Project #2: Timer with millisecond precision

Revision: March 16, 2012

www.digilentchina.com
25A, No.1369, Dongfang Road, Shanghai
Tel: 021-58893151 Fax: (86)21-58894598

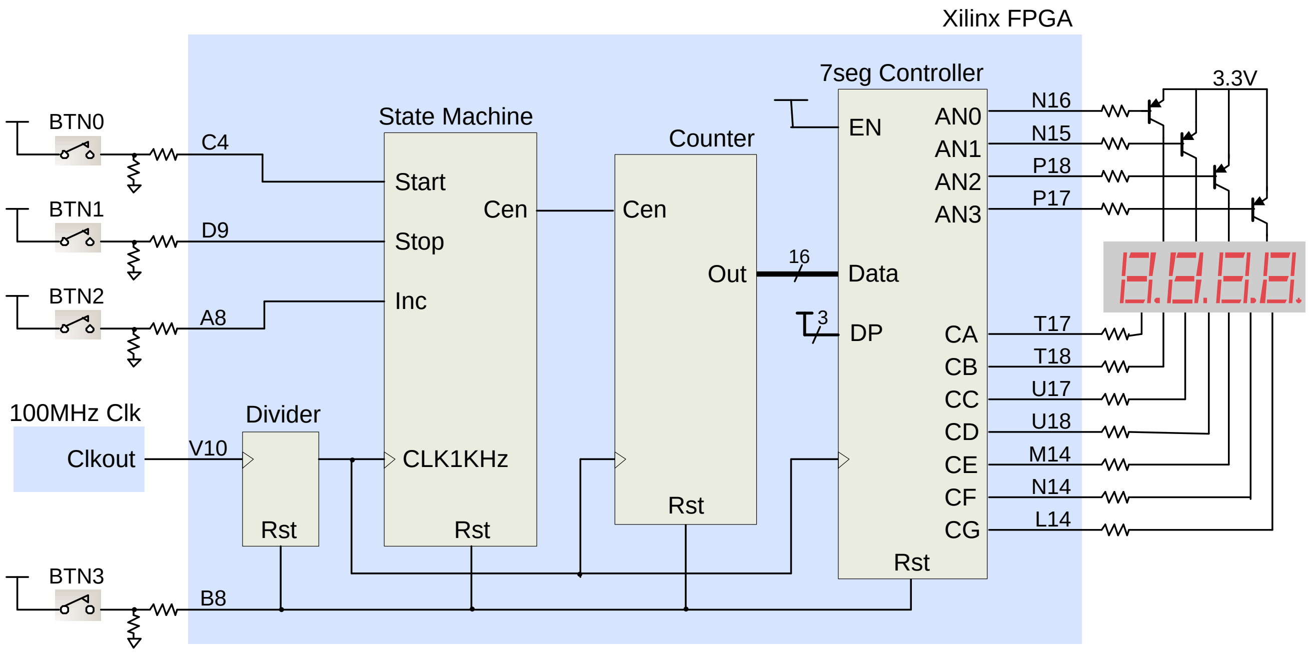
Overview

This project uses the Nexys3 board to make a timer circuit that can measure millisecond intervals. A simple state machine provides start, stop, and increment features, and the buttons, LEDs, and seven-segment display will be used for circuit inputs and outputs.

Design requirements

Using the Xilinx Verilog tools, build the timer circuit on the Nexys3 board. A complete block diagram, including pin definitions, is shown below. No requirement is placed on the Verilog source files – you may use any coding style you wish.

