ECE 451 Verilog Exercises

Sept 14, 2007

James Barnes (James.Barnes@colostate.edu)

Organization

- These slides give a series of self-paced exercises. Read the specification of each exercise and write your code before proceeding to the solution slide.
- These exercises will be most useful if you have access to a verilog simulator (modelsim, Icarus verilog) as you read these slides. See Dr. Barnes if you need help installing a simulator.

Coding guidelines and standard practices

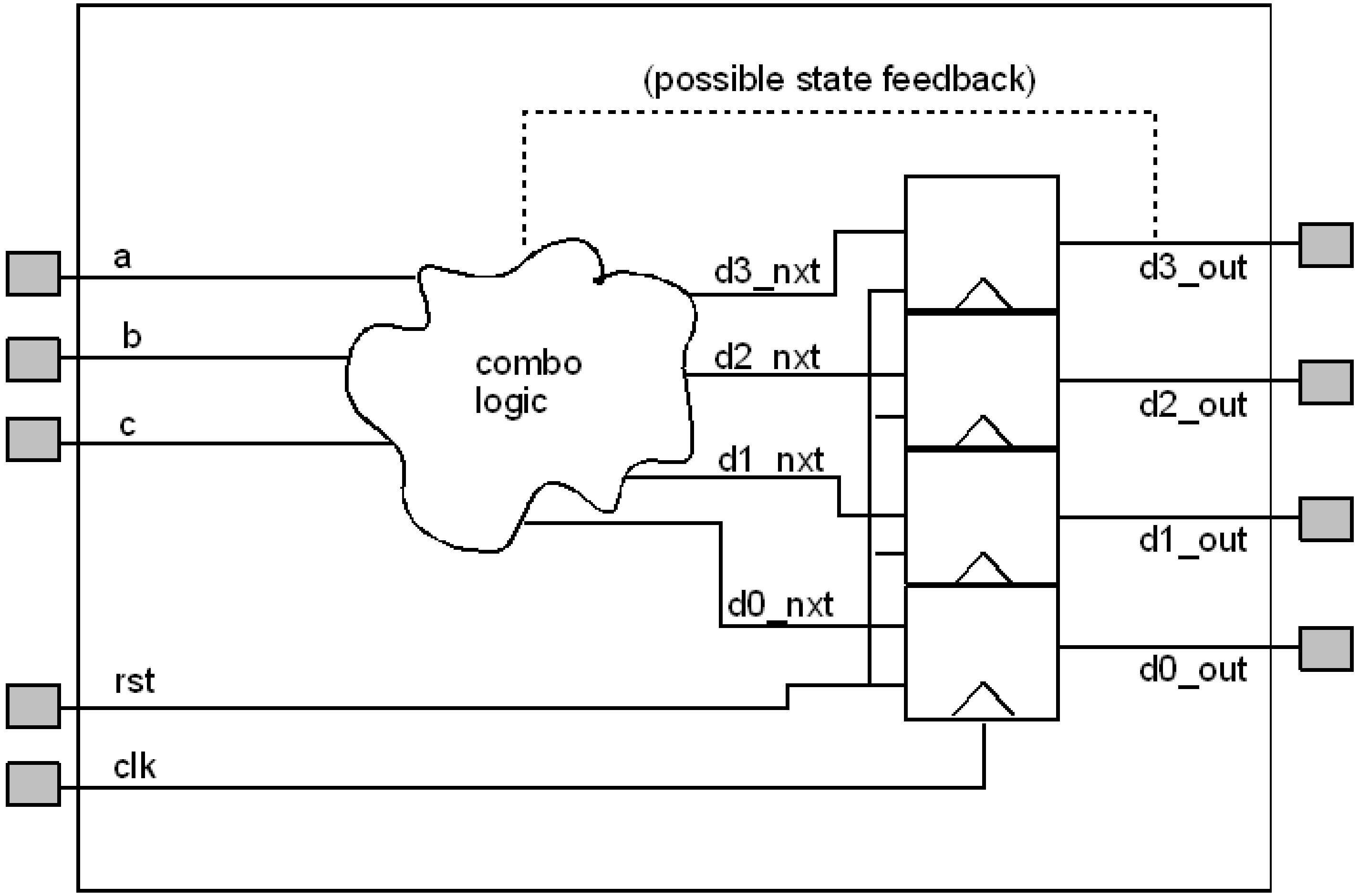
- Most design groups larger than a few designers have guidelines and standard practices for writing code. These make it easier to share code, as the code is organized in a way that makes it easier to find things.
- Disclaimer: each design organization has its own way of doing things. You may encounter different guidelines, but the principle is pretty universal.

Guidelines

- Logic blocks typically have the form of a combinational logic cloud feeding a register, with possible state feedback.
  - An example would be a counter, where the register holds the count value and the combinational logic updates the count value. This block would have state feedback.
  - Some blocks may have only one of either a combinational logic or a register block.
  - Some blocks may have multiple instances of combinational logic and register blocks. Most organizations do not impose rules on how much can be in any one module – this is up to the designer.
- One common rule is that the outputs must be registered. This gives the next block the maximum time to process and register the signals.

