Australian National University

Lab Two

Implementing designs onto FPGA and Essential sequential designs

Overview

This lab completes your awareness of the basic digital design steps by following your Lab One projects through to implementation.

It will then introduce you to the world of synchronous sequential logic design, where information flow is often controlled by a combination of clocks, resets and enables. These essential signals and constructs are ever-present in FPGA designs.

Note:
This lab is extensive and will require your full attention for the entire duration. **You are highly encouraged to go through this document before attending the lab.**

When adding a file (design source, constraints and simulation) to the project, make sure the option the option "Copy sources into project" is checked.

In this lab, you will learn to:

- Identify and define real-world inputs and outputs for the purpose of FPGA interfacing.
- Synthesise your designs and implementing them on an FPGA chip.
- Create variable output rate clocks using clock dividers.
- Implement synchronous resets to return registers to known states.
- Use enable signals to control the behaviour of different design elements.

You will also become increasingly familiar with:

- Available peripherals on the (Basys3) FPGA board.
- Verilog constructs such as procedural blocks and non-blocking assignments

Australian National University

Table of Contents

Australian National University

Lab Overview

Learning Outcomes

By completing this lab, you will learn to:

1. Create constraints files to specify the properties of the interfaces with physical I/O on the FPGA board.
2. Synthesise your design and implement it on the FPGA chip.
3. Design sequential logic processes in Verilog and create basic designs such as counters.
4. Define dedicated clock signals derived from a physical oscillator on the FPGA board.
5. Use synchronous reset and enable signals to control sequential logic processes.

To complete this lab, you will require access to Vivado and the Basys 3 Board.

Link to Basys 3 Reference manual:
Basys 3 Reference Manual - Digilent Reference

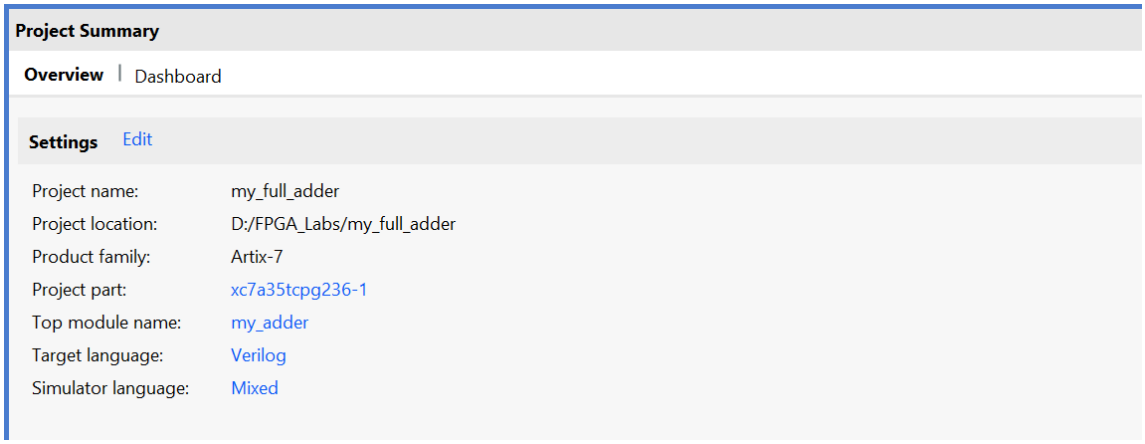
Activity 1: Implementation of the Full Adder [3 marks, approx. 15 min]

Procedure

Step One: Open the full adder project

Open the full adder project in Lab 1. Check the source content and hierarchy tree of your design. The "my_adder" module should be listed as a design source and correctly identified as the "top module" of your design. **The test bench source should only appear in the list of simulation sources.** Test benches are only useful for virtual simulation of a design and are not deployable to the FPGA hardware.

Please check the Project part in the Project Summary window. The part number **must** agree with the FPGA chip on your board. If the Project Summary window is hidden, click on 'Window' and then choose 'Project Summary'.



