Laboratory Exercise 4

Clocks and Counters

Revision of September 30, 2025

In this exercise you will be introduced to shift registers and counters. You will also learn how to control the sequencing of operations when the actual clock rate is much faster than the rate the operations are occurring. Each part of this lab uses flip flops with an **active-high, synchronous** reset.

WARNING: Parts II and III of this lab are complex designs. You are **strongly encouraged** to read the lab document carefully, more than once to make sure you understand what you must do. Then, **design and test your code as you go**. Writing a lot of code and testing it all at the end can lead to you wasting a lot of time.

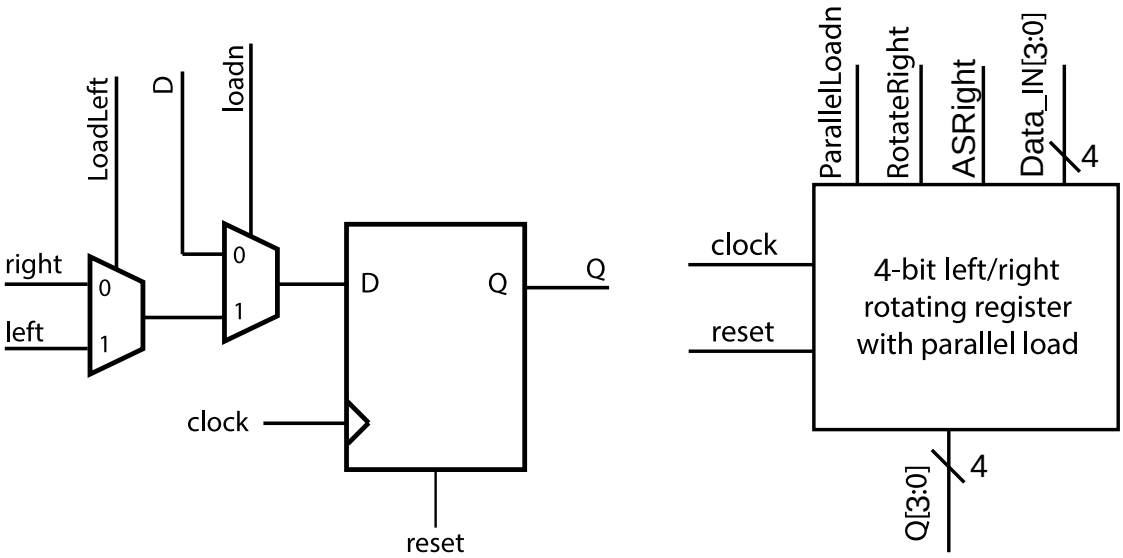
This is also a good lab to try to run on the FPGA. You should try to complete as many parts as you can before your lab session and run them on the FPGA during your lab session. If you cannot do this, try to run your code on the FPGAs in the drop-in lab (BA3135) which you can access 24/7 using your T-card.

1 Part I

A basic element in designing sequential logic is the `shift register` (textbook Section 5.4.2); we will see shift registers in lecture as part of the module on Finite State Machines. When bits are shifted in a register, it means that the bits are copied to the next flip flop on the left or the right. For example, to shift the bits left, each flip flop loads the value of the flip flop to its right when the clock edge occurs. In a normal shift register, the right-most register is loaded with an input value or with a fixed value such as 0.

A more advanced version of the shift register is the `rotating register`, which feeds the value shifted out back to the input of the shift register, thus 'rotating' the value. The only way to change the value in a rotating register is to use the 'parallel load' input to set all the D-flip flops at the same time. In this part of the lab, you must design a 4-bit rotating register with parallel load. Each bit of this rotating register consists of a positive edge-triggered D flip flop and several multiplexers as shown in Figure 1a. The mux closest to the flip flop supports parallel load using the `loadn` input. The second mux controls the direction of rotation. The D flip flops should have an **active-high synchronous** reset.

