LSU EE 4755 **Homework 2** **Due: 25 September 2017**

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `https://www.ece.lsu.edu/koppel/v/2017/hw02.v.html`.

Problem 1: Let $f\|\Re \rightarrow \Re$ be an approximately linear function, for example, $f(x)$ might be the height of a tree that has been growing for x years. Suppose further that we have two pairs of values for the function, $a_1 = f(x_1)$ and $a_2 = f(x_2)$ with $x_1 < x_2$. (For example, $x_1 = 1$ year and $a_1 = 1\,\text{m}$, and $x_2 = 2$ years and $a_2 = 1.5\,\text{m}$.) To compute values of x between x_1 and x_2 we can use linear interpolation $a(x) = a_1 + (x - x_1)\frac{a_2 - a_1}{x_2 - x_1}$. Notice that $a(x_1) = a_1$ and $a(x_2) = a_2$.

Module `interp_behav` has four 32-bit inputs `x1`, `a1`, `x2`, and `a2`. Each will hold a number in `shortreal` representation, their values indicate the endpoints of a region to linearly interpolate, as described in the previous paragraph. Input `j`, `jw` bits, is expected to be $\lfloor x - x_1 \rfloor$, where x is a value to interpolate. (For example, suppose $x_1 = 20$. For $x = x_1 = 20$, input `j` would be `0`, for $x = 25.7$, `j=5`.) Output `aj` is an 8-bit integer and is to be set to $\lfloor a(x_1 + j) \rfloor$. Though input `j` and output `aj` are integers, it's important that some of the calculation be done in floating-point to avoid rounding errors, such as the computation $\frac{a_2 - a_1}{x_2 - x_1}$. (Note: If you're having trouble following this, don't worry. All you really need to do is to look at what `interp_behav` is doing.) Finally, 1-bit output `valid` is set to `1` if $\lfloor x_1 \rfloor + j \leq \lfloor x_2 \rfloor$ and `0` otherwise.

```
module interp_behav
  #( int jw = 12,  amax = 255 )
   ( output logic valid,                    output logic [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,     input uwire [jw-1:0] j );

   always_comb begin

      automatic shortreal x1r = $bitstoshortreal(x1);
      automatic shortreal x2r = $bitstoshortreal(x2);
      automatic shortreal a1r = $bitstoshortreal(a1);
      automatic shortreal a2r = $bitstoshortreal(a2);

      automatic int x1i = $floor(x1r);
      automatic int x2i = $floor(x2r);
      automatic int xj = x1i + j;

      shortreal dadx, ajr;

      valid = xj <= x2i;

      dadx = ( a2r - a1r ) / ( x2r - x1r );
      ajr = a1r + j * dadx;
      aj = ajr < 0 ? 0 : ajr > amax ? amax : $floor(ajr);

   end

endmodule
```

The code in `interp_behav` computes these values using behavioral code. It is not synthesiz-

able because it uses operators to perform floating-point arithmetic. Module `interp` has the same connections as `inter_behav`, and has some starter code. Modify module `interp` so it computes the same values and is synthesizable. To do so instantiate ChipWare modules to perform floating-point operations. Module `interp` already instantiates an adder and a float-to-int converter. Find additional modules in the ChipWare documentation, which can be found on the course references page. When using a ChipWare module **put in an include directive at the end of the file**. See the end of `hw02.v` for examples.

```
module interp
  #( int jw = 12, amax = 255 )
   ( output logic valid,                    output logic [7:0] aj,
     input uwire [31:0] x1, a1, x2, a2,     input uwire [jw-1:0] j );

   uwire [jw:0] x1i, x2i;

   fp_ftoi #( jw+1 ) ftoi1(x1i, x1);
   fp_ftoi #( jw+1 ) ftoi2(x2i, x2);

   uwire [31:0] sum;
   fp_add add1(sum, x1, x2);  // An instantiation example, not otherwise useful.

   // These are obviously incorrect, but they avoid synthesis errors.
   assign      aj = {1'b0, sum[6:0]};
   assign      valid = sum[8];

endmodule
```

The testbench will test module `interp`, it should initially report lots of errors. Of course, when you are done there should be zero errors.

Follow the synthesis steps on the course procedures page to determine if `interp` is synthesizable. If the elaborate step is successful then the module is synthesizable.

Problem 2: Floating-point hardware is relatively costly. Compare the cost of FP and integer arithmetic units by synthesizing equivalent FP and integer adders and dividers. Wrap the ChipWare modules in your own modules, (such as `fp_add` in `hw02.v`) and set parameters so the FP and integer units are comparable. Then modify the synthesis script, `syn.tcl`, so that it will synthesize these modules. The modules should be added to the list assigned on the `set modules` line.

Based on the data collected above, indicate how much less you think the cost would be of an `interp` module that used integer arithmetic.