Step Two: Plan your I/O and create a constraints file

Peripherals (switches, LEDs, ports, etc.) are hardwired to various pins located on the underside of the FPGA chip. To interact with the FPGA in a meaningful way, you need to match the I/O entities of your design with available hardware entities.

Hardware input/output entities are characterised by their location (an alphanumeric identifier, e.g., R2, W19) and by a series of properties, some of which are configurable, which determine the electrical behaviour of the associated pin. A key property to be aware of is the logic standard, i.e., the expected voltage ranges for digital signals.

The information required to match design and hardware entities is in the **constraints file**.

The General-Purpose I/O schematic provided by the Basys3 reference manual is in Figure 1.

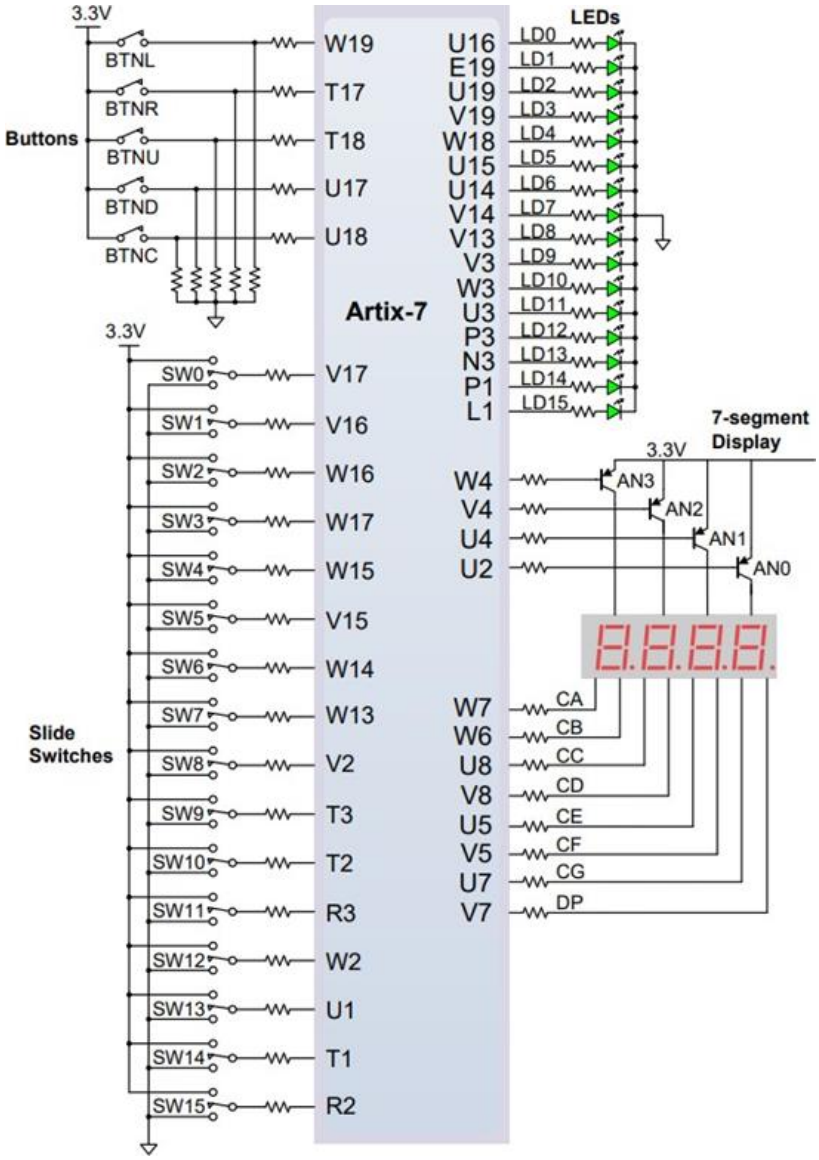


Figure 1 GPIO Schematic of Basys 3.

Slide switches, buttons are connected to the 3.3 V rail. The LEDs connected to pins U16 – L1 are supplied with 3.3 V since they are connected to a 3.3 V voltage bank on the FPGA chip via a resistor. The resistors connected in series with LEDs restricts the current when the LEDs are forward-biased.