The top-level circuit is shown in Figure 1b and consists of 4 instances of the circuit shown in Figure 1a. Let's take a look at the function of each of the inputs shown in Figure 1b.



(a) Sub-circuit for Part I.

(b) Top-level circuit for Part I.

Figure 1: Rotating register for Part I.

1. *Data_IN* connects to the D input of the 4 registers.
2. *ParallelLoadn* controls loading a value into all 4 registers.
3. RotateRight in Figure 1b is 1 when you want to rotate the register to the right. When rotating to the right, you are loading a new bit from the left, therefore, the LoadLeft which controls the first mux is Figure 1a should be 1. When you rotate left (RotateRight = 0), you select the bit coming from the right.
4. *ASRight* is a special input to perform an *arithmetic shift right* on the register. Arithmetic shift is done to support shifting of *signed* numbers (textbook Section 1.4.6). You will learn about signed numbers in class in Module D2. For now, we will explain what you have to do to implement this functionality below.

This is how your design should work:

1. When *ParallelLoadn = 0*, the value on *Data_IN* is stored in the flip-flops on the next positive clock edge (i.e., parallel load behaviour). *RotateRight* and *ASRight* are ignored.
2. When *ParallelLoadn = 1*, *RotateRight = 1* and *ASRight = 0* the bits of the register rotate to the right on each positive clock edge (notice the bits rotate to the right with wrap around):

   $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
   $Q_0Q_7Q_6Q_5Q_4Q_3Q_2Q_1$
   $Q_1Q_0Q_7Q_6Q_5Q_4Q_3Q_2$
   ...
3. When *ParallelLoadn = 1*, *RotateRight = 1* and *ASRight = 1* the bits of the register rotate

to the right on each positive clock edge but the most significant bit is copied. This is called an *Arithmetic shift right*. Note that each clock cycle, a bit on the right is lost:

$Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
$Q_7Q_7Q_6Q_5Q_4Q_3Q_2Q_1$
$Q_7Q_7Q_7Q_6Q_5Q_4Q_3Q_2$
...

4. When *ParallelLoadn = 1* and *RotateRight = 0*, the bits of the register rotate to the left on each positive clock edge. *ASRight* is ignored:

   $Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
   $Q_6Q_5Q_4Q_3Q_2Q_1Q_0Q_7$
   $Q_5Q_4Q_3Q_2Q_1Q_0Q_7Q_6$
   ...

1.1 What to Do

The top-level module of your design should have the following signature declaration:

```
module part1(input logic clock, reset, ParallelLoadn, RotateRight, ASRight,
             input logic [3:0] Data_IN, output logic [3:0] Q);
```

Pre-Lab work:

1. *Prior to coming to lab,* draw a schematic for the 4-bit rotating register with parallel load. Your schematic should contain four instances of the sub-circuit in Figure 1a and all the wiring required to implement the desired behaviour. Label the signals on your schematic with the same names you will use in your System Verilog code.
2. Using the D Flip Flop code from Lab3, write a module to implement the circuit shown in Figure 1a.
3. Now, write a module that instantiates four instances of the previous module to implement Figure 1b.

   In Lab:

4. Simulate your rotating register with ModelSim to satisfy yourself that your circuit is working. In your simulation, you should perform the reset operation first. Then, clock the register for several cycles to demonstrate rotation in the left and right directions. Be prepared to justify that your test cases are enough to give confidence that your circuit is working.
5. When you are satisfied with your simulations, you can submit to the Automarker.

NOTE: If you do not perform a reset first, your simulation will not work! Try simulating without doing reset first and see what happens. Can you explain the results?

