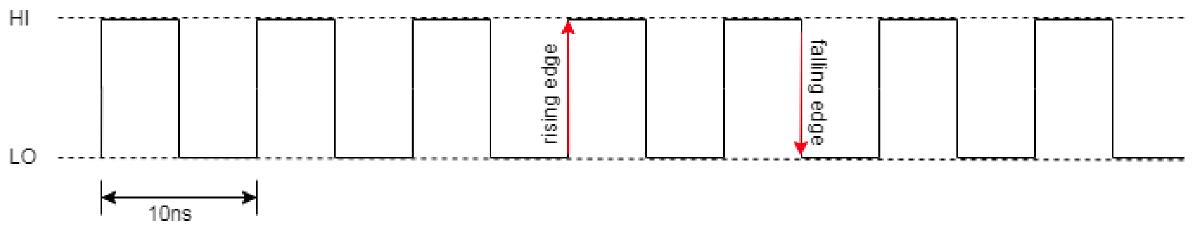
3C7 Digital Systems Design
Using Clocks

What is a clock?
A clock signal is one which varies from a logic high to a logic low periodically (usually a square wave). A clock signal with a period of 10ns (and hence a frequency of 100 MHz) is shown below:



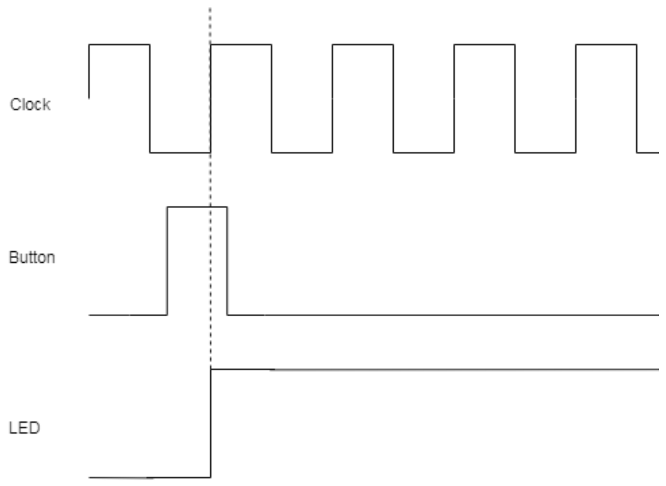
100MHz clock waveform

A transition of the signal from logic low to logic high is referred to as a **rising edge**. A transition of the signal from logic high to logic low is known as a **falling edge.**

Clocks in Sequential Logic
Thus far in 3C7, we have only looked at combinatorial logic, where the output of the circuit is a function of the current input only. However, in sequential logic, there is another factor: time. This means that the system will consist of a time sequence of inputs, internal states and outputs. The output will be a function not only of the current inputs, but also the sequence of previous inputs and outputs, and the clock. (Refer to Lecture 7 for more details).

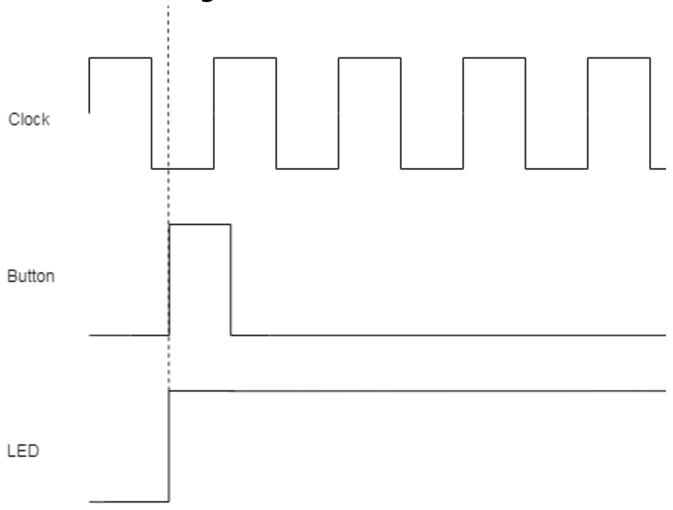
An adder is a combinational logic circuit – it takes two inputs, A and B, and outputs their sum (which depends only on the current value of A and B). A counter is a sequential logic circuit – it depends on the previous value output by the counter, derived from previous inputs, as well as the current inputs.

Synchronous vs. Asynchronous Sequential Circuits
In a synchronous circuit, all behavioural changes occur only on a clock edge. For example, if a button is used to turn on an LED in the circuit, it may only capture a button press on a rising edge.



Synchronous button

In an asynchronous circuit, the behaviour of the circuit may change at any time, regardless of whether at a clock edge or not.



Asynchronous button

Clocked (synchronous) sequential circuits are generally more common.

Clocks in Verilog: Blocking vs. Nonblocking Assignment
So far, as we have been looking at combinational circuits, we have used only blocking statements in our designs. A blocking statement uses the "=" operator. All other assignments are "blocked" until execution of the statement is completed.