You can directly read the pin locations on the Basys3. They are usually printed next to or below the hardware I/O and enclosed in brackets.

Now prepare the constraints file:

1. Right-click the Constraints directory under the Sources panel and select 'Add Sources'.
2. Select the option to 'Add or Create Constraints' and then click Next.
3. Name your constraints file as 'full_adder_constraints_Basys3'.
4. Once finished, the constraints file you just created should appear in the 'Constraints' directory in the Sources pane.

Design constraints inform Vivado's implementation tools of the pin location and I/O standard for all physical connections to the FPGA. Xilinx uses *.XDC files (which stands for Xilinx Design Constraints) which are written in a language called Tcl. Properties of specific interfaces can be set using the 'set_property' command.

In this activity, the inputs X, Y, and Cin are connected to the slide switches. The outputs Z and Cout are connected to the LEDs.

5. To set the properties of the slide switch connected to pin R2, type in the following commands:

```
set_property PACKAGE_PIN R2 [get_ports X]
set_property IOSTANDARD LVCMOS33 [get_ports X]
```

   The `PACKAGE_PIN` property identifies the physical pin location at which we want to connect our signal. The `IOSTANDARD` property specifies the expected type of logic signal, including its logic voltage levels. LVCMOS33 stands for Low-Voltage CMOS 3.3 Volts.

6. To set the properties of the LED connected to pin L1, type in the following commands:

```
set_property PACKAGE_PIN L1 [get_ports Z]
set_property IOSTANDARD LVCMOS33 [get_ports Z]
```

7. Add the constraints for another two slide switches (one for Y, the other for Cin) and the second LED (for Cout).

Note: To comment out text in the constraints file, please use the # key. Also, if you need to write in-line comments in the constraints file, please put a semicolon between the command and the # key.

Step Three: Synthesize and implement your design

1. Click on 'Run Synthesis'. After the design is synthesised, click on 'Run Implementation'. Finally, click on 'Generate Bitstream' and a bit-file with the file extension *.bit is generated. This file, which will have the name as your top-module, should be called 'my_adder.bit'.
2. Upon completion, Vivado will present you with a pop-up window. Select 'Open Hardware Manager' and click Next.

3. Connect your FPGA development board to your computer. Power up the board. A default FPGA design should load demonstrating the board's features.
4. At the top of Vivado, you will notice a green bar has appeared after opening the Hardware Manager. Click 'Open Target' then 'Auto Connect' to establish a connection.
5. Click 'Program Device'. The default design will have been replaced by the full adder. Check if the full adder behaves as intended.

Please show your work to the tutor.

Activity 2: Overflow counters
[4 marks, approx. 30 min]

Overflow counter is a counter that can overflow back to zero when it fills up.

Procedure

Step One: Create a new project in Vivado

1. Create a new project called 'overflow_counter'.

Step Two: Create a new Verilog design source

1. Create a new Verilog design source 'overflowCounter'.
2. The module's inputs are clk, led, reset and enable.
3. Note the syntax for the bus entity `[15:0] led`. The square brackets indicate that signal 'led' comprises of 16 bits, of which bit number 15 is the leftmost (most significant) bit, while bit number 0 is the rightmost (least significant) bit.

```
module overflowCounter(
    input wire clk,
    input wire reset,
    input wire enable,
    output wire [15:0] led
    );
```

Step Three: Describe the behaviour of a 32-bit overflow counter in Verilog

1. Declare an internal 32-bit reg called `counter`. This entity will not be part of the I/O list of signals, i.e., it will be declared in the body of the module code.

```
reg [31:0] counter;
```

Note: The use of `reg` is necessary as we will be describing the behaviour of the counter using a **procedural block**.

2. The counter should have the following behaviours:

2.1 A every positive (i.e. rising) edge of the clock signal, the counter's value will increase by 1. It is achieved by a procedural block that will be edge-triggered by the clock. This is specified by the condition `posedge clk` .

