Implementing Artificial Neural Network Designs: Final Report

Rob Chapman (806461), Steven Sutankayo (224431)

1 Abstract

With the increasing density of programmable hardware and the usability of hardware programming languages, it has become more attractive to implement, in hardware, a wider range of designs traditionally done in software. Artificial Neural Networks (ANN) is one such area. This report describes the implementation of two ANN designs. Each ANN is implemented in two different ways. One ANN is a simple pattern separator and is used to indicate when two inputs are different. While this is not a demanding application for ANNs it illustrates the principles and neuroprocessor (NP) model rather well. The second ANN is used to recognize digits in a seven segment display, when they are as they should be, and even with a little distortion. The two different implementations allow us to explore alternate design trade-offs for implementing an ANN. One implementation is single processor based while the other consists of many optimized processors. The two ANN designs are described in VHDL and targeted for the MAX and FLEX chips on the Altera UP1 board.

2 Overview

Conventional digital serial algorithms described in programming languages express much of the way in which we think: serially, connected, thought after thought. Our consciousness is modeled by such programs, but what of the hardware which runs it? Fast digital computers with mega/giga potential try to run these programs fast, but there are still many problems for which they will never be fast enough or even close in skills when compared to a very different hardware platform, a bio-neural net (BNN) — the brain.

While the digital computer makes spatial as well as temporal connections in time, a BNN makes spatial connections in parallel, allowing for highly connected problems to be solved in much shorter times than a digital computer. Problems like image recognition and visual navigating are easily performed by a BNN. There is a well trod path for the digital computer evolution, but, for a certain set of problems, a BNN structure will always be superior. ANNs simulating some qualities of a BNN can be run on a digital computer, but then, the power they gain in pattern emulation, is lost in time spent running all the parallel connections serially. What is needed, is to drastically simplify the digital computer, let it simulate a neuron instead of a network, clone many of them and connect them with weights corresponding to an ANN design. This report implements two different ANNs with two different architectures.

The first ANN design was chosen because it was simple and it would be a good debugging aid in getting the NPNs up and going. The Second ANN design was a little more complicated but it used a

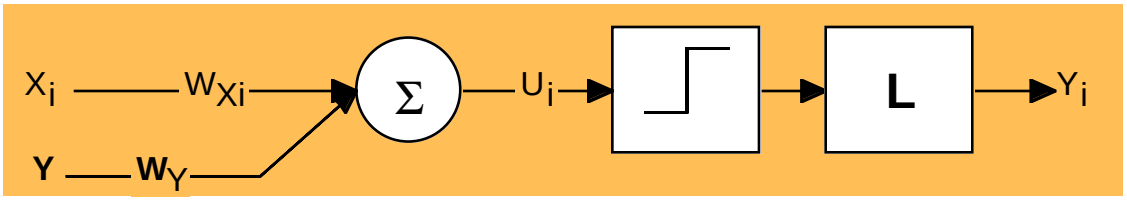
more mature interface where we could plug in different NPN implementations without changing a thing other than doing a recompile. This proved to be very powerful, exhibiting the good things that VHDL was meant for. As a matter of fact, it turned it into a bit of a competition to see who could get their NPN working first. The single processor NPN won out as it was much easier to modify for a new set of weights. The multiple processor NPN lacked a full generic interface and progress was hampered by compiler bugs.

In the original design, the user interface (UI) components were to be a mouse and a VGA display. Due to time constraints, these were supplanted with UP1 UI components. Also in the original design, the second ANN was to be a digit recognizer consisting of sixty NPs. Earlier measurements indicated that it would be difficult or impossible to fit the NPN inside of the given FLEX chip so the ANN was modified to work with fourteen NPs and instead of a five by six grid of pixels, a seven segment LED display was used.

This document describes the NP model and then goes through two different architectures. These two architectures are then used to implement two different ANN designs. Finally the results are tallied and a few ideas are explored.

3 Neural Network Components

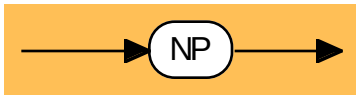
When transferring ANN designs to hardware, it helps to have a simple model to describe them in. For our designs, we make use of the neuroprocessor (NP) model described previously[1].



The model presented here for a neuron has one input (X_i), plus all the outputs (**Y**), feeding into a summing junction (Σ) whose output feeds a hard limiting activation function ⎍ where the result is held in the output latch (L).

Each NP has only one input. If there are several inputs, then several NPs will be needed with one of them serving as a summing junction to sum several inputs. Although one may be struck by this seemingly lavish use of an NP, this is only the model. When translated to hardware, certain optimizations will be performed taking advantage of the simplicity of the model, reducing it to a minimum circuit (Section 7.7, “NP Reduction,” on page 20).

A more compact version of the model which is used in the paper is:



3.1 Sum, Sign and Latch

Inputs and outputs are binary in value. In hardware they are the logic values 0 and 1. For modeling, they are the sign bit of the sum. The logic value 1 represents a negative value and the logic value 0 represents a positive value. The weights are any positive or negative integer value. Summing all the

inputs is simply done by adding up the weights, changing the sign of a weight if the input value is negative. The activation function takes the sign of the resultant summation and presents it to the latch. The latch passes the value on to the output on a positive clock edge. The equations for an NP are:

$$U_i = X_i \times W_{X_i} + \sum_{j=1}^{N} Y_j \times W_{Y_{j,i}} \qquad \text{(EQ 1)}$$

$$Y_i(t+1) = \operatorname{sgn} U_i \qquad \text{(EQ 2)}$$

where X_i and Y_j are either 1 or -1.

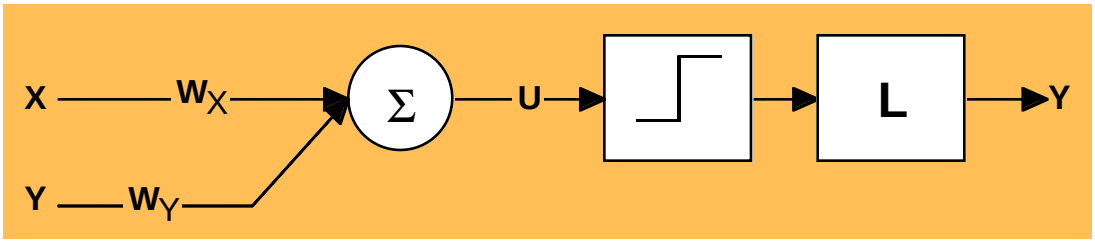
3.2 NP Pseudocode

The requirements of the NP are quite simple and they may be described in pseudocode:

```
forever
    choice = 0
    sum = weight[choice] * (-2 * input[choice] + 1)
    till choice = N
        choice + 1
        sum + weight[choice] * (-2 * input[choice] + 1)
    do again
    output = sum < 0
do again
```

A continuous summation is done over the weight matrix negating the weights when the respective input is negative. To transfer the mathematical sign from the input to the weight, the input is converted from the value zero or one to one or minus one by multiplying by minus two and then adding one. The weight is then multiplied by this value effectively changing the sign of the weight. At the end of the loop, the sign of the summation is output.

3.3 Topologies



A network of NPs or an NP net (NPN), can be shown by feeding all the inputs and outputs into a summing junction. The X and Y inputs and W_x weights for the X inputs are Nx1 matrices while W_y weights for the Y inputs is a NxN matrix, where N is the number of NPs. All topologies are represented by this diagram, but when implementing an ANN design, the NPs are usually drawn to illustrate the structure eliminating the lines between NPs which have a weight of zero. This technique will be used later to describe two ANN designs.

3.4 Grand Matrix

It is possible to describe all topologies with one sparse weight matrix of dimension (N+1)xN, where N is the number of NPs in the network. The NPs are numbered from 1 to N. Each row of the matrix is the transposed weight vector for that NP. The weights for the inputs are in the zeroth column of the

matrix. The first to the N^{th} column represents the weights for the outputs of all the NPs fed back as inputs. This matrix is called the grand matrix (GM). For most, if not all topologies, the GM will be heavily populated with zeros or ones representing the connections between nodes. It is derived in vector notation:

$$\begin{aligned} U &= XW_X^T + YW_Y^T \\ &= \begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} W_X^T & W_Y^T \end{bmatrix} \\ &= P \times W_{GM} \end{aligned} \qquad \text{(EQ 3)}$$

where P is the input vector containing all the external inputs and the outputs fed back as inputs.

3.4.1 GM Pseudocode

The GM acts like a massively parallel memory where each NP accesses their weights independent of the other NPs. Pseudocode for this operation is:

```
for each NP from 1 to N
  NPweightvector = gm[NP]
loop
```

3.4.2 Minimizing Precision

Given that the target implementation is to be hardware, this gives us certain flexibility and some limitations. Since we are not constrained to 32-bit arithmetic, we can choose the minimum necessary precision for the weights and the sum to reduce the number of bits required to implement the ANN design. This minimizing allows designs to fit within the limits of the target hardware.

4 Resources

The target language chosen to describe the designs is VHDL. There is good support for the language and it can span descriptions from simple connections to general behavioural descriptions. The target programmable hardware is a 2500 gate MAX chip with flash memory and a 20,000 gate FLEX chip both from Altera. Both of these chips are supplied with a programming environment called the UP1 which Altera has made available to Universities for a reasonable cost.

4.1 VHDL

As a measure of the power of VHDL to capture different design architectures, we implemented the ANN designs using two completely different architectures. The first architecture uses writable computer parts [3] to build an NP, describing the algorithms in program memory and instruction sequences. The NP operates much like a conventional processor and runs a program which allows it to work through all the nodes calculating the outputs sequentially. The second architecture trims down the NP to a set of minimum pieces to try and create a small enough image so that every node in the design has its own NP.

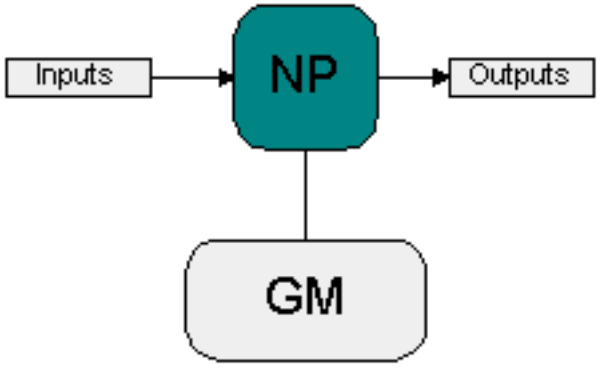
4.2 FPGA Technology

Two different chips from Altera are used to implement the ANN designs. They are the MAX 7128S84 and the FLEX 10K20-240. The FPGA architectures work well with ANN designs modeled with NPs. There are plenty of input and output pins, internal connections are very cheap for the massive

interconnections required, lots of space for NPs on a single chip and the clock rate is high enough to do real time reactions.

5 Single NP Implementation

This architecture is similar to a microprocessor-based ANN. A single neuroprocessor computes the sums for each node serially. The ANN's inputs, outputs, and weights are multiplexed into a neuroprocessor core.

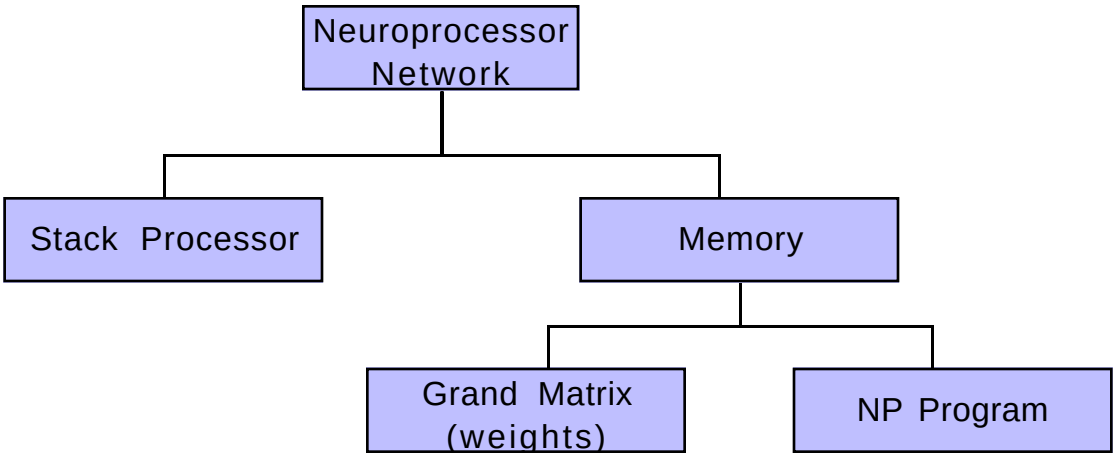


This has the greatest scalability in terms of remaining small since the only thing that increases is the size of the weight matrix.

The trade off here, clearly, is speed. One NP has to be each NP, consecutively, eliminating any speed that could be gained by running multiple NPs in parallel. Thus the total run time of the network goes up linearly with each NP added. On the other hand, if this is within the time constraints of the required solution, then it is an acceptable solution.

> NOTE: Also there is only one actual NP in this implementation, this section will refer to each node as an NP to be consistent with the rest of the document.

5.1 Architecture

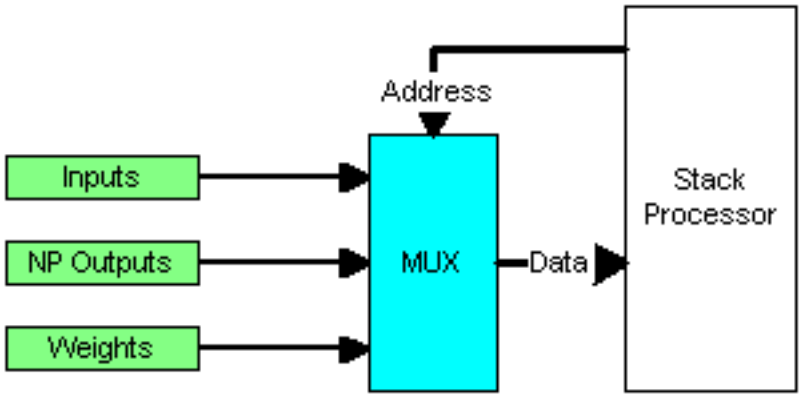


All weights for the ANN may be stored in a single block representing the grand matrix for the network.

A network of N NPs requires an N x (N+1) sized grand matrix multiplied by the weight bit precision to determine the number of bits.

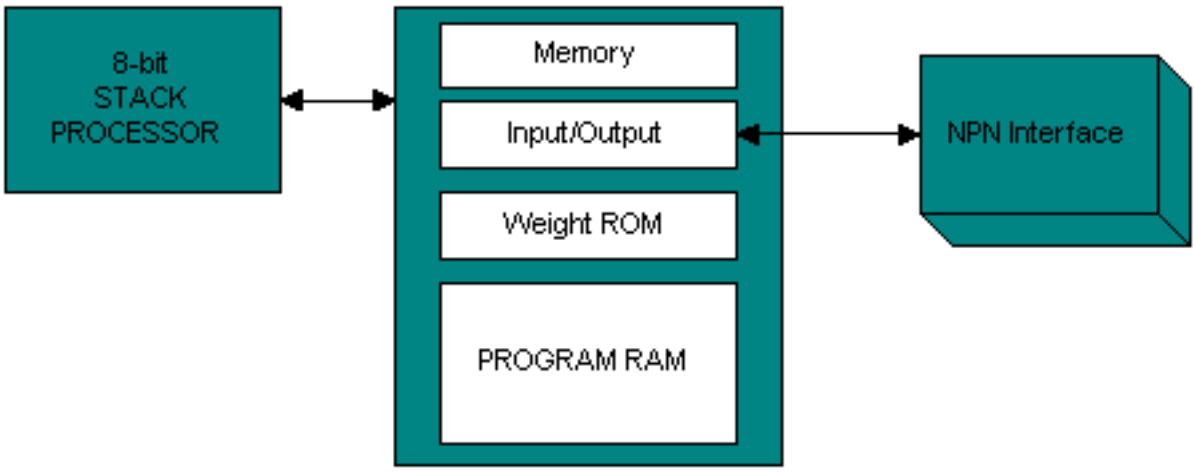
5.2 Design

The nature of the single-NP ANN implementation is that of a processor based system. A stack-based processor is the core of the network. It computes the NP sums and interfaces with the rest of the network via its data and address buses. Since no tri-state buffers are available inside the FPGA, the inputs, outputs, and weights must be multiplexed to drive the shared data bus.



It is our goal to implement neural nets of different size, and it was originally thought that the width of the data and address busses would increase with the number of NPs, to handle the larger grand matrix size. This was not necessary. In order to address larger weight arrays, the weight memory block was made external to the processor's memory space by using a weight register to address the "external" memory. This demonstrated one great advantage of the FPGA implementation, because the difference between the two methods of accessing the weights was merely a few extra processes to multiplex the grand matrix to the data bus.