Guidelines

- A typical logic block is shown below.



Guidelines for register updates

- Standard practice (a requirement in some organizations) is that the register updates should be done in an `always` block with NO logic except a synchronous reset. Non-blocking assignments (<=) must be used. The register update section for the previous example would look like

```
always @(posedge clk)
    begin
    d3_out <= rst ? 1'b0 : d3_nxt;
    d2_out <= rst ? 1'b0 : d2_nxt;
    d1_out <= rst ? 1'b0 : d1_nxt;
    d0_out <= rst ? 1'b0 : d0_nxt;
    end
```

- Reset is done with the conditional assignment. If the test variable (`rst` here) is true, the value before the : is taken; otherwise, the value after the : is taken.

Guidelines on combinational logic

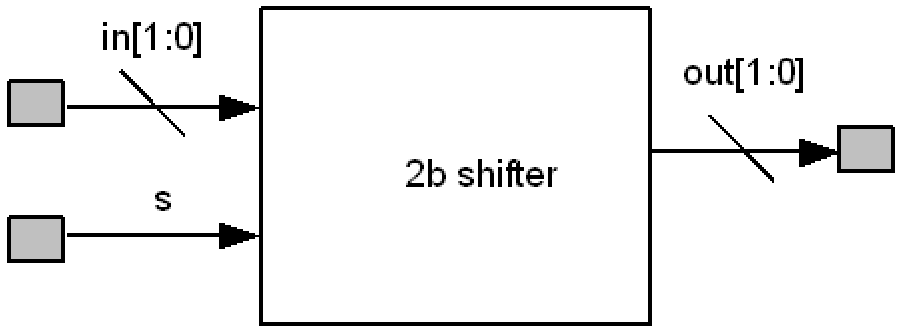
- All the combinational logic should be done in one place in the module code and indicated to anyone reading the code through comments. This is true whether the combinational logic is done using `wire` assignments or an `always` block.
- If combinational logic is implemented within an `always` block, the blocking assignment (=) must be used. This will assure that the combinational logic is evaluated before the register update is done.

Verilog syntax

- Reminder on some verilog syntax rules:
  - All inputs in a module are of the `wire` type. You cannot declare inputs to be `reg` type.
    - You cannot re-assign or change an input. Inputs can only appear on the RHS of assignments or as a test variable in a conditional assignment or control flow statement.
  - Outputs can be either `wire` or `reg`.
  - There will be some internal signals which are neither input nor output. For example, the signals `d3_nxt`,... in the example.
  - All multi-bit signals of `wire` type must be declared. Single bit input signals need not be declared as they default to `wire` type.
  - All variables assigned inside a procedural block (`initial` or `always`) must be of reg type.
  - All variables of `reg` type must be explicitly declared.

Exercise 1 – 2b shifter

- This is the same logic block as in Q2a of HW #2, but with bus notation for the inputs and outputs. This is a purely combinational logic block. The logic equations are:
  - out[0] = s' ● in[0]
  - out[1] = s' ● in[1] + s ● in[0]

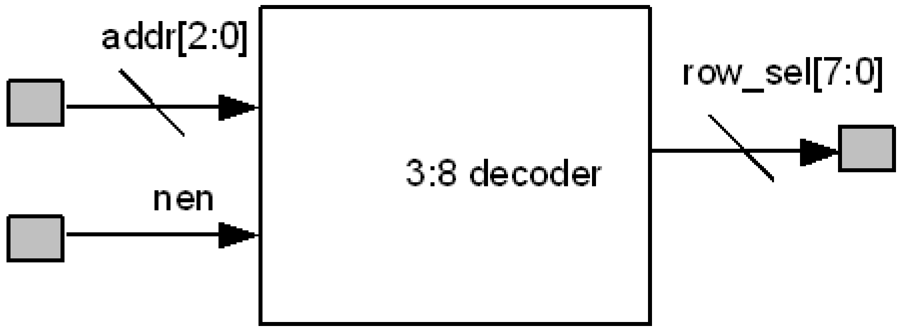


Ex 1 Solution

```
module shifter(in, out, s);
   input [1:0] in;
   input       s;

   output [1:0] out;

   wire   s;
   wire [1:0] in;
   wire   out;

   assign out[1] = (~s) & in[1] | s & in[0];
   assign out[0] = (~s) & in[0];

endmodule
```

Ex 2 – 3:8 row decoder with enable

- This decoder has inputs `addr[2:0]` and an active low enable `nen`. It drives 8 active high output lines `row_sel[7:0]`, one of which is driven when `nen` is asserted.

