(0), nul, p1);
      stage2: spblock port map (clk, reset, qx(6), qy(0), p1, p2);
      stage3: spblock port map (clk, reset, qx(5), qy(0), p2, p3);
      stage4: spblock port map (clk, reset, qx(4), qy(0), p3, p4);
      stage5: spblock port map (clk, reset, qx(3), qy(0), p4, p5);
      stage6: spblock port map (clk, reset, qx(2), qy(0), p5, p6);
      stage7: spblock port map (clk, reset, qx(1), qy(0), p6, p7);
      stage8: spblock port map (clk, reset, qx(0), qy(0), p7, p8);
      stage9:  dff16  port map (clk, dcount, qcount);
      stage10: dff16  port map (clk, dr, qr);
      stage11: dff8   port map (clk, dy, qy);
      stage12: dff8   port map (clk, dx, qx);

      rekenen: process(clk,reset)
```

```
      begin
            if reset='1' then
                  dcount<="0000000000100000";
                  dy<="00000000";
                  dx<="00000000";
            elsif clk'event and clk='1' then
                  if qcount(4) = '1' then
                        dy<=y;
                        dx<=x;
                  else
                        dy(7)<='0';
                        dy(6 downto 0)<=qy(7 downto 1);
                  end if;
                  dr(14 downto 0) <= qr(15 downto 1);
                  if qcount(1) = '1' then
                        p<=qr;
                  end if;
                  if qcount(0) = '1' then
                        dcount<="1000000000000000";
                  else
                        dcount(15)<='0';
                        dcount(14 downto 0)<=qcount(15 downto 1);
                  end if;
                  dr(15)<=p8;
            end if;
      end process;
end structure;
```

spblock.vhd

```
-------------------------------------------------------------------------------
--  block of series-parallel multiplier
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity spblock is
      port (      clk         : in  bit;
                  reset       : in  bit;
                  x           : in  bit;
                  y           : in  bit;
                  b           : in  bit;
                  p           : out bit );
end spblock ;

architecture structure of spblock is
      component fulladd
            port (      Cin, x, y   : in  bit ;
                        s, Cout     : out bit ) ;
      end component ;
      component dff is
            port (      clk         : in bit;
                        data        : in bit;
                        q           : out bit );
      end component;
      signal d1,d2,q1,q2,a,sum,carry: bit;
begin
      stage1:  dff      port map ( clk, d1 , q1 );
      stage2:  dff      port map ( clk, d2 , q2 );
```

```
      stage3:  fulladd port map ( q1, a, b, d2, d1 );

      rekenen: process(clk)

      begin
            a<=x and y;
            p<=q2;
      end process;
end structure;
```

dff16.vhd

```
-------------------------------------------------------------------------------
--  16 bit edge triggered flip flop
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity dff16 is
      port (      clk         : in  bit;
                  data        : in  bit_vector(15 downto 0);
                  q           : out bit_vector(15 downto 0) );
end dff16;

architecture rtl of dff16 is
begin
      infer : process (clk)

      begin
            if (clk'event and clk = '0') then
                  q <= data;
            end if;
      end process infer;
end rtl;
```

fulladd.vhd

```
-------------------------------------------------------------------------------
--  1 bit fulladder with carry in and carry out
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity fulladd is
port (      Cin, x, y : in  bit;
            s, Cout   : out bit );
end fulladd;

architecture fulladd_arch of fulladd is
begin
      s <= x xor y xor Cin;
      Cout <= (x and y) or (Cin and x) or (Cin and y);
end fulladd_arch;
```

A.2 Digilog multiplier

digi.vhd

```
-------------------------------------------------------------------------------
--  not pipelined version of the Digilog multiplier
-------------------------------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;

entity digi is
      port (      clk   : in  bit;
                  reset : in  bit;
                  mulgo : in  bit;
                  op1   : in  bit_vector (7 downto 0) ;
                  op2   : in  bit_vector (7 downto 0) ;
                  s_in  : in  bit;
                  s_out : out bit;
                  mul   : out bit_vector (15 downto 0) ;
                  ready : out bit );
end digi ;

architecture structure of digi is
      signal d1: bit_vector(15 downto 0);
      signal d2: bit_vector(15 downto 0);
      signal d3: bit_vector(15 downto 0);
      signal q1,q2,q3, bar1, bar2, sum1, sum2: bit_vector(15 downto 0);
      signal ds, qs, nzq1, nzq2, ddel, qdel :bit;
      signal msbq1, msbq2 : bit_vector(3 downto 0);
      signal slbq1, slbq2 : bit_vector(15 downto 0);
      component adder16 is
            port (      x           : in  bit_vector (15 downto 0) ;
                        y           : in  bit_vector (15 downto 0) ;
                        s           : out bit_vector (15 downto 0) );
      end component ;
      component slbmsbnz is
            port (      i           : in  bit_vector(15 downto 0);
                        slb         : out bit_vector(15 downto 0);
                        msb         : out bit_vector(3 downto 0);
                        notzero     : out bit );
      end component;
      component barrel is
            port (      i           : in  bit_vector(15 downto 0);
                        b           : in  bit_vector(3 downto 0);
                        r           : out bit_vector(15 downto 0) );
      end component ;
      component dff16 is
            port (      clk         : in bit;
                        data        : in bit_vector(15 downto 0);
                        q           : out bit_vector(15 downto 0) );
      end component;
      component dff is
            port (      clk         : in bit;
                        data        : in bit;
                        q           : out bit );
      end component;

begin
      stage1:  dff16    port map ( clk, d1 , q1 );
      stage2:  dff16    port map ( clk, d2 , q2 );
      stage3:  dff16    port map ( clk, d3 , q3 );
      stage4:  dff      port map ( clk, ds , qs );
      stage4d: dff      port map ( clk, ddel , qdel );
      stage5:  slbmsbnz port map ( q1, slbq1, msbq1, nzq1 );
      stage6:  slbmsbnz port map ( q2, slbq2, msbq2, nzq2 );
      stage11: barrel   port map ( slbq1, msbq2, bar1 );
      stage12: barrel   port map ( q2,    msbq1, bar2 );
```

