FPGA Implementation of Neural Networks

Presented by Nariman Dorafshan

Main Contents

• Neural Networks: A Brief Introduction
• Practical Example: Character Recognition using MATLAB
• VHDL: An Instant Approach
• A Neuron Implementation
• Activation Function Implementation
• Overview of the Implemented Network

FPGA Implementation of Neural Networks
Semnan University – Spring 2012

Neural Networks

• Introduction to Neural Networks
  o Computational Intelligence
  o Artificial Neuron
  o Network Structures
    • MLP & Back-propagation
    • Regulation
• Practical Example: Character Recognition
  o Problem Definition
  o Training the Network using MATLAB

FPGA Implementation of Neural Networks
Semnan University – Spring 2012

Computational Intelligence

• A branch of AI, which implements the nature inspired algorithms is called computational intelligence.
• Its goal is to find ways for System Modeling, Pattern Recognition, Series Prediction, Optimization, etc.
• Understanding the performance of brain inspired the creation of artificial neural networks (ANNs) as an information processing system to generate human recognition models based on biological nervous systems.
Biological Neuron

• A single biological neuron is composed of three major parts including the cell body (called soma), the axon, and the dendrite.
• The cell body of a neuron is connected to the dendrite of a neighboring neuron.
• Signal communications between neurons are continuously generated which is delivered from one neuron to the others by firing an electrical signal generated through a chemical reaction.
• The other neurons receive the signal through the interfaces with the neighboring neurons, referred to as a synapse.
• This system is capable of learning, recalling, and generating output corresponding to external signals.

Artificial Neural Networks (ANNs)

• An artificial neuron network (ANN) is an information-processing system that has certain performance characteristics in common with biological neural networks.
• Based on examples, together with some feedback from a “teacher,” we learn easily to recognize the letter A or distinguish a cat from a bird.
• More experience allows us to refine our responses and improve our performance.
• Although eventually, we may be able to describe rules by which we can make such decisions, these do not necessarily reflect the actual process we use.
• Even without a teacher, we can group similar patterns together.

Artificial Neuron

• ANNs has been developed as generalization of mathematical models of human cognition or neural biology, based on the assumptions that:
  o Information processing occurs at many simple elements called neurons, units, cells, or nodes.
  o Signals are passed between neurons over connection links.
  o Each connection link has an associated weight, which, in a typical neural net, multiplies the signal transmitted.
  o Each neuron applies an activation function (usually nonlinear) to its net input (sum of weighted input signals) to determine its output signal.
    It is also called “Transfer Function”.

Artificial Neuron in a layer
Artificial Neuron

- Each neuron is connected to other neurons by means of directed communication links, each with an associated weight.
- The weights represent information being used by the net to solve a problem.

Characteristics & Applications

- A neural network is characterized by:
  - Its pattern connections between the neurons (called architecture)
  - Its method of determining the weights on the connections (called training, or learning algorithms)
  - Its activation function
- Those above also depends on training data spaces.
- Neural nets can be applied to a wide variety of problems, such as storing and recalling data or patterns, classifying patterns, performing general mappings from input patterns to output patterns, grouping similar patterns (clustering), or finding solutions to constrained optimization problems.
- Most common applications lies in "Fitting a Function", "Recognizing Patterns", "Clustering Data", and "Time Series Prediction".

Learning: Supervision

- Supervised training:
  - Training is accomplished by presenting a sequence of training vectors, or patterns, each with an associated target output vector.
  - The weights are then adjusted according to a learning algorithm.
- Unsupervised training:
  - A sequence of input vectors is provided, but no target vectors are specified.
  - The net modifies the weights so that the most similar input vectors are assigned to the same output (or cluster) unit.

Activation Functions

- The output of the neuron is related to the network input via a linear or nonlinear transformation called the activation function or transfer function.
- Based on different applications and network structure, we can use different type of activation functions in neuron layers.

Hard-Limit Transfer Function

\[ a = \begin{cases} 
1 & n \geq 0 \\
0 & n < 0 
\end{cases} \]
**Activation Functions**

- Linear Transfer Function
- Log-Sigmoid Transfer Function
- Radial Basis Function
- Tan-Sigmoid Transfer Function

**Network Topologies**

- **Acyclic Topology**
  - Consists of no feedback loops.
  - Also called feedforward topology.
  - Often used to approximate a nonlinear mapping between its inputs and outputs.