2.2 When reset == 1, the counter's value resets to 0. When reset == 0, the counter increments.

2.3 The counter will increment when enabled (enable == 1) and stop incrementing when disabled (enable == 0).

3. The complete procedural block is shown here

```
always @(posedge clk) begin
      if (reset == 1'b1) begin
            counter <= 32'd0;
      end else if (enable == 1'b1) begin
            counter <= counter + 1'b1;
      end else begin
            counter <= counter; // redundant but harmless line
      end
end
```

The contents of `@(…)` is called a sensitivity list and describes the conditions or events that will activate the process. This code basically says: "The behaviour described within this procedural block is triggered at every positive edge of `clk`".

This type of procedural block is called a *'clocked process'* and forms the basis of practically all sequential logic designs in FPGAs. Without clocked processes, the design cannot support dynamic states (i.e., memory, operations, events or conditions that vary over time).

IMPORTANT:

Note the use of the operator `<=` to make the assignment instead of using just an equal sign. The operator `<=` introduces a **nonblocking assignment. You should always use nonblocking assignments inside clocked processes**.

4. Assign the most significant 16 bits of `counter` to `led`. This line should be **outside** the always block.

```
1    assign led = counter[31:16];
```

With a clock at 100MHz, counter LEDs would blink too quickly for the human eye. Setting only the most significant bits of the counter as output slows down the blinking significantly.

5. Review the elaborated RTL schematic.

Step Four: Constrain your design

1. Create a constraints file called 'overflowCounter_constraints_Basys3'. Constraints file for the Basys 3 is available online. Link to the constraints file is provided here:

   digilent-xdc/Basys-3-Master.xdc at master · Digilent/digilent-xdc · GitHub.

   Simply copy and paste the lines you need. Modify the names of the I/O ports so that they agree with your design.

   In the linked constraints file, multiple properties are set in a single line of code. This uses

the flag -dict followed by any required <property><value> pairs. For instance,

```
set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 } [get_ports {led[0]}]
```

A traditional but equivalent way of writing the constraint is

```
set_property PACKAGE_PIN U16 [get_ports {led[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]
```

Note: to identify a single bit in a bus for constraints file purposes, curly brackets need to be used, e.g, `{led[0]}`. The curly brackets are required because Tcl interprets anything in square brackets as a function, which means that passing an argument 'led[0]' into the `get_ports` function would cause Tcl to call the function '0' and ignore the square brackets as literal characters. Tcl will treat anything within curly brackets as literal characters, which allows us to achieve correct identification of our entity.

Create the constraints for the 16-bit output led.

2. We are now going to recruit a clock for the input `clk`. Use the following two lines for clk. It generates **a 100 MHz clock with period of 10 nanoseconds**.

```
# Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -period 10.00 [get_ports clk]
```

The command create_clock informs the implementation tools that the signal entering the FPGA on pin W5 is a clock and must be considered separately to the other general purpose I/O (e.g. LEDs, switches). **The creation of a dedicated clock is crucial because they are routed separately throughout the FPGA along dedicated 'clock lines', the purposes of which are to minimise latency.**

3. Connect the reset signal to the centre push button.
4. Connect the enable signal to the rightmost slide switch.

Step Five: Synthesise, implement and deploy your overflow counter

1. After the device is programmed, switch on enable. You should see the LEDs turning on and off at different rates. The LEDs on the left should oscillate more slowly than those on the right.
2. Test the function of reset by pressing the centre push button.
3. Try resetting the counter while it is disabled. Does it reset the way you expect?

   Reflect on the relative priority of enable and reset signals from a system operation perspective, and on how different coding choices may impact this.

Please show your work to the tutor.

Australian National University

Activity 3: Clock division using an overflow counter
[5 marks, approx. 40 min]

In many cases, it may be necessary to generate clock signals slower than 100 MHz. A very efficient way to do this is to perform clock division using an overflow counter.

The simplest way to generate a clock divided by a factor of 2^n is to output the (n-1)-th bit of an overflow counter. 'n-1' arises from bus indexing starting from 0. The table below should give a clear visual representation of this (look at the alternating dark/light cells down each column).