A system view:



5.2.1 Instructions

This table lists the available instructions for the NP:

Instruction	Description	Action
psh_mem_DS	push the contents of the given memory location to the data stack	DS_index <= DS_index - 1 DS(index)<= memory(PC+1)
pop_DS_TOP	pop the data stack and store its contents to TOP	TOP<= DS(index) DS_index<= DS_index + 1
sto_DS_memimm	store the contents of the data stack to the given memory location	memory(PC+1)<= DS(index)
sto_memimm_DS_and_sto_DS_TOP	fetch the contents of the given memory address and store to the data stack, while storing the current data stack to TOP	TOP<= DS(index) DS(index)<= memory(memory(PC+1))
bra_mem	branch to the given memory location	PC <= memory(PC+1)
bnzero_imm	branch to the given memory location if TOP is nonzero	if TOP != 0 PC<= memory(PC+1)
bzero_imm	branch to the given memory location if TOP is zero	if TOP = 0 PC<= memory(PC+1)
add	add TOP to the data stack, place the result in the data stack	DS(index) <= DS + TOP

Instruction	Description	Action
subtract	subtract TOP from the data stack, place the result in the data stack	DS(index) <= DS - TOP
sto_mem_DS	store the contents of the given memory location to the data stack	DS(index <= memory(PC+1)
psh_memptr_DS	push the contents of the location referenced by the given pointer to the data stack	DS(index) <= memory(memory(memory(PC+1)))
sto_DS_TOP	copy the data stack to TOP	TOP <= DS
swap	exchange the contents of the data stack and TOP	TOP <= DS, DS <= TOP
sto_TOP_DS	copy TOP to the data stack	DS <= TOP
incr_ptr	increment given pointer by 1	ptr = memory(memory(PC+1)) memory(ptr) <= memory(ptr) + 1
sto_DS_memptr	store the data stack to the location referenced by the given pointer	ptr = memory(memory(PC+1)) memory(ptr) <= DS

5.2.2 Program

```
Pseudocode for the ANN program:

-- START

-- N => number of Neuroprocessors in the ANN

input_ptr = INPUT_START
weight_ptr = WEIGHT_START
np_counter = N
```

```
NP loop:

   input_counter = N
   np_output_ptr = NP_OUT_START
   output_ptr = OUTPUT_START
   sum = 0

   summation loop:

       fetch weight
       fetch input

       if input = 0
           sum = sum - weight
       else
           sum = sum + weight
       end if

       weight_ptr = weight_ptr + 1

       if input_counter = 0 then
           exit loop
       else
           input_counter = input_counter - 1
       end if

       fetch weight
       fetch np_output

       np_output_ptr = np_output_ptr +1

   end summation loop


   #perform hard-limiting function by storing sign bit of resultant sum
   *output_ptr = MSB(sum)
   output_ptr = output_ptr + 1

   if np_counter = 0 then
       exit NP loop
   else
       np_counter = np_counter + 1
   end if

end NP loop
```

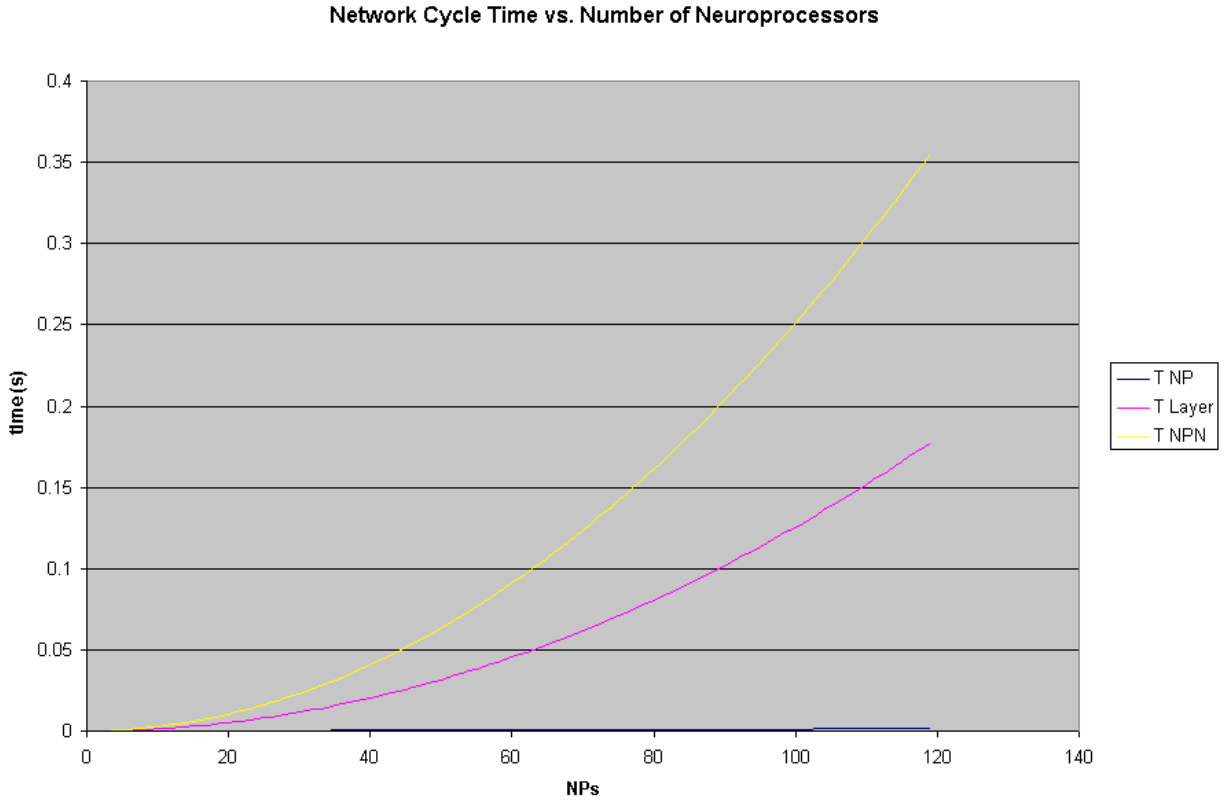
5.2.3 Assembler

An assembler written in Perl was included in the Appendix of the previous document.

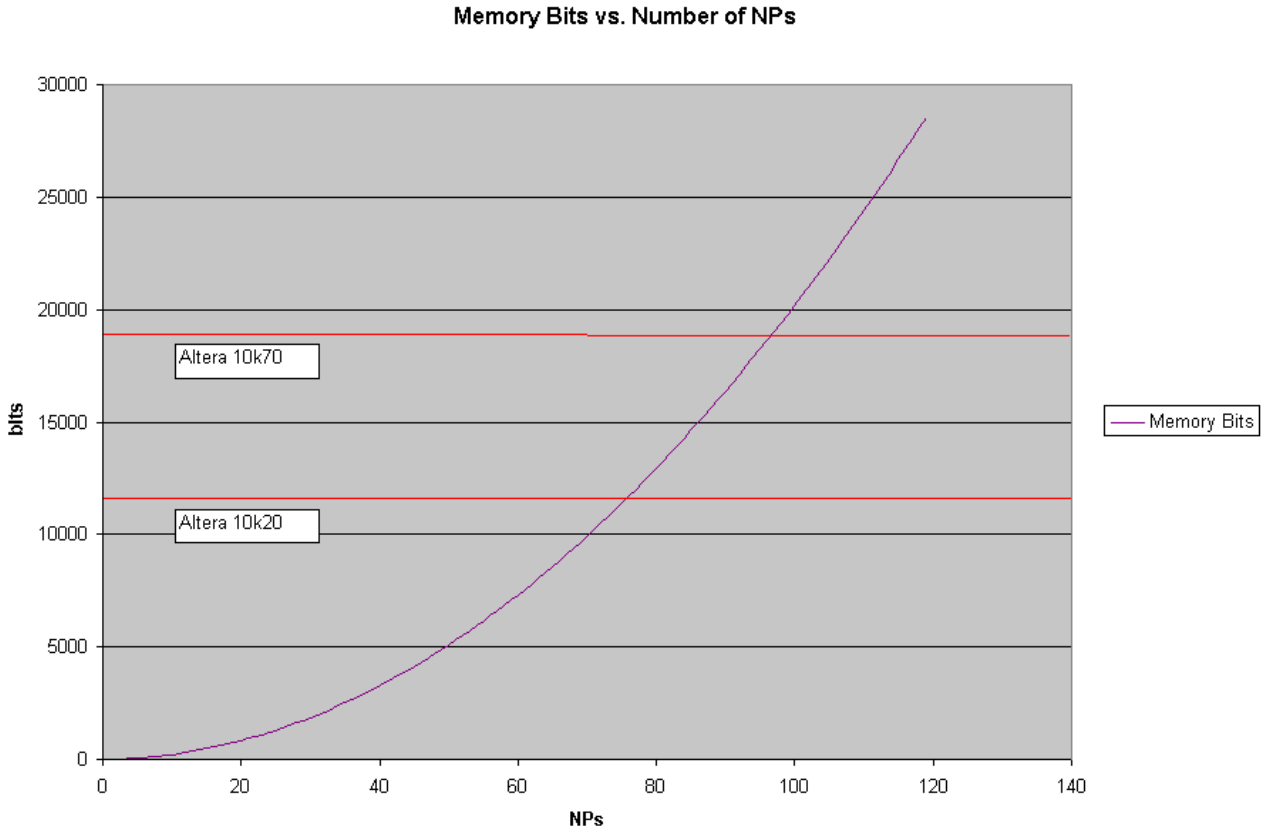
5.2.4 Memory and Cycle Time versus NPs

This graph shows what happens as the number of NPs is increased. The time for a layer strongly

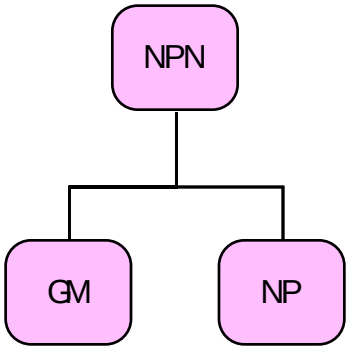
depends on the topology and it can vary anywhere between the line for an NP and for and NPN.



This graph show the relationship between memory size and the number of NPs for a fixed precision:

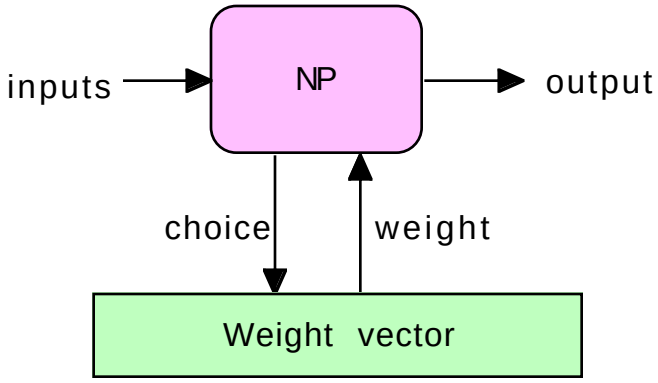


6 Multiple NP Implementation



In this architectural approach, there are only three components to the design, but there is one NP instance for each NP in the ANN design. The GM contains all the weight vectors for all the NPs. Each NP in the network is a copy of an optimized dedicated processor summing a weight vector for a set of inputs. The NP is described with a simple behavioural description as this allows many to fit inside of an FPGA. In small designs, such as the ones in this report, it is prudent to implement the GM as a truth table in VHDL with case statements.

6.1 NP



An NP has 1+N inputs, where N is the number of NPs in the NPN, and one output. The zeroth input is an external input to the network. The other N inputs are the sign bits of the outputs of the other NPs. The weight vector is shown separate from the NP here as it resides in the GM component but it accessed exclusively by one NP.

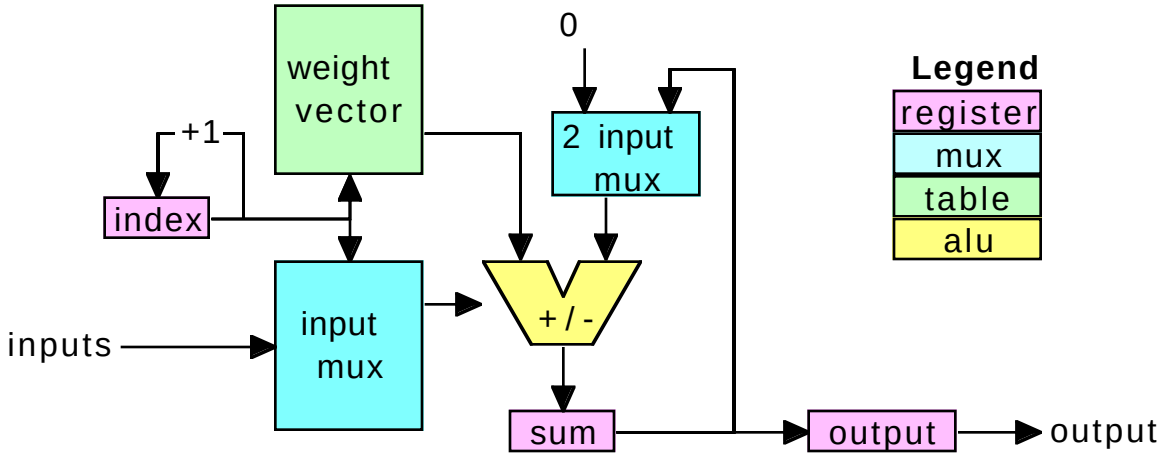
6.1.1 Architecture

The NP itself is described in about 40 lines of behavioural VHDL code:

```
entity sevennp is
  port(clock  : in std_logic;  -- synchronous design
       input  : in std_logic;  -- external input
       inputs : in iobus;      -- inputs from outputs
       weight : in  aweight;   -- weight from GM
       choice : out achoice;   -- which weight from GM
       output : out asum);     -- the final sum
end entity;

architecture behaviour of np is
  signal index         : achoice;                  -- for choosing weights
  signal sum, next_sum : asum;                     -- for summing
  signal np_input      : std_logic_vector(0 to N); -- vector for all inputs

  begin

    choice <= index;              -- index is passed onto the GM

    np_input(0) <= input;         -- actual input
    np_input(1 to N)  <= inputs;  -- output inputs

    with np_input(index) select   -- negate the weight if input is not 0
      next_sum  <=  weight when '0',
                   -weight when others;

    process  -- run NP algorithm
    begin
      wait until clock = '1';

      if  index = N  then       -- end of summation
        index  <= 0;              -- reset index
        output <= sum;            -- output final sum; NPN extracts sign
        sum    <= next_sum;       -- initial sum
      else
        index  <= index + 1;      -- increment index
        sum    <= sum + next_sum; -- sum the weight
      end if;
```

```
    end process;
end architecture behaviour;
```

This code is compiled down to optimal hardware for executing the NP algorithm. A block diagram which helps illustrate the data flow described in this code is:



The index counter, which increments up to N and then starts again at zero, is used to select an input and a weight. The input causes an addition or a subtraction of the weight from the current sum or zero to flow back into the sum. The sum is passed to the output register when finished.

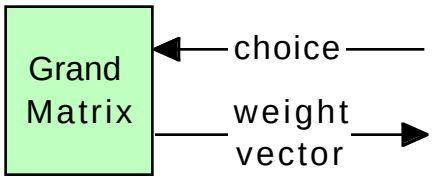
6.1.2 NP Test

A simple NP test is to connect one input to zero and one while holding all the other inputs to zero, vary the corresponding weight between 1 and -1 and check the outputs. This table describes an expected

input	weight	output
0	1	1
1		-1
0	-1	-1
1		1

input output pattern. The input is a single logic bit while the weight and output are multiple bit precision values. This was used to verify that the NP code was working before using it as a component in an NPN.

6.2 GM



Several architectures were attempted for the GM, and as it turns out, for the small ANNs implemented in this report, the truth table approach proved to work best. An example of one of the weight vectors described in a VHDL process is:

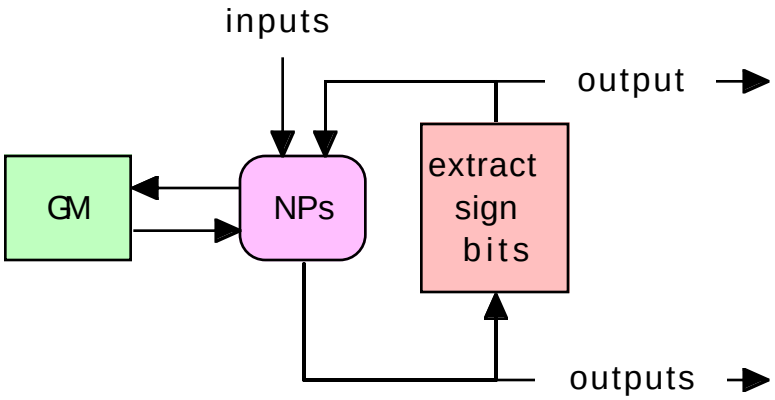
```
process(choice(9))    -- be sensitive to NP9's choice
begin
  case choice(9) is   -- supply the weight vector
    when 2      => weight(9) <= one;
    when 6      => weight(9) <= minus1;
    when others => weight(9) <= zero;
  end case;
end process;
```

In the above code, the weight values have been implemented as constants. This is part of the design which is in flux. It is a matter of convenience to be able to express the weights as integers but the current design uses std_logic_vectors as the signal types and problems were experienced with the conversion functions on negative numbers. So as an inconvenient but certainly less than elegant solution, the weights have been coded as constants of the type std_logic_vector. With due diligence, a more elegant method, that the compiler agrees with, could be found.

The ROM and RAM megafunctions supplied with the Altera compiler tools were tried but they proved to be very capricious and difficult to get just right. Their benefit is that they allow the GM component to be made generic and therefore usable in many designs by supplying the weights in a memory initialization file.

On one of the target chips, it is possible to have logic implemented in the on-chip embedded array blocks (EAB)s. For the second ANN, this option was attempted for the GM but the EAB capacity was exceeded by a small margin so this compile option was not used.

6.3 NPN



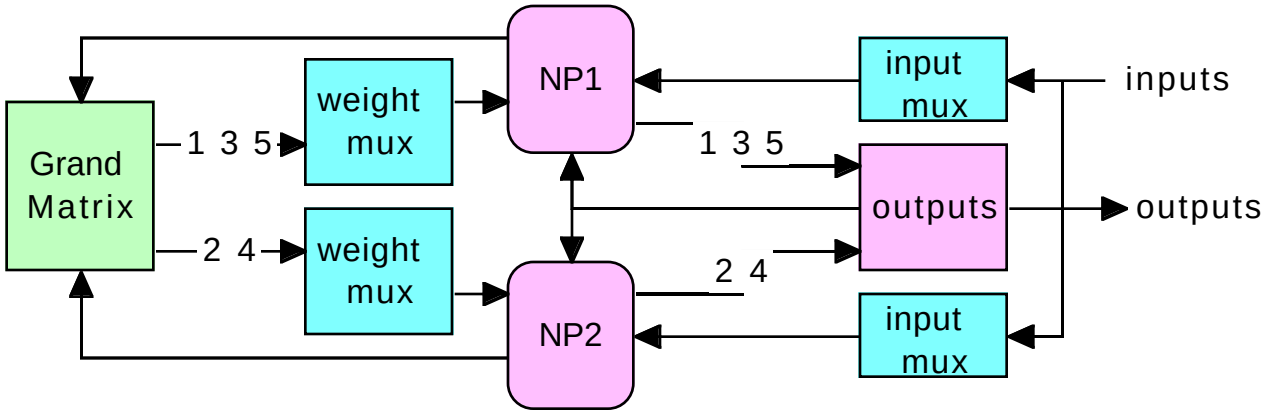
The NPN component mostly consists of wires and instances of the NPs and the GM. It also contains the activation functions for all the NPs (extract sign bits) which is simply just a wire for each sign bit from the outputs. The full precision of the NP outputs is available for monitoring. This is taken advantage of later by the user interface in the two ANN designs.

