Lab 3 Car Detector and Counter Circuit
CSE 311 Advanced Digital Design
Aaron Chamberlain, ID: 003392908

I Objective:

The purpose of the following lab is to gain experience in meaningful FSM design, and then the subsequent implementation of the design in hardware. The circuit that I have been asked to design is a circuit that makes use of two sensors that detect a car entering or exiting a parking lot, with a counter that will inform people if the parking complex is full or not. The two sensors are placed in order the direction the cars will enter or exit the complex, with the distance being such that even a small car will be able to cover both at a given moment, while a human will not be able to cover both. As will be explained further in the design process, a car is said to have entered the parking complex when it passes through the first sensor, blocks both sensors, then blocks only the second sensor. This is what would define a forward motion. When the car has finally passed the second sensor, then the car count can be increased because the car has completely entered the parking complex.

II Design:

To begin the design process, we first need to define the method by which cars are detected as entering or exiting. Given the setup listed above, where even the smallest of compact cars will cover both sensors at a given time, we can determine whether a car is entering or exiting completely based off of which sensors are fired first. For example, we will define a car as having entered when it moves forward and blocks the first sensor, blocks both sensors, blocks just the second sensor, and then blocks none of the sensors (see Figure 1.) This process is then reversed to determine if a car is exiting the parking complex. When a car passes in front of the second sensor, blocks both sensors, blocks the first sensor, and then blocks none, that will decrease the number of cars in the parking structure due to it leaving the complex.

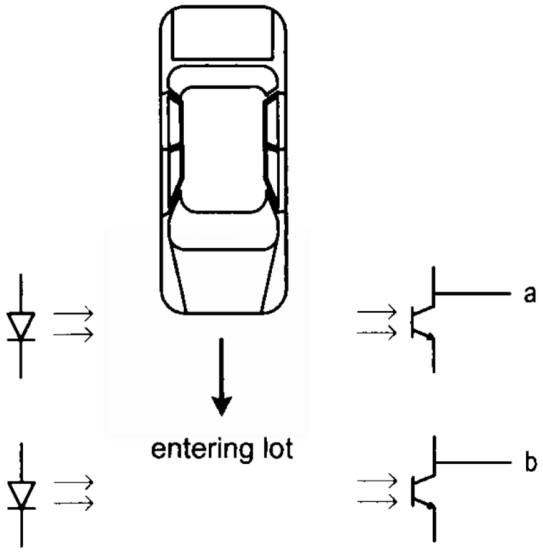


Figure 2. The layout of the two parking lot sensors. A car is shown entering and passing sensors.

With this simple definition, we can begin to design the Finite State Machine, or FSM that will dictate how the circuit will behave based off of the signals from both sensors. It should be noted that a mechanical, real world change has been made to the requested setup to ease the simplicity of design and reliability of the circuit. The change I would recommend for implementation is to place single direction spikes that inhibit the ability to reverse one passed over. This means that in order to exit, the driver must proceed fully into the parking lot, then get into the exiting lane and fully exit the lot just as any other car. In a real-world deployment of this technology, it would almost certainly lower the cost of deployment while also improving security and safety. With this in mind, we can begin to observe the FSM as shown in Figure 2.