Consider a 4-bit counter being incremented at a rate of 100 MHz (period of 10 nanoseconds):

Time (ns)	counter[3]	counter[2]	counter[1]	counter[0]
0	0	0	0	0
10	0	0	0	1
20	0	0	1	0
30	0	0	1	1
40	0	1	0	0
50	0	1	0	1
60	0	1	1	0
70	0	1	1	1
80	1	0	0	0
90	1	0	0	1
100	1	0	1	0
110	1	0	1	1
120	1	1	0	0
130	1	1	0	1
140	1	1	1	0
150	1	1	1	1
	6.25 MHz	12.5 MHz	25 MHz	50 MHz

The least-significant bit (bit 0) of the 4-bit counter is a periodic signal with a frequency of 100 MHz / 2^1 = 50 MHz, whereas the most-significant (bit 3) bit has a frequency of 100 MHz / 2^4 = 6.25 MHz.

So, if we want to derive a 1 kHz clock from the 100 MHz master clock, how big must the overflow counter be? Use the equation: $f_{div} = f_{clk} / 2^n$ to find out!

Notice that division by powers of two cannot yield a divided clock running at exactly 1 kHz. The closest that can be achieved is 762.94 Hz, which can be generated by observing the most-significant bit of a 17-bit counter incrementing at 100 MHz.

Procedure

Step One: Create a new project in Vivado

1. Create a new project 'overflow_clock_divider'.

Step Two: Create a new Verilog design source

1. Create a new Verilog design source 'overflowClockDivider'.
2. Define the module's inputs to be 'clk', 'reset' and 'enable'. Declare its output to be 'dividedClk'.
3. Inside the module, declare an internal 17-bit reg 'counter' and describe its behaviour with both the reset and enable signals using a procedural block.

```
`timescale 1ns / 1ps

module overflowClockDivider (
      input wire clk,
      input wire reset,
      input wire enable,
      output wire dividedClk
);

      reg [16:0] counter;

      always @(posedge clk) begin
            if (reset == 1'b1) begin
                  counter <= 17'd0;
            end else if (enable == 1'b1) begin
                  counter <= counter + 1'b1;
            end
      end

endmodule
```

4. Continuously assign `dividedClk` to be the most-significant bit of the 17-bit counter. You only need to add one line to the above code.

Step Three: Simulate your clock divider

1. Import the testbench file called 'TEST_overflowClockDivider' from the supplied resources. **Make sure the option "Copy sources into project" is checked.**
2. Inspect the testbench file and make sure you can identify the following elements:
    a. signals `clk`, `reset` and `enable` of type 'reg' , which are used to provide the test inputs
    b. a wire to read the output divided clock, called `dClk`
    c. an instance of your overflowClockDivider module, named UUT (Unit Under Test), duly connected to the relevant testbench signals
    d. a behavioural block creating the master 100 MHz clock, using the keyword `forever`

e. a sequence of assignments determining the sequence of test inputs. These are defined inside an `initial` block.

3. Run the behavioural simulation. When the simulation window appears, press the *'zoom-to-fit'* button at the top of the simulation window.
4. The default simulation time is probably not long enough to see if the clock divider module is behaving as designed. You can run the simulation for longer by typing in a desired simulation time and then pressing the play button with a τ symbol next to it in the menu at the top of the Vivado window.

Figure 3 Simulator toolbar showing controls to extend a simulation.

Simulate the clock divider module for an additional **10 ms** and then zoom out. Measure the period of `dClk` using two cursors (using shift + left click to generate additional cursors) and positioning them at the beginning and the end of a period of `dClk`. The measured period should be 1.31072 ms, corresponding to a frequency of approximately 762.94 Hz.

Please show the simulation results to the tutor before proceeding.

Step Four: Modify your code to generate an approximately 1 Hz clock

Modify 'overflowClockDivider.v' to generate a clock at a frequency close to 1 Hz. You need to change the size of the sounter.

Do not simulate this. If you do, the simulation will take a very long time to complete due to the large number of timesteps (1 second is 10^9 ns!).

