University of the Philippines Los Baños
College of Arts and Sciences
Institute of Computer Science
CMSC 132 - Computer Architecture
First Semester AY 2022-2023

Laboratory Topic 04: Sequential Circuits

RNC Recario {rcrecario@up.edu.ph}

Learning Outcomes:

At the end of the laboratory session, the student is expected to:

- Define what a sequential circuit is
- Implement a sequential circuit

----------**Topic Proper**----------

What is a sequential circuit?

A **sequential circuit** is a type of circuit whose "current output depends on past inputs as well as current inputs" [1]. Additionally, in a sequential circuit, the "outputs of the sequential circuits depend on both the combination of present inputs and previous outputs", and that "the previous output is treated as the present state" [2].

There are two major types of sequential circuits: **asynchronous** and **synchronous**. Whether or not you have taken CMSC 130 at this point, we will simplify the details and point out that synchronous sequential circuits make use of a clock (CLK). While we did point out this difference, you should be responsible to recall (if you have taken CMSC 130) or study in advance (for those who are currently taking or have not taken CMSC 130 just yet) this topic.

At this point, everything you learned so far will be used on top of what you will still learn from this topic.

What is a clock/clock signal?

A clock typically represented as CLK or clk is a signal that produces a "HIGH" (1) or "LOW" (0) signal in a regular period, typically with the same amount of time. For example, if say a CLK, started with "LOW" (0) with 1 nanosecond (1 ns), then it will be followed by a "HIGH" with 1 ns and then the same pattern of those values (0 and 1) in an alternate fashion.

An example of a CLK in signal form is shown below:

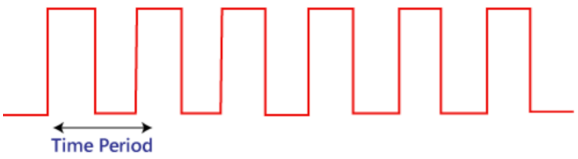


Figure 1. An example visualization of a CLK is signal form.

As you can see in figure 1, a time period is defined by the **rising edge** known as the positive edge and the **falling edge** or negative edge. The rising edge is the state where the signal moves from "LOW" to "HIGH" (0 to 1) while the falling edge does the opposite: from "HIGH" to "LOW" (1 to 0). The reason why we are pointing this out is because many synchronous sequential circuits make use of the positive edge as a "trigger" for their outputs.

How do we create a clock?
Depending on your approach and style, the clock can be created as a component file, a design file or included in the test bench itself. The third approach is the most common and the most logical to do. This is also the approach which will be shown on the latter part of this handout.

Independently, you write a clock as shown below:

simple_clock.vhdl

```
1   entity simple_clock is
2   end entity;
3
4   architecture rtl of simple_clock is
5     constant num_cycles : integer := 640; -- the number of cycles for the clock
6     signal clock: std_logic := '1';
7   begin
8           process
9                   begin
10                  for i in 1 to num_cycles loop --loop from 1 to the declared num_cycles
11                          clock <= not clock; --pulse from 1 to 0 or 0 to 1
12                          wait for 1 ns; --as stated, wait for 1ns
13                  end loop;
14                  wait;
15          end process;
16  end architecture;
```

The code simply shows a signal clock oscillating from '0' to '1' repeatedly. Do note that we cannot realistically implement a "never-ending" clock and so, we only show the behavior of the clock up until the 640th cycle (640 ns). This is illustrated by the declared constant num_cycles and the amount of wait needed per instruction as shown in line 12.

Interestingly, the signal clock is not implemented using a **bit** datatype but with **STD_LOGIC**. The data type std_logic is a more realistic data type to use in fact, even for the other signal inputs/outputs you will be using from this point on.

An **STD_LOGIC** data type allows the signal to assume values X, 0, 1 or Z. There are "other values that this data type can have, but the other values are not synthesizable – i.e. they can not be used in VHDL code that will be implemented on a CPLD or FPGA" [3].

