```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- config.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;
    use ieee.fixed_float_types.all;

package constants is
    /* Compile-Time Data File's Location */
    -- NOTE: You could also use relative paths (../) here, but they
    -- vary between simulators/synthesizers, defeating the purpose.
    constant c_DAT_PATH : string :=
        "C:/Users/Thrae/Desktop/Innervator/data";
    /* FPGA Constrains & Configurations */
    -- Clock
    constant c_CLK_FREQ : positive   := 100e6;
    constant c_CLK_PERD : time       := 1 sec / c_CLK_FREQ;
    -- NOTE: Apparently, some FPGAs (e.g., Xilinx 7 series) work better
    -- (internally) with active-high resets, because their flip-flops
    -- were designed to take reset signals as so, and using active-low
    -- would require an inverter before each flip-flop's SR port; yet,
    -- board vendors might use negative/active-low signals for their
    -- reset buttons, because electronics are designed easier that way.
    -- If we have no control over the (external) polarity of our reset
    -- signal, a solution is to place a single inverter in the top-
    -- level hierarchy of the I/O pin logic ("IOB") & use that instead.
    -- SEE:
    --     ednasia.com/coding-consideration-for-pipeline-flip-flops
    --     01signal.com/electronics/iob-registers
    --
    --     Also, you might want to "synchronize" (i.e., de-glitch, NOT
    -- related to sync./async. types of reset) and possibly debounce,
    -- if it is a button, your external reset signal prior to using it.
    -- signal.
    -- Reset (int./ext. = internal/external; neg. = negative)
    constant c_RST_INVT : boolean    := true; -- Invert ext. reset pin
    constant c_RST_POLE : std_ulogic := '1';  -- '0' = int. neg. reset
    constant c_RST_SYNC : boolean    := true; -- false = async. reset
    -- Input port synchronization (num. = number)
    constant c_SYNC_NUM : natural    := 3; -- Port sync./deglitch num.
    -- Button/switch debouncing
    constant c_DBNC_LIM : time       := 30 ms; -- Debounce timeout
    -- TODO: Have a constant that chooses rising_ or falling_edge
    --constant c_EDG_RISE
    /* Internal Fixed-Point Sizing */
    constant c_WORD_INTG   : natural  := 4;
    constant c_WORD_FRAC   : natural  := 4;
    constant c_WORD_SIZE   : positive := c_WORD_INTG + c_WORD_FRAC;
    constant c_GUARD_BITS  : natural  := 0;
    constant c_FIXED_ROUND : fixed_round_style_type    :=
        fixed_truncate;
    constant c_FIXED_OFLOW : fixed_overflow_style_type :=
        fixed_saturate;
    /* Neuron Settings */
    -- TODO: Make these arrays to configure each layer/neuron.
    --
    -- Number of data to be concurrently processed in a single neuron
    -- (More = Faster network, at the expense of more FPGA logic usage)
```

```
62      constant c_BATCH_SIZE : positive := 1; -- < or = to data's length.
63      -- Number of pipelining registers in each neuron; this would be
64      -- the amount in clock cycles of latency in input --> output, too.
65      -- (Less = Faster network, at the expense of route timing failure)
66      constant c_PIPE_STAGE : natural := 3; -- 0 = Disable pipelining
67      -- TODO: Add options to select between executing the activation
68      -- function and/or setting the done signal inside or outside
69      -- the neurons' busy state. (Speed/Size trade-off)
70      /* UART Parameters */
71      -- NOTE: Bitrate = Baud, in the digital world
72      constant c_BIT_RATE : positive := 9_600;
73      constant c_BIT_PERD : time     := 1 sec / c_BIT_RATE;
74  end package constants;
75
76
77  -- ------------------------------------------------------------------------
78  -- END OF FILE: config.vhd
79  -- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- debouncer.vhd is a part of Innervator.
-- ------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library config;
    use config.constants.all;

-- Background: When you press a physical button, the metal contacts
-- don't make a perfect, clean contact instantly; instead, they might
-- "bounce" against each other several times, over a few milliseconds,
-- before settling into a closed state.  Additionally, Microcontrollers
-- and FPGAs are incredibly fast, and they can detect each of those
-- tiny bounces as if they were separate button presses; this could
-- lead to a single button press being interpreted as multiple presses.
--     There are many ways to resolve this matter, and they could be
-- done using hardware approaches (e.g., using a resister-capacitor)
-- or software-based ones.  In a software approach, we could detect
-- a button transition and sample it again at a later point in time,
-- which is at least a few milliseconds long (like 10 ms); if the
-- button's state had remained the same (i.e., it was "stable"), we
-- output that the button was "pressed" once.
--     Be aware that other problems arising from external, wired
-- interfaces might still apply: we had better accounted for
-- metastability and asynchnorized clock domains.
entity debouncer is
    generic (
        -- Timeout in milliseconds
        g_TIMEOUT_MS : time := 30 ms
    );
    port (
        i_clk     : in  std_ulogic;
        i_button  : in  std_ulogic;
        o_button  : out std_ulogic
    );
end entity debouncer;

architecture behavioral of debouncer is
    -- Logically, we are supposed to divide the time by 1000,
    -- but VHDL simulators just don't like it when you perform
    -- integer math with physical units like time.
    constant timeout_ticks  : positive :=
         (g_TIMEOUT_MS / ms) * (c_CLK_FREQ / 1000);
    signal timeout_count  : natural range 0 to timeout_ticks-1 := 0;

    signal previous_state : std_ulogic := '0';
begin

    process (i_clk) is
    begin
        if rising_edge(i_clk) then
            if (i_button /= previous_state and
                timeout_count < timeout_ticks-1)
            then
                -- If there's been a change in the button's state, we
                -- begin to track it as long as it hasn't stayed
                -- "stable" (i.e., unchanged) over the given timeout.
```

```
62                  timeout_count <= timeout_count + 1;
63              elsif timeout_count = timeout_ticks-1 then
64                  -- Otherwise, for the duration of the timeout, the
65                  -- button did not change and can be registered.
66                  previous_state <= i_button;
67              else
68                  -- No change in button state; keep waiting.
69                  timeout_count <= 0;
70              end if;
71          end if;
72      end process;
73
74      o_button <= previous_state;
75
76  end architecture behavioral;
77
78  -- ------------------------------------------------------------------------
79  -- END OF FILE: debouncer.vhd
80  -- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- neural_typedefs.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


-- NOTE: You NEED to re-declare used libraries AFTER you instantiate
-- the packages here; otherwise, basic types like "std_logic" will not
-- get recognized!  This is because VHDL's and Ada's ancient compilers
-- were "one-pass" and did not keep track of contexes.  See:
-- https://insights.sigasi.com/tech/use-and-library-vhdl/
--library work, std, ieee;

library ieee;

library config;
    use config.constants.all;

-- TODO: Provide an option to use unsigned types, too.
package fixed_neural_pkg is
    generic (
        -- NOTE: these two generic numbers should NOT be taken as
        -- generics, because Vivado 2023.2 has a bug where it throws
        -- random error messages if types, which are defined in a
        -- generic package based on generic parameters, are used
        -- outside; this is not a problem with VHDL or the code but
        -- with Vivado itself, as always.  The solution is to
        -- "hard-code" them as constants from a global config file.
        --g_INTEGRAL_BITS, g_FRACTIONAL_BITS : natural;
        package g_FIXED_PKG_INSTANCE is new ieee.fixed_generic_pkg
            generic map ( <> ) -- VHDL-2008 Formal Generic Package
    );
    use g_FIXED_PKG_INSTANCE.all; -- Import our custom fixed_pkg

    -- Vivado bug workaround (see the note above)
    constant g_INTEGRAL_BITS   : natural := c_WORD_INTG;
    constant g_FRACTIONAL_BITS : natural := c_WORD_FRAC;

    -- NOTE: A word is the "primary" size of values that we are
    -- going to use.  Also, historically, a byte was the number
    -- of bits used to encode a character of text in a computer.
    -- SEE: https://en.wikipedia.org/wiki/Units_of_information
    --
    -- NOTE: In fixed-point arithmetic, the "Q" notation is used to
    -- indicate the minimum number of bits required to represent a
    -- range of values. For example, signed Q0.7 uses 1 bit for the
    -- signdedness, 0 bit for  the integer part, and 7 bits for the
    -- fractional part.  Similarly, unsigned Q2.5 represents 2
    -- integer and 5 fractional bits.
    -- See: inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html
    --
    -- NOTE: Bit Width = |INTEGRAL_BITS| + |FRACTIONAL_BITS|; the
    -- integral (i.e., whole) part comprises of [INTEGRAL_BITS-1,
    -- 0] and the fractional (i.e., after the .decimal point)
    -- comprises of [-1, -FRACTIONAL_BITS]


    /*
        Types derived from the generically given specifications
    */

```

```
    /* Scalars */
    subtype neural_bit    is -- An "analogue" bit with the range [0, 1)
        u_ufixed (-1 downto -(g_FRACTIONAL_BITS*2)); -- UNSIGNED and <1
    subtype neural_nibble is -- Half-word
        u_sfixed ((g_INTEGRAL_BITS/2)-1 downto -(g_FRACTIONAL_BITS/2));
    subtype neural_word   is -- Full-word
        u_sfixed (g_INTEGRAL_BITS-1 downto -g_FRACTIONAL_BITS);
    subtype neural_dword  is -- Double-word
        u_sfixed ((g_INTEGRAL_BITS*2)-1 downto -(g_FRACTIONAL_BITS*2));
    subtype neural_qword  is -- Quadruple-word
        u_sfixed ((g_INTEGRAL_BITS*4)-1 downto -(g_FRACTIONAL_BITS*4));
    subtype neural_oword  is -- Octuple-word
        u_sfixed ((g_INTEGRAL_BITS*8)-1 downto -(g_FRACTIONAL_BITS*8));
    /* Vectors */
    -- A single-row (1-D) array of neural_*word values.
    type  neural_vector  is array (natural range <>) of neural_word;
    type  neural_bvector is array (natural range <>) of neural_bit;
    type  neural_nvector is array (natural range <>) of neural_nibble;
    alias neural_wvector is neural_vector;
    type  neural_dvector is array (natural range <>) of neural_dword;
    type  neural_qvector is array (natural range <>) of neural_qword;
    type  neural_ovector is array (natural range <>) of neural_oword;
    /* Matrices */
    -- A multi-row/nested (NOT 2-D) array of neural_*vector arrays.
    type  neural_matrix  is array (natural range <>) of neural_vector;
    type  neural_bmatrix is array (natural range <>) of neural_bvector;
    type  neural_nmatrix is array (natural range <>) of neural_nvector;
    alias neural_wmatrix is neural_matrix;
    type  neural_dmatrix is array (natural range <>) of neural_dvector;
    type  neural_qmatrix is array (natural range <>) of neural_qvector;
    type  neural_omatrix is array (natural range <>) of neural_ovector;

    /*
        Shorthand notations of the above-mentioned types
    */

    /* Scalars */
    alias nrl_bit  is neural_bit;
    alias nrl_nib  is neural_nibble;
    alias nrl_wrd  is neural_word;
    alias nrl_dwd  is neural_dword;
    alias nrl_qwd  is neural_qword;
    alias nrl_owd  is neural_oword;
    /* Vectors */
    alias nrl_vec  is neural_vector;
    alias nrl_bvec is neural_bvector;
    alias nrl_nvec is neural_nvector;
    alias nrl_wvec is neural_wvector;
    alias nrl_dvec is neural_dvector;
    alias nrl_qvec is neural_qvector;
    alias nrl_ovec is neural_ovector;
    /* Matrices */
    alias nrl_mat  is neural_vector;
    alias nrl_bmat is neural_bmatrix;
    alias nrl_nmat is neural_nmatrix;
    alias nrl_wmat is neural_wmatrix;
    alias nrl_dmat is neural_dmatrix;
    alias nrl_qmat is neural_qmatrix;
    alias nrl_omat is neural_omatrix;

    -- Custom helper types for use in finding files' dimensions
    -- (i.e., number of neurons and weights within each neuron)
```

```
124     type dimension is record
125         rows : natural;
126         cols : natural;
127     end record;
128     type dimensions_array is
129         array (natural range <>) of dimension;
130 
131     -- An enumeration/structure of a layer's neuron(s)' weights (array
132     -- of array) AND each neuron's associated bias (array).
133     type layer_parameters is record
134         dims    : dimension; -- Array limitations' workaround
135         weights : neural_matrix;
136         biases  : neural_vector;
137     end record;
138 
139     -- NOTE: This is more limited than it seems, as each record would
140     -- have to be of exactly the same length in a VHDL array of records
141     -- On the other hand, dynamically allocated arrays, which are
142     -- able to hold variable-length elements, are NOT allowed in
143     -- synthetizable VHDL at all.
144     -- See: https://stackoverflow.com/a/61031840
145     --
146     -- Array of records to hold the parameters (i.e., weights and
147     -- biases) of each and every layer, amounting to the entire network
148     type network_layers is
149         array (natural range <>) of layer_parameters;
150 
151 
152 -- ----------------------------------------------------------------------
153 -- rtl_synthesis off
154 -- pragma translate_off
155 -- ----------------------------------------------------------------------
156     -- Simulation-Time Pointer types:  You may access these types'
157     -- dereferenced variables' values using ".all" suffixes after them.
158     -- Also, you may have "dynamically sized" arrays by concatenating
159     -- new "new" definitions with previous ones.  Lastly, you may re-
160     -- use declared pointer types by using "deallocate()" on them.
161     type neural_word_ptr    is access neural_word;
162     type neural_vector_ptr  is access neural_vector;
163     type neural_matrix_ptr  is access neural_matrix;
164     type network_layers_ptr is access network_layers;
165 
166     -- NOTE: VHDL doesn't permit files of multidimensionals (e.g.,
167     -- matrices) directly, so some improvization with pointer types
168     -- is required in order to retrieve them as arrays of arrays.
169     --
170     -- This could also be a file of fixed-points as fixed_pkg provides
171     -- textio.read, although ONLY in binary, octal, or hex formats.
172     type neural_file is file of neural_word;
173 -- ----------------------------------------------------------------------
174 -- pragma translate_on
175 -- rtl_synthesis on
176 -- ----------------------------------------------------------------------
177 
178 
179     -- NOTE: These "partially" resize their inputs; they are used as
180     -- tricks to force synthetis tools to let us use arrays of
181     -- differing element sizes.  In practice, these 'resize's only
182     -- "expand" their inputs and don't fully scale the data; they
183     -- _partially_ fill-in the expanded result.
184     function resize(
185         input_arr  : in neural_vector;
```

```
        target_dim : in dimension
    ) return neural_vector;

    function resize(
        input_mat  : in neural_matrix;
        target_dim : in dimension
    ) return neural_matrix;


    -- NOTE: These refuse to work with unconstrained generic types;
    -- so, you will have to define the manually for each constant
    -- decleration, if you want to use them.
