Learning Objectives:

- Additional practice doing logic simulation
- Practice dataflow modeling
- Practice behavioral modeling

Homework 2

Assignment: Verilog Dataflow and Behavioral modeling

The purpose of this assignment is to provide experience in coding complex logic in RTL (Dataflow and Behavioral modeling). The project consists of implementing several new modules and integrating them with existing modules, resulting in a two-digit Digital Scoreboard.

This assignment will be graded.

Functional Specification

THE CONCEPT FOR THIS PROJECT CAME FROM "DIGITAL SYSTEMS DESIGN IN VERILOG" BY CHARLES ROTH, LIZY KURIEN JOHN, AND BYEONG KIL LEE, CENGAGE LEARNING, 2016. THE ORGANIZATION OF THE PROJECT AND THE SAMPLE CODE, HOWEVER, ARE MINE.

In this project we will design a simple scoreboard capable of displaying scores from 0 - 99. Input to the scoreboard is from three buttons:

- Increment Score – increases the score by 1 if the score is < 99. Holds at 99 if already at 99.
- Decrement Score – decreases the score by 1 if the score is > 0. Holds at 00 if already at 00.
- Clear Score – sets the score back to 00. The button must be pressed 5 times to clear the score. This prevents the score from accidentally being cleared while a game is in progress

Output from the scoreboard is to two 7-segment display digits. A 7-segment display digit can display numbers from 0 – 9 and A-F (for Hex) and many other combinations that may or may not be useful. For the scoreboard, each digit will display 0 – 9, or taken together 00 – 99. To do this we need to keep score by implementing a two digit BCD counter instead of a simple binary counter.

BCD Counting

In computing and electronic systems, **binary-coded decimal** (**BCD**) is a binary encoding of decimal numbers where each decimal digit is represented by a fixed number of bits, usually four. BCD's main virtue is ease of conversion between machine-readable and human-readable formats such as a 7-segment display. In Homework #1 we built a 4-bit BCD adder. The adder we implemented could easily be extended to 8-bits or more by chaining the carry-out from one stage to the carry-in of the next stage. It seems that repurposing Homework #1 to this application would be easy (add 1 or 0), but alas, it is not that simple. We also need to subtract 1 when we are decrementing the count and we would need to hold the results within the range 00-99.

While it is certainly possible to build a BCD Adder/Subtractor (and you can find Verilog code online that does this) we are going to take a different approach. We will build a counter that counts in BCD instead of binary. That means when the least significant digit is at 9 and we increment the count, we set that digit back to 0 and add one to the most significant digit if it is less than 9. If both the least and most significant digit are 99 (e.g. count is 99) we hold. Similarly, when the least significant digit is 0 and we decrement the count, we set that digit back to 9 and subtract 1 from the most significant digit…that is unless it is also 0 in which case we hold at 00.

Hardware

There is no hardware required for this project. For this assignment you will have to limit your satisfaction to watching your skillfully crafted Verilog model simulate correctly. While we will only be simulating the design in this project it would be but a small step to take this design to an FPGA development platform…but that is beyond the scope of the workshop.

Implementation Architecture

This design consists of several Verilog modules as summarized below:

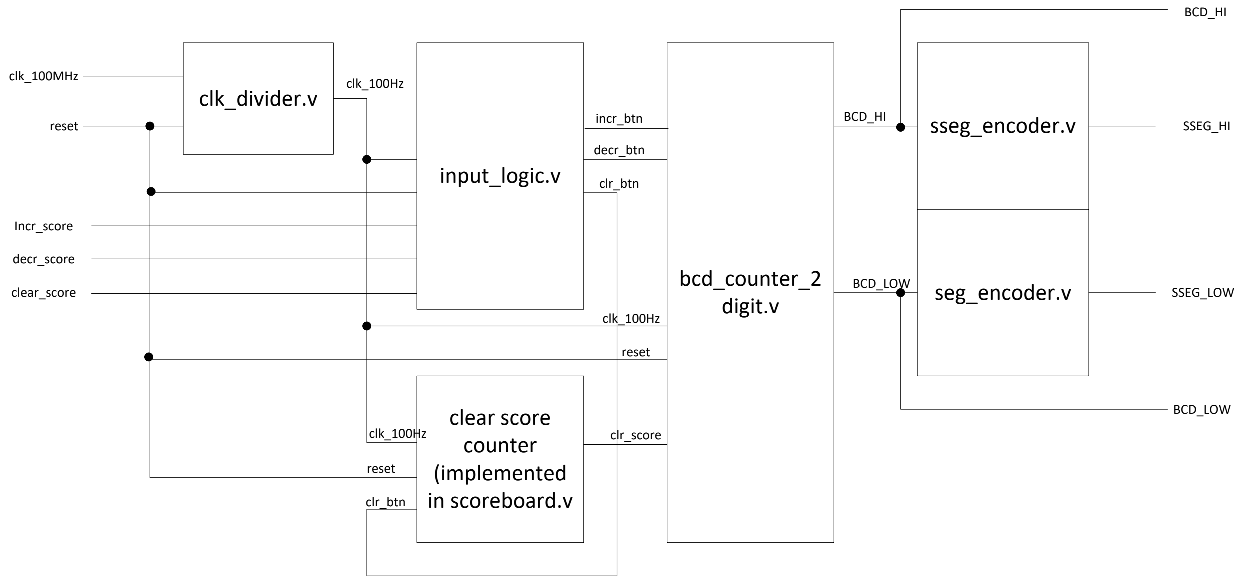


Figure 1 - Block Diagram

Clock Divider (clk_divider.v)

The clock divider block can be used to generate a slower clock from a faster clock. It is fundamentally a counter that counts up at the input clock rate and generates a 1 cycle pulse every time the counter matches a “top count”. The top count is calculated by dividing the input frequency by the output frequency.

In this design the clock divider generates a 100 Hz clock from a 100 MHz input clock. The scoreboard logic is driven with this 100 Hz clock. Why do we need a slower clock? The slower clock is necessary to properly condition the pushbutton inputs so that a single button press only increments or decrements the BCD counter by 1. More on this in the next section.

The clock divider is provided as IP (Intellectual Property) to you for this project. It is implemented as a parameterized module where the input and output clock frequencies can be set.

Input Conditioning Logic (input_logic.v)

The digital scoreboard receives its input from mechanical pushbuttons that are external to the design. These pushbutton inputs change asynchronously to the system clock (this is called crossing clock domain - a topic we deal with in both ECE 540 and ECE 581) and need to be synchronized with the clock in the digital system they are driving. Mechanical pushbuttons have an additional problem called “switch bounce.” When a mechanical pushbutton is pressed and/or released, the output of the pushbutton will “bounce” causing multiple edges which could be incorrectly applied to the BCD counter causing it to increment or decrement by more than 1. The switch bounce could last several milliseconds before the output settles to a steady value. The 100Hz clock provided by the Clock Divider is slow enough that the switch value will settle before being used.

The Input Conditioning Logic performs two operations. First each of the pushbuttons is synchronized to the 100Hz clock. This is done with the `btn_sync` and `btn_d` flip-flops. Second, the pushbutton input which will likely (even at 100Hz) last more than one clock is conditioned to provide only a single pulse each time the button is pressed. This is done with the `btn_dd` flip-flop and the AND gates implemented in Dataflow format (`assign`).

The Input Conditioning logic is provided as IP (Intellectual Property) to you for this project.

Clear Score Counter (implemented in Scoreboard.v)

This is a binary counter that counts `clr_btn` presses. The purpose of the Clear Score counter is to prevent the scorekeeper from accidentally clearing the score while a game is in progress. The scorekeeper must press the `clr_btn` at least five (5) times to clear the score. A binary counter keeps track of the times the `clr_btn` is pressed. Once the count reaches 5, the Clear Score Counter should generate a single cycle pulse called `clear_score`. The `clear_score` signal is sent to the BCD counter which sets the count back to 00 when `clear_score` is asserted.