- **Cyclic Topology**
  - Contains at least one feedback loop.
  - Also known as a recurrent network.
  - Leads to a nonlinear dynamic system model that contains internal memory.
  - Used in competitive structure.

**Network Structures: Connections**

- Multiple neurons are interconnected to form a network to facilitate distributed computing.
- This configuration can be described with a directed graph.
- Based on the applications, we use appropriate structure with regards to training data spaces, learning algorithms, etc.
- As in the case with most neural networks, the aim is to train the net to achieve a balance between the ability to respond correctly to the input patterns that are used for training (memorization) and the ability to give reasonable responses to input that is similar, but not identical, to that used in training (generalization).
For problems that involve mapping a given set of inputs to a specified set of target outputs (which is supervised learning), using a multi-layer perceptron is very common.

The goal is to achieve a balance between:

- Respond correctly to the input patterns that are used for training (memorization)
- Give reasonable responses to input that is similar, but not identical, to that used in training (generalization)

**Mean Square Error (MSE)**

- The Delta rule, or Least Mean Square (LMS) rule is used as error parameter in training.
- The learning rule minimizes the mean squared error (MSE) between the activation and the target values of each layer.
- This allows the net to continue learning on all training patterns even after the correct output value is generated for some patterns.

\[ E(k) = \frac{1}{N} \sum_{p=1}^{N} [t_p - y_p]^2 \]
\[ E = \frac{1}{N} \sum_{k=1}^{N} E(k) = \frac{1}{N} \sum_{p=1}^{N} E_p \]
Universal Approximators

- One use of neural network is to approximate a continuous mapping $f$.
  - A powerful tool for modelling the issues in which the relations between different variables present no clearly defined forms.
  - Very important in modelling the laboratory samples.

- The question is, how well a multilayer net can perform this task?
  - How many layer?
  - How many units?
  - What about precision?
  - Which structure?

Universal Approximators

- Kolmogorov Mapping Neural Network Existence Theorem:
  A feedforward neural network with two layers of neurons (input units, a sigmoid layer and a linear output layer) can represent any function with a finite number of discontinuities.

NN Implementation

- Neural Networks are highly parallel.
- A basic problem in all forms of parallel computing is how best to map applications onto hardware.
- FPGAs, based on their concurrent nature, are a promising candidate for construction of design, implementation, and use of large scale parallel architectures for neural network applications.
- In case of MLP and back-propagation, the goal is to completely parallelize the entire computations.

Practical Example

- One of the most common applications of neural networks is "Pattern Recognition".
- Likewise other supervised training processes, associated target vectors are used.
- For each input vector, only one of the output values would be high, representing the associated pattern, while the others would be low.
- An Excel file has been prepared that contains all the input and output vectors in bipolar format.
- Now let's define the patterns and move on to Matlab workspace.
**NN Location in process**

- Pre-processing units such as image processing units, databases, filters, and etc., provide usable data for NN unit. It may be even a manual process. They extract input vectors from raw data.
- Post-processing units may be used for inference & decision making, optimization, data compression, etc.

**Input Vectors**

- In pre-processing unit, input forms has been converted into binary strings.

**Training Vectors**

- Three input forms are applied in binary string format to train an MLP network.
**VHDL: An Instant Approach**

- VHDL Basics
  - Code Structure
  - Arrays
  - Operations
- Concurrent versus Sequential Code
- Data Values
- Miscellaneous Examples
  - DFF plus NAND gate
  - Sine Wave Generation
  - ALU
- Memory Design: ROM

**VHDL Basics: Code Structure**

- A standalone piece of VHDL code is composed of at least three fundamental sections:
  - **LIBRARY**: a collection of commonly used pieces of code.
  - **ENTITY**: a list with specifications of all input and output pins (PORTS) of the circuit.
  - **ARCHITECTURE**: a description of how the circuit should behave (function).
**VHDL Basics: Library**

- A collection of commonly used pieces of code. Placing such pieces inside a library allows them to be reused or shared by other designs.

```vhdl
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

**VHDL Basics: Entity**

- A list with specifications of all input and output pins (PORTS) of the circuit.