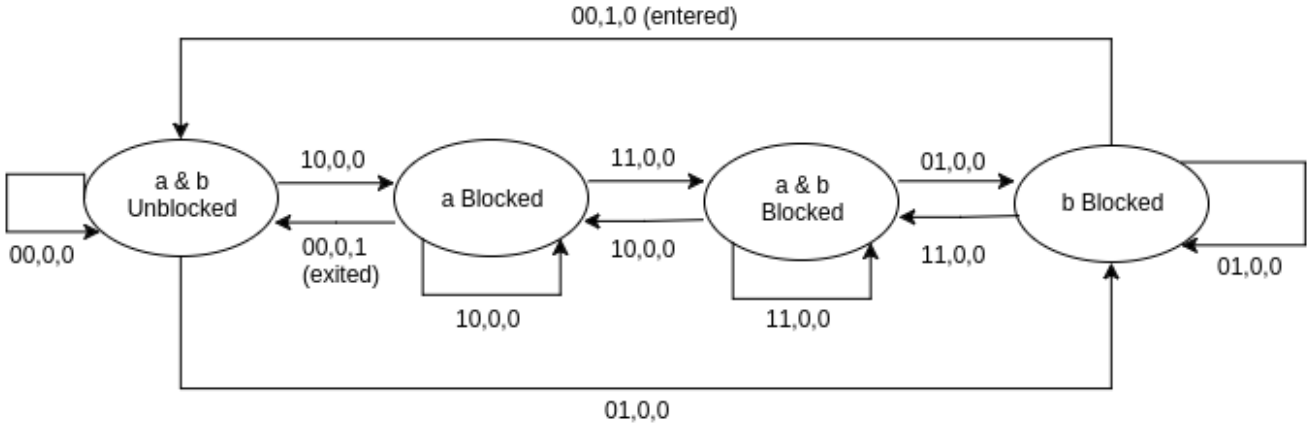


Figure 2. The complete FSM for the Parking Lot Counter Circuit.

As discussed above, for the case of a car entering the parking lots, the first sensor will be tripped, going to state “a Blocked”, then it will block both sensors going to state “a & b Blocked”, then finally only sensor b will be blocked leading to state “b Blocked”. As the car progresses to fully entering the parking lot, it will go to state “a & b Unblocked”, which will increase the counter. The path for exiting the parking lot is also detailed and will allow the count to be decreased. At any state, if the car stops, it will remain in the same state. The FSM also accounts for pedestrians, but will result in semi-unpredictable behavior. For example, if the pedestrian walks in front of the first sensor, it will go to state “a Block” then return to “a & b Unblocked”. This will result in the decrease of the counter. If the pedestrian continues to walk in front of just the second sensor, it will just to state “b Blocked” and will increase the count. This will result in a net change of zero, and will continue to have a valid count. If only one of the sensors is covered, then of course it result in a false count. This could be accounted for in a similar manner to how the reversing was handled, that is, a physical structure such as a bush or wall could force pedestrians to either walk in front of none of the sensors, or both.

III Implementation:

With the design of out of the way, I moved onto implementing the design in Verilog on the provided Spartan 3 Development Board. To begin, it is of most use to take a look at how this problem was broken down via the Block Diagram seen below in Figure 3.

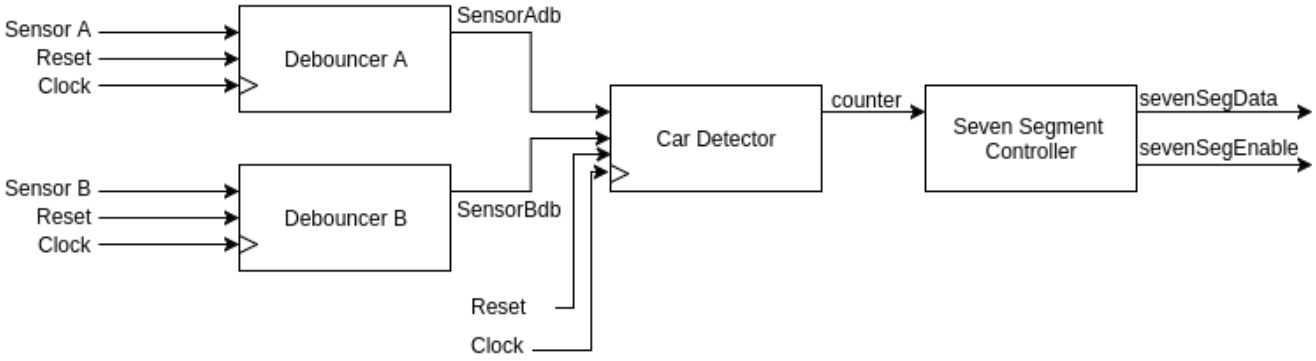


Figure 3. Complete Block Diagram.

As the Block Diagram makes explicitly clear, the circuit itself is not complicated at all. It simply takes the inputs of two Sensors (buttons or switches) through individual debouncing circuits. The circuit then routes the clean input signals to the Car Detector module, which is a combination of the FSM and Counter logic. Finally, the final count signal is sent to the Seven Segment Controller Module that exports the data required to appropriately display the counter value on the display. With that taken care of, we can now begin to observe the code that underlies each of these modules.

