LSU EE 4755 Homework 2 Due: 11 October 2021

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2021/hw02.v.html`.

Problem 0: If necessary, follow the instructions at `https://www.ece.lsu.edu/koppel/v/proc.html` to set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw02.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Background

The flurry of activity machine learning is due to the success of deep neural networks (DNNs) in providing much improved solutions to otherwise hard-to-tackle problems such as natural language translation and image recognition. Deep neural network consists of multiple layers (more than two or three, otherwise they would not be deep). A *fully connected* layer computes matrix/vector products. The matrix coefficients are called *weights*, and in typical computations there are a large number of weights, so many that performance is limited by the time needed to move them around. Normally with n_i input neurons and n_o output neurons, there would be $n_i n_o$ weights, one for each input/output pair. One way to reduce the number of weights is to not require a weight for each input/output pair. In trained networks many weights are close to zero, so their removal ought to have little effect. If inference hardware (the hardware that computes the output of a layer) supports *sparse* weights then the network can be trained taking into account that some weights will be zero.

Sparsity is easier said than done because it makes the task of moving inputs and their weights to a functional unit (a multiply/add unit) more difficult. One way of lessening the difficulty is limiting which weights can be set to zero. NVidia Volta-generation GPUs support sparsity in which each group of four inputs used to compute one output is limited to two non-zero weights. Two inputs will go unused for that output (but may be used for others.)

In this assignment a module for sparse computation will be completed, `nn_sparse`. Like the Nvidia design it will operate on four inputs. But unlike the Nvidia design it can operate in both sparse and dense modes, determined by a `fmt` input. In dense mode there are four weights, but those weights have a very low precision. In sparse mode there are two weights with higher precision.

There are two challenges. One is a Verilog coding issue: instantiating an `nn2` module (see problem description) for the sparse case, and connecting it to the correct inputs, and making sure the `nn2` output reaches the module output. The other challenge is to do this in a way that maintains high performance. That is, the wider multipliers used for the sparse case will take more time and so we want to take care to not increase the critical path more than is necessary.

Testbench Output

The testbench will instantiate the `nn_sparse` module with several different parameter sets. It will then present dense and sparse patterns and check for the correct outputs. In the unmodified code all of the dense patterns should pass but nearly all of the sparse patterns should fail.

The testbench will show details on the first four errors for each configuration, followed by a tally of the total. Here is a sample showing the last error and the tally:

```
Error tn=4 for fmt 0101  084cca0 =  4.7993 !=  4.4000 (correct)
      1.0000 2.0000 + 1.2000 2.0000
      2.0000      + 2.4000
      acc1 = 0806640 = 2.1997
Done with ex6,ac18,in12,wd3 5000 tests, 2555, 0  sp, den errors found.
For ex6,ac18,in12,wd3  max diff 21132739836.039532, 0.097594  sp, den.
```

Here is what is shown for each reported error: `tn` gives a test number, `fmt` shows the value of the `fmt` input. Note that `fmt[0]` is the least significant big. After the format the error line shows the output value in hexadecimal and decimal. In the sample above they are `084cca0 =  4.7993`. After that the correct (or at least what the testbench assumes is correct) value is shown: `4.4000 (correct)`. The next line shows the expression to be computed, which will consist of four terms if the error is for a dense calculation and two terms for a sparse calculation. The example above is for a sparse calculation. The next line, `2.0000        + 2.4000`, shows the products.

Finally, the value of an object in the module is shown, `acc1`. It is shown in hexadecimal, and in decimal. Note that `acc1` is floating point, so that the hexadecimal value will let you see the sign, exponent, and significand. In most cases though, it will let you see if the value has any `x` or `z` bits.

You are encouraged to add code at this point to print out values of other signals in your module. The code to do that is:

```
// Feel free to modify or add to this to help with your solution.
$write( "       acc1 = %h = %.4f\n",
           nnsp.acc1, conv#(wexp,wsig_ac)::ftor(nnsp.acc1));
```

The `nn_sparse` instance is named `nnsp`, so `nnsp.acc1` refers to an object in the module. The function `conv#(wexp,wsig_ac)::ftor(X)` converts `X` from a floating-point format with exponent length `wexp` and significand length `wsig_ac` into a `real`. The code for this function is in `hw02.v`. Any object could be named, but remember to adjust the `$write` for data type, and the parameters to `conv` if necessary.

