Sequence Detectors

Ben Heard

Thus far we've covered a fwe different ways to implement digital logic using Verilog. For the sake of terminology we use Verilog to describe hardware (thus the HDL or Hardware Description Language part of the name). Specifically, we've implemented combinational and sequential designs structurally by instantiating built in primitives like and, or, not (gate-level) and by instantiating other modules that have already been designed for us like the DFF or 2:1 mux in the ECE310 library.

We've also implemented purely combinational designs using dataflow with assign statements. And, just recently, implemented both combinational and sequential design procedurally using always @ blocks. Both of these styles of description are behavioral. It's still possible to work out what the gates would look like but there is another layer of abstraction that allows us to describe what we want but not have to work out the details all the way down to the gates.

In this lab you'll be able to put to use some of the design techniques that you've seen to implement a pair of sequence detectors.

1 Sequence Detectors

In the most general sense, a sequence detector is able to identify when a pattern exists in a data set. In the application for this lab, a sequence detector is able to detect a binary string that exists in a transmitted stream. Specifically, a single bit wide data input to the sequence detector changes each clock cycle sending a binary string over time.

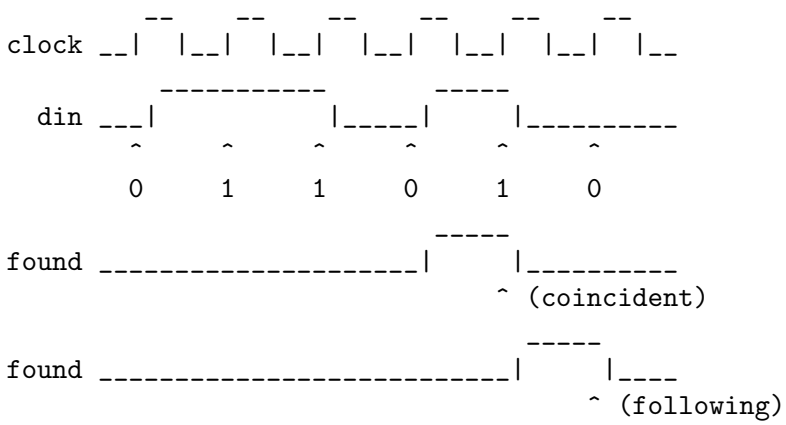
This particular implementation may be useful in systems that need to minimize the number of physical connections between a transmitter and a receiver. Using a shared clock between the two and a single wire for data is about as minimal as you can get. For a transmitter, then, to provide data to a receiver and the receiver be able to reconstruct the data a pattern may be used to indicate the start of the data message.

1.1 Characteristics

For our application there are two pair of sequence detector characteristics. The first is whether overlapping patterns are allowed. For example, the pattern 1101 may be found 3 times in the following string when overlapping patterns are allowed but only twice if overlapping patterns are not allowed.

```
   ____   _______         overlapping
1001101000110110100101 10
   ^^^^   ^^^^           non-overlapping
```

The other dimension is when the assertion of any found, detected, seen output is to be asserted. Should the output be asserted at the same time that the final bit of the pattern is seen by the machine? This would coincident with the last bit of the pattern. If the output is asserted in the clock cycle following the receipt of the last bit then the output is said to be non-coincident or following.



2 Implementations

2.1 State Diagram

A sequence detector may be implemented as a state machine from a state diagram. In this style of implementation a two process state machine implementation works well. You may use parameters to provide more information to a reader of the design and keep things well organized. For example, see the listing in section 4 that shows a detector for the pattern 1101 non-overlapping and coincident output.

2.2 Shift Register

Another option for a sequence detector implementation is to use a shift register. This implementation is very convenient because it's compact. And, the shift register functionality already captures bits as they come in. It is especially useful when the requirements are for an overlapping and following output.

See the listing in section 5 for an implementation using a shift register for an overlapping non-coincident detection of the pattern 101.

While the state diagram implementation may be updated by changing the arcs to get a coincident output, the shift register implementation requires a little more work. Specifically, the first register must be removed and the output based on the input along with the previously shifted bits.

Similarly, a structural change is also required to implement the non-overlapping case. A shift register naturally lends itself to the overlapping detection because it retains the previous state once a pattern is detected once. If the requirement is for non-overlapping then the register must be cleared once a pattern is seen. And, that value it's cleared to (either 0's or 1's) is dependent on the pattern that you're trying to detect.

