EEC 281 - Homework/Project #4

Winter 2020

Work individually, but I strongly recommend working with someone in the class nearby so you can help each other when you get stuck, with consideration of the Course Collaboration Policy. Please send me email if something is not clear and I will update the assignment using green font.

Notes:

- **Submit:** (1) all *.v hardware and necessary testing code you wrote (no generated or provided files), and (2) other requested items such as diagrams:
    i. A paper copy of (1) and (2) [instructions], and
    ii. An electronic copy of (1) uploaded to Canvas (under "Assignments") in a tar or zip file. Label directories or files so it is clear to which problem they belong. For example, prob1.v, prob1.vt,...
- **Diagrams.** If a problem requires a diagram, include details such as datapath, memory, control, I/O, pipeline stages, word widths in bits, etc. There must be enough detail so that the exact *functional* operation of the block can be determined by someone with your diagram and explanation, and a reasonable knowledge of what simple blocks do. A satisfactory diagram may require multiple pages of paper taped together into a single large sheet.
- **Verilog.** If a problem requires a verilog design, turn in paper copies of both hardware and test verilog code.
    - *** Where three '*'s appear in the description, perform the required test(s) and turn in a printout of either:
        1. a table printed by your verilog testbench module listing all inputs and corresponding outputs,
        2. a simvision waveform plot which shows (labeled and highlighted) corresponding inputs and outputs, or
        3. verilog test code which compares a) your hardware circuit and b) a simple reference circuit (using high-level functions such as "+")—no third circuit. Include two copy & paste sections of text from your simulation's output (one for pass, and one for fail where you purposely make a *very small* change to either your designed hardware circuit or your reference circuit to force the comparison to fail) that look something like this:
```
input=0101, out_hw=11110000, out_ref=11110000, ok
...

input=0101, out_hw=11110000, out_ref=11110001, Error!
...
```

      For 1 and 3, the output must be copied & pasted directly from the simulator's output without any modifications.
    - In all cases, **Show how you verified** the correctness of your simulation's outputs.
    - Keep "hardware" modules separate from testing code. Instantiate a copy of your processing module(s) in your testing module (the highest level module) and drive the inputs and check the outputs from there.
- **Synthesis.** If a problem requires synthesis, turn in paper copies of the following. Print in a way that results are easy to understand but conserves paper (multiple files per page, 8 or 9 point font, multiple columns). Delete sections of many repeated lines with a few copies of the line plus the comment: `<many lines removed>` .
    1. `dc_compile` (or equivalent)
    2. `*.area` file
    3. `*.log` file; Edit and reduce "Beginning Delay Optimization Phase" and "Beginning Area-Recovery Phase" sections.
    4. `*.tim` file; first (longest) path only

  The "`always @(*)`" verilog construct may be used but keep an eye out for any situations where Design Compiler may not be compatible with it.

  Run all compiles with "medium" effort. Do not modify the synthesis script except for functional purposes (e.g., to specify source file names).
- **Functionality.** For each design problem, you must write by hand 1) whether the design is fully functional, and 2) the failing sections if any exist.
- **Point deductions/additions.** `TotalProbPts` is the sum of all points possible.
    - [Up to `TotalProbPts × 50%`] point reduction for not plainly certifying/showing that your circuit is **functionally correct**. This sounds drastic but you should have checked the correctness of your circuits' outputs anyway, it is impractical for the grader to check every result of every submission by eye, and thus an un-certified design will be treated like a marginally-functional design after a cursory glance at the hardware. Here is an example of a fine way to certify correctness, if the "Y/N" is written either a) by hand individually for each test or b) automatically with a golden reference checker.
```
  inA      inB     outExp  outMantissa     I Certify Correct
-------- --------  ------ --------------   -----------------
10101100 00110101  110010 01100110100101           Y
00000101 10110101  101010 01010101010101           Y
01010100 11101010  010100 11010101100101           no   // this indicates I recognize there is an error here
```
    - [Up to `TotalProbPts × 10%`] point reduction if parts of different problems are mixed up together (please don't do it; it makes grading much

more difficult than you probably realize)

- [Up to `TotalProbPts × 10%`] extra credit will be given for especially thorough, well-documented, or insightful solutions.

