LSU EE 4755 Homework 3 Due: 13 October 2020

The deadline has been extended by one day, to 13 October (late at night) due to power outages caused by Hurricane Delta.

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/2020/hw03.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if you haven't already), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw03.v`.

Homework Overview and Neural Network Background

The goal of Homework 2 was to describe a 4×4 matrix/vector multiply circuit hierarchically. That goal is generalized here where an $n_i \times n_o$ matrix is multiplied by an n_i-element vector. In Homework 2 each `ao[o]` was computed by a tree connection of multipliers. Here both linear and tree connections will be tried. Also, the module in this assignment will optionally do something about overflow.

The modules in this assignment and in Homework 2 could be used any place where matrix/vector multiplication is needed, but they were designed with a particular application in mind that some students might have guessed from the names used: artificial neural networks. The `nn` prefix is for neural network. The output and one input name starts with `a`, that's for *activation*, which can be though of as a neuron. The weights model connections between neurons.

A completely connected neural network layer performs a matrix vector multiplication. The multiply/add operation needed to compute that is also an important operation for other compute-intensive workloads, including graphics and many forms scientific computation. General-purpose CPUs and GPUs were designed in part to perform multiply/add operations efficiently—on some workloads, including graphics and scientific computation.

One thing that sets neural network (a technique for *machine-learning [ML]*) workloads apart is operand precision. Graphics uses 32-bit values for coordinates, many scientific computation uses 64-bit values. Lower precision would be less effective. But machine learning can get by with less precision, and with different precision for the weights than the activations. Lower precision reduces the amount of energy needed for computation (which is often a limiter), and the amount of data that needs to be moved. This is especially important for weights in fully-connected layers.

The modules in this assignment allow for different precision for inputs, outputs, and weights. When the precision of the output is low there is a danger of overflow. That is often handled by saturating a value at the maximum representable quantity.

Reference Module, `nnOxIbe`

A goal of this assignment is to write a Verilog description of a module performing the same computation as a reference module, `nnOxIbe`. Module `nnOxIbe` has two inputs, an n_i-element vector of w_i-bit integers, `ai`, and an $n_o \times n_i$ matrix of w_w-bit integers, `wht`; the module has one output, an n_o-element vector of w_o-bit integers, `ao`, where n_i, n_o, w_i, w_w, and w_o are the values of the similarly named module parameters. All integers are unsigned. Output `ao` is set to the product of matrix `wht` and column vector `ai` with overflow handled as described further below.

Most will find it easiest to inspect the code in `nnOxIbe` (below) to resolve any remaining certainty about what this module does. For the others let $r(p) = \sum_{q=0}^{n_i-1} H_{p,q} a_i(q)$, where $H_{p,q}$ is the equivalent of the Verilog `wht[p][q]`, and $a_i(q)$ is the equivalent of `ai[q]`. Then either

$a_o(p) = \min\{\, r(p),\, 2^{w_o} - 1 \,\}$ or $a_o(p)$ is set to the low w_o bits of $r(p)$, depending on the value of parameter `sat`.

Module `nnOxIbe` initially computes a 32-bit precision value (see variable `acc`) for each `ao[i]`. If `sat=0` then `ao[i]` is assigned the low `wo` bits of this value. If `sat!=0` then `ao[i]` is set to the minimum of `acc` and $2^{w_o} - 1$. As some may have guessed, `sat` is short for saturating arithmetic. (In saturating arithmetic an overflow is replaced by the maximum representable value. For example, for 4-bit unsigned integers and a saturating add: $11_2 + 1110_2 = 1111_2$.)

```
module nnOxIbe
  #( int no = 4, ni = 4, wo = 10, wi = 4, ww = 5, sat = 0 )
   ( output logic [wo-1:0] ao[no],
     input uwire  [wi-1:0] ai[ni],  input uwire [ww-1:0] wht[no][ni] );

   // The maximum possible value of each element of ao.
   localparam logic [wo-1:0] smax = ~wo'(0);

   always_comb
     for ( int o = 0;  o < no;  o++ ) begin
        automatic int unsigned acc = 0;
        for ( int i=0; i<ni; i++ ) acc += ai[i] * wht[o][i];

        // If sat is non-zero replace a value that would overflow
        // ao[o] with the maximum value that ao[o] can hold.
        ao[o] = sat && acc > smax ? smax : acc;
     end

endmodule
```

