PICOBLAZE BASED SELF ORGANIZING LEARNING ARRAY AND ITS EXPERIMENTAL SETTING

A Project Report Presented to

The Faculty of the Russ College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

By

Yongtao Guo

November, 2004

This report entitled

Picoblaze Based Self Organizing Learning Array And Its Experimental Setting

by Yongtao Guo

has been approved for

the School of Electric Engineering and Computer Science

and the Russ College of Engineering and Technology by

Janusz A. Starzyk

Professor of School of Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Fritz J. and Dolores H. Russ

College of Engineering and Technology

Acknowledgements

First of all, I want to thank my advisor – Dr. Starzyk, who recognized that the PicoBlaze microcontroller core is adaptable to neural network application at the very early stage after PicoBlaze was developed. And before then, I had been working on the SOLAR implementation using a kind of serial based processing in the hardware. This new design component (PicoBlaze) gave me much interest and challenge. And also during this whole period, he gave me much direct help and directions.

Secondly, I would also thank all the group members who helped me a lot via discussion, advice or even argument. They are Zhineng Zhu, Haibo He, Mingwei Ding, Zhen Zhu, Yingying Liu and etc. Among them, Zhineng Zhu's work has some overlap with me, so we had many discussions. He contributed to the routing work for the future improvement of this structure. His work was combined with mine resulting in a conference paper and a journal paper on this topic.

Finally, I would thank my family. They are always with me, give me encouragement, and teach me to how to face and overcome difficulty.

Table of Contents

List of Tables 5

List of Figures 6

Abbreviations 7

1. Introduction 8

2. What Is PicoBlaze 9

3. How To Build A SOLAR Neuron From Modified PicoBlaze 10

3.1 Neuron Structure 10

3.2 Neuron Programming 11

4. How To Connect 28 Neurons 15

5. Some Explanation About The VHDL Code 17

6. How To Setup The Experiment 25

7. How To Modify The Design To From 2x14 To 4x7 Structure 38

8. Summary And Future Work 46

Reference 48

List of Tables

Table 1. Register definition and description ……………………………………………24

Table 7.1 Implementation reports………………………………………………………….40

List of Figures

Figure 3.1 Single neuron's schematic …………………………………………………...11

Figure 3.2 Single neuron's schematic …………………………………………………...14

Figure 4.1 Array neuron's organization………………………………………………….16

Figure 6.1 Experimental network and result……………………………………………..33

Figure 6.2 VHDL simulation for partial configuration…………………………………..34

Figure 6.3 ISE project implementation ………………………………………………….35

Figure 6.4 ISE project configuration ……...…………………………………………….36

Figure 6.5 Implementation reports ………..……………………………………………..36

Figure 6.6 Mapping result ……………………………………………………………….37

Figure 7.2 Mapping result…………… ………………………………………………….41

Figure 7.3 SOLAR simulation snapshot ……...………………………...……………….43

Abbreviations

RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computers
FPGA	Field programmable gate array
PLD	Programmable logic device
CPLD	Complex programmable logic device
MUX	Multiplexer
LST	Least significant bit
MSB	Most significant bit
DLL	Dynamical link library
HEX	Hexadecimal
CPU	Central processing unit
LUT	Look-up table
VHDL	Very high-speed integrated circuit hardware description language
MIPS	Million instructions per second
PCI	Peripheral component interconnect
M68K	Motorola 68000 microcontroller
I/O	Input/Output
FF	Flip flop
BRAM	Block RAM
ANN	Artificial neural network

1. Introduction

In this report, dynamically reconfigurable neuron hardware architecture and its experiment setup using Nallatech board with Xilinx Virtex XCV800 Field Programmable Gate Array (FPGA) are described. The neuron structure is based on the modified Xilinx PicoBlaze [1] microcontroller. This modification allows the neuron to reconfigure itself during the runtime. Neurons have identical initial software code, are fully connected in a single chip, and can be expanded to a large multiple-chip system to build the 3D SOLAR [2] learning machine. The 3D SOLAR learning machine will be composed by 384 high-end VIRTEX XCV1000 FPGA chips. It is based on the main PCB board with 4 chips, and then PCB boards are both connected in the horizontal and vertical dimensions resulting in expanding SOLAR from 1D to 2D and then 3D. We are expecting the 3D SOLAR will have powerful processing ability based on its self-organizing algorithm and interconnections to perform tasks such as pattern recognition, prediction and modeling of unknown systems without being programmed beforehand.

This report is divided into eight parts. Section 2 introduces the PicoBlaze; Section 3 describes the neuron's 2x14 architecture based on the modified PicoBlaze; Section 4 deals with the interconnections between 28 neurons in the single FPGA chip; Section 5 explains the VHDL codes to implement SOLAR in a single FPGA chip; Section 6 talks about the experiment setup; Section 7 demonstrates how to change the

design, for example, change the structure from 2x14 to 4x7 neurons. Finally, Section 8 summarizes this work and describes some future work.

2. What Is PicoBlaze

The PicoBlaze soft microcontroller is an 8-bit Reduced Instruction Set Computer (RISC) microprocessor from Xilinx Corp., which supports an 8-bit data bus and 16-bit instruction bus. It has the Harvard architecture with separate data and instruction ports. It currently supports 49 instructions that operate within any one of several Xilinx CoolRunner™-II CPLDs (complex programmable logic device); it has 100% digital core with low power consumption and high-speed execution. Its speed will vary depending on executed instructions and the implementation platform. However, since it is tiny and has small instruction space, its functionality is not as strong as the traditional single chip computer. Although it has its own assembler, it does not have C/C++ compiler. So a user has to learn its assembly language, which is close to the 8086/8088 or M68K instruction set. For more information, refer to Xilinx webpage, (especially the introduction of the instruction set.). Because of these characteristics, the PicoBlaze is a compact structure suitable for the neural network implementation in hardware.

3. How to Build A SOLAR Neuron on Modified PicoBlaze

SOLAR has the self-organizing structure expressed in both its hardware and software part. In the hardware part, the neurons are fully interconnected in a single chip. So if we put 28 neurons in a single chip and assign two inputs from outside, then every neuron will have a 30 to 1 multiplexer to select its inputs either from the two inputs or from the outputs of other neuron (including itself); In the software part, every neuron has its own software can be dynamically updated from the PC through the PCI bus and through the inner neuron configuration bus. So we can load the individual self-organizing algorithm into every single neuron to give these neurons more wisdom to think.

3.1 Neuron Structure

The SOLAR neuron hardware architecture developed in this report is based on the PicoBlaze microprocessor. The PicoBlaze has been modified to adapt to the needs of SOLAR's self-organizing architecture. A block level of single neuron architecture based on the 2x14 array of neurons (so every neuron has 2x14+2=30 inputs) is shown on Fig.3.1. Each neuron module contains the circuits to be reprogrammed dynamically, and to execute its program without affecting other neurons' operations. The dynamics of neuron operation comes from the Dual port instruction memory. The instruction memory is originally a single port memory, which means that the instructions are fixed and can not be updated after loading. In the SOLAR architecture, every neuron's functionality must be changed based on the self organizing principle. So, I modified the instruction

memory to be dual ports. The dynamical programming is implemented by the dual-port 256x16-bit memory. The PicoBlaze reads the current program in one port, while the other port can be used to store the new program. The two ports of the dual-port Random Access Memory (RAM) operate independently, and the operation is via shared programming bus among all the neurons. Therefore, the self-reconfiguration process can be performed affecting only the current neuron. One port is for the microprocessor execution module to fetch instructions, and the other port can be used to dynamically update this neuron's parameters like the interconnection, threshold, functionality, and etc.

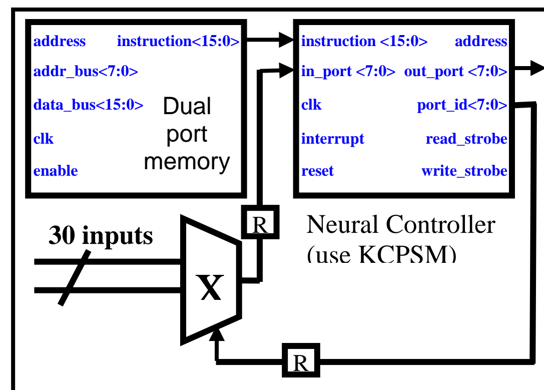

Fig.3.1 Single neuron's schematic

3.2 Neuron Programming

Within each single neuron, basis functions are used to form a complete non-linear space to mimic any function for every neuron, just like we can use sine/cosine, harr wavelet, etc to approximate any non-linear function. These functions used in SOLAR

contain ADD, SUB, MUL, DIV, INVR, QDRE, LOG2, QDRT, ROOT, SQRE [4]. More functions are under development including the popular sigmoid function. We have to consider the hardware implementation when we choose these basis functions since we have limited instruction space inside every single neuron to hold these basis functions.