    /*
    attribute num_inputs  : natural;
    attribute num_outputs : natural;

    attribute num_inputs  of neural_vector : type is
        neural_vector'length;
    attribute num_outputs of neural_vector : type is
        neural_vector'length;

    attribute num_inputs  of neural_matrix : type is
        neural_matrix(0)'length; -- or 'length(dimension) for 2-D array
    attribute num_outputs of neural_matrix : type is
        neural_matrix'length;

    -- NOTE: Records can't have methods; there's no protected record.
    -- So, we can use the rather obscure 'group' keyword to accomplish
    -- a similar objective.
    group layer_group_type is ( constant <> );
    attribute num_neurons : natural;

    group layer_group : layer_group_type (layer_parameters);
    attribute num_neurons of layer_group : group is
        layer_parameters.biases'length; -- or weights'length
    */

    -- Another way of realizing the above idea is by using functions.
    /*
    function len_inputs(
        input_arr  : in neural_vector
    ) return natural;
    function len_outputs(
        input_arr  : in neural_vector
    ) return natural;

    function len_inputs(
        input_mat  : in neural_matrix
    ) return natural;
    function len_outputs(
        input_mat  : in neural_matrix
    ) return natural;

    function len_neurons(
        input_arr  : in neural_vector
    ) return natural;
    function len_neurons(
        input_arr  : in neural_vector
    ) return natural;  -- array of naturals?
    */
end package fixed_neural_pkg;

```

```
248 -- TODO: Whenever Synthesis tools support protected types inside
249 -- synthesis (for constant initializations), as opposed to just
250 -- simulation, convert the methods inside this package body to
251 -- be inside a 'protected' type of 'neural_vector' and '_matrix'.
252 package body fixed_neural_pkg is
253
254     function resize(
255         input_arr  : in neural_vector;
256         target_dim : in dimension
257     ) return neural_vector is
258         variable resized_arr : neural_vector (0 to target_dim.rows-1);
259     begin
260         resized_arr(input_arr'range) := input_arr; -- Partially fill it
261
262         return resized_arr;
263     end function;
264
265     function resize(
266         input_mat  : in neural_matrix;
267         target_dim : in dimension
268     ) return neural_matrix is
269         variable resized_mat : neural_matrix
270             (0 to target_dim.rows-1) (0 to target_dim.cols-1);
271     begin
272         for row in input_mat'range loop
273             --resized_mat(row) := resize(input_mat(row), target_dim);
274             resized_mat(row)(input_mat(row)'range) := input_mat(row);
275         end loop;
276
277         return resized_mat;
278     end function;
279
280 end package body fixed_neural_pkg;
281
282
283
284 -- "'guard_bits' defaults to 'fixed_guard_bits,' which defaults
285 -- to 3. Guard bits are used in the rounding routines. If guard
286 -- is set to 0, the rounding is automatically turned off.
287 -- These extra bits are added to the end of the numbers in the
288 -- division and "to_real" functions to make the numbers more
289 -- accurate." (Fixed point package user's guide by Mr. David Bishop)
290 library ieee;
291 library config;
292     use config.constants.all;
293 package fixed_pkg_for_neural is new ieee.fixed_generic_pkg
294     generic map ( -- NOTE: ieee_proposed pre VHDL-08
295         fixed_round_style    => c_FIXED_ROUND,
296         fixed_overflow_style => c_FIXED_OFLOW,
297         fixed_guard_bits     => c_GUARD_BITS,
298         no_warning           => false
299     );
300
301 library neural;
302 library config;
303     use config.constants.all;
304 package neural_typedefs is new neural.fixed_neural_pkg
305     generic map (
306         --INTEGRAL_BITS        => c_WORD_INTG, -- NOTE: Signed
307         --FRACTIONAL_BITS      => c_WORD_FRAC,
308         g_FIXED_PKG_INSTANCE   => work.fixed_pkg_for_neural
309     );
```

```
-- After instanation, you may use the Packages above as follows:
-- use work.fixed_pkg_for_neural.all, work.neural_typedefs.all;
--
-- Alternatively, you may use their bundle as a context
-- (defined near the end of this file).

-- NOTE: IEEE's official fixed-point package has a bug (in
-- IEEE 1076-2008, a.k.a. VHDL-08), where to_ufixed(sfixed) isn't
-- defined in the Package's header file, while its parallel
-- to_sfixed(ufixed) is.  So, the solution is to copy-paste the
-- decleration of said function from fixed_generic_pkg-body.vhdl
-- into our local library.
-- SEE: https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/269
library ieee;
    use ieee.std_logic_1164.all; -- Added to fix ModelSim's errors

library work;
    use work.fixed_pkg_for_neural.all, work.neural_typedefs.all;

-- TODO: Include arrays of std_(u)logic_vector types?
--type t_slv_arr is array (natural range <>) of std_logic_vector;
--type t_suv_arr is array (natural range <>) of std_ulogic_vector;
package fixed_generic_pkg_bugfix is
    function to_ufixed(
        arg : UNRESOLVED_sfixed
    ) return UNRESOLVED_ufixed;
end package fixed_generic_pkg_bugfix;

package body fixed_generic_pkg_bugfix is
    -- null array constants
    constant NAUF : UNRESOLVED_ufixed (0 downto 1) :=
        (others => '0');

    -- Special version of "minimum" to do some
    -- boundary checking with errors
    function mine(l, r : INTEGER) return INTEGER is
    begin
        if (L = INTEGER'low or R = INTEGER'low) then
            report fixed_generic_pkg_bugfix'instance_name
                & " Unbounded number passed, was a literal used?"
                severity error;
            return 0;
        end if;

        return minimum (L, R);
    end function mine;

    -- converts an sfixed into a ufixed.  The output is the same
    -- length as the input, because abs("1000") = "1000" = 8.
    function to_ufixed(
        arg : UNRESOLVED_sfixed
    ) return UNRESOLVED_ufixed is
        constant left_index  : INTEGER := arg'high;
        constant right_index : INTEGER := mine(arg'low, arg'low);
        variable xarg        :
            UNRESOLVED_sfixed (left_index+1 downto right_index);
        variable result      :
            UNRESOLVED_ufixed (left_index downto right_index);
    begin
        if arg'length < 1 then
            return NAUF;
        end if;
```

```
372
373         xarg   := abs(arg);
374         result := UNRESOLVED_ufixed (
375             xarg (left_index downto right_index)
376         );
377
378         return result;
379
380     end function to_ufixed;
381 end package body fixed_generic_pkg_bugfix;
382
383 -- After instanation, you may use the Packages above as follows:
384 -- use work.fixed_pkg_for_neural.all, work.neural_typedefs.all;
385 --
386 -- Alternatively, you may use their bundle as a context:
387 context neural_context is -- VHDL-2008 feature
388     library neural;
389         use neural.fixed_pkg_for_neural.all,
390             neural.neural_typedefs.all;
391         use neural.fixed_generic_pkg_bugfix.all; -- REQUIRED!
392 end context neural_context;
393
394 -- NOTE: Unfortunately, even if I play by Vivado Simulator's rules by
395 -- placing the VHDL-93 compatibility version of IEEE's fixed_pkg into
396 -- a local directory AND commenting-out all homographes of std_logic_
397 -- vectors AND removing all references to 'line' datatypes, it still
398 -- finds a way to crash abruptly in other areas (e.g., File I/O),
399 -- without any log whatsoever, in simulation; working with Vivado's
400 -- Simulator is pointless, as even a ModelSim version from 8 years ago
401 -- (as of 2024) far outperforms it.
402
403 -- ----------------------------------------------------------------------
404 -- rtl_synthesis off
405 -- pragma translate_off
406 -- ----------------------------------------------------------------------
407
408 -- NOTE: Here, in simulation, we OVERWRITE the previous declaration;
409 -- this is done because there is no other way to detect whether the
410 -- code is being simulated and skip synth's code, in the latter case we
411 -- also need to use a custom version of IEEE's fixed_pkg due to Vivado.
412 /*
413 context neural_context is
414     library ieee_proposed;
415         use ieee_proposed.fixed_pkg.all;
416     -- NOTE: Xilinx Vivado does not support fixed- or floating-point
417     -- packages for use within its simulator, even though it can
418     -- synthetize them just fine; as a workaround, use
419     -- a LOCAL 'ieee_proposed' instead of 'ieee'.
420     --
421     -- SEE:
422     --     docs.xilinx.com/r/en-US/ug900-vivado-logic-simulation/
423     --         Fixed-and-Floating-Point-Packages
424     --
425     --     insights.sigasi.com/tech/list-known-vhdl-metacomment-pragmas
426     library config;
427         use config.neural_typedefs.all;
428         use config.fixed_generic_pkg_bugfix.all; -- REQUIRED!
429 end context neural_context;
430 */
431
432 -- ----------------------------------------------------------------------
433 -- pragma translate_on
```

```
434 -- rtl_synthesis on
435 -- ----------------------------------------------------------------------
436
437 -- ----------------------------------------------------------------------
438 -- END OF FILE: neural_typedefs.vhd
439 -- ----------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- activation.vhd is a part of Innervator.
-- ------------------------------------------------------------------------

library ieee;
    use ieee.std_logic_1164.all;
    --use ieee.math_real.MATH_E; -- NOT synthetizable; use as generics!

library work;
    context work.neural_context;


-- TODO: Implement more activation functions (e.g., ReLU, etc.)
package activation is

    alias in_type  is neural_dword;
    alias out_type is neural_bit;

    function sigmoid(
        x : in_type
    ) return out_type;

end package activation;


package body activation is

    function sigmoid(
        x : in_type
    ) return out_type is
        -- TODO: Automate the generation of these constant look-up
        -- parameters to be done at compile-time, based on their sizes.

        -- Boundaries beyond which linear approximation begins erring.
        constant UPPER_BOUND : in_type :=
            to_sfixed(+2.0625, in_type'high, in_type'low);
        constant LOWER_BOUND : in_type :=
            to_sfixed(-2.0625, in_type'high, in_type'low);

        constant LINEAR_M : in_type := -- M (the coefficient)
            to_sfixed(0.1875, in_type'high, in_type'low);
        constant LINEAR_C : in_type := -- C (the displacement)
            to_sfixed(0.5, in_type'high, in_type'low);

        -- Look-Up constants for when the input value falls beyond
        -- the Linear Approximation's acceptable error range.
        --
        -- NOTE: ZERO and ONE do not correspond to exactly 0.0 and 1.0;
        -- instead, they are a bit "leaky," similar to Sigmoid's output
        --
        -- Actually, these are not really required and only add to the
        -- problem by introducing additional logic timing delay.
        /*
        constant ZERO : out_type :=
            to_ufixed(0.0625, out_type'high, out_type'low);
        constant ONE  : out_type :=
            to_ufixed(0.9375, out_type'high, out_type'low);
        */

        -- A temporary variable is required, because only VHDL-2019
```

```
62          -- allows conditional assignment (i.e., when...else) in
63          -- front of return statements, and we are using VHDL-2008.
64          variable result : out_type;
65
66          --attribute use_dsp : string;
67          --attribute use_dsp of result : variable is "no";
68      begin
69
70          -- It is rather hard to implement exponents and logarithms
71          -- in an FPGA; it would use too many logic blocks.  Even
72          -- approximating Sigmoid using the absolute value would
73          -- still require 'x' to be divided, very slowly, by (1 + |x|).
74          -- So, a linear approximation is used instead.
75          result := --ZERO when x < LOWER_BOUND else
76                    --ONE  when x > UPPER_BOUND else
77                    resize(to_ufixed((LINEAR_M * x) + LINEAR_C), result);
78
79          return result;
80
81      end function sigmoid;
82
83  end package body activation;
84
85  -- ------------------------------------------------------------------------
86  -- END OF FILE: activation.vhd
87  -- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- file_parser.vhd is a part of Innervator.
-- ------------------------------------------------------------------


-- TODO: Because the std.textio is so limited in VHDL, maybe
-- try and investigate VHDL's VHPI to call C/C++ functions?

library std;
    use std.textio.all;

library ieee;
    use ieee.std_logic_1164.all;
    --use ieee.std_logic_textio.all;

-- TODO: Make these subtypes given through generics, in VHDL-2019.
library neural;
    context neural.neural_context;

-- NOTE: The reason this package has been divided into two, including
-- an auxiliary one, is because I could not use the same functions
-- defined in a single package's body inside constant initializations
-- of its header (since the functions were not 'elaborated' on yet);
-- while you might think that 'deferred constants' were a solution to
-- this, they actually were not.
--     Because I also needed to use said deferred constants inside
-- the subtypes/types defined in the actual package's header, I
-- would trigger simulator errors related to "illegal use of deferred
-- constants."  So, this was the only solution to access both.
--     NOTE: Actually, the above-mentioned note no longer applies,
-- because I was able to find a way to constrain array types as
-- subtypes which could then be connected to external, unconstrained
-- types in constant evaluations, but it is nonetheless a good
-- practice to separate auxiliary functionality here.
package file_parser_aux is

    -- NOTE: Because the fixed-point package only provides procedures
    -- to read its types from files in the textual format of binary,
    -- octal, and hexadecimal (i.e., NOT decimal), I initially chose
    -- to read the inputs as floating-point 'real' types which would
    -- then be converted into fixed-point internally.  HOWEVER, this
    -- brought me to the second issue: Xilinx Vivado has a bug in its
    -- implementation of the 'read()' procedure for 'real' datatypes,
    -- which prevents you from reading more than 7 elements from a
    -- single row.  THUS, I had to resort to the lesser-known 'sread()'
    -- procedure, which takes in a (forcibly constrained) string,
    -- strips the given line of any and all types of whitespace
    -- characters and, lastly, tokenizes each whitespace-separated
    -- collection of characters into the input string; this way,
    -- I could get past Vivado's bug and convert the string
    -- representation of floating-points into a 'real' and finally
    -- an actual fixed-point... or, could I?
    --     Vivado, unsurprisingly, did NOT care to implement 'sread()'
    -- AT ALL; while it appears to synthetize correctly, it hard-
    -- crashes the simulator with NO log and also does not even work
    -- in synthesis itself (tested using 'assert' statements).
    -- To make matters even worse, Vivado implemented the fixed-point
    -- package---an official IEEE package---in an incredibly poor and
    -- "hacky" way; not only does the simulator outright REFUSE to work
    -- with the ieee.fixed_generic_pkg, but it also suggets, even as of
```

```
    -- the 2023.2 version, that you use the now-OUTDATED workaround of
    -- replacing ieee with ieee_proposed as to use the VHDL-93 version
    -- of the Package.  The problem here is that, sometime around 2021,
    -- they removed ieee_proposed from the package list, making their
    -- OWN workaround OBSOLETE.  Not only that but, because they again
    -- failed to implement std.textio's 'line' type properly, you also
    -- cannot place the raw files of fixed-point package into your own
    -- project's directory and resolve the issue; Vivado does not even
    -- let you define or use a procedure that takes in a 'line' type
    -- parameter.  This also brings us to the next issue: I could NOT
    -- use fixed-point package's implementation of sfixed/ufixed parser
    -- [i.e., read()].  SEE:
    -- https://support.xilinx.com/s/question/0D52E00006lLghFSAS
    --     ULTIMATELY, I had to resort back to using Vivado's broken
    -- implementation of read() (which is the ONLY choice) and
    -- flatten my input files to not contain more than a single
    -- element per line.