```
      stage13: adder16  port map ( bar1, bar2, sum1 );
      stage14: adder16  port map ( q3,   sum1, sum2 );

      rekenen: process(clk)

      begin
            if reset='1' then
                  ready<='0';
                  d1<="00000000"&op1;
                  d2<="00000000"&op2;
                  d3<="0000000000000000";
                  ds<=s_in;
                  ddel<='0';
            elsif clk'event and clk='1' then
                  if qdel='0' then
                        ddel<='1';
                  else
                        if nzq1='1' and nzq2='1' then
                              d3<=sum2;
                              d1<=slbq1;
                              d2<=slbq2;
                        else
                              ready<='1';
                              mul<=q3;
                              s_out<=qs;
                        end if;
                  end if;
            end if;
      end process rekenen;
end structure ;


slbmsbnz.vhd
-------------------------------------------------------------------------------
--  slb strips the leading bit of i
--  msb is a binary coding for the most significant bit of i
--  notzero is '1' when i is not zero
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
                        -- msb won't be used when i="0000000000000000"
                        -- (msb 000 = 0 and msb 001 = 0)
entity slbmsbnz is
port (      i           : in  bit_vector(15 downto 0);
                  slb         : out bit_vector(15 downto 0);
                  msb         : out bit_vector(3 downto 0);
                  notzero     : out bit );
end slbmsbnz;

architecture structure of slbmsbnz is
signal j : bit_vector(15 downto 0);       --    0001011 becomes 0001111

begin
      j(15) <= i(15);                     -- or '0';
      j(14) <= i(14) or j(15);
      j(13) <= i(13) or j(14);
      j(12) <= i(12) or j(13);
      j(11) <= i(11) or j(12);
      j(10) <= i(10) or j(11);
      j(9)  <= i(9)  or j(10);
```

```
      j(8)  <= i(8)  or j(9);
      j(7)  <= i(7)  or j(8);
      j(6)  <= i(6)  or j(7);
      j(5)  <= i(5)  or j(6);
      j(4)  <= i(4)  or j(5);
      j(3)  <= i(3)  or j(4);
      j(2)  <= i(2)  or j(3);
      j(1)  <= i(1)  or j(2);
      j(0)  <= i(0)  or j(1);
      notzero <= j(0);
      slb(15) <= i(15) and not j(15);
      slb(14) <= i(14) and (not j(14) or j(15));
      slb(13) <= i(13) and (not j(13) or j(14));
      slb(12) <= i(12) and (not j(12) or j(13));
      slb(11) <= i(11) and (not j(11) or j(12));
      slb(10) <= i(10) and (not j(10) or j(11));
      slb(9)  <= i(9)  and (not j(9)  or j(10));
      slb(8)  <= i(8)  and (not j(8)  or j(9));
      slb(7)  <= i(7)  and (not j(7)  or j(8));
      slb(6)  <= i(6)  and (not j(6)  or j(7));
      slb(5)  <= i(5)  and (not j(5)  or j(6));
      slb(4)  <= i(4)  and (not j(4)  or j(5));
      slb(3)  <= i(3)  and (not j(3)  or j(4));
      slb(2)  <= i(2)  and (not j(2)  or j(3));
      slb(1)  <= i(1)  and (not j(1)  or j(2));
      slb(0)  <= i(0)  and (not j(0)  or j(1));
   msb(3) <= j(8);
   msb(2) <= (j(4) xor j(8))  or j(12);
   msb(1) <= (j(2) xor j(4))  or (j(6) xor j(8))  or (j(10) xor j(12)) or j(14);
   msb(0) <= (j(1) xor j(2))  or (j(3) xor j(4))  or (j(5)  xor j(6))  or
             (j(7) xor j(8))  or (j(9) xor j(10)) or (j(11) xor j(12)) or
            (j(13) xor j(14)) or j(15);
end structure;


barrel.vhd
------------------------------------------------------------------------------
--  16 bit barrel shifter
--  shifts 1, 2, 4 and/or 8 places in 4 steps
--  depending on the 4 bits of b
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity barrel is
port (      i: in  bit_vector(15 downto 0);
                b: in  bit_vector(3 downto 0);
                r: out bit_vector(15 downto 0) );
end barrel;

architecture structure of barrel is
     signal buffer_a, buffer_b, buffer_c : bit_vector(15 downto 0);
begin

buffer_a(15 downto 0) <=
i(14 downto 0) & '0' when b(0)='1'
else i(15 downto 0);

buffer_b(15 downto 0) <=
```

```
buffer_a(13 downto 0) & "00" when b(1)='1'
else buffer_a(15 downto 0);

buffer_c(15 downto 0) <=
      buffer_b(11 downto 0) & "0000" when b(2)='1'
else buffer_b(15 downto 0);

r(15 downto 0) <=
buffer_c(7 downto 0) & "00000000" when b(3)='1'
else buffer_c(15 downto 0);

end structure;
```

adder16.vhd

```
--------------------------------------------------------------------------------
--  16 bit fulladder (without carry in and carry out)
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity adder16 is
port (      x     : in  bit_vector (15 downto 0);
                  y     : in  bit_vector (15 downto 0);
                  s     : out bit_vector (15 downto 0) );
end adder16;