1.2 Running on the FPGA

If you wish to run your code on the FPGA, you can use the port mapping shown in Table 1.

module Port Name	Direction	DE1-SoC Pin Name
`clock`	Input	KEY[0]
`reset`	Input	KEY[1]
`Data_IN`	Input	SW[3:0] and HEX[0]
`ParallelLoadn`	Input	SW[9]
`RotateRight`	Input	SW[8]
`ASRight`	Input	SW[7]
`Q`	Output	LEDR[3:0] and HEX[4]

Table 1: Module port mapping to DE1-SoC/DE10-Lite pin names

2 Part II

2.1 Understanding Parameters

`Parameters` (textbook Section 4.8), are System Verilog constants that can be defined in a module. Parameters are used to enhance design flexibility by allowing you to modify the sizes of your input and output values without changing your code.

Listing 1 provides an example of a parameterized adder, where the parameter `WIDTH` specifies the bit-width of a, b, and sum, with a default value of 8. Instantiating the adder module without explicitly setting `WIDTH` creates an 8-bit adder, as shown below:

```
adder u0(a, b, sum)
```

Alternatively, you can instantiate a 16-bit adder by passing a value of 16 to the parameter WIDTH, as demonstrated here:

```
adder #(16) u0(a, b, sum)
```

```
module adder
#(parameter WIDTH = 8)(
    input  logic [WIDTH-1:0] a,
    input  logic [WIDTH-1:0] b,
    output logic [WIDTH-1:0] sum
```

```
);
    assign sum = a + b;
endmodule
```

Listing 1: Parameterized adder

2.2 Part II Design

In this part, you will design a counter that continuously outputs the hexadecimal values 0 through F, to an output called `CounterValue`. The rate at which the outputs change will be configured using the `Speed` input.

The top-level module of your design should have the following declaration:

```
module part2
#(parameter CLOCK_FREQUENCY=500)(
    input logic ClockIn,
    input logic Reset,
    input logic [1:0] Speed,
    output logic [3:0] CounterValue
);
```

You should build your counter with D-flip flops. Recall, you can build a counter by using a register and adding 1 to its value (textbook Section 5.4.1).

```
Q <= Q + 1;
```

Once you've built the counter, you will need to create a module that will increment the counter at different speeds.

The rate at which the numbers change based on the `Speed` input is given in the Table 2.

Speed[1:0]	CountRate	Description
00	Full	Once every clock period
01	1 Hz	Once a second
10	0.5 Hz	Once every two seconds
11	0.25 Hz	Once every four seconds

Table 2: Speed settings

For example, let's say `ClockIn` is 50 MHz, which is the clock available on the FPGA boards. In this case, *full speed* means that the display flashes at 50 MHz, i.e., 50 million times a

second. **Question**: If you ran a counter at 50 MHz and displayed the value on a HEX, what do you think you would see?

Running the counter at full speed is easy; you simply increment/decrement the counter by 1 every cycle. Supporting different speeds requires a `Rate Divider` module and a module to update the counter that will be displayed: `Display Counter`.

Note: Since this design uses two complex modules, it is important to draw a well-labelled schematic before writing any code.

Pre-Lab Work:

You should prepare the full Part 2 schematic prior to coming to lab. Your schematic should include block diagrams of the `part2` module and show the `Rate Divider` and a `Display Counter` modules inside it. You must have this schematic ready to show during your lab session.

2.3 Rate Divider

Let's start with the `Rate divider` module. A `Rate Divider` is a counter that is used to create a slow *pulse* given a faster clock signal. A *pulse* is when a signal is 1 for a single clock period but is 0 at other times. Figure 2 shows a timing diagram for a 1 Hz pulse for an input clock of 50 MHz.

Creating a pulse. To create a pulse, you need to count N-cycles of your input clock before generating your pulse. For example, to create a 5 MHz pulse given a 50 MHz clock, you count 10 cycles of the 50 MHz clock. As you can see, the sizing of your counter depends on ratio between your `clock frequency` and your `pulse frequency`. **Question**: For the case shown in Figure 2, how many bits should the counter be?

Outputting the pulse when the counter is zero can be done using a *conditional assign statement* like:

```
assign pulse = (RateDividerCount == 'b0)?'1:'0;
```

2.4 Making the Rate Divider flexible

To make the `Rate Divider` flexible, we want it to support two things: 1) counting different numbers of clock cycles and 2) different clock frequencies.