    --     But then I encountered yet another problem: Vivado's
    -- 'readline()' behaved very strangely, reporting the 'length
    -- attribute to be >0 even for otherwise-blank lines.  The issue
    -- as I later found out, was that Vivado, for some reason, treats
    -- files with a '.txt' extension in a special manner and readline()
    -- completely breaks on those; so, I had to rename it to something
    -- else (e.g., '.dat').  This should not happen since 'text' is
    -- defined as 'file of string' in std.textio, and, if I wanted to
    -- have my file read as binary or any other type, I would define
    -- my own 'type binary is file of bit', yet Vivado makes this
    -- (undocumented) assumption on its own...  SEE:
    -- reddit.com/r/FPGA/comments/16th9ok
    --     /textio_not_reading_negative_integers_from_file_in
    --     That seemed to solve the Issue only in simulation, although
    -- read() would still not read more than 7 elements per line; in
    -- synthesis, read() was STILL broken beyond repair.  Eventually,
    -- I found that Vivado actually implemented the read() procedure
    -- correctly ONLY in case of 'bit_vector' and 'std_logic_vector';
    -- NOTHING else---not even simple positive integers---works.
    --     Also, I found out that another limitation is that you CANNOT
    -- detect "blank" lines; for some reason, Vivado will always report
    -- the current line's 'length attribute to be the same throughout
    -- an entire file.  Fortunately, this could be worked-around by
    -- merely using rows full of 'X', 'Z', or 'U' as "delimiters."
    -- inside COMMENTS, and its read() was completely disfunctional).
    --     Never have I had to wrestle so much with a toolchain just to
    -- get something so incredibly trivial to work, but this seems to
    -- be commonplace within EDA tools: I also looked into Synopsys'
    -- Synplify to see if they implemented file I/O properly there,
    -- but Synplify seemed even buggier than Vivado (e.g., it would
    -- break when you used ASCII grave accents and specific keywords
    -- such as protected).
    --
    -- NOTE: Do NOT use 'ulogic' as Vivado's read() is broken for it.
    subtype read_t is std_logic_vector(neural_word'length-1 downto 0);
    --
    -- NOTE: Also, since you cannot leave out the 'out' parameters of
    -- procedures as 'open' in VHDL (unlike entities and components),
    -- I NEED to use this "dummy" placeholder evem if we don't actually
    -- care about the read values and only want to count their numbers.
    alias dummy_t is read_t;
    -- Be aware that you cannot combine 'sread()' (which is broken
    -- anyway) with a non-static while-loop, or else Vivado will
    -- still complain about using an unsynthesizable procedure.
    --constant FAKE_LIMIT : natural := 2**16 - 1;
```

```
124
125     constant ROW_DELIMITER : read_t := (others => 'X');
126     --constant NULL_SLV      : std_logic_vector (0 downto 1) :=
127     --    (others => '0');
128
129
130     -- Helper macros; TODO: Make this package a generic based on these
131     function get_weights_file(
132         layer_path : in string;
133         layer_idx  : in natural
134     ) return string;
135
136     function get_biases_file(
137         layer_path : in string;
138         layer_idx  : in natural
139     ) return string;
140
141     impure function get_num_layers(
142         network_path : in string
143     ) return natural;
144
145     impure function get_network_dimensions(
146         network_dir : in string;
147         num_layers  : in natural
148     ) return dimensions_array;
149
150     function max(x : dimensions_array) return dimension;
151
152 end package file_parser_aux;
153
154 package body file_parser_aux is
155
156     function get_weights_file(
157         layer_path : in string;
158         layer_idx  : in natural
159     ) return string is
160     begin
161         return layer_path & "/weights_" &
162         natural'image(layer_idx) & ".dat";
163     end function get_weights_file;
164
165     function get_biases_file(
166         layer_path : in string;
167         layer_idx  : in natural
168     ) return string is
169     begin
170         return layer_path & "/biases_" &
171         natural'image(layer_idx) & ".dat";
172     end function get_biases_file;
173
174     -- Returns the number of layers in a network, depending on how
175     -- many 'weights' files were present in a given directory.
176     -- Unfortunately, as a side effect, it has to create an extra
177     -- file named "weights_{N+1}.dat" to bypass Vivado's limitations.
178     --
179     -- NOTE: Due to Vivado's nonstandard implementation of file_open(),
180     -- we cannot depend on "open_status = name_error," because Vivado
181     -- just quits whenever it encounters a file that cannot be opened.
182     -- SEE: support.xilinx.com/s/question/0D54U00008CO8pTSAT/
183     --         bug-fileopen-is-not-consistent-with-ieee-standards
184     impure function get_num_layers(
185         network_path : in string
```

```
186     ) return natural is
187         file     test_handle  : text;
188         variable open_status  : file_open_status;
189         variable file_no      : natural := 0;
190     begin
191         -- Exit when there are no more layers/files to process.
192         try_file : loop
193             -- NOTE: In VHDL-2008, there is no 'read_write_mode' as
194             -- in VHDL-2009; this is a rather lengthy workaround.
195             --
196             -- NOTE: Do NOT use 'write_mode'; it wipes the file out.
197             -- use 'append_mode' instead.
198             file_open(
199                 open_status,
200                 test_handle,
201                 get_weights_file(network_path, file_no),
202                 append_mode -- Create it if it doesn't exist
203             );
204             file_close(test_handle);
205 
206             -- 'open_status' doesn't work in Vivado.
207             exit when (open_status /= open_ok);
208 
209             file_open(
210                 open_status,
211                 test_handle,
212                 get_weights_file(network_path, file_no),
213                 read_mode -- Re-open it in readmode
214             );
215 
216             was_empty : if endfile(test_handle)  then
217                 file_close(test_handle);
218                 exit; -- Exit entirely
219             else
220                 -- The file opened was valid & there exists a layer.
221                 file_no := file_no + 1;
222             end if was_empty;
223 
224             file_close(test_handle);
225         end loop try_file;
226 
227         return file_no;
228     end function get_num_layers;
229 
230     -- This function returns the number of rows and columns in a file.
231     impure function get_layer_dimension(
232         layer_dir : in string;
233         layer_idx : in natural
234     ) return dimension is
235         constant sample_file     : string :=
236             get_weights_file(layer_dir, layer_idx);
237         file     file_to_test    : text open read_mode is sample_file;
238         variable current_line    : line;
239         variable dummy_element   : dummy_t;
240         variable read_succeded   : boolean;
241         variable n_rows          : natural := 1; -- No ending delimiter
242         variable n_cols          : natural := 0;
243         variable layer_dimension : dimension;
244     begin
245 
246         count_rows : while not endfile(file_to_test) loop
247             --count_cols : while not endfile(file_to_test) loop
```

```
248
249             readline(file_to_test, current_line);
250             read(current_line, dummy_element, read_succeded);
251
252             if dummy_element = ROW_DELIMITER then
253                 n_rows := n_rows + 1;
254             end if;
255
256             if n_rows = 1 then -- Avoid re-counting multiple times
257                 n_cols := n_cols + 1;
258             end if;
259
260             --end loop count_cols;
261         end loop count_rows;
262
263         assert n_rows > 0
264             report "No rows existed in layer no. "
265             & natural'image(layer_idx) & "."
266                 severity failure;
267         assert n_cols > 0
268             report "No columns existed in layer no. "
269             & natural'image(layer_idx) & "."
270                 severity failure;
271
272         layer_dimension.rows := n_rows;
273         layer_dimension.cols := n_cols;
274
275         return layer_dimension;
276     end function get_layer_dimension;
277
278     -- This returns an array of rows and cols in a network's layers
279     impure function get_network_dimensions(
280         network_dir : in string;
281         num_layers  : in natural
282     ) return dimensions_array is
283         variable network_dimensions :
284             dimensions_array (0 to num_layers-1);
285     begin
286
287         per_layer : for idx in network_dimensions'range loop
288             network_dimensions(idx) :=
289                 get_layer_dimension(network_dir, idx);
290         end loop per_layer;
291
292         return network_dimensions;
293     end function get_network_dimensions;
294
295
296     -- Returns the maximum of either columns# and rows# together
297     -- as a single dimension; takes an array of dimensions.
298     function max(x : dimensions_array) return dimension is
299         variable x_max : dimension := x(x'low); -- Start with the first
300     begin
301         -- Starting from 'low+1 because 'low was assigned beforehand.
302         comparator : for i in x'low+1 to x'high loop
303             x_max.rows := x(i).rows when (x(i).rows > x_max.rows);
304             x_max.cols := x(i).cols when (x(i).cols > x_max.cols);
305         end loop comparator;
306
307         return x_max;
308     end function max;
309
```

```
end package body file_parser_aux;


library std;
    use std.textio.all;

library ieee;
    use ieee.std_logic_1164.all;

library work;
    use work.file_parser_aux.all;

library neural;
    context neural.neural_context;

package file_parser is

    impure function parse_network_from_dir(
        network_dir : string
    ) return network_layers;

end package file_parser;

-- NOTE: In VHDL, you are not able to pass or return ranges (e.g.,
-- '7 downto 0') to or from functions/procedures; so, the start/end
-- both have to be given SEPARATELY.
--
-- NOTE: In VHDL, you also cannot pass 'file' handles/objects; so,
-- the solution is to reduntantly open/close the files EACH time.
package body file_parser is
    -- NOTE Due to VHDL limitations, and to avoid allocators (i.e.,
    -- 'access', 'new', and dynamic concatenation), which are
    -- completely disallowed in synthesis (EVEN to just initialize
    -- other constants) we have to know the number of rows/cols in
    -- each file as a constant BEFORE calling these functions.  In
    -- other words, the '_dim' parameters are the actual dimensions
    -- and not the ones we'd subsequently use for the Resizing Trick.
    -- (Thanks to Mr. Brian Padalino for his dimension-measuring idea.)

    -- NOTE: Make sure to review the following link:
    -- support.xilinx.com/s/question/0D54U00008ADvMRSA1
    -- TL;DR: Vivado (as of 2023.2) has a bug within its
    -- 'read()' procedure of 'real' datatypes that prevents
    -- reading out more than 7 times from a single line.
    --
    -- This means that "rows" hereinafter refer to a _flattened_ list
    -- of elements on multiple lines, NOT multiple elements per line.


    -- Variant for arrays (e.g., biases)
    impure function parse_elements(
        file_path : in string;
        file_dim  : in dimension
    ) return neural_vector is
        file     file_handle : text open read_mode is file_path;
        variable file_row    : line;
        variable row_elem    : read_t; -- Intermediary for conversion
        variable result_arr  : neural_vector (0 to file_dim.rows-1);
    begin
        -- No need for an outer loop; there is only ONE "row" of biases

        parse_row : for col in result_arr'range loop
```

```
372
373             readline(file_handle, file_row);
374             read(file_row, row_elem);
375
376             -- Convert to the internally used fixed-point type.
377             result_arr(col) := to_sfixed(
378                 row_elem, neural_word'high, neural_word'low
379             );
380
381         end loop parse_row;
382
383         return result_arr;
384     end function parse_elements;
385
386     -- Variant for nested arrrays/matrices (e.g., weights)
387     impure function parse_elements(
388         file_path : in string;
389         file_dim  : in dimension
390     ) return neural_matrix is
391         file     file_handle  : text open read_mode is file_path;
392         variable file_row     : line;
393         variable row_elem     : read_t; -- Intermediary for conversion
394         variable dummy_length : natural;
395         variable result_mat   : neural_matrix
396             (0 to file_dim.rows-1) (0 to file_dim.cols-1);
397     begin
398
399         parse_rows : for row in result_mat'range loop
400             parse_cols : for col in result_mat(0)'range loop
401
402                 readline(file_handle, file_row);
403                 read(file_row, row_elem);
404
405                 -- If the row hasn't ended, convert and store the item.
406                 result_mat(row)(col) := to_sfixed(
407                     row_elem, neural_word'high, neural_word'low
408                 );
409
410             end loop parse_cols;
411
412             -- Skip the delimiters (except at EOF)
413             if not endfile(file_handle) then
414                 readline(file_handle, file_row);
415             end if;
416         end loop parse_rows;
417
418         return result_mat;
419     end function parse_elements;
420
421
422     -- NOTE: A very frustrating limitation was that anything earlier
423     -- than VHDL-2019 cannot use 'block' in subprograms (e.g., function
424     -- bodies), and I really needed to use 'block's because they allow
425     -- you to have a varying number of locally scoped constant types.
426     -- Because I have an unknown (though not at compile-time) number of
427     -- layers to parse, I cannot simply hard-code 10 or 100 constant
428     -- types.  So, one way was to constrain the entire holder (record)
429     -- inside the function's declaration region, but this proved a
430     -- challenge: I now could not return my value from the function in
431     -- any way.
432     --     Constraining a record inside a function's LOCAL declaration
433     -- area meant that I could not use it to define the function's own
```

```
434     -- return type, much less use it outside the function for the
435     -- caller's type.  Other than using VHDL-2019 (which, as of 2024,
436     -- is not supported anywhere), a solution to this was using a
437     -- for-generate loop (which DOES have a locally scoped declaration
438     -- region) inside an entity.  However, this meant having to
439     -- instantiate an entity each time you want to use the parser;
440     -- the syntax would look rather clunky that way.
441     --     Ultimately, I chose to take advantage of VHDL-2008's generic
442     -- packages and perform some of the processing (i.e., finding the
443     -- number of layers or their dimensions), which will be used by the
444     -- custom constrained record types, earlier in the package's header
445     impure function parse_network_from_dir(
446         network_dir : string
447     ) return network_layers is
448 
449         constant NUM_LAYERS : natural :=
450             get_num_layers(network_dir);
451         constant LAYER_DIMS : dimensions_array :=
452             get_network_dimensions(network_dir, NUM_LAYERS);
453         constant MAX_DIM    : dimension :=
454             max(LAYER_DIMS);
455 
456         subtype constr_params_arr_t is network_layers
457             (0 to NUM_LAYERS-1)
458             (
459                 weights (0 to MAX_DIM.rows-1) (0 to MAX_DIM.cols-1),
460                 biases  (0 to MAX_DIM.rows-1)
461             );
462         -- SEE: https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/312
463         subtype constr_params_t is constr_params_arr_t'element;
464 
465         variable current_layer : constr_params_t;
466         variable result_arr : constr_params_arr_t;
467     begin
468 
469         per_layer : for i in 0 to NUM_LAYERS-1 loop
470 
471             current_layer.dims    := LAYER_DIMS(i);
472             current_layer.biases  := resize(parse_elements(
473                 get_biases_file(network_dir, i), LAYER_DIMS(i)
474             ), MAX_DIM);
475             current_layer.weights := resize(parse_elements(
476                 get_weights_file(network_dir, i), LAYER_DIMS(i)
477             ), MAX_DIM);
478 
479             result_arr(i) := current_layer;
480         end loop per_layer;
481 
482         return result_arr;
483     end function parse_network_from_dir;
484 
485 
486 end package body file_parser;
487 
488 
489 -- ------------------------------------------------------------------------
490 -- END OF FILE: file_parser.vhd
491 -- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- pipeliner.vhd is a part of Innervator.