architecture structure of adder16 is
      signal c1, c2, c3, c4, c5, c6, c7, c8,
             c9, c10, c11, c12, c13, c14, c15, c16 : bit;
      component fulladd
            port (      Cin, x, y   : in  bit;
                        s, Cout     : out bit );
      end component;
begin
      stage0:  fulladd port map ('0', x(0), y(0), s(0), c1 );
      stage1:  fulladd port map ( c1, x(1), y(1), s(1), c2 );
      stage2:  fulladd port map ( c2, x(2), y(2), s(2), c3 );
      stage3:  fulladd port map ( c3, x(3), y(3), s(3), c4 );
      stage4:  fulladd port map ( c4, x(4), y(4), s(4), c5 );
      stage5:  fulladd port map ( c5, x(5), y(5), s(5), c6 );
      stage6:  fulladd port map ( c6, x(6), y(6), s(6), c7 );
      stage7:  fulladd port map ( c7, x(7), y(7), s(7), c8 );
      stage8:  fulladd port map ( c8, x(8), y(8), s(8), c9 );
      stage9:  fulladd port map ( c9, x(9), y(9), s(9), c10 );
      stage10: fulladd port map ( c10, x(10), y(10), s(10), c11 );
      stage11: fulladd port map ( c11, x(11), y(11), s(11), c12 );
      stage12: fulladd port map ( c12, x(12), y(12), s(12), c13 );
      stage13: fulladd port map ( c13, x(13), y(13), s(13), c14 );
      stage14: fulladd port map ( c14, x(14), y(14), s(14), c15 );
      stage15: fulladd port map ( c15, x(15), y(15), s(15), c16 );
end structure;
```

A.3 Booth multiplier

booth.vhd

```
--------------------------------------------------------------------------------
--  Booth multiplier
--------------------------------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;

entity booth is
      port (      clk         : in  bit;
                  x           : in  bit_vector(3 downto 0);
                  y           : in  bit_vector(1 downto 0);
                  p           : out bit );
end booth ;

architecture structure of booth is
      component encode2 is
            port (      i3,i2,i1,i0 : in  bit;
                        r1,r0       : out bit );
      end component;
      component fulladd
            port (      Cin, x, y   : in  bit ;
                        s, Cout     : out bit ) ;
      end component ;
      component dff is
            port (      clk         : in bit;
                        data        : in bit;
                        q           : out bit );
      end component;
      component boothblockfirst is
      port (      clk         : in  bit;
                        x           : in  bit_vector(1 downto 0);
                        y           : in  bit_vector(1 downto 0);
                        q7          : out bit );
      end component;
      component boothblock is
            port (      clk         : in  bit;
                        x           : in  bit_vector(1 downto 0);
                        y           : in  bit_vector(1 downto 0);
                        a1          : in  bit;
                        q7          : out bit );
      end component;
      signal a2,a4,sum5,cout5:bit;
      signal q13,q14,q15,q16:bit;

begin
stage1: boothblockfirst port map (clk, x(3 downto 2),
y(1 downto 0), a2);
stage2: boothblock      port map (clk, x(1 downto 0),
y(1 downto 0), a2, a4);
      stagea5: fulladd port map ( q14, a4, q15, sum5, cout5);
      stage13: dff     port map ( clk, sum5 , q13 );
      stage14: dff     port map ( clk, cout5 , q14 );
      stage15: dff     port map ( clk, q16 , q15 );
      stage16: dff     port map ( clk, y(1) , q16 );

      rekenen: process(clk)

      begin
            if clk'event and clk='1' then
                  p<=q13;
            end if;
      end process;
end structure;
```

encode2.vhd

```
--------------------------------------------------------------------------------
--  Booth encoder
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity encode2 is
      port (      i3,i2,i1,i0 : in  bit;
                  r1,r0       : out bit );
end encode2;

architecture structure of encode2 is
      signal cin1,cin2,add1,add2,sum1,add3,add4,sum2,cout1,cout2 : bit ;
      component fulladd
            port (      Cin, x, y   : in  bit ;
                        s, Cout     : out bit ) ;
      end component ;

begin
      stage0: fulladd port map ( cin1, add1, add2, sum1, open ) ;
      stage1: fulladd port map ( cin2, add3, add4, sum2, open ) ;

      add1<=not i3 and i1 and not i0;
      add2<=i3 and not i1 and i0;
      cin1<=i1 and i0;
      add3<=not i2 and i1 and not i0;
      add4<=i2 and not i1 and i0;
      cin2<=i1 and i0;
      r1<=sum1;
      r0<=sum2;
end structure;
```

boothblockfirst.vhd

```
--------------------------------------------------------------------------------
--  leftmost Booth block
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity boothblockfirst is
      port (      clk         : in  bit;
                  x           : in  bit_vector(1 downto 0);
                  y           : in  bit_vector(1 downto 0);
                  q7          : out bit );
end boothblockfirst;

architecture structure of boothblockfirst is
      component encode2 is
            port (      i3,i2,i1,i0 : in  bit;
                        r1,r0       : out bit );
      end component;
      component fulladd
            port (      Cin, x, y   : in  bit ;
                        s, Cout     : out bit ) ;
      end component ;
      component dff is
            port (      clk         : in bit;
```

```
                        data        : in bit;
                        q           : out bit );
        end component;
        signal sum1,sum2,cout1,cout2:bit;
        signal d1,d2:bit;
        signal q1,q2,q5,q6,q8:bit;

begin
        stage0a: encode2 port map ( x(1), x(0), y(1), y(0), d1, d2);
        stagea1: fulladd port map ( q6, q5, q1, sum1, cout1);
        stagea2: fulladd port map ( q8, q5, q2, sum2, cout2);
        stage01: dff     port map ( clk, d1 , q1 );
        stage02: dff     port map ( clk, d2 , q2 );
        stage05: dff     port map ( clk, sum1 , q5 );
        stage06: dff     port map ( clk, cout1 , q6 );
        stage07: dff     port map ( clk, sum2 , q7 );
        stage08: dff     port map ( clk, cout2 , q8 );
end structure;
```

boothblock.vhd