If you are detecting a pattern that starts with a '1' then you would clear the register to all zeroes so that nothing in the register may be misinterpreted as the start of a pattern. Similarly, if the start of the pattern was to be a 0 then you would clear the register to all 1's to ensure that none were misinterpreted as the start of the pattern.

3 Assignment

1. (**50 points**) Implement a sequence detector that will find overlapping sequences of 1101. The output, **found** shall assert following each receipt of the sequence (non-coincident). You may use any implementation style you choose. Your implementation shall conform to the following requirements.

    - The module name shall be **lab6_seq_1101**.
    - The module shall have an active low synchronous reset input called **rst_n**.
    - The module shall have a free-running clock input called **clock**.
    - The module shall accept on the rising clock edge from an input called **d_in**.

- The module shall assert a single bit output called **found** following identification of the sequence.

2. (**50 points**) Implement a sequence detector that finds non-overlapping sequences of 101. The output, **found**, shall assert following each receipt of the sequence. The implementation shall be shift register based. Your implementation shall conform to the following requirements.

   - The module name shall be **lab6_seq_101**.
   - The module shall have an active low synchronous reset input called **rst_n**.
   - The module shall have a free-running clock input called **clock**.
   - The module shall accept on the rising clock edge from an input called **d_in**.
   - The module shall assert a single bit output called **found** following identification of the sequence.
   - The implementation shall be shift register based.
   - The implementation shall find non-overlapping sequences. E.g. in the pattern 101101, two instances will be found while in 10101 only the first will be found.

   NOTE: This detector is required to be shift register based and detecs non-overlapping sequences. This means that you'll need to determine how to clear your shift register when you receive a pattern.

4 Non-overlapping Coincident 1101

```
module sd_1101 {
  input clk, rst_n, din,
  // reg type because assigned in procedural block
  output reg found
};

  // some parameters to help read what's going
  // on in the state machine
  localparam INIT = 2'b00;
  localparam FIRST_ONE = 2'b01;
  localparam SECOND_ONE = 2'b10;
  localparam ZERO = 2'b11;
```

```
// reg data type because they are assigned in
// a procedural block
reg [1:0] cstate, nstate;

// simple state vector that sets to the INIT
// state on reset and on each clock edge takes
// on the value of nstate
always @( posedge clk )
  if( !rst_n )
    cstate <= INIT;
  else
    cstate <= nstate;

always @* begin
  case( cstate )

    // in this init state we wait for the first
    // one '1' to be see in the input
    INIT :
      begin
        found  = 0;
        nstate = din ? FIRST_ONE : INIT;
      end

    // now that we've seen the first one we look
    // for the second one; if there is a zero then
    // we've blown the pattern and need to start
    // over
    FIRST_ONE :
      begin
        found  = 0;
        nstate = din ? SECOND_ONE : INIT;
      end

    // now that we've seen the second one we are
    // looking for the zero; however, if we see
    // a one '1' that could just be the second
    // one of a pattern that had leading ones
    SECOND_ONE :
```

```
        begin
          found  = 0;
          nstate = din ? SECOND_ONE : ZERO;
        end

      // finally, we've seen 110 and we're waiting
      // on the last one '1'; if we see the one
      // then we return to the beginning (because of
      // the non-overlapping requirement) and output
      // a '1' indicating that we found the patter;
      // if we see a zero then we've blown the
      // pattern and need to go back to the beginning
      // to start over
      ZERO :
        begin
          found  = din;
          nstate = INIT;
        end

      // always have a default case
      default :
        begin
          found  = 0;
          nstate = INIT;
        end
```

5 Overlapping Following 101

```
module sd_101 {
  input clk, rst_n, din,
  output found
);

  // 3-bit register to hold the pattern
  reg [2:0] sr;

  always @( posedge clk }
    if( !rst_n )
      sr <= 0;
```

```
    else
      // if not in reset then shift;
      sr <= {din, sr[2:1]};

  // the found output comes from comparing the shift
  // register contents to the value that we're looking
  // for; in this case, since the pattern is symmetric
  // as 101, we just compare; if the pattern were not
  // symmetric then you would need to flip the order
  // of the bits because of the way the bits come in
  // to the register
  //
  // recall, too, that values are both numeric and
  // logical; we perform the logical comparison and that
  // will yield a single bit logical result that we then
  // use for its numeric value
  assign found = ( sr == 3'b101 );

endmodule
```