The actual operation of each neuron is controlled by dynamically loaded structural information or parameters' values. The PicoBlaze assembler is used to implement neuron's functions at the initial stage. The actual operation of each neuron is controlled by dynamically loaded structural information or parameters' values. For instance, a simple PicoBlaze assembly code sub2.psm to implement subtraction of two neuron's inputs is shown as follows:

```
;This program implements a sub function
;
;function z = sub(x,y)
;z = max(0,x-y);
;
;By Y. Guo
;(c) 2004 FPGA lab Ohio U.

NAMEREG    s0,A
NAMEREG    s1,B

DISABLE INTERRUPT
start:
    INPUT A, 00
    INPUT B, 01
    SUB A,B
    JUMP NC, a_b
    LOAD A,00
a_b:
    OUTPUT A,01
    JUMP start
```

An assembly code can be written with notepad tools. The file is then saved with a .psm extension. Place the executive assembler file provided by Xilinx (C:\solar\hwsw\psm\KCPSM.exe) in the same directory as the program file, open a DOS window and navigate to the working directory that contains this programs. These programs are named as add.psm, divd.psm, exp2.psm, invr.psm, log2.psm, mult.psm, qdre.psm, qdrt.psm, root.psm, sqre.psm, sub.psm, and etc. For how to use the assembly language, you can refer to Xilinx data sheet. [3]. Then run the assembler to assemble the program. For instance,

Kcpsm sub2.psm <ENTER>

If there is no syntax error, we will obtain the “KCPSM complete” information as shown in Fig. 3.2. If there is syntax error, you can correct it based on the given error information. So if success, under the current directory, a binary VHDL program, named sub2.vhd will be created and stored.

```
Command Prompt
0C OUTPUT A, 02
0D JUMP start

PASS 5 - Writing reformatted PSM file to <psm_name>.fmt

PASS 6 - Writing assembler log file 'sub2.log'.

PASS 7 - Writing coefficient file 'sub2.coe'.

PASS 8 - Writing VHDL memory definition file 'sub2.vhd'.

PASS 9 - Writing memory definition files
'sub2.hex' and 'sub2.dec'.

KCPSM successful.

KCPSM complete.

C:\gyt\gateway\project\EXPAND~1\psm>
```

Fig. 3.2 Run assembler to assemble program

After translation from the assembler to the binary code, this binary code is written into the dual port memory by calling the Peripheral Component Interconnect (PCI) functions. The configuration time and contents of every single neuron can be controlled by the software outside the chip or via PC simulation. In the future, we can utilize the distributed memories at the edge of FPGA chip to pre-store some configurations for these neurons and in that case, the configuration and reconfiguration process will be finished by the chip itself.

The neuron inputs are obtained either from the primary inputs or other neurons' outputs via a 30 to 1 multiplexer (MUX) as introduced in the following section. The selection signal of the MUX to select the inputs to the current neuron is decided by the

content of the programming dual-port memory via execution of the programming commands for this particular neuron. For example, if a neuron is picking the outputs from another neuron, then the corresponding MUX selection signal is stored in this neuron's memory. Except for the MUX selection signal to determine neurons' interconnections, other configuration information including neurons' threshold, operation function are also obtained via training using Matlab software. These obtained configuration data is written into the Picoblaze neuron. So every neuron is configured differently from other neurons in most cases but they are all trained off-chip from the software simulation. Occasionally, some neurons are not fired during the software simulation and in that case, those neurons are "retired" from SOLAR.

4. How to Connect 28 Neurons

The single neuron architecture is expanded to an array of neurons in a single FPGA chip. The Xilinx Virtex XCV800 FPGA can contain an array of up to 28 neurons organized as shown in Fig.4.1 (Virtex XCV1000 FPGAs that will be used to build 3D learning machine will contain 64 neurons on each chip).

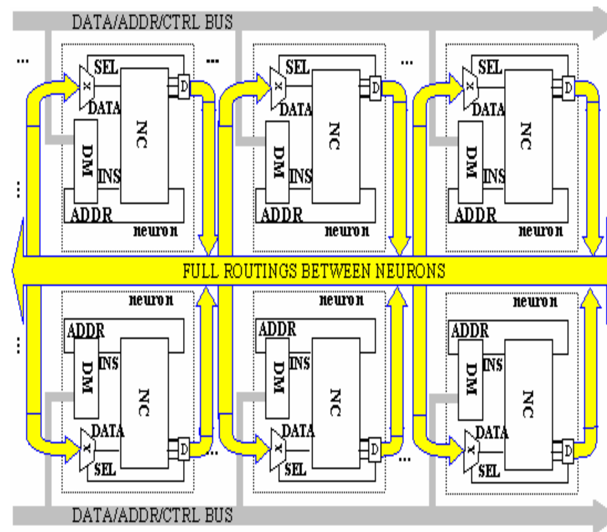

Fig. 4.1 Array neurons' organization

These neurons are fully interconnected via the connection bus and a 30 to 1 MUX, thus forming a local cluster of neurons. 30 to 1 multiplexer allows network to use 2 independent inputs, thus 28 and system inputs are fully interconnected. These numbers must be modified if the size of the network changes such that the multiplexer is of the size equal to the number of neurons plus the number of inputs. The full connections implementation mimics the dense connection scheme in the neighborhood neurons. A 3D expansion of these chips represents the sparse connections between remote neurons. These inter-chip connections scale linearly with the number of neurons added. The neurons connections are decided by each neuron based on its learning results. The programming contents can be dynamically updated via the configuration bus or set locally by a neuron. The configuration bus used to configure every single neuron is divided into 16-bit data, 8-bit address and 5-bit neuron selection buses. To demonstrate functionality of 28 neurons cluster, PCI interface controller is integrated to transfer the data/configuration via the PCI bus to neurons. In Fig. 4.1, the data/addr/ctrl bus is

connected to every neuron's programming memory to send the configuration information to every single neuron. The bus represents the fully interconnected bus between the neurons. How to use the ctrl bus to select particular neuron and how to configure it is explained in the following sections.

5. Explanation of the VHDL Code

Since the SOLAR neuron is implemented based on the PicoBlaze micro controller. Therefore, the neuron component in the pico_solar.vhd is called PicoBlaze and is defined as follows as a 2x14 network. (The full files are located in the hard drive in Rm.323 C:\solar\hwsw\nn_vote_4\pico_solar.vhd - initially located at Gateway P4 PC c:\PicoBlaze). The 4x7 network can be easily obtained from the 2x14 network and it is further explained in Section 7.

```
component PicoBlaze
Port (      clk : in std_logic;
            reset:in std_logic;
            data_out : out std_logic_vector(7 downto 0);
            data_in0 : in std_logic_vector(7 downto 0);
            data_in1 : in std_logic_vector(7 downto 0);
            data_in2 : in std_logic_vector(7 downto 0);
            data_in3 : in std_logic_vector(7 downto 0);
            data_in4 : in std_logic_vector(7 downto 0);
            data_in5 : in std_logic_vector(7 downto 0);
            data_in6 : in std_logic_vector(7 downto 0);
            data_in7 : in std_logic_vector(7 downto 0);
            data_in8 : in std_logic_vector(7 downto 0);
            data_in9 : in std_logic_vector(7 downto 0);
            data_in10 : in std_logic_vector(7 downto 0);
            data_in11 : in std_logic_vector(7 downto 0);
            data_in12 : in std_logic_vector(7 downto 0);
```

```
            data_in13 : in std_logic_vector(7 downto 0);
            data_in14 : in std_logic_vector(7 downto 0);
            data_in15 : in std_logic_vector(7 downto 0);
            data_in16 : in std_logic_vector(7 downto 0);
            data_in17 : in std_logic_vector(7 downto 0);
            data_in18 : in std_logic_vector(7 downto 0);
            data_in19 : in std_logic_vector(7 downto 0);
            data_in20 : in std_logic_vector(7 downto 0);
            data_in21 : in std_logic_vector(7 downto 0);
            data_in22 : in std_logic_vector(7 downto 0);
            data_in23 : in std_logic_vector(7 downto 0);
            data_in24 : in std_logic_vector(7 downto 0);
            data_in25 : in std_logic_vector(7 downto 0);
            data_in26 : in std_logic_vector(7 downto 0);
            data_in27 : in std_logic_vector(7 downto 0);
            data_in28 : in std_logic_vector(7 downto 0);
            data_in29 : in std_logic_vector(7 downto 0);
            ADDR_bus : in STD_LOGIC_VECTOR (7 downto 0);
            DATA_bus : in STD_LOGIC_VECTOR (15 downto 0);
            Enable   : in STD_ULOGIC
            );
end component;
```

In the above example, we built 2 rows and 14 columns SOLAR array. Every neuron can have up to 2x14=28 inputs from other neurons plus two inputs from outside (the number of external inputs is equal to the number of rows of the SOLAR array during the algorithm simulation). In the above example, we only have a single output for every neuron. Later example explains how to expand the output space. The following describes the network's signals.

clk signal is achieved from the CLKDLL - a clock delay-locked loop to minimize clock skew.

data_out is the output register of every single neuron, the output is connected to all the neurons' 30-1 mux inputs including itself.

data_in0 to **data_in29** are the input signals to the neuron's 30-1 MUX. Every neuron's input is from all neurons outputs including itself.

