Lab 2: Gambling

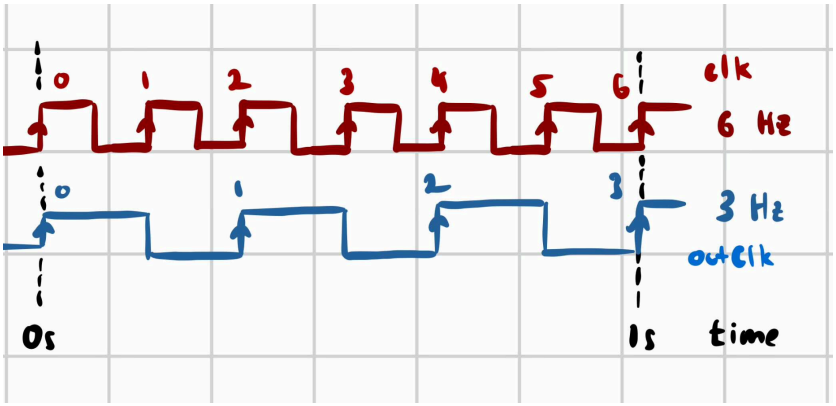
Introduction

By the end of this lab, you'll be familiar with clock dividers, block diagrams, and finite state machines! So far we've taught you the basics of combinational versus sequential logic, and we've told you how this sort of logic is implemented via registers, and inside those, chained flip flops, D-latches and gates at the base level. You'll be putting this all to use in this project!

I. Clock dividers

It's about time...

1. The clock divider module



Recall that a clock divider's purpose is simply to divide an incoming clock signal into a slower one. Here you'll write a module to do this.

The short version is this. You tell your clock divider three things: how fast your input clock is, how fast you want the output clock to be, and you pass in the base clock signal. In return, you receive the output clock signal. This is described in greater detail below.

The incoming clock signal is your FPGA built-in 50 MHz clock. The base speed is constant and thus we can make it a *parameter* with a default value of 50 MHz. This enables you to use a different input clock if needed. Parameters are computed at compile-time and don't change past that. The clock speed you need, however, might vary as your circuit is active—it will be an input to the module.

Your clock divider's output signal will need to be generated sequentially, not combinationally. This is to ensure the input and output clocks are in sync with each other.

Your `clock_divider` module will have the following:

- A clock signal (`clk`), speed value in Hz (`speed`), and a reset button (`rst`) as inputs.
- A clock signal as an output (`outClk`)
- A `parameter` for the base speed, with a default value 50 MHz.
- Your speed input should have enough bits to indicate a clock speed of at most 1 MHz. For faster clocks, we use a different tool that we'll get to at a later time!
- The clock divider uses a counter to determine when to flip the clock signal. The counter's maximum value is your division ratio.
    - If your base clock is 16 Hz and your output speed is 4 Hz, your counter should increment every clock cycle, and reset once it hits 3 (we start the counter range at 0, so 4 clock cycles on the 16 Hz input corresponds to a single cycle on the 4 Hz output — the output clock is 4 times slower).
- This module requires both a sequential and combinational block. We try to keep as much of the logic in the combinational block, reserving the sequential block for synchronous assignments (assigning values each new clock cycle)
    - In the combinational block, determine the value of the output clock for the next clock cycle. If the reset button is pressed, set the output clock to 0. If the counter is less than halfway to the maximum, set the output to 0. Otherwise, set the output to 1. This ensures a 50% duty cycle, which will be sufficient for most of our peripherals.
        - A common naming convention is `regName_d` to indicate the value of `regName` on the next clock cycle.

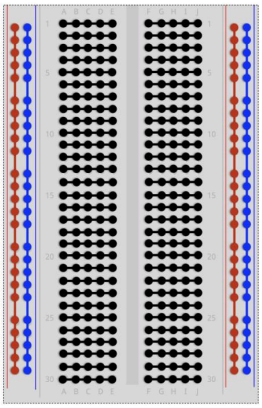
2. Testing your clock divider

You'll be creating a testbench to test your clock divider! The idea is to create a simulated clock, and observe the output from your clock divider module. How do you do this?