The Clear Score Counter is implemented as a clocked procedural block in *Scoreboard.v*. The structure of the Clear Counter as I've implemented it in the starter code, is similar to the more complex structure that you will use to implement the BCD Counter. More specifically, it is a common design practice for synthesized designs to structure sequential logic (registers, counters, etc.) as follows:

```
reg [4:0] regA;

always @(posedge system_wide_clock) begin
   if (system_wide_reset) begin
      // initialize the register or counter
      regA <= 5’b00000;
   end
   else if (register_specific_reset) begin
      // initialize the register or counter
      regA <= 5’b01010;
   end
   else if (register_specific_enable) begin
      // update the contents of the register
      regA <= new_regA;
   end
   else begin
      // retain the value
      regA <= regA;
   end
end // always block
```

In this design approach all of the sequential logic blocks are clocked with a system clock that (in an FPGA or ASIC) is distributed on dedicated routes. The register or counter, or … contents are updated with an enable signal that is generated elsewhere (either an `assign` or a different `always` block). The `else…begin` at the end of the block retains the old value (can avoid unwanted implied latches). Note the use of non-blocking assigns in clocked (sequential) logic. Much of this is beyond the scope of the workshop but you may as well start developing good design techniques from the get-go.

BCD Counter (bcd_counter_2digit.v)

The BCD Counter is a two-digit counter that counts from 00 – 99 (in BCD not binary). The counter is incremented by 1 when it receives a pulse on the `incr_btn` signal and decremented by 1 when it receives a pulse on the `decr_btn` signal. Both of those signals are generated in the Input Conditioning Logic. The counter should hold when it reaches its maximum count of 99 or its minimum count of 00. Each digit counts from 0 – 9 so a typical increment sequence would be something like:

00 -> 01 -> 02 -> 03 -> 04 -> 05 -> 06 -> 07 -> 08 -> 09 -> 10 > 11 -> …-> 98 -> 99 -> 99 -> 99 …

And a typical decrement sequence would be something like:

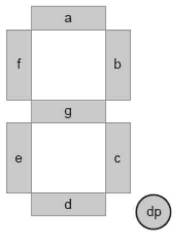
99 -> 98 -> …-> 11 -> 10 -> 09 -> 08 -> 07 -> 06 -> 05 -> 04 -> 03 -> 02 -> 01 -> 00 -> 00 -> 00 …

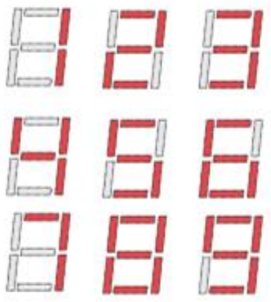
The count should be set to 00 whenever the reset signal is asserted and whenever the `clr_score` input is asserted. It would be wise to take no action (e.g. do not change the count) when none of the buttons are pressed or when the `incr_btn` and `decr_btn` are pressed at the same time.

You will implement this module from scratch.

Seven Segment Digit Encoder (sseg_encoder.v)

The Seven Segment Digit Encoder translates a 4-bit BCD number to control signals that light the individual LED segments in the display. Lighting a segment can be done by driving either a 1 or 0 to that segment depending on how the segments are connected to the driving signals. It is necessary to read the schematic to understand whether lighting a segment is done by driving a 1 or a 0 to the segment. A 7-segment display digit typically has the following configuration:





As you can see from the figure, the digits are formed by selectively lighting the segments on the display. For example, the digit `1` is formed by lighting segments `b` end `c.` The digit `8` is formed by lighting all of the segments and so on.

The following table can be used to convert the BCD digits 0 – 9 to the equivalent 7-segment digit encoding:

segment	gfedcba
0	0111111
1	0000110
2	1011011
3	1001111
4	1100110
5	1101101
6	1111100
7	0000111
8	1111111
9	1100111
10-15	1111001*

*Displays upper case E (Error)

The Seven Segment Display Digit encoder is a combinational circuit (no clock, no saved state) that you will implement as a parameterized module.

Design Software

ModelSim PE Student Edition or QuestaSim or another logic simulator.

Project Tasks

NOTE: THIS ASSIGNMENT ASSUMES THAT YOU HAVE GAINED EXPERIENCE WITH LOGIC SIMULATION AND HAVE SUCCESSFULLY COMPLETED THE BCD ADDER ASSIGNMENT.

STEP 1: Download the Homework #2 release package

Download and unzip the Homework #2 .zip file from the course website. This .zip file contains this document and an hdl directory containing a number of modules. Don't panic at the number of files and the amount of documentation. Remember Dr Hall's 5 minute rule. It's OK to panic for 5 minutes but after that, calm down, figure out what information you need and what information you have, formulate a plan and dig in.