```
--------------------------------------------------------------------------------
--  other Booth block, can be repeated for larger widths
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity boothblock is
      port (      clk         : in  bit;
                  x           : in  bit_vector(1 downto 0);
                  y           : in  bit_vector(1 downto 0);
                  a1          : in  bit;
                  q7          : out bit );
end boothblock;

architecture structure of boothblock is
      component encode2 is
            port (      i3,i2,i1,i0       : in  bit;
                        r1,r0             : out bit );
      end component;
      component fulladd
            port (      Cin, x, y   : in  bit ;
                        s, Cout     : out bit ) ;
      end component ;
      component dff is
            port (      clk         : in bit;
                        data        : in bit;
                        q           : out bit );
      end component;
      signal sum1,sum2,cout1,cout2:bit;
      signal d1,d2:bit;
      signal q1,q2,q5,q6,q8:bit;

begin
      stage0a: encode2 port map ( x(1), x(0), y(1), y(0), d1, d2);
      stagea1: fulladd port map ( q6, a1, q1, sum1, cout1);
      stagea2: fulladd port map ( q8, q5, q2, sum2, cout2);
      stage01: dff     port map ( clk, d1 , q1 );
      stage02: dff     port map ( clk, d2 , q2 );
      stage05: dff     port map ( clk, sum1 , q5 );
```

```
      stage06: dff     port map ( clk, cout1 , q6 );
      stage07: dff     port map ( clk, sum2 , q7 );
      stage08: dff     port map ( clk, cout2 , q8 );
end structure;
```

A.4 Pipelined Digilog multiplier

```
digi3.vhd
-------------------------------------------------------------------------------
--  pipelined version of the Digilog multiplier
--  see A.2 for the components
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity digi3 is
port (      clk   : in  bit;
                reset : in  bit;
                mulgo : in  bit;
                op1   : in  bit_vector (7 downto 0) ;
                op2   : in  bit_vector (7 downto 0) ;
                s_in  : in  bit;
                s_out : out bit;
                mul   : out bit_vector (15 downto 0) ;
                ready : out bit );
end digi3 ;

architecture structure of digi3 is
     signal dA,dB,dT,dR,qA,qB,qT,qR,split,slb,barrelin,add1,add2,sum
: bit_vector(15 downto 0);
     signal di,qi,msb,barrelnr          : bit_vector(3 downto 0);
signal nz,dtik,qtik,firsttime,dnzoldA,dnzoldB, dnzoldoldA,
    qnzoldA,qnzoldB,qnzoldoldA : bit;
     component adder16 is
          port (      x          : in  bit_vector (15 downto 0) ;
                      y          : in  bit_vector (15 downto 0) ;
                      s          : out bit_vector (15 downto 0) );
     end component ;
     component slbmsbnz is
          port (      i          : in  bit_vector(15 downto 0);
                      slb        : out bit_vector(15 downto 0);
                      msb        : out bit_vector(3 downto 0);
                      notzero    : out bit );
     end component;
     component barrel is
          port (      i          : in  bit_vector(15 downto 0);
                      b          : in  bit_vector(3 downto 0);
                      r          : out bit_vector(15 downto 0) );
     end component ;
     component dff16 is
          port (      clk        : in bit;
                      data       : in bit_vector(15 downto 0);
                      q          : out bit_vector(15 downto 0) );
     end component;
     component dff4 is
          port (      clk        : in bit;
                      data       : in bit_vector(3 downto 0);
                      q          : out bit_vector(3 downto 0) );
     end component;
```

```
      component dff is
            port (      clk         : in bit;
                        data        : in bit;
                        q           : out bit );
      end component;

begin
      stage1:  dff16    port map ( clk, dA, qA );
      stage2:  dff16    port map ( clk, dB, qB );
      stage3:  dff16    port map ( clk, dT, qT );
      stage4:  dff16    port map ( clk, dR, qR );
      stage5:  dff4     port map ( clk, di, qi );
      stage6:  dff      port map ( clk, dtik, qtik );
      stage6a: dff      port map ( clk, dnzoldA, qnzoldA );
      stage6b: dff      port map ( clk, dnzoldB, qnzoldB );
      stage6c: dff      port map ( clk, dnzoldoldA, qnzoldoldA );
      stage7:  slbmsbnz port map ( split, slb, msb, nz );
      stage8:  barrel   port map ( barrelin, barrelnr, dT );
      stage9:  adder16  port map ( add1, add2, dR );

      rekenen: process(clk)

      begin
            if reset='1' then
                  dA<="00000000"&op1;
                  dB<="00000000"&op2;
                  split<=qA;
                  dtik<='0';
            elsif clk'event and clk='1' then
                  if qtik='0' then dtik<='1'; else     dtik<='0'; end if;
                  if firsttime='1' then
                        if qtik='0' then
                              di<=msb;
                              dA<=slb;
                              dnzoldA<=nz;
                              dnzoldoldA<='1';
                              split<=qB;
                        else
                              barrelin<=qB;
                              barrelnr<=qi;
                              di<=msb;
                              dB<=slb;
                              dnzoldB<=nz;
                              split<=qA;
                        end if;
                  else
                        if qtik='0' then
                              di<=msb;
                              dA<=slb;
                              dnzoldA<=nz;
                              if qnzoldA='1' and qnzoldB='1' then
                                    barrelin<=qA;
                                    barrelnr<=qi;
                                    add1<=qR;
                                    add2<=qT;
                                    split<=qB;
                              else
                                    add1<=qR;
                                    add2<="0000000000000000";
                                    split<="00000000"&op2;
```

