COSE221: Computer Architecture
Design Lab #4

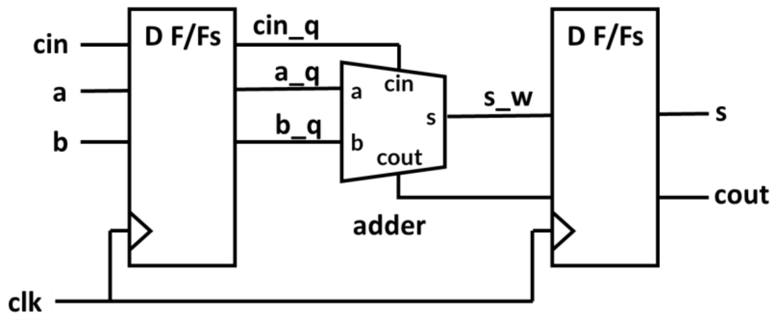
Due: June 12, 2023 (Monday) 11:59pm on Blackboard

Total score: 40 pts

In this lab, you will practice *structural model design* using SystemVerilog. You will also explore the *timing characteristics of sequential logic*. For structural models of HDLs, we can build more complex modules by instantiating smaller modules inside of the higher-level module. In this design, we will use basic gate modules for building simple building blocks (e.g. adders), then the simple building blocks will be instantiated in more complex modules (remember the three -y's). Note that the basic gate modules include timing information (i.e. propagation delay), thus the timing of the higher-level modules can be computed by the timing specification of the basic gate modules when the gate modules are instantiated. In order to get correct results from sequential logic, we need to set appropriate clock frequency considering the delay of combinational logic between series of registers.

1. Target design

You are requested to a 16-bit adder as shown in the figure below. The input signals (i.e. a, b, and cin) are sampled by registers, thus the input signals to adders are synchronized by clock. The output signals (i.e. s and cout) are also synchronized by flip-flops. Hence, the delay of the adder decides the maximum clock frequency of this system.



2. Clock signal

We kindly provide the testbench modules that can be used for verifying your design. The testbench modules will check the correctness of output signals (s and cout) automatically by using the native add operator in SystemVerilog. The testbench module provides a clock signal to your design. The default clock frequency is defined as follows.

```
`timescale 1ns/1ps
`define CLK_T 0.200
```

Note that the default clock period is set as 200 ps since the time unit of the testbench module is 1 ns. If the clock period is smaller than the longest delay of the combinational logic between registers, your system will generate incorrect outputs. Thus, we recommend you set longer clock periods when you verify the functional correctness of your design.

3. 16-bit ripple carry adder design

You are requested to design 16-bit ripple carry adder (RCA) in structural models of SystemVerilog. We provide `gates.sv` file that includes the functional model and timing information of basic gates. You must use only the basic logic gates in `gate.sv` file for designing required modules (i.e. combinational logic parts) for adders. The following code exhibits the example of xor2 gate.

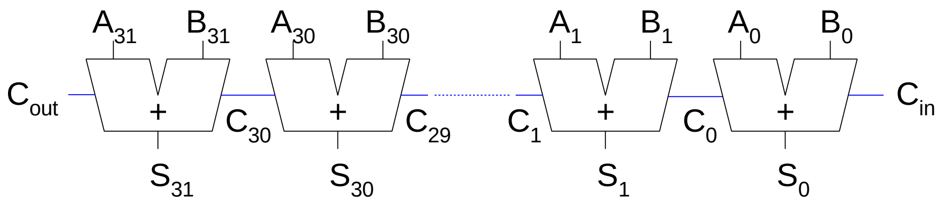
```
`timescale 1ns/1ps

module xor2 (
	input  logic  a, b,
	output logic  y
);
	assign #(0.010) y = a ^ b;	// delay: 10 ps
endmodule
```

This module defines the functional behavior and the timing information of xor2 gate. The delay of the xor2 represents the propagation delay of xor2 gate.