6.4 Modifying for Different ANNs

This multiple NP design provided some difficulty in moving to different ANN designs and different chips. Some of the difficulty exists because the architecture is not mature enough to be fully generic.

Another difficulty arises in modifying the VHDL code for the GM as it is hand coded for each new set of weights in the form of case statements. Two ways around this are to either make use of a memory megafunction, which won't be portable across different compilers, or to generate the VHDL file automatically from a set of weights. In the single NP design, the mif file for the memory megafunction is automatically generated from the GM.

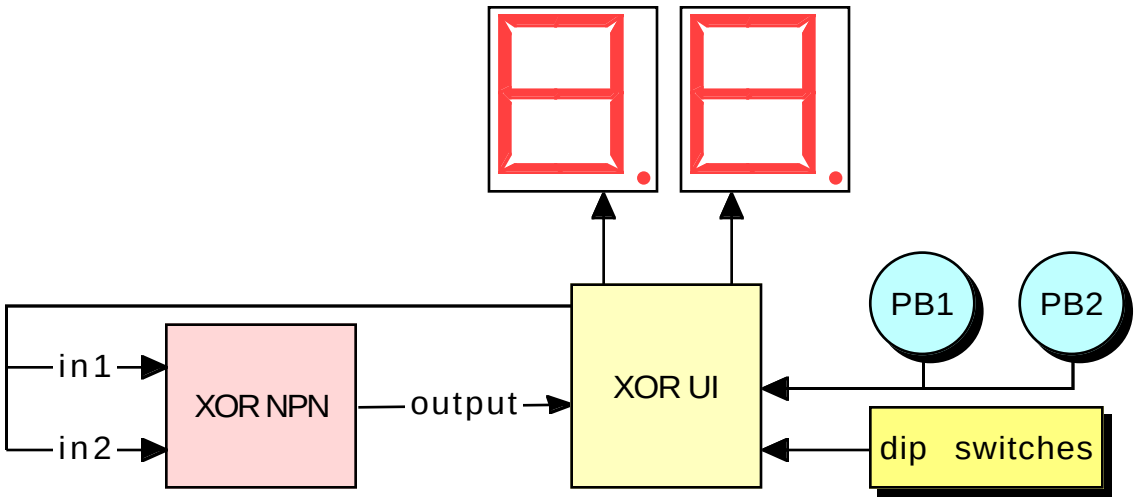
For the smaller target chip, there was room left for only two NPs so they were muxed to the GM with a counter cycling them through as different NPs. This complicated the design a certain amount but it was necessary to fit the NPN and the user interface into the chip. This design optimization is discussed further in Section 11.2.1, "One NP per String," on page 27. This is a basic block diagram for such an architecture:



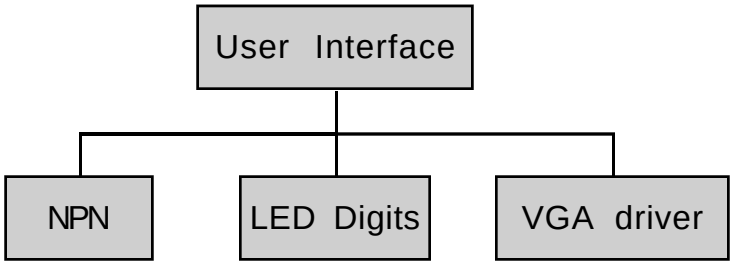
7 A Difference Detector

Detecting a single input pattern with an ANN is fairly simple and with the proper training set, it can be detected from a vast array of input patterns. Detecting when inputs are different, however, adds another layer to the ANN design.

This ANN design implements a simple two input difference detector which outputs a one when the inputs are different and a zero when they are the same. In logic, this function is provided by an XOR gate which is a two level boolean logic equation. As a result, the NPN is referred to, in places, as the XOR NPN. The XOR UI is interfaced to two dip switches for the inputs, two push buttons to control the NP monitoring and two seven segment led digit displays to display the output and NP sums.

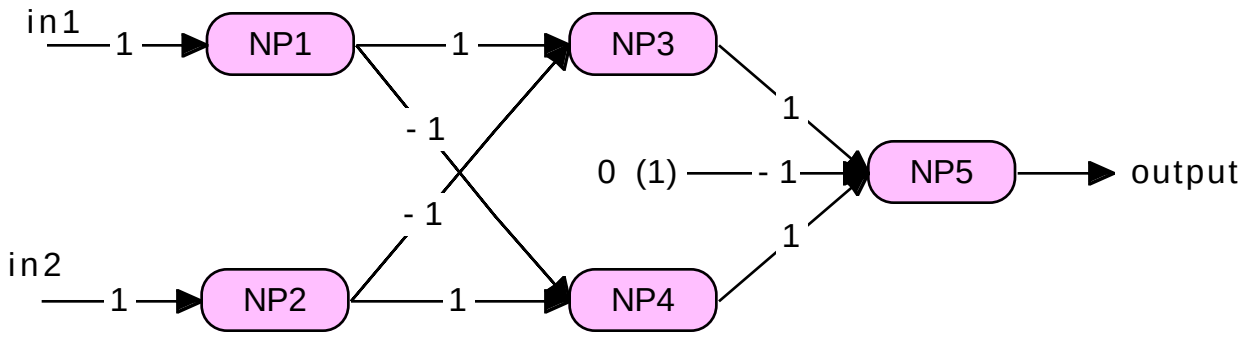


7.1 Component Hierarchy



The user interface is the top level entity. It creates instances of the NPN, the LED display driver and the VGA driver. The VGA driver was not used and a stub component was used during the testing. With this component style architecture, different NPN components can easily be substituted without affecting the other components.

7.2 NPN Design



In this ANN design there are few connections so it is easy to draw the NPs connected to illustrate the structure. The two inputs, `in1` and `in2`, accept a binary value and `output` outputs a binary value. Each NP outputs a logic zero or one which is interpreted mathematically as a one or minus one by the next NP. The fifth NP uses one input as an offset value to obtain the proper output (this use of an offset value to shift the threshold of the NP is discussed in Section 7.7, "NP Reduction," on page 20). It's input is tied to logic zero which is mathematically one while the weight is set to -1. This table shows all the possible inputs, the output and the NP summations:

Logic: Response			**Math: Summations**				
in1	**in2**	**output**	**NP1**	**NP2**	**NP3**	**NP4**	**NP5**
0	0	0	1	1	0	0	1
0	1	1	1	-1	2	-2	-1
1	0	1	-1	1	-2	2	-1
1	1	0	-1	-1	0	0	1

As required, the output is one when the inputs are different and zero when they are the same.

Each summation is the sum of each weight times its input. Each NP has one external input and all the outputs from all the NPs as inputs. The GM containing all of the weight vectors for the NPs in the difference detector NPN is:

Weight vectors	Input source					
	external	output1	output2	output3	output4	output5
NP1	1	0	0	0	0	0
NP2	1	0	0	0	0	0
NP3	0	1	-1	0	0	0
NP4	0	-1	1	0	0	0
NP5	-1	0	0	1	1	0

7.2.1 Weight Precision

Since the weights range from -1 to 1, only two bits are required to encode the weights. And as we can

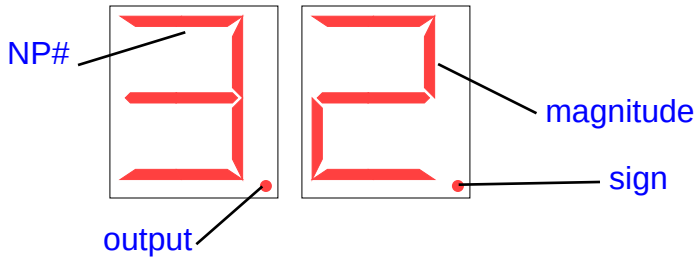
see from the summations, the sums range from -2 to 2 requiring a precision of 3 bits.

7.3 Simulation

The timing diagram for the MAX implementation of the XOR NPN is included on the next page.

7.4 User Interface Design

For simplicity and to reduce the scope of the project, we decided to make use of the user interface components supplied on the UP1 board. This consists of eight dip-switches, two push-button and two seven segment displays:



The two displays are used to monitor the NP sums and the NPN output. The left digit displays the current NP number which is being examined while the right digit displays its final sum. The dot on the right digit is the sign since there is no sign segment. The dot on the left digit is the output of the network. So reading this display tells us that the output of the network is one, indicating that the inputs are different, and the final sum of NP3 is -2. The NP number is incremented and decremented with the left and right push-button. Two of the dip switches are used as inputs to the network where up is minus one and down is plus one.

When digging through the user interface file, one will find a declaration of a VGA driver component. It was intended to be used for a visually appealing and informative display of the network but in the end, it was not used due to time constraints. As it turned out, the onboard UI components proved to be quite adequate for the task.

7.4.1 Push Buttons

The push buttons are momentary contact switches. The circuit interfacing to them must produce a single activation for a single press and it must provide debouncing as well. The circuit used in this UI is a little bulkier and less refined than the one used in the second ANN UI. Nonetheless, it performed adequately. It is based on the push button providing a continuous pressed signal for one millisecond. This done by counting up when pressed and counting down when not pressed. So when the switch bounces, the count will go up and down. When a terminal count is reached the counting is halted and the NP number is incremented or decremented depending on which push button was pressed. The counting is re-enabled when the push button is released. In this UI, the push button code and the incrementing and decrementing of the NP number is all within one process. In the second UI, the push button process is factored out.

7.5 Test

To test the user interface, a stub NPN was used:

```
-- XOR NPN stub  Rob Chapman  Mar 7, 1998
```

```
Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

use work.params.all;

entity xor_npn is
  port ( in1, in2, clock : in std_logic;
         output          : out std_logic;
         outputs         : buffer std_logic_vector(14 downto 0));
end entity;

architecture stub of xor_npn is
begin

  output <= '1';
  outputs(2 downto 0) <= "000";
  outputs(5 downto 3) <= "001";
  outputs(8 downto 6) <= "010";
  outputs(11 downto 9) <= "111";
  outputs(14 downto 12) <= "110";
end stub;
```

This provided the user interface with the necessary stimulus to test out the functionality. The NP number was incremented and decremented with the push buttons and the display was verified for the proper digits. The UI was debugged first, as this gives us a powerful tool to debug the NPNs. It was a lot faster to download the NPN and run it on the board than it was to setup the simulation waveforms, run them and then try and interpret the results. The UI gave a direct read of the NP sums and they were compared to the sums from the table. If they were in error, then it was usually quite easy to find the fault in the VHDL code. For the single NP architecture where simulation was quite cumbersome due to the number of components and the number of cycles required, the UI was a much better debugging tool in helping straighten out bus wiring and other problems.

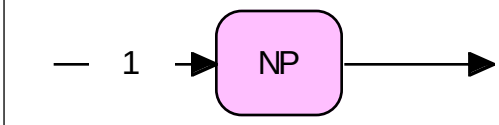
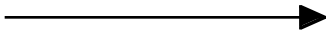
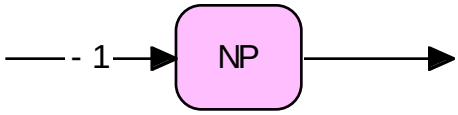
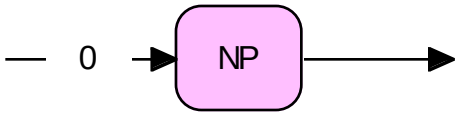
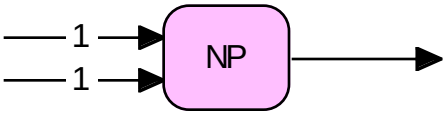
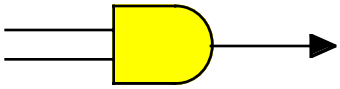
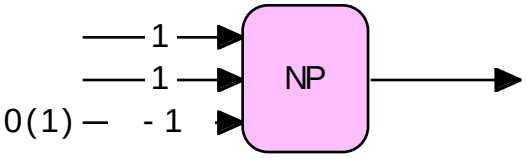
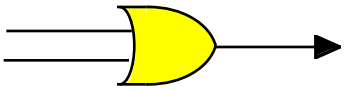
7.6 MAX and FLEX

The ANN design was implemented in both chips with the multiple NP design (modified for the smaller chip) going in the MAX chip and the single NP design in the FLEX chip.

7.7 NP Reduction

For certain weight vectors for an NP, the NP (and its weights) can be replaced with simpler components. This allows the hardware to be reduced and it also relates the NP as a sort of superset of logic gates with equivalent logic gates for certain configurations. This table lists some equivalent gates

for certain weight vectors.

NP circuit with weights	Equivalent circuit	Comment
		When there is only one non-zero weight and it is a one, then the equivalent circuit is a wire.
		When there is only one weight and it is minus one, then the equivalent is just an inverter.
		When all the weights are zero, then there is no connection.
		If there are two inputs and they both have weights of one, then the equivalent circuit is an AND gate.
		If there are two inputs with a weight of one and one input tied to zero with a weight of minus 1, then the equivalent gate is an OR gate.

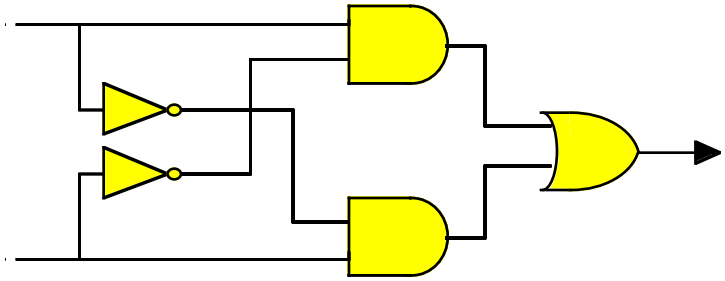
Let's have a closer look and see what's going on with the AND and especially the OR gate. Recall that a logic zero is a mathematical one and a logic one is a mathematical minus one. Plugging these values into an NP with two inputs with weights of one, and obtaining the outputs we get:

Inputs				Outputs	
Logic		Math		Math	Logic
0	0	1	1	2	0
0	1	1	-1	0	0
1	0	-1	1	0	0
1	1	-1	-1	-2	1

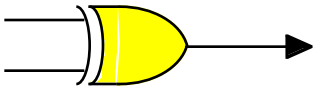
We see that we get three different values for the summation. Since addition is commutative, it doesn't matter which input is one when the inputs are different. For three of the cases the sign of the output is positive so the logic output is zero. For the last case, the output is negative so the logic output is one. Now when we add a third input which is tied to logic zero and it has a weight of -1, this shifts the math values down by one so only the first case is positive and the last three are negative. This gives the equivalent of an OR gate. In effect, the third input acts like a threshold shifter and with this we can propose that an OR gate is an AND gate with a different threshold.

Now if we apply these NP reductions to the XOR NPN, we can replace it with the following logic

circuit:



Which anyone in a basic logic course would recognize as an XOR gate:



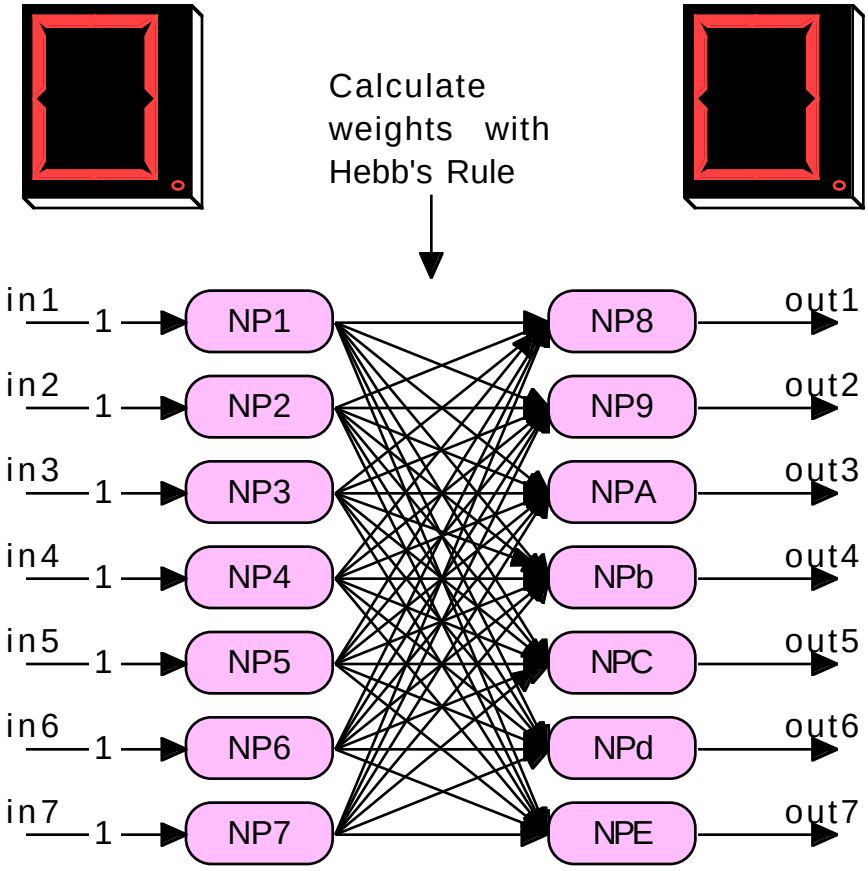
8 A Digit Recognizer

This ANN design is a little more complex with seven inputs, seven outputs and fourteen NPs. The inputs and outputs are to the two seven segment displays on the UP1 board. The goal of the ANN design is to recognize the digits zero, one and two, even if there are extra segments on or off. The user interface uses all eight dip switches to deal with these extra inputs and outputs and still display the NP final sums.

8.1 Design

Using the same ANN design procedure described previously [1], we see that for seven inputs and seven

outputs, we will need fourteen NPs connected in a feed forward network:



The numbering of the NPs past NP9 is due to the limitations of the seven segment display for displaying letters of the alphabet or more than one digit. These are the hexadecimal digits that are displayed in the monitor mode.