Testbench

(This part is best read after looking at Problems 1 and 2.) The testbench will instantiate sixteen (as of this writing) configurations of `nnOxI`. For each configuration, three sets of tests are performed, similar to the ones performed for Homework 2. A grand total of errors is printed at the end, such as `Total number of errors:  660`. Above that the number of errors are grouped in various ways. For example:

```
All Sat 0      220 errors.
All Sat 1      220 errors.
All Sat 2      220 errors.
Linear         330 errors.
Linear Sat 0   110 errors.
Linear Sat 1   110 errors.
Linear Sat 2   110 errors.
Tree           330 errors.
Tree Sat 0     110 errors.
Tree Sat 1     110 errors.
Tree Sat 2     110 errors.
Total number of errors: 660
```

The line reading `Linear 330 errors` shows the total number of errors of all configurations for which `tr=0`. The line `Linear Sat 0 110 errors.` shows the number of errors on linear modules with `sat=0`.

Further up specific inputs and incorrect outputs are shown. For example:

```
** Starting tests for no=3, ni=5, wo=15, wi=9, ww=8, sat=2
Testing module Linear

** Starting test set n12  (1 outputs, 2 inputs) for Linear **
Error test # 0, output 0: z != 32767 (correct)
Error test # 1, output 0: z != 32767 (correct)
Error test # 2, output 0: z != 26759 (correct)
Done with 10 n12 tests on Linear: 10 errors found.
```

In the example above output `ao[0]` was `z` (unconnected) but should have been 32767.

In test set `n12` the inputs and weights are chosen so that the only non-zero output should be `ao[0]` and so that only `ai[0]` and `ai[1]` are non-zero. In set `n1*` all inputs can have non-zero values but weights are chosen so that only `ao[0]` is non-zero. In test set `n**` all inputs can be non-zero and all outputs can be non-zero.

Problem 1: Complete module `nnOxI` so that it produces the same output as `nnOxIbe` and does so using generate statements to either describe a linear or recursive module as described below.

Module `nnOxI` is to be the starting point in all cases. It has the same parameters as `nnOxIbe`, plus it also has a parameter `tr`. The solution to this problem requires modification to `nnOxI` and to module `nn1xI`. Both are in `hw03.v`.

Multiplication and addition of values should be performed by instances of the provided arithmetic modules, `nnAdd`, `nnMult`, and `nnMADD` (multiply/add). These modules can perform saturating arithmetic.

Module `nnOxI` should instantiate `no` (that's a number) `nn1xI` modules. Each `nn1xI` instance should compute one output of `nnOxI`. The `tr` parameter in `nnOxI` indicates whether each `nn1xI` should compute its output using a linear arrangement of modules or a tree arrangement.

For the linear arrangement `nn1xI` should use a generate loop to instantiate `nnMADD` modules. The critical path (without optimization) should be $O(n_i)$ multiply/add operations. For the tree arrangement `nn1xI` should either instantiate two copies of itself or for the base case, the arithmetic modules.

For an example of a module describing a linear arrangement of hardware see `min_n` in the generate/elaborate lecture code, `https://www.ece.lsu.edu/koppel/v/2020/l025-gen-elab.v.html`. For an example of a module describing a tree arrangement of hardware see `min_t` in the lecture code.

Be sure to specify the appropriate parameters when instantiating modules, including the `sat` parameter.

- Do not make ports wider than they need to be.
- Make sure that the modules pass all tests.
- Make sure that the module is synthesizable. (Use command `genus -files syn.tcl` to synthesize.) The area should be > 0.
- Code should be clearly written.

Problem 2: Module `nnOxIbe` honors the `sat` parameter after it has computed a 32-bit `ao[o]` value. (That is, it first computes a 32-bit result, then it checks if it's too large.) That's fine for software, but it would be wasteful for our hardware because we'd need to provide 32-bit precision

hardware for all arithmetic. Or is it really that wasteful? First, we don't necessarily need 32 bits. The maximum value of `ao[o]` depends on `wi`, `ww`, and `ni`, so we only need enough bits to hold that. Also, the saturating arithmetic modules may be inflating cost for two reasons: the cost of detecting and handling saturation, and the fact that algebraic optimizations are impeded when saturation is performed. So, it may be less expensive to compute a value for `ao[o]` to a precision greater than `wo`, and then just saturate that value. This way saturation is performed once per output, rather than `ni` times.

Modify your modules so that when `sat=2` saturation is performed as described above.