ADDR_bus is the dual-port memory configuration address bus connected to every neuron.

DATA_bus is the dual-port memory configuration data bus connected to every neuron.

Enable is the dual-port memory enable signal connected to every neuron.

The following describes the hardware operation:

Write a 28 bit data via PCI bus to register "ADDRDATAEN_BUS".

In the 28-bit data, bits 7 downto 0 (LSB) are the neuron's instruction address space; bits 23 downto 8 (LSB) are the neuron's instruction data space; bits 28 downto 24 (LSB) are the neuron's instruction memory enable signals. This 28-bit data is latched at the rising edge of clock with the permission of "ADDRDATAEN_BUS_WR". The memory enable signal is derived from the "ADDRDATAEN_BUS_WR" after four clock periods, to assure a sufficient setup/hold time for the instruction memory address and data signals.

```
process (RST, DSP_CLKi)
begin
    if RST='1' then
        Enable_bus_d <= (others => '1');
        Enable_bus <= (others => '1');
        ADDRDATAEN_BUS_WREN_d <= '0';
        ADDRDATAEN_BUS_WREN_d1 <= '0';
        ADDRDATAEN_BUS_WREN_d2 <= '0';
        ADDRDATAEN_BUS_WREN <= '0';
    elsif DSP_CLKi'event and DSP_CLKi='1' then
        Enable_bus_d <= ADDRDATAEN_BUS(28 downto 24);
```

```
            Enable_bus <= Enable_bus_d;
            ADDRDATAEN_BUS_WREN_d <= ADDRDATAEN_BUS_WR;
            ADDRDATAEN_BUS_WREN_d1 <= ADDRDATAEN_BUS_WREN_d;
            ADDRDATAEN_BUS_WREN_d2 <= ADDRDATAEN_BUS_WREN_d1;
            ADDRDATAEN_BUS_WREN <= ADDRDATAEN_BUS_WREN_d2;
    end if;
end process;
```

The enable signal for every single neuron is decoded by the "ADDRDATAEN_BUS" after two clock periods to ensure a sufficient setup/hold time to latch. This is implemented by the following code.

```
process (RST, Enable_bus,ADDRDATAEN_BUS_WREN)
begin
        if RST='1' then
                MY_ENA_BUS(1,1) <= '0';
                MY_ENA_BUS(1,2) <= '0';
                MY_ENA_BUS(2,1) <= '0';
                MY_ENA_BUS(2,2) <= '0';
                MY_ENA_BUS(3,1) <= '0';
                MY_ENA_BUS(3,2) <= '0';
                MY_ENA_BUS(4,1) <= '0';
                MY_ENA_BUS(4,2) <= '0';
                MY_ENA_BUS(5,1) <= '0';
                MY_ENA_BUS(5,2) <= '0';
                MY_ENA_BUS(6,1) <= '0';
                MY_ENA_BUS(6,2) <= '0';
                MY_ENA_BUS(7,1) <= '0';
                MY_ENA_BUS(7,2) <= '0';
                MY_ENA_BUS(8,1) <= '0';
                MY_ENA_BUS(8,2) <= '0';
                MY_ENA_BUS(9,1) <= '0';
                MY_ENA_BUS(9,2) <= '0';
                MY_ENA_BUS(10,1) <= '0';
                MY_ENA_BUS(10,2) <= '0';
                MY_ENA_BUS(11,1) <= '0';
                MY_ENA_BUS(11,2) <= '0';
                MY_ENA_BUS(12,1) <= '0';
                MY_ENA_BUS(12,2) <= '0';
                MY_ENA_BUS(13,1) <= '0';
```

```
                    MY_ENA_BUS(13,2) <= '0';
                    MY_ENA_BUS(14,1) <= '0';
                    MY_ENA_BUS(14,2) <= '0';
    else
      case Enable_bus is
            when "00010" => MY_ENA_BUS(1,1) <= ADDRDATAEN_BUS_WREN;
            when "00011" => MY_ENA_BUS(1,2) <= ADDRDATAEN_BUS_WREN;
            when "00100" => MY_ENA_BUS(2,1) <= ADDRDATAEN_BUS_WREN;
            when "00101" => MY_ENA_BUS(2,2) <= ADDRDATAEN_BUS_WREN;
            when "00110" => MY_ENA_BUS(3,1) <= ADDRDATAEN_BUS_WREN;
            when "00111" => MY_ENA_BUS(3,2) <= ADDRDATAEN_BUS_WREN;
            when "01000" => MY_ENA_BUS(4,1) <= ADDRDATAEN_BUS_WREN;
            when "01001" => MY_ENA_BUS(4,2) <= ADDRDATAEN_BUS_WREN;
            when "01010" => MY_ENA_BUS(5,1) <= ADDRDATAEN_BUS_WREN;
            when "01011" => MY_ENA_BUS(5,2) <= ADDRDATAEN_BUS_WREN;
            when "01100" => MY_ENA_BUS(6,1) <= ADDRDATAEN_BUS_WREN;
            when "01101" => MY_ENA_BUS(6,2) <= ADDRDATAEN_BUS_WREN;
            when "01110" => MY_ENA_BUS(7,1) <= ADDRDATAEN_BUS_WREN;
            when "01111" => MY_ENA_BUS(7,2) <= ADDRDATAEN_BUS_WREN;
            when "10000" => MY_ENA_BUS(8,1) <= ADDRDATAEN_BUS_WREN;
            when "10001" => MY_ENA_BUS(8,2) <= ADDRDATAEN_BUS_WREN;
            when "10010" => MY_ENA_BUS(9,1) <= ADDRDATAEN_BUS_WREN;
            when "10011" => MY_ENA_BUS(9,2) <= ADDRDATAEN_BUS_WREN;
            when "10100" => MY_ENA_BUS(10,1) <= ADDRDATAEN_BUS_WREN;
            when "10101" => MY_ENA_BUS(10,2) <= ADDRDATAEN_BUS_WREN;
            when "10110" => MY_ENA_BUS(11,1) <= ADDRDATAEN_BUS_WREN;
            when "10111" => MY_ENA_BUS(11,2) <= ADDRDATAEN_BUS_WREN;
            when "11000" => MY_ENA_BUS(12,1) <= ADDRDATAEN_BUS_WREN;
            when "11001" => MY_ENA_BUS(12,2) <= ADDRDATAEN_BUS_WREN;
            when "11010" => MY_ENA_BUS(13,1) <= ADDRDATAEN_BUS_WREN;
            when "11011" => MY_ENA_BUS(13,2) <= ADDRDATAEN_BUS_WREN;
            when "11100" => MY_ENA_BUS(14,1) <= ADDRDATAEN_BUS_WREN;
            when "11101" => MY_ENA_BUS(14,2) <= ADDRDATAEN_BUS_WREN;
            when others => null;
   end case;
  end if;
end process;
```

In general a neuron may have many input registers, but for neurons on layer0, we only use two - MY_NN_REGS(0,1), MY_NN_REGS(0,2) to represent neurons themselves. It is exactly the same as in the Matlab software simulation(

C:\solar\hwsw\main0.m) that we treat the inputs to be special neurons on layer0. Since the number of input features is equal to the number of neurons per layer, thus SOLAR structure can be changed depending on the input features. This read-in process for the “input” neurons are implemented by the following codes.

```
process (RST, DSP_CLKi)
begin
        if RST='1' then
    MY_NN_REGS(0,1) <= (others => '0');
         MY_NN_REGS(0,2) <= (others => '0');
         ADDRDATAEN_BUS  <= (others => '0');
        elsif DSP_CLKi'event and DSP_CLKi='1' then
                if MY_WR_REGS_01 = '1' then
                        MY_NN_REGS(0,1) <= DATA(7 downto 0);
                end if;
                if MY_WR_REGS_02 = '1' then
                        MY_NN_REGS(0,2) <= DATA(7 downto 0);
                end if;
                if ADDRDATAEN_BUS_WR = '1' then
                        ADDRDATAEN_BUS <= DATA(28 downto 0);
                end if;
        end if;
end process;
```

The neuron’s output is connected to an input of all other neurons. In addition, it could be read out for debugging and monitoring. This is implemented by the following code. The “and” operation is utilized to handle high-bit error happened very occasionally when huge data is read very rapidly and continuously via PCI bus. Possible reason is still not so clear and one conjecture is the interface function timing problem.

```
LAYRD: for i in 0 to LAYERS generate
 NEURD: for j in 1 to NEURONS generate
        DATA <=("000000000000000000000000"&MY_NN_REGS(i,j)) and
          "00000000000000000000000011111111"  when MY_RD_BUS(i,j)='1' else
           (others => 'Z');
```

```
  end generate NEURD;
end generate LAYRD;
```