Counting different number of clock cycles. A common way to do this is to parallel load the counter with the appropriate starting value and count down to zero. With this

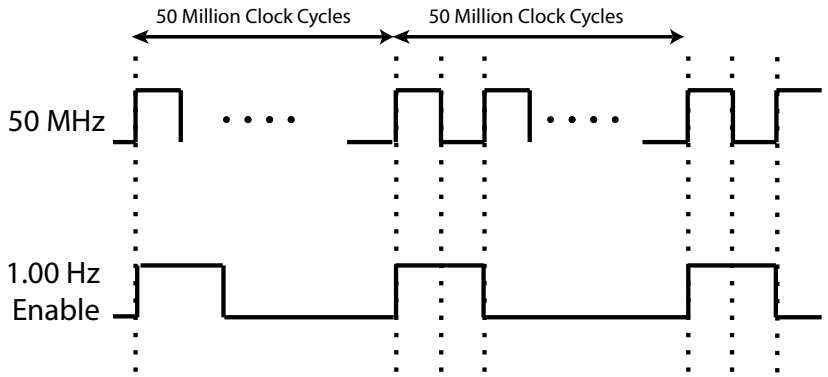


Figure 2: Timing diagram for a 1 Hz enable signal

approach, the end condition is always 0, and you can just load the counter with different starting values depending on the period you want to count. For example, if you want to count 50 million clock cycles, load the counter with 50 million - 1. **Question**: Why subtract 1? Do you think it would make a difference in the above case? What about the case of creating a 10 MHz clock from 50 MHz clock?

Supporting different frequencies. To support different clock frequencies, we will use parameters which were described in the previous subsection. We pass in a parameter, called `CLOCK_FREQUENCY`, to our `Rate Divider` module to specify the frequency of our clock. For example, the tester and marker use a clock frequency of 500 Hz, while the FPGAs use a clock frequency of 50 MHz. You must calculate how many clock periods you have to count to support the different `Speed` values, for a given clock frequency.

Sizing your counter. A flexible Rate Divider means that sizing your counter is non-trivial. One approach is to make your counter big enough to fit the largest number you will count to? For counting smaller values, the upper bits of your counter will simply be unused. Another, more efficient, way is to size the counter based on the `CLOCK_FREQUENCY` parameter. For example, to count N values, you will need $\lceil log_2(N) \rceil$ bits. You can then use the built-in `$clog2()` function in System Verilog, to use the exact right counter size for your design.

2.4.1 Building the final Rate Divider

Now that you know how to build a flexible `Rate Divider`, you can implement it to have the following declaration:

```
module RateDivider
```

```
#(parameter CLOCK_FREQUENCY=500)(
    input logic ClockIn,
    input logic Reset,
    input logic [1:0] Speed,
    output logic Enable
);
```

Important: To work with the Automarker, your `Rate Divider` must follow these requirements:

1. Your counter must count down to 0 and generates an enable pulse when it reaches 0.
2. If `Speed` changes while counting down, the counter should **continue to count down to 0** and only change speed after generating the enable signal.

2.4.2 Simulating the Rate Divider

Before proceeding, you are **strongly encouraged** to write a separate `do` file to test your `Rate Divider` module and make sure it works before proceeding. Your TA will ask to see this do file prior to helping you debug your circuit.

When simulating the `Rate Divider` module, pay attention to the number of cycles you have to simulate to check that the `Enable` pulse is working correctly. **Question:** What can you change to speed up simulation, while still ensuring that your module works correctly? This will also help you test your `part2` module later on.

2.5 Display Counter module

The `Rate Divider` module allows us to generate slower pulses, given a faster input clock. These `Enable` pulses from the `RateDivider` are used to drive the `EnableDC` signal on `DisplayCounter`. Recall that an enable signal determines whether a flip flop, register, or counter will change on a clock pulse. Compared to the `Rate Divider`, the `DisplayCounter` is much simpler. It should continuously output the hexadecimal values 0 through F, to an output called CounterValue. It will count up by 1 when EnableDC is 1.