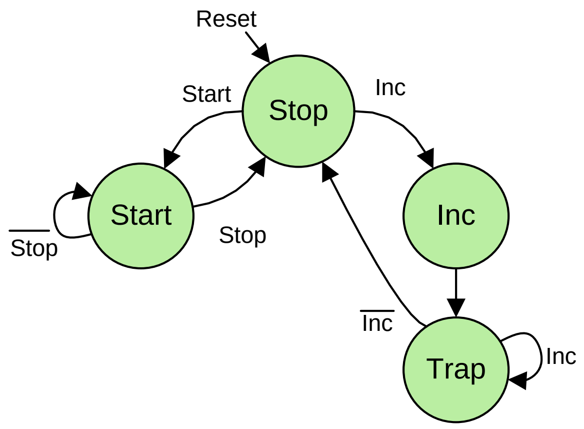
The timer must count from 0.000 to 9.999 seconds and then roll over back to 0.000, with the least significant digit updating once per millisecond. The timer output should be displayed on the seven-segment display, with a decimal point between the most significant digit (digit 3) and the next digit (digit 2).Three pushbuttons will be used to provide start, stop, and increment functions. Once the start button is pressed, the timer should increment at 1KHz until stopped. Pressing the stop button should top the counter and display the current timer state. The increment function should cause the timer to increment by 1ms for each button press.



Discussion

Note that not all behaviors are fully specified in the design requirements section above. For example, if the timer is currently running and the increment button is pressed, no requirement is given to specify how the timer should behave. As an engineer, you must understand the design intent, and create behaviors for underspecified behaviors that are consistent with your understanding of the intent. In practice, it is difficult (or even impossible) to completely specify every behavior in a written document, so engineers frequently invent behaviors where specifications are lacking. Figures providing greater detail for the blocks shown above are shown below.

State Machine



A possible state diagram for a timer control state machine is shown, but this is just an example. The sequential behavior captured in the state diagram might change depending on how and individual engineer interprets the design requirements. In practice, an engineer might sketch several candidate state diagrams before settling on one to implement.

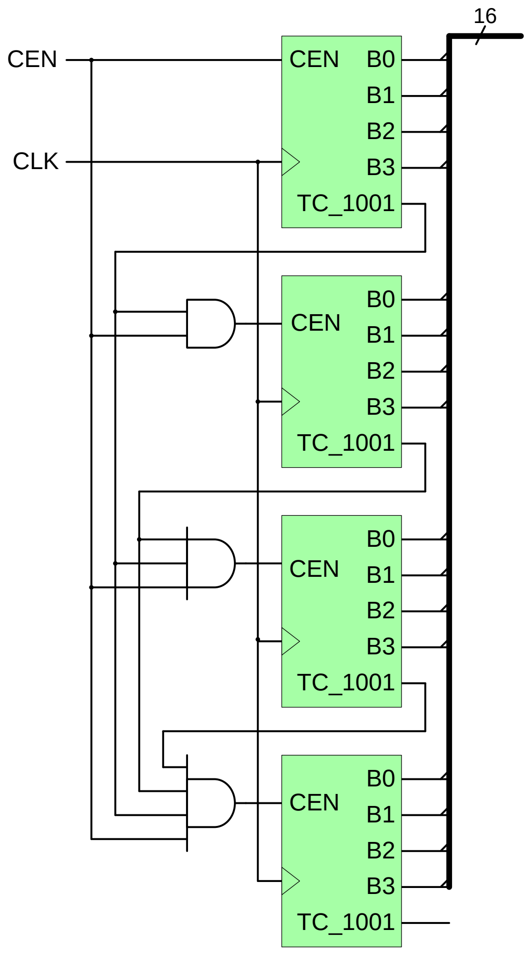
Counter

The timer requires a four-digit decimal counter. Such a counter can be built from four individual counters that each count between 0 and 9 (0000 to 1001), and then roll over from 9 back to 0, creating a continuous count pattern with 10 binary numbers.

Individual counters are based on 4-bit binary counters that each detect when the pattern “1001” is present on their outputs. When “1001” is detected, the counters reset themselves back to “0000” and issue a terminal count (“TC”) signal to indicate the count range is complete and counter is resetting back to zero.

Each counter also uses an enable signal (“CEN”). When CEN is asserted, the counter will increment with each clock edge, but when it is deasserted, the counter will ignore the clock and hold at its present state. The CEN signal can be used to enable more significant counters each time a less significant counter completes its count range.