To aid in debugging the testbench starts out with sparse patterns in which only one weight is 1 and the others are zero. It will then use weights of 2, 0.1, 10.1. It will repeat the pattern again with two non-zero weights. After that it will use randomly chosen weights and formats. Feel free to modify the testbench to aid in your debugging. Keep in mind that the ta-bot won't test your module using the testbench in your file so removing the tests that your module fails won't help.

Synthesis Script

The synthesis script will synthesize the module at two different target delays. It takes a significant amount of time to run, so only one set of parameters is included. Feel free to modify the script, `syn.tcl` to add other sets.

Problem 1: Module `nn_sparse`, has one w_o-bit output, `o`, four w_i-bit inputs, `i[0]` to `i[3]`, a w_w-bit input, `w`, and a four-bit input, `fmt`. Input `w` can carry either two or four values, called *weights*. If `fmt=4'b1111` then `w` carries four weights, each $w_w/4$ bits. These are called *dense* weights. Otherwise `w` carries two weights, each $w_w/2$ bits, called *sparse* weights. To help get started quickly the module assigns the dense weights to four-element net `wd`.

The module is to compute `o` in one of two possible ways, depending on the value of `fmt`. When `fmt=4'b1111` the module computes `o` using the dense weights and all four values of `i`: $o = i_0w_0 + i_1w_1 + i_2w_2 + i_3w_3$, where i_0 and w_0 are values of `i[0]` and `wd[0]`. The Verilog code to do this is already in the module.

The module should work for six additional values of `fmt`: `4'b0011`, `4'b0110`, `4'b1100`, `4'b1010`, `4'b0101`, and `4'b1001`, these will be referred to as the *sparse formats*. For each of these the module should set the output to $o = i_aW_0 + i_bW_1$, where W_0 and W_1 are the two sparse weights and where a is the position of the rightmost (least significant) `1` in `fmt` and b is the position of the leftmost (most significant) `1` in `fmt`. For example, if `fmt=4'b0011` then $a = 0$ and $b = 1$ and the hardware should compute $o = i_0W_0 + i_1W_1$, and if `fmt=4'b1010` then $a = 1$ and $b = 3$ and the hardware $o = i_1W_0 + i_3W_1$.

All values are floating-point. They share a common exponent, specified by parameter `wexp`. The width of the significand of the output is specified by parameter `sig_ac`, the width of the significand of the inputs is specified by `wsig_in`, and the width of the significand of the dense weights is specified by parameter `wsig_wd`. The layout follows IEEE 754: The most significant bit is a sign bit, that is followed by the exponent, and that is followed by the significand. So the total size of the output is `1+wexp+wsic_ac`.

To compute the dense output `nn_sparse` instantiates three modules: two `nn2` modules and `fp_add`. The `nn2` module computes $i_0 w_0 + i_1 w_1$. The `nn2` module instantiates two `hy_mult` and one `fp_add` (both described below). Details on the `nn2`, including parameters, can be learned by inspecting the module (it is in the homework file).

The `fp_add` module is a convenience wrapper around the Chipware `CW_fp_add` module.

Module `hy_mult` wraps `CW_fp_mult`, but it provides functionality that you'd think would be part of the Chipware library. Unlike the Chipware module, `hy_mult` can be instantiated so that the multiplier, multiplicand, and product each have different significand sizes, though they all share the same exponent size. (The `hy` is for hybrid, referring to the different sizes.) The module instantiates the Chipware module using the product significand size. It then widens (or shrinks) the significands of the multiplier and multiplicand inputs (called `a` and `b`). The inputs are widened by placing zeros in the least significant bits of the widened significands. This was done with the hope that the synthesis program, when performing optimization, would see that these bits were zero and so optimize away the affected partial products. Experiments using Genus (version 211) confirmed that optimization was occurring.

(*a*) The table below shows synthesis script output for the hybrid multiplier at a variety of sizes. Based on this table there is a good and bad way to connect the hybrid multiplier. Design your module taking this data into account. In the table the parameter values are concatenated with the module name, and numbers are added on the end to avoid duplicating a name. Remember that with a large delay target cost is the only goal, and with a 1ns goal speed is the primary goal.