8.2 Weight Calculation

The calculation of the weights is done by using supervised Hebbian learning as described in [1].

8.2.1 Hebbian Learning

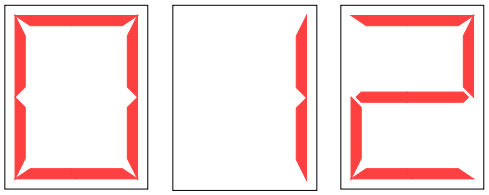
Hebbian learning uses the inputs and outputs to directly calculate the weights. The inputs are basically, multiplied by the outputs and the resultant values are used as the weight values. Since we are implementing autoassociative memory, our outputs are the same as our inputs. The weights are calculated by the formula:

$$W_Y = P \times P^+ \qquad \text{(EQ 4)}$$

where P is the matrix of all the input patterns and P^+ is the pseudoinverse of P. Since the inputs are not orthogonal, we have used the pseudoinverse instead of the transpose.

8.2.2 Input Patterns

The input patterns for the digits 0, 1 and 2 are:



Translating these patterns into a matrix of 1 and -1, where -1 is used to represent an OFF segment and 1 is used to present an ON segment, we have:

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & 1 & 1 & -1 & 1 \end{bmatrix}^T \quad \text{(EQ 5)}$$

The input matrix is presented transposed to put it into a horizontal form. To calculate the weights, MATLAB is used as it has extensive support for matrices and matrix functions. The weight matrix was calculated and converted from floating point to integers with the following MATLAB program:

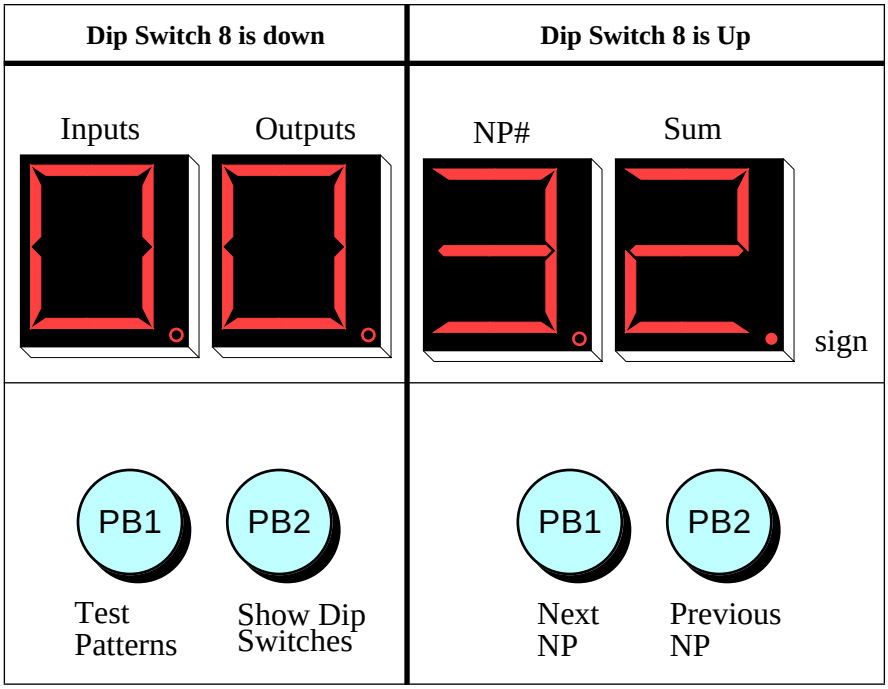
```
>> W=P*pinv(P)

W =

    0.2941    0.1176   -0.0588    0.2941    0.2941    0.1176    0.0588
    0.1176    0.6471    0.1765    0.1176    0.1176   -0.3529   -0.1765
   -0.0588    0.1765    0.4118   -0.0588   -0.0588    0.1765   -0.4118
    0.2941    0.1176   -0.0588    0.2941    0.2941    0.1176    0.0588
    0.2941    0.1176   -0.0588    0.2941    0.2941    0.1176    0.0588
    0.1176   -0.3529    0.1765    0.1176    0.1176    0.6471   -0.1765
    0.0588   -0.1765   -0.4118    0.0588    0.0588   -0.1765    0.4118

>> round(20*W)

ans =

     6     2    -1     6     6     2     1
     2    13     4     2     2    -7    -4
    -1     4     8    -1    -1     4    -8
     6     2    -1     6     6     2     1
     6     2    -1     6     6     2     1
     2    -7     4     2     2    13    -4
     1    -4    -8     1     1    -4     8
```

Smaller weights could be tried to find the optimal balance between precision and accuracy. This can be done with a simulator and a scripting interface.

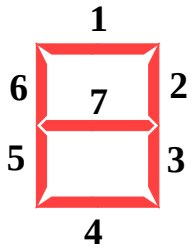
8.3 User Interface

The two seven segment displays are used to display either the inputs and outputs or the NP number and

it’s final sum as in the first ANN UI. Dip switch eight controls the display mode. If it is up, then the



sums are available with the left and right push buttons incrementing and decrementing the NP number. When it is down, the inputs are the left digit and the outputs are the right digit. In this mode, the left push button cycles through eleven different test patterns for the inputs. While the right push button transfers whatever is on the first seven dip switches to the inputs. The mapping is as follows:



There are eleven test patterns. They include the digits zero, one and two which should be recognized and then the top halves of the digits followed by a few of the one hundred and twenty-two possible other patterns. As a comparison, some test subjects were asked to try and recognize the test patterns as well. In general the test subjects and the ANN design were in agreement.

8.4 Test

Once the ANN was designed, it was simulated in Timbre [1]. The output of the simulation was compared to the output of the Seven NPN to verify proper operation. Eight of the eleven patterns were

Inputs											
Outputs											

recognized as a digit while three patterns weren't recognized as anything and produced output similar to the input. This behaviour was duplicated with the implemented NPN. As done in the previous work [1], the precision of the weights could be reduced to try and improve the recognition while at the same time reducing the hardware requirements of the circuit.

8.5 Extending the Digit Recognizer

It is quite easy to extend the digit recognizer to more complex digit displays by increasing the number of NPs to match the inputs. An ANN design for a five by six pixel display which uses sixty NPs with a weight precision of minus one to one, was compiled to compare the size of the resultant NPN to the previous designs. Using the single NP architecture, the NPN took up 67% of the LCs and 25% of the memory bits in a FLEX 10K20 part.

9 Metrics

These measurements were made on the different designs and architectures and are tabled here for reference:

Architecture	Entity	number of NPs	LCs	LCs/NPs	flip flops	ffs/NPs	memory bits
multiple NPs	XOR NPN	5	76	15.2	38	7.6	0
	Seven NPN	14	284	20.3	154	11	0
	Sixty NPN	60	[a]				
single NP	XOR NPN	5	544	108.8	177	35.4	1168
	Seven NPN	14	645	46	244	17.4	3072
	Sixty NPN	60	783	13	264	4.4	8344
User Interface	XOR UI	5	124	n/a	37	n/a	0
	Seven UI	14	155		33		0

a. figures unavailable since the compiler could not compile the design. Strange errors.

As can be expected, in the single NP implementation, the ratio of LCs to NPs goes down as the number of NPs is increased. And for the multiple NP implementation it increases. The same is true for the number of flip flops per NP. With more time, the size trade-off between the implementations could be further explored and better understood.

10 Summary

This project explored two different ANN designs and implemented them in two different ways using the NP model. The results are encouraging and create many new areas to explore. The two different implementations are just two of many possible architectures one can consider when optimizing for:

- Extensibility - how easy can the NPN be extended?
- Reconfigurability - how easy can the NPN be changed?
- Response Time - what are the requirements for real time?
- Propagation Delay - what are the constraints for delay?
- Resource Requirements - what resources are available for the ANN implementation?
- Simulation - is simulation important?

11 What's Next

Now that the ideas have been taken all the way down to the silicon, it is time to look at the process. What ways can it be improved?

11.1 Automation

Automation is key in improving all parts of the process. One can picture a black box where an ANN design is fed in one side and a FLEX sram object file pops out the other side. It would take care of:

- Design Transfer from ANN Parameters
- Topology
- Precision
- Inputs
- Outputs
- Graphical

11.2 Optimization

Optimization is also key in making the technology more useful. This can be directly measured in the cost to implement the ANN design in terms of logic cells and flip flops.

11.2.1 One NP per String

As explored in the previous document (simulation documentation), the topology of a network can be implemented with one NP per node or one per string when using multiple NPs. The one per string approach is compelling because propagation delay and response time can be the same. In the one per string approach, one NP is all the NPs that are serially connected to it. At most, there will be S NPs where S is the maximum number of parallel operating peer NPs.

11.2.2 Statistical Addition

The Single NP implementation uses Statistical Addition [3] to save gates. This could also be used in the multiple NP implementation. It would be interesting to see the timing and speed differences with this change.

11.3 Schedule

There are still a few parts of the project which could be completed and are areas for further research

and exploration. This table is a collection of some of the tasks:

Task
1. simplify using generics and generates for NP and NPN
2. replace adder in NP with statistical adder
3. automate the generation of the VHDL code from the GM
4. add multi-level optimization
5. use dynamic weights
6. incorporate backprop net for dynamic learning

12 References

1. EE563 Project Document: "Using A Neuroprocessor Model for Describing Artificial Neural Nets", http://www.compusmart.ab.ca/rc/Papers/NeuroprocessorModeling.pdf
2. EE602 Project Document: "A Stack Processor: Synthesis", http://www.compusmart.ab.ca/rc/Papers/spsynthesis.pdf
3. EE635 Project Document: "A Writable Computer", http://www.compusmart.ab.ca/rc/Papers/writablecomputer.pdf
4. Paper "Simulating a Neuroprocessor", http://www.compusmart.ab.ca/rc/Papers/Simulating a Neuroprocessor.pdf

Appendices

A VHDL Code for Multiple NP Design

A.1 Xor NPN

```
-- Difference detector NP network  Rob Chapman  Mar 7, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_npn is
  port ( in1, in2, clock : in std_logic;
         output          : out std_logic;
         outputs         : buffer std_logic_vector(14 downto 0));
end entity;

architecture structure of xor_npn is

  component xor_np
    port(clock  : in std_logic;
         inputs : in std_logic_vector(5 downto 0);
         weight : in integer range -1 to 1;
         choice : out integer range 0 to 7;
         output : out integer range -2 to 2);
  end component;

  component xor_gm
    port(np1, np2, np3, np4, np5 : in  integer range 0 to 7;
         w1, w2, w3, w4, w5      : out integer range -1 to 1);
  end component;

  signal cycle135, cycle244 : natural range 0 to 2;

  signal input1, input2 : std_logic_vector(5 downto 0); -- NP inputs

  signal out1, out2, out3, out4, out5 : std_logic_vector(2 downto 0);

  signal choice135, choice244 : natural range 0 to 5;
  signal w1, w2, w3, w4, w5, weight135, weight244 : integer range -1 to 1;

  signal out135, out244: integer range -2 to 2; -- interconnects

begin

  -- cycle for summing one input and 5 outputs is 6 clocks long
  process
  begin
    wait until clock = '1';

    if  choice135 = 5  then  -- np1 has reached final sum
      case cycle135 is       -- on to next cycle
        when 0      => cycle135 <= 1;
        when 1      => cycle135 <= 2;
        when others => cycle135 <= 0;
      end case;
    end if;

    if  choice244 = 5  then  -- np2 has reached final sum
```

```
    case cycle244 is       -- next cycle
      when 0      => cycle244 <= 1;
      when 1      => cycle244 <= 2;
      when others => cycle244 <= 0;
    end case;
  end if;

end process;

-- create input vectors for NPs
with cycle135 select               -- get current input vector for NP1
  input1 <= "00000"&in1               when 0,
            "000"&out2(2)&out1(2)&"0" when 1,
            "0"&out4(2)&out3(2)&"010" when others;

with cycle244 select               -- get current input vector for NP2
  input2 <= "00000"&in2               when 0,
            "000"&out2(2)&out1(2)&"0" when others;

with cycle135 select               -- select GM interface for NP1
  weight135 <= w1 when 0,
               w3 when 1,
               w5 when others;


with cycle244 select               -- select GM interface for NP2
  weight244 <= w2 when 0,
               w4 when others;

-- transfer output of NP1 and NP2 to the current NP outputs
--  Note: done on first clock of beginning of next summation.  This
--        is ok because the first clock is used by the NPs to
--        check the input.  By the second clock, the outputs are
--        ready to be used as inputs to the NPs.
begin
  wait until clock = '1';
  if  choice135 = 0  then          -- for NP1
    case cycle135 is
      when 0 =>
        out5 <= conv_std_logic_vector(out135,3);
      when 1 =>
        out1 <= conv_std_logic_vector(out135,3);
      when others =>
        out3 <= conv_std_logic_vector(out135,3);
    end case;
  end if;

  if  choice244 = 0  then           -- for NP2
    if  cycle244 = 1  then
      out2 <= conv_std_logic_vector(out244,3);
    else
      out4 <= conv_std_logic_vector(out244,3);
    end if;
  end if;

end process;

-- map in the weights for all
Grand_Matrix : xor_gm
  port map (np1 => choice135, np2 => choice244, np3 => choice135,
            np4 => choice244, np5 => choice135,
            w1 => w1,   w2 => w2,  w3 => w3,  w4 => w4,  w5 => w5);

-- connect up the neuroprocessors
NP1 : xor_np -- neuroprocessor 1,3,5
```

```
    port map (clock => clock,
              inputs => input1,
              weight => weight135,
              choice => choice135,
              output => out135);

  NP2 : xor_np -- neuroprocessor 2,4
    port map (clock => clock,
              inputs => input2,
              weight => weight244,
              choice => choice244,
              output => out244);

  -- assign output sums
  outputs( 2 downto 0)  <= out1;
  outputs( 5 downto 3)  <= out2;
  outputs( 8 downto 6)  <= out3;
  outputs(11 downto 9)  <= out4;
  outputs(14 downto 12) <= out5;
  output <= out5(2);

end structure;

-- Neuroprocessor for "Difference Detector" NPN  Rob Chapman  Mar 10, 1998

 -- This file contains a behavioural description of a neuroprocessor for
 -- the difference dectector NPN.

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_np is
  port(clock  : in std_logic;
       inputs : in std_logic_vector(5 downto 0);
       weight : in integer range -1 to 1;  -- interface to GM
       choice : out integer range 0 to 7;  -- ditto
       output : out integer range -2 to 2);
end entity xor_np;

architecture behaviour of xor_np is
  signal index : natural range 0 to 5;
  signal sum : integer range -2 to 2;
  signal next_sum : integer range -1 to 1;

  begin

    choice <= index;

    with inputs(index) select
      next_sum  <=  weight  when '0',
   -weight  when others;

    process
    begin
      wait until clock = '1';

      if  index = 5  then
        index  <= 0;
        output <= sum;
        sum    <= next_sum;
      else
        index  <= index + 1;
        sum    <= sum + next_sum;
```

```
      end if;

    end process;

end architecture behaviour;

-- The Grand Matrix for the "detect a difference" NPN  Rob Chapman  Mar 9, 1998

 -- This file contains all the weights for the detect a difference
 -- neuroprocessor network.  A truth table is used to represent the
 -- values.  The interface uses one input vector and one output vector
 -- which is partitioned into individual weight vectors for each NP.

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_gm is
  port(np1, np2, np3, np4, np5 : in  integer range 0 to 7;
       w1, w2, w3, w4, w5      : out integer range -1 to 1);
end entity xor_gm;

architecture truth_table of xor_gm is
  begin

    -- the individual weights
    process(np1)
    begin
      case np1 is
        when 0 => w1 <= 1;
        when others => w1 <= 0;
      end case;
    end process;

    process(np2)
    begin
      case np2 is
        when 0 => w2 <= 1;
        when others => w2 <= 0;
      end case;
    end process;

    process(np3)
    begin
      case np3 is
        when 1 => w3 <= 1;
        when 2 => w3 <= -1;
        when others => w3 <= 0;
      end case;
    end process;

    process(np4)
    begin
      case np4 is
        when 1 => w4 <= -1;
        when 2 => w4 <= 1;
        when others => w4 <= 0;
      end case;
    end process;

    process(np5)
    begin
      case np5 is
        when 0 => w5 <= -1;
```

```
          when 3 => w5 <= 1;
          when 4 => w5 <= 1;
          when others => w5 <= 0;
        end case;
      end process;

  end architecture truth_table;
```