A nonblocking assignment is associated with timing control and uses the "<=" operator. When used in a clocked always block (i.e. an always block with a clock edge in its sensitivity list) it means that all values in that block will be updated at the same time, on the clock edge. This means that the order of these statements within the block will not change the behaviour of the circuit and ensures that other assignments are not "blocked" during execution.

In general, nonblocking statements only should be used within clocked always blocks and blocking statements used elsewhere.

To emphasise this, and also help you think more clearly about what logic in your design combinational and what is sequential, 3C7 uses a specific naming convention. Consider a counter, that either counts conventionally in increments of 1, or if an input skip is high, will count in increments of 2. To simplify the example, presume the count always starts at one. There is another output max_tick which goes high when the maximum count of 255 is reached. Both the counter and max_tick are registered. We explicitly consider the _next value as the combinatorial version of the counter which directly changes depending on inputs that influence the counter. The _reg version is the registered or stored version on the counter and is denoted counter_reg. That is sampled (here) from counter_next on the rising edge of the clock, and will remain unchanged until the next rising edge of the clock. Note that the use of the non-blocking assignments ensure all the registers get updated at the same time, i.e. here at the clock edge. The code below will infer a d-type flip-flop (8 bits) to store counter_reg.

```
// clocked always block
always@(posedge clk)
begin
    if (reset) //reset to entire block - go back to start
    begin
      counter_reg <= 8'b0000_0001; //starts at 1
      max_tick_reg <= 1'b0; // no max count reached
    end
    else
    begin
      counter_reg <= counter_next; // pass comb value
      max_tick_reg <= max_tick_next;
    end // always@(posedge clk)
end

always @* // combinational logic to determine next value to register
begin
    if (counter_reg == 8'b1111_1111)
    // have reached max val, go back to 1, and assert max_tick to flag count is reached
    begin
          counter_next = 8'b0000_0001;
          max_tick_next = 1'b1;
    end
    else if(skip)
    // skip high means increment in 2s
    begin
        counter_next = counter_reg + 2'b10; //increment by 2
        max_tick_next = 1'b0; // no max reached
    end
    else
    // default to counting in normal way, since skip not high, no max reached
    begin
          counter_next = counter_reg +2'b01; //increment by 1
          max_tick_next = 1'b0; //no max reached
    end
end // always @*
```

Clock on FPGA

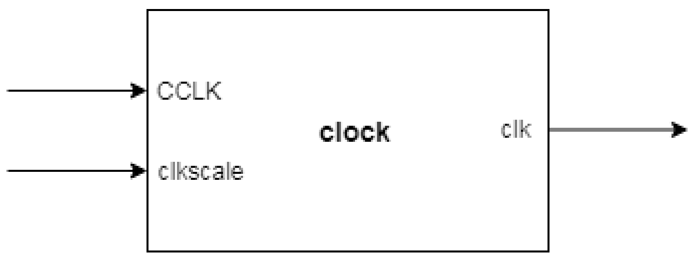
The Basys3 board includes a single 100MHz oscillator connected to pin W5. For some purposes, we may require a slower clock. To do this, we must change the period of the clock signal.

Specifying the Clock Using the XDC File
A create-clock constraint is used to specify the period and duty cycle of a clock. Simply including this constraint – and specifying the period you need for your design – will **not** alter the period of the crystal oscillator clock. It is used to make the compiler aware of the period so that it can figure out if other timing constraints are met (this is important if you think gate propagation delays may be a problem for your design). We will not be concerned with such delays in 3C7.

```
create_clock -period 10.000 -name clk_p -waveform {0.000 5.000} [get_ports clk_p]
```

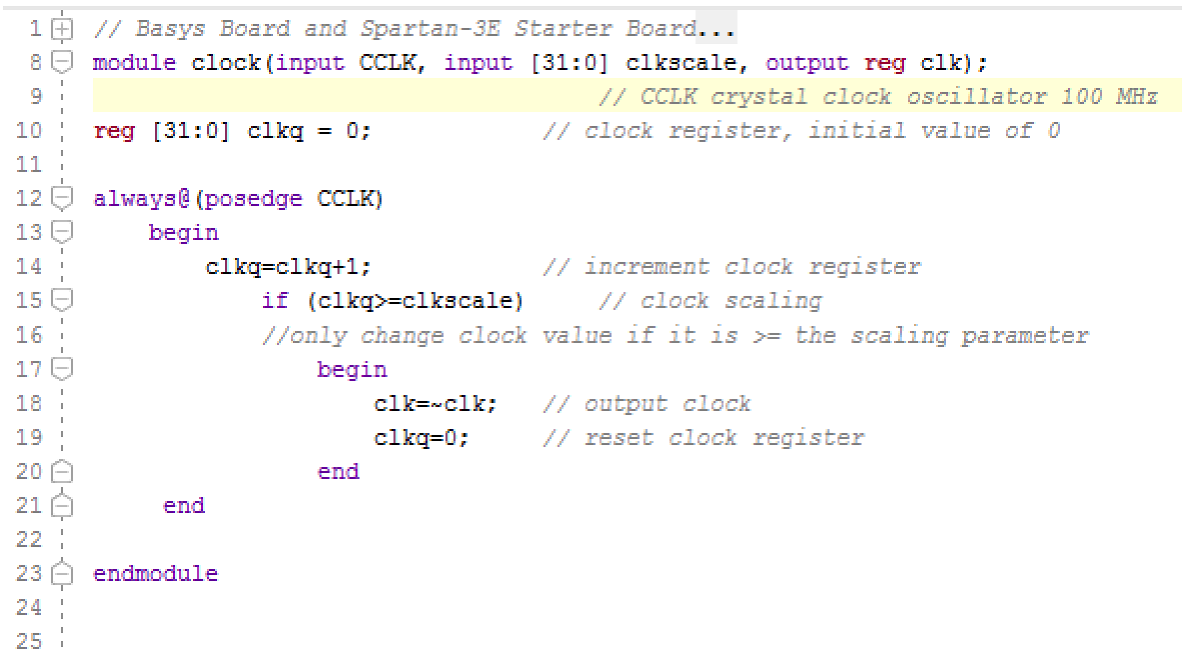
Create_clock constraint in XDC file. A clock period of 10ns is specified with "-period". The "-waveform" tag here specifies a 50% duty cycle (going high at 0ns and low at 5ns).