```
                        dB<="00000000"&op2;
                    end if;
                else
                    di<=msb;
                    dB<=slb;
                    dnzoldB<=nz;
                    dnzoldoldA<=qnzoldA;
                    if qnzoldoldA='1' and qnzoldB='1' then
                        barrelin<=qB;
                        barrelnr<=qi;
                        add1<=qR;
                        add2<=qT;
                        split<=qA;
                    else
                        barrelin<="0000000000000000";
                        barrelnr<="0000";
                        dnzoldA<='1';
                        dnzoldB<='1';
                        dnzoldoldA<='1';
                        add1<="0000000000000000";
                        add2<="0000000000000000";
                        mul<=qR;
                        split<="00000000"&op1;
                        dA<="00000000"&op1;
                    end if;
                end if;
            end if;
        end if;
    end process rekenen;
end structure;
```

Appendix B Networks

B.1 Neural network with Digilog multiplier

neural.vhd

```
--------------------------------------------------------------------------------
--  neural network implementation with digilog multiplier
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity neural is
      port (      clk   : in bit;
                  ready : out bit );
end neural ;

architecture structure of neural is
component adder8 is
      port (      x           : in  bit_vector (7 downto 0) ;
                  y           : in  bit_vector (7 downto 0) ;
                  s           : out bit_vector (7 downto 0) );
      end component ;
component adder16 is
      port (      x           : in  bit_vector (15 downto 0) ;
                  y           : in  bit_vector (15 downto 0) ;
                  s           : out bit_vector (15 downto 0) );
      end component ;
component digi
      port (      clk,reset,mulgo : in  bit;
                  op1,op2     : in  bit_vector (7 downto 0);
                  s_in        : in  bit;
                  s_out       : out bit;
                  mul         : out bit_vector (15 downto 0);
                  ready       : out bit );
end component ;
component dff8 is
      port (      clk         : in  bit;
                  data        : in  bit_vector(7 downto 0);
                  q           : out bit_vector(7 downto 0) );
end component;
component dff is
      port (      clk         : in  bit;
                  data        : in  bit;
                  q           : out bit );
end component;
component nvs is
      port (      clk         : in  bit;
                  init        : in  bit;
                  i           : in  bit_vector(7 downto 0);
                  address     : in  bit_vector(7 downto 0);
                  write       : in  bit;  -- 0 read, 1 write
                  r           : out bit_vector(7 downto 0) );
end component;
component bvs is
      port (      clk         : in  bit;
                  init        : in  bit;
                  address     : in  bit_vector(7 downto 0);
                  bias              : out bit_vector(7 downto 0);
```

```
                biassign          : out bit;
                nextbias          : out bit_vector(7 downto 0);
                nextbiassign      : out bit;
                offsets           : out bit_vector(7 downto 0) );
end component;
component svs is
      port (    clk               : in  bit;
                init              : in  bit;
                address           : in  bit_vector(7 downto 0);
                synweight         : out bit_vector(7 downto 0);
                synweightsign     : out bit;
                offsetn           : out bit_vector(7 downto 0);
                last              : out bit );
end component;
signal d1,d2,d3,d2del,q1,q2,q3,q2del, a1, a2    : bit_vector(7 downto 0);
signal d2sign, q2sign, d2delsign, q2delsign, s_in, s_out    : bit;
signal a3    : bit_vector (15 downto 0) ;
signal mrdy,dmulreset,qmulreset,mulgo     : bit ;

signal nvsadd, bvsadd, svsadd, svsadd2    : bit_vector(7 downto 0); -- addresses
signal setneuronval, neuronval, offsets, offsetn,
         bias, synweight                  : bit_vector(7 downto 0);
signal write, last, biassign, synweightsign: bit;
signal init, aanuit                       : bit;
signal x1, y1, x2, y2, sum1, sum2         : bit_vector(7 downto 0);
signal x3, y3, z3, sum3                   : bit_vector(15 downto 0);

begin
      nvstore:  nvs port map (clk, init, setneuronval, nvsadd, write,
neuronval);
      bvstore:  bvs port map (clk, init, bvsadd, bias, biassign, open, open,
offsets);
      svstore:  svs port map (clk, init, svsadd, open, open, offsetn, last);
      svstore2: svs port map (clk, init, svsadd2, synweight, synweightsign,
open, open);

      fadder81: adder8  port map (x1, y1, sum1);
      fadder82: adder8  port map (x2, y2, sum2);
      fadder16: adder16 port map (a3, z3, sum3);
      stagedig: digi port map (clk, qmulreset, mulgo, a1, a2, s_in, s_out, a3,
mrdy);
      stage1: dff8 port map (clk, d1 , q1);
      stage2: dff8 port map (clk, d2 , q2);
      stage3: dff8 port map (clk, d3 , q3);
      stage2d:dff8 port map (clk, d2del , q2del);
      stage4: dff  port map (clk, d2sign, q2sign);
      stage2ds:dff port map (clk, d2delsign, q2delsign);
      stagemr :dff port map (clk, dmulreset, qmulreset);
      rekenen: process(clk)
      begin
            if init='1' then
                  d1<="00000011";
                  d2<="00000001";
                  d2sign<='1';
                  d2del<="00000001";
                  d2delsign<='1';
                  d3<="00000011";
                  dmulreset<='1';
                  nvsadd<=q3;
                  svsadd<=q2;
```

```
                svsadd2<=q2del;
                bvsadd<=q1;
          elsif clk'event and clk='1' then
                nvsadd<=q3;
                svsadd<=q2;
                svsadd2<=q2del;
                bvsadd<=q1;
                x1<=q1;
                y1<="00000001";
                x2<=q2;
                y2<="00000001";
                if mrdy='1' then
                      dmulreset<='1';
                end if;

                if qmulreset='1' then
                      if q2delsign='1' then
                            a1<=q2del;
                            a2<=bias;
                            s_in<='1';
                      else
                            a1<=synweight;
                            a2<=neuronval;
                            s_in<=synweightsign;
                      end if;
                            d2del<=q2;
                      d2delsign<=q2sign;
                      if q2sign='1' then
                            d2<=offsets;
                            d2sign<='0';
                            d3<=q1;
                            z3<=sum3;
                      elsif last='1' then
                            d2<="00000001";
                            d2sign<='1';
                            d1<=sum1;
                            d3<=offsetn;
                            z3<=a3;
                      else
                            d2<=sum2;
                            d2sign<='0';
                            d3<=offsetn;
                            z3<=sum3;
                      end if;
                      dmulreset<='0';
                end if;
          end if;
      end process rekenen;
end structure ;

svs.vhd
------------------------------------------------------------------------------
--  example of a value store
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity svs is
      port (      clk                     : in bit;
```