Step Five: Constrain your design

Create a design constraints file called 'overflowClockDivider_constraints_Basys3'.

You can save some time by copying the constraints you created for Activity 2.

You **only need to constrain one LED** as the output.

Step Six: Synthesis, implement and deploy your design on the FPGA chip

Check that the LED flashes at approximately 1 Hz (it should be a little slower). Also, please test the functions of reset and enable.

Please show your work to the tutor.

Australian National University

Activity 4: Flexible clock division and heartbeat generators
[8 marks, approx. 1 hr 30 mins]

We now try to overcome the limitation of 2^n clock division. We look for an alternative design approach for a clock divider which allows for the creation of an arbitrary frequency. This involves resetting a counter every time it reaches a specific value instead of relying on its periodic overflow behaviour.

For example, if we want to derive a 1 MHz clock from a 100 MHz master clock, then we need to generate a signal that flips between 1 and 0 every 500 nanoseconds. One way to do this is to increment a counter by one at 100 MHz and manually reset it to zero every time it reaches 49 (i.e., after 50 clock cycles). The counter will therefore reset every 500 nanoseconds, and every time it does we can invert the current value of the clock.

Unlike the clock, which usually has a 50% duty cycle (half the time it is 1, half the time it is 0), a **heartbeat** is a signal that delivers a pulse of one clock cycle duration at regular intervals. Heartbeat generators are quite useful as **event sequencers**, i.e., when used to trigger the start of some other circuit event with precise timing.

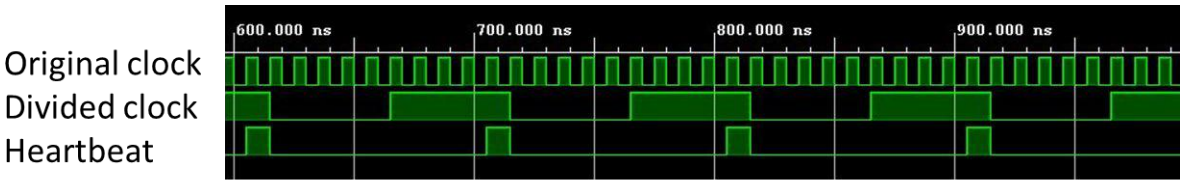


Figure 4 Example original clock, divided clock and heartbeat.

Procedure

Step One: Create a new project in Vivado

1. Create a new project 'arbitrary_clock_divider'.

Step Two: Create a new Verilog design source

1. Create a new Verilog design source 'clockDividerHB'.
2. Define the module's inputs to be 'clk', 'enable' and 'reset', and its outputs to be 'dividedClk' and 'beat'.
3. Set the types of `clk`, `reset`, `enable`, `beat` to be 'wire', and `dividedClk` to be 'reg'.

Step Three: Force the counter to reset after a specific period of time

1. Inside the module, create a 32-bit counter of type reg.

```
reg [31:0] counter;
```

Our desired behaviour is to increment the counter for a specific number of clock cycles before resetting it back to zero. In general, we can refer to the number of clock cycles we need to count as 'threshold'. The behaviour of our periodically resetting counter can

thus be described logically by the statement: "If the counter is greater than or equal to the threshold minus one, set its value to 0".

2. To declare `THRESHOLD`, we are going to use a `parameter`, which can be used to declare constants. Parameters are useful to support a powerful coding technique called **parameter substitution**.

   To generate a 1 Hz clock, at a master clock rate of 100 MHz we will need to count for 50,000,000 clock cycles each time we invert the clock. The final module declaration should look like

```
module clockDividerHB #(parameter integer THRESHOLD = 50_000_000)(
    input wire clk,
    input wire reset,
    input wire enable,
    output reg dividedClk,
    output wire beat
    );
```

   The reason for writing `THRESHOLD` in all caps is a style choice, which helps to quickly identify parameters in the code.