A.2 Seven NPN

```
  -- Network parameters  Rob Chapman  Mar 27, 1998

  Library IEEE;
  use IEEE.STD_Logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_signed.all;

  -- parameterize the network
  package params is

    constant N          : positive := 14;  -- number of neuroprocessors
    constant sumbits    : positive := 6;   -- precision of the sums
    constant weightbits : positive := 6;   -- precision of the weights

    subtype achoice is natural range 0 to N;
    subtype aweight is std_logic_vector(weightbits-1 downto 0);
    subtype asum    is std_logic_vector(sumbits-1 downto 0);
    subtype iobus   is std_logic_vector(0 to N);

    type choices is array (natural range N downto 0) of achoice;
    type weights is array (natural range N downto 0) of aweight;
    type sums    is array (natural range N downto 0) of asum;

  end package;

  -- Digit recognizer NP network  Rob Chapman  Mar 7, 1998

  Library IEEE;
  use IEEE.STD_Logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_signed.all;

  use work.params.all;

  entity sevennpn is
    port ( clock   : in  std_logic;
           input   : in  iobus;
           output  : buffer iobus;
           outputs : buffer sums);
  end entity;

  architecture structure of sevennpn is

    component sevennp
      port(clock  : in  std_logic;
           input  : in  std_logic; -- input for NP
           inputs : in  iobus;     -- inputs from other NPs
           weight : in  aweight;   -- interface to GM
           choice : out achoice;   -- ditto
           output : out asum);     -- final sum
    end component;

    component sevengm
      port(choice : in  choices;
```

```
         weight : out weights);
  end component;

  signal weight : weights; -- weight bus
  signal choice : choices; -- choice bus

begin

  -- bug2 avoidance for Altera compiler
  choice(0) <= 15;
  outputs(0) <= "000000";
  output(0) <= '0';

  -- map in the weights for all
  Grand_Matrix : sevengm
    port map (choice => choice, weight => weight);

  network:
    for np in 1 to N generate
    begin
      node : sevennp
        port map (clock  => clock,
                  input  => input(np),
                  inputs => output,
                  weight => weight(np),
                  choice => choice(np),
                  output => outputs(np));
    end generate;

  -- connect up output bits to sign bit of final sums
  the_outputs:
    for np in 1 to N generate
    begin
      process(outputs(np))
        variable tmp : asum;
      begin
        tmp := outputs(np);
        output(np) <= tmp(sumbits-1);
      end process;
    end generate;

end structure;

-- Neuroprocessor  Rob Chapman  Mar 10, 1998

 -- This file contains a behavioural description of a neuroprocessor

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

use work.params.all;

entity sevennp is
  port(clock  : in std_logic;
       input  : in std_logic;
       inputs : in iobus;
       weight : in  aweight;  -- interface to GM
       choice : out achoice;  -- ditto
       output : out asum);
end entity;

architecture behaviour of sevennp is
  signal index     : achoice;
```

```
    signal sum       : asum;
    signal next_sum  : aweight;
    signal np_input  : std_logic_vector(0 to N);

    begin

      choice <= index;

      np_input(0) <= input;
      np_input(1 to N)  <= inputs;

      with np_input(index) select
        next_sum  <=  weight when '0',
                     -weight when others;

      process
      begin
        wait until clock = '1';

        if  index = N  then
          index  <= 0;
          output <= sum;
          sum    <= next_sum;
        else
          index  <= index + 1;
          sum    <= sum + next_sum;
        end if;

      end process;

  end architecture behaviour;

  -- The Grand Matrix for NPN  Rob Chapman  Mar 9, 1998

   -- This file contains all the weights for the  neuroprocessor network.
   --  A truth table is used to represent the values.  The interface uses
   -- one input vector and one output vector which is partitioned int
   -- individual weight vectors for each NP.

  Library IEEE;
  use IEEE.STD_Logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_signed.all;

  use work.params.all;

  entity sevengm is
    port(choice : in choices;
         weight : out weights);
  end entity;

  architecture truth_table of sevengm is

    -- Weight values used
    constant zero     : aweight := "000000"; -- conv_std_logic_vector(0,weight-
  bits);
    constant one     : aweight := "000001"; -- conv_std_logic_vector(1,weightbits);
    constant two      : aweight := "000010";
    constant three    : aweight := "000011";
    constant four     : aweight := "000100";
    constant five     : aweight := "000101";
    constant six      : aweight := "000110";
    constant seven    : aweight := "000111";
    constant eight    : aweight := "001000";
    constant nine     : aweight := "001001";
```

```
  constant ten      : aweight := "001010";
  constant eleven   : aweight := "001011";
  constant twelve   : aweight := "001100";
  constant thirteen : aweight := "001101";
  constant minus1   : aweight := "111111"; -- conv_std_logic_vector(-1,weight-
bits);
  constant minus2   : aweight := "111110";
  constant minus3   : aweight := "111101";
  constant minus4   : aweight := "111100";
  constant minus5   : aweight := "111011";
  constant minus6   : aweight := "111010";
  constant minus7   : aweight := "111001";
  constant minus8   : aweight := "111000";

  begin

    -- bug2 avoidance for Altera compiler
    weight(0) <= zero;

    -- the input weights
    in_weights:
      for index in 1 to 7 generate
        process(choice(index))
        begin
          case choice(index) is
            when 0      => weight(index) <= one;
            when others => weight(index) <= zero;
          end case;
        end process;
      end generate;

    -- second layer weights
    process(choice(8))
    begin
      case choice(8) is
        when 1 => weight(8) <= six;
        when 2 => weight(8) <= two;
        when 3 => weight(8) <= minus1;
        when 4 => weight(8) <= six;
        when 5 => weight(8) <= six;
        when 6 => weight(8) <= two;
        when 7 => weight(8) <= one;
        when others => weight(8) <= zero;
      end case;
    end process;

    process(choice(9))
    begin
      case choice(9) is
        when 1 => weight(9) <= two;
        when 2 => weight(9) <= thirteen;
        when 3 => weight(9) <= four;
        when 4 => weight(9) <= two;
        when 5 => weight(9) <= two;
        when 6 => weight(9) <= minus7;
        when 7 => weight(9) <= minus4;
        when others => weight(9) <= zero;
      end case;
    end process;

    process(choice(10))
    begin
      case choice(10) is
        when 1 => weight(10) <= minus1;
        when 2 => weight(10) <= four;
```

```
    when 3 => weight(10) <= eight;
    when 4 => weight(10) <= minus1;
    when 5 => weight(10) <= minus1;
    when 6 => weight(10) <= four;
    when 7 => weight(10) <= minus8;
    when others => weight(10) <= zero;
  end case;
end process;

process(choice(11))
begin
  case choice(11) is
    when 1 => weight(11) <= six;
    when 2 => weight(11) <= two;
    when 3 => weight(11) <= minus1;
    when 4 => weight(11) <= six;
    when 5 => weight(11) <= six;
    when 6 => weight(11) <= two;
    when 7 => weight(11) <= one;
    when others => weight(11) <= zero;
  end case;
end process;

process(choice(12))
begin
  case choice(12) is
    when 1 => weight(12) <= six;
    when 2 => weight(12) <= two;
    when 3 => weight(12) <= minus1;
    when 4 => weight(12) <= six;
    when 5 => weight(12) <= six;
    when 6 => weight(12) <= two;
    when 7 => weight(12) <= one;
    when others => weight(12) <= zero;
  end case;
end process;

process(choice(13))
begin
  case choice(13) is
    when 1 => weight(13) <= two;
    when 2 => weight(13) <= minus7;
    when 3 => weight(13) <= four;
    when 4 => weight(13) <= two;
    when 5 => weight(13) <= two;
    when 6 => weight(13) <= thirteen;
    when 7 => weight(13) <= minus4;
    when others => weight(13) <= zero;
  end case;
end process;

process(choice(14))
begin
  case choice(14) is
    when 1 => weight(14) <= one;
    when 2 => weight(14) <= minus4;
    when 3 => weight(14) <= minus8;
    when 4 => weight(14) <= one;
    when 5 => weight(14) <= one;
    when 6 => weight(14) <= minus4;
    when 7 => weight(14) <= eight;
    when others => weight(14) <= zero;
  end case;
end process;
```

```
end architecture truth_table;
```