-- ------------------------------------------------------------------


-- For background, sometimes operations/logic inside a process might
-- take too long, due to the limitations of the physical world, and
-- therefore not be ready within a single clock cycle; this is referred
-- to as "timing violation" in FPGA design, and it is especially
-- prevalent when using too many combination logic and/or having too
-- fast of a clock (e.g., 500MHz -> 2 ns).  So, a workaround is to
-- "delay" (i.e., spread) the processing of data over multiple clock
-- cycles; this is done by assigning inputs/outputs to one or more
-- "register" signals, which makes said ins/outs reach the outside
-- entities by multiple clock cycles (e.g., 50 ns instead of 10 ns)
-- as to give the FPGA fabric more time to complete the operations
-- we ask it to.  Also, certain built-in blocks, such as DSPs, will
-- "pull in" our externally registered (pipelined) signals and have
-- the same delaying effect.
--     Another type of delay is "routing" delay, which is actually
-- more common nowadays and in newer FPGAs.  Routing delay refers
-- to the actual, physicel delay caused by electricity moving too
-- "slowly" in an internal FPGA route (i.e., wire), and it is often
-- caused when you try to access specific hard-macro components
-- (e.g., a DSP or BRAM) that only exists in a specific location
-- of your FPGA chip, while you have too much logic depend on that,
-- which, in turn, makes it physically impossible for the router to
-- place them all next to said hard-macro instantiation.  The solution
-- is exactly the same: convert the routed path to a "multi-cycle"
-- clock using flip-flop chains.
--     Note that delaying does imply that external users have to
-- "wait" for the processing to be finished before they can supply
-- new data to us; while the old data is going through flip-flop
-- chains, new data could follow after just a single clock cycle.
-- If the data D1 is given at nanosecond 0 and D2 at 10, then
-- (in a double-registering pipeline), output O1 would be returned
-- at nanosecond 20 AND O2 would STILL appear at 30.  In other words,
-- we didn't have to wait for O1 to be fully done before passing D2,
-- and you could think of it as a car assembly line.
--     Be aware that timing violations do not 100% mean that your
-- design won't work in practice.  Synthesis tools account for all
-- extremities when calculating timing violations; they consider
-- worst-case scenarios and ascertain that your described logic will
-- finish under those circumstances within the allocated clock cycle.
-- If you proceed with having tiny timing errors, your design might
-- work just fine, but the moment your FPGA package gets too heated
-- or too cold (for example), it'll have a higher tendency to fail.
--     Also, this closely relates to a "synchronizer," which also
-- uses delay lines to solve a different problem (metastability).
--     Lastly, you might have to enable "retiming" optimizations in
-- your synthesis tools so that they may take advantage of pipelining.

library ieee;
    use ieee.std_logic_1164.all;

-- NOTE: Currently, there is a language inconsistency (reported by me)
-- in VHDL that disallows output ports from being able to use
-- "aggregated" expressions (i.e., tuples) such as (1, 2, 3) or
-- (A, B, C):
--     https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/311
```

```
--
-- had they been allowed, the entity could have been used like so:
--
--    -- Here, we expect that the entity constructs a "trit vector"
--    -- type internally, using it to denote its input/output signals.
--    test_instance : entity work.pipeliner
--        generic map (2, trit)
--        port map (
--            (alpha, beta, gamma), -- This works, as of VHDL-08.
--            (alpha_reg, beta_reg, gamma_reg) -- EXPRESSION ERROR!
--        );
--
-- However, because dis-aggregation is not done on 'out' ports, and
-- because arrays cannot be constructed based on a base type in VHDL
-- versions earlier than 2019, the next interface could have been this:
--
--    -- trit_vector is defined by us earlier in the Architecture.
--    test_instance : entity work.pipeliner
--        generic map (2, trit_vector, 3)
--        port map (
--            (alpha, beta, gamma), -- This works, as of VHDL-08.
--            o_signals(0) => alpha_reg,
--            o_signals(1) => beta_reg,
--            o_signals(2) => gamma_reg
--        );
--
-- Sadly, you cannot "convince" VHDL's strong typing system that the
-- generically provided type is, in fact, an array that you can index
-- over; this means that the next alternative was either "dividing" the
-- interface to take/return one signal at a time, or continue with
-- tuples but provide a duplicate overload for every single array type.
--
-- NOTE: The aforementioned topic also applies equally to generic
-- subprograms (i.e., generically typed procedures) with the difference
-- being that they cannot have indexed/composite port assignments
-- whatsoever; the compiler complains about non-static (?) ports.
--
-- NOTE: Subprograms (like procedures) can take generic arguments,
-- similar to entities.  However, a nuanced difference is that an
-- entity's  'generic(...);' and 'port(...);' clauses are statements
-- (hence the semicolon; after them), while 'generic(...)' and
-- 'parameter(...)' are NOT statements and do not have semicolons.
-- This also means that we cannot declare constants in-between the
-- 'procedure' and 'is' keywords, unlike entities.
package pipeliner is
    procedure registrar
        generic (
            constant g_NUM_STAGES :
                in natural range 2 to natural'high := 3;
            type     t_ARG_TYPE
        )
        parameter (
            signal i_clk    : in  std_ulogic;
            signal i_signal : in  t_ARG_TYPE;
            signal o_signal : out t_ARG_TYPE
        );

end package pipeliner;

-- Instantiate in the following format:
--
--    procedure register_signal is new core.pipeliner.registrar
```

```
--         generic map (3, std_logic_vector);
--
--     register_signal(i_clk, sig_unreg, sig_reg);
--
package body pipeliner is

    -- Works on a single signal at a time
    procedure registrar -- Singular
        generic (
            constant g_NUM_STAGES :
                in natural range 2 to natural'high := 3;
            type     t_ARG_TYPE
        )
        parameter (
            signal i_clk    : in  std_ulogic;
            signal i_signal : in  t_ARG_TYPE;
            signal o_signal : out t_ARG_TYPE
        )
    is
        type t_arg_arr is array (natural range <>) of t_ARG_TYPE;
        -- NOTE: Even though this is a variable, and variables are
        -- purported to update their values immediately, these are,
        -- in reality, no different from normal signals in this case.
        -- Unlike simulation, hardware is not 100% perfect and a
        -- daisy-chain of variables mean that the following variables
        -- would be longer "wires" connecting them to preceding ones,
        -- meaning that there could be a delay (albeit very tiny)
        -- between the time it takes for the preceding variables to
        -- update their own values and the time it takes for variables
        -- later in the chain to take effect of said updates.
        --     I find a lot of tutorials and beginners' guides very
        -- misleading as a result of this; had such pitfalls been
        -- mentioned, variables would not be treated so arcanely.
        --     Also, since variables could effectively be used to
        -- replicate signals' delayed assignment behavior in synthesis,
        -- one cannot help but wonder why VHDL just doesn't let us
        -- declare signals directly in the first place; the reason
        -- is rather arbitrary and lies in (now-ancient) design
        -- choices of VHDL and Ada.
        variable pipeline_regs : t_arg_arr (g_NUM_STAGES-2 downto 0);

    begin
        pipeline_regs(0) := i_signal when rising_edge(i_clk);

        for i in 1 to g_NUM_STAGES-2 loop
            pipeline_regs(i) :=
                pipeline_regs(i - 1) when rising_edge(i_clk);
        end loop;

        o_signal <= pipeline_regs(pipeline_regs'high);
    end procedure registrar;


    -- NOTE: The rationale behind using array types was to
    -- replicate the same "variadic" number of parameters
    -- that we have in C.
    --
    -- TODO: Whenever VHDL-2019 is widespread, make this take
    -- in a base type and construct its array ITSELF; this
    -- way, users won't have to define an otherwise-unused
    -- array type just to pass it to this entity, and they can
    -- use a single entity instantiation for multiple signals.
```

```
    --
    -- https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/311
    /*
    procedure register_signals -- Plural
        generic (
            constant g_NUM_STAGES :
                in natural range 2 to natural'high := 3;

            type t_ELEMENT is private;
            type t_ARRAY is array (natural) of t_ELEMENT
        )
        parameter (
            signal i_clk     : in  std_ulogic;
            signal i_signals : in  t_ARRAY (0 downto 0);
            signal o_signals : out t_ARRAY (0 downto 0)
        )
    is
        -- NOTE: We subtract by 2 because the incoming (input) signal
        -- itself also counts; when we talk about "double registering"
        -- signals, we know that there already WAS a flip-flop register
        -- (i.e., the 'in' signal) and we need to add just 1 more to it
        constant NUM_ELEMS : positive := i_signals'length;
        -- Constrained subtypes of the array's elements.
        subtype t_CONSTRAINED is t_ARG_TYPE (i_signals'element'range);

        procedure register_custom_typed is new registrar
            generic map (g_NUM_STAGES, t_CONSTRAINED);
    begin
        -- Compile-time assertions will go here to ascertain
        -- that the 'input' and 'output' are of equal lengths.
        --assert i_signals'length = o_signals'length
            --report "Aggregated input/output tuples " &
            --       "are not equal in size."
                --severity failure;

        per_signal : for i in 0 to NUM_ELEMS-1 loop
            register_custom_typed(i_clk, i_signals(i), o_signals(i));
        end loop per_signal;
    end procedure register_signals;
    */

end package body pipeliner;


library ieee;
    use ieee.std_logic_1164.all;

entity pipeliner_single is
    generic (
        g_NUM_STAGES : positive := 2;
        type arg_type
    );
    port (
        i_clk    : in  std_ulogic;
        i_signal : in  arg_type;
        o_signal : out arg_type
    );

    type arg_arr_type is array (natural range <>) of arg_type;
end entity pipeliner_single;

architecture behavioral of pipeliner_single is
```

```
    -- Synthesis tools will often replace a series of flip-flops
    -- with better primitives, like shift-registers, that "achieve"
    -- the same delaying effect.  However, we might sometimes WANT
    -- to use flip-flops specifically, so we can turn off that
    -- optimization by using vendor-specific attribute definitions.
    --
    -- Also, note that avoiding the usage of explicit reset signals
    -- may also result in the same shift-register (SRL) conversion.
    --     ednasia.com/coding-consideration-for-pipeline-flip-flops

    /* Xilinx ISE */
    attribute register_balancing : string;
    /* Xilinx Vivado */
    attribute shreg_extract : string; -- No shift-reg. conversion
    /* Altera Quartus */
    attribute syn_allow_retiming : boolean;

    -- TODO: Do we also apply these to input/output or just cascades?
    attribute register_balancing of
        i_signal : signal is "backward";
    attribute shreg_extract of
        i_signal : signal is "no";
    attribute syn_allow_retiming of
        i_signal : signal is true;
    -- Output
    attribute register_balancing of
        o_signal : signal is "backward";
    attribute shreg_extract of
        o_signal : signal is "no";
    attribute syn_allow_retiming of
        o_signal : signal is true;
begin

    -- NOTE: Concurrent assignments mean that there is no
    -- delay involved; both wires "connect" and act as one.
    --
    -- "Delaying" by 1 stage means that we will not have
    -- to place any additional flip-flops in the middle.
    -- The reason is that the unregistered i_signal
    -- would also have a propagation delay of 1 whenever
    -- something is assigned to it.
    single_pipeline : if g_NUM_STAGES = 1 generate
        o_signal <= i_signal; -- CONCURRENT assignment
    -- Otherwise, implement actual delay with flip-flops.
    else generate
        constant STAGES_HIGH : natural := g_NUM_STAGES - 2;
        signal pipeline_regs : arg_arr_type
            (0 to STAGES_HIGH);

        attribute register_balancing of
            pipeline_regs : signal is "backward";
        attribute shreg_extract of
            pipeline_regs : signal is "no";
        attribute syn_allow_retiming of
            pipeline_regs : signal is true;
    begin
        pipeline_chain : for j in 0 to STAGES_HIGH generate
            pipeline_register : process (i_clk) begin
                if rising_edge(i_clk) then
                    pipeline_regs(j) <= i_signal when j = 0
                        else pipeline_regs(j-1);
                end if;
```

```
310             end process pipeline_register;
311         end generate pipeline_chain;
312 
313         o_signal <= pipeline_regs(STAGES_HIGH); -- CONCURRENT assignment
314     end generate;
315 
316 end architecture behavioral;
317 
318 
319 
320 -- ------------------------------------------------------------------------
321 -- END OF FILE: pipeliner.vhd
322 -- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- neuron.vhd is a part of Innervator.
-- ------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library work;
    context work.neural_context;

library config;
    use config.constants.all;

library core;

-- TODO: Generically 'type' these, at least in VHDL-2019.
entity neuron is
    generic (
        g_NEURON_WEIGHTS  : neural_wvector;
        g_NEURON_BIAS     : neural_word;
        /* Sequential (pipeline) controllers */
        -- Number of inputs to be processed at a time (default = all)
        g_BATCH_SIZE      : positive := g_NEURON_WEIGHTS'length;
        -- Number of pipeline stages/cycle delays (defualt = none)
        g_PIPELINE_STAGES : natural := 0
    );
    -- NOTE: There are also other types such as 'buffer' and the
    -- lesser-known 'linkage' but they are very situation-specific.
    -- NOTE: Apparently, it is better to use Active-High (like o_done
    -- when '1' instead of o_busy '1') internal signals in most FPGAs.
    port (
        -- NOTE: Do NOT name these as mere 'input' or 'output' because
        -- std.textio also defines those, and they can conflict.
        i_inputs : in  neural_bvector (0 to g_NEURON_WEIGHTS'length-1);
        o_output : out neural_bit; -- The "Action Potential"
        /* Sequential (pipeline) controllers */
        i_clk    : in  std_ulogic; -- Clock
        i_rst    : in  std_ulogic; -- Reset
        i_fire   : in  std_ulogic; -- Start/fire up the neuron
        o_done   : out std_ulogic  -- Are we done processing the batch?
    );

    constant NUM_INPUTS : positive := i_inputs'length;
begin

    -- TODO: See if the 'instance_name or _path attributes are
    -- supported in Vivado synthesis; if so, use them here.
    assert g_NEURON_WEIGHTS'length mod g_BATCH_SIZE = 0
        report "Size of input data is not evenly divisble " &
               "by the given batch size."
            severity failure;

end entity neuron;


-- TODO: While anything with registers and flip-flops can be called
-- a pipeline, it might still not be very appropriate to call this
-- a pipelined neuron, because the actual registers are used to
-- resolve routing delays and, essentially, function as multi-cycle
```

```
-- paths.  (Revamp this to actually function like a pipeline.)
architecture pipelined of neuron is
    -- TODO: For UNJUSTIFIED reasons, you cannot declare a signal
    -- within a process' decleratory section, and yet variables can
    -- be FUNCTIONALLY the same, in case of counters.  While you
    -- could use 'block' clauses to wrap the process and have
    -- "locally scoped" signals that way, it is still going to add
    -- an extra indention nest, and it is not very ideal.
    -- TODO: Decide if we want to move these otherwise-local signals
    -- to become variables in their respective processes.  (One dis-
    -- advantage is that simulators might not show them in waveforms.)