```vhdl
ENTITY entity_name IS
  PORT (port_name : signal_mode signal_type;
        port_name : signal_mode signal_type;
        ...);
END entity_name;
```

**VHDL Basics: Architecture**

- A description of how the circuit should behave (function).

```vhdl
ARCHITECTURE architecture_name OF entity_name IS
  [declarations]
BEGIN
  [code]
END architecture_name;
```

- Has two parts:
  - Declarative part (optional): signals and constants are declared.
  - Code part (from BEGIN down)

**VHDL Basics: Arrays**

- Arrays are collections of objects of the same type.
- They can be one-dimensional (1D), two-dimensional (2D), or one-dimension-by-one-dimensional (1D×1D).

```
(a) [0 1 0 0]
(b) [0 1 0 0]
(c) [0 0 1 0]
(d) [0 1 0 1]
```

(a) a single value (scalar), (b) a vector (1D), (c) an array of vectors (1D×1D), and (d) an array of scalars (2D).
VHDL Basics: Arrays

- To specify a new array type:
  ```vhdl
  TYPE type_name IS ARRAY (specification) OF data_type;
  ```

- 1D array example:
  ```vhdl
  TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
  TYPE matrix IS ARRAY (0 TO 3) OF row;
  SIGNAL x: matrix;
  ```

- 2D array example:
  ```vhdl
  TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
  ```

- Array initialization:
  ```vhdl
  ... := "0001";
  ... := ['0', '0', '0', '0'];
  ... := ['0', '0', '1', '1', '1', '1', '1', '1'];
  ```

VHDL Basics: Port Array

- There are no pre-defined data types of more than one dimension.
- We might need to specify the ports as arrays of vectors, in the specification of the input or output pins (PORTS).
- The solution is to declare user-defined data types in a PACKAGE. It will be visible to the whole design (the TYPE declarations are not allowed in an ENTITY).

```vhdl
PACKAGE my_data_types IS
  TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
    STD_LOGIC_VECTOR(7 DOWNTO 0);
END;
```

```vhdl
ENTITY mux IS
  PORT (inp : IN VECTOR_ARRAY (0 TO 3);
  ...);
END;
```

VHDL Basics: Operations

- VHDL provides several kinds of pre-defined operators:
  - Assignment operators
  - Logical operators
  - Arithmetic operators
  - Relational operators
  - Shift operators
  - Concatenation operators

Assignment Operators:
- `<=` Used to assign a value to a SIGNAL.
- `:=` Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Used also for establishing initial values.
- `=>` Used to assign values to individual vector elements or with OTHERS.

```vhdl
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Leftmost bit is MSB
SIGNAL w : STD_LOGIC_VECTOR(0 TO 7); -- Rightmost bit is MSB

* Assignments:
  x <= '1';  -- '1' is assigned to SIGNAL x using "<="
  y := "0000";  -- "0000" is assigned to VARIABLE y using "="
  w <= "10000000";  -- LSB is '1', the others are '0'
  w <= (0 =>'1', OTHERS =>'0');  -- LSB is '1', the others are '0'
```
**VHDL Basics: Operations**

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Operators</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>&lt;=, /=, =:=</td>
<td>Any</td>
</tr>
<tr>
<td>Logical</td>
<td>NOT, AND, NAND, OR, NOR, XOR, XNOR</td>
<td>BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR, INTEGER, SIGNED, UNSIGNED</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+, -, *, /, **</td>
<td>BIT_VECTOR, INTEGER, SIGNED, UNSIGNED</td>
</tr>
<tr>
<td>Comparison</td>
<td>=, /=, &lt;, &lt;=, &gt;=</td>
<td>All above</td>
</tr>
<tr>
<td>Shift</td>
<td>shl, shr, sla, sra, rol, ror</td>
<td>BIT_VECTOR</td>
</tr>
<tr>
<td>Concatenation</td>
<td>&amp;, (, )</td>
<td>Same as for logical operators, plus SIGNED and UNSIGNED</td>
</tr>
</tbody>
</table>

* The constructs that are not synthesizable (or have little synthesis support) are marked with the "•" symbol.

**VHDL Design Styles (Architecture)**

- **DATAFLOW**
  - "concurrent" statements
  - Gates
  - Simple Comb. Logic

- **STRUCTURAL**
  - components and interconnects
  - State machines
  - Registers
  - Complex Comb. Logic

- **SYTHESIZABLE**
- **NON-SYNTHESIZABLE**
  - "sequential" statements

**Concurrent vs. Sequential Logic**

**Combinational Logic**
- The output of the circuit depends solely on the current inputs.
- Requires no memory, can be implemented using conventional logic.

**Sequential Logic**
- The output depend on previous inputs.
- Storage elements are required, which are connected to the combinational logic block through a feedback loop.

**Concurrent vs. Sequential Code**

- VHDL is inherently concurrent (parallel); so to implement any clocked circuits we have to "force" VHDL to be sequential.
- Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential.
- Though within these blocks the execution is sequential, the block, as a whole, is concurrent with any other (external) statements.
- Concurrent processes with sequential execution within a process offers maximum flexibility
- Sequential code also referred to as "Behavioral code".
- Concurrent code is also called "dataflow code."
  - Dataflow is a software architecture based on the idea that changing the value of a variable should automatically force the recalculation of the variables which depends on its value.
**VHDL Behavior Models**

**Concurrent**