We provide the design file (i.e. `rca16.sv`) that includes skeleton codes for 16-bit RCA design. In order to design multi-bit RCA, you first need to complete the structural model of a full adder module (`fadd`). Complete the `fadd` module by using only the basic gates in gate.sv.

Then, you can design `rca16` module by instantiating `fadd` modules. The registers for input signals are already completed in the `rca16` module. Note that you need to instantiate the `fadd` modules for designing combinational logic parts of the 16-bit RCA. The following figure shows the example design of 32-bit RCA. You need to connect input/output ports of the multiple `fadd` modules to implement a multi-bit adder.



Once you complete your 16-bit RCA design, you can compile and simulate your design as follows.

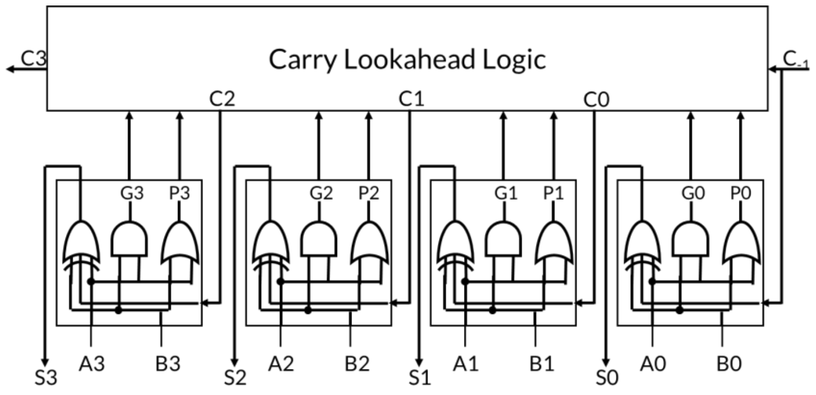
```
$ iverilog -g2005-sv -o rca16 tb_rca16.sv rca16.sv gates.sv
$ vvp -v rca16
```

Since your 16-bit RCA is implemented using the basic gates that include delay information, you can observe timing delay of combinational logic signals in the generated waveforms. Furthermore, if the period of the clock signal is longer than the critical delay (please figure out) of the 16-bit RCA, your design will generate incorrect results even though your design is functionally correct. In this case, you need to adjust the clock period in the testbench module.

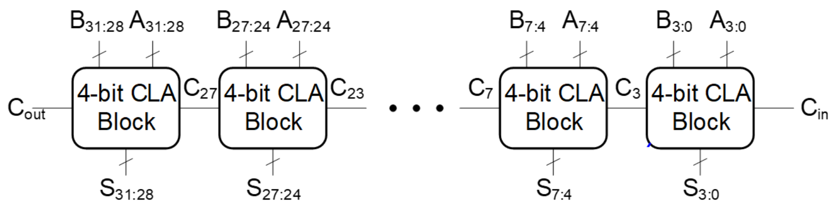
4. 16-bit carry lookahead adder design

In the class, we learned a carry lookahead adder (CLA) can reduce the critical delay of a multi-bit adder. In this lab, you are requested to design 16-bit CLA in structural models of SystemVerilog. You also need to use basic logic gates in `gates.sv` to implement combinational logic parts of 16-bit CLA. You can design 16-bit CLA hierarchically. Namely, you first need to complete the design of `fadd_cla` module which generates propagate (`p`) and generate (`g`) signals as well as sum (`s`). Then you can design 4-bit carry lookahead logic (`cll4`) by instantiating basic logic gates. Note that carry lookahead logic generates carry-outs for `fadd_cla` modules and next carry lookahead logic. You should implement the carry lookahead logic by instantiating the basic logic gates in `gates.sv`.

Then, you can design the structural model of 4-bit CLA (`cla4`) by instantiating `fadd_cla` and `cll4` modules as shown in the following figure.



Finally, you can implement a multi-bit CLA by combining 4-bit CLAs (`cla4`) as follows. The registers for input signals are already completed in the `cla16` module. You can instantiate `cla4` modules to implement 16-bit CLA.