The registers are addressed by the decoder logic as follows.

```
MY_WR_REGS_01<= '1' when WRITE_STROBE='1' and ADDRESS(5 downto 0)="000010" else '0';
MY_WR_REGS_02 <= '1' when WRITE_STROBE='1' and ADDRESS(5 downto 0)="000011" else '0';

PRGRAMMEM_RST <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="000100" else '0';
ADDRDATAEN_BUS_WR <= '1' when WRITE_STROBE='1' and ADDRESS(5 downto 0)="000101"
else '0';

MY_RD_BUS(0,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="000010" else '0';
MY_RD_BUS(0,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="000011" else '0';
MY_RD_BUS(1,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001100" else '0';
MY_RD_BUS(1,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001101" else '0';
MY_RD_BUS(2,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001111" else '0';
MY_RD_BUS(2,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001110" else '0';
MY_RD_BUS(3,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001010" else '0';
MY_RD_BUS(3,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001011" else '0';
MY_RD_BUS(4,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001001" else '0';
MY_RD_BUS(4,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="001000" else '0';
MY_RD_BUS(5,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011000" else '0';
MY_RD_BUS(5,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011001" else '0';
MY_RD_BUS(5,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011010" else '0';
MY_RD_BUS(6,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011011" else '0';
MY_RD_BUS(6,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011010" else '0';
MY_RD_BUS(7,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011110" else '0';
MY_RD_BUS(7,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011111" else '0';
MY_RD_BUS(8,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011101" else '0';
MY_RD_BUS(8,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="011100" else '0';
MY_RD_BUS(9,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010100" else '0';
MY_RD_BUS(9,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010101" else '0';
MY_RD_BUS(10,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010111" else '0';
MY_RD_BUS(10,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010110" else '0';
MY_RD_BUS(11,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010010" else '0';
MY_RD_BUS(11,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010011" else '0';
MY_RD_BUS(12,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010001" else '0';
MY_RD_BUS(12,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="010000" else '0';
MY_RD_BUS(13,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="110000" else '0';
MY_RD_BUS(13,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="110001" else '0';
MY_RD_BUS(14,1) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="110011" else '0';
MY_RD_BUS(14,2) <= '1' when READ_STROBE='1' and ADDRESS(5 downto 0)="110010" else '0';
```

The signal names and their description are listed in Table. 1.

REG	ADDR	DESCRIBTION
MY_WR_REGS_01	2	Layer 0 register 1Write signal
MY_WR_REGS_02	3	Layer 0 register 1Write signal
PRGRAMMEM_RST	4	Reset
ADDRDATAEN_BUS_WR	5	Addr, Data and Enable bus
MY_RD_BUS(0,1)	2	Layer 0 register 1 Read signal
MY_RD_BUS(0,2)	3	Layer 0 register 2 Read signal
MY_RD_BUS(1,1)	12	Layer 1 register 1 Read signal
MY_RD_BUS(1,2)	13	Layer 1 register 2 Read signal
MY_RD_BUS(2…14,1)	X	Similar to the above
MY_RD_BUS(2…14,2)	X	Similar to the above

(Addresses 0 & 1 are reserved for the main CSR and DMA counter)

Table.1 Register definition and description

In the original PicoBlaze, the programs are stored in a small instruction memory. But the original design uses a single-port memory, which means the user can not change the contents of the instruction memory once they are put into the FPGA chips since the only port is read by the PicoBlaze “CPU”. The detailed implementation can be referred to the PicoBlaze document [1]. In my design, I revised the original design to use dual port memory. In that case, we can reprogram the instruction memory without need to reload the whole design into the chip, and the process can be dynamically finished based on the self-organizing process for every single neuron without interfering other neuron’s

execution. The modified dual-port memory causes no difference in the original PicoBlaze implementation.

Many useful features of PicoBlaze architecture have not been fully utilized in this example. For instance, a PicoBlaze can have more than one single register output and this capability could be utilized if we have more information to exchange between neurons. The following code uses the output port "00000001" only. Later example uses two output port spaces.

```
data_registers: process(clk)
begin
if clk'event and clk='1' then
        if port_id(7 downto 0)="00000001" and write_strobe='1' then
                data_out <= data_outi;
        end if;
end if;
end process data_registers;
```

From the described VHDL code, it should be clear how to write the content of each neuron and how to send the created HEX file to every neuron.

6. How to Setup the Experiment

To experiment with the 28 SOLAR neurons based on PicoBlaze micro controllers, we have to have both the hardware and software platforms. The hardware platform contains the Ballynuey 2 PCI card and PC with PCI slot, which is easy to setup. The

software platform includes the operating system, Matlab, VC++, and etc. The following explains how to set the experiment.

The operating system is either WIN98/2000 or WIN XP. If we choose WIN98/2000, a DLL file facilitates to read/write registers and other I/O operation. The DLL file for Matlab is developed using MATLAB MEX programming. Basically, we need a MEX interface function to translate your C/C++ function IO data to MATLAB environment. This gives users an option to develop C/C++ function which can be executed in MATLAB environment for some time-consuming tasks in MATLAB. The DLL files are developed by myself combining both the C/C++ library from Nallatech. and Matlab MEX programming.

In this project, we use the MATLAB algorithm simulation for SOLAR. With these DLL, we can easily operate the PCI card within Matlab environment. If we choose to work within WIN XP, the current development library provided by the Nallatech uses new C/C++ interface functions and there is a potential problem to develop necessary Matlab DLL files. So we may need to use C/C++ function without Matlab environment. Actually, there is no big issue here if you are familiar with C/C++ environment. In the following, I assume we are using WIN2000 which is the environment that I used. The developed DLL contains:

matOpenDIMEBoard.dll

matCloseDIMEBoard.dll

matResetDIMEBoard.dll

matviDIME_ReadRegister.dll

matviDIME_WriteRegister.dll

matviDIME_DMARead.dll

matviDIME_DMAWrite.dll

…

To create these DLL, you need a MEX header to connect Matlab and C/C++. This MEX header called mexFunction as following example. The rest of the function is your implementation function as DIME_WriteRegister function in this example. Inside this function, you can implement whatever you want. In my code, I use viDIME_WriteRegister function provided by Nallatech in the Nallatech CD in Rm 323. To compile this function, you need go to Matlab console, type mex to first configure your compiler, then you can compile your C/C++ function in Matlab environment. More functions can be provided if needed.

```
#include "mex.h"
#include "dimesdl.h"
#include "vidime.h"

void DIME_WriteRegister(double * handle,double * addr,double * data,double *flag)
{
	DIME_HANDLE CurrHandle;
	if((CurrHandle=GetDIMEHandle())==NULL)
	{
		printf("Open Virtex Board failed when checking handle status!");
		return;
	}
   *flag=viDIME_WriteRegister(CurrHandle,(DWORD)*addr,(DWORD)*data,5000);
}
```

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
   double *handle;
         double *addr;
         double *data;
         double *flag;

         if (nrhs != 3) {
                  mexErrMsgTxt("three arguments required - handle addr & data.");
   }
         else if(nlhs > 1)
                  mexErrMsgTxt("Too many output arguments");

   handle = mxGetPr(prhs[0]);

         addr = mxGetPr(prhs[1]);

         data = mxGetPr(prhs[2]);

   plhs[0]=mxCreateDoubleMatrix(1,1,mxREAL);
   flag=mxGetPr(plhs[0]);

   DIME_WriteRegister(handle,addr,data,flag);
}
```

These DLL files are developed for the basic applications including register read/write, DMA transferring, DIME board operation, and etc. In the future application, more DLLs may be developed based on the modified design. Among these DLLs, the matResetDIMEBoard.dll is not designed to reset your design but to reset the PCI interface FIFO, control state machine, and etc. To reset the whole board, you not only need to reset the PCI interface but also need to reset your design. To reset the PCI interface, basically, three reset functions are needed as follows:

DIME_PCIReset(hDIME);

DIME_VirtexReset(hDIME);

DIME_SystemReset(hDIME);

(here, hDIME is the returned handle for the board)

To reset your design, there are many ways to do it. You can assign a reset register or you can create a reset pulse based on simple read/write logic. Again, to reset your design, you have to add one more function to create the reset DLL to reset the whole board. In my design, the RESET signal is associated with the system reset. So there is no additional reset pulse generated. The mex code is as follows.