The declaration for `DisplayCounter` is as follows:

```
module DisplayCounter (
    input logic Clock,
    input logic Reset,
    input logic EnableDC,
```

```
    output logic [3:0] CounterValue
);
```

Once again, you are encouraged to test your `DisplayCounter` circuit separately before proceeding. Once that is done, you can now put these parts together to build the final circuit.

2.6 Putting it all together

Hopefully, you have already tested your `Rate Divider` and `Display Counter` modules separately. If so, you can put them together in the `part2` module, test it and then submit it for marking.

2.7 Running your design on FPGA

To test your design on an FPGA, you need to make some changes to your code. First, to make it easier to see the numbers counting from 0 to F, you should instantiate a `HEX Decoder` module. You can re-use the `HEX Decoder` from Lab 3. Simply connect the `CounterValue` output from the `DisplayCounter` module to the input of the `HEX Decoder`.

Also, keep in mind that the FPGA uses a 50 MHz clock. So you will need to pass in the right parameter to your `part2` module.

Once you have made these changes, you can use the mapping shown in the Table below:

module Port Name	Direction	Pin Name
`Clock`	Input	CLOCK_50
`Reset`	Input	SW[9]
`Speed`	Input	SW[1:0]
`CounterValue`	Output	HEX0

Table 3: Module port mapping to DE1-SoC/DE10-Lite pin names

3 Part III – Optional for Bonus Mark

You may optionally complete and submit Section 3 for one bonus mark. This part builds on the previous parts but is fairly complex and may require a significant time investment.

In this part, you need to implement a `Morse code` encoder. Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of

dots (a short pulse), and dashes (a long pulse). For example, starting from A, the first eight letters of the alphabet are represented as:

A	• —
B	— • • •
C	— • — •
D	— • •
E	•
F	• • — •
G	— — •
H	• • • •

The input to your circuit is one of the eight letters shown in the table above. You must then output the appropriate Morse code for that letter to the output `DotDashOut` using short and long pulses. Short (0.5s) pulses represent dots and long (1.5s) pulses represent dashes. The time between pulses is 0.5 seconds. Similar to Part 2, you must determine how many cycles to count for 0.5 seconds, for a given `CLOCK_FREQUENCY`.

You should encode the pattern for each letter using a sequence of 1's and 0's, to correspond to on and off. Since the minimum time is 0.5 seconds, set 0s and 1s to be 0.5 seconds in duration. This means that a single 0 is a pause or off, a single 1 is a dot, and 111 is a dash.

First, write out the exact sequence of 0s and 1s corresponding to each of the letters you need to display. You will see that the sequences have different lengths. For simplicity and to be consistent with what the automarker expects, you should store all the letters using 12-bits. For example, the pattern for A would be stored as `101110000000`.

The top-level module for this part should have the following declaration:

```
module part3
#(parameter CLOCK_FREQUENCY=500)(
    input logic ClockIn,
    input logic Reset,
    input logic Start,
    input logic [2:0] Letter,
    output logic DotDashOut,
    output logic NewBitOut
);
```

Figure 3 shows a timing diagram of how your circuit should operate, for the letter *A*. The letter is selected by the `Letter` input using 000 for A, 001 for B, etc. The Morse code for