Manipulating the Clock Using a Clock Divider
A clock divider circuit has many applications. It takes the crystal oscillator clock from the FPGA board and a scaling factor as inputs, and outputs a clock with a lower frequency. The file **clock.v** is provided on Blackboard, in the Peripherals folder. You can use this clock divider module by instantiating it in your own designs.



Block diagram of clock.v module

The Verilog description of this module is shown below:



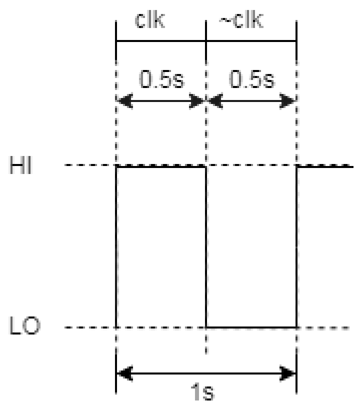
```
// Basys Board and Spartan-3E Starter Board...
module clock(input CCLK, input [31:0] clkscale, output reg clk);
                                    // CCLK crystal clock oscillator 100 MHz
reg [31:0] clkq = 0;            // clock register, initial value of 0

always@(posedge CCLK)
    begin
        clkq=clkq+1;            // increment clock register
            if (clkq>=clkscale)     // clock scaling
            //only change clock value if it is >= the scaling parameter
                begin
                    clk=~clk;   // output clock
                    clkq=0;     // reset clock register
                end
     end

endmodule
```

Code for clock.v module

Note: the above code is an exception to the general rule for using blocking and nonblocking statements. This is reasonable as it defines the clock to be used by the entire design.

In this module, **CCLK** represents the crystal oscillator clock, **clkscale** represents the scaling factor and **clk** represents the scaled output clock.
The register **clkq** here is used as a counter. It increments on every rising edge of the crystal clock. Once it has reached the value of the dividing factor, only then is the output signal inverted.

For a 1Hz (1second period) clock:
The input, CCLK, will be a 100MHz clock, with a rising edge occurring every 10ns, i.e. the signal changes/inverts every 5ns.
We want our output clock to have a rising edge every 1s. To do this, it must change every 0.5s.



1 period of a 1Hz clock signal

It takes $\frac{0.5}{10\times10^{-9}} = 50{,}000{,}000$ crystal clock ticks to reach 0.5s, hence our dividing factor (clkscale) will be 50000000. In general,

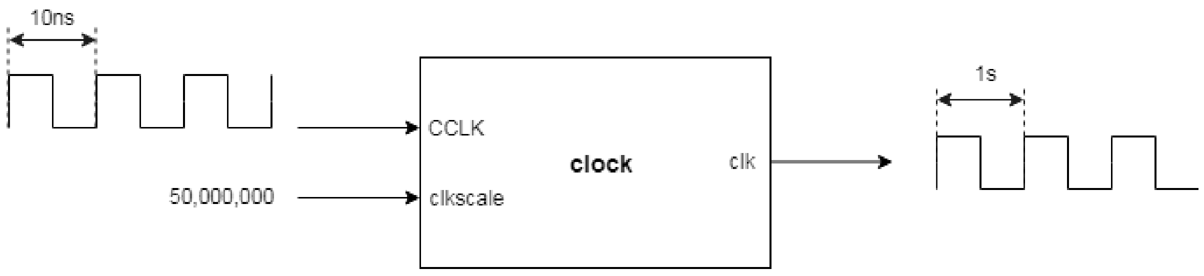
$$clkscale = \frac{desired\ period}{2 \times (CCLK\ period)} = \frac{CCLK\ frequency}{2 \times (desired\ frequency)}$$

The instantiation will look like this:

```
clock M0 (.CCLK(CCLK), .clkscale(50000000), .clk(LD0));
```

Code for instantiating clock module

Here, the 1Hz signal is used to drive an LED (**LD0**).



Producing a 1Hz signal using a clock divider

For a 32-bit clkscale input, the range of output clock periods can vary between 10ns and 85s.