```
#include "mex.h"
#include "dimesdl.h"
#include "vidime.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
mxArray *prhs[])
{
     DIME_HANDLE hDIME;
     if (nrhs != 0) {
          mexErrMsgTxt("No arguments required.");
    }
    if((hDIME=GetDIMEHandle())==NULL)
     {
          printf("Open Virtex Board failed when checking
handle status!");
          return;
     }
     DIME_PCIReset(hDIME);
     DIME_SystemReset(hDIME);
    DIME_VirtexResetEnable(hDIME);
     printf("Virtex Board Reset totally.\n");
     }
```

After the SOLAR training in software, we have achieved the optimal configuration results of the network neurons. We can use Matlab to create the HEX file based on the learned results to configure the instruction memory for every single neuron on the Picoblaze microcontroller and to use these HEX files to configure every single neuron. How to transfer the HEX file to every single neuron is explained in later content. The HEX file is the instruction file created by the provided assembler (KCPSM.exe) from Xilinx, as introduced in section 3.2. Currently, there are eleven learning functions to use in SOLAR

'ADD.HEX','SUB.HEX','MULT.HEX','DIVD.HEX','SQRE.HEX','ROOT.HEX',
'LOG2.HEX','INVR.HEX','EXP2.HEX','QDRT.HEX','QDRE.HEX'.

Although these functions are HEX files, we can change some values inside them (for detailed HEX file structure, refer to the recent PicoBlaze document). For instance, 8100E000C014A101A000 is a simple instruction to add two ports together. We can change "C014" to "C016" to do the subtraction instead of addition. This configuration work has been done in Matlab since our neurons are learning and parameters are updated. Using Matlab program, we can change the parameters at the right memory location in PicoBlaze.

The complete developed Matlab file is merged with the whole system to dynamically configure the neurons based on the training results. The following Matlab file may be used as a starting point. This file is easy to understand. The main function is

to configure the chip, load different instruction memory for every single neuron, and read back the voting results. In the example code, VirtexHandle is the board memory space and ADDRESS(5 downto 0) is used to select which neuron to configure after the address signal is further decoded.

```
close all;
clear all;

load TRAININGRESULT;

input1=[64];
input2=[56];

MaxNeuron=6;

test=TEST; %function selection
threshold=NEURONSTHRESHOLD; %threshold
connection1= NEURONSCONNECTION1; %connection1
connection2= NEURONSCONNECTION2; %connection2

hexfiles={'ADD.HEX','SUB.HEX','MULT.HEX','DIVD.HEX','SQRE.HEX', ...
'ROOT.HEX','LOG2.HEX','INVR.HEX','EXP2.HEX','QDRT.HEX','QDRE.HEX'};

fid_s = fopen('send2neuron.txt','W');

VirtexHandle=matOpenDIMEBoard;

succ=matConfigDIMEBoard(VirtexHandle, 'pico_solar.bit');

for neuron=1:MaxNeuron

   hexfile_pointer=test(neuron);

   fid_r=fopen(hexfiles{hexfile_pointer},'r');

   writedata=sprintf('%s801000',dec2hex(neuron+1,2));
   flag=matviDIME_WriteRegister(VirtexHandle,5,hex2dec(writedata));
   writedata=sprintf('%sA1%s02',dec2hex(neuron+1,2),dec2hex(connection2(neuron),2))
   flag=matviDIME_WriteRegister(VirtexHandle,5,hex2dec(writedata));
```

```
    writedata=sprintf('%s0F%s03',dec2hex(neuron+1,2),dec2hex(threshold(neuron),2));
    flag=matviDIME_WriteRegister(VirtexHandle,5,hex2dec(writedata));

    i=0;
    t=fgets(fid_r);
    tpre=t;

    while(isempty(strfind(t,'0000')) | isempty(strfind(tpre,'0000')) )

    if isempty(strfind(t,'8010')) & isempty(strfind(t,'A000')) & ...
                isempty(strfind(t,'A101')) & isempty(strfind(t(1:2),'0F'))
            writedata=sprintf('%s%s%s',dec2hex(neuron+1,2),t(1:4),dec2hex(i,2));
            flag=matviDIME_WriteRegister(VirtexHandle,5,hex2dec(writedata));
        end
      end
end

reset=matviDIME_ReadRegister(VirtexHandle,4);

for i=1:length(input1)

    flag=matviDIME_WriteRegister(VirtexHandle,2,input1(i));
    flag=matviDIME_WriteRegister(VirtexHandle,3,input2(i));
    neuron2(i)=matviDIME_ReadRegister(VirtexHandle,7)
    neuron3(i)=matviDIME_ReadRegister(VirtexHandle,8)
    neuron4(i)=matviDIME_ReadRegister(VirtexHandle,9)
    neuron5(i)=matviDIME_ReadRegister(VirtexHandle,10)
    neuron6(i)=matviDIME_ReadRegister(VirtexHandle,11);
    neuron7(i)=matviDIME_ReadRegister(VirtexHandle,12);

    vote2(i)=matviDIME_ReadRegister(VirtexHandle,35)
    vote3(i)=matviDIME_ReadRegister(VirtexHandle,36)
    vote4(i)=matviDIME_ReadRegister(VirtexHandle,37)
    vote5(i)=matviDIME_ReadRegister(VirtexHandle,38)
    vote6(i)=matviDIME_ReadRegister(VirtexHandle,39);
    vote7(i)=matviDIME_ReadRegister(VirtexHandle,40);