    -- This is the pipeline's propagation delay counter.  For example,
    -- in a pipeline with 3 stages, the "external" input will take
    -- 3 clock cycles to arrive "inside" the pipelined processor,
    -- and said processor's output (back to the external source)
    -- would also take 3 clock cycles to reach.
    --     Hence, we need to wait (i.e., count each cycle) until the
    -- data is fully loaded into the pipeline, in both directions.
    --     This might need a re-design to be more clear; the reason
    -- it starts at 1 is that the first batch in the pipeline will
    -- have already gotten filled in the "pre-processing" stage.
    signal pipe_delay : natural range 0 to g_PIPELINE_STAGES-1 := 1;
    -- Current iteration number (total number of
    -- iterations = number of data / size of batches).
    --
    -- Because the pipeline is always "ahead" of the processing
    -- multiplier by the number of stages (in clock cycles),
    -- two separate indices are used to keep track.
    signal pipe_iter_idx : natural range 0 to NUM_INPUTS := 0;
    signal proc_iter_idx : natural range 0 to NUM_INPUTS := 0;

    -- The Neuron's Finite-State Machine (FSM)
    type neuron_state_t is (
        idle, initializing, processing, finalizing, activating, done
    );
    signal neuron_state : neuron_state_t := idle;

    -- This is the "localized" version of the input signal; given that
    -- our given input itself might reset or become cleared right after
    -- 'i_fire' is set to high, we need to sample and locally store the
    -- input at that time for later use within the processing stages.
    signal inputs_local  : i_inputs'subtype;
    -- These are the unregistered input and registered output signals.
    -- TODO: Somehow use 'subtype and 'element to derive these
    -- Input data (DSP input 1)
    signal inputs_unreg     : neural_bvector (0 to g_BATCH_SIZE-1);
    signal inputs_reg       : neural_bvector (0 to g_BATCH_SIZE-1);
    -- Weights (DSP input  2)
    signal weights_unreg    : neural_wvector (0 to g_BATCH_SIZE-1);
    signal weights_reg      : neural_wvector (0 to g_BATCH_SIZE-1);
    -- Internal DSP multiplier (product) pipeline
    --
    -- NOTE: If you are using a very small (i.e., < 4) number of
    -- bits for either the integral or fractional part, you may
    -- consider using a slightly larger multiple of the _word type
    -- (e.g., neural_word or neural_qword) here for the accumulator
    -- to accomodate for the many additions that occur within
    -- the inner for-loop and would otherwise overflow.  After the
    -- activation function/clamping takes place, and the variable
    -- gets its range restricited within [0, 1), we can safely
    -- resize it back to a smaller bit width.
```

```
    --     Also, this might result in the synthesizer using
    -- available DSP (dedicated multiplier) blocks on your
    -- FPGA, which would conserve other logic resources.
    signal products_unreg   : neural_dvector (0 to g_BATCH_SIZE-1);
    signal products_reg     : neural_dvector (0 to g_BATCH_SIZE-1);
    -- Multiplied-Accumulated weighted sum (DSP output)
    signal outputs_unreg    : neural_dvector (0 to g_BATCH_SIZE-1);
    signal outputs_reg      : neural_dvector (0 to g_BATCH_SIZE-1);
    -- The activation function (not batched)
    signal activation_unreg : neural_bit;
    signal activation_reg   : neural_bit;

    function activation_function(
        x : neural_dword
    ) return neural_bit is
    begin
        -- TODO: Automatically select between activations, as needed.
        return work.activation.sigmoid(x);
    end function activation_function;


    -- Yet another VHDL annoyance:
    --     stackoverflow.com/questions/31044965/
    --         procedure-call-in-loop-with-non-static-signal-name
    -- In short, procedures cannot take elements from array
    -- signals, such as test_sig(i), even if the index is
    -- static.  For that reason, we have to take indices
    -- rather than pre-indexed array elements.
    --
    -- Multiplier-Accumulator ("MAC")
    procedure multiply_accumulate(
        constant idx        : in  natural;
        signal   mul_a      : in  neural_wvector;
        signal   mul_b      : in  neural_bvector;
        signal   acc_in     : in  neural_dvector;
        signal   acc_out    : out neural_dvector;
        signal   prod_unreg : out neural_dvector;
        signal   prod_reg   : in  neural_dvector
    ) is
        variable products  : neural_dword;
        variable summation : neural_dword;

        -- NOTE: Unfortunately, procedures' 'out' parameters
        -- cannot be assigned to 'open', unlike actual entities/
        -- components; use dummies or placeholders as a workaround.
        variable dummy_carry : std_ulogic;
    begin
        products := resize(
            mul_a(idx)
            * -- Multiply
            resize( -- Resize, if needed
                to_sfixed(mul_b(idx)),
            mul_a(idx)),
        acc_in(idx));

        prod_unreg(idx) <= products;

        add_carry(
            L      => acc_in(idx),
            R      => prod_reg(idx),
            c_in   => '0',
            result => summation,
```

```
186             c_out  => dummy_carry -- IGNORED!
187         );
188 
189         acc_out(idx) <= summation;
190     end procedure multiply_accumulate;
191 
192     -- These cause simulation/synthesis mismatch
193     /*
194     procedure register_inputs is new core.pipeliner.registrar
195         generic map (2, inputs'element);
196 
197     procedure register_weights is new core.pipeliner.registrar
198         generic map (2, g_NEURON_WEIGHTS'element);
199     */
200 begin
201     -- NOTE: This form of pipelining would only fix timing issues
202     -- related to physical routing, not resource/logic consumption.
203     --
204     -- When g_PIPELINE_STAGES is 0, the _reg output and _unreg
205     -- input get concurrently connected (with no delay).
206     no_pipeline : if g_PIPELINE_STAGES = 0 generate
207         create_pipeline : for i in 0 to g_BATCH_SIZE-1 generate
208             inputs_reg(i)   <= inputs_unreg(i);
209             weights_reg(i)  <= weights_unreg(i);
210             products_reg(i) <= products_unreg(i);
211             outputs_reg(i)  <= outputs_unreg(i);
212         end generate create_pipeline;
213 
214         activation_reg      <= activation_unreg;
215 
216     else generate
217         create_pipeline : for i in 0 to g_BATCH_SIZE-1 generate
218             register_inputs   : entity core.pipeliner_single
219                 generic map (g_PIPELINE_STAGES, neural_bit)
220                 port map (i_clk, inputs_unreg(i), inputs_reg(i));
221             register_weights  : entity core.pipeliner_single
222                 generic map (g_PIPELINE_STAGES, neural_word)
223                 port map(i_clk, weights_unreg(i), weights_reg(i));
224             register_products : entity core.pipeliner_single
225                 generic map (g_PIPELINE_STAGES, neural_dword)
226                 port map(i_clk, products_unreg(i), products_reg(i));
227             register_outputs  : entity core.pipeliner_single
228                 generic map (g_PIPELINE_STAGES, neural_dword)
229                 port map(i_clk, outputs_unreg(i), outputs_reg(i));
230         end generate create_pipeline;
231 
232         register_activation   : entity core.pipeliner_single
233             generic map (g_PIPELINE_STAGES, neural_bit)
234             port map(i_clk, activation_unreg, activation_reg);
235 
236     end generate;
237 
238     -- NOTE: Combinational (i.e., un-clocked and stateless) logic,
239     -- sensitive only to changes in inputs, will be much "faster" and
240     -- perform everything in a SINGLE clock cycle.  However, it will
241     -- also use a much, much higher number of logic blocks in the FPGA,
242     -- meaning that a single neuron with 64 inputs could potentially
243     -- take up 10% of a small FPGA's (e.g., Artix-7) LUTs.
244     --     A workaround is to convert the combination logic to a
245     -- sequential (i.e., clocked and stateful) one, where weighted sums
246     -- are calculated in small "batches" in each clock cycle; this
247     -- does have the disadvantage of requiring MULTIPLE clock cycles
```

```
248     -- for the entire calculation to be done (e.g., for 64 inputs and
249     -- a 100MHz clock, the combinational approach would take 10ns while
250     -- the sequential one, with segments of 2, might take 320+20ns).
251     --     Additionally, if you go with the combination approach while
252     -- keeping track of the previous states, you can introduce latches.
253     -- Lastly, if you go with the sequential approach, you also need to
254     -- have additional communication mechanism with the external logic
255     -- to let them know whenever this neuron is done processing its
256     -- batch or whenever it should begin processing the given batch;
257     -- otherwise, since your sequential process is already clocked and
258     -- you can't use the 'input' signal's event in its sensitivity list
259     -- you would have to maintain its previous states and compare them.
260     --
261     -- TODO: Improve the pipelining to actually overlap and be
262     -- continuously fed from the input data.
263     neuron_loop : process (i_clk, i_rst) is
264         -- NOTE: Use 'variable' as opposed to a 'signal' because these
265         -- for-loops are supposed to unroll inside a _single_ tick of
266         -- the Process, meaning that any subsequent assignments to
267         -- a 'singal' accumulator would be DISCARDED; by using
268         -- variables, we can resolve this issue.
269         --
270         -- NOTE: Somehow, using an initial value here adds a huge
271         -- ~3 ns setup timing slack; this should NOT be happening!
272         variable weighted_sum : neural_dword; --:=
273           --resize(g_NEURON_BIAS, neural_dword'high, neural_dword'low);
274
275         -- Number of iterations (if batch processing is enabled)
276         constant ITER_HIGH : natural := NUM_INPUTS - g_BATCH_SIZE;
277
278         procedure perform_reset is
279         begin
280             neuron_state <= idle;
281         end procedure perform_reset;
282     begin
283
284         if not c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
285         elsif rising_edge(i_clk) then
286             if c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
287             else
288
289                 case neuron_state is
290                     when idle =>
291                         neuron_state  <= idle;
292
293                         -- Reset back to default values
294                         proc_iter_idx <= 0;
295                         pipe_iter_idx <= g_BATCH_SIZE; -- 0 in the loop
296                         pipe_delay    <= 1; -- 1 delay's accounted here
297
298                         o_output      <= to_ufixed(0, o_output);
299                         o_done        <= '0';
300
301                         weighted_sum  := -- Start with the Bias
302                             resize(g_NEURON_BIAS, weighted_sum);
303
304                         inputs_local  <= (others =>
305                             to_ufixed(0, inputs_local'element'high,
306                                 inputs_local'element'low)
307                             );
308                         -- Because we will be "adding" these before
309                         -- they are truly filled with calculated
```

```
                    -- values, we initialize them to a known
                    -- (i.e., 0) value at first.
                    products_unreg <= (others =>
                        to_sfixed(0, products_unreg'element'high,
                            products_unreg'element'low)
                        );
                    outputs_unreg  <= (others =>
                        to_sfixed(0, outputs_unreg'element'high,
                            outputs_unreg'element'low)
                        );
                    /*
                    inputs_unreg   <= (others =>
                        to_ufixed(0, inputs_unreg'element));
                    weights_unreg  <= (others =>
                        to_sfixed(0, weights_unreg'element));
                    */

                    if i_fire = '1' then
                        -- Save the external inputs (which
                        -- can change after i_fire).
                        inputs_local <= i_inputs;

                        for i in 0 to g_BATCH_SIZE-1 loop
                            inputs_unreg(i)  <= i_inputs(i);
                            weights_unreg(i) <= g_NEURON_WEIGHTS(i);
                        end loop;

                        -- NOTE: when pipeline stages == 1,
                        -- skip initializing and do the processing.
                        -- Switching cases counts as 1 delay itself
                        if g_PIPELINE_STAGES = 1 then
                            neuron_state <= processing;
                        else
                            neuron_state <= initializing;
                        end if;

                    end if;
                -- ------------------------------------------------

                -- Here, we "wait" (for a number of clock cycles
                -- equal to pipeline stages) so that the first data
                -- arrives through the pipeline; otherwise, the
                -- weighted sum would have uninitialized values.
                when initializing =>
                    neuron_state <= initializing;

                    -- Even though we are "waiting," we should still
                    -- continue to fill the upcoming pipeline stages
                    --
                    -- TODO: Account for the scenario where the
                    -- pipeline stages exceed the number of inputs
                    -- (stop filling the pipeline at that point.)
                    for i in 0 to g_BATCH_SIZE-1 loop
                        inputs_unreg(i)  <=
                            inputs_local(pipe_iter_idx+i);
                        weights_unreg(i) <=
                            g_NEURON_WEIGHTS(pipe_iter_idx+i);
                    end loop;
                    pipe_iter_idx <= pipe_iter_idx + g_BATCH_SIZE;

                    if (pipe_delay < g_PIPELINE_STAGES-1) then
                        pipe_delay <= pipe_delay + 1;
```

```
372                         else
373                             pipe_delay   <= 0; -- Reuse for 'finalizing'
374                             neuron_state <= processing;
375                         end if;
376                     -- ------------------------------------------------
377
378                     when processing =>
379                         neuron_state <= processing;
380
381                         -- TODO: Have a generic switch to toggle
382                         -- between computing the activation function
383                         -- DURING the last batch iteration (1 clock
384                         -- cycle less latency), or AFTER a clock cycle
385                         -- passes, like now (better logic timing).
386
387                         -- NOTE: The pipeline still needs to be filled
388                         -- at g_NUM_STAGES ahead-of-time, but it will
389                         -- also need to stop sooner; this is why we
390                         -- keep track and stop it separately.
391                         -- TODO: Account for when batch processing's off
392                         pipeline_unsaturated : if
393                             pipe_iter_idx < NUM_INPUTS
394                         then
395                             for i in 0 to g_BATCH_SIZE-1 loop
396                                 -- Continue filling the pipeline, which
397                                 -- will eventually reach the multiplier
398                                 -- after a number of clock cycles (i.e.,
399                                 -- pipeline stages).
400                                 inputs_unreg(i)  <=
401                                     inputs_local(pipe_iter_idx+i);
402                                 weights_unreg(i) <=
403                                     g_NEURON_WEIGHTS(pipe_iter_idx+i);
404                             end loop;
405                             pipe_iter_idx <=
406                                 pipe_iter_idx + g_BATCH_SIZE;
407                         end if pipeline_unsaturated;
408
409
410                         -- NOTE: This loop is unrolled into actual
411                         -- hardware; this is why we don't multiply
412                         -- an entire matrix all in one pass (it
413                         -- would be far too much in one clock cycle)
414                         for i in 0 to g_BATCH_SIZE-1 loop
415                             -- This is a running accumulator; the
416                             -- result of the multiplication of each
417                             -- weight by its associated input is
418                             -- resized (IF NEEDED) to the size of
419                             -- the Accumulator and then added to it
420                             multiply_accumulate(
421                                 -- VHDL limitation workaround
422                                 idx        => i,
423                                 -- Numbers to multiply
424                                 mul_a      => weights_reg,
425                                 mul_b      => inputs_reg,
426                                 -- Accumulator
427                                 acc_in     => outputs_reg,
428                                 acc_out    => outputs_unreg,
429                                 -- Internal multiplier pipeline
430                                 prod_unreg => products_unreg,
431                                 prod_reg   => products_reg
432                             );
433                         end loop;
```

```
                    proc_iter_idx <= proc_iter_idx + g_BATCH_SIZE;

                    -- NOTE: Short-circuited to one at compile-time
                    sum_calculated : if
                        -- Not processing in batches (iterate once)
                        (ITER_HIGH  = 0 and
                            proc_iter_idx /= 0) or
                        -- OR: Processing in batches
                        (ITER_HIGH /= 0 and
                            proc_iter_idx >= ITER_HIGH)
                    then
                        neuron_state <= finalizing;
                    end if sum_calculated;
                -- ------------------------------------------------

                -- Here, similar to initializing, we will wait for
                -- the pipelined OUTPUT to "catch up" and finish.
                when finalizing =>
                    neuron_state <= finalizing;

                    -- TODO: Do we also pipeline this?
                    for i in 0 to g_BATCH_SIZE-1 loop
                        weighted_sum := resize(
                            weighted_sum
                            + outputs_reg(i)
                            + products_reg(i),
                        weighted_sum);
                    end loop;

                    if (pipe_delay < g_PIPELINE_STAGES-1) then
                        pipe_delay <= pipe_delay + 1;
                    else
                        pipe_delay   <= 0;
                        if g_PIPELINE_STAGES = 1 then
                            neuron_state <= done;
                        else
                            neuron_state <= activating;
                        end if;
                    end if;
                -- ------------------------------------------------

                -- Wait for activation on the weighted sum.