These values have the following meanings:

```
X – unknown
0 – logic 0
1 – logic 1
Z – high impedance (open circuit) / tristate buffer
```

"When assigning a value to a STD_LOGIC data type, the value must be enclosed in single quotes: 'X', '0', '1' or 'Z' ".[3]

Note that in our simple_clock.vhdl, there is already a process and so, it will no longer need a test bench.
Simply run the code given the commands you have seen several times already:

```
ghdl -a simple_clock.vhdl
ghdl -e simple_clock
ghdl -r simple_clock --vcd=simple_clock.vcd &
```

The output waveform is shown below:

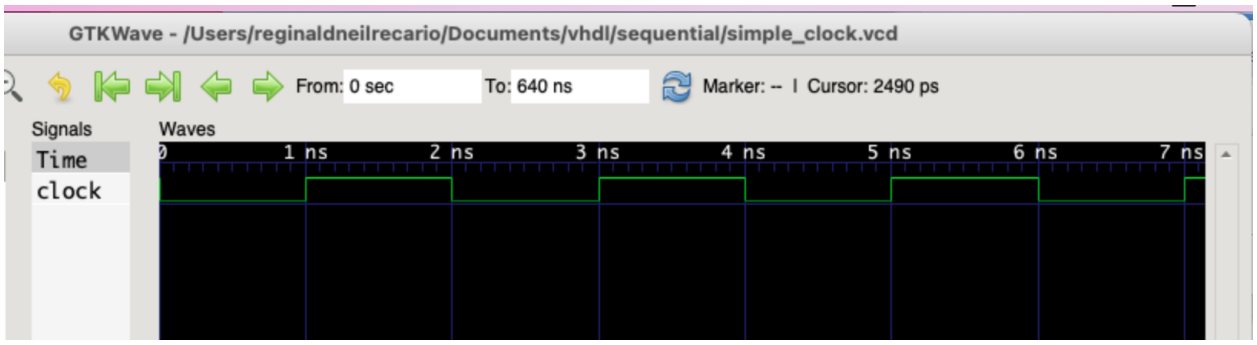


Figure 2. GTKWave view of the simple_clock vhdl file.

How do we implement a synchronous sequential circuit?
The implementation is the same but with some additional concepts such as the clock which we have discussed already. In addition, we need to know when the CLK signal changes-- to be more specific, when does the CLK have a positive edge. This is because many synchronous sequential circuits are positive-edge triggered.

We now introduce you to the function called **rising_edge()**. The function rising_edge() accepts a signal parameter of type std_logic/std_ulogic. It returns True if the signal is rising i.e., from "LOW" to "HIGH", else returns False.

A code snippet below is an example use of the **rising_edge()** function:

```
1  if rising_edge(clock) then
2      mysignal <= '1';
3  end if;
```

As you can see, the code wants to determine if the current state of the clock transitions from LOW to HIGH. If that is the case, then mysignal's value is replaced with '1' (mysignal is assumed to be a bit signal too).

Another thing that we need to be aware of specifically when implementing the design already is the use of the **wait statement**. In previous topics, we have seen design files using only the `wait;` as is. However, we can introduce variations by invoking specific time periods of wait. An example is shown below:

```
wait for 1 ns;
```

The code above is something you might have remembered from your test bench. It follows the syntax:

```
wait for <amount of time> <unit of time>;
```

where <*amount of time*> is a number and <*unit of time*> is the unit of time in shorthand form.

The following are valid time units in VHDL:

Shorthand form	**Unit of time**
fs	femtoseconds
ps	picoseconds
ns	nanoseconds
us	microseconds
ms	milliseconds
sec	seconds
min	minutes
hr	hours

The statement `wait for 1 ns;` means exactly how it is written: to wait or delay for 1 nanosecond. The knowledge of using the delay in various amounts of time can be useful later on when performing some tests on the behavior of the circuit.

Truth table vs characteristics table vs excitation table: what's what?

A **truth table** is a table that "shows all possible combinations of inputs and, for each combination, the output that the circuit will produce". [4]

A **characteristics table** (at times called characteristic table), typically for a flip-flop, is a table that shows the next state (i.e., Qn+1) based on the input and the current state.

An **excitation table** is a table that "has the minimum inputs, which will excite or trigger the flip flop to go from its present state to the next state" and "is derived from the truth table". [5]

D Flip-Flop Example

Let us now use the D flip-flop as an example.

A **D flip-flop,** where D stands for delay, is a "digital electronic circuit used to delay the change of state of its output signal (Q) until the next rising edge of a clock timing input signal occurs". [6] This means that a D flip-flop is positive edge-triggered.

