Counters and Timers

Version 2: corrected calculation of initial timer values.

Introduction

In this lab you will explore the design of counters and timers by designing a simple kitchen timer.

Requirements

The timer has two pushbutton inputs, `reset_n` and `run_stop`. It has one 4-bit output, `seconds` that drives four LED's and a one-bit output, `alarm`, that drives one LED. It also has a clock input, `clk`.

Your timer should operate as follows:

- The timer can be in one of two states: `run`=0 and `run`=1.
- `run` is set to 0 when `reset_n` is asserted; it is set to the logical inverse of `run` (`not run`) when there is a rising edge on `run_stop`.
- `seconds`, the time remaining, is set to the initial timer value (see below) when `reset_n` is asserted; it is decremented by 1 once per second when `run`=1 and `seconds/=0` (that is, when the timer is running and there is still time remaining).
- The `alarm` LED is turned on if `run`=1 and the time remaining is zero.

The initial time remaining should be $8+(n \bmod 8)$ seconds where n is the last digit of your BCIT ID. For example, if your BCIT ID is A01456789 then $n = 9$ and your timer will count from $8 + (9 \bmod 8) = 8 + 1 = 9$ seconds down to 0.

Your design should be synchronous – all registers should use the same clock and not use asynchronous sets or clears. In particular, `reset_n` does not directly reset any registers, it is used to determine the value loaded into the register(s) on the next rising edge of the clock.

Pre-Lab

Read the Hints section below and write the VHDL source code for a kitchen timer that meets the above requirements.

It is strongly recommended that you check your VHDL description for syntax errors before the lab. If you don't have access to Quartus or Modelsim you can use the EDA Playground on-line HDL simulator[1].

If you've installed Quartus on your PC you can also follow the procedure below, complete the lab at home and bring your circuit to the lab to have it checked.

Procedure

You will need the same components as for the previous lab.

Follow the general procedure in Appendix A of the "State Machines" lab and Appendix A of the "Introduction to VHDL" lab to create a project, compile your VHDL description, and configure your CPLD.

Download and add the `sync_debounce.vhd` file from the course web site to your project. A `sync_debounce` entity should be instantiated in your design to synchronize and debounce the `run_stop` pushbutton (see below for details).

The CPLD should be connected to the input and output devices as in the previous lab. However:

- the pushbutton switch on pin 2 that previously simulated a clock will now be `run_stop`, a run/stop pushbutton control;
- the clock for your design, `clk`, will instead come from an on-board 50 MHz oscillator that is connected to pin 12 of the CPLD;
- the `reset_n` signal on pin 29 will reset the timer to the initial timer value computed above;

[1] Use the Aldec Rivera simulator and an empty testbench file; comment out the switch debouncer.

- the **led** bus (pins 44, 48, 50 and 52) will display the time remaining (**seconds**) and
- the on-board LED connected to pin 77 that displayed the debounced clock pushbutton will now be an **alarm** output indicating that the timer has expired.

Pin Assignments

Follow the instructions in the previous lab to configure internal pull-up resistors on pins 2 and 29, the **run_stop** and **reset_n** inputs.

The 50 MHz on-board clock, **clk**, is connected to CPLD pin 12 on your board.

Hints

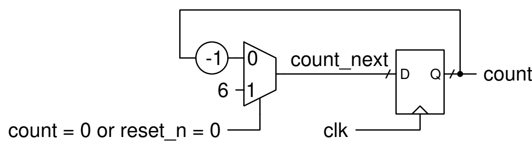
Modeling Registers

It is common to use one signal name for the input and one for output of a register. For example, if the output of a register is called **count**, the signal **count_next** would be its input:

```
signal count, count_next : unsigned(3 downto 0) ;
...
count_next <=
   to_unsigned(6,4) when count = 0 or
      reset_n = '0' else
   count-1 ;

count <= count_next when rising_edge(clk) ;
```

The first conditional assignment describes the combinational logic that computes the next value of the **count** register and the second one describes the rising-edge-triggered register. The circuit would be:

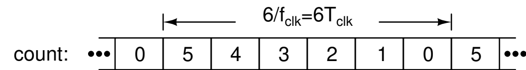


Timers

If we set a register to $N-1$ and decrement it by one every clock cycle it will reach zero[2] after N clock cycles which happens after a delay of N/f_{clk} seconds.

[2]Timers traditionally count down to zero because no additional hardware is required to determine the final value – the subtracter's borrow bit indicates when the count has reached zero.

If, as in the example above, the counter is re-initialized to $N-1$ when it reaches zero, the counter values will repeat periodically with frequency of f_{clk}/N. For example, if $N=6$:



By making other actions (e.g. decrementing the number of seconds remaining) conditional on specific counter values (e.g. zero) we can carry out these actions at different rates or delays. For example:

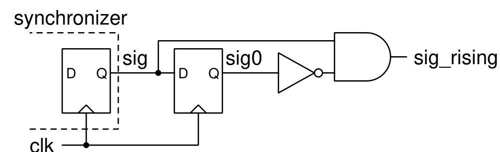
```
-- decrement timeleft at a rate f_clk/6
timeleft_next <=
   timeleft - 1 when count = 0 else
   timeleft ;

-- 50% duty cycle at f_clk/6 (for illustration)
signalhigh <= '1' when count < 3 else '0' ;
```

It is usually a bad idea to use a periodic signal generated this way as a clock (in the **rising_edge()** part of a conditional assignment). The reasons are explained in Appendix A.