                -- (It is done here to avoid logic/gate delay
                -- that'd occur in the previous case's branch,
                -- because it also has to be pipelined.)
                when activating =>
                    -- Activate only once
                    if pipe_delay = 0 then
                        activation_unreg <=
                            activation_function(weighted_sum);
                    end if;

                    if (pipe_delay < g_PIPELINE_STAGES-1) then
                        pipe_delay <= pipe_delay + 1;
                    else
                        neuron_state <= done;
                    end if;
                -- ------------------------------------------------

                when done =>
                    o_output     <= activation_reg;

```

```
496                         -- Signal that we're no longer busy,
497                         -- for a single clock cycle
498                         o_done       <= '1';
499 
500                         -- Back to idle state; await new data
501                         neuron_state <= idle;
502                     -- ------------------------------------------------
503 
504                     -- Hardening in case of "unknown" states
505                     when others => -- 1-clock-long cleanup phase
506                         perform_reset;
507                     -- ------------------------------------------------
508                 end case;
509 
510             end if;
511         end if;
512     end process neuron_loop;
513 
514 
515 
516 end architecture pipelined;
517 
518 
519 -- ----------------------------------------------------------------------
520 -- END OF FILE: neuron.vhd
521 -- ----------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- layer.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library work;
    context work.neural_context;

library config;
    use config.constants.all;


-- TODO: Generically 'type' these, at least in VHDL-2019.
entity layer is
    generic (
        g_LAYER_WEIGHTS   : neural_matrix;
        g_LAYER_BIASES    : neural_vector;
        /* Sequential (pipeline) controllers */
        -- Number of inputs to be processed at a time (default = all)
        g_BATCH_SIZE      : positive := g_LAYER_WEIGHTS'element'length;
        -- Number of pipeline stages/cycle delays (defualt = none)
        g_PIPELINE_STAGES : natural := 0
    );
    port (
        i_inputs  : in  neural_bvector
            (0 to g_LAYER_WEIGHTS'element'length-1);
        o_outputs : out neural_bvector
            (0 to g_LAYER_WEIGHTS'length-1);
        /* Sequential (pipeline) controllers */
        i_clk     : in  std_ulogic; -- Clock
        i_rst     : in  std_ulogic; -- Reset
        i_fire    : in  std_ulogic; -- Start/fire up all the neurons
        o_done    : out std_ulogic  -- Is the layer processing done?
    );

    -- NOTE: This assumes that the upper hierarchy (i.e., network)
    -- supplies the sanitized/actual slices of the layer's parameters.
    constant NUM_INPUTS  : positive := i_inputs'length;
    -- NOTE: This one is also the number of "neurons" in this layer.
    constant NUM_OUTPUTS : positive := o_outputs'length;
end entity layer;


architecture dense of layer is -- [Structural arch.]
    signal neurons_done :
        std_ulogic_vector (0 to NUM_OUTPUTS-1);
begin

    neural_layer : for i in 0 to NUM_OUTPUTS-1 generate
        neuron_instance : entity work.neuron
            generic map (
                g_NEURON_WEIGHTS  => g_LAYER_WEIGHTS(i),
                g_NEURON_BIAS     => g_LAYER_BIASES(i),
                g_BATCH_SIZE      => g_BATCH_SIZE,
                g_PIPELINE_STAGES => g_PIPELINE_STAGES
            )
            port map (
```

```
62                  i_inputs => i_inputs,
63                  o_output => o_outputs(i),
64                  i_clk    => i_clk,
65                  i_rst    => i_rst,
66                  i_fire   => i_fire,
67                  o_done   => neurons_done(i)
68              );
69      end generate neural_layer;
70
71      -- TODO: Decide if the Layer's busy signal should be based
72      -- on ALL neurons' busy signals (or just one of them?)
73      --o_busy <= '0' when (neurons_done = (others => '0')) else
74      --          '1' when (neurons_done = (others => '1'));
75      o_done <= neurons_done(0);
76
77  end architecture dense;
78
79
80  -- ------------------------------------------------------------------------
81  -- END OF FILE: layer.vhd
82  -- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- synchronizer.vhd is a part of Innervator.
-- ------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

-- Metastability occurs in flip-flops when the input signal changes
-- too close to the clock edge, violating "setup and hold" times;
-- this leaves the flip-flop in an unresolved state where the
-- output can be unpredictable and could potentially cause errors
-- in connected logic.  To mitigate this, "cascading" multiple
-- flip-flops at the input is a common solution; it provides
-- additional time for the metastable signal to settle into a
-- stable 0 or 1 before being utilized elsewhere in the circuit.
--     This is called "synchronizing" or "de-glitching."
--
-- NOTE: A good portion of this synchronization also overlaps with
-- "pipelining," because both use a series of clocked flip-flops.
-- However, a good reason to separate them is to (hopefully) have
-- the synthesization tool lay out pipelines or synchronizers
-- close to their own respective groups.  Also, the synchronizer
-- might be extended later on (maybe to support multiple clocks)
-- and separating them early-on would be beneficial, in that case.
--
-- TODO: Have a variadic variant for std_(u)logic VECTORS.
entity synchronizer is
    generic (
        g_NUM_STAGES : positive := 2
    );
    port (
        i_clk     : in  std_ulogic;
        i_signal  : in  std_ulogic;
        o_signal  : out std_ulogic
    );
end entity synchronizer;

architecture behavioral of synchronizer is
    -- Synthesis tools will often replace a series of flip-flops
    -- with better primitives, like shift-registers, that "achieve"
    -- the same delaying effect.  However, we might sometimes WANT
    -- to use flip-flops specifically, so we can turn off that
    -- optimization by using vendor-specific attribute definitions.
    --
    -- Also, note that avoiding the usage of explicit reset signals
    -- may also result in the same shift-register (SRL) conversion.
    --     ednasia.com/coding-consideration-for-pipeline-flip-flops

    /* Xilinx Vivado/XST */
    -- Disable the conversion of flip-flops to to shift-registers
    attribute shreg_extract : string;
    -- Specifies that registers receive async. data.
    attribute async_reg     : boolean; -- Also implies DONT_TOUCH.

    -- TODO: See if we should apply this to the input/output?
    -- Input
    attribute shreg_extract of i_signal : signal is "no";
    attribute async_reg     of i_signal : signal is true;
    -- Output
```

```
62      attribute shreg_extract of o_signal : signal is "no";
63      attribute async_reg     of o_signal : signal is true;
64  begin
65
66      -- NOTE: Concurrent assignments mean that there is no
67      -- delay involved; both wires "connect" and act as one.
68      --
69      -- "Delaying" by 1 stage means that we will not have
70      -- to place any additional flip-flops in the middle.
71      -- The reason is that the unregistered i_signal
72      -- would also have a propagation delay of 1 whenever
73      -- something is assigned to it.
74      single_pipeline : if g_NUM_STAGES = 1 generate
75          o_signal <= i_signal; -- CONCURRENT assignment
76      -- Otherwise, implement actual delay with flip-flops.
77      else generate
78          -- NOTE: We subtract by 2 because the incoming (input)
79          -- signal itself also counts; so, when we talk about "double
80          -- registering" something, we know that there already WAS
81          -- a flip-flop register (i.e., the 'in' signal) and
82          -- we need to add just 1 more to it.
83          constant STAGES_HIGH : natural := g_NUM_STAGES - 2;
84
85          -- NOTE: I initially wanted to place this vector as individual
86          -- values inside the for-generate loop, but you cannot refer
87          -- back to a previous iteration of for-generate to access (i-1);
88          -- the solution was to declare it as a vector here.  SEE:
89          -- https://groups.google.com/g/comp.lang.vhdl/c/rm97yoJwcWc
90          signal sync_regs : std_ulogic_vector
91              (0 to STAGES_HIGH) := (others => '0');
92
93          attribute shreg_extract of sync_regs : signal is "no";
94          attribute async_reg     of sync_regs : signal is true;
95      begin
96          cascade_chain : for i in 0 to STAGES_HIGH generate
97              cascade_register : process (i_clk) begin
98                  if rising_edge(i_clk) then
99                      -- If it is the first instance, then we must use
100                     -- the actual input signal; no previously cascaded
101                     -- registers can exist before i=0.
102                     sync_regs(i) <=
103                         i_signal when i = 0 else sync_regs(i-1);
104                 end if;
105             end process cascade_register;
106         end generate cascade_chain;
107
108         o_signal <= sync_regs(STAGES_HIGH); -- CONCURRENT assignment
109     end generate;
110
111 end architecture behavioral;
112
113
114 -- ------------------------------------------------------------------------
115 -- END OF FILE: synchronizer.vhd
116 -- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- uart.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

-- A hierarchical interface for a full- or half-Duplex asynchronous
-- receiver/transmitter in the 8-N-1 frame: eight (8) data bits,
-- no (N) parity bit, and one (1) stop bit, plus an implicit start
-- bit; in this case, only 80% of the throughput is used for the data.
entity uart is
    -- In digital communications, "baud" (i.e., symbol rate) is equal
    -- to the bitrate (bit-rate).  However, when the communications is
    -- modulated to analog, a baud _can_ encode more than 1 bit.
    generic (
        -- TODO: Take data length as a generic.
        g_CLK_FREQ : positive := 100e6;
        g_BAUD     : positive range positive'low to g_CLK_FREQ := 9_600
    );
    -- NOTE: 'Buffer' data flows out of the entity, but the entity can
    -- read the signal (allowing for internal feedback); however, the
    -- signal cannot be driven from outside the entity, unlike inputs.
    port (
        -- UART-Rx/Tx (Receive/Transmit) Shared Ports
        i_clk       : in  std_ulogic; -- Internal FPGA clock
        i_rst       : in  std_ulogic; -- Reset
        -- UART-Rx (Receive) Ports
        i_rx_serial : in  std_logic; -- External connection (wire)
        o_rx_done   : out std_ulogic; -- "Done Reading" signal
        o_rx_byte   : out std_ulogic_vector (7 downto 0); -- LSB first
        -- UART-Tx (Transmit) Ports
        i_tx_send   : in  std_ulogic; -- "Start Sending" signal
        i_tx_byte   : in  std_ulogic_vector (7 downto 0); -- LSB first
        o_tx_active : out std_ulogic; -- Half-Duplex transmitters ONLY
        o_tx_done   : out std_ulogic; -- "Done Transmitting" signal
        o_tx_serial : out std_logic -- External connection (wire)
    );

    -- NOTE: constants here are applied to ALL architectures
    constant TICKS_PER_BIT : positive :=
        positive(g_CLK_FREQ / g_BAUD) - 1;
    constant DATA_HIGH     : natural  := o_rx_byte'high;
begin
end entity uart;


library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

-- UART's standalone sub-components (i.e., UART-RCVR and -XMTR)
package uart_pkg is
 component uart_rcvr
        generic (
            g_CLK_FREQ : positive;
            g_BAUD     : positive
        );
        port (
```

```
62              i_clk       : in  std_ulogic;
63              i_rst       : in  std_ulogic;
64
65              i_serial    : in  std_logic;
66              o_done      : out std_ulogic;
67              o_byte      : out std_ulogic_vector (7 downto 0)
68          );
69  end component uart_rcvr;
70
71  component uart_xmtr
72          generic (
73              g_CLK_FREQ : positive;
74              g_BAUD     : positive
75          );
76          port (
77              i_clk       : in  std_ulogic;
78              i_rst       : in  std_ulogic;
79
80              i_send      : in  std_ulogic;
81              i_byte      : in  std_ulogic_vector (7 downto 0);
82              o_active    : out std_ulogic;
83              o_done      : out std_ulogic;
84              o_serial    : out std_logic
85          );
86  end component uart_xmtr;
87 end package uart_pkg;
88
89
90 -- ------------------------------------------------------------------------
91 -- END OF FILE: uart.vhd
92 -- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- uart_xcvr.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library work;
    use work.uart_pkg.all;

library config;
    use config.constants.all;

-- A half- or full-duplex async. receiver/transmitter (in 8-N-1 frame)
architecture transceiver of uart is -- [Structural arch.]
begin

    receiver_component : component work.uart_pkg.uart_rcvr
        generic map (
            g_CLK_FREQ  => g_CLK_FREQ,
            g_BAUD      => g_BAUD
        )
        port map (
            i_clk       => i_clk,
            i_rst       => i_rst,

            i_serial    => i_rx_serial,
            o_done      => o_rx_done,
            o_byte      => o_rx_byte
        );

    transmitter_component : component work.uart_pkg.uart_xmtr
        generic map (
            g_CLK_FREQ  => g_CLK_FREQ,
            g_BAUD      => g_BAUD
        )
        port map (
            i_clk       => i_clk,
            i_rst       => i_rst,

            i_send      => i_tx_send,
            i_byte      => i_tx_byte,
            o_active    => o_tx_active,
            o_done      => o_tx_done,
            o_serial    => o_tx_serial
        );

    -- [Place for other concurrent statements]

end architecture transceiver;


library work;
    use work.all;

-- NOTE: 'configuration' in VHDL is a barely documented and arcane
-- keyword, and the excessive repetation of component/entity ports
-- might also be defeating its purpose.
--     From what I have gathered, components are "idealized"
```

```
-- _placeholders_ for future "realized" entities.  In an electronics
-- sense, they are chip sockets for upcoming chips; most of the time
-- they are not useful and merely add an unneeded layer of abstraction.
-- However, they could sometimes be useful for giving a hierarchical
-- organization to "sub-entities."
--     Additionally, components should be declared in an architecture's
-- header and correspond exactly (name- & port-wise) to their entities.
-- They will also require separate instantations in said architecture's
-- body (i.e., structural VHDL), often resulting in lots of duplicated
-- port assignments.