The D flip-flop has two input values: the CLK and D and two output values: Q and Q' .

The following is the block diagram for the D flip-flop.

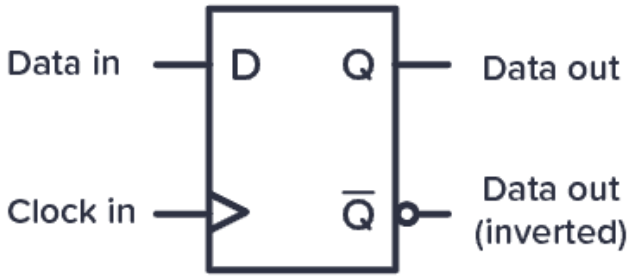


Figure 3. The block diagram of D Flip-flop.

In the interest of time, we will no longer cover the details of how a D flip-flop is implemented.

Given the description, we can create this truth table for the D flip-flop:

CLK	D	Qn+1
0	x	Qn
1	0	0
1	1	1

The following is the characteristics table from CMSC 130:

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

Figure 4. The characteristics table of D Flip-flop [1]. Figure from CMSC 130 slides.

The excitation table for a D flip-flop is shown below:

Qn	Qn+1	D
0	0	0
0	1	1
1	0	0
1	1	1

Here is an example how a D flip-flop behaves (for simplicity, Q' is not included in the diagram):

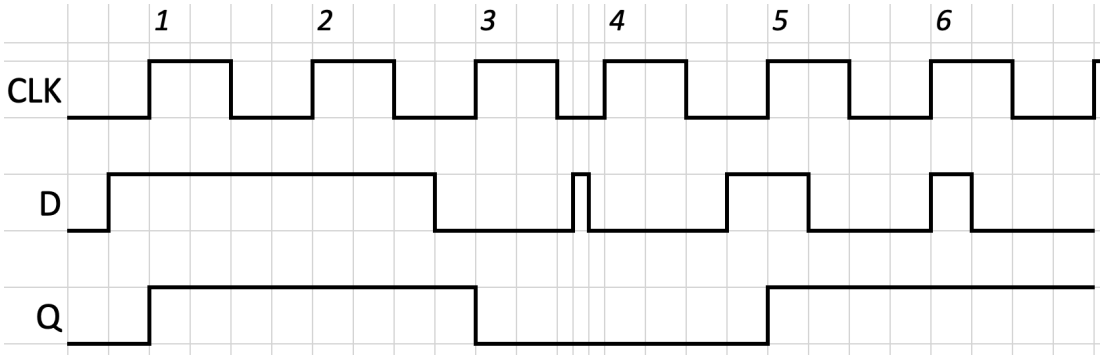


Figure 5. An example timing diagram of how a D flip-flop behaves.

In the first positive edge (time 1), when D is already "HIGH" (1), the Q value is set to "HIGH" (1). Notice that when the clock has a negative edge, even when D is "HIGH" the Q is still set to "HIGH". Something to take note of is the one before time 3 when CLK is LOW. At that time, D transitions from HIGH TO LOW and yet Q is still HIGH until time 3. This is to emphasize that again, Q and Q' only change during the positive/rising edge, hence the change at time 3 (HIGH to LOW). From time unit 3 to before the start of time 4, we see that D briefly went to HIGH. However, that is not reflected in Q given that the HIGH state occurred when the CLK is LOW. D went on to HIGH state before time 5 which is captured and reflected to D. D went to D in the middle of the HIGH for time 5 then set to LOW and then again went to HIGH at the half of HIGH state of the CLK. Notice that in those periods covered, Q remains high since when CLK is LOW, it only retains the value in the previous state which is already in HIGH. In the next positive edge, it noted that D is still HIGH, hence retaining only the HIGH state.