end

matCloseDIMEBoard;
```

Example

In this part, I used the neuron structure obtained after learning, shown in the left part of Fig. 6.1. I prototyped the simple SOLAR architecture containing 28 neurons onto a single VIRTEX FPGA chip. In this example, the configuration of 6 neurons among the 28 neurons is shown in Fig. 6.1.

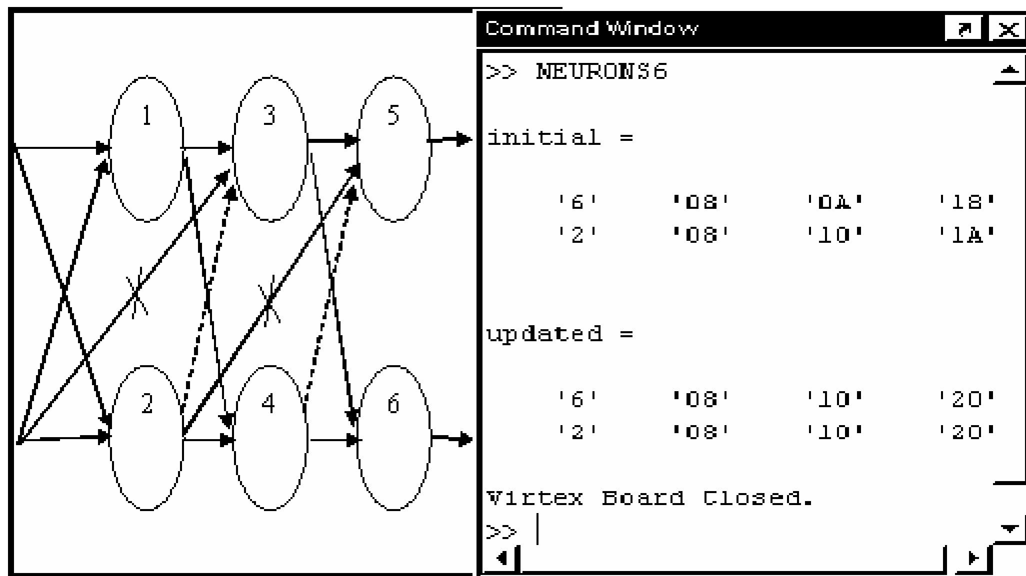

Fig.6.1 Experimental network and result

The initial connections are shown in solid lines. In this simplified example, every neuron simply adds two inputs together. For instance, the neuron 1 adds inputs 1 and 2 to its content; the neuron 3 adds input 2 and the output of the neuron 1; the neuron 5 adds the outputs of neuron 2 and 3, etc. Then, the connections of the neurons 3 and 5 are dynamically reconfigured, as shown by the dotted line. The updated neuron 3 has inputs as the outputs from the neurons 1 and 2, and the neuron 5 has inputs as the outputs from the neurons 3 and 4. The results can be read out from the chip via PCI bus, as shown in the Matlab console. In the inserted Matlab command console in Fig.6.1, “initial” values include primary input values (6 and 2) and neuron outputs of 6 neurons in 3 layers, while

"updated" values show inputs and neuron outputs after dynamical reconfiguration. We developed the Matlab DLLs to implement the I/O functions including read and write. The results from the VHDL simulation results is shown in Fig. 6.2. In the waveform, "Enable_bus" represents the neuron selection signal. It selects a particular neuron to be configured at a certain time. For instance, "Enable_bus" value 4 means that we are updating the contents of the neuron 3 and "Enable_bus" value 6 means we are updating the contents of the neuron 5. In this way, any neuron's configuration information can be updated without affecting other neurons. Once the configuration process for all neurons is over, the outputs from neurons are stable and ready to be read out. In this example, only neurons 3 and 4 connections are updated when the "Enable_bus" has value of 4 and 6, respectively. Based on this experiment's inputs valued as 6 and 2, the initial outputs from the neurons 1 to 6 are HEX 8, 8, A, 10, 18, respectively. After the connections are updated as shown in the left of Fig. 6.2, the neuron outputs are updated to be hex 8, 8, 10, 10, 20, 20, which agree with a manual calculation.

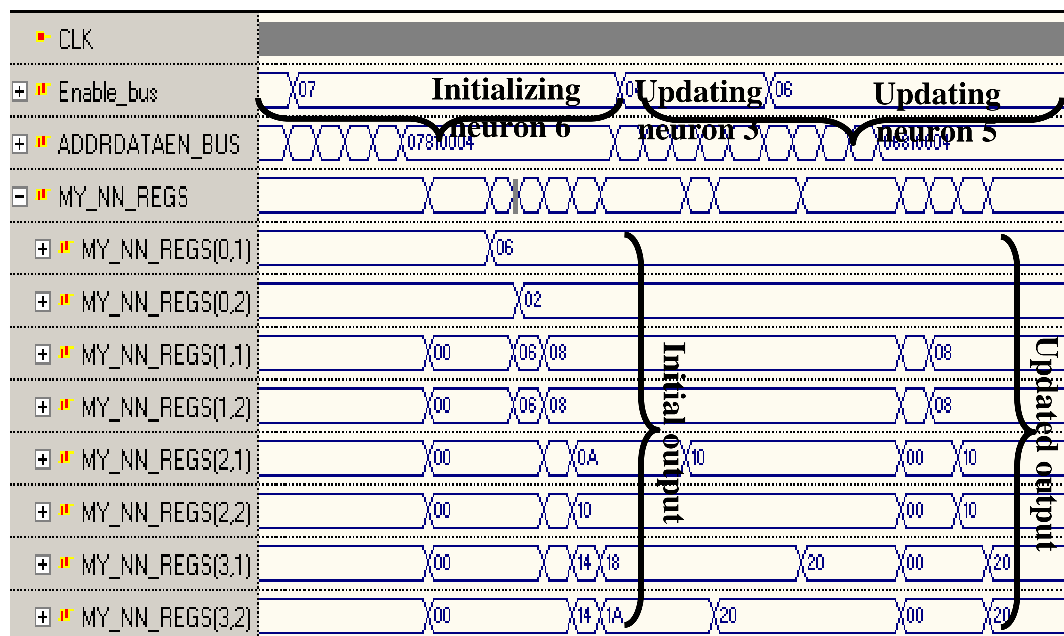

Fig. 6.2 VHDL simulation for partial configuration

The design has been described in VHDL and the simulation verifies the design. But we have to implement this design in FPGA. Real FPGA execution may differ from the functional simulation because of the clock, delay and etc. Since the whole design files are developed in RTL level. We can easily obtain the FPGA configuration file using XILINX ISE software. The usage of ISE can refer to the labs of EE414/514 [4]. A snapshot of the project in ISE is shown in Fig. 6.3.

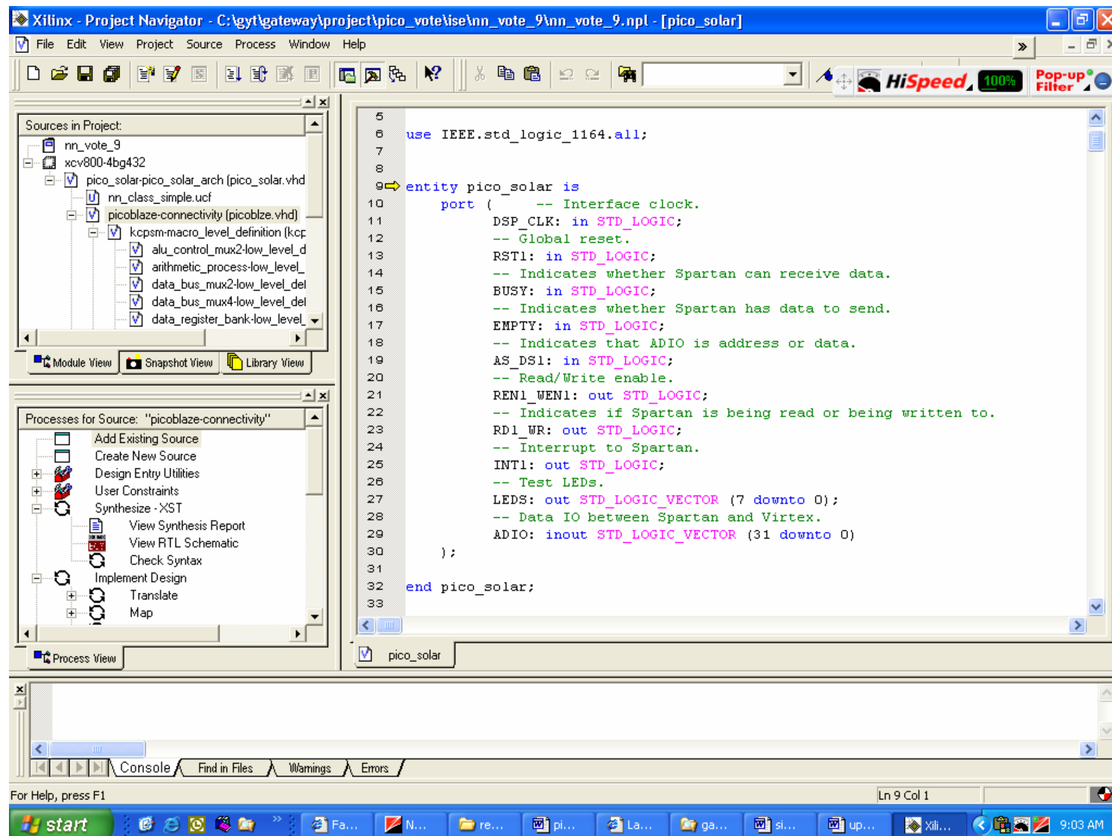

Fig.6.3 ISE project implementation

To implement the design, JTAB clock is assigned during the last step (Create programming file). See Fig. 6.4.

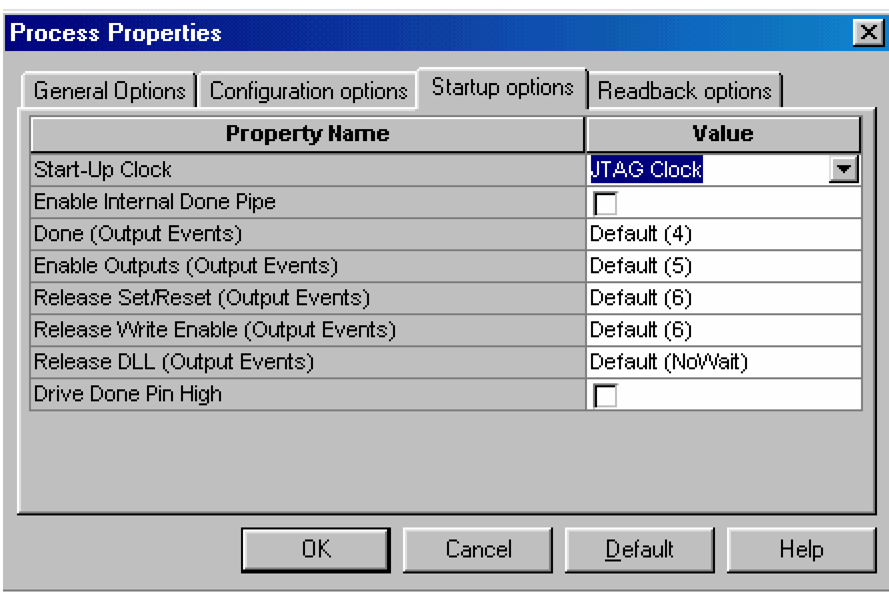

Fig.6.4 ISE project configuration

The implementation results are summed up in Fig. 6.5

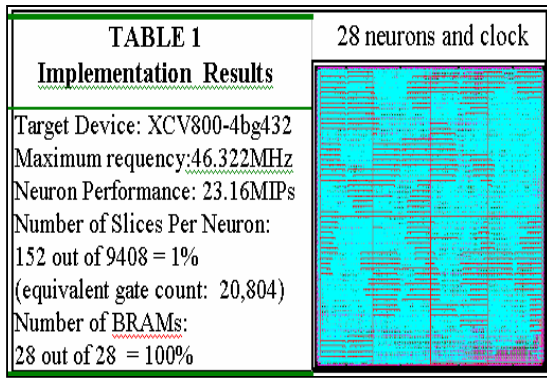

TABLE 1 Implementation Results	28 neurons and clock
Target Device: XCV800-4bg432 Maximum requency:46.322MHz Neuron Performance: 23.16MIPs Number of Slices Per Neuron: 152 out of 9408 = 1% (equivalent gate count: 20,804) Number of BRAMs: 28 out of 28 = 100%	

Fig. 6.5 Implementation reports

Based on the implementation results, it is concluded that this neural network architecture realizes a maximum parallel instruction throughput of 23.16x28 MIPs with 28 fully connected neurons. The neuron number is limited to 28 since there are only 28 BRAM modules on the chip, although there are still some other resources remained unused. If we can lower the neuron's program memory size, we can put more neurons on

a single chip to overcome the BRAM bottleneck. Due to the parallel processing in hardware and the distributed network memory, the speed improvement using FPGA implementation is supposed to be significant comparing to the software simulation because of the parallel processing array – SOLAR is aimed at adopting multiple processing elements to achieve "real time" application. In this experiment, most of the time is consumed in the interface when data is transmitted to SOLAR (2*14) and read back. In addition, the current software simulation is based on 1.1 Ghz CPU clock, which is unbeatable by the low-end FPGA chip. Due to above reasons, in this environment, speed improvement is not obvious or even hardware computer is slower than the software simulation because of the I/O operation.

The mapping result is shown in Fig. 6.6. It illustrates how the neurons are distributed inside the chip after the mapping process. Every single neuron occupies a compact and concentrated logic area. The compactness depends on the structure-oriented hardware design. For instance, we adopt LUTs and dedicated multiplexer module in addition to the optimized PicoBlaze structure to build every single neuron.

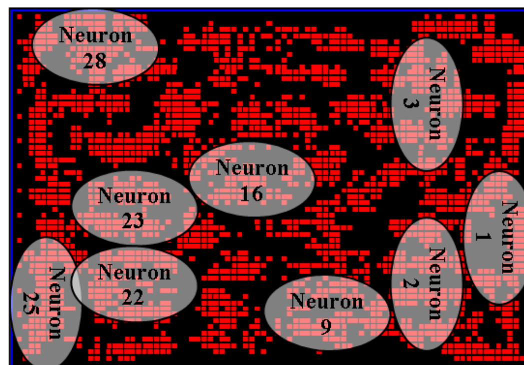

Fig. 6.6 Mapping result

7. How to Modify the Design From 2x14 to 4x7 Structure

To change the SOLAR structure from 2x14 to 4x7, in the VHDL file (C:\solar\hwsw\nn_vote_4\pico_solar.vhd), several modifications should be made. First, the number of layers "LAYERS" and number of neurons per layer "NEURONS" should be changed as follows