B VHDL Code for Single NP Design

B.1 Seven NPN

```
detect-a-difference circuit:
---------------------------

seven.vhd
    +sevenui.vhd
        +params.vhd
    +seven_npn_sp.vhd
        +seven_npn_defs.vhd
        +params.vhd
        +memory_seven.vhd
            +sevengmnew.mif
        +stack_processor.vhd
            +np_program_seven.mif


-- Seven segment digit recognizer  Rob Chapman  Mar 25, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

use work.params.all;

entity seven is
  port ( clock    : in  std_logic;
         switches : in  std_logic_vector(1 to 10);
         led      : out std_logic_vector(1 to 16));
end;

architecture structure of seven is

  component seven_npn_sp
    port ( input   : in  iobus;
   clock   : in  std_logic;
           output  : out iobus;
           outputs : out sums);
  end component;

  component sevenui
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10);
           inputs_to_npn : out iobus;
   output_from_npn : in iobus;
           outputs  : in sums;
           led      : out std_logic_vector(1 to 16));
  end component;

  signal pulse, quickpulse        : std_logic; -- for setting network heartbeat
  signal outputs       : sums;
  signal input, output : iobus;


begin

  -- bug2 avoidance for Altera compiler
```

```
--  input(0) <= '0';

  -- set the pulse for the nps; could be event triggered too for low
  -- power and instant response to input change
  HB : process
  begin
    wait until clock = '1';
    if  quickpulse = '0' then
      quickpulse <= '1';
    else
      quickpulse <= '0';
    end if;
  end process;

  right_ventricle : process
  begin
    wait until quickpulse = '1';
    if  pulse = '0' then
      pulse <= '1';
    else
      pulse <= '0';
    end if;
  end process;


  -- connect switches to inputs
--  input(1 to 7) <= switches(3 to 9);
--  off: for i in 8 to N  generate
--         input(i) <= '0';
--       end generate;

  -- connect up components
  ui : sevenui -- user interface
    port map ( clock    => pulse,
               switches => switches,
             inputs_to_npn => input,
   output_from_npn => output,
               outputs  => outputs,
               led      => led);

  npn : seven_npn_sp -- neural network
    port map ( clock   => pulse,
               input   => input,
               output  => output,
               outputs => outputs);

end structure;

library ieee;
use ieee.std_logic_1164.all;

package seven_npn_types is
subtype sp_word is std_logic_vector( 7 downto 0 );
end seven_npn_types;

library ieee;
use ieee.std_logic_1164.all;

use work.seven_npn_types.all;
use work.seven_npn_defs.all;
use work.params.all;

entity seven_npn_sp is

    port ( input : in iobus;
```

```
--reset,
clock : in std_logic;
result : out sp_word;
output : out iobus;
np_sum_array_port : out np_sum_array_type;
outputs : out sums
);
end;

architecture structure of seven_npn_sp is

component stack_processor
generic( data_stack_index_width : integer := 3;
  data_stack_index_depth : integer := 8;
  return_stack_index_width : integer := 3;
  return_stack_index_depth : integer := 8
);
port (
reset, clock : in std_logic;
memdata_in : in std_logic_vector( 7 downto 0);
memdata_out : out std_logic_vector ( 7 downto 0);
address_out : out std_logic_vector( 7 downto 0);
fetch_store : out std_logic;
topout : out std_logic_vector( 7 downto 0);
pcout : out std_logic_vector( 7 downto 0)
);
end component;

component memory_seven
  port (
ain,
din:  in std_logic_vector( 7 downto 0);

clock,
fetch_store:  in std_logic;

 input_vector: in npn_io_type;

        dout,
result: out std_logic_vector( 7 downto 0);

output: out npn_io_type;

--out1, out2, out3, out4, out5 : out std_logic_vector(2 downto 0)
np_sum_array: out np_sum_array_type
);
end component;

signal  address,
data_tomem,
data_tosp,
periph_tomem,
program_counter,
periph_toout: sp_word;

signal  fetch_store,
reset: std_logic;

signal  np_sum_array_buffer : np_sum_array_type;

--signal result : sp_word;

signal output_buffer, input_buffer : npn_io_type;

begin
```

```
reset <= '0';
np_sum_array_port <= np_sum_array_buffer;

--output <= output_buffer( 0 to output'high );

--outputs <= np_sum_array_buffer( 0 to outputs'high );

--outputs(0) <= "000";

connect_sums : for i in 1 to 14 generate
outputs(i) <= np_sum_array_buffer(16-i);
output(i) <= output_buffer(i-1);
--input_buffer(14-i) <= input(i+2);
end generate;

--output <= "101010101010101";

output(0) <= '0';
outputs(0) <= "000";
--outputs(1) <= "001";
--outputs(2) <= "010";
--outputs(3) <= "011";
--outputs(4) <= "100";
--outputs(5) <= "101";
--outputs(6) <= "110";
--outputs(7) <= "111";
--outputs(8) <= "000";
--outputs(9) <= "001";
--outputs(10) <= "010";
--outputs(11) <= "011";
--outputs(12) <= "100";
--outputs(13) <= "101";
--outputs(14) <= "110";

--input_buffer(15) <= '0';

--input_buffer(0) <= '0';
input_buffer <= (input(1 to 14) & "00" );

my_memory : memory_seven
port map (
ain => address,
din => data_tomem,
input_vector => input_buffer,
clock => clock,
fetch_store => fetch_store,
dout => data_tosp,
result => result,
output => output_buffer,
np_sum_array => np_sum_array_buffer
);

my_sp : stack_processor
generic map (
data_stack_index_width => 3,
data_stack_index_depth => 8,
return_stack_index_width => 2,
return_stack_index_depth => 4
)
port map (
reset => reset,
clock => clock,
memdata_in => data_tosp,
memdata_out => data_tomem,
```

```
address_out => address,
fetch_store => fetch_store
);

end structure;


library ieee;
use ieee.std_logic_1164.all;

package seven_npn_defs is

subtype np_sum_type is std_logic_vector(2 downto 0);
type np_sum_array_type is array(0 to 15) of np_sum_type;

subtype npn_io_type is std_logic_vector(0 to 15);

end seven_npn_defs;

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


library lpm;
use lpm.lpm_components.all;
use work.seven_npn_defs.all;

entity Memory_seven is
  port (
ain,
din:  in std_logic_vector( 7 downto 0);

clock,
fetch_store:  in std_logic;

 input_vector: in npn_io_type;

        dout,
result: out std_logic_vector( 7 downto 0);

output: out npn_io_type;

--out1, out2, out3, out4, out5 : out std_logic_vector(2 downto 0)
np_sum_array: out np_sum_array_type
);
end;

architecture behaviour2 of Memory_seven is

subtype Cell is std_logic_vector(7 downto 0);

type decode_state_type is ( ram,  periph, io );
signal decode_state, next_decode_state : decode_state_type;

type io_state_type is ( input_port, np_outputs );
signal io_state, next_io_state : io_state_type;

type periph_state_type is ( weight_ptr, weight_register, result_register,
output_sum_register );
signal periph_state, next_periph_state : periph_state_type;

signal program_dout,
   periph_dout,
```

```
   weight_dout,
   weight_ptr_dout,
   input_dout    : Cell;

--signal zero_2bit : std_logic_vector ( 1 downto 0 );

signal weight_dout_8bit, zero_8bit : std_logic_vector( 7 downto 0 );

signal np_output, din_signbit : std_logic_vector( 0 downto 0 );

signal clock_inv,
zero,
input_bit,
input_port_bit : std_logic;

signal io_select, input_we, np_output_we, ram_we, result_we, np_sum_we,
weight_ptr_we, np_output_bit : std_logic;

signal din_3bit : std_logic_vector( 2 downto 0 );

signal weight_ptr_dout_6bit : std_logic_vector( 5 downto 0 );
signal np_output_array : npn_io_type;
signal np_sum_array_sig : np_sum_array_type;

signal address_7bit : std_logic_vector( 6 downto 0 );

begin


-- program ram spans memory locations 0 to $7f ( 7 bits )
program_ram : component LPM_RAM_DQ
generic map (
lpm_width => Cell'length,
lpm_widthad => 7,
lpm_file => "np_program_seven.mif"
)
port map (
inclock => clock,
outclock => clock_inv,
we  => ram_we,
address => address_7bit,
data => din,
q => program_dout
);

-- weight rom spans memory locations $80 to $bf ( 6 bits )
weight_rom : component LPM_RAM_DQ
generic map (
lpm_width => 8,
lpm_widthad => 8,
lpm_file => "sevengmnew.mif"
)
port map (
inclock => clock,
outclock => clock_inv,
we  => zero,
address => weight_ptr_dout,
data => zero_8bit,
q => weight_dout_8bit
);

-- this is just one-bit ram for the outputs of the NPs
-- found in memory locations 64 to 64+15=79 (16 locations)
-- however io memory space spans $c0 to $ff
--np_output_ram : component LPM_RAM_DQ
```

```
--generic map (
--lpm_width => 1,
--lpm_widthad => 4
--)
--port map (
--inclock => clock,
--outclock => clock_inv,
--we  => input_we,
--address => address_4bit,
--data => din_signbit,
--q => np_output
--);

-------------------------------------------------
-- address_decode
--
-- do address decoding
-------------------------------------------------
address_decoder : process( ain )
begin
    case ain(7 downto 6) is
when "00" | "01" =>-- select program ram
next_decode_state <= ram;

when "10"  => -- select weight rom
next_decode_state <= io;

when "11" => -- select input/output ram
next_decode_state <= periph;

when others =>  -- shouldn't happen...
next_decode_state <= ram;
end case;

--
-- take care of write enable signals
--
case next_decode_state is
when ram =>
ram_we <= fetch_store;
when others =>
ram_we <= '0';
end case;

-------------------------------------------------------------
-- io block decoder
case ain(4) is
when '0' =>
next_io_state <= input_port;
when others =>
next_io_state <= np_outputs;
end case;

if next_decode_state = io and next_io_state = np_outputs then
np_output_we <= fetch_store;
else
np_output_we <= '0';
end if;

-------------------------------------------------------------
-- peripheral block decoder:  output sums located in first 7 locations.
-- next two locations contain weight pointer and weight peripheral
case ain( 5 downto 0) is
when "111101" =>
next_periph_state <= weight_ptr;
```

```
when "111110" =>
next_periph_state <= weight_register;
when "111111" =>
next_periph_state <= result_register;
when others =>
next_periph_state <= output_sum_register;
end case;

if next_decode_state = periph then
if next_periph_state = result_register then
result_we <= fetch_store;
else
result_we <= '0';
end if;

if next_periph_state = output_sum_register then
np_sum_we <= fetch_store;
else
np_sum_we <= '0';
end if;

if next_periph_state = weight_ptr then
weight_ptr_we <= fetch_store;
else
weight_ptr_we <= '0';
end if;
else
result_we <= '0';
np_sum_we <= '0';
weight_ptr_we <= '0';
end if;

end process;

change_decoder_states : process(clock)
begin
if clock'event and clock='1' then
decode_state <= next_decode_state;

io_state <= next_io_state;

periph_state <= next_periph_state;
end if;
end process;

-------------------------------------------------------
-- state output logic

main_output_logic : process( decode_state, program_dout, weight_dout, input_dout
)
begin
case decode_state is
when ram =>
dout <= program_dout;

when periph =>
dout <= periph_dout;

when io =>-- inputs, and also output latch
dout <= input_dout;
end case;
end process;

io_output_logic : process( io_state, input_port_bit, np_output_bit )
begin
```

```
case io_state is
when input_port =>
input_bit <= input_port_bit;
when np_outputs =>
input_bit <= np_output_bit;
end case;
end process;

periph_output_logic : process( periph_state, weight_ptr_dout, weight_dout)
begin
case periph_state is
when weight_ptr =>
periph_dout <= weight_ptr_dout;
when weight_register =>
periph_dout <= weight_dout;
when result_register =>
periph_dout <= x"00";
when output_sum_register =>
periph_dout <= x"00";
end case;
end process;

-------------------------------------------------------------------------

assorted_registers : process( clock )

begin
if clock'event and clock='1' then
-- select which np output is fetched
np_output_bit <= np_output_array(conv_integer(ain(3 downto 0)));

-- select which input depending on address
input_port_bit <= input_vector(conv_integer(ain(3 downto 0)));

-- write the weight ptr
if weight_ptr_we = '1' then
weight_ptr_dout <= din;
end if;

-- write the np sum -- this uses the np_output_we signal because the sums
-- are mapped to the same addresses as the outputs.  The outputs just use the
-- sign bit of the sum.
if np_output_we = '1' then
np_sum_array_sig(conv_integer(ain(3 downto 0))) <= din(2 downto 0);
end if;

-- write the np output
if np_output_we = '1' then
np_output_array(conv_integer(ain(3 downto 0))) <= din(7);
end if;

-- take care of the "result" register
if fetch_store = '1' then
if ain = x"ff" then
result <= din;
end if;
end if;

end if;
end process;


-------------------------------------------------------------------------

output <= np_output_array;
```

```
np_sum_array <= np_sum_array_sig;

input_dout(0) <= input_bit;
input_dout(1) <= '0';
input_dout(2) <= '0';
input_dout(3) <= '0';
input_dout(4) <= '0';
input_dout(5) <= '0';
input_dout(6) <= '0';
input_dout(7) <= '0';


--
-- assign ain to the different address signals
--
--address_4bit <= ain( 3 downto 0 );
--address_6bit <= ain( 5 downto 0 );
address_7bit <= ain( 6 downto 0 );

-- put the LSB of din on a separate signal
din_signbit(0) <= din(7);
din_3bit( 2 downto 0 ) <= din(2 downto 0);

weight_ptr_dout_6bit <= weight_ptr_dout( 5 downto 0 );

-- assign the 2bit weight to the full weight

weight_dout(7 downto 0) <= weight_dout_8bit(7 downto 0);
-- sign-extend the weights
--assign_weights : for i in 4 to 7 generate
---weight_dout(i) <= weight_dout_4bit(3);
--end generate;

clock_inv <= not clock;
zero <= '0';
zero_8bit <= "00000000";


end behaviour2;

-- Grand Matrix Weight .mif file auto-generated by genmatrix
-- copyright 1998
--

WIDTH = 8;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN

00: 01;
01: 00;
02: 00;
03: 00;
04: 00;
05: 00;
06: 00;
07: 00;
08: 00;
09: 00;
0a: 00;
0b: 00;
0c: 00;
```

```
0d: 00;
0e: 00;
0f: 01;
10: 00;
11: 00;
12: 00;
13: 00;
14: 00;
15: 00;
16: 00;
17: 00;
18: 00;
19: 00;
1a: 00;
1b: 00;
1c: 00;
1d: 00;
1e: 01;
1f: 00;
20: 00;
21: 00;
22: 00;
23: 00;
24: 00;
25: 00;
26: 00;
27: 00;
28: 00;
29: 00;
2a: 00;
2b: 00;
2c: 00;
2d: 01;
2e: 00;
2f: 00;
30: 00;
31: 00;
32: 00;
33: 00;
34: 00;
35: 00;
36: 00;
37: 00;
38: 00;
39: 00;
3a: 00;
3b: 00;
3c: 01;
3d: 00;
3e: 00;
3f: 00;
40: 00;
41: 00;
42: 00;
43: 00;
44: 00;
45: 00;
46: 00;
47: 00;
48: 00;
49: 00;
4a: 00;
4b: 01;
4c: 00;
4d: 00;
```

```
4e: 00;
4f: 00;
50: 00;
51: 00;
52: 00;
53: 00;
54: 00;
55: 00;
56: 00;
57: 00;
58: 00;
59: 00;
5a: 01;
5b: 00;
5c: 00;
5d: 00;
5e: 00;
5f: 00;
60: 00;
61: 00;
62: 00;
63: 00;
64: 00;
65: 00;
66: 00;
67: 00;
68: 00;
69: 00;
6a: 06;
6b: 02;
6c: ff;
6d: 06;
6e: 06;
6f: 02;
70: 01;
71: 00;
72: 00;
73: 00;
74: 00;
75: 00;
76: 00;
77: 00;
78: 00;
79: 02;
7a: 0d;
7b: 04;
7c: 02;
7d: 02;
7e: f9;
7f: fc;
80: 00;
81: 00;
82: 00;
83: 00;
84: 00;
85: 00;
86: 00;
87: 00;
88: ff;
89: 04;
8a: 08;
8b: ff;
8c: ff;
8d: 04;
8e: f8;
```

```
8f: 00;
90: 00;
91: 00;
92: 00;
93: 00;
94: 00;
95: 00;
96: 00;
97: 06;
98: 02;
99: ff;
9a: 06;
9b: 06;
9c: 02;
9d: 01;
9e: 00;
9f: 00;
a0: 00;
a1: 00;
a2: 00;
a3: 00;
a4: 00;
a5: 00;
a6: 06;
a7: 02;
a8: ff;
a9: 06;
aa: 06;
ab: 02;
ac: 01;
ad: 00;
ae: 00;
af: 00;
b0: 00;
b1: 00;
b2: 00;
b3: 00;
b4: 00;
b5: 02;
b6: f9;
b7: 04;
b8: 02;
b9: 02;
ba: 0d;
bb: fc;
bc: 00;
bd: 00;
be: 00;
bf: 00;
c0: 00;
c1: 00;
c2: 00;
c3: 00;
c4: 01;
c5: fc;
c6: f8;
c7: 01;
c8: 01;
c9: fc;
ca: 08;
cb: 00;
cc: 00;
cd: 00;
ce: 00;
cf: 00;
```

```
d0: 00;
d1: 00;

END;

library IEEE;
use IEEE.STD_Logic_1164.all;

package stack_processor is
component stack_processor
generic( data_stack_index_width : integer := 3;
  data_stack_index_depth : integer := 8;
  return_stack_index_width : integer := 2;
  return_stack_index_depth : integer := 4
);
port (
reset, clock : in std_logic;
memdata_in : in std_logic_vector( 7 downto 0);
memdata_out : out std_logic_vector ( 7 downto 0);
address_out : out std_logic_vector( 7 downto 0);
fetch_store : out std_logic;
topout : out std_logic_vector( 7 downto 0);
pcout : out std_logic_vector( 7 downto 0)
);
end component;
end stack_processor;

library IEEE;
use IEEE.STD_Logic_1164.all;

use work.sp_defs.all;

entity stack_processor is
generic ( data_stack_index_width : integer := 3;
  data_stack_index_depth : integer := 8;
  return_stack_index_width : integer := 2;
  return_stack_index_depth : integer := 4
);
port (
reset, clock : in std_logic;
memdata_in : in std_logic_vector( 7 downto 0);
memdata_out : out std_logic_vector ( 7 downto 0);
address_out : out std_logic_vector( 7 downto 0);
fetch_store : out std_logic;
topout : out std_logic_vector( 7 downto 0);
pcout : out std_logic_vector( 7 downto 0)
);
end;

architecture structure of stack_processor is

constant word_width : positive := 8;
subtype Cell is std_logic_vector( word_width-1 downto 0);

  signal P, PP, R, M, T, F, U, D : Cell;  -- data paths
  signal nz : std_logic; -- not zero signal from top  to instruction
  signal mcontrol, pcontrol, tcontrol : std_logic_vector( 1 downto 0 );
  signal rcontrol, dcontrol : std_logic_vector( 2 downto 0 );
  signal fcontrol : std_logic_vector( 2 downto 0 );
  signal sp_opcode : Opcode;

  component instruction
    port ( iin   : in Cell;
           reset, nz, clock : in std_logic;
           iout  : out Opcode );
```

```
  end component;
  for all : instruction use entity work.instruction(behaviour2);

  component FunctionUnit
  generic ( width : positive :=4 );
  port ( selfunction :  in std_logic_vector( 2 downto 0 );
         in1, in2    :  in std_logic_vector( width-1 downto 0 );
    out1, out2  : out std_logic_vector( width-1 downto 0 )
 );
  end component;
  for all : FunctionUnit use entity work.FunctionUnit(behaviour);

  component ProgramCounter
    port ( ain1, ain2 : in Cell;
           clock,reset: in std_logic;
           control    : in std_logic_vector( 1 downto 0);
           aoutport: out Cell);
  end component;
  for all : ProgramCounter use entity work.ProgramCounter(behaviour);

  component Stack

generic ( width : positive := 8;
           widthad : positive := 3;
depth : positive := 8 );

    port ( clock, push, pop, d1_d2 : in std_logic;
           din1, din2              : in Cell;
reset: in std_logic;
           dout               : out Cell);
  end component;
  for all : Stack use entity work.Stack(behaviour);

  component Top
generic ( width : positive := 4 );
  port ( din1, din2          : in std_logic_vector(width-1 downto 0);
         d1_d2, clock, store : in std_logic;
         notzero             : out std_logic;
         dout                : out std_logic_vector(width-1 downto 0));
  end component;
  for all : Top use entity work.Top(behaviour2);

begin

pcout <= P;
topout <= T;

     rcontrol(0)   <=       sp_opcode(0)  ;
     rcontrol(1)   <=       sp_opcode(1)  ;
     rcontrol(2)   <=       sp_opcode(2)  ;
     dcontrol(0)   <=       sp_opcode(3)   ;
     dcontrol(1)   <=       sp_opcode(4)   ;
     dcontrol(2)   <=       sp_opcode(5)   ;
     tcontrol(0)   <=       sp_opcode(6)  ;
     tcontrol(1)    <=      sp_opcode(7)   ;
     mcontrol(0)    <=      sp_opcode(8) ;
     mcontrol(1)    <=      sp_opcode(9)  ;
     pcontrol(0)   <=       sp_opcode(10) ;
     pcontrol(1)   <=       sp_opcode(11) ;
     fcontrol(0)    <=      sp_opcode(12) ;
     fcontrol(1)    <=      sp_opcode(13) ;
     fcontrol(2)   <=       sp_opcode(14) ;

-- assign memory signals --
fetch_store <= mcontrol(1);
```

```
M <= memdata_in;
memdata_out <= D;

-- Memory --
memory_address_select : process( T, P, mcontrol(0)) is
begin
-- mcontrol(0) = a1_a2 --
if mcontrol(0)='0' then
address_out <= P;-- PC select
else
address_out <= T;-- TOP select
end if;
end process;



  IR : instruction -- instruction register
    port map ( iin => M, reset => reset, nz => nz, clock => clock,
               iout  => sp_opcode );

  DS : Stack  -- data stack
generic map ( width => word_width, widthad => data_stack_index_width,
depth => data_stack_index_depth)
    port map ( reset => reset, clock => clock, push => dcontrol(2), pop => dcon-
trol(1),
               d1_d2 => dcontrol(0), din1 => U, din2 => M, dout => D );

  RS : Stack  -- return stack
generic map ( width => word_width, widthad => return_stack_index_width,
depth => return_stack_index_depth)
    port map ( reset => reset, clock => clock, push => rcontrol(2), pop => rcon-
trol(1),
               d1_d2 => rcontrol(0), din1 => T, din2 => P, dout => R );

  PC : ProgramCounter
    port map ( ain1 => M, ain2 => R, clock => clock, reset => reset,
               control => pcontrol, aoutport => P );

  FU : FunctionUnit
generic map ( width => Cell'length )
    port map ( selfunction => fcontrol,
               --leftin => lio, rightin => rio,
               --leftout => open, rightout => open,
               --leftinout => open, rightinout => open,
               in1 => T, in2 => D,
               out1 => F, out2 => U );

  TP : Top  -- top register
generic map ( width => Cell'length )
    port map ( din1 => F, din2 => R, d1_d2 => tcontrol(0), clock => clock,
               store => tcontrol(1), notzero => nz, dout => T );


end structure;

-- .mif file auto-generated by npcc
-- copyright 1998
--
-- This file is covered by over a million patents, none of which
-- are actually worth anything except while negotiating a buyout
-- with a larger, better corporation.

WIDTH = 8;
DEPTH = 128;
```

```
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN

-- 7-segment display Artificial Neural Network program file

-- NOTE: All numeric values are in _HEXADECIMAL_

-- Addresses of pointers and counters:

-- starting address for inputs is 80
-- starting address for NP outputs is 90
-- address for weight register

-- Program Counter variables for various routine

-- an output register is available for an external output.
-- address of "result latch" is FF

-- initialize input_ptr to input_address
00: 0e; -- sto_mem_DS
01: 80; -- input_address
02: 05; -- sto_DS_memimm
03: 7e; -- input_ptr

-- initialize weight_ptr to zero
04: 0e; -- sto_mem_DS
05: 00;
06: 05; -- sto_DS_memimm
07: fd; -- weight_ptr

-- initialize NP counter
08: 0e; -- sto_mem_DS
09: 0e;
0a: 05; -- sto_DS_memimm
0b: 7c; -- NP_counter

-- initialize output_ptr
0c: 0e; -- sto_mem_DS
0d: 90; -- np_output_address
0e: 05; -- sto_DS_memimm
0f: 7a; -- output_ptr

---------------------------------------
-- This loop executes for each NP
---------------------------------------

-----------
-- next_np
-----------

-- initialize input counter
10: 0e; -- sto_mem_DS

-- NOTE: counter is 15, not 16 because adding the
-- first NP input does not increment the counter
11: 0e ;
12: 05; -- sto_DS_memimm
13: 7b; -- input_counter

-- initialize np_output_ptr
14: 0e; -- sto_mem_DS
15: 90; -- np_output_address
16: 05; -- sto_DS_memimm
```

```
17: 7f; -- np_output_ptr

-- initialize SUM
18: 03; -- psh_mem_DS
19: 00;

---------------------------------------
-- Add this NP's input
---------------------------------------

-- push dummy value to stack
1a: 03; -- psh_mem_DS
1b: 00;

-- fetch weight
1c: 08; -- sto_memimm_DS_and_sto_DS_TOP
1d: fe; -- weight_address

-- fetch input
1e: 12; -- psh_memptr_DS
1f: 7e; -- input_ptr

-----------
-- domath
-----------

-- transfer input to TOP
20: 04; -- pop_DS_TOP

-- make decision on add/subtract
21: 09; -- bnzero_imm
-- branch to **subtract**
22: 28; -- subtract_routine

-- o/p is one, so add
23: 04; -- pop_DS_TOP
24: 0c; -- add
25: 00;
-- branch to **donemath**
26: 0a; -- bra_mem
27: 2b; -- donemath_routine

-----------
-- subtract
-----------

28: 04; -- pop_DS_TOP
29: 0d; -- subtract
2a: 00;

-----------
-- donemath
-----------

-- increment weight pointer
2b: 1a; -- incr_ptr
2c: fd; -- weight_ptr

-----------------------------------
-- check_counter
--   Fetch counter and check if zero. If not, decrement
--   and do next addition/subtraction.
-----------------------------------
-- push a dummy value onto the DS
2d: 03; -- psh_mem_DS
```

```
2e: 00;
-- Fetch input_counter
2f: 08; -- sto_memimm_DS_and_sto_DS_TOP
30: 7b; -- input_counter
-- if the counter is zero, branch to done NP
31: 17; -- sto_DS_TOP
32: 0b; -- bzero_imm
33: 48; -- done_np_routine

-- decrement input_counter
34: 0e; -- sto_mem_DS
35: 01;
36: 18; -- swap
37: 0d; -- subtract
38: 00;
39: 05; -- sto_DS_memimm
3a: 7b; -- input_counter

--**********
-- DEBUG: store input counter to result latch
3b: 05; -- sto_DS_memimm
3c: FF;

-- drop top of stack
3d: 04; -- pop_DS_TOP

------------------------------------------------------
-- Fetch the NP output
------------------------------------------------------
-- push dummy value to stack
3e: 03; -- psh_mem_DS
3f: 00;

-- fetch weight
40: 08; -- sto_memimm_DS_and_sto_DS_TOP
41: fe; -- weight_address

-- fetch npoutput
42: 12; -- psh_memptr_DS
43: 7f; -- np_output_ptr

-- increment npoutput pointer
44: 1a; -- incr_ptr
45: 7f; -- np_output_ptr

-- branch to **domath**
46: 0a; -- bra_mem
47: 20; -- domath_routine

------------------------------------------------
-- done_np
--   entry con: stack contains 00, then value of sum.
------------------------------------------------

--pop stack to uncover sum
48: 04; -- pop_DS_TOP

-- store sum to output
49: 24; -- sto_DS_memptr
4a: 7a; -- output_ptr

-- push stack to save sum
4b: 03; -- psh_mem_DS
4c: 00;
```

```
-- increment input pointer
4d: 1a; -- incr_ptr
4e: 7e; -- input_ptr

-- increment output pointer
4f: 1a; -- incr_ptr
50: 7a; -- output_ptr

-----------------------------------------------------
-- check_NP_counter
--   Fetch counter and check if zero. If not, decrement
--   and goto next neroprocessor.
-----------------------------------------------------

-- test to see if all the NPs have been computed
51: 08; -- sto_memimm_DS_and_sto_DS_TOP
52: 7c; -- NP_counter

-- if the counter is zero, branch to done_ann
53: 17; -- sto_DS_TOP
54: 0b; -- bzero_imm
55: 63; -- done_ann_routine

-- decrement NP counter
56: 0e; -- sto_mem_DS
57: 01;
-- swap since after subtract, DS = DS - TOP
58: 18; -- swap
59: 0d; -- subtract
5a: 00;
5b: 05; -- sto_DS_memimm
5c: 7c; -- NP_counter

--**************
-- DEBUG: store input counter to result latch
5d: 05; -- sto_DS_memimm
5e: FF;
-----------------------------------------------------

-- reset the DS
5f: 04; -- pop_DS_TOP
60: 04; -- pop_DS_TOP

-- branch to **next_np**
61: 0a; -- bra_mem
62: 10; -- next_np_routine

----------------------------------------
-- done_ann
--    entry con: stack contains 00 and then sum
----------------------------------------

-- write "be" to the output latch for show
63: 0e; -- sto_mem_DS
64: BE;
65: 05; -- sto_DS_memimm
66: FF;

-- branch to beginning
67: 0a; -- bra_mem
68: 00;

-- end.

END;
```