Top:

A simple top module that does nothing more than instantiate the required modules as per the block diagram. Note that same names may be formatted differently to appeal to syntax rules of the Verilog language.

```
module top(
        input wire clk, reset,
        input wire sensorA,
        input wire sensorB,
        output wire sensorAdb,
        output wire sensorBdb,
        output wire [3:0] enable,  // enable 1-out-of-4 asserted low
   output wire [7:0] sseg // led segments
);

wire [3:0] counter;

// Debounce sensorA
debouncer dbA (
        .clk (clk),
        .reset (reset),
        .sw (sensorA),
        .db (sensorAdb)
);

// Debounce sensorA
debouncer dbB (
        .clk (clk),
        .reset (reset),
```

```
        .sw (sensorB),
        .db (sensorBdb)
);

//instantiation of car_detector
car_detector count1(
        .clk (clk),
        .res (reset),
        .sensorA (sensorAdb),
        .sensorB (sensorBdb),
        .car_count (counter)
);

//instantiation of sevenSegValue controller
sevenSegValue  brd1(
        .d_in (counter),
        .an (enable),
        .sseg (sseg)
);

endmodule
```

Debouncer:

Taken from Listing 5.6 from the included Book source material. This denouncer uses a free counter to count the duration of a signal level. Once a signal input has been stable for at least 10ms, then it will change the output value. This smooths out any mechanial noise derived from the inputs.

```
// Listing 5.6
module debouncer
  (
   input wire clk, reset,
   input wire sw,
   output reg db
  );

  // symbolic state declaration
  localparam  [2:0]
            zero    = 3'b000,
            wait1_1 = 3'b001,
            wait1_2 = 3'b010,
            wait1_3 = 3'b011,
            one     = 3'b100,
            wait0_1 = 3'b101,
            wait0_2 = 3'b110,
            wait0_3 = 3'b111;

  // number of counter bits (2^N * 20ns = 10ms tick)
  localparam N =19;
```

```
// signal declaration
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
wire m_tick;
reg [2:0] state_reg, state_next;

// body

//=============================================
// counter to generate 10 ms tick
//=============================================
always @(posedge clk)
   q_reg <= q_next;
// next-state logic
assign q_next = q_reg + 1;
// output tick
assign m_tick = (q_reg==0) ? 1'b1 : 1'b0;

//=============================================
// debouncing FSM
//=============================================
// state register
 always @(posedge clk, posedge reset)
    if (reset)
      state_reg <= zero;
    else
      state_reg <= state_next;

// next-state logic and output logic
always @*
begin
   state_next = state_reg;  // default state: the same
   db = 1'b0;               // default output: 0
   case (state_reg)
      zero:
         if (sw)
            state_next = wait1_1;
      wait1_1:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
               state_next = wait1_2;
      wait1_2:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
```

```
            state_next = wait1_3;
        wait1_3:
          if (~sw)
            state_next = zero;
          else
            if (m_tick)
              state_next = one;
        one:
          begin
            db = 1'b1;
            if (~sw)
              state_next = wait0_1;
          end
        wait0_1:
          begin
            db = 1'b1;
            if (sw)
              state_next = one;
            else
              if (m_tick)
                state_next = wait0_2;
          end
        wait0_2:
          begin
            db = 1'b1;
            if (sw)
              state_next = one;
            else
              if (m_tick)
                state_next = wait0_3;
          end
        wait0_3:
          begin
            db = 1'b1;
            if (sw)
              state_next = one;
            else
              if (m_tick)
                state_next = zero;
          end
        default: state_next = zero;
      endcase
    end

endmodule
```

Car Detector:

The meat of my implementation is found here. The code is completely based off of the Verilog code found above. It recieves the two seperate sensor inputs, and then progresses through the FSM as described above. Most important of note is that my counter logic is handled in the same always block as the FSM. This was done to remove some instability in the circuit and ensure that the increase and decrease values could only be high for at most one clock cycle.

```
module car_detector(
	input wire clk, res,
	input wire sensorA,
	input wire sensorB,
	output wire [3:0] car_count
);

localparam abUnblocked=2'b00, aBlocked=2'b01, abBlocked=2'b10, bBlocked=2'b11;
reg [1:0] current_state;
reg increase, decrease;
reg [3:0] count = 0;

//initialize all necessary values
initial begin
	increase = 0;
	decrease = 0;
	current_state = 0;
end

// State register logic
always@(posedge clk, posedge res) begin
	if (res)
		current_state <= abUnblocked;
	else
		begin
			increase <= 0;
			decrease <= 0;
			case(current_state)
				abUnblocked: if ({sensorA,sensorB} == 2'b10) begin
							current_state <= aBlocked;
						end
						else if ({sensorA,sensorB} == 2'b01) begin
							current_state <= bBlocked;
						end
						else if ({sensorA,sensorB} == 2'b00) begin
							current_state <= abUnblocked; //stay
						end
				aBlocked:	if ({sensorA,sensorB} == 2'b10) begin
							current_state <= aBlocked; //stay
						end
						else if ({sensorA,sensorB} == 2'b11) begin
							current_state <= abBlocked;
						end
```

```
                                        else if ({sensorA,sensorB} == 2'b00) begin
                                                current_state <= abUnblocked;
                                                decrease <= 1; //finished exiting
                                        end
                        abBlocked:      if ({sensorA,sensorB} == 2'b10) begin
                                                current_state <= aBlocked;
                                        end
                                        else if ({sensorA,sensorB} == 2'b11) begin
                                                current_state <= abBlocked; //stay
                                        end
                                        else if ({sensorA,sensorB} == 2'b01) begin
                                                current_state <= bBlocked;
                                        end
                        bBlocked:       if ({sensorA,sensorB} == 2'b00) begin
                                                current_state <= abUnblocked;
                                                increase <= 1;
                                        end
                                        else if ({sensorA,sensorB} == 2'b11) begin
                                                current_state <= abBlocked;
                                        end
                                        else if ({sensorA,sensorB} == 2'b01) begin
                                                current_state <= bBlocked; //stay
                                        end
                        default: current_state <= abUnblocked;
                endcase
        if(increase)
                count <= count + 1'b1; //increment the total counter
        else if(decrease)
                count <= count - 1'b1; //decrement the total counter
        end //end else
end //end always

assign car_count = count;

endmodule
```

Seven Segment Controller:

Taken from Listing 4.15 of the Book's Source Material, with some modification, this module allows for the simple translation from a 4-Bit value, and mapping it to a single Seven Segment Display. At an earlier point in the lab session I had attempted an implementation of the Double Dabble algorithm to allow for multi-digit display, but found it also unstable.

```
// Listing 4.15
module sevenSegValue (
        input wire [3:0] d_in,
        output reg [3:0] an,  // enable 1-out-of-4 asserted low
  output reg [7:0] sseg // led segments
);
```

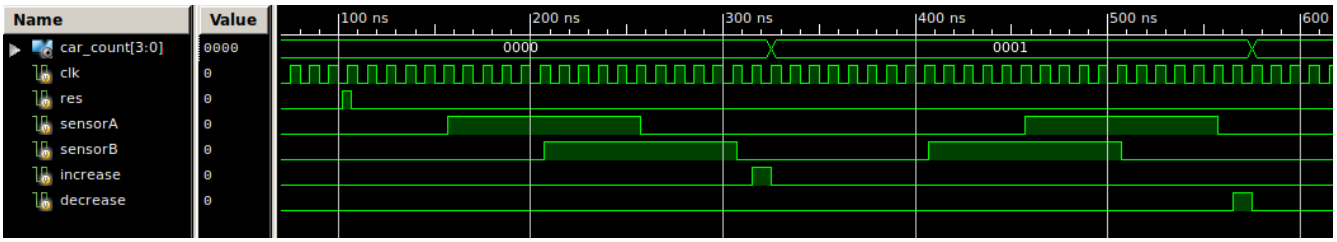
```
  // hex to seven-segment led display
  always @*
  begin
    case(d_in)
      4'b0000: sseg[7:0] = 8'b00000011;
      4'b0001: sseg[7:0] = 8'b10011111;
      4'b0010: sseg[7:0] = 8'b00100101;
      4'b0011: sseg[7:0] = 8'b00001101;
      4'b0100: sseg[7:0] = 8'b10011001;
      4'b0101: sseg[7:0] = 8'b01001001;
      4'b0110: sseg[7:0] = 8'b01000001;
      4'b0111: sseg[7:0] = 8'b00011111;
      4'b1000: sseg[7:0] = 8'b00000001;
      4'b1001: sseg[7:0] = 8'b00001001;
      default: sseg[7:0] = 8'b01110001;  //4'hf
   endcase
         an = 4'b1110; //only use single digit counts
  end

endmodule
```