```
constant LAYERS :integer := 7;
constant NEURONS :integer := 4;
```

Second, since there are 4 inputs to the network plus 28 inputs to 28 neurons, the PicoBlaze component in the VHDL file is changed as follows.

```
component PicoBlaze
Port ( clk : in std_logic;
        reset:in std_logic;
        data_out : out std_logic_vector(7 downto 0);
        vote_out : out std_logic_vector(7 downto 0);
        data_in0 : in std_logic_vector(7 downto 0);
        data_in1 : in std_logic_vector(7 downto 0);
        data_in2 : in std_logic_vector(7 downto 0);
        data_in3 : in std_logic_vector(7 downto 0);
        data_in4 : in std_logic_vector(7 downto 0);
        data_in5 : in std_logic_vector(7 downto 0);
        data_in6 : in std_logic_vector(7 downto 0);
        data_in7 : in std_logic_vector(7 downto 0);
        data_in8 : in std_logic_vector(7 downto 0);
        data_in9 : in std_logic_vector(7 downto 0);
        data_in10 : in std_logic_vector(7 downto 0);
        data_in11 : in std_logic_vector(7 downto 0);
        data_in12 : in std_logic_vector(7 downto 0);
        data_in13 : in std_logic_vector(7 downto 0);
        data_in14 : in std_logic_vector(7 downto 0);
        data_in15 : in std_logic_vector(7 downto 0);
        data_in16 : in std_logic_vector(7 downto 0);
        data_in17 : in std_logic_vector(7 downto 0);
        data_in18 : in std_logic_vector(7 downto 0);
```

```
        data_in19 : in std_logic_vector(7 downto 0);
        data_in20 : in std_logic_vector(7 downto 0);
        data_in21 : in std_logic_vector(7 downto 0);
        data_in22 : in std_logic_vector(7 downto 0);
        data_in23 : in std_logic_vector(7 downto 0);
        data_in24 : in std_logic_vector(7 downto 0);
        data_in25 : in std_logic_vector(7 downto 0);
        data_in26 : in std_logic_vector(7 downto 0);
        data_in27 : in std_logic_vector(7 downto 0);
        data_in28 : in std_logic_vector(7 downto 0);
        data_in29 : in std_logic_vector(7 downto 0);
        data_in30 : in std_logic_vector(7 downto 0);
        data_in31 : in std_logic_vector(7 downto 0);
        ADDR_bus : in STD_LOGIC_VECTOR (7 downto 0);
        DATA_bus : in STD_LOGIC_VECTOR (15 downto 0);
        Enable   : in STD_ULOGIC
        );
end component;
```

Third, the I/O address need to be changed to contain two more null neurons (for the input features 3 and 4), and following that, the I/O address decoder should be changed. (refer to the provided 4x7 source code C:\solar\hwsw\nn_vote_4\pico_solar.vhd)

```
MY_WR_REGS_01<= '1' when WRITE_STROBE='1' and ADDRESS(5 downto
0)="000010" else '0';
MY_WR_REGS_02 <= '1' when WRITE_STROBE='1' and ADDRESS(5 downto
0)="000011" else '0';
MY_WR_REGS_03<= '1' when WRITE_STROBE='1' and ADDRESS(5 downto
0)="000100" else '0';
MY_WR_REGS_04 <= '1' when WRITE_STROBE='1' and ADDRESS(5 downto
0)="000101" else '0';

process (RST, DSP_CLKi)
begin
      if RST='1' then
        MY_NN_REGS(0,1) <= (others => '0');
        MY_NN_REGS(0,2) <= (others => '0');
        MY_NN_REGS(0,3) <= (others => '0');
```

```
    MY_NN_REGS(0,4) <= (others => '0');
    ADDRDATAEN_BUS  <= (others => '0');
  elsif DSP_CLKi'event and DSP_CLKi='1' then
      if MY_WR_REGS_01 = '1' then
          MY_NN_REGS(0,1) <= DATA(7 downto 0);
      end if;
      if MY_WR_REGS_02 = '1' then
          MY_NN_REGS(0,2) <= DATA(7 downto 0);
      end if;
      if MY_WR_REGS_03 = '1' then
          MY_NN_REGS(0,3) <= DATA(7 downto 0);
      end if;
      if MY_WR_REGS_04 = '1' then
          MY_NN_REGS(0,4) <= DATA(7 downto 0);
      end if;
      if ADDRDATAEN_BUS_WR = '1' then
          ADDRDATAEN_BUS <= DATA(28 downto 0);
      end if;
  end if;
end process;
```

Finally, the Matlab file (C:\solar\hwsw\NNorgVote_hardwareconf.m) used to acquire the data also needs to be changed based on the updated I/O address. The design has been described in the VHDL and synthesized using the Xilinx XST 6.2 and the Xilinx Alliance tools for place and route. The implementation results are summed up in Table.6.1.

Implementation Results Per Neuron			
Selected Device :			v800bg432-4
Maximum Frequency:			45.568MHz
Neuron Performance:			22.784MHz
Number of Slices:	163 out of	9408	1.73%
Number of Slice FFs:	97 out of	188	0.5%
Number of 4 input LUTs:	290 out of	1881	1.53%
Number of TBUFs:	73 out of	9408	0.75%
Number of BRAMs:	1 out of	28	3.57%

Table 7.1 Implementation reports

The mapping result is shown in Fig.7.2. As before, every single neuron occupies a compact and concentrated logic area.

Fig. 7.2 Mapping result

The following example presents the Matlab training based on a set of real world data from the Iris dataset [5]. The Iris dataset contains data of 3 classes with 4 features and each class has 50 samples. So totally there are 150 training samples for each feature. In this simulation, half of data was used for training and the other half used for testing based on random data order. 7 layers of neurons (28 neurons total) plus 4 inputs are used. The simulation result is encouraging with probability 98.67% and presented in [6].

```
>> neurons{7}.func
ans =
    'half'    'half'    'f1'
>> neurons{7}.threshold
ans =
   52.7944
```

```
>> neurons{7}.features
ans =
    [1]    [3]
>> neurons{8}.func
ans =
    'ident'    'half'    'f2'
>> neurons{8}.threshold
ans =
   74.0095
>> neurons{8}.features
ans =
    [4]    [3]


function z = f1(x,y)
xt = bitshift(x,-1);
yt = bitshift(y,-1);
z=xt+yt;

function z = f2(x,y)
z = max(0,x-y);

  Neuron 7 threshold: 52.7944 = (hex 35)
Neuron 8 threshold: 74.0095 = (hex 4A)

>> Scaled_Testing_data(:, 1:30)

ans =

  Columns 1 through 10

   243   228   210   236   239    14    35    69   113   216
    58   195   113   189   234    89    51    50   238   134
   155   116   157    44   104   208    50     3   118    51
   123     4   202   103   228     2   154   191   106   172

  Columns 11 through 20

   214   213    77    77    96   126   209    87   214   140
     5   129    48   139   220   230   169   137   145   113
   174   181    49    38   218   210    87   186    94   178
    96   109   174   178   152   165    73    78   180   159