After all the discussion, here is how we implement D flip-flop:

DFlipFlop.vhdl

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   --example D Flip-flop
4   entity DFlipFlop is
5      port (clk,Din : in std_logic;
6            Q: out std_logic;
7            Qnot : out std_logic);
8    end DFlipFlop;
9   architecture behav of DFlipFlop is
10     begin
11        process (clk,Din)
12           begin
13           if(rising_edge(clk)) then
14              Q <= Din;
15              Qnot <= (not Din);
16           end if;
17        end process;
18  end behav;
```

Nothing has changed with how we write the code with the exception of the inclusion of the library from lines 1 to 2. This allows us to use std_logic. The inputs clk and Din are representative of the inputs CLK and D whereas the outputs Q and Qnot are the representatives of Q and Q' output signals.

The architecture body is standard with special attention on lines 13 to 16. These lines capture the essence of how a D flip-flop works noting that it only "updates" the outputs Q and Qnot when there is a rising edge.

While the design file is pretty straightforward, creating the testbench file would entail a certain level of creativity. Using a pattern would no longer be sufficient. An example of how the author created his test bench is shown below. Do note that the test bench can be implemented in a number of ways. The important thing here for both the test bench and the design file/s is that you really understand how they are intended to work based on how they were implemented/written.

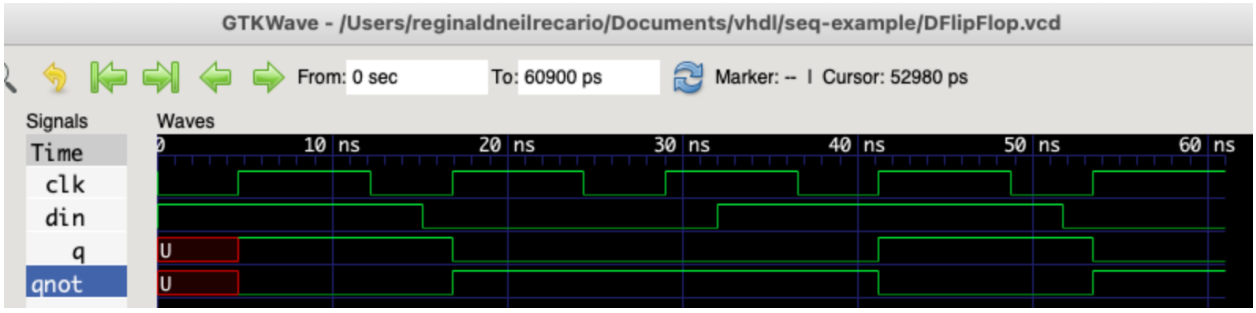
DFlipFlop_tb.vhdl

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity DFlipFlop_tb is
5   end entity;
6
7   architecture rtl of DFlipFlop_tb is
8           signal Din : std_logic:='1';
9           signal Q : std_logic;
10          signal Qnot : std_logic;
11          signal clk: std_logic := '1';
12          component DFlipFlop is
13                  port(clk,Din : in std_logic; Q: out std_logic; Qnot : out std_logic);
14          end component;
15
16  begin
17          dff: DFlipFlop port map(clk=>clk, Din=>Din, Q=>Q, Qnot=>Qnot);
18          process
19          begin
20                  for i in 1 to 10 loop
21                          clk <= not clk;
22                          wait for 3 ns;
23                          if(i mod 3 =0) then
24                                  Din <= not Din;
25                          end if;
26                          wait for 1.65 ns;
27                          if(i mod 2 = 0) then
28                                  wait for 2.88 ns;
29                          end if;
30                  end loop;
31                  wait;
32          end process;
33  end architecture;
```

In the author's implementation, there is a loop from 1 to 10 allowing the clk to oscillate ten times together with the other signals. Inside the loop body, there are several if statements with wait statements inside them. This is done in order to ensure a certain degree of randomness. With this, the signals will not always align in terms of the rising and falling edges.

Do take note that when implementing your own test bench, you should not copy the above code down to the letter.

Using GTKWave, the test bench and the design file will generate this view:



References:

[1] CMSC 130 Slides 8.1 Flip-flops. RDCM.
[2] https://www.javatpoint.com/sequential-circuits-in-digital-electronics
[3] https://startingelectronics.org/software/VHDL-CPLD-course/tut13-VHDL-data-types-and-operators
[4] https://isaaccomputerscience.org/concepts/sys_bool_construct_truth_table
[5] https://www.electrically4u.com/what-is-the-excitation-table/
[6] https://www.maximintegrated.com/en/glossary/definitions.mvp/term/D%20Flip-Flop/gpk/1208