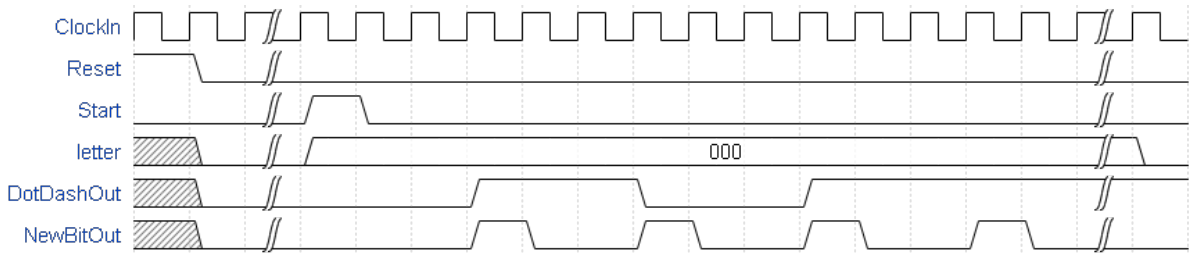


Figure 3: Timing diagram for Part 3

the letter is output when the `Start` signal is asserted, for 1 clock period. Figure 3 shows the first 4 bits of A being flashed, for a very low clock frequency. Using the clock in the marker or the FPGA will cause the `DotDashOut` signal to stay 0 or 1 for many more clock cycles.

To make sure your design meets the requirements of the marker, please note the following:

1. There maybe many cycles between `Reset` being de-asserted and `Start` being asserted. This is shown with the two wavy lines in Figure 3.
2. Note the delay between when `Start` is asserted and when `DotDashOut` starts displaying the sequence.
3. You must assert `NewBitOut` to 1 for 1 clock period for all 12 bits of the sequence, regardless of how many dots/dashes the letter needs. This is to ensure that the marker tests your outputs at the right cycles.
4. You can assume that the `Letter` input is held constant while you flash the entire sequence.

With the requirements of the design covered, let's look at how you should design your circuit.

3.1 Designing the encoder

Implementing the `Morse Code` encoder will use concepts you learned earlier such as the `Rate divider` and `Shift registers`.

Since you need to read each bit out, one at a time, you should use a `shift register` to load the letter to flash. When a letter is selected, use the `parallel load` feature of the shift register to load the pattern for that letter to the shift register. Then read each 0 or 1 individually out of a *shift r*egister at 0.5 seconds per read.

Similar to supporting different `Speed` values in Part 2, you should use a mux to select the letter to flash, based on the `Letter` input. You should use counters, to count the cycles that the `DotDashOut` signal should be high for. Lastly, you should create a pulse, similar to Part 2, for `NewBitOut`.

In attempting this part, we recommend that **prior to coming to lab**, you should draw a block diagram of the circuit you are implementing for this part.

3.2 Running on FPGA

To run on an FPGA, use the mapping shown in Table 4. Similar to part 2, make sure to update the `CLOCK_FREQUENCY` parameter since the FPGA uses a 50 MHz clock.

module Port Name	Direction	Pin Name
`ClockIn`	Input	CLOCK_50
`Reset`	Input	SW[9]
`Start`	Input	KEY[0]
`Letter`	Input	SW[2:0]
`DotDashOut`	Output	LEDR[0]
`NewBitOut`	Output	LEDR[1]

Table 4: Module port mapping to DE1-SoC/DE10-Lite pin names

4 Submission

When submitting to the Automarker make sure you have modules declared as shown below as the Automarker will be looking for modules with these exact signatures. Ensure you have properly followed the instructions prior to submitting your code.

4.1 Part I

For Part I, you need to submit a file named `part1.sv` with the following module in it:

```
module part1(
    input logic clock, reset, ParallelLoadn, RotateRight, ASRight,
    input logic [3:0] Data_IN,
    output logic [3:0] Q
);
```

4.2 Part II

For Part II, you need to submit a file named `part2.sv` with the following module in it:

```
module part2
#(parameter CLOCK_FREQUENCY=500)(
    input logic ClockIn, Reset,
    input logic [1:0] Speed,
    output logic [3:0] CounterValue
);
```

4.3 Part III (Optional)

For Part III, you need to submit a file named `part3.sv` with the following module in it:

```
module part3
#(parameter CLOCK_FREQUENCY=500)(
    input logic ClockIn, Reset, Start,
    input logic [2:0] Letter,
    output logic DotDashOut, NewBitOut
);
```