- **Clarity.** For full credit, your submission must be easily readable, understandable, and well commented.

Total: 400 points

1. [150 pts] Design a block which calculates the X output of the complex radix-2 DIT FFT butterfly.

```
X =  A + BW
```

The latency may be as many cycles as needed however the multipliers must be the only logic inside their own pipeline stages.

The block's I/O signals are described below. Recall that since there is no decimal point in the hardware, you may think of the inputs as being in any x.x format you like. Having done that, the decimal point of the output will be fixed and you will need to take that into consideration when comparing in matlab.

- `clk` input
- `A` input
  16-bit fixed-point signed 2's complement complex (a_r, a_i)
- `B` input
  16-bit fixed-point signed 2's complement complex (b_r, b_i)
- `W` input
  16-bit fixed-point signed 2's complement complex (w_r, w_i) in 2.14 rectangular-complex format where inputs always have a magnitude of 1.0
- `X` output
  16-bit fixed-point signed 2's complement complex (x_r, x_i)

Design and write verilog for the block.

- With outputs scaled with maximum precision but also so they never overflow, underflow, or saturate.
- Appropriately pipelined so corresponding inputs enter at the same time.
- Use +, –, and * for arithmetic operations.

The test procedure is as follows:

1. Generate test cases in your verilog testbench:
   1) A minimum of 20 hand-picked extreme case inputs (e.g., max pos and max neg inputs)
   2) A minimum of 1000 random inputs using $random (which returns a 32-bit number each time it is called). Use $random(seed) once at the beginning of your test to set the random number generator's seed to some arbitrary value so tests can be repeated for debugging.
   3) W inputs should include the first 6 multiples of –45°, i.e., –0°, –45°, –90°, –135°, –180°, –225°. Or in matlab, simply `exp(-i*2*pi/8 * k)` where k varies from zero to 5.

2. Output both the a) inputs and b) verilog output to a plain-text matlab-readable *.m file. For example, a file such as:
   ```
   a_r(1) = -643; a_i(1) = 0; ... % matlab can not have index = 0
   a_r(2) = 123; a_i(2) = -6; ...
   a_r(3) = 000; a_i(3) = -243; ...
   ```
   where values can be printed out and then re-scaled in matlab however it is most convenient.

   Use "signed" reg's only for the printf statement. Suggestion: print integers in verilog.

3. Compare a) verilog output and b) matlab calculation of the butterfly equations using difff.m in matlab. Do not scale the matlab equations from how they are written above, but you may scale your verilog output by any power-of-2—which is the same as selecting the location of the decimal point.

Submit the following.

a) [30 pts] Detailed pipelined block diagram with all functional details.

b) [60 pts] Accuracy points for smallest error compared to matlab: 60pts: within 1 bit, 50pts: within 2 bits, 30pts: within 3 bits
Write the Energy_diff/Energy_data0 value in dB in your report and also submit the four plots and printout produced by difff.m.

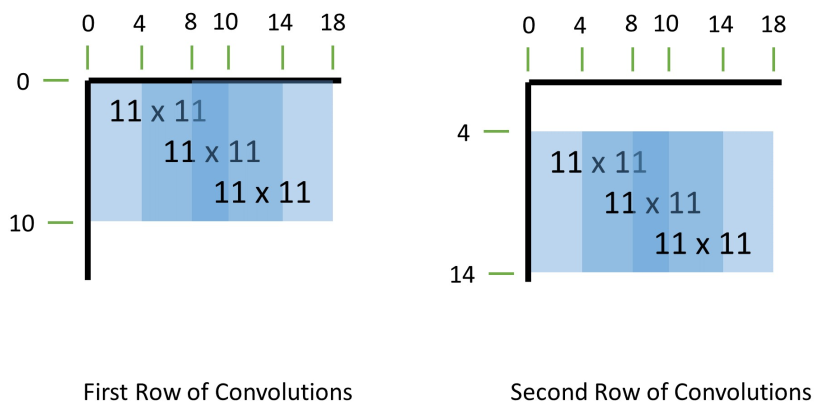
c) [60 pts] Synthesize your design at the following 3 cycle time values and report the 1) **achieved** cycle time (clock frequency) and 2) area for each:

1. a very long cycle time, e.g., 1 ms = 1 KHz, to find the minimum area;
2. a very short cycle time, e.g., 0.1 ns = 10 GHz, to find the minimum cycle time;
3. the cycle time **achieved** in the synthesis run for case (2) multiplied times 1.5

No points are possible for (b) or (c) unless the design is fully functional and without synthesis errors or serious warnings. See the Synthesis handout for details on the *achievable cycle time* and reading synthesis timing reports.

2. [250 pts] The Alexnet convolutional neural net is widely credited with the dramatic rise in popularity of neural nets. Read the 2012 Alexnet paper paying particular attention to Sections 1, 2, and 3.5. This project consists of building and synthesizing custom hardware for the first convolutional layer of Alexnet using a reduced image size. The primary specifications for this simplified project are as follows:

- input image size of 23 pixels x 23 pixels x 3 colors (R,G,B)
- filter size of 11 pixels x 11 pixels x 3 colors
- 16 3-dimensional 11 x 11 x 3 convolutions total with a stride of four pixels for each convolution. The first 3 convolutions of the first two rows are shown in the figure below, without showing the third dimension of RGB color.
- both the input image and the filter coefficients are 8-bit unsigned integers.



The testing environment is built as follows:

- the testbench performs the following steps in order one after another: 1) generates random data for the pixel memory and inputs one pixel (8+8+8=24 bits) at a time into the processor, 2) generates random data for the filter one pixel coefficient (8+8+8=24 bits) at a time into the processor, and 3) starts the processor calculating the 16 3D convolutions using control circuits entirely within the processor. See the "verilog: example code" web page for example code. Use random seed "123".
- all pixel data, filter data, and outputs are printed to a *.m file for analysis in matlab.

Other requirements:

- the 16 convolutions should be calculated in approximately 176 clock cycles.
- use only single-ported memories
- use "*" for multipliers, and carry-save adders plus one "+" CPA for the partial-convolution calculation.
- Because synthesis times will be too long (approx 30 minutes) if all memories are synthesized, place all memory modules outside the top-level processor, registering values immediately before leaving and after entering the main module.

Submit the following.

a) [25 pts] A bulleted list and a few sentences describing your design including such features as: the number and size of memories, number of multipliers and adders, exact number of cycles to complete the 16 convolutions, number of pipeline stages, in what order are the pieces and in what order are the 16 convolutions calculated, etc.

b) [25 pts] Detailed pipelined block diagram with all functional details.

c) [75 pts] Printed results for the 16 convolutions.

d) [50 pts] Matlab copy and pasted output showing whether your design matches the matlab model or not. The matlab model is provided.

e) [75 pts] Synthesize your design at the following three cycle time values and report the 1) *achieved* cycle time (and corresponding clock frequency) and 2) area for each:

1. a very long cycle time, e.g., 1 ms = 1 KHz, to find the minimum area;
2. a very short cycle time, e.g., 0.1 ns = 10 GHz, to find the minimum cycle time;
3. a synthesis run with the cycle time set to the minimum cycle time result for the synthesis run for case (2) multiplied times 1.5

No points are possible for (c), (d), or (e) unless the design is fully functional and without synthesis errors or serious warnings.

Hint: See the Synthesis handout for details on the *achievable cycle time* and reading synthesis timing reports.

Hint: See the "matlab: tips for 281" web page for suggestions on addressing memories in matlab.

Updates:

```
2020/03/02  Posted
```