B.2 XOR NPN

```
xor_ui.vhd
    +xor_led.vhd
    +xor_npn_sp.vhd
        +memory.vhd
            +weight_rom.mif
        +stack_processor.vhd
            +np_program.mif


library ieee;
use ieee.std_logic_1164.all;

package npn_types is
subtype sp_word is std_logic_vector( 7 downto 0 );
end npn_types;

library ieee;
use ieee.std_logic_1164.all;

use work.npn_types.all;

entity xor_npn_sp is
    port ( in1,
in2,
--reset,
clock : in std_logic;

--result : out sp_word;
output : out std_logic;
outputs : out std_logic_vector(14 downto 0)
);
end;

architecture structure of xor_npn_sp is

signal reset : std_logic;
signal result : sp_word;

signal xor_inputs : std_logic_vector( 0 to 1 );

signal outputs_buffer : std_logic_vector( 14 downto 0);

component stack_processor
generic( data_stack_index_width : integer := 3;
  data_stack_index_depth : integer := 8;
  return_stack_index_width : integer := 3;
  return_stack_index_depth : integer := 8
);
port (
reset, clock : in std_logic;
memdata_in : in std_logic_vector( 7 downto 0);
memdata_out : out std_logic_vector ( 7 downto 0);
address_out : out std_logic_vector( 7 downto 0);
fetch_store : out std_logic;
topout : out std_logic_vector( 7 downto 0);
pcout : out std_logic_vector( 7 downto 0)
);
end component;

component memory
```

```
  port (
ain,
din: in std_logic_vector( 7 downto 0);

clock,
fetch_store: in std_logic;

 input_vector: in std_logic_vector( 0 to 1 );

        dout,
result: out std_logic_vector( 7 downto 0);

output: out std_logic;

out1, out2, out3, out4, out5 : out std_logic_vector( 2 downto 0 )
);
end component;

signal  address,
data_tomem,
data_tosp,
periph_tomem,
program_counter,
periph_toout: sp_word;

signal  fetch_store
: std_logic;

signal  out1, out2, out3, out4, out5 : std_logic_vector( 2 downto 0);

signal blah : std_logic;
begin

output <= blah;
xor_inputs(0) <= in1;
xor_inputs(1) <= in2;

reset <= '0';

outputs <= outputs_buffer;

outputs_buffer(14 downto 12) <= out5;
outputs_buffer(11 downto 9) <= out4;
outputs_buffer(8 downto 6) <= out3;
outputs_buffer(5 downto 3) <= out2;
outputs_buffer(2 downto 0) <= out1;

my_memory : memory
port map (
ain => address,
din => data_tomem,
input_vector => xor_inputs,
clock => clock,
fetch_store => fetch_store,
dout => data_tosp,
result => result,
output => blah,
out1 => out1,
out2 => out2,
out3 => out3,
out4 => out4,
out5 => out5
);

my_sp : stack_processor
```

```
generic map (
data_stack_index_width => 3,
data_stack_index_depth => 8,
return_stack_index_width => 2,
return_stack_index_depth => 4
)
port map (
reset => reset,
clock => clock,
memdata_in => data_tosp,
memdata_out => data_tomem,
address_out => address,
fetch_store => fetch_store
);

end structure;

Library IEEE;
use IEEE.STD_Logic_1164.all;

library lpm;
use lpm.lpm_components.all;

entity Memory is
  port (
ain,
din:  in std_logic_vector( 7 downto 0);

clock,
fetch_store:  in std_logic;

 input_vector: in std_logic_vector( 0 to 1 );

        dout,
result: out std_logic_vector( 7 downto 0);

output: out std_logic;

out1, out2, out3, out4, out5 : out std_logic_vector( 2 downto 0)
);
end;

architecture behaviour2 of Memory is

subtype Cell is std_logic_vector(7 downto 0);

type decode_state_type is ( ram,  weight, io );
signal decode_state, next_decode_state : decode_state_type;

type input_ram_state_type is ( input_port, output_ram );
signal input_ram_state, next_input_ram_state : input_ram_state_type;

signal decode_bits : std_logic_vector( 1 downto 0 );

signal address_4bit : std_logic_vector( 3 downto 0 );
signal address_6bit : std_logic_vector( 5 downto 0 );
signal address_7bit : std_logic_vector( 6 downto 0 );

signal program_dout,
   weight_dout,
   input_dout    : Cell;

signal weight_dout_2bit, zero_2bit : std_logic_vector ( 1 downto 0 );

signal np_output, din_signbit : std_logic_vector( 0 downto 0 );
```

```
signal clock_inv,
zero,
input_bit,
input_port_bit : std_logic;

signal io_select, input_we, ram_we : std_logic;

signal din_3bit : std_logic_vector( 2 downto 0 );

begin

clock_inv <= not clock;
zero <= '0';
zero_2bit <= "00";

-- program ram spans memory locations 0 to $7f ( 7 bits )
program_ram : component LPM_RAM_DQ
generic map (
lpm_width => Cell'length,
lpm_widthad => 7,
lpm_file => "np_program.mif"
)
port map (
inclock => clock,
outclock => clock_inv,
we  => ram_we,
address => address_7bit,
data => din,
q => program_dout
);

-- weight rom spans memory locations $80 to $bf ( 6 bits )
weight_rom : component LPM_RAM_DQ
generic map (
lpm_width => 2,
lpm_widthad => 6,
lpm_file => "weight_rom.mif"
)
port map (
inclock => clock,
outclock => clock_inv,
we  => zero,
address => address_6bit,
data => zero_2bit,
q => weight_dout_2bit
);

-- this is just one-bit ram for the outputs of the NPs
-- found in memory locations 64 to 64+15=79 (16 locations)
-- however io memory space spans $c0 to $ff
np_output_ram : component LPM_RAM_DQ
generic map (
lpm_width => 1,
lpm_widthad => 4
)
port map (
inclock => clock,
outclock => clock_inv,
we  => input_we,
address => address_4bit,
data => din_signbit,
q => np_output
);

--
```

```
-- assign ain to the different address signals
--
decode_bits  <= ain( 7 downto 6 );
address_4bit <= ain( 3 downto 0 );
address_6bit <= ain( 5 downto 0 );
address_7bit <= ain( 6 downto 0 );

-- put the LSB of din on a separate signal
din_signbit(0) <= din(7);
din_3bit( 2 downto 0 ) <= din(2 downto 0);

-- assign the 2bit weight to the full weight
weight_dout(0) <= weight_dout_2bit(0);
weight_dout(1) <= weight_dout_2bit(1);
weight_dout(2) <= weight_dout_2bit(1);
weight_dout(3) <= weight_dout_2bit(1);
weight_dout(4) <= weight_dout_2bit(1);
weight_dout(5) <= weight_dout_2bit(1);
weight_dout(6) <= weight_dout_2bit(1);
weight_dout(7) <= weight_dout_2bit(1);

-----------------------------------------------------
-- address_decode
--
-- do address decoding
-----------------------------------------------------
address_decode_state : process( decode_bits )
begin
    case decode_bits is
when "00" | "01" =>-- select program ram
next_decode_state <= ram;

when "10"  => -- select weight rom
next_decode_state <= weight;

when "11" => -- select input/output ram
next_decode_state <= io;

when others =>  -- shouldn't happen...
next_decode_state <= ram;
end case;

--
-- take care of write enable signals
--
case next_decode_state is
when ram =>
ram_we <= fetch_store;
input_we <= '0';
when io =>
ram_we <= '0';
input_we <= fetch_store;
when others =>
ram_we <= '0';
input_we <= '0';
end case;
end process;

change_decode_state : process(clock)
begin
if clock'event and clock='1' then
decode_state <= next_decode_state;
end if;
end process;
```

```
address_decode : process( decode_state, program_dout, weight_dout, input_dout )
begin
case decode_state is
when ram =>
dout <= program_dout;

when weight =>
dout <= weight_dout;

when io =>-- inputs, and also output latch
dout <= input_dout;
end case;
end process;

-------------------------------------------------
-- input_ram controller
--
-- Control memory access to input and NP output signals
-------------------------------------------------

input_dout(0) <= input_bit;
input_dout(1) <= '0';
input_dout(2) <= '0';
input_dout(3) <= '0';
input_dout(4) <= '0';
input_dout(5) <= '0';
input_dout(6) <= '0';
input_dout(7) <= '0';

input_ram_nextstate : process(address_6bit)
begin
case address_6bit is
when o"00" => -- this is octal.  bin = "xx000000"
next_input_ram_state <= input_port;
when o"01" =>
next_input_ram_state <= input_port;
when others =>
next_input_ram_state <= output_ram;
end case;
end process;

input_ram_changestate : process(clock)
begin
if clock'event and clock='1' then
input_ram_state <= next_input_ram_state;
end if;
end process;

input_ram_decode : process( input_ram_state, input_port_bit, np_output(0) )
begin
case input_ram_state is
when input_port =>
input_bit <= input_port_bit;

when output_ram =>
input_bit <= np_output(0);
end case;
end process;

input_port_register : process( clock, input_vector )
begin
if clock'event and clock='1' then
case address_6bit is
when o"00" =>
input_port_bit <= input_vector(0);
```

```
when o"01" =>
input_port_bit <= input_vector(1);
when others =>
-- nothing
end case;
end if;
end process;

--------------------------------------------------------------------------------

---------------------------------
-- output latch
--
-- fully decode the output latch address and
-- write to it
---------------------------------
output_latch : process( clock )

begin
if clock'event and clock='1' then
if fetch_store = '1' then
if ain = x"c5" then
out1 <= din_3bit;
end if;
if ain = x"c6" then
out2 <= din_3bit;
end if;
if ain = x"c7" then
out3 <= din_3bit;
end if;
if ain = x"c8" then
out4 <= din_3bit;
end if;
if ain = x"c9" then
out5 <= din_3bit;
output <= din_signbit(0);
end if;

if ain = x"ff" then
result <= din;
end if;
end if;
end if;
end process;

end behaviour2;


-- MAX+plus II - generated Memory Initialization File

-- Copyright (C) 1991-1997 Altera Corporation
-- Any megafunction design, and related net list (encrypted or decrypted),
-- support information, device programming or simulation file, and any other
-- associated documentation or information provided by Altera or a partner
-- under Altera's Megafunction Partnership Program may be used only to
-- program PLD devices (but not masked PLD devices) from Altera.  Any other
-- use of such megafunction design, net list, support information, device
-- programming or simulation file, or any other related documentation or
-- information is prohibited for any other purpose, including, but not
-- limited to modification, reverse engineering, de-compiling, or use with
-- any other silicon devices, unless such use is explicitly licensed under
-- a separate agreement with Altera or a megafunction partner.  Title to
-- the intellectual property, including patents, copyrights, trademarks,
-- trade secrets, or maskworks, embodied in any such megafunction design,
-- net list, support information, device programming or simulation file, or
```

```
-- any other related documentation or information provided by Altera or a
-- megafunction partner, remains with Altera, the megafunction partner, or
-- their respective licensors.  No other licenses, including any licenses
-- needed under any third party's intellectual property, are provided herein.

WIDTH = 2;
DEPTH = 64;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT BEGIN
0:01;
1:00;
2:00;
3:00;
4:00;
5:00;
6:01;
7:00;
8:00;
9:00;
a:00;
b:00;
c:00;
d:01;
e:11;
f:00;
10:00;
11:00;
12:00;
13:11;
14:01;
15:00;
16:00;
17:00;
18:11;
19:00;
1a:00;
1b:01;
1c:01;
1d:00;
1e:00;
1f:00;
20:00;
21:00;
22:00;
23:00;
24:00;
25:00;
26:00;
27:00;
28:00;
29:00;
2a:00;
2b:00;
2c:00;
2d:00;
2e:00;
2f:00;
30:00;
31:00;
32:00;
33:00;
34:00;
```

```
35:00;
36:00;
37:00;
38:00;
39:00;
3a:00;
3b:00;
3c:00;
3d:00;
3e:00;
3f:00;
END;

-- .mif file auto-generated by npcc
-- copyright 1998
--
-- This file is covered by over a million patents, none of which
-- are actually worth anything except while negotiating a buyout
-- with a larger, better corporation.

WIDTH = 8;
DEPTH = 128;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN

-- Pattern Recognition Artificial Neural Network program file

-- NOTE: All numeric values are in _HEXADECIMAL_

-- Addresses of pointers and counters:

-- starting address for inputs is c0
-- starting address for NP outputs is c5
-- starting address for weights is 80

-- Program Counter values for various routine
-- (necessary because this is only a one-pass compiler!)

-- an output register is available for an external output.
-- address of "result latch" is FF

-- initialize input_ptr to input_address
00: 0e; -- sto_mem_DS
01: c0; -- input_address
02: 05; -- sto_DS_memimm
03: 7e; -- input_ptr

-- initialize weight_ptr to weight_address
04: 0e; -- sto_mem_DS
05: 80; -- weight_address
06: 05; -- sto_DS_memimm
07: 7d; -- weight_ptr

-- initialize NP counter
08: 0e; -- sto_mem_DS
09: 05;
0a: 05; -- sto_DS_memimm
0b: 7c; -- NP_counter

-- initialize output_ptr
0c: 0e; -- sto_mem_DS
0d: c5; -- np_output_address
```

```
0e: 05; -- sto_DS_memimm
0f: 7a; -- output_ptr

---------------------------------------
-- This loop executes for each NP
---------------------------------------

-----------
-- next_np
-----------

-- initialize input counter
10: 0e; -- sto_mem_DS

-- NOTE: counter is 5, not 6 because adding the
-- first NP input does not increment the counter
11: 05 ;
12: 05; -- sto_DS_memimm
13: 7b; -- input_counter

-- initialize np_output_ptr
14: 0e; -- sto_mem_DS
15: c5; -- np_output_address
16: 05; -- sto_DS_memimm
17: 7f; -- np_output_ptr

-- initialize SUM
18: 03; -- psh_mem_DS
19: 00;

---------------------------------------
-- Add this NP's input
---------------------------------------

-- fetch weight
1a: 12; -- psh_memptr_DS
1b: 7d; -- weight_ptr

-- fetch input
1c: 12; -- psh_memptr_DS
1d: 7e; -- input_ptr

-----------
-- domath
-----------

-- transfer input to TOP
1e: 04; -- pop_DS_TOP

-- make decision on add/subtract
1f: 09; -- bnzero_imm
-- branch to **subtract**
20: 26; -- subtract_routine

-- o/p is one, so add
21: 04; -- pop_DS_TOP
22: 0c; -- add
23: 00;
-- branch to **donemath**
24: 0a; -- bra_mem
25: 29; -- donemath_routine

-----------
-- subtract
-----------
```

```
26: 04; -- pop_DS_TOP
27: 0d; -- subtract
28: 00;

-----------
-- donemath
-----------

-- increment weight pointer
29: 1a; -- incr_ptr
2a: 7d; -- weight_ptr

-----------------------------------
-- check_counter
--   Fetch counter and check if zero. If not, decrement
--   and do next addition/subtraction.
-----------------------------------
-- push a dummy value onto the DS
2b: 03; -- psh_mem_DS
2c: 00;
-- Fetch input_counter
2d: 08; -- sto_memimm_DS_and_sto_DS_TOP
2e: 7b; -- input_counter
-- if the counter is zero, branch to done NP
2f: 17; -- sto_DS_TOP
30: 0b; -- bzero_imm
31: 44; -- done_np_routine

-- decrement input_counter
32: 0e; -- sto_mem_DS
33: 01;
34: 18; -- swap
35: 0d; -- subtract
36: 00;
37: 05; -- sto_DS_memimm
38: 7b; -- input_counter

--*************
-- DEBUG: store input counter to result latch
39: 05; -- sto_DS_memimm
3a: FF;

-- drop top of stack
3b: 04; -- pop_DS_TOP

-----------------------------------------------------
-- Fetch the NP output
-----------------------------------------------------
-- fetch weight
3c: 12; -- psh_memptr_DS
3d: 7d; -- weight_ptr

-- fetch npoutput
3e: 12; -- psh_memptr_DS
3f: 7f; -- np_output_ptr

-- increment npoutput pointer
40: 1a; -- incr_ptr
41: 7f; -- np_output_ptr

-- branch to **domath**
42: 0a; -- bra_mem
43: 1e; -- domath_routine
```

```
-------------------------------------------------
-- done_np
--   entry con: stack contains 00, then value of sum.
-------------------------------------------------

--pop stack to uncover sum
44: 04; -- pop_DS_TOP

-- store sum to output
45: 24; -- sto_DS_memptr
46: 7a; -- output_ptr

-- push stack to save sum
47: 03; -- psh_mem_DS
48: 00;

-- increment input pointer
49: 1a; -- incr_ptr
4a: 7e; -- input_ptr

-- increment output pointer
4b: 1a; -- incr_ptr
4c: 7a; -- output_ptr

-----------------------------------------------------
-- check_NP_counter
--   Fetch counter and check if zero. If not, decrement
--   and goto next neroprocessor.
-----------------------------------------------------

-- test to see if all the NPs have been computed
4d: 08; -- sto_memimm_DS_and_sto_DS_TOP
4e: 7c; -- NP_counter

-- if the counter is zero, branch to done_ann
4f: 17; -- sto_DS_TOP
50: 0b; -- bzero_imm
51: 5f; -- done_ann_routine

-- decrement NP counter
52: 0e; -- sto_mem_DS
53: 01;
-- swap since after subtract, DS = DS - TOP
54: 18; -- swap
55: 0d; -- subtract
56: 00;
57: 05; -- sto_DS_memimm
58: 7c; -- NP_counter

--**********
-- DEBUG: store input counter to result latch
59: 05; -- sto_DS_memimm
5a: FF;
-------------------------------------------------

-- reset the DS
5b: 04; -- pop_DS_TOP
5c: 04; -- pop_DS_TOP

-- branch to **next_np**
5d: 0a; -- bra_mem
5e: 10; -- next_np_routine

----------------------------------------
-- done_ann
```

```
--    entry con: stack contains 00 and then sum
----------------------------------------

-- write "be" to the output latch for show
5f: 0e; -- sto_mem_DS
60: BE;
61: 05; -- sto_DS_memimm
62: FF;

-- should branch, but we're just going to stop.
63: 0a; -- bra_mem
64: 00;

-- end.

END;
```