First, specify a timescale: `` `timescale 1ns/1ns ``
You can declare a 1-bit output, to represent the output clock signal from your clock divider module. Instantiate your clock divider and pass in a 1-bit input, which you can declare as the register `clock`. Then, make use of `initial` and `always` blocks to flip your clocks after a number of simulated delays. Recall that `#N;` inside a block will delay simulation by `N` nanoseconds using the above timescale.

II. Buzzers & Breadboards

Is that an OPS iPoduino reference?



Each dot here is a pin on the breadboard. The lines indicate that the dots are electrically connected. Each rail, or separate dotted line, represents a *node* in your circuit. Your buzzer has two leads — these will be connected to two separate *nodes*. One of these nodes will be connected back to ground on your FPGA, the other end to an IO pin on your FPGA.

You will pass a clock signal to your buzzer to generate a sound. The frequency of your signal will determine the tone that results. Different musical notes, such as a C or an A#, will correspond to different frequencies! Your task here is to write a module that uses a clock divider to generate a clock signal at a chosen frequency, and you should drive your buzzer at that frequency to verify that it will sound.

III. Slot Machines

Don't tell mom

1. Setup

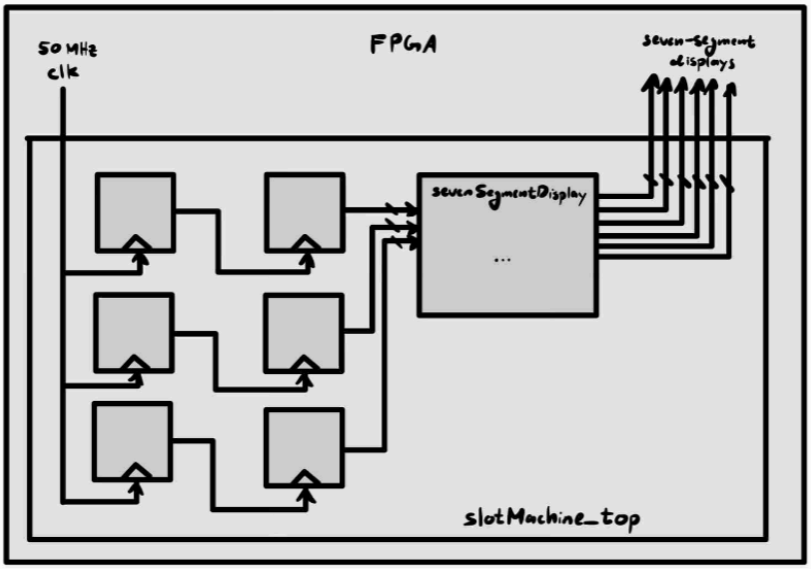
Your FPGA can display six digits; let's split this into three groups of two. We'll then create a slot module. The slot module operates on a clock, and its function is simply to change a number (We describe how exactly the number changes here). The three numbers will be displayed using the segment display you implemented in Lab 1.

The next two sections involve no Verilog. These will help you plan the structure of your project—check in with us after doing these, then move to the next two.

2. Block diagram

At a glance, your block diagram tells you how your project is structured. Here you'll plan out the connections between your modules; which inputs and outputs go where, which modules can encompass others.

Unfortunately it looks like we forgot to complete the block diagram. Here's what we've got so far; fill out the rest and run it by us before proceeding!



- Each block represents a named module.
- Wires going from the FPGA and the top-level module will need to be routed; i.e. you'll see them in your pin planner and will have to connect them! These pins are responsible for the clock from within the FPGA, the switches, and more.
- Ports coming from the smaller modules will need to be declared as wires and must be driven by the outputs of the module (e.g. the outputs of your clock divider are used again in your slot modules).
- The triangle at the bottom of the block indicates the module is sequential and accepts the corresponding wire as its clock signal. Your clock divider provides a clock signal; where does its own clock signal come from?
- A wire with a line through it indicates the wire is made of many bits. These multi-bit wires are called *buses*.