Once you complete your 16-bit CLA design, you can compile and simulate your design as follows.

```
$ iverilog -g2005-sv -o cla16 tb_cla16.sv cla16.sv gates.sv
$ vvp -v cla16
```

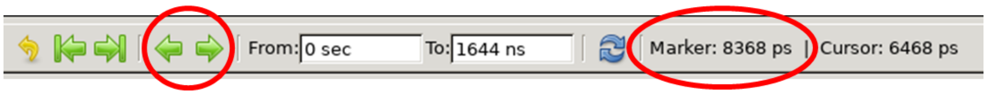
You can also observe timing delay of combinational logic signals in 16-bit CLA. Note that the delay of the critical path in the 16-bit CLA is shorter than the longest delay of the 16-bit RCA.

3. What to do

(a) Complete the provided design files (i.e. `rca16.sv` and `cla16.sv`). You can complete "`/* FILL THIS */`"parts in the provided files. You can verify your design using the provided testbench modules that can perform *self-checking* for DUT. You need to increase the clock period of `tb_rca16.sv` in order to check the functional correctness of the 16-bit RCA.

(b) You can observe the delay of the critical path in adders when carries propagate across all digits in the adders. The provided testbench also includes this case. Namely you can observe the longest delay in the adders when `a=16'h0000` and `b=16'hFFFF` and `cin` is changed from `1'b0` to `1'b1`. Figure out which path is a critical path of each 16-bit adder design (i.e. `cin` → ??). Justify your answer.

(c) For each 16-bit adder design, figure out the delay of the longest path using GTKWave. You can see individual bit signals when you double-click a vector signal in the Signals window of GTKWave. Then you can move the marker line to the edge of the target signal by clicking the arrow icons as shown the figure below. You can also read the location (time) of a marker in GTKWave.



In order to figure out the delay of the critical path in the 16-bit adder design, you need to append the signals in DUT (not a testbench module) in GTKWave. Capture the waveforms in GTKWave to exhibit how you figure out the delay of the critical path in each 16-bit adder design. Include the captured images in your report document.

(d) For each 16-bit adder design, figure out the shortest clock period that can generate correct results. You can check if your 16-bit adder design generates correct results by changing clock periods in the testbench modules. Figure out the clock periods in 10 ps resolution (e.g. 230 ps not 234 ps). For this simulation setup we ignore the setup time of a D flip-flop and clock skew of clock signals.

(e) Compress your PDF file (a report that includes answers and captured images), source codes (`rca16.sv` and `cla16.sv`), the generated VCD files (`*.vcd`), and output files (`*.out`) in **one zip file**. You must name your zip file as "`FirstName_LastName.zip`". (e.g. `Gildong_Hong.zip` for Gildong Hong) If your submission file does not meet this rule, we will reduce 1 point from your score. ☹

<NOTE>

SystemVerilog syntax `always_comb`, `always_ff`, and `always_latch` is supported by Icarus Verilog v11.0. Unfortunately, the older versions of Icarus Verilog cannot understand `always_comb`, `always_ff`, and `always_latch`. You can install Icarus Verilog v11.0 from Ubuntu's default repository if you are using Ubuntu newer than version 22.04. If not, you can try one of the following methods.

Method 1 (recommended):

Change always statements in the source codes as follows.

```
always_comb → always @ (*)
always_ff → always
```

Method 2:

You can compile the newer version of Icarus Verilog from the source code. Please follow the instructions in https://iverilog.fandom.com/wiki/Installation_Guide#Compiling_on_Linux/Unix. Before installing the newer version of Icarus Verilog, you need to remove the installed Icarus Verilog.

```
$ sudo apt purge iverilog
$ sudo apt update
$ sudo apt install bison, flex, autoconf, gperf
$ mkdir -p workspace
$ cd workspace
$ git clone https://github.com/steveicarus/iverilog.git
$ cd iverilog
$ git checkout --track -b v11-branch origin/v11-branch
$ git pull
$ ./configure
$ make
$ sudo make install
```