Detecting Edges

To detect rising edges on a signal that is not a clock, such as a press of the **run_stop** pushbutton, we can store the previous value of the signal in a flip-flop and compare it to the current value. If the previous value was low and the current value is high then there must have been a change in level from low to high (i.e. a rising edge):



The VHDL code for this circuit would be:

```
sig0 <= sig when rising_edge(clk) ;

sig_rising <=
   '1' when sig0 = '0' and sig = '1' else
   '0' ;
```

Synchronizing Inputs

Since inputs such as those from pushbuttons are asynchronous to the clocks used in your circuit, there is no way to ensure that the setup time requirements of the various flip-flops in your design will be met.

We can minimize the likelihood of metastable events[3] by passing asynchronous inputs through a flip-flop called a synchronizer[4]. The output of this flip-flop will be synchronous with the clock and we can verify that the setup requirements of the other flip-flops in our design will be met.

For this lab you will be supplied with a synchronous debouncer which also provides the input synchronization function. This **sync_debounce** debouncer needs to be supplied with your circuit's clock as well as the switch input and output signals. It can be instantiated in your design as described below.

Entity Instantiation

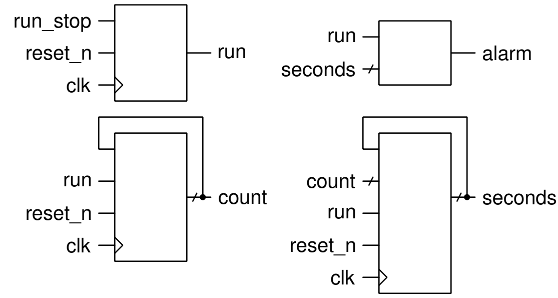
The **sync_debounce** is provided in a file as an entity/architecture pair rather than as a component in a package. The syntax to directly instantiate an entity instead of a component is slightly different. For example, the following code instantiates the **sync_debounce** entity:

```
debounce1: entity work.sync_debounce
   port map ( run_stop_in, clk, run_stop ) ;
```

The **work.** prefix on the entity name indicates that the entity is found in the **work** library along with the other design units currently being compiled.

Block Diagram

From the functional description of the timer, we will need a timer with a period of one second to determine when to decrement the time remaining, a counter for the number of seconds remaining, and a flip-flop to store the **run** state. The **alarm** output can be generated with combinational logic. Based on the functional description we can determine the inputs and outputs of each functional block. The result might be:



Each of these blocks will correspond to one or two VHDL statements.

Coding Style

As a minimum, each source file must include near the start of the file a comment that includes: the file name, a line describing the purpose of the file, the author's name and the date.

Additional comments next to port and signal declarations and for non-obvious portions of your design are also a good idea.

Comments should explain why you're doing something rather than repeating what is obvious from the code. Here are some examples of what the author considers good and bad comments:

```
-- open door if power off and pressure low (good)
-- set door to the NOR of pwr and hi_p (bad)

door <= not pwr and not hi_p ;
```

It should be possible to figure out how your design works by reading only the comments.

Troubleshooting

Follow the usual troubleshooting strategy if your circuit isn't working. First check power, ground and clocks. Then check for a discrepancy between the measured and expected signals. The testing should proceed in order from the inputs to the outputs.

Connect signals (e.g. the pushbutton inputs, the **run** state flip-flop, and groups of bits of the count values) to the LED displays to verify that each section is operating as expected. Signals that change too quickly to be seen can be measured with the 'scope.

[3]Meaning that flip-flop outputs do not settle by their specified t_{CO}.

[4]Synchronizers typically use two flip-flops in series to make them more robust.

Demonstration and Marking

The preparation mark will be based on coming to the lab with reasonably-complete VHDL code.

The lab instructor will determine the demonstration part of the lab mark by first checking that your circuit:

- displays the correct value and does not start counting down when reset,
- counts down when **run_stop** is pushed, stops when pushed again, restarts when pushed a third time,
- resets the time remaining and stops when **reset_n** is pushed,
- turns on the **alarm** led when the count reaches zero, and
- **alarm** turns off when **run_stop** is pushed.

Then the lab instructor will ask you to make a minor change to the behaviour of your design to make sure you understand how it works. For example, the instructor could ask you to modify the initial timer value, the rate at which the timer operates, the operation of the switches (active-high reset, falling-edge run/stop control), etc.

Optional Extensions

If you found this lab too easy, you can try adding[5] the following features (listed in order of increasing complexity):

- Have the time remaining display "blink" while the timer is running.
- Define additional outputs to display the time remaining on a 7-segment LED display as described in the previous lab.

A Multiple Clocks

It is usually not a good idea to use signals generated by logic, for example the different bits of a counter, as clocks. Among other issues, clock signals generated by logic circuits will have more uncertainty in their timing which in turn will reduce the speed at which your design can operate.

However, power consumption is sometimes more important than speed and reducing the clock rate reduces power consumption because power consumption is linearly related to the clock rate. In this application for example, a clock rate as low as 100 Hz would be sufficient because users would not notice the resulting 10 ms lag in response to button presses. Thus, if our timer were battery powered we might want to divide the clock to 100 Hz and use that as the sole clock in the design[6]. However, you need not do that for this lab.

The use of multiple clocks that are not derived from the same clock (i.e. generated by different oscillators) poses different problems and special techniques are needed to cross "clock domains."

[5] You must leave the required functionality intact.

[6] Watches and battery-powered timers typically use an oscillator operating at $2^{15} = 32768$ Hz.