Now that the code is observed, we may take a quick look at the UCF file to see the way in which the circuit was chosen to be mapped to the board. For the circuit we made use of the onboard 50MHz crystal oscillator for the clock signal. After that, I list the 8-Bit locations of the Seven Segment Display, as well as the four enable bits. Finally, I made use of two of the mechanical switches to ensure that the remain at a given value once debounced, and a single button for the global asynchronous reset value.

```
##-------------------------------------
## Clock
##-------------------------------------
NET "clk"    LOC = "T9" ;

##-----------------------------------
## Seven Segment Display Definition
##-----------------------------------
NET "sseg<7>" LOC = "E14";
NET "sseg<6>" LOC = "G13";
NET "sseg<5>" LOC = "N15";
NET "sseg<4>" LOC = "P15";
NET "sseg<3>" LOC = "R16";
NET "sseg<2>" LOC = "F13";
NET "sseg<1>" LOC = "N16";
NET "sseg<0>" LOC = "P16";

NET "enable<3>" LOC = "E13";
NET "enable<2>" LOC = "F14";
NET "enable<1>" LOC = "G14";
```

```
NET "enable<0>" LOC = "D14";

# Switches as sensors
NET "sensorA" LOC = "K13";
NET "sensorB" LOC = "K14";

NET "reset" LOC = "L14";
```

To ensure the validity of my circuit before placing it on the FPGA, a test was derived only for the FSM itself, as all other modules were either from the text book, or re-used from previously tested code written by myself. The test bench first asserts the reset value, which shows to take an effect. We then go about stepping through the FSM by asserting the signal inputs going from A Blocked, to A&B Blocked, to B Blocked, and finally to A&B Unblocked. Asserting these signals in orders shows the correct output of increase goes high for exactly one clock cycle. The inverse exiting states are then stepped through and the decrease signal is also asserted for exactly one clock cycle.



Some valuable information can also be derived from the summary reports generated by ISE itself at compile time. As noted in the design summary, I am utilizing only 1 of available logic cells, Flip-Flops, and LUTs. As my design makes use of many of the seven Segment Display pins, I am using 10% of all input output blocks, and utilizing 1 of the 12 available Global Clocks.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	27	1920	1%
Number of Slice Flip Flops	33	3840	0%
Number of 4 input LUTs	52	3840	1%
Number of bonded IOBs	18	173	10%
Number of GCLKs	1	8	12%

Finally, by looking at the timing constraint statement, we can observe if all of these changes will occur fast enough for the real world situations in which this circuit may be placed. While this particular circuit won't be in very high-speed situations, it an important value to note because placement in something like a factory line will completely determine the ability to function properly over long periods of time. If a high delay timing propagation is possible even once, then this can result in many broken systems. As noted this should not be a problem for this design, as there are no timing errors and the worst case scenario still only takes 5.7ns, well within the range of a car even traveling at 100 mph.

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net clk_BUFGP	SETUP HOLD	0.821ns	4.983ns	0	0 0
2	Yes	Autotimespec constraint for clock net count1/increase	SETUP HOLD	1.071ns	2.758ns	0	0 0

IV Summary:

Overall, this was a very simple lab that allowed for the introduction of many important topics to the domain of FPGA hardware design. Of most important is the introduction of the finite state machine to our design process. The design of a proper FSM may take some time to develop and ensure that it functions logically under all cases, but once developed, allows for simple one-to-one translation to Verilog code. If the the FSM is presumed to have been logically functional, then the Verilog code can be considered safe to a certain degree, even without testing.

This lab also introduced several common design problems within the scope of Verilog that I learned to address one at a time. Throughout the process of implementation, I was at first receiving very unpredictable results, despite a good software test. With the help of the lab assistant pointing out some of the warning that I presumed to be harmless, these errors were resolved and allowed for very predictable function of the circuit. For example, take just the implementation of my FSM. I had followed the template of a FSM implementation from the book to a tee. This however began to result in the increase and decrease values being asserted for for more than one clock cycle and consequently increasing the counter in even intervals. This was fixed by removing an intermediate state and simply jumping directly to the next state on the next clock cycle. This greatly improved the timing of the circuit overall. This lab allowed me to understand the concepts of FSM design more thoroughly and begin to see some of the associated problems, providing enough time to resolve all of the issues as well.