```
begin
    statement
    statement
    statement
end
```

**Sequential**

```
begin
    statement
    statement
    statement
end
```

VHDL provides concurrent statements using 'structural' statements and serial operations using 'behavioral statements'.

**Concurrent Mechanism**

- Basic granularity of concurrency is the process
  - Processes are executed concurrently
  - Concurrent signal assignment statements are one-line processes.
- Mechanism for achieving concurrency:
  - Processes communicate with each other via signals.
  - Signal assignments require delay before new value is assumed.
  - Simulation time advances only when all active processes complete.
  - Effect is concurrent processing.
    - So, the order in which processes are actually executed by simulator does not affect behaviour.

**Concurrent Code: When**

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...?

WITH identifier SELECT
    assignment WHEN condition value,
    assignment WHEN condition ELSE
    "010";

WITH control SELECT
    output <= "000" WHEN reset,
    "111" WHEN set,
    UNEXPECTED WHEN OTHERS;
```

**Sequential Code: Wait**

```
WAIT UNTIL signal-condition;

WAIT ON signal1, signal2, ...

--- With WHEN/ELSE ---
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
      "001" WHEN reset='1' ELSE
      "010";

--- With WHEN/SELECT/WHEN ------------------------
WITH control SELECT
    output <= "000" WHEN reset,
    "111" WHEN set,
    UNEXPECTED WHEN OTHERS;
```

FPGA Implementation of Neural Networks
Semnan University – Spring 2012
Data Values

- VHDL provides objects for dealing with values:
  - For established (static) values: CONSTANT and GENERIC
  - For non-static data values: SIGNAL and VARIABLE

- CONSTANT
  - Serves to establish default values.
  - Can be global (seen by the whole code)
  - Can be declared in a PACKAGE, ENTITY, or ARCHITECTURE.

```vhdl
CONSTANT name : type := value;
```

- SIGNAL serves to pass values in and out the circuit, and between its internal units.
- A signal represents circuit interconnections (wires).
- All PORTS of an ENTITY are signals by default.
- Its declaration can be made in the same places as the declaration of a CONSTANT.
- The assignment operator for a SIGNAL is “<=”.
- The initial value in the syntax above is not synthesizable, being only considered in simulations.

Data Values: Generics

- A generic parameter is a static parameter that can be easily modified and adapted to different applications. The purpose is more flexibility and reusability.
- A GENERIC statement must be declared in the ENTITY.
- The specified parameter will be truly global (visible to the whole design).

```vhdl
GENERIC (parameter_name : parameter_type := parameter_value);
```

- Example:

```vhdl
ENTITY my_entity IS
  GENERIC (n : INTEGER := 8);
  PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF my_entity IS...
END my_architecture;
```

Data Values: Signals

- SIGNAL allows a design entity to be described so that, for each use of that component, its structure and behaviour can be changed by generic values.
- In general they are used to construct parameterized hardware components.
- "GENERIC" is a great asset when you use your design in many places with slight changes in the register sizes, input sizes, etc.
- But if the design is very unique you don’t need to have generic parameters.
**Data Values: Signals**

- When used inside a section of sequential code (e.g., PROCESS)
  - Its update is **not immediate**.
  - Its new value should not be expected to be ready before the corresponding
    PROCESS, FUNCTION, or PROCEDURE.

```vhdl
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

**Data Values: Variables**

- Represents only **local** information.
  - Can only be used **inside a sequential code**.
  - Its declaration can only be done in the declarative part of a sequential code.
  - Its values **cannot be passed out directly**. It must be assigned to a SIGNAL first.
  - Its update is **immediate**.
    - The new value can be promptly used in the next line of code.
    - The assignment operator for a variable is `:=`.
    - The initial values are synthesizable.