--     It is also possible to use configurations to "bind" a specific
-- instance (or even 'all' instances) of a component to a specific
-- entity-architecture pair or other sub-configurations.  Afterward,
-- you may even instatiate the configuration itself as you would do
-- so with an entity or component.
--
-- NOTE: Here, it is important to note that 'for' does NOT refer to
-- a for-loop and means a literal 'for' (i.e., FOR X, use Y).
--
-- NOTE: You cannot leave ports of type 'input' as 'open' but you can
-- assign 'U', 'X', 'Z', or '-' to them to achieve the same effect;
-- among these, '-' makes the most sense, semantically, but 'Z' seems
-- to be the one replicating the 'open' effect in schematics.
configuration uart_xcvr of uart is -- [config. name] of entity
    for transceiver -- (i.e., the encapsulating architecture)

        for all : work.uart_pkg.uart_rcvr -- (i.e., component instance)
            use entity work.uart (receiver)
                generic map (
                    g_CLK_FREQ  => g_CLK_FREQ,
                    g_BAUD      => g_BAUD
                )
                port map (
                    i_clk       => i_clk,
                    i_rst       => i_rst,

                    i_rx_serial => i_serial,
                    o_rx_done   => o_done,
                    o_rx_byte   => o_byte,

                    i_tx_send   => 'Z',
                    i_tx_byte   => (others => 'Z'),
                    o_tx_active => open,
                    o_tx_done   => open,
                    o_tx_serial => open
                );
        end for;

        for all : work.uart_pkg.uart_xmtr -- (i.e., component instance)
            use entity work.uart (transmitter)
                generic map (
                    g_CLK_FREQ  => g_CLK_FREQ,
                    g_BAUD      => g_BAUD
                )
                port map (
                    i_clk       => i_clk,
                    i_rst       => i_rst,

                    i_rx_serial => 'Z',
                    o_rx_done   => open,
                    o_rx_byte   => open,

```

```
124                     i_tx_send   => i_send,
125                     i_tx_byte   => i_byte,
126                     o_tx_active => o_active,
127                     o_tx_done   => o_done,
128                     o_tx_serial => o_serial
129                 );
130         end for;
131 
132     end for;
133 end configuration uart_xcvr;
134 
135 
136 -- ----------------------------------------------------------------------
137 -- END OF FILE: uart_xcvr.vhd
138 -- ----------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- network.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library work;
    context work.neural_context;

library config;
    use config.constants.all;

entity network is
    generic (
        g_NETWORK_PARAMS  : network_layers;
        g_BATCH_SIZE      : positive;
        g_PIPELINE_STAGES : natural
    );
    port (
        i_inputs  : in  neural_bvector
            (0 to g_NETWORK_PARAMS(g_NETWORK_PARAMS'low).dims.cols-1);
        o_outputs : out neural_bvector
            (0 to g_NETWORK_PARAMS(g_NETWORK_PARAMS'high).dims.rows-1);
        /* Sequential (pipeline) controllers */
        i_clk     : in  std_ulogic; -- Clock
        i_rst     : in  std_ulogic; -- Reset
        i_fire    : in  std_ulogic; -- Start/fire up all the layesr
        o_done    : out std_ulogic  -- Is the network done processing?
    );

    constant NUM_LAYERS  : positive := g_NETWORK_PARAMS'length;
    -- Number of neurons in the first (i.e., input) layer.
    constant NUM_INPUTS  : positive := i_inputs'length;
    -- Number of neurons in the last (i.e., output) layer.
    constant NUM_OUTPUTS : positive := o_outputs'length;
end entity network;


architecture neural of network is

    -- Helper functions to "deflate" (or sanitize) max-sized arrays
    -- introduced due to the workaround around VHDL's lack of
    -- variable-sized elements in arrays.  You can find more details
    -- on the specifics of this workaround (and how to reverse it)
    -- in the file_parser.vhd file.
    function deflate( -- Arrays
        inflated_data : neural_vector;
        size_key      : natural
    ) return neural_vector is
        constant deflated_array : neural_vector (0 to size_key-1) :=
            inflated_data (0 to size_key-1);
    begin
        return deflated_array;
    end function deflate;

    function deflate( -- Matrices (i.e., Nested Arrays of Arrays)
        inflated_data : neural_matrix;
        size_key      : dimension
```

```
    ) return neural_matrix is
        variable deflated_matrix : neural_matrix
            (0 to size_key.rows-1) (0 to size_key.cols-1);
    begin
        deflate_rows : for i in deflated_matrix'range loop
            deflated_matrix(i) := -- Deflate individual sub-arrays.
                deflate(inflated_data(i), size_key.cols);
        end loop deflate_rows;

        return deflated_matrix;
    end function deflate;

    -- Unfortunately, VHDL has a language-level limitation where it
    -- does not allow you to refer back to a previous instance of a
    -- for-generate's local signals, even though you could manually
    -- "unroll" it into a single 'block' clause, containing mangled
    -- names of signals that cannot collision.  A solution is to
    -- employ the same max-dimension workaround from file_parser.vhd.
    -- Fortunately, the max-dimension has already been calculated by
    -- the file parser, earlier; we get to re-use it here.
    --     Unused/dummy elements would get discarded and optimized
    -- by the synthesizer, but this can still be very redundant
    -- and clunky, especially for nested arrays and very large ones.
    signal layers_done    : std_ulogic_vector (0 to NUM_LAYERS-1);
    -- This is a NESTED array of 'neural_bvector' (an array type).
    --
    -- NOTE: We cannot use the 'element attribute here; because
    -- each element of the parameter array is a record on its own,
    -- toolchains will break apart on such complicated expressions.
    signal layers_outputs : neural_bmatrix (0 to NUM_LAYERS-1)
        (0 to g_NETWORK_PARAMS(0).weights'length-1);

    -- The entire network is done whenever the last layer of it is.
    signal network_done    : std_ulogic;
    -- The output of the entire network is its layer layer's output.
    -- This could be arg-max'ed or used as-is, as needed.
    signal network_outputs : neural_bvector (0 to NUM_OUTPUTS-1);
begin

    -- NOTE: This is a "feed-forward" neural network, meaning that
    -- each layer's output is connected to the proceeding layer's
    -- input, in a chain-like formation.
    neural_network : for i in 0 to NUM_LAYERS-1 generate
        -- NOTE: These are intermediary signals to get around a VHDL
        -- limitation where you cannot have "conditionally mapped"
        -- port or generic maps (even in static for-generate clauses);
        -- we first assign the condition to a signal and then the port
        --     Because these are considered "concurrent" signal
        -- connections, there's no assignment or clock cycle delay.
        --     Also, in < VHDL-2019, you cannot define these as
        -- constants, because you cannot use when...else in them.
        -- TODO: See if we can somehow get the UNCONSTRAINED subtype
        -- of 'inputs' here, rather than hard-coding it.
        signal inputs_im : neural_bvector
            (0 to g_NETWORK_PARAMS(i).dims.cols-1);
        signal i_fire_im : i_fire'subtype;

        -- NOTE: in VHDL, you cannot constrain port assignments
        -- directly, because these would count as "locally non-static
        -- ranges," even though they are constants at compile-time;
        -- we have to constrain each generated instance's input here.
        constant sanitized_weights : neural_matrix :=
```

```
124             deflate(
125                 g_NETWORK_PARAMS(i).weights,
126                 g_NETWORK_PARAMS(i).dims
127             );
128         constant sanitized_biases  : neural_vector :=
129             deflate(
130                 g_NETWORK_PARAMS(i).biases,
131                 g_NETWORK_PARAMS(i).dims.rows
132             );
133     begin
134         -- NOTE: Somehow, using when...else instead of if..generate
135         -- seems to result in indices such as -1; it seems that the
136         -- condition after the 'else' part is "evaluated" even if
137         -- it is not suppoesd to be [i.e., when i=0, it tries to do
138         -- i-1 and index the array as (-1)].
139         input_layer_condition : if i = 0 generate
140             inputs_im <= i_inputs;
141             i_fire_im <= i_fire;
142         else generate
143             inputs_im <= layers_outputs(i-1)
144                 (0 to g_NETWORK_PARAMS(i).dims.cols-1);
145             i_fire_im <= layers_done(i-1);
146         end generate input_layer_condition;
147 
148         neural_layer : entity work.layer (dense)
149             generic map (
150                 -- NOTE: These arrays are "sliced" due to a workaround,
151                 -- which was used to bypass the lack of variable-sized
152                 -- arrays in VHDL; said workaround (explained in the
153                 -- file_parser.vhd file) would declare an array to be
154                 -- the "maximum" possible size (the biggest out of all
155                 -- its elements) and keep track of their "true" sizes
156                 -- in a separate field called .dims.  Here, we have
157                 -- simply sliced the "inflated" array, discarding the
158                 -- unused/dummy elements based on the true sizes.
159                 g_LAYER_WEIGHTS   => sanitized_weights,
160                 g_LAYER_BIASES    => sanitized_biases,
161                 g_BATCH_SIZE      => g_BATCH_SIZE,
162                 g_PIPELINE_STAGES => g_PIPELINE_STAGES
163             )
164             port map (
165                 i_inputs  => inputs_im,
166                 o_outputs => layers_outputs(i)
167                     (0 to g_NETWORK_PARAMS(i).dims.rows-1),
168                 i_clk     => i_clk, -- Clock
169                 i_rst     => i_rst, -- Reset
170                 -- The first layer (i.e., the "input layer") will
171                 -- activate whenever the network is told to.
172                 -- Each subsequent layer will "fire" (i.e., activate)
173                 -- whenever its previous layer is "done" processing.
174                 i_fire    => i_fire_im,
175                 o_done    => layers_done(i) -- Is it done processing?
176             );
177 
178     end generate neural_network;
179 
180     network_done    <= layers_done(layers_done'high);
181     network_outputs <= layers_outputs(layers_outputs'high)
182         (0 to NUM_OUTPUTS-1);
183 
184     -- The Network's 'done' signal is different than the that of the
185     -- neurons in the sense that it "stays" done; that is to let us
```

```
    -- know that that it has finished its processing and remains IDLE,
    -- ready to accept another set of data.  Neurons' done signals
    -- lasted for a single clock cycle and then switched back to 0;
    -- because each layer's done signal was connected to the following
    -- layers' 'fire' signal, continuing to keep said layer's done
    -- at 1 would result in proceeding layers never ending firing.

    toggle_out : process (i_clk, i_rst) is
        procedure perform_reset is
        begin
            o_done <= '1';
            o_outputs <= (others => (others => '0'));
        end procedure perform_reset;
    begin

        if not c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
        elsif rising_edge(i_clk) then
            if c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
            else

                if network_done = '1' then
                    o_done  <= '1';
                    o_outputs <= network_outputs;
                -- "elsif" so 'network_done' lasts at least one cycle.
                elsif i_fire = '1' then
                    o_done  <= '0';
                    o_outputs <= (others => (others => '0'));
                end if;

            end if;
        end if;
    end process toggle_out;

    /*
    toggle_out : process (all) is
    begin

        if network_done = '1' then
            o_done  <= '1';
            outputs <= network_outputs;
        -- "elsif" so 'network_done' lasts at least one cycle.
        elsif i_fire = '1' then
            o_done  <= '0';
            outputs <= (others => (others => '0'));
        end if;

    end process toggle_out;
    */
end architecture neural;


-- ------------------------------------------------------------------------
-- END OF FILE: network.vhd
-- ------------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- math.vhd is a part of Innervator.
-- ------------------------------------------------------------------------

library ieee;
    use ieee.std_logic_1164.all;

library neural;
    context neural.neural_context;


-- TODO: Implement more activation functions (e.g., ReLU, etc.)
package math is

    -- Returns the index of the highest number in an unsigned array.
    -- TODO: What if two numbers are equal?
    -- TODO: Maybe provide a pipelined/multi-cycle variant.
    function arg_max(
        values : neural_bvector
    ) return natural;

end package math;


package body math is

    function arg_max(
        values : neural_bvector
    ) return natural is
        variable max_index   : natural;
        variable current_max : neural_bvector'element;
    begin

        current_max := (others => '0');

        for i in values'range loop
            if values(i) > current_max then
                max_index := i;
                current_max := values(i);
            end if;
        end loop;

        return max_index;
    end function arg_max;

end package body math;

-- ------------------------------------------------------------------------
-- END OF FILE: math.vhd
-- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- uart_rcvr.vhd is a part of Innervator.
-- ------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library config;
    use config.constants.all;

-- A simplex async. receiver (in 8-N-1 frame)
architecture receiver of uart is -- [RTL arch.]
    -- Receiver's Finite-State Machine (FSM)
    type uart_rx_state_t is (
        idle, started, reading, done
    );
    signal uart_rx_state : uart_rx_state_t := idle;

    -- Synchronized signal (from a metastable input signal).
    signal synced_serial : std_ulogic;

    signal tick_cnt  : natural range 0 to TICKS_PER_BIT := 0;
    signal bit_index : natural range 0 to DATA_HIGH := 0;
begin

    -- NOTE: The input signal is assumed to have been synchronized/
    -- deglitched beforehand, at the top module.
    synced_serial <= i_rx_serial; -- CONCURRENT assignment


    -- NOTE: As a little history, the reason why the 'start bit' is
    -- checked to be 'active low' as opposed to 'active high' is
    -- because physical cable connections could run far and were also
    -- suspectible to damage along the path; by constantly driving an
    -- 'active high' singal you could know for sure that (long) pauses
    -- meant a disruption in the line, unlike what would otherwise be
    -- interpreted as intentional "silence" in an 'active low' setup.
    receive : process (i_clk, i_rst) is
        procedure perform_reset is
        begin
            uart_rx_state <= idle;
        end procedure perform_reset;
    begin

        if not c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
        elsif rising_edge(i_clk) then
            if c_RST_SYNC and i_rst = c_RST_POLE then perform_reset;
            else
                -- NOTE: There is no need for an 'others' default case,
                -- because we are dealing with enumerated types and not
                -- std_logics; all enumerated cases are accounted for.
                -- SEE: sigasi.com/tech/
                --          vhdl-case-statements-can-do-without-others
                -- Despite this, the 'others' case can still be used to
                -- harden the state machine (against radiation, maybe)
                -- and make it safer by recovering from unknown values.
                --
                -- NOTE: Always assign a value to signals in a state
                -- machine; otherwise, latches (hard to synth, slow, &
```

```
62                  -- prone to metastability) may be inferred. Also, since
63                  -- we cannot really use default values to solve that
64                  -- issue here, we instead re-assign the same state in
65                  -- branches that do not result in a change of state.
66                  case uart_rx_state is
67                      -- Due to UART being asynchronous, a "start bit" is
68                      -- utilized by the external transmitter to signal
69                      -- that the actual data bits are forthcoming; the
70                      -- start bit is simply a falling edge (from idle
71                      -- high to a low pulse) immediately followed by
72                      -- user data bits.