The TC signal is typically driven by logic gate combining the count bits according to some function (in our case, TC_1001 is generated by a 4-input AND gate detecting “1001”). TC_1001 will be asserted for as long the “1001” pattern is present on the counter outputs. In the least significant counter, this is a single clock period. But in the next most significant counter (representing the 10’s position in a four digit decimal number), the “1001” pattern will be present for 10 clock periods, and so its TC_1001 signal will be asserted for 10 system clock



4-digit decimal counter

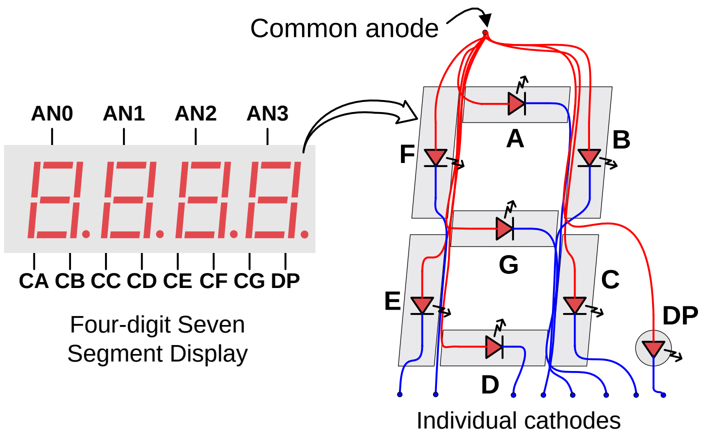
periods. To keep the next most significant counter from counting 10 times (during the period TC_1001 is asserted), the TC_1001 signal from the first counter must be combined with the TC_1001 signal from the next most significant counter. In fact, a little thinking about the problem will reveal that each TC_1001 from more significant counters must be combined with all such signals from all less significant counters.

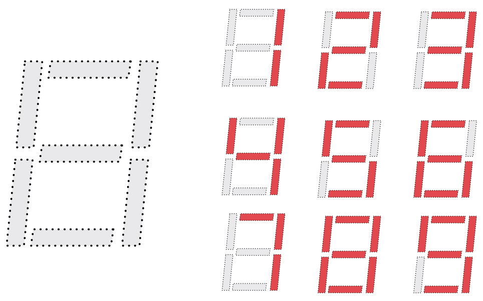
The outputs of the four individual counters can be assembled into a single 16 bit bus for transport to the seven-segment display controller.

Seven segment controller

The Nexys3 board contains a four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments arranged in a "figure 8" pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.

The anodes of the seven LEDs forming each digit are tied together into one "common anode" circuit node, but the LED cathodes remain separate. The common anode signals are available as four "digit enable" input signals to the 4-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG (so, for example, the four "D" cathodes from the four digits are grouped together into a single circuit node called "CD"). These seven cathode signals are available as inputs to the 4-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.




An un-illuminated seven-segment display, and nine illumination patterns corresponding to decimal digits

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession, at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-quarter of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update or "refresh" rate is slowed to around 45 hertz, most people will begin to see the display flicker.

In order for each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of 1KHz to 60Hz. For example, in a 60Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for ¼ of the refresh cycle, or 4ms. The controller must drive the cathodes with the correct pattern when the corresponding anode signal is driven. To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a “1” will be displayed in digit position 1. Then, if AN1 is asserted while CA, CB and CC are asserted, then a “7” will be displayed in digit position 2. If AN0 and CB, CC are driven for 4ms, and then A1 and CA, CB, CC are driven for 4ms in an endless succession, the display will show “17” in the first two digits. An example timing diagram for a four-digit controller is provided.

