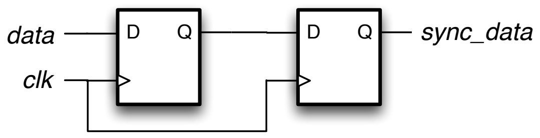
CE2003 Laboratory 1

Suhaib A. Fahmy

Debouncing and Synchronisation (approx. 20 mins)

When implementing synchronous digital systems that interact with external stimuli, we must ensure we synchronise the signals as they enter the system. This is usually done by passing the input into a register. Recall that if a register's input changes between during the setup and hold times, this can cause unpredictable behaviour, and a longer settling time for the output. This can be mitigated by adding a second register at the output of the first. This ensures that any adverse behaviour at the output of the first register is not passed through to the circuit. This is called a *synchronisation register chain*, as shown here:



If we want a single pulse from the external button or switch, we can implement a simple edge-detector. This adds another register to the output of the register chain. Now, when the third register is 0, and the second is 1, it means our signal has changed from 0 to 1, signifying a rising edge. Hence, we output a pulse.

Another problem encountered when interfacing with physical buttons and switches is bouncing: when a button is pressed or a switch toggled, the real electrical manifestation is not a perfect transition. Hence, we need to process external stimulus through a debouncer.

This consists of the synchronisation register chain mentioned above, followed by a saturating counter. This is a counter that counts up as long as the button is pressed, and resets whenever it is released. We set a maximum value for the counter, to prevent it from looping round, perhaps at a count equivalent to 50ms. We then create a pulse by using a simple assign statement to check for a specific number (less than the counter maximum), perhaps equivalent to 30ms. Hence, only continuous presses of 30ms will register as a press, producing a single-cycle high pulse. This means short oscillations are ignored, and long presses do not register as multiple transitions.

You have been provided with three files:

- `clkgen.v` takes the 100MHz board clock and uses it to drive an 11-bit counter. Bit 10 will toggle every time the counter counts up to 1024, then stay high for 1024 cycles, effectively dividing the clock by 2048, giving us an output clock signal with a frequency of around 40kHz. *Note that this isn't the most advisable way to slow down a clock, but it is simple, and sufficient enough for our purposes.*

- `edgedetect.v` implements a synchronisation register chain as described above, along with an additional register and some basic logic to detect a rising edge. This is ensures we get a pulse with each button press.

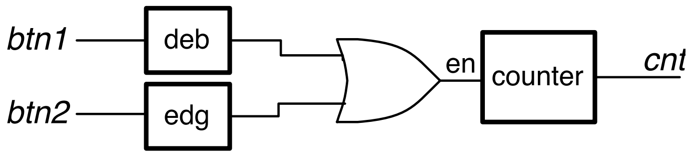
- `debounce.v` implements a debouncing circuit as described above. The output of the synchronisation chain drives a saturating counter (i.e. it does not count beyond a certain upper limit). At a value less than the count limit, we generate the single cycle pulse required.

Task 1

Compare the performance of the edge detector and debouncer circuits:

1. Start ISE and create a new project targeting the Spartan 6 XC6SLX45-CSG324-3.
2. Create a new module (Project, New Source) with two 1-bit button inputs, *btn1* and *btn2*, a clock and reset, and an 8-bit output, *cnt*.
3. Implement a standard enabled 8-bit counter for the *cnt* signal, it should use a clock named *iclk*, and and enable signal named *cnten*. Remember the reset on the Atlys board is active low, so use *!rst* in the always block.
4. Add the provided modules to the project (Project, Add Source)
5. Instantiate the clock divider from **clkgen.v**, and connect its output to a signal named *iclk*.
6. Instantiate the edge-detector circuit from **edgedetect.v**, and the debouncer circuit from **debounce.v**.
7. Connect one of the button inputs to the debouncer input, and the other to the edge-detector input.
8. OR the outputs of the debouncer and edge-detector to generate *cnten*.
9. Create a constraints file source (Project, New Source). Assign the *btn1* and *btn2* inputs to pins corresponding to buttons on the Atlys board. Assign the 8-bits of the count output to the 8 LEDs above the switches. *Remember, all switches, buttons, and LEDs have the pin names displayed in parentheses on the board. The clock pin is under a label* GCLK *on the bottom right (L15), similarly for the reset button.* Note the format for a UCF pin location constraint:

```
NET "count_out<0>" LOC = "U18" | IOSTANDARD = LVCMOS25;
```



Implement the design, then use Adept to load it onto the FPGA, and observe the difference between using a debounced switch and a basic edge-detector. You should see the edge-detected input cause the counter to skip more than one transition on occasion, while the debouncer and pulse generator will work as expected.

Vending Machine (approx. 90 mins)

You are to design a finite state machine controller for a vending machine. The vending machine accepts three coins, 10c, 20c, and 50c. A coin insertion is represented by a pulse input on the corresponding input. The state machine has a vend output that indicates a drink should be deposited, along with three change outputs for 10c, 20c, and 20c. The state machine should vend the drink and change as soon as a sufficient amount (50c) has been inserted and wait for a reset. The machine can refund the inserted amount if the *cancel* input is asserted in any intermediate state.

Task 1

Draw a state diagram that represents the above behaviour. You should use a Moore machine, where the outputs depend only on the current state. The most straightforward way is to think of the states as representing the amount inserted so far. The end states represent either vending a drink, vending a drink and change, or returning an amount of money. Two 20c change outputs are required to accommodate a refund of 40c or a vend and change from 90c. **We assume only one input is asserted at any point in time.** You should have 5 vending end states and 4 refund end states in addition to the intermediate states.

Task 2

Start a new project in Xilinx ISE. Implement the state machine in Verilog, using a button for each of the coin and cancel inputs, and five LEDs for the vend and change outputs. Note that when you have mutually exclusive inputs, it is better to use parallel ifs:

```
if (a) nst = stA;
if (b) nst = stB;
```

rather than nested ifs:

```
if (a) nst = stA;
else if (b) nst = stB;
```

because the priority in the nested version consumes extra logic.

Task 3

Now you will test the finite state machine on the board. First, create a new top-level module that instantiates your FSM design, and instantiates a debouncer circuit for all button inputs, as well as the *clkgen.v* module to generate the system clock. Create a user constraints file to connect the inputs and outputs to the board. Use three buttons on the board for the coin inputs, one button for the cancel input, then 1 LED for the vend output, and three more LEDs for the change outputs. Remember to wire the reset input to the reset button, which is **active low** and the clock to the correct pin.