EEC 281 - Homework/Project #1

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 showing the **functionality** requirements have been met. 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, assuming the "Y/N" is written individually for each test.

```
  inA       inB    outExp  outMantissa     Correct?
-------- --------  ------ --------------   --------
10101100 00110101  110010 01100110100101      Y
00000101 10110101  101010 01010101010101      Y
01010100 11101010  010100 11010101100101      no
```

  - [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.

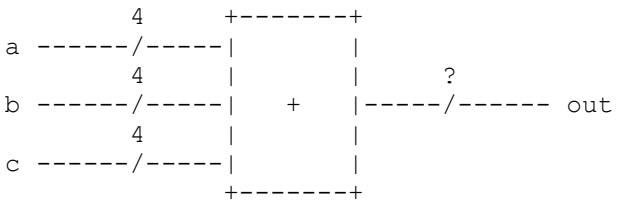
- **Clarity.** For full credit, your submission must be easily readable, understandable, and well commented.

Total: 190 points

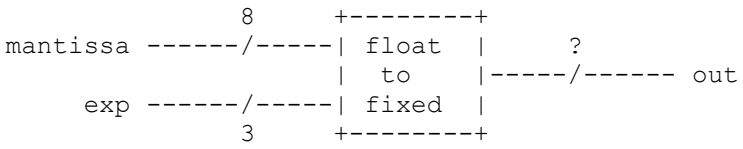
Before getting started, you should go through the verilog notes located under Course Readings on the course home page.

1. [35 pts] Design and write the verilog for a block that adds three 4-bit numbers into a 2's complement output that is sufficiently large to represent all inputs but with no extra bits. Use one stage of 3:2 carry-save adders and one carry-propagate adder (CPA) using a "+" in verilog. The three inputs are as follows:
   - `a` is in 2's complement 1.3 format
   - `b` is in unsigned 3.1 format
   - `c` is in sign-magnitude format where the magnitude portion is in 1.2 format

   a) [2 pts] How many bits does the output have and where is its decimal point?
   b) [4 pts] Show the adder's dot diagram.
   c) [3 pts] What is the output's minimum attainable negative value (most negative)?
   d) [3 pts] What is the output's minimum attainable positive value?
   e) [3 pts] What is the output's maximum attainable positive value?
   f) [20 pts] Test the circuit over at least 15 input values (including extreme cases). Turn in ***, opt. 1



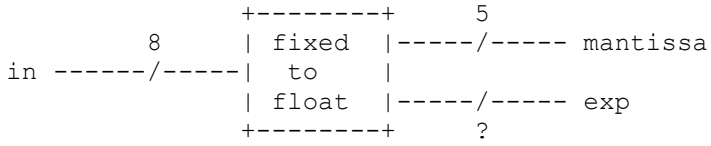
2. [35 pts] Design and write the verilog for a block that performs floating point to fixed point number conversion. The floating point input is always normalized and has an 8-bit signed, 2's complement mantissa in "6.2" format and a 3-bit unsigned integer exponent. The fixed point output has enough bits to fully represent the converted floating point number, but no more.
   a) [6 pts] How many bits does the fixed-point output have and where is its decimal point?
   b) [3 pts] What is the output's minimum attainable negative value?
   c) [3 pts] What is the output's minimum attainable positive value?
   d) [3 pts] What is the output's maximum attainable positive value?
   e) [20 pts] Test the circuit over at least 15 input values (including extreme cases). Turn in ***, opt. 1



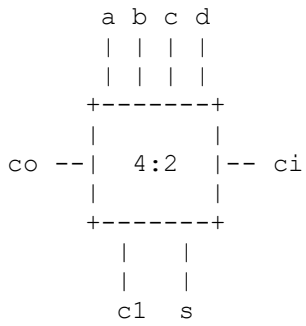
3. [35 pts] Design and write the verilog for a block that performs fixed-point to floating-point number conversion. The input fixed-point number has 9 bits and is in "7.2" 2's complement notation. The floating point output has a 5-bit "4.1" 2's complement mantissa and a 2's complement integer exponent.

   Normalize the output mantissa—the output must never be denormalized. Also keep the maximum possible number of bits from the input in the output mantissa. Note that for some input values, the output will not be able to represent all bits in the input and it will be necessary to reduce the number of bits (through rounding or truncation); use truncation.
   a) [6 pts] How many bits are required for the exponent?
   b) [3 pts] What is the output's minimum attainable negative value (most negative)?
   c) [3 pts] What is the output's minimum attainable positive value?

d) [3 pts] What is the output's maximum attainable positive value?
e) [20 pts] Test the circuit over at least 15 input values (including all extreme cases). Turn in ***, opt. 1



4. [15 pts] As mentioned in class, there are a number of ways to design a 4:2 adder.
a) Using the diagram for the 4:2 given below and the truth table below, fill out the truth table with the values that **must** be a certain value (0 or 1) for the circuit to operate correctly. Leave others blank. A few of these required values have been filled in.