```
                  init                    : in bit;
                  address           : in  bit_vector(7 downto 0);
                  synweight         : out bit_vector(7 downto 0);
                  synweightsign     : out bit;
                  offsetn                 : out bit_vector(7 downto 0);
                  last                    : out bit );
end svs ;

architecture structure of svs is
component sdff9 is
      port (      clk               : in  bit;
                  data              : in  bit_vector(8 downto 0);
                  s                 : in  bit;
                  q                 : out bit_vector(8 downto 0) );
end component;
signal i0, i1, i2, i3, i4, i5, i6, i7, i8,
           r0, r1, r2, r3, r4, r5, r6, r7, r8: bit_vector(8 downto 0);
signal write: bit;
signal j0, j1, j2, j3, j4, j5, j6, j7, j8,
          offn0, offn1, offn2, offn3, offn4, offn5, offn6, offn7, offn8:
          bit_vector(8 downto 0);

begin
      synweight0: sdff9 port map (clk, i0, write, r0);
      synweight1: sdff9 port map (clk, i1, write, r1);
      synweight2: sdff9 port map (clk, i2, write, r2);
      synweight3: sdff9 port map (clk, i3, write, r3);
      synweight4: sdff9 port map (clk, i4, write, r4);
      synweight5: sdff9 port map (clk, i5, write, r5);
      synweight6: sdff9 port map (clk, i6, write, r6);
      synweight7: sdff9 port map (clk, i7, write, r7);
      synweight8: sdff9 port map (clk, i8, write, r8);
      offsetn0: sdff9 port map (clk, j0, write, offn0);
      offsetn1: sdff9 port map (clk, j1, write, offn1);
      offsetn2: sdff9 port map (clk, j2, write, offn2);
      offsetn3: sdff9 port map (clk, j3, write, offn3);
      offsetn4: sdff9 port map (clk, j4, write, offn4);
      offsetn5: sdff9 port map (clk, j5, write, offn5);
      offsetn6: sdff9 port map (clk, j6, write, offn6);
      offsetn7: sdff9 port map (clk, j7, write, offn7);
      offsetn8: sdff9 port map (clk, j8, write, offn8);

      svsproc: process(clk)
begin
      if init='1' then
            i0<="000000010";  -- synweightvalues, at place 8 is the sign bit!
            i1<="000000011";
            i2<="000000100";
            i3<="000000101";
            i4<="000000110";
            i5<="000000111";
            i6<="000001000";
            i7<="000001001";
            i8<="000001010";
            j0<="000000000";  -- last value at place 8!!, followed by offsetn
            j1<="000000001";
            j2<="100000010";
            j3<="000000000";
            j4<="000000001";
            j5<="100000010";
```

```
            j6<="000000000";
            j7<="000000001";
            j8<="100000010";
            write<='1';
      elsif clk'event and clk='0' then
            write<='0';
            case address is
                  when "00000000" =>
                        synweight<=r0(7 downto 0);
                        synweightsign<=r0(8);
                        offsetn<=offn0(7 downto 0);
                        last<=offn0(8);
                  when "00000001" =>
                        synweight<=r1(7 downto 0);
                        synweightsign<=r1(8);
                        offsetn<=offn1(7 downto 0);
                        last<=offn1(8);
                  when "00000010" =>
                        synweight<=r2(7 downto 0);
                        synweightsign<=r2(8);
                        offsetn<=offn2(7 downto 0);
                        last<=offn2(8);
                  when "00000011" =>
                        synweight<=r3(7 downto 0);
                        synweightsign<=r3(8);
                        offsetn<=offn3(7 downto 0);
                        last<=offn3(8);
                  when "00000100" =>
                        synweight<=r4(7 downto 0);
                        synweightsign<=r4(8);
                        offsetn<=offn4(7 downto 0);
                        last<=offn4(8);
                  when "00000101" =>
                        synweight<=r5(7 downto 0);
                        synweightsign<=r5(8);
                        offsetn<=offn5(7 downto 0);
                        last<=offn5(8);
                  when "00000110" =>
                        synweight<=r6(7 downto 0);
                        synweightsign<=r6(8);
                        offsetn<=offn6(7 downto 0);
                        last<=offn6(8);
                  when "00000111" =>
                        synweight<=r7(7 downto 0);
                        synweightsign<=r7(8);
                        offsetn<=offn7(7 downto 0);
                        last<=offn7(8);
                  when "00001000" =>
                        synweight<=r8(7 downto 0);
                        synweightsign<=r8(8);
                        offsetn<=offn8(7 downto 0);
                        last<=offn8(8);
                  when others =>
            end case;
      end if;
end process;
end structure;
```

B.2 Neural network using RAM and multiplier macro's

networks2.vhd

```
--------------------------------------------------------------------------------
--  neural network implementation using macro's
--  this code is in an experimental phase
--  and should be generalised
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity networks2 is
      port (
            clock : in std_logic;
            init : in std_logic;
            initmul : in std_logic;
         initbias : in std_logic;
         copy1 : in std_logic;
         copy2 : in std_logic;
         copy3 : in std_logic;
            Adebuga1 : out std_logic_vector(7 downto 0);
            Adebuga2 : out std_logic_vector(7 downto 0);
            Adebugsum_out : out std_logic_vector(15 downto 0);
            Adebugmult_out : out std_logic_vector(15 downto 0);
            Adebugactivity : out std_logic_vector(15 downto 0);
            Aneurvalout : out std_logic_vector(7 downto 0);
            Adebugnvsadd : out std_logic_vector(8 downto 0);
            Adebugadd : out std_logic_vector(7 downto 0);
            Adebugsynweight : out std_logic_vector(7 downto 0);
            Adebugq6 : out std_logic;