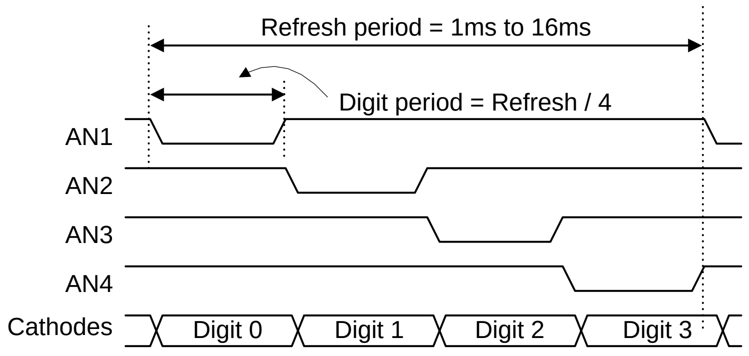
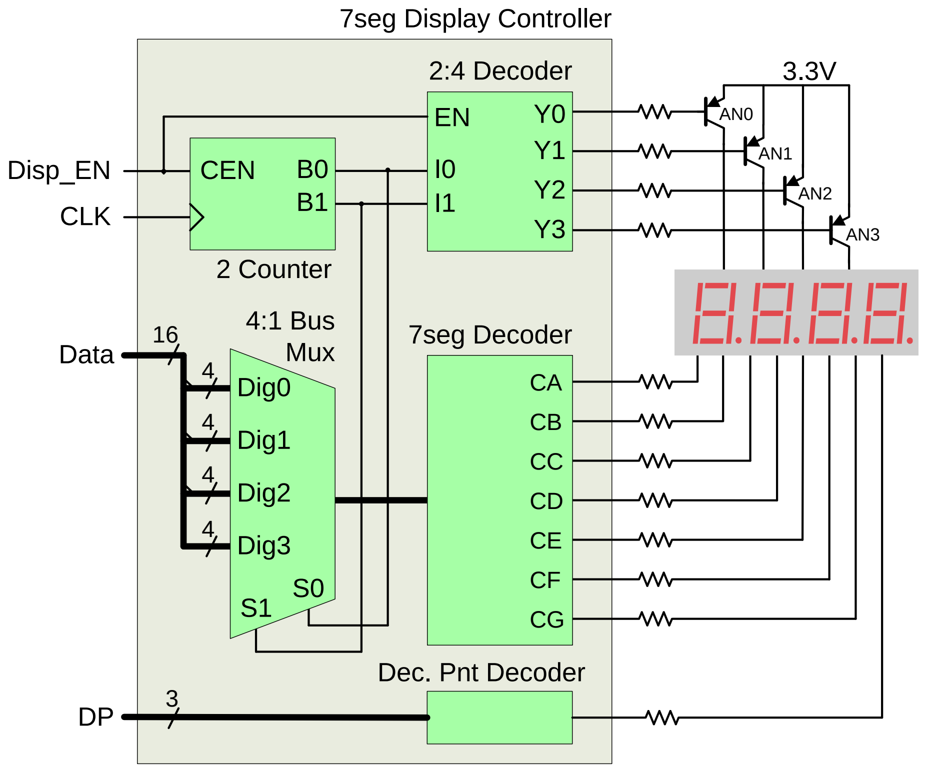


Figure 10: Seven-segment display timing diagram

Since a seven segment display controller is potentially useful in many different designs, it makes sense to design the controller as a reusable component. When designing a component that has the potential to be reused in different designs, it is a good idea to identify an interface and feature set that not only meets the needs of the current design, but that is as generic as possible so that other designs can more easily include it. An effective strategy is to identify different possible use scenarios, and then use those scenarios as motivation to identify useful features that will support as many designs as possible.

In this case, we have already noted that a refresh frequency in the 60HZ to 1KHz range is acceptable, so it follows that the 7seg controller interface should accommodate any clock frequency in that range. This single input clock signal can then be used inside the controller as a basis for refreshing the display elements. In the current application, we are only displaying 10 of the possible 128 different characters (the 10 decimal numbers), but there are more characters that we could (and probably should) support. In particular, adding the characters A, b, C, d, E, and F will allow hex digits to be displayed, in addition to decimal digits. If we restrict the controller to displaying any one of the 16 hex digits in each of the four display positions, we can use a four-bit input bus to define the character to be displayed at each digit. Thus, our controller requires four 4-bit input buses (here shown grouped together as one 16-bit bus). Since the decimal point is not really part of the numerical data from the counters, a separate data port is provided to drive the decimal point as needed. The three bit input to the decimal point decoder includes an “on/off” bit (the most



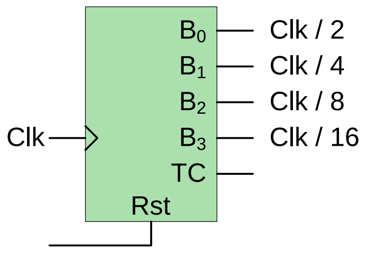
significant bit) followed by a two bit code to indicate which decimal point should be driven.