  Columns 21 through 30
```

203	44	224	228	72	149	110	136	200	203
244	250	189	50	119	107	57	164	174	15
134	69	34	76	16	132	148	53	117	154
225	64	2	169	253	85	194	96	145	12

VHDL simulation is shown in Fig. 7.3.

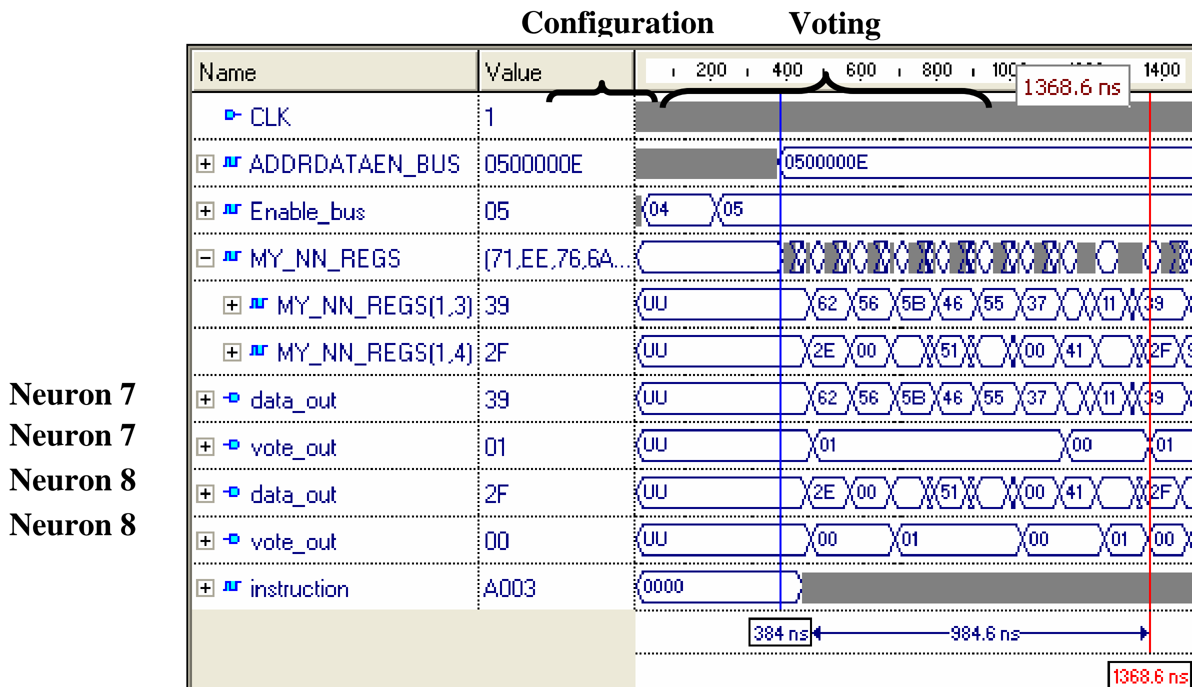

Fig. 7.3 SOLAR simulation snapshot

After the training stage, the 28 neurons will have different values based on their parameters including the connections, functions, threshold and etc. This information will be transferred into the chip during the voting stage. The VHDL simulation provides further reference to the hardware execution. In Fig.7.3, we provide a snapshot of the hardware simulation. In this simulation, the neurons 7 and 8 in the first layer of the 4x7 network are configured based on the training results.

Some parameters for the neurons 7 and 8 after the Matlab training are shown as follows:

neurons{7}.func={ 'half', 'half', 'f1'}

neurons{7}.threshold = 52.7944

neurons{7}.features = [1] [3]

neurons{8}.func={ 'ident' 'half' 'f2'}

neurons{8}.threshold = 74.0095

neurons{8}.features = [4] [3]

These results can be translated as

Neurons{7}.output = f1(half (Neurons{1}.output), half (Neurons{3}.output)) =

Neurons{1}.output / 2 / 2 + Neurons{3}.output / 2 / 2=

Neurons{1}.output / 4 + Neurons{3}.output / 4

Neurons{8}.output = f1(half (Neurons{4}.output), half (Neurons{3}.output))=

MAX((Neurons{4}.output / 2 - Neurons{3}.output / 2), 0)

In the simulation waveform, after the configuration, testing data are sent to the 4 input neurons in the 4x7 network. The neuron outputs are stored in their individual output registers. For example, “MY_NN_REGS(1,3)” and “MY_NN_REGS(1,4)” represent the outputs from the neurons 7 and 8. The output result of a certain neuron is used to compare with its corresponding subspace thresholds to determine the voting

results. In the waveform, “neuron 7 vote” and “neuron 8 vote” are the voting results for neuron 7 and 8 respectively.

In this simulation waveform, the current testing features are hex values 71, EE, 76, 6A and they are the outputs from the 4 input neurons. We treat them as “input” neurons 1-4. So neuron 1 has output 71, neuron 2 has output EE and etc. Since the neuron 7 is connected to neurons 1 and 3, then the output from neuron 7 is hex value (71/4+76/4)=39. In the MATLAB simulation result, it is seen that neuron 7 takes two half inputs, left bit shift 1 (equal to divide 2) and then add them together.

```
>> neurons{7}.func
ans =
   'half'    'half'    'f1'
```

and f1 function is as following

```
function z = f1(x,y)
xt = bitshift(x,-1);
yt = bitshift(y,-1);
z=xt+yt;
```

Thus [(neuron1 output) /2 + (neuron3 output) /2] /2 = (71/4+76/4)=39, which can be verified in the VHDL simulation (shown as neuron 7 result).

Similarly, the output from the neuron 8 is hex value MAX(6A -76/2)=2F since neuron 8 is connected to the neurons 4 and 3. These output results are compared to their subspace threshold which are 52.7944 (or hex 35) and 74.0095 (or hex 4A) for th3e neurons 7 and 8 respectively. Since the output from the neuron 7 is hex value 39 which

is greater than its threshold hex value 35, its voting output is 1; since the output from the neuron 8 has hex value 2F which is less than its threshold hex value 4A, its voting output is 0. All of the testing data are passed through the SOLAR feed forward network and voting output from each neuron are collected to determine the final voting results as 0 or 1.

8. Summary and Future Work

SOLAR represents a new idea in hardware design of artificial neural networks (ANNs). It is a modular and expandable system. It also defines a new breed of dynamically reconfigurable architectures that can dynamically reconfigure themselves based on information included in the input data. This presented architecture is a novel dynamically reconfigurable (via dual-port memory) neural network implementation that used simple general-purpose processor (KCPSM) architecture. Firstly, it has a regular expandable parallel architecture. Therefore, its speed and learning abilities can be greatly improved comparing to the software simulation. Secondly, it has data-driven self-organizing learning structure based on a new self-organizing learning algorithm. Furthermore, design flexibility is attained by exploiting the features of self-reconfigurable neuron units. Finally, hardware re-configurability is achieved in this self-organizing learning array by involving reconfigurable routing modules. According to the implementation results, this neuron architecture realizes a maximum parallel instruction throughput of 648 MIPs with 28 fully connected neurons. System performance increases as more neurons are connected. The PicoBlaze for Virtex-II Series FPGAs reaches

performance levels of up to 55 MIPS. With up to 336 PicoBlaze nodes on the chip SOLAR will reach the performance of 18.5 GIPS on a single chip. These numbers could be a little different within the same chip based on the different structure. So its parallel processing ability can be further improved. With the popular PowerPC on VIRTEX II Pro FPGAs used as the main CPU, the PicoBlaze neurons can be used as slave peripherals to further improve the throughput for system on a chip implementation of neural networks. Hence it can be of a practical use for the embedded hardware applications in signal processing, wireless communications, multimedia systems, data networks, and so forth.

In the neuron's design, the number of neurons/chip is limited by the number of BRAMs, so the total logic utilization is comparatively low. The remaining LUT RAM will be used for the communication between neurons on other chips and to reuse the same PicoBlaze for several neurons, to fully utilize the hardware resource and accommodate more neurons in a single FPGA chip. In this way, the design seems device-dependent, since it may need to carefully arrange additional resource for different FPGAs. Each device type has to be individually optimized. In scaling our design to 3D system, design optimization will be necessary in order to accommodate more neurons in a single FPGA chip.

This report describes the SOLAR implementation based on PicoBlaze in a single chip. I hope this experimental setup can be helpful to prototype development of novel neuron and routing architectures for the 3D SOLAR.

Reference

[1] http://www.xilinx.com/publications/xcellonline/xcell_45/xc_PicoBlaze45.htm

[2] J. A. Starzyk, Y. Guo, and Z. Zhu, "Dynamically Reconfigurable Neuron Architecture for the Implementation of Self-Organizing Learning Array", Proc. 18th Int. Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, April 26– 30, 2004.

[3] http://direct.xilinx.com/bvdocs/appnotes/xapp213.pdf

[4] http://www.ent.ohiou.edu/~starzyk/network/Class/ee514/index.html

[5] "UCI Machine Learning Repository," ftp://ftp.ics.uci.edu/pub/machine-learning- databases/iris/

[6] J. A. Starzyk, Y. Guo, And Z. Zhu, "Dynamically Reconfigurable Neuron Architecture For The Implementation Of Self-Organizing Learning Array," submitted to International Journal Of Embedded System".