For purposes of interpreting the data below, assume your design will be instantiated with parameters `{wexp 5} {wsig_ac 14} {wsig_in 8} {wsig_wd 4}`. (These can be found in the synthesis script.)

```
Module Name                               Area   Delay   Delay
                                                Actual  Target
hy_mult_wsig_a5_wsig_b5_wsig_p20         62541    7.466  100.000 ns
hy_mult_wsig_a10_wsig_b10_wsig_p20      131839   12.799  100.000 ns
hy_mult_wsig_a10_wsig_b5_wsig_p20        84546   10.636  100.000 ns
hy_mult_wsig_a5_wsig_b10_wsig_p20        92111    9.851  100.000 ns
hy_mult_wsig_a15_wsig_b5_wsig_p20       108593   13.440  100.000 ns
hy_mult_wsig_a5_wsig_b15_wsig_p20       123209   12.643  100.000 ns
hy_mult_wsig_a4_wsig_b8_wsig_p14         71354    8.435  100.000 ns
hy_mult_wsig_a8_wsig_b4_wsig_p14         63890    9.007  100.000 ns
hy_mult_wsig_a14_wsig_b8_wsig_p14       131244   12.047  100.000 ns
hy_mult_wsig_a8_wsig_b14_wsig_p14       144388   11.824  100.000 ns
hy_mult_wsig_a3_wsig_b7_wsig_p12         59985    7.737  100.000 ns
hy_mult_wsig_a7_wsig_b3_wsig_p12         53501    8.081  100.000 ns
hy_mult_wsig_a12_wsig_b7_wsig_p12       110260   12.113  100.000 ns
hy_mult_wsig_a7_wsig_b12_wsig_p12       117097   11.660  100.000 ns
hy_mult_wsig_a5_wsig_b5_wsig_p20_22     130160    2.398    1.000 ns
hy_mult_wsig_a10_wsig_b10_wsig_p20_22   324729    3.046    1.000 ns
hy_mult_wsig_a10_wsig_b5_wsig_p20_22    189191    2.690    1.000 ns
hy_mult_wsig_a5_wsig_b10_wsig_p20_22    214533    2.684    1.000 ns
hy_mult_wsig_a15_wsig_b5_wsig_p20_22    248189    2.742    1.000 ns
```

```
hy_mult_wsig_a5_wsig_b15_wsig_p20_22    302877   2.900   1.000 ns
hy_mult_wsig_a4_wsig_b8_wsig_p14_22     171041   2.369   1.000 ns
hy_mult_wsig_a8_wsig_b4_wsig_p14_22     135568   2.232   1.000 ns
hy_mult_wsig_a14_wsig_b8_wsig_p14_22    296160   3.030   1.000 ns
hy_mult_wsig_a8_wsig_b14_wsig_p14_22    321123   3.232   1.000 ns
hy_mult_wsig_a3_wsig_b7_wsig_p12_22     127217   2.308   1.000 ns
hy_mult_wsig_a7_wsig_b3_wsig_p12_22     132936   1.994   1.000 ns
hy_mult_wsig_a12_wsig_b7_wsig_p12_22    263353   2.823   1.000 ns
hy_mult_wsig_a7_wsig_b12_wsig_p12_22    260279   2.951   1.000 ns
```

(*b*) Modify `nn_sparse` so that it computes the correct outputs for both sparse and dense inputs, and is coded for higher speed. Since a sparse weight is larger than a dense weight a multiplier designed to use sparse weights would cost more and take more time than one designed for dense weights. But, when computing sparse weights only one addition operation is needed. Design your module so that this benefit is realized.

Modify only `nn_sparse` to solve the problem, and use the provided FP units. (Contact me if you feel modifying other modules is needed. (Note that you are free to modify the testbench and related files to help with debugging. But the solution itself should only involve changes to `nn_sparse`.)

Solving this problem requires good debugging skills. Use SimVision (see the course procedures page) to view what is going on inside your module. Also take advantage of the testbench output, and don't hesitate to modify it so that it provides tests that will help you better understand your module.