            Adebugneuronval : out std_logic_vector(7 downto 0);
            Bdebuga1 : out std_logic_vector(7 downto 0);
            Bdebuga2 : out std_logic_vector(7 downto 0);
            Bdebugsum_out : out std_logic_vector(15 downto 0);
            Bdebugmult_out : out std_logic_vector(15 downto 0);
            Bdebugactivity : out std_logic_vector(15 downto 0);
            Bneurvalout : out std_logic_vector(7 downto 0);
            Bdebugneuronval : out std_logic_vector(7 downto 0)
      );
end networks2;

architecture structure of networks2 is
      COMPONENT n3
      port (      clock   : in std_logic;
                  init    : in std_logic;
                  initmul : in std_logic;
                  initbias : in std_logic;
                  copy1 : in std_logic;
                  copy2 : in std_logic;
                  copy3 : in std_logic;
                  AAdebugneuronval : out std_logic_vector(7 downto 0);
                  debuga1 : out std_logic_vector(7 downto 0);
                  debuga2 : out std_logic_vector(7 downto 0);
                  debugsum_out : out std_logic_vector(15 downto 0);
                  debugmult_out : out std_logic_vector(15 downto 0);
                  debugactivity : out std_logic_vector(15 downto 0);
                  neurvalout : out std_logic_vector(7 downto 0);
                  debugnvsadd : out std_logic_vector(8 downto 0);
                  debugadd : out std_logic_vector(7 downto 0);
```

```
                debugsynweight : out std_logic_vector(7 downto 0);
                debugq6 : out std_logic;
                debugneuronval : out std_logic_vector(7 downto 0)
                    );
    END COMPONENT;
    COMPONENT n3b
    port (      clock   : in std_logic;
                init    : in std_logic;
                initmul : in std_logic;
                initbias : in std_logic;
                copy1 : in std_logic;
                copy2 : in std_logic;
                copy3 : in std_logic;
                AAdebugneuronval : in std_logic_vector(7 downto 0);
                debuga1 : out std_logic_vector(7 downto 0);
                debuga2 : out std_logic_vector(7 downto 0);
                debugsum_out : out std_logic_vector(15 downto 0);
                debugmult_out : out std_logic_vector(15 downto 0);
                debugactivity : out std_logic_vector(15 downto 0);
                neurvalout : out std_logic_vector(7 downto 0);
                debugneuronval : out std_logic_vector(7 downto 0)
                    );
    END COMPONENT;
    signal AAdebugneuronval: std_logic_vector(7 downto 0);
begin
    netwerk1: n3 PORT MAP(
        clock,init,initmul,initbias,copy1,copy2,copy3,AAdebugneuronval,
        Adebuga1,Adebuga2,Adebugsum_out,Adebugmult_out,
        Adebugactivity,Aneurvalout,Adebugnvsadd,Adebugadd,Adebugsynweight,
        Adebugq6,Adebugneuronval
    );
    netwerk2: n3b PORT MAP(
        clock,init,initmul,initbias,copy1,copy2,copy3,AAdebugneuronval,
        Bdebuga1,Bdebuga2,Bdebugsum_out,Bdebugmult_out,
        Bdebugactivity,Bneurvalout,Bdebugneuronval
    );


end Structure;


n3.vhd
--------------------------------------------------------------------------------
--  neural network implementation using macro's
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity n3 is
    port (      clock   : in std_logic;
                init    : in std_logic;
                initmul : in std_logic;
                initbias : in std_logic;
                copy1 : in std_logic;
                copy2 : in std_logic;
                copy3 : in std_logic;
                AAdebugneuronval : out std_logic_vector(7 downto 0);
                debuga1 : out std_logic_vector(7 downto 0);
                debuga2 : out std_logic_vector(7 downto 0);
```

```
                debugsum_out : out std_logic_vector(15 downto 0);
                debugmult_out : out std_logic_vector(15 downto 0);
                debugactivity : out std_logic_vector(15 downto 0);
--              debugneuronout : out std_logic_vector(7 downto 0);
                neurvalout : out std_logic_vector(7 downto 0);
                debugneuronval : out std_logic_vector(7 downto 0);
                debugnvsadd : out std_logic_vector(8 downto 0);
                debugadd : out std_logic_vector(7 downto 0);
                debugsynweight : out std_logic_vector(7 downto 0);
--              debugoldweight : out std_logic_vector(7 downto 0);
--              debuglast : out std_logic;
--              debugq4 : out std_logic;
                debugq6 : out std_logic
--              debugwe_a : out std_logic;
--              debugdi_a : out std_logic_vector(7 downto 0)
                    );
end n3 ;

architecture structure of n3 is
COMPONENT layout41
      PORT(
            add : IN std_logic_vector(7 downto 0);
            clock : IN std_logic;
            data_in_a : IN std_logic_vector(7 downto 0);
            data_in_b : IN std_logic_vector(15 downto 0);
            enable_a : IN std_logic;
            enable_b : IN std_logic;
            nvsadd : IN std_logic_vector(8 downto 0);
            reset_a : IN std_logic;
            reset_b : IN std_logic;
            write_enable_a : IN std_logic;
            write_enable_b : IN std_logic;
            neuronval : OUT std_logic_vector(7 downto 0);
            won : OUT std_logic_vector(15 downto 0)
            );
END COMPONENT;
COMPONENT ma
   PORT( clock          :     IN    STD_LOGIC;
         init           :     IN    STD_LOGIC;
         mul_a          :     IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
         mul_b          :     IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
         reset_adder    :     IN    STD_LOGIC;
         carry_out      :     OUT   STD_LOGIC;
         mult_out       :     OUT   STD_LOGIC_VECTOR (15 DOWNTO 0);
         sum_out        :     OUT   STD_LOGIC_VECTOR (15 DOWNTO 0));
END COMPONENT;
component adder9 is
      port (      x           : in  std_logic_vector (8 downto 0) ;
                  y           : in  std_logic_vector (8 downto 0) ;
                  s           : out std_logic_vector (8 downto 0) );
end component ;
component dff9 is
      port (      clk         : in  std_logic;
                  data        : in  std_logic_vector(8 downto 0);
                  q           : out std_logic_vector(8 downto 0) );
end component;
component dff is
      port (      clk         : in  std_logic;
                  data        : in  std_logic;
                  q           : out std_logic );
```