73                      when idle =>
74                          uart_rx_state <= idle;
75
76                          -- Reset back to default values
77                          o_rx_done <= '0';
78                          o_rx_byte <= (others => '0');
79                          bit_index <= 0;
80                          tick_cnt  <= 0;
81
82                          if synced_serial = '0' then
83                              uart_rx_state <= started;
84                          end if;
85                      -- ------------------------------------------------
86
87                      -- It is important to note that, even after the
88                      -- start bit it detected, we still re-check the
89                      -- start bit at its middle (one-half 'bit time')
90                      -- as to make sure it was really valid.  If not,
91                      -- it is considered a spurious pulse (noise)
92                      -- and is ignored.
93                      --     Also, by waiting for the mid-point and
94                      -- only then moving to the next state, we'd
95                      -- also be shifting future data reads (sampling)
96                      -- to their respective mid-points.
97                      when started =>
98                          uart_rx_state <= started;
99
100                         if (tick_cnt < TICKS_PER_BIT / 2) then
101                             tick_cnt <= tick_cnt + 1; -- Not middle
102                         else -- Reached the middle
103                             if (synced_serial = '0') then -- Real start
104                                 tick_cnt <= 0; -- Found mid; reset cnt
105                                 uart_rx_state <= reading; -- Begin read
106                             else -- Spurious pulse; ignore
107                                 uart_rx_state <= idle;
108                             end if;
109                         end if;
110                     -- ------------------------------------------------
111
112                     when reading =>
113                         uart_rx_state <= reading;
114
115                         if (tick_cnt < TICKS_PER_BIT) then
116                             tick_cnt <= tick_cnt + 1;
117                         else
118                             tick_cnt <= 0;
119                             -- NOTE: Data is sent with Least-Sig. Byte
120                             -- first; holding type should be 'downto'
121                             o_rx_byte(bit_index) <= synced_serial;
122
123                             -- Check if more bits remain to be received
```

```
124                             if (bit_index < DATA_HIGH) then
125                                 bit_index <= bit_index + 1;
126                             else
127                                 bit_index     <= 0;
128                                 uart_rx_state <= done;
129                             end if;
130                         end if;
131                     -- ---------------------------------------------
132 
133                     -- Once the data bits finish, a "stop bit" will
134                     -- indicate so; the stop bit is normally a
135                     -- transition back to the idle state or remaining
136                     -- at the high state for an extra bit time.
137                     -- A second (optional) stop bit can be configured,
138                     -- usually to give the receiver time to get ready
139                     -- for the next frame, but that is uncommon.
140                     --     Considering how we already know the frame's
141                     -- size beforehand, it might seem unnecessary to
142                     -- even have a stop bit.  However, the UART
143                     -- does not depend on the start bit itself, for
144                     -- synchronization, but the falling _edge_ between
145                     -- the previous stop bit AND the start bit.
146                     -- There might not be such an edge without both
147                     -- the start and stop bits.
148                     when done =>
149                         uart_rx_state <= done;
150 
151                         if (tick_cnt < TICKS_PER_BIT) then
152                             tick_cnt <= tick_cnt + 1;
153                         else
154                             -- TODO: Error-check the stop bit as an
155                             -- extra protection against de-syncing.
156                             --if (synced_serial /= '1') then
157                             --    perform_reset;
158                             --end if;
159 
160                             o_rx_done     <= '1';
161                             tick_cnt      <= 0;
162                             uart_rx_state <= idle;
163                         end if;
164                     -- ---------------------------------------------
165 
166                     -- Hardening in case of "unknown" states
167                     when others => -- 1-clock-long cleanup phase
168                         perform_reset;
169                     -- ---------------------------------------------
170                 end case;
171 
172             end if;
173         end if;
174     end process receive;
175 
176 end architecture receiver;
177 
178 
179 -- ------------------------------------------------------------------
180 -- END OF FILE: uart_rcvr.vhd
181 -- ------------------------------------------------------------------
```

```vhdl
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- uart_xmtr.vhd is a part of Innervator.
-- ------------------------------------------------------------------------


library ieee;
    use ieee.std_logic_1164.all;

library config;
    use config.constants.all;

-- TODO (when needed)
-- A simplex async. transmitter (in 8-N-1 frame)
architecture transmitter of uart is -- [RTL arch.]
    -- Placeholder
begin
    -- Placeholder
end architecture transmitter;


-- ------------------------------------------------------------------------
-- END OF FILE: uart_xmtr.vhd
-- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
-- prescaler.vhd is a part of Innervator.
-- ------------------------------------


-- ------------------------------------------------------------------------
-- END OF FILE: prescaler.vhd
-- ------------------------------------------------------------------------
```

```
-- ------------------------------------------------------------------
-- SPDX-License-Identifier: LGPL-3.0-or-later or CERN-OHL-W-2.0
--
-- Innervator: Hardware Acceleration for Neural Networks
--
-- Copyright (C) 2024  Fereydoun Memarzanjany
--
-- This hardware-descriptive model is free hardware design dual-
-- licensed under the GNU LGPL or CERN OHL v2 Weakly Reciprocal: you
-- can redistribute it and/or modify it under the terms of the...
--     * GNU Lesser General Public License as published by
--       the Free Software Foundation, either version 3 of the License,
--       or (at your option) any later version; OR
--     * CERN Open Hardware Licence Version 2 - Weakly Reciprocal.
--
-- This is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU Lesser General Public License for more details.
--
-- You should have received a copy of the GNU Lesser General
-- Public License and the CERN Open Hardware Licence Version
-- 2 - Weakly Reciprocal along with this.  If not, see...
--     * <https://spdx.org/licenses/LGPL-3.0-or-later.html>; and
--     * <https://spdx.org/licenses/CERN-OHL-W-2.0.html>.
--
-- ------------------------------------------------------------------


-- NOTE: All of the source files herein conform to the RFC 678
-- plaintext document standard, as well as the Ada 95 Quality and
-- Style Guide 2.1.9 (Source Code Line Length); it is good readability
-- practice to limit a line of code's columns to 72 characters (which
-- include only the printable characters, not line endings or cursors).
--     I specifically chose 72 (and not some other limit like 80/132)
-- to ensure maximal compatibility with older technology, terminals,
-- paper hardcopies, and e-mails.  While some other guidelines permit
-- more than just 72 characters, it is still important to note that
-- American teletypewriters could sometimes write upto only 72, and
-- older code (e.g., FORTRAN, Ada, COBOL, Assembler, etc.) used to
-- be hand-written on a "code form" in corporations like IBM; said
-- code form typically reserved the first 72 columns for statements,
-- 8 for serial numbers, and the remainder for comments, which was
-- finally turned into a physical punch card with 80 columns.
--     Even in modern times, the 72 limit can still be beneficial:
-- you can easily quote a 72-character line over e-mail without
-- requiring word-wrapping or horizontal scrolling.
--     As a sidenote, the reason that some guidelines, like PEP 8
-- (Style Guide for Python Code), recommended 79 characters (i.e.,
-- not 80) was that the 80th character in a 80x24 terminal might
-- have been a bit hard to read.

-- Thanks to yet another arcane bug within Vivado 2024, in which it
-- completely breaks apart when you try to access attributes of this
-- constant within the same declaratory region (even though it is
-- perfectly valid VHDL and ModelSim also has no problems with it),
-- we have no choice but to declare it in a "separate" area:
library work;
    use work.constants.all;
library neural;
    context neural.neural_context;
```

```
    use     neural.file_parser.all;
package attribute_bugfix is
    constant debug_NETWORK_OBJECT : network_layers :=
        parse_network_from_dir(c_DAT_PATH);
end package attribute_bugfix;


library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

library work;
    use work.constants.all;

library core;

library neural;
    context neural.neural_context;
    use     neural.file_parser.all;


entity neural_processor is
    port (
        i_clk  : in std_ulogic;
        i_rst  : in std_ulogic;
        i_uart : in std_logic;
        o_uart : out std_logic;
        o_led  : out std_logic_vector (3 downto 0)
    );
begin
end entity neural_processor;

architecture structural of neural_processor is
    alias NETWORK_OBJECT is work.attribute_bugfix.debug_NETWORK_OBJECT;

    -- Number of layers in network (excluding the input data themself)
    constant NUM_LAYERS  : positive :=
        NETWORK_OBJECT'length;
    -- Number of neurons in the first (i.e., input) layer
    constant NUM_INPUTS  : positive :=
        NETWORK_OBJECT(NETWORK_OBJECT'low).dims.cols;
    -- Number of neurons in the last (i.e., output) layer
    constant NUM_OUTPUTS : positive :=
        NETWORK_OBJECT(NETWORK_OBJECT'high).dims.rows;


    -- Synchronized/deglitched ports
    signal i_rst_synced    : std_ulogic;
    signal i_uart_synced   : std_ulogic;
    -- In case the reset button requires inverting
    signal i_rst_corrected : std_ulogic;
    -- Synchronized buttons
    signal i_rst_debounced : std_ulogic;


    /* UART signals */
    signal byte_read_done  : std_ulogic := '0';
    signal byte_read_value : std_logic_vector (7 downto 0) :=
        (others => '0');

    -- The input array that will be received via UART from a computer.
    signal input_data          : neural_bvector (0 to NUM_INPUTS-1) :=
```

```
        (others => (others => '0'));
    signal input_data_count    : natural range 0 to NUM_INPUTS := 0;
    signal input_data_received : std_ulogic := '0';

    -- UART Transmitter signals
    signal result_ready : std_ulogic;
    signal result_byte  : std_logic_vector (7 downto 0);

    signal network_done       : std_ulogic := '0';
    signal network_outputs    : neural_bvector (0 to NUM_OUTPUTS-1);
    -- TODO: Have an actual function to binary-encode decimals.
    signal network_prediction : unsigned (3 downto 0); -- Arg-Max'ed
begin
    -- TODO: Print network metadata here (though Vivado has assert bugs)
    --assert false
    --    report natural'image(NETWORK_OBJECT'element.weights'length)
    --        severity failure;

    /*
        Port Setup
    */

    -- Synchronize/deglitch the incoming data, allowing it to be used
    -- in our own clock domain, avoiding metastabiliy problems.
    synchronize_input_ports : block
    begin
        sync_reset   : entity core.synchronizer
            generic map (g_NUM_STAGES => c_SYNC_NUM)
            port map (
                i_clk    => i_clk,
                i_signal => i_rst,
                o_signal => i_rst_synced
            );
        sync_uart_in : entity core.synchronizer
            generic map (g_NUM_STAGES => c_SYNC_NUM)
            port map (
                i_clk    => i_clk,
                i_signal => i_uart,
                o_signal => i_uart_synced
            );
    end block synchronize_input_ports;

    -- Invert the reset button; even if the FPGA board has a negative
    -- reset, the FPGA might work "better" with positive resets,
    -- internally. (more info in config.vhd)
    -- TODO: Investigate if the reset will have skew, in this case
    invert_reset : if c_RST_INVT generate
        i_rst_corrected <= not i_rst_synced;
    else generate -- Else, don't invert
        i_rst_corrected <= i_rst_synced;
    end generate invert_reset;

    -- Remove the bouncing "noise" from input buttons
    debounce_buttons : block
    begin
        debounce_reset : entity core.debouncer
            generic map (g_TIMEOUT_MS => c_DBNC_LIM)
            port map (
                i_clk    => i_clk,
                i_button => i_rst_corrected,
                o_button => i_rst_debounced
            );
```

```
186     end block debounce_buttons;
187
188
189
190     /*
191         Part Instantiations
192     */
193
194     uart_transceiver : configuration core.uart_xcvr
195         generic map (
196             g_CLK_FREQ => c_CLK_FREQ,
197             g_BAUD     => c_BIT_RATE
198         )
199         port map (
200             i_clk       => i_clk,
201             i_rst       => i_rst_debounced,
202             -- The Receiver Component
203             i_rx_serial => i_uart_synced,
204             o_rx_done   => byte_read_done,
205             o_rx_byte   => byte_read_value,
206             -- TODO: Implement the UART Transmitter, too.
207             i_tx_send   => 'Z',
208             i_tx_byte   => (others => 'Z'),
209             o_tx_active => open, -- Unused
210             o_tx_done   => open, -- Unused
211             o_tx_serial => open
212         );
213
214
215     -- TODO: Eventually, have the option to select between using LUTRAM
216     -- (which is, confusingly, also known as DistRAM or DRAM) and BRAM,
217     -- the dedicated---but single/dual channel---block ram on FPGAs.
218     -- TODO: Also, maybe have this in a separate entity?
219     receive_data : process (i_clk)
220         procedure perform_reset is
221         begin
222             input_data          <= (others => (others => '0'));
223             input_data_count    <= 0;
224             input_data_received <= '0';
225         end procedure perform_reset;
226     begin
227         if not c_RST_SYNC and i_rst_debounced = c_RST_POLE
228             then perform_reset;
229         elsif rising_edge(i_clk) then
230             if c_RST_SYNC and i_rst_debounced = c_RST_POLE
231                 then perform_reset;
232             else
233                 input_data_received <= '0';
234
235                 data_remain : if (input_data_count < NUM_INPUTS) then
236                     byte_read : if (byte_read_done = '1') then
237
238                         input_data(input_data_count) <=
239                             to_ufixed(byte_read_value,
240                                 input_data'element'high,
241                                 input_data'element'low
242                             );
243
244                         input_data_count <= input_data_count + 1;
245                     end if byte_read;
246                 else -- All of the data got received
247                     input_data_received <= '1';
```

```
248                     input_data_count    <= 0;
249                 end if data_remain;
250
251             end if;
252         end if;
253     end process receive_data;
254
255
256     neural_engine : entity neural.network
257         generic map (
258             g_NETWORK_PARAMS  => NETWORK_OBJECT,
259             g_BATCH_SIZE      => c_BATCH_SIZE,
260             g_PIPELINE_STAGES => c_PIPE_STAGE
261         )
262         port map (
263             i_inputs  => input_data,
264             o_outputs => network_outputs,
265             i_clk     => i_clk,
266             i_rst     => i_rst_debounced,
267             i_fire    => input_data_received,
268             o_done    => network_done
269         );
270
271
272     -- NOTE: Unlike popular beliefs, subprograms (like functions)
273     -- CAN be concurrently called outside of processes.
274     -- TODO: Pipeline and clock this later.
275     network_prediction <= to_unsigned(
276         neural.math.arg_max(network_outputs),
277         network_prediction'length
278     );
279
280     -- TODO: Transfer the data back using the UART as opposed to LEDs
281     -- and also check network_done first.
282     o_led(3) <= network_prediction(3);
283     o_led(2) <= network_prediction(2);
284     o_led(1) <= network_prediction(1);
285     o_led(0) <= network_prediction(0);
286     /*
287     transmit_result : process (i_clk)
288         procedure perform_reset is
289         begin
290
291         end procedure perform_reset;
292     begin
293         if not c_RST_SYNC and i_rst_debounced = c_RST_POLE
294             then perform_reset;
295         elsif rising_edge(i_clk) then
296             if c_RST_SYNC and i_rst_debounced = c_RST_POLE
297                 then perform_reset;
298             else
299
300             end if;
301         end if;
302     end process transmit_result;
303     */
304
305 end architecture structural;
306
307
308 -- ------------------------------------------------------------------------
309 -- END OF FILE: main.vhd
```

```
310 -- ------------------------------------------------------------------
```