3. The finite state machine

Your slot machine will have four states:

1. **SET:** In this stage, your six digits should reset to some fixed values when you press the `reset` button at any point.
2. **RUN:** When you press the `start/stop` button, your slot machine will start to run. Each slot should run at a different speed.

3. **STOP:** When you press the `start/stop` button while your slot machine is running, your machine will stop, and blink with the values it landed on.
4. **WIN:** While in the stop state, if the three numbers on the machine match, the machine stops, blinking at a different speed than in the `STOP` state, and will activate the buzzer.

The module represented by your FSM in will have a sequential and combinational block. The combinational block determines the next state based on the inputs and perhaps the current state, while the sequential block includes the actual state change (you want state changes to be *synchronous*).

Be sure to show us your FSM!

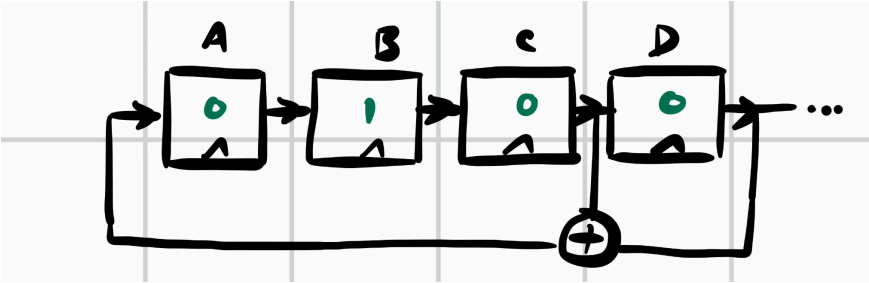
4. The slot module

Your slot module can take two inputs: one for its clock, and one to determine if the slot is "running" or not. How many bits should each input be?
The module outputs a number to be displayed. A 4-bit number will do for now!

One way to introduce a sense of "randomness" is to use what's known as a linear feedback shift register, described below.

- Begin with a chain of registers; each holds one bit.
- With each clock cycle, right-shift the bits.
- What goes into the leftmost register? We can apply operations on the bits that were there previously and feed that back into the register chain.
    - The choice of which *taps*, or bits in the chain, that we use will determine the output signal. That signal will *appear* random until it eventually repeats itself.

Your task is to implement such a shift register! A simple way to do this is to make use of bit concatenation when you write the expression to assign the next set of bits to your register (See Verilog Docs/FAQ). *Hint: should the expression in question be combinational or sequential?*



Suppose these are the values at t=0. One clock cycle later, the bits are right-shifted. That is, B-C-D becomes 0-1-0. The bits in C and D at t=0 are XOR'd together, resulting in a 0 at A at t=1. The first few iterations yield:

Cycle	A-B-C-D	Output (Decimal)
0	0100	4
1	0010	2
2	1001	9
3	1100	12
4	0110	5

Note: We represent the XOR gate with an addition sign, because we assume we are adding modulo 2. Thus, 0 + 0 = 0 and 1 + 1 = 2*mod*2 = 0, whereas a 0-1 combination results in 1.

5. Seven Segment Display

You may have to modify your seven segment display to properly show the outputs from your slots. One approach is this: your input might be a 3N-bit number, with N bits from each of your slots. You could change your digit calculations in the following way. Suppose your N bits from the left-most slot represent a number from 0 to 15. Extract the ones and tens place and set the two-left most digits on your FPGA accordingly. Repeat for the remaining digits! *Hint: Can you make use of division or modulo when extracting digits?*

IV. Checkoff

Ask Bryan and I before testing your slot machine on an FPGA! Once you've verified with us that your project works, we may ask you a few questions and then you'll be good to go!

If you would like to test your project on the auto-grader, make a new branch on the auto-grader repository, and upload your .sv files. From there, the actions tab should display a list of "workflow runs," where you can find the auto-grader results. For more detailed instructions, see the ReadMe.

Resources and Reference Material:

Lecture 1
Lecture 2
Pin Sheet
Verilog Docs, FAQ
Bryan and Adrian
Auto-grader