ECE 351- Spring 2021
Homework #1

Submit your deliverables to your Homework #1 dropbox by 10:00 PM on Monday, 26-Apr-2021. We will only grade the submission with the latest date stamp. NO LATE ASSIGNMENTS WILL BE ACCEPTED AFTER NOON ON THURSDAY, 29-APR-2021 BECAUSE I'D LIKE TO REVIEW THE SOLUTION IN CLASS BEFORE THE EXAM ON 06-MAY.

True/False, multiple choice and short answer questions should be answered in the `ece351sp1_hw1a.txt` file provided in the release. This will be the same procedure we will use for the midterm and final exams.

Source code for Question 2 (the Carry Lookahead adder) should be structured and commented with meaningful variables. Include a header at the top of each file listing the author and a description. You may use the following template:

```
////////////////////////////////////////////////////////////
// <filename>.sv - <one line description>
//
// Author:  <your name> (<your email address>)
// Date:    <date you created the code>
//
// Description:
// ------------
// <text description of what function the module performs>
////////////////////////////////////////////////////////////
```

Question 1 – True/False, multiple choice, short answers (60 pts)

Submit your answers in the file `ece351sp1_hw1a.txt`.

Question 2 – SystemVerilog coding and simulation (40 pts)

The in-class exercise was based on a ripple-carry adder which is among the simplest of adder architectures, but also one of the slowest. The Carry Lookahead adder that we are going to implement in this question is faster, and more complex, than the ripple-carry-adder. Per Wikipedia (https://en.wikipedia.org/wiki/Carry-lookahead_adder):

> "A **carry-lookahead adder** (CLA) or **fast adder** is a type of electronics adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple-carry adder (RCA), for which the carry bit is calculated alongside the sum bit, and each stage must wait until the previous carry bit has been calculated to begin calculating its own sum bit and carry bit. The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder… Carry-lookahead depends on two things:

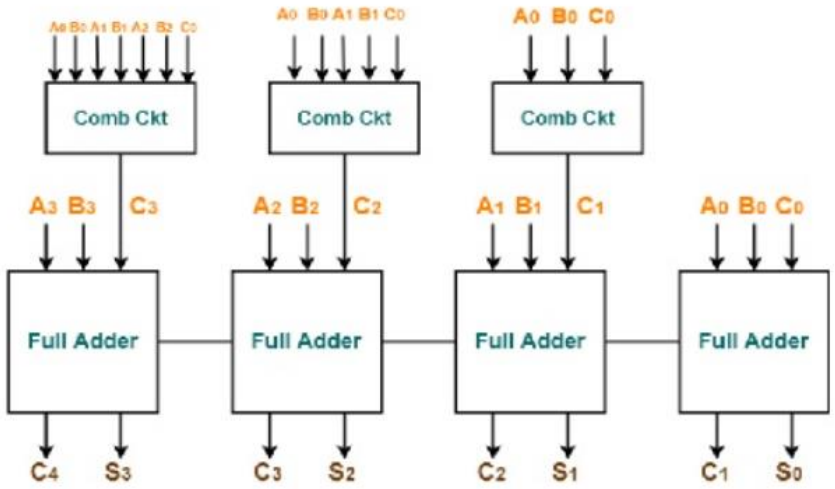
1. Calculating for each digit position whether that position is going to propagate a carry if one comes in from the right.
2. Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of four digits are chosen the sequence of events goes something like this:

1. All 1-bit adders calculate their results. Simultaneously, the lookahead units perform their calculations.
2. If a carry arises in a particular group, that carry will emerge at the left-hand end of the group within at most five gate delays and start propagating through the group to its left.
3. If that carry is going to propagate all the way through the next group, the lookahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group*, the lookahead unit is immediately (within one gate delay) able to tell the *next* group to the left that it is going to receive a carry – and, at the same time, to tell the next lookahead unit to the left that a carry is on its way.

The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move four times as fast, leaping from one lookahead-carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group."

A block diagram and logic equations for a 4-bit carry lookahead adder are shown below (Source: https://www.elprocus.com/carry-look-ahead-adder/



The full adders are just that – a full adder much like the one we built for the ripple-carry-adder, but we will implement this full adder in RTL, not as a gate level model, but as an RTL model of combinational logic using the following logic equation (where $_x$ is the bit number):

$$S_x = A_x \text{ ^ } B_x \text{ ^ } CIN_x$$

The carry propagation/generation logic can be implemented with the following logic equations (where $_x$ is the bit number):

General case:

$P_x = A_x \text{ ^ } B_x$;
$G_x = A_x \text{ \& } B_x$
$C_{x+1} = C_x \text{ \& } P_x \text{ | } G_x$

Carry logic equations (the *Comb Ckt* in the block diagram:

C_0 = carry in to the adder
$C_1 = C_0 \text{ \& } P_0 \text{ | } G_0$
$C_2 = (C_1 \text{ \& } P_0 \text{ | } G_0) \text{ \& } P_1 \text{ | } G_1$
$C_3 = (C_1 \text{ \& } P_1 \text{ | } G_1) \text{ \& } P_2 \text{ | } G_2$
$C_4 = (C_0 \text{ \& } P_0 \text{ \& } P_1 \text{ \& } P_2 \text{ \& } P_3) \text{ | } (P_3 \text{ \& } P_2 \text{ \& } P_1 \text{ \& } G_0) + (P_3 \text{ \& } P_2 \text{ \& } G_1) \text{ | } (G_2 \text{ \& } P_3) \text{ | } G_3$

Task List for Question 2

Step 1: Create a combinational RTL model for a 4-bit expandable carry lookahead adder (CLA). We will simplify the design by removing any propagation times from the circuit.

A combinational RTL model uses only continuous assignment (`assign`) statements or `always_comb` procedural block with or without gate delays. You do not need to create or use gate-level modules for the adders. You may want to create a single bit adder module and instantiate it multiple times.

Step 2: Create 4-bit CLA module. Your 4-bit CLA module should have the following signature:

```
module CLA4Bit(
      input logic [3:0] ain, bin,   // A and B inputs to the adder
      input logic       cin,        // carry-in input to the adder
      output logic [3:0] sum,       // 4-bit sum
      output logic       cout       // carry out
);
```

Include your SystemVerilog source code files with your deliverables.

Step 3: Simulate your 4-bit CLA module with the testbench we provided. Save and include a transcript of your successful simulation results in your deliverables.

Step 4: Create an 8-bit adder using two instantiations of the 4-bit CLA module you created and verified. Your 8-bit adder should have the following signature:

```
module CLA8Bit(
      input [7:0] ain, bin,   // A and B inputs to the adder
      input       cin,        // carry-in input to the adder
      output logic [7:0] sum, // 8-bit sum
      output logic       cout // carry out
);
```

Include your SystemVerilog source code files with your deliverables.

Step 5: Simulate your 8-bit CLA Adder with the testbench we provided. Save and include a transcript of

your successful simulation results in your deliverables.

<finis>