```
end component;

signal nvsadd, svsadd, svsadd2, bvsadd,
                        sum1,sum2,sum3: std_logic_vector(8 downto 0);
signal carry_out,sign_in, q4,q5,q6,q7,q8,q9,q10,d4,d5,d6,d7,d8,d9,d10:std_logic;
signal q1,q2,q2del,q3,d1,d2,d2del,d3: std_logic_vector(8 downto 0);
signal we_a, q2sign, d2sign, q2delsign, d2delsign, biassign,
      synweightsign, last: std_logic;
signal add,nextadd,a1,a2,di_a,bias,neuronval,offsetn,offsets,synweight,
      oldoldweight, oldweight,neuronout   : std_logic_vector(7 downto 0);
signal reset_adder: std_logic;
signal writeadd: std_logic_vector(8 downto 0);
signal mult_out, sum_out, activity, won : std_logic_vector(15 downto 0);

begin
      stageram: layout41 PORT MAP(
            add => add,
            clock => clock,
            data_in_a => di_a,
            data_in_b => "0000000000000000",
            enable_a => '1',
            enable_b => '1',
            nvsadd => nvsadd,
            reset_a => '0',
            reset_b => '0',
            write_enable_a => we_a,
            write_enable_b => '0',
            neuronval => neuronval,
            won => won
      );
      stagemul: ma port map (clock, initmul, a1, a2 , q5, carry_out, mult_out,
sum_out);
      adder91: adder9 port map (q1, "000000001", sum1);

      stage1: dff9 port map (clock, d1 , q1);
      stage4: dff port map (clock, d4 , q4);
      stage5: dff port map (clock, d5 , q5);
      stage6: dff port map (clock, d6 , q6);
      stage7: dff port map (clock, d7 , q7);
      stage8: dff port map (clock, d8 , q8);

      rekenen: process(clock,won,copy1)
      begin

            debugsum_out<=sum_out;
            debugmult_out<=mult_out;
            debugactivity<=activity;
--          debugneuronout<=neuronout;
            debugneuronval<=neuronval;
            AAdebugneuronval<=neuronval;
            debugnvsadd<=nvsadd;
            debugadd<=add;
            debuga1<=a1;
            debuga2<=a2;
--          debugq4<=q4;
            debugq6<=q6;
            debugsynweight<=synweight;
--          debugoldweight<=oldweight;
--          debuglast<=last;
--          debugwe_a<=we_a;
```

```
--          debugdi_a<=di_a;
            neurvalout<=neuronval;

            if copy1='1' then
             nvsadd<="100001000";
             a1<="00000000";
            elsif copy2='1' then
             nvsadd<="100001001";
             a1<="00000000";
            else
                  if q5='0' and q6='0' then
                        synweight<=won(15)&won(15)&won(15 downto 10);
                        last<=won(9);
                        nvsadd<="100000"&won(8 downto 6);
                        add<=nextadd;
                  end if;
                  if q5='1' then
                  if sum_out(15)='1' or sum_out(14)='1' or sum_out(13)='1' or
                  sum_out(12)='1' or sum_out(11)='1' or sum_out(10)='1' or
                  sum_out(9)='1' or sum_out(8)='1' or sum_out(7)='1'
                  then
                        nvsadd<="000010000";
                  else
                        nvsadd<="00000"&sum_out(6 downto 3);
                  end if;
                  end if;

                  if q5='1' and q6='0' then
                        activity<=sum_out;
                  end if;
                  if q6='1' then
                        nvsadd<=q1;
                        we_a<='1';
                        di_a<=neuronval;
                  end if;
                  if q7='1' then
                        we_a<='0';
                  end if;
                  if q5='1' or q6='1' then
                        a1<="00000000";
                  elsif initmul='0' then
                        a1<=oldoldweight;
                  elsif init='0' and initmul='1' then
                        a1<="11111111";
                  else
                        a1<="00000000";
                  end if;
            end if;

            if init='1' then
                  d1<="100000011"; -- will be read from nvs later!
                  add<="00010100";
                  nextadd<="00010101";
                  nvsadd<="100000000";
                  reset_adder<='0'; activity<="0000000000000000";
                  a1<="00000000"; a2<="00000000"; di_a<="00000000";
                  d4<='0';d5<='0';d6<='0';d7<='0'; we_a<='0';
                  d8<='0';--d9<='0';d10<='0';
            elsif clock'event and clock='0' then
```

```
                    d4<=last;
                    d3<=sum2;
                    if q5='0' and q6='0' then
                          oldweight<=synweight;
                          oldoldweight<=oldweight;
                          nextadd<="00"&won(5 downto 0);
                    end if;
                    if q5='1' then
                          d1<=sum1;
                    end if;
                    if q5='1' or q6='1' then
                          a2<="00000000";
                    elsif q7='1' or q6='1' then
                          a2<="11111111";
                    elsif init='0' and initmul='1' then
                          a2<="01111000"; --hack
                    else
                          a2<=neuronval;
                    end if;
                    d5<=q4;
                    d6<=q5;
                    d7<=q6;
                    d8<=q7;
              end if;
        end process;
  end structure;
```