```vhdl
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

---

**Data Values: Signals vs. Variables**

<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>VARIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td><code>&lt;=</code></td>
</tr>
<tr>
<td>Utility</td>
<td>Represents circuit interconnects (wires)</td>
</tr>
<tr>
<td>Scope</td>
<td>Can be global (seen by entire code)</td>
</tr>
<tr>
<td>Behavior</td>
<td>Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)</td>
</tr>
<tr>
<td>Usage</td>
<td>In a PACKAGE, ENTITY, or ARCHITECTURE, in an ENTITY, all PORTS are SIGNALS by default</td>
</tr>
</tbody>
</table>

---

**Examples: DFF plus NAND gate**

```
1 2 ENTITY example IS
2 3 PORT (a, b, clk: IN BIT;
2 4       q: OUT BIT);
2 5 END example;
2 6 ARCHITECTURE example OF example IS
2 7 BEGIN temp : BIT;
2 8 SIGNAL temp : BIT;
2 9 BEGIN temp <= a AND b;
2 10 PROCESS (clk)
2 11 BEGIN IF (clk EVENT AND clk='1') THEN q<=temp;
2 12 END IF;
2 13 END PROCESS;
2 14 END example;
```

- Line 10 is executed **concurrently** with block 11-15.
Examples: Sine Wave Generation

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
architecture Behavioral of sinewave is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      data <= sine_wave(i);
      i <= i + 1;
    end if;
  end process;
end Behavioral;
```

Examples: ALU

<table>
<thead>
<tr>
<th>op</th>
<th>function</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>y := a</td>
<td>add a &amp; b</td>
</tr>
<tr>
<td>0001</td>
<td>y := a+1</td>
<td>increment a</td>
</tr>
<tr>
<td>0010</td>
<td>y := a-1</td>
<td>decrement a</td>
</tr>
<tr>
<td>0100</td>
<td>y := b+1</td>
<td>increment b</td>
</tr>
<tr>
<td>0101</td>
<td>y := b-1</td>
<td>decrement b</td>
</tr>
<tr>
<td>0110</td>
<td>y := a x b</td>
<td>multiply a &amp; b</td>
</tr>
<tr>
<td>1000</td>
<td>y := NOT a</td>
<td>complement a</td>
</tr>
<tr>
<td>1001</td>
<td>y := a AND b</td>
<td>AND</td>
</tr>
<tr>
<td>1010</td>
<td>y := a OR b</td>
<td>OR</td>
</tr>
<tr>
<td>1100</td>
<td>y := a NAND b</td>
<td>NAND</td>
</tr>
<tr>
<td>1101</td>
<td>y := a NOR b</td>
<td>NOR</td>
</tr>
<tr>
<td>1110</td>
<td>y := a XOR b</td>
<td>XOR</td>
</tr>
<tr>
<td>1111</td>
<td>y := a XNOR b</td>
<td>XNOR</td>
</tr>
</tbody>
</table>

Memory Design: ROM

- Since it is read-only memory, no clock signal or write-enable pin is necessary.
- The circuit is a pile of pre-stored words, being the one selected by the address input (addr) presented at the output (data).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY rom IS
  GENERIC ( bits: INTEGER := 8; -- # of bits per word
             words: INTEGER := 8; -- # of words in the memory
          );
  PORT ( addr: IN INTEGER RANGE 0 TO words-1;
          data: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END rom;
```

A LUT is simply a ROM.

