EEC 281 - Homework/Project #2

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: 165 points

For this homework/project, you may use the `"always @(*)"` verilog construct but keep an eye out for any situations where Design Compiler is not compatible with it.

1. [25 pts] The purpose of this problem is to familiarize you with the synthesis process and to give you a rough feeling for the size of a few simple circuits in our standard cell library's technology. Copy the files from the DC tutorial (see link on main EEC281 page) to get started. Synthesize the following blocks and report their total cell area. Do not include registers (flip-flops) in these blocks. Also, do not declare any wires or registers as "signed", but assume words are all 2's complement signed unless stated otherwise. No need to simulate, but your verilog must compile correctly (run "make check"). Also, for this problem, do not worry if designs do not meet timing (negative slack time).

   Turn in: 1) source verilog, 2) totals in a single table so it can be used as a note sheet in the future. Do not submit any output synthesis reports.

   Blocks

   a) [2 pts] bitwise AND of two 10-bit numbers (10-bit output)

   b) [2 pts] A 3:2 adder using verilog "`&`" "`|`", "`^`", "`~`".
   Draw your circuit and the circuit output by DC.

   c) [2 pts] A 3:2 adder using verilog "+".

   d) [3 pts] An 8-bit adder (9-bit output). Use "+" in verilog.

   e) [5 pts] An adder which adds 19 6-bit numbers using verilog "+" (i.e., something like, assign out = in0 + in1 + in2 + ...) and produces a 6-bit sum.

   f) [5 pts] Your 19-input adder from hwk/proj 1, Problem 7. If your adder is not functional, improve it so it is at least synthesizable, synthesize it anyway, and write a note on your submission that it is not functional.

   g) [3 pts] 8-bit x 8-bit unsigned multiplier (16-bit output). Use "*" in verilog.

   h) [3 pts] 16-bit x 16-bit unsigned multiplier (32-bit output). Use "*" in verilog.

2. [20 pts] Build a ripple-carry adder with 16-bit inputs and 16-bit output using full adders from part 1(c). Register all inputs and outputs (to make synthesis timing accurate).

   a) [10 pts] Write design in verilog, test with at least 15 test cases. Verify using method ***(3).

   b) [10 pts] Synthesize the design with a high clock frequency to find the maximum clock rate. State the maximum clock rate and corresponding area. Submit *.area and *.tim (longest path only) reports only.

3. [25+10 pts] Repeat Problem 2 with a carry-select adder composed of two 8-bit sections.

4. [25+10+10 pts] Repeat Problem 2 with a carry-select adder composed of three sections whose widths are chosen to minimize delay.

   c) [10 pts] Justify the partitioning you chose.

5. [25+10 pts] Repeat Problem 2 but pipeline the ripple-carry adder into 8 pipeline stages.

6. [5 pts] Write a single table with 1) max clock frequency and 2) area for problems 2–5.

Updates:

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