STEP 2: Study the Verilog code provided in the release

The majority of the Verilog needed to implement this project is provided in full source code or in a "fill-in-the-code" type file. Take time to understand what the code does and how it is structured. Many of the constructs that you will need to complete the project are modeled in the existing source code. For example, you will implement the Seven Segment Digit Encoder as a parameterized model…the clock divider module which is provided in full source code is also a parameterized model. Similarly, you will implement the BCD counter based on the algorithm described earlier in this document. The Clear Score counter in *Scoreboard.v* is also a counter…albeit a simpler counter but the way the code is structured in that counter is similar to the way you can structure the BCD counter. If you have questions post them to the Discussion forum on D2L or work them out with your colleagues.

You can collaborate on the design, but the work you submit must be code you developed and debugged. The type of Verilog used in this design is fair game for the final exam which you will be taking by yourself.

STEP 3: Implement the Seven Segment Display Encoder

This is a simple module to implement. It expects a 4-bit BCD digit (range 0..9) and outputs the drive signals for each of the 7 segments in a display digit; essentially you are implementing a truth table in Verilog. The module should be implemented as a parameterized module that can reused whether the segments are lit with a 1 or a 0. A single parameter called SEG_POLARITY should be implemented in your module. When SEG_POLARITY = 1 the function will light any segment with a 1. When SET_POLARITY = 0 the function lights a segment by driving it with a 0. Make sure you set the default value of SET_POLARITY to match the way you encoded the segments (that is, drive high or low to light)

There are several ways to implement this circuitry (as a ROM using a Verilog `array`, as a Verilog `case` statement, as a nested `if..else` block, and so on. The choice is yours.

STEP 4: Implement the two digit BCD Counter

This is the heart of the project. You can model the structure of this counter in a similar manner as the clear score counter. Algorithmically, counting in BCD can be expressed as follows (C-like syntax):

INCREMENT COUNT

```
// bcd0 is least significant digit, bcd1 is most significant
if (bcd0 < 9)  // low order digit is in range so increment it
{
   bcd0 = bcd0 + 1;
   bcd1 = bcd1;
} // low order digit is in range
else // low order digit rollover, increment high order digit
{
   if (bcd1 < 9)  // high order digit is in range so increment it
   {
      bcd0 = 0;
      bcd1 = bcd1 + 1;
   } // high order digit is in range
   else // score is 99 so hold
   {
      bcd0 = bcd0;
      bcd1 = bcd1;
   } // score is 99
} // low order digit rollover
```

DECREMENT COUNT

```
// bcd0 is least significant digit, bcd1 is most significant
if (bcd0 > 0) // low order digit is in range so decrement it
{
   bcd0 = bcd0 - 1;
   bcd1 = bcd1;
} // low order digit is in range
else // low order digit underflow, decrement high order digit
{
   if (bcd1 > 0) // high order digit is range so decrement it
   {
      bcd0 = 9;
      bcd1 = bcd1 - 1;
   } // high order digit is in range
   else // score is 00 so hold
   {
      bcd0 = bcd0;
      bcd1 = bcd1;
   } // score is 99
} // low order digit rollover
```

STEP 5: Integrate your modules into the top level module (Scoreboard.v)

The homework release package includes a top-level model called *Scoreboard.v*. The module instantiates the other modules (including the modules you right) and implements the clear counter logic. Study this code and, if possible, match your Verilog module signature (the ports) with those in the template. If you can't match the ports edit *Scoreboard.v* to match your design.

STEP 6: Simulate your design

Now that you have completed your design you're 60% of the way done (Roy's rule ☺). Use the test bench provided in the release to debug and simulate your code. As with the other simulation projects you would start by building a ModelSim project, compiling all of the files and make the corrections until you get a clean compile. Then start the simulation (in ModelSim you would double click on `work/Scoreboard_tb`). I have included a sim/*wave.do* file to load into the Waveform window if you would like. You may have to change some of the signal names.

Study the test bench code because it is a pretty good way to write test benches that explicitly apply test vectors to the UUT (unit under test). I have implemented some simple Verilog tasks to simulate button presses. The test bench varies the number of cycles between button presses, and, in general, should provide pretty good coverage. Feel free to add additional test vectors for cases I may have missed.

References

[1] *Digital Systems Design Using Verilog*. Written by Charles Roth Jr., Lizy Kurian John, and Byeong Kil Lee. Cengage Learning, 2016. Page 226 – 232.

[2] Wikipedia.org. Article on binary coded decimal (http://en.wikipedia.org/wiki/Binary_coded_decimal