Memory Design: ROM

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY rom IS
  GENERIC ( bits: INTEGER := 8; -- # of bits per word
             words: INTEGER := 8; -- # of words in the memory
          );
  PORT ( addr: IN INTEGER RANGE 0 TO words-1;
          data: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END rom;
```

15
Memory Design: ROM

```
ARCHITECTURE rom OF rom IS
  TYPE vector_array IS ARRAY (0 TO words-1) OF
  STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
  CONSTANT memory: vector_array := ( "00000000", "00000010", "00000100", "00001000", "00010000", "00100000", "01000000", "10000000");
BEGIN
  data <= memory(addr);
END rom;
```

VHDL: Any Questions?

A Neuron Implementation

- A Layer Structure
- Small Scale Approach
- Large Scale Approach

Activation function implementation will be discussed in the next section.

The main difference between these two solutions is that the first code is not as generic, and therefore adequate for specific, small designs. The second solution, being generic, is reusable and easily adaptable to any NN size.

A Layer Structure: Schematic
A Layer Structure: Description

- This approach is appropriate for general purpose NNs (that is, with programmable weights).
- It employs only one input to load all weights (thus saving on chip pins).
- The weights are shifted in sequentially until each register is loaded with its respective weight.
- The weights are then multiplied by the inputs and accumulated to produce the desired outputs.

Small Scale Approach

- Has the advantage of being simple, easily understandable, and self-contained in the main code.
- Its limitation is that the inputs (x) and outputs (y) are specified one by one.

Small Scale Approach: Entity

```vhdl
6 ENTITY nn IS
7 GENERIC ( n : INTEGER := 3; -- # of neurons
8 m : INTEGER := 3; -- # of inputs or weights per neuron
9 b : INTEGER := 4; -- # of bits per input or weight
10 PORT
11 x1 : IN SIGEND(b-1 DOWNTO 0);
12 x2 : IN SIGEND(b-1 DOWNTO 0);
13 x3 : IN SIGEND(b-1 DOWNTO 0);
14 w : IN SIGEND(b-1 DOWNTO 0);
15 clk : IN STD_LOGIC;
16 test : OUT SIGEND(b-1 DOWNTO 0); -- register test output
17 y1 : OUT SIGEND(2*b-1 DOWNTO 0);
18 y2 : OUT SIGEND(2*b-1 DOWNTO 0);
19 y3 : OUT SIGEND(2*b-1 DOWNTO 0);
20 END nn;
```

Small Scale Approach: Architecture

```vhdl
21 ARCHITECTURE neural OF nn IS
22 TYPE weights IS ARRAY (1 TO n*4) OF SIGEND(b-1 DOWNTO 0);
23 TYPE inputs IS ARRAY (1 TO n) OF SIGEND(b-1 DOWNTO 0);
24 TYPE outputs IS ARRAY (1 TO n) OF SIGEND(2*b-1 DOWNTO 0);
25 BEGIN
26 process (clk, w, x1, x2, x3)
27 variable weight, weights;
28 variable height, inputs;
29 variable outputs, outputs;
30 variable prod, ans, sign; -- SIGN(1,2,b-1 DOWNTO 0);
31 variable sign: STD_LOGIC;
32 BEGIN
33 ---- shift register instructions -------------------
34 IF (clk'EVENT AND clk='1') THEN
35 weight := w & weight(1 TO n*4-1);
36 END IF;
37 ---- initialization -------------------
38 input(1) := x1;
39 input(2) := x2;
40 input(3) := x3;
41 ---- multiply-accumulate -------------------
42 for i in 1 TO n loop
43 acc := (others => '0');
```
Large Scale Approach

- We use a more generic approach which is appropriate for large scale networks.
- The idea is to use arrays for representing inputs, outputs, and weights.
- Since type declaration is not permitted in the entity, we use a package for it.

```
-- Large Scale Approach: Entity
7 ENTITY nn3 IS
8 GENERIC ( n: INTEGER := 3; -- # of neurons
9 m: INTEGER := 3; -- # of inputs or weights per
10 neuron
11 b: INTEGER := 3); -- # of bits per input or
12 weight
13 PORT ( x: IN VECTOR_ARRAY_IN (1 TO m));
14 w: IN SIGNED(b-1 DOWNTO 0));
15 clk: IN STD_LOGIC;
16 test: OUT SIGNED(b-1 DOWNTO 0)); -- register test
17 -- output
18 y: OUT VECTOR_ARRAY_OUT(1 TO n));
19 END nn3;
20 -------------------------------
```