Finally, many circuits include an enable (or on/off control) so that circuit blocks can be turned off for power conservation or noise suppression reasons; our controller should include an enable signal.

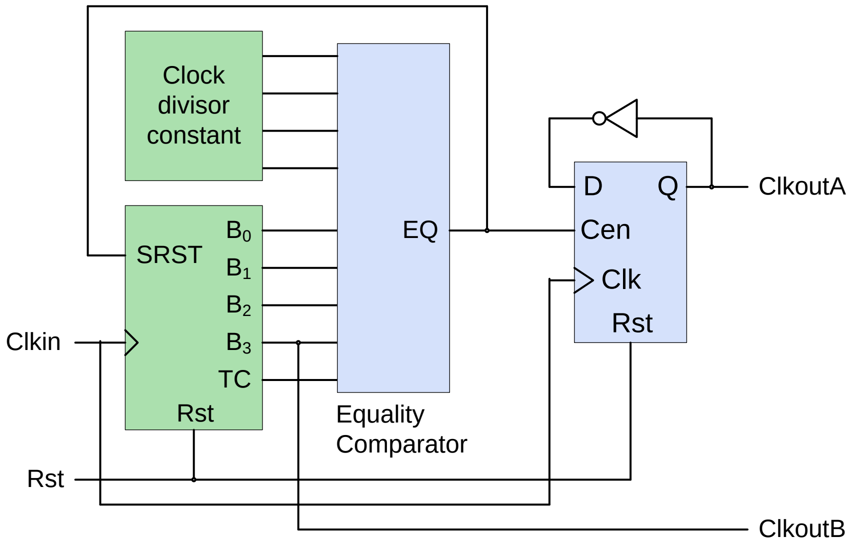
The block diagram shows a controller that accommodates these requirements. Based on this discussion, a structural Verilog design is indicated so that the component can be reused in later designs.

Clock divider

Clock dividers based on binary counters are commonly used in digital systems. In a simple "power of 2" divider, a higher frequency clock signal drives a counter's clock input, and the counter outputs provide lower frequency signals at $1/2^n$ of the input frequency, where n is the counter output bit number (assuming bit #1 is the LSB). This simple divider works well for generating frequencies that are power-of-two divisors of the input frequency. To create divider frequencies that are any integer divisor of the input frequency, an equality comparator can be used to compare the count value to a divisor. If a clock with frequency 1/N is required, then a divisor of N/2 can drive one side of the comparator (with the counter driving the other side). The output of the comparator can be used as a synchronous reset to restart the counter from '0' (at twice the desired frequency), and also as a clock-enable for flip-flop that has its output tied to its input through an inverter (CkloutA in the figure). The output of this flip-flop will produce the desired frequency with a 50% duty cycle (duty cycle is the fraction of time a signal spends at '1'; a 50% duty cycle means the signal is '1' half the time and '0' half the time) . Note that a simpler circuit can produce a clock frequency of 1/N if a 50% duty cycle is not required (and in most applications, duty cycle is not important). This simpler circuit resets the counter when it reaches N (instead of N/2 as above), and then uses the MSB of the counter as the output clock. This signal will have the desired frequency, but it will not have a 50% duty cycle.



Simple clock divider



Clock divider for any integer divisor

Behavioral Verilog code for a clock divider that divides a 100MHz clock to a 1Hz clock is shown below. In the example code, note that a constant has been used to define the divider ratio; this constant can be changed to set any desired divider ratio. Note also the MSB of the counter is used as clkout, resulting in a clock signal with the correct frequency that does not have a 50% duty cycle.

```
module clk_div (
   clk_in,
   rst,
   clk_50MHz);


input   clk_in;
input   rst;
output   clk_50MHz;

wire    clk_50MHz;
reg     [3:0] cnt;


always @(posedge clk_in)
   begin : process_1
   if (rst == 1'b 1)
      begin
      cnt <= {4{1'b 0}};
      end
   else
      begin
      cnt <= cnt + 1;
      end
   end

assign clk_50MHz = cnt[0]; // 100Mhz divided by 2 to get 50Mhz

endmodule // module clk_div
```