   Note the declaration of a parameter did not involve the usual notation (i.e., `<size>'<radix><value>`) to specify the numeric value. The reason for this is that `THRESHOLD` has been declared as an *integer* (a type that assists with coding but *is not implemented* in programmable logic), meaning it will be interpreted as a decimal integer by default.

   With `THRESHOLD` declared, we can describe the behaviour of the counter.

```
always @(posedge clk) begin
      if (reset==1'b1 || counter >= THRESHOLD-1) begin
            counter <= 32'd0;
      end else if (enable == 1'b1) begin
            counter <= counter + 1'b1;
      end
end
```

Step Four: Describe the behaviour of the divided clock in Verilog

1. Create a new clocked process with two conditions: one for the input reset, which should set `dividedClk` to zero if reset is true; and the other to invert the current value of `dividedClk` every time the counter reaches `THRESHOLD-1`.

```
1   always @(posedge clk) begin
2         if (reset == 1'b1) begin
3               dividedClk <= 1'b0;
4         end else if (counter >= THRESHOLD-1) begin
5               dividedClk <= ~dividedClk;
6         end
7   end
```

   This clocked process will execute concurrently with the one describing the behaviour of

the counter, which means that both processes will respond to all logic conditions at the same time.

Note that there is no explicit `else` statement included in this procedural block. In sequential logic this is sometimes acceptable, while it is never acceptable in a combinational logic block due to inferred latch.

Step Five: Introduce a heartbeat signal output

Heartbeat generation can be obtained by introducing a simple combinatorial function which yields a TRUE output for 1 clock cycle and only once per every divided clock's period. There are countless possibilities to achieve this. Here is one:

```
assign beat=(counter==THRESHOLD-1)&(dividedClk);
```

Step Six: Simulate your arbitrary clock divider and HB generator

Please use the provided 'TEST_clockDividerHB.v' to simulate your clock divider. Pay attention to the module instantiation.

```
clockDividerHB #(
        .THRESHOLD(50_000)
) UUT (
        .clk(clk),
        .reset(reset),
        .enable(enable),
        .dividedClk(dclk),
        .beat(beat)
);
```

In the instantiation, `THRESHOLD` has been overridden from its default value of 50,000,000 to 50,000. This means that your module will generate a divided clock frequency of 1 kHz. Review the simulation result and use two cursors to measure the period of `dividedClk`. **Do not simulate the 1 Hz case.**

Please show the simulation results to the tutor before proceeding.

Step Seven: Modify your design to meet your experimental conditions

The heartbeat signal is too short to be picked up by human eye if it is connected to an LED. Therefore, we need to increase the duty cycle of the heartbeat signal. Suppose we wish the duty cycle to be 20%, we need to declare another parameter called `ON_TIME`. Create a new design source file called 'clockDividerHB2'. The module declaration should look like

```
module clockDividerHB2
    #(parameter integer THRESHOLD = 50_000_000,
      parameter integer ON_TIME = 20_000_000)
    (
    input wire clk,
    input wire reset,
    input wire enable,
    output reg dividedClk,
    output wire beat
    );
```


Apart from the expression for beat, the remaining lines should be the same as those in clockDivederHB. The expression for beat is:

```
assign beat=(counter>=THRESHOLD-ON_TIME)&(counter<=THRESHOLD-1)&(dividedClk);
```

Note: Changes applied in this step cause the output signal to be no longer a true heartbeat. However, they are useful for us to be able to visualise a (pseudo) 'beat' for the purposes of this exercise. It is most common for heartbeat signals to remain internal to a logic design, i.e., they are not usually output.

Step Eight: Simulate your design

Please use TEST_clockDividerHB2.v provided in the Wattle resources folder.

Step Nine: Implement your design on the development board

1. Create a constraints file.
2. Run Synthesis, Run Implementation, Generate Bitstream, then program the device.
3. Program the board and check visually that the LED corresponds to the 1Hz clock flashes at 1 Hz.
4. The LED corresponds to the heartbeat is only ON for a very short time just before the LED corresponds to the 1Hz clock switches from ON to OFF.

Please show both the simulation result in Step 8 and the deployed result in Step 9 to the tutor.

End of Lab Two