C XOR UI

```
-- Difference detector NP network UI  Rob Chapman  Mar 18, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_ui is
  port ( clock : in std_logic;
         vga : out std_logic_vector(1 to 5);
         led : out std_logic_vector(1 to 16);
         mouse : in std_logic_vector(1 to 2);
         switches : in std_logic_vector(1 to 10));
end;

architecture structure of xor_ui is

  component xor_npn
    port ( in1, in2, clock : in std_logic;
           output : out std_logic;
           outputs : out std_logic_vector(14 downto 0));
  end component;

  component xor_led
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10);
           led      : out std_logic_vector(1 to 16);
           outputs  : in std_logic_vector(14 downto 0));
  end component;

  component xor_vga
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10);
           output   : in std_logic;
           vga      : out std_logic_vector(1 to 5));
  end component;

  signal pulse : std_logic; -- for setting network clock
  signal outputs : std_logic_vector(14 downto 0);
  signal in1, in2, output  : std_logic;
  signal led_not : std_logic_vector(1 to 16); -- inverted led sense

begin
```

```
  -- set the pulse for the nps; could be event triggered to for low
  -- power and instant response to input change
  pulse <= clock;

  -- connect switches to inputs
  in1 <= switches(3);
  in2 <= switches(4);

  -- led output with 50% duty cycle
  process
    variable blank : std_logic;
  begin
    wait until clock = '1';
    if  blank = '1'  then
      led <= (others => '1');
    else
      led <= not led_not;
    end if;
    blank := not blank;
  end process;

  -- connect up components
--  led <= not led_not;
  leds : xor_led -- led digit display
    port map ( clock => pulse,
               switches => switches,
               led => led_not,
               outputs => outputs);

  display : xor_vga -- vga display
    port map ( clock => pulse,
               switches => switches,
               output => output,
               vga => vga);

  npn : xor_npn -- neural network
    port map ( in1 => in1, in2 => in2,
               clock => pulse,
               output => output,
               outputs => outputs);

end structure;

-- XOR NPN VGA objects  Rob Chapman  Mar 18, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_led is
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10);
           led      : out std_logic_vector(1 to 16);
           outputs  : in std_logic_vector(14 downto 0));
end;

architecture structure of xor_led is

  -- digits 0-9: order of segments is "abcdefg"
  subtype digit is std_logic_vector(1 to 7);
  constant digit0 : digit := "1111110";
  constant digit1 : digit := "0110000";
  constant digit2 : digit := "1101101";
  constant digit3 : digit := "1111001";
```

```
  constant digit4 : digit := "0110011";
  constant digit5 : digit := "1011011";
  constant digit6 : digit := "1011111";
  constant digit7 : digit := "1110000";
  constant digit8 : digit := "1111111";
  constant digit9 : digit := "1111011";

  -- letter E for error
  constant letterE : digit := "1001111";

  -- np counter and led signals
  signal np : natural range 1 to 5;
  signal sum : std_logic_vector(2 downto 0);

  alias left_digit  : std_logic_vector (1 to 7) is led(1 to 7);
  alias left_dot    : std_logic is led(8);
  alias right_digit : std_logic_vector (1 to 7) is led(9 to 15);
  alias right_dot   : std_logic is led(16);

begin

  -- user interface
   -- push button one is used to cycle the left led digit from np 1 to 5
   -- push button two is used to cycle the digit the other way
   -- the sum of each np is displayed in right led digit with dot as sign
   -- dip switch one and two are used as the two inputs into the xor npn
   -- the dot on the left led digit is the output of the xor npn,
   -- it is on, if the dip switches are different

  -- led digits and dots
  with np select
    left_digit <= digit1 when 1,
                  digit2 when 2,
                  digit3 when 3,
                  digit4 when 4,
                  digit5 when 5,
                  letterE when others;

  left_dot <= outputs(14);

  with sum select
    right_digit <= digit0 when "000",
                   digit1 when "001",
                   digit1 when "111",
                   digit2 when "010",
                   digit2 when "110",
                   digit3 when "011",
                   digit3 when "101",
                   letterE when others;

  right_dot <= sum(2);

  -- sum from currently selected neuroprocessor
  with np select
    sum <= outputs( 2 downto 0 ) when 1,
           outputs( 5 downto 3 ) when 2,
           outputs( 8 downto 6 ) when 3,
           outputs(11 downto 9 ) when 4,
           outputs(14 downto 12) when 5,
           "100" when others;

  -- incrementing of np by push button 1
  process
    constant limit   : integer := 25175;
    variable pbstate : integer range -limit to limit;
```

```
    variable pressed : std_logic;
    constant press   : integer :=  (limit - 1);  -- 25,175,000 hz clock
    constant release : integer := -(limit - 1);
     -- a millisecond of delay is used to debounce the input
  begin
    wait until clock = '1';

      if  switches(1) = '1' and  switches(2) = '1'  then
        pressed := '0';
      end if;

      if  pressed = '0'  then
        if  switches(1) = '0' or switches(2) = '0'  then
          pbstate := pbstate + 1;
        else
          pbstate := pbstate - 1;
        end if;
      end if;

      if  pbstate = press  then
        if  switches(1) = '0'  then
          if  np = 5  then
            np <= 1;
          else
            np <= np + 1;
          end if;
        else
          if  np = 1  then
            np <= 5;
          else
            np <= np - 1;
          end if;
        end if;
      end if;

      if  pbstate > press  then
        pbstate := 0;
        pressed := '1';
      elsif  pbstate < release  then
        pbstate := 0;
      end if;

  end process;

end architecture;


-- XOR NPN VGA objects  Rob Chapman  Mar 18, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity xor_vga is
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10);
           output   : in std_logic;
           vga      : out std_logic_vector(1 to 5));
end;

architecture structure of xor_vga is
begin
  vga <= "00000"; -- stubbed
end architecture;
```

D Seven UI

```
-- Seven segment digit recognizer  Rob Chapman  Mar 25, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

use work.params.all;

entity seven is
  port ( clock    : in  std_logic;
         switches : in  std_logic_vector(1 to 10);
         led      : out std_logic_vector(1 to 16));
end;

architecture structure of seven is

  component sevennpn
    port ( clock   : in  std_logic;
           input   : in  iobus;
           output  : inout iobus;
           outputs : inout sums);
  end component;

  component sevenui
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10); -- pb1-2 sw1-8
           inputs   : buffer iobus;
           output   : in iobus;
           outputs  : in sums;
           led      : out std_logic_vector(1 to 16));
  end component;

  signal pulse         : std_logic; -- for setting network heartbeat
  signal outputs       : sums;
  signal inputs, output : iobus;

begin

  -- set the pulse for the nps; could be event triggered too for low
  -- power and instant response to input change
  HB : process
  begin
    wait until clock = '1';
    if  pulse = '0' then
      pulse <= '1';
    else
      pulse <= '0';
    end if;
  end process;

  -- connect up components
  ui : sevenui -- user interface
   port map ( clock    => pulse,
              switches => switches,
              inputs   => inputs,
              output   => output,
              outputs  => outputs,
              led      => led);
```

```
  npn : sevennpn -- neural network
    port map ( clock   => pulse,
               input   => inputs,
               output  => output,
               outputs => outputs);

end structure;

-- user interface for digit recognizer  Rob Chapman  Mar 25, 1998

Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

use work.params.all;

entity sevenui is
    port ( clock    : in std_logic;
           switches : in std_logic_vector(1 to 10); -- pb1-2 sw1-8
           inputs   : buffer iobus;
           output   : in iobus;
           outputs  : in sums;
           led      : out std_logic_vector(1 to 16));
end;

architecture structure of sevenui is

  -- digits 0-9: order of segments is "abcdefg"
  --   a_
  --  f|_|b  g in the middle
  --  e|_|c
  --    d  .p for the point

  subtype digit is std_logic_vector(1 to 7);

  constant digit0 : digit := "1111110";
  constant digit1 : digit := "0110000";
  constant digit2 : digit := "1101101";
  constant digit3 : digit := "1111001";
  constant digit4 : digit := "0110011";
  constant digit5 : digit := "1011011";
  constant digit6 : digit := "1011111";
  constant digit7 : digit := "1110000";
  constant digit8 : digit := "1111111";
  constant digit9 : digit := "1111011";

  -- letters
  constant letterA : digit := "1110111";
  constant letterB : digit := "0011111";
  constant letterC : digit := "0001101";
  constant letterD : digit := "0111101";
  constant letterE : digit := "1001111";
  constant letterF : digit := "1000111";
  constant letterG : digit := "1011110";
  constant letterH : digit := "0110111";
  constant letterI : digit := "0000110";
  constant letterJ : digit := "0111100";
  constant letterK : digit := "0100111";
  constant letterL : digit := "0001110";
  constant letterM : digit := "1110110";
  constant letterN : digit := "0010101";
  constant letterO : digit := "0011101";
  constant letterP : digit := "1100111";
  constant letterQ : digit := "1110011";
```

```
constant letterR : digit := "0000101";
constant letterS : digit := "1011011";
constant letterT : digit := "1000110";
constant letterU : digit := "0011100";
constant letterV : digit := "0010100";
constant letterW : digit := "0111110";
constant letterX : digit := "0110110";
constant letterY : digit := "0110011";
constant letterZ : digit := "1001001";

-- Test patterns
-- Input#1
--     _
--    | |
--    |_|
constant test1 : digit := "1111110";

-- Input#2
--
--      |
--      |
constant test2 : digit := "0110000";

-- Input#3
--     _
--     _|
--    |_
constant test3 : digit := "1101101";

-- Input#4
--     _
--    | |
--
constant test4 : digit := "1100010";

-- Input#5
--
--      |
--
constant test5 : digit := "0100000";

-- Input#6
--     _
--     _|
--
constant test6 : digit := "1100001";

-- Input#7
--     _
--    |_|
--    |_|
constant test7 : digit := "1111111";

-- Input#8
--
--     _|
--      |
constant test8 : digit := "0110001";

-- Input#9
--     _
--    |_|
--    |_
constant test9 : digit := "1101111";
```

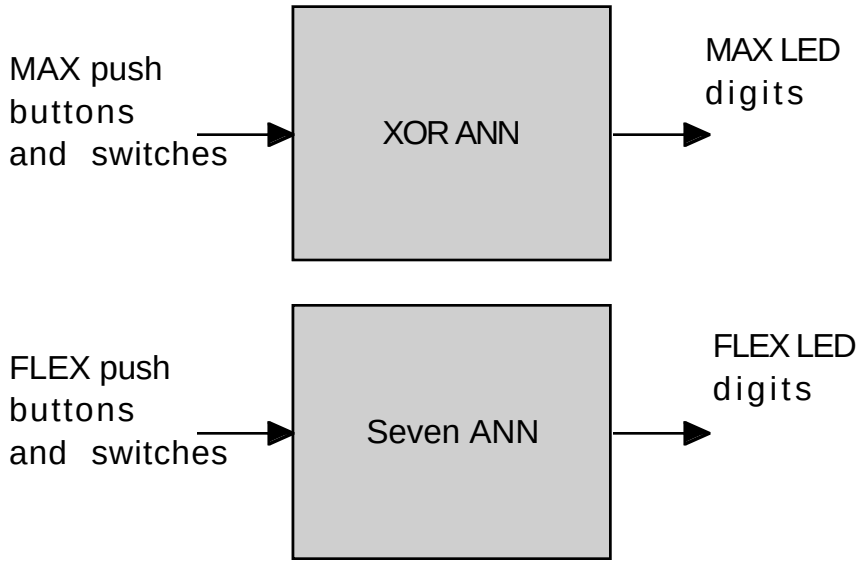
```
  -- Input#10
  --     _
  --    _|
  --   |_|
  constant test10 : digit := "1111101";

  -- Input#11
  --     _
  --    | |
  --    |_
  constant test11 : digit := "1101110";

  signal pattern : digit;  -- current test pattern

  -- np counter and led signals
  signal np  : natural range 1 to N;
  signal sum : asum;

  signal led_not    : std_logic_vector(1 to 16);
  alias left_digit  : std_logic_vector (1 to 7) is led_not(1 to 7);
  alias left_dot    : std_logic is led_not(8);
  alias right_digit : std_logic_vector (1 to 7) is led_not(9 to 15);
  alias right_dot   : std_logic is led_not(16);

  -- push buttons
  signal pushed : std_logic; -- set to one for a clock cycle when button pressed
  signal button : std_logic; -- zero for left button, one for right button

begin

  -- bug2 avoidance for Altera compiler
  inputs(0) <= '0';

  -- user interface
   -- The first seven switches are used to set the segments for the left
   -- digit.  The eighth switch toggles between displaying the selected
   -- np with it's sum and the input and output characters.  In first mode:
   -- push button one is used to cycle the left led digit from np 1 to 14H
   -- The sum of each np is displayed in right led digit with dot as sign.
   -- Push button two cycles down.  In the second mode, the left push
   -- button is used to cycle through eleven test patterns and the right
   -- button is used to select the input switches.  The input selection
   -- can only be changed when switch 8 is down.

  -- push button detector; note they are '0' when pressed
  process
    constant limit   : integer := 25175;  -- 25,175,000 hz clock
    variable pbstate : integer range 0 to limit;
    variable pressed : std_logic;
    constant press   : integer :=  (limit - 1);
     -- a millisecond of delay is used to debounce the input
  begin
    wait until clock = '1';

      if  pressed = '0'  and  (switches(1) = '0' or switches(2) = '0')  then
        pbstate := pbstate + 1;
        if  pbstate = press  then
          pressed := '1';
          pushed <= '1';
        end if;
      else
        pbstate := 0;
        pushed <= '0';
      end if;
```

```
    if  switches(1) = '1' and  switches(2) = '1'  then
      pressed := '0';
    end if;

    button <= switches(1);

end process;

-- changing of np by push button 1 and 2
process
begin
  wait until clock = '1';

  if  pushed = '1' and switches(10) = '1'  then -- a push button is pressed
    if  button = '0'  then  -- test for pb1 pressed

      if  np = N  then  -- and increment
        np <= 1;
      else
        np <= np + 1;
      end if;

    else                          -- assume pb2 is pressed

      if  np = 1  then  -- else decrement
        np <= N;
      else
        np <= np - 1;
      end if;

    end if;

  end if;

end process;

-- sum from currently selected neuroprocessor
sum <= outputs(np);

-- led digits and dots
with np select
  left_digit <= digit1 when 1,
                digit2 when 2,
                digit3 when 3,
                digit4 when 4,
                digit5 when 5,
                digit6 when 6,
                digit7 when 7,
                digit8 when 8,
                digit9 when 9,
                letterA when 10,
                letterB when 11,
                letterC when 12,
                letterD when 13,
                letterE when 14,
                letterZ when others;

left_dot <= '0';

with sum select
  right_digit <= digit0  when "000000",
                 digit1  when "000001"|"111111",
                 digit2  when "000010"|"111110",
                 digit3  when "000011"|"111101",
                 digit4  when "000100"|"111100",
```

```
                    digit5  when "000101"|"111011",
                    digit6  when "000110"|"111010",
                    digit7  when "000111"|"111001",
                    digit8  when "001000"|"111000",
                    digit9  when "001001"|"110111",
                    letterA when "001010"|"110110",
                    letterB when "001011"|"110101",
                    letterC when "001100"|"110100",
                    letterD when "001101"|"110011",
                    letterE when "001110"|"110010",
                    letterF when "001111"|"110001",
                    letterG when "010000"|"110000",
                    letterH when "010001"|"101111",
                    letterI when "010010"|"101110",
                    letterJ when "010011"|"101101",
                    letterK when "010100"|"101100",
                    letterL when "010101"|"101011",
                    letterM when "010110"|"101010",
                    letterZ when others;

  right_dot <= sum(sumbits-1);

  testpattern : process
    variable counter : positive range 1 to 11;
  begin
    wait until clock = '1';

    if  pushed = '1' and button = '0' and switches(10) = '0'  then
      if  counter = 11  then
        counter := 1;
      else
        counter := counter + 1;
      end if;
    end if;

    case counter is
      when  1   => pattern <= test1;
      when  2   => pattern <= test2;
      when  3   => pattern <= test3;
      when  4   => pattern <= test4;
      when  5   => pattern <= test5;
      when  6   => pattern <= test6;
      when  7   => pattern <= test7;
      when  8   => pattern <= test8;
      when  9   => pattern <= test9;
      when  10  => pattern <= test10;
      when  11  => pattern <= test11;
      when  others  => pattern <= letterZ;
    end case;

  end process testpattern;

  -- choose network input
  process(switches(10), pushed, button)
  begin
    if  switches(10) = '0'  and  pushed = '1'  then
      if  button='0'  then
        inputs(1 to 7) <= pattern;
      else
        inputs(1 to 7) <= switches(3 to 9);
      end if;
    end if;
  end process;

  -- second layer inputs
```

```
    sli : for i in 8 to N generate
            inputs(i) <= '0';
          end generate;

    -- led output either inputs and outputs or np# and sum
    with  switches(10)  select
      led <= not (inputs(1 to 7)&'0'&output(8 to 14)&'0') when '0',
             not led_not when others;

end architecture;
```

E Data Sheet



These two chips implement two different artificial neural networks. The first one detects when the inputs are different while the second one does digit recognition. The MAX part is 90% full and take 204 LCs while the FLEX part takes up 645 LCs and 3072 bits of memory.

The hookup pins for the MAX part are as follows:

Pin name	Pin number
led1	58
led2	60
led3	61
led4	63
led5	64
led6	65
led7	67
led8	68
led9	69
led10	70
led11	73
led12	74
led13	76
led14	75
led15	77
led16	79
pb1	54
pb2	57
switch1	56
switch2	55

The hookup pins for the FLEX part are as follows:

Pin name	Pin number
led1	6
led2	7
led3	8
led4	9
led5	11
led6	12
led7	13
led8	14
led9	17
led10	18
led11	19
led12	20
led13	21
led14	23
led15	24
led16	25
pb1	28
pb2	29
switch1	41
switch2	40
switch3	39
switch4	38
switch5	36
switch6	35
switch7	34
switch8	33