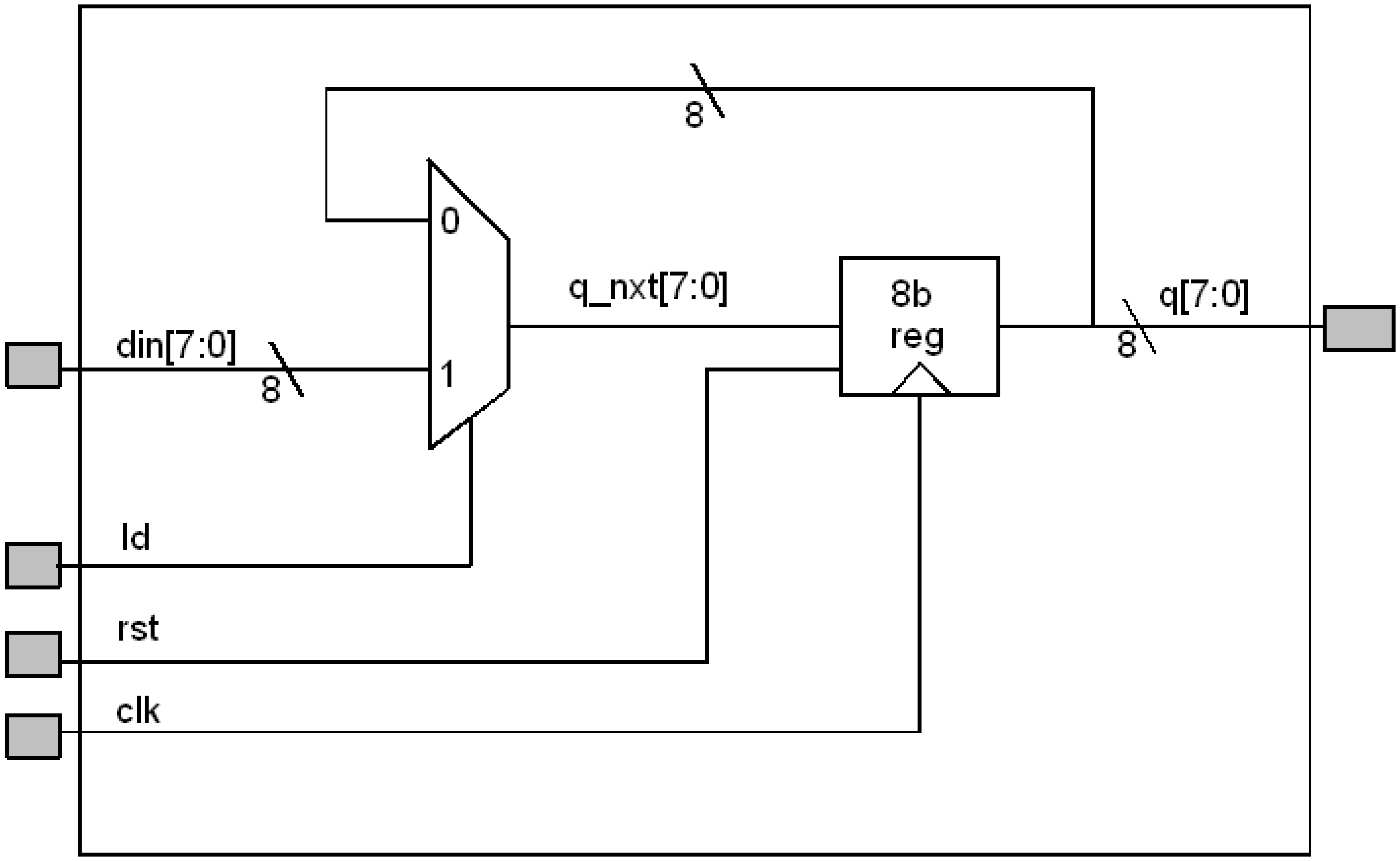


Ex 2 Solution

```
module row_decoder(row_sel, addr, nen);
   input [2:0] addr;
   input       nen;

   output [7:0] row_sel;

   wire [2:0]    addr;
   wire  nen;
   reg [7:0]     row_sel;

   // Use a case statement
   always @(addr or nen)
     begin
    if (nen)
      row_sel = 8'h0;
    else
      case (addr)
        3'h0: row_sel = 8'b0000_0001; // The _ is just for readability
        3'h1: row_sel = 8'b0000_0010;
        3'h2: row_sel = 8'b0000_0100;
        3'h3: row_sel = 8'b0000_1000;
        3'h4: row_sel = 8'b0001_0000;
        3'h5: row_sel = 8'b0010_0000;
        3'h6: row_sel = 8'b0100_0000;
        3'h7: row_sel = 8'b1000_0000;
      endcase // case(addr)
     end // always @ (addr or nen)

endmodule // row_decoder
```

Ex 3 – 8b register with load and synchronous reset

- This block implements an 8b wide register from DFFs. All flops are driven by a common clock and have a common reset `rst`. An input mux allows new data to be loaded into the register when `ld` is high; otherwise, the old data is recirculated.



Ex 3 Solution

```
module ld_reg8(din, clk, rst, ld, q);

   input [7:0] din;
   input       clk, rst, ld;
   output [7:0] q;

   wire [7:0]    din, q_nxt;
   wire   clk, rst, ld;
   reg [7:0] q;

   //Logic on register inputs
   assign     q_nxt = ld ? din : q;

   // Update register
   always @(posedge clk)
     q <= rst ? 8'h0 : q_nxt;

endmodule // row_decoder
```