inputs					outputs		
				c	c	c	
a	b	c	d	i	o	1	s
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0			1
0	0	0	1	1			0
0	0	1	0	0			1
0	0	1	0	1			
0	0	1	1	0			
0	0	1	1	1			
0	1	0	0	0			
0	1	0	0	1			
0	1	0	1	0			
0	1	0	1	1			
0	1	1	0	0			
0	1	1	0	1			
0	1	1	1	0			
0	1	1	1	1			
1	0	0	0	0			
1	0	0	0	1			
1	0	0	1	0			
1	0	0	1	1			
1	0	1	0	0			
1	0	1	0	1			
1	0	1	1	0			
1	0	1	1	1			
1	1	0	0	0			
1	1	0	0	1			
1	1	0	1	0			
1	1	0	1	1			
1	1	1	0	0			

```
1 1 1 0 1 |
1 1 1 1 0 |
1 1 1 1 1 | 1 1 1
```

5. [10 pts] Write the verilog for a Full Adder module using xor, and, or, inv operators. Also write the verilog for a 4:2 adder module using two full adder cells. Turn in *** option 1.

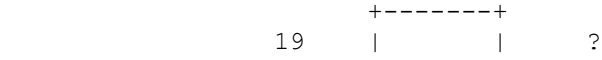
6. [25 pts] Six-input adder

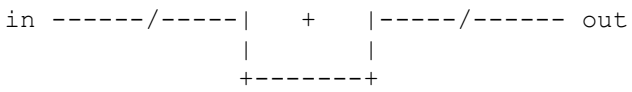
   a) [10 pts] Draw a dot diagram and write the verilog for a fast adder with six 4-bit signed 2's complement inputs and a 6-bit 2's complement output. Compresses the inputs in carry-save form using your 4:2 and 3:2 adder modules, and add the final "carry" and "save" words using a "+" operator in verilog.

   b) [15 pts] Write a testbench module which instantiates the six-input adder module and test the circuit over the input values shown: Turn in ***

```
=   0 +   0 +   0 +   0 +  0 +  0
=   1 +   0 +   0 +   0 +  0 +  0
=   0 +   1 +   0 +   0 +  0 +  0
=   0 +   0 +   1 +   0 +  0 +  0
=   0 +   0 +   0 +   1 +  0 +  0
=   0 +   0 +   0 +   0 +  1 +  0
=   0 +   0 +   0 +   0 +  0 +  1
=  -1 +   0 +   0 +   0 +  0 +  0
=   0 +  -1 +   0 +   0 +  0 +  0
=   0 +   0 +  -1 +   0 +  0 +  0
=   0 +   0 +   0 +  -1 +  0 +  0
=   0 +   0 +   0 +   0 + -1 +  0
=   0 +   0 +   0 +   0 +  0 + -1
=   7 +   0 +   0 +   0 +  0 +  0
=   0 +   7 +   0 +   0 +  0 +  0
=   0 +   0 +   7 +   0 +  0 +  0
=   0 +   0 +   0 +   7 +  0 +  0
=   0 +   0 +   0 +   0 +  7 +  0
=   0 +   0 +   0 +   0 +  0 +  7
=  -8 +   0 +   0 +   0 +  0 +  0
=   0 +  -8 +   0 +   0 +  0 +  0
=   0 +   0 +  -8 +   0 +  0 +  0
=   0 +   0 +   0 +  -8 +  0 +  0
=   0 +   0 +   0 +   0 + -8 +  0
=   0 +   0 +   0 +   0 +  0 + -8
=   1 +   1 +   1 +   1 +  1 +  1
=  -1 +  -1 +  -1 +  -1 + -1 + -1
=   1 +   2 +   3 +   4 +  5 +  6
=   7 +   4 +   5 +   5 +  5 +  5
=  -7 +  -5 +  -5 +  -5 + -5 + -5
```

7. [35 pts] Design a block which adds 19 single-bit numbers, where each input is not a standard binary number—but instead each wire represents –1 or +1. A zero input is –1 and a one input is +1. The output of the adder is a 2's complement number sufficiently wide to represent all input combinations. This is an important structure in a CDMA transmitter. You may use 4:2, 3:2, and half adders (implemented as submodules however you wish; e.g., wire, reg, table) to add the inputs efficiently. Use only one carry-propagate adder, which you can implement with a "+" or "–" in verilog.





a) [10 pts] Draw a "dot diagram" for an efficient adder.

b) [5 pts] Total up and state how much hardware (in area) your design requires in units of 3:2 adders assuming a 4:2 adder costs the same as two 3:2 adders and a half adder costs 0.5 3:2 adders.

c) [5 pts] Estimate the delay of your complete adder by finding the longest path through your adder and clearly showing that path on your dot diagram using these estimates: 4:2 delay of 150 ps, 3:2 delay of 100 ps, and HA delay of 75 ps. (Delays do not scale with logic complexity to very roughly account for wire delays.)

d) [15 pts] Write your design in verilog, test it, and turn in ***. You may find these as helpful *starting* points: prob1.v, prob1_ref.v, and prob1.vt.

EEC 281 | B. Baas | ECE Dept. | UC Davis

2020/01/17 Posted