```
-- Large Scale Approach: Architecture
21 ARCHITECTURE neural OF nn3 IS
22 BEGIN
23 PROCESS (clk, w, x)
24 VARIABLE weight: VECTOR_ARRAY_IN (1 TO n);w);
25 VARIABLE prod, acc: SIGNED(2*b-1 DOWNTO 0);
26 VARIABLE sign: STD_LOGIC;
27 BEGIN
28 ----- shift register inference:---------------------
29 IF (clk'EVENT AND clk='1') THEN
30 weight := w & weight(1 TO n-1);
31 END IF;
```
Large Scale Approach: Architecture

```plaintext
32    a out += weight(i*n);  
33    -- initialization: ------------------------------  
34    a out := (OTHERS => '0');  
35    ------ multiply-accumulate: ---------------------  
36    L1: FOR i IN 1 TO n LOOP  
37        L2: FOR j IN 1 TO n LOOP  
38            prod := x[j]*weight(i*n+j);  
39            sig := acc (acc+prod);  
40            acc := acc + prod;  
41        END LOOP L2;  
42    END LOOP L1;  
43    END PROCESS;  
44    RETURN a out;  
```

Neuron Implementation: Any Questions?

Activation Function Implementation

- Implementation Problem
- Look-Up Tables (LUT)
- Linear Approximation Methods
- Range Addressable LUTs (RALUT)
- Hybrid Methods

Implementation Problem

- The sigmoid and hyperbolic tangent functions are usually used as the activations in ANNs.
- Straightforward implementation of these functions in hardware is not practical due to their exponential nature.
- Several different approaches exist for this hardware approximation, including lookup tables (LUT), piecewise linear approximation, piecewise non-linear approximation, and hybrid methods.

\[
\begin{align*}
\text{Sigmoid}(x) &= \frac{1}{1 + e^{-x}} \\
\text{Tanh}(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}
\end{align*}
\]
**Lookup Tables (LUT)**

- Generally, LUT implementations are the fastest, while they consume a large area of silicon.
- They are based on ROM design. Get an input range or input address.

**Piecewise Linear Approximation**

- Piecewise linear approximations are slow, but consume less area; they are also the most common way of implementing the activation functions.

**Comparison of Implementation Methods**

- Generally, LUT implementations are the fastest, while they consume a large area of silicon.
- Piecewise linear approximations are slow, but consume less area; they are also the most common way of implementing the activation functions.
- Piecewise non-linear approximations achieve excellent approximation (with low maximum error) but are the slowest type of approximation, as they usually make use of multipliers in their architecture.
- Hybrid methods typically achieve good performance regarding area and delay as they employ a combination of approaches, such as the use of piecewise approximation with lookup tables.

**Complexity comparison**

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Function</th>
<th>Max-Error</th>
<th>AVG-Error</th>
<th>Area</th>
<th>Delay</th>
<th>Area × Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>Tanh</td>
<td>0.050%</td>
<td>57647.4 µm²</td>
<td>2.61 ns</td>
<td>98.26 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>RALUT</td>
<td>Tanh</td>
<td>0.331%</td>
<td>9714.7 µm²</td>
<td>2.26 ns</td>
<td>647.46 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>Tanh</td>
<td>0.203%</td>
<td>12100.1 µm²</td>
<td>3.03 ns</td>
<td>37.57 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>LUT</td>
<td>Sigmoid</td>
<td>2.28%</td>
<td>9646.5 µm²</td>
<td>2.31 ns</td>
<td>8.42 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>RALUT</td>
<td>Sigmoid</td>
<td>0.436%</td>
<td>34159.9 µm²</td>
<td>2.42 ns</td>
<td>83.01 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>Sigmoid</td>
<td>0.260%</td>
<td>9723.6 µm²</td>
<td>3.01 ns</td>
<td>27.91 × 10⁻⁶</td>
<td></td>
</tr>
<tr>
<td>LUT</td>
<td>Hybrid</td>
<td>0.52%</td>
<td>3004.5 µm²</td>
<td>2.36 ns</td>
<td>1.01 × 10⁻⁶</td>
<td></td>
</tr>
</tbody>
</table>
Activation Function: Any Questions?

The Entire Design

Any Questions?