Verilog RTL Modeling

- This assignment introduces you to Verilog RTL modeling
- Similar in concept to VHDL RTL, just different syntax
- Will use serial data transfer as the problem to be solved

Serial Communication

- Serial communication is as widely (or even more widely used) than parallel communication
  - Especially true if communication occurs over long wires
- Many new high speed serial communication standards have been developed
  - USB, IEEE Firewire, HyperTransport, etc.
- This lab will introduce you to some basic serial communication concepts, namely *bit-stuffing* and *NRZI encoding*
  - These techniques are used in the USB (Universal Serial Bus) interface.

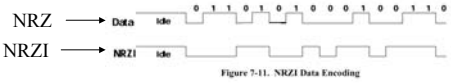


Figure 7-11. NRZI Data Encoding

Non-return to zero (NRZ) - normal data transitions.

NRZ Inverted (NRZI, not a good description, is not inverse of NRZ). A transition for every zero bit.

Strings of zeros means lots of transitions. Strings of '1's means steady line.

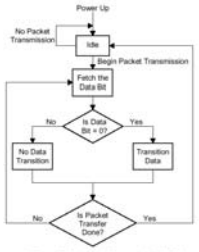


Figure 7-12. Flow Diagram for NRZI

Bit Stuffing – a '0' is inserted after every six consecutive '1's in order to ensure a signal transition so that receiver clock can remain synchronized to the bit stream.

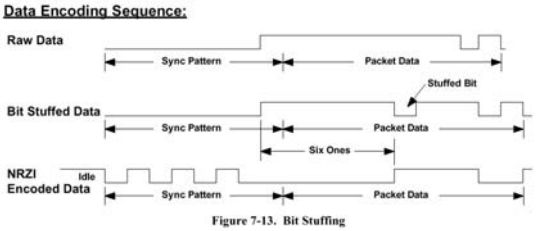
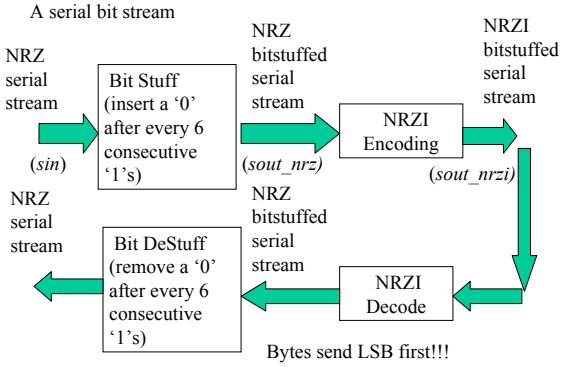
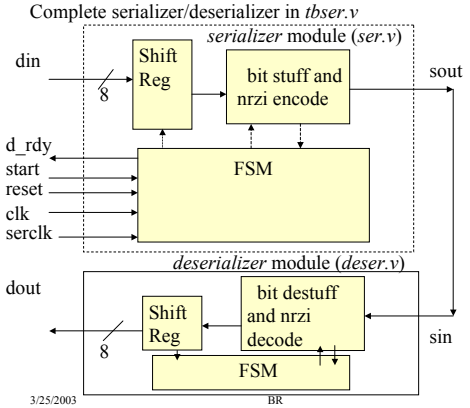


Figure 7-13. Bit Stuffing

Bit stuffing done automatically by sending logic. Sync pattern starts data transmission and is seven '0's followed by a '1'.





The Task

- You are to design the *serializer* module (in file *ser.v*) using Verilog RTL
  - May need several modules within file *ser.v*, top level module is called *serializer* and has the interface shown
  - Your *ser.v* code must be synthesizeable
- I have designed *deserializer* module (in file *deser.v*) and testbench (*tbser.v*).
  - Testbench connects the serializer/deserializer modules together
  - Also sends 32 bytes to serializer/deserializer for testing purposes

Serializer Module

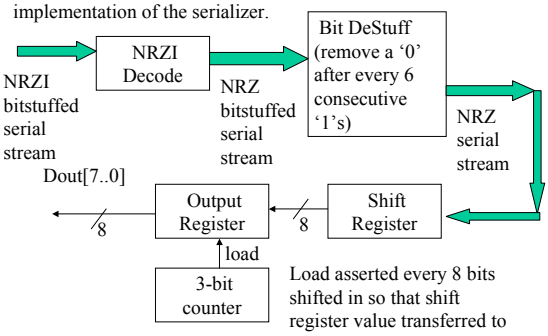
- Should wait until *start* is asserted
- Send value on *din* serially over *sout*
- Request new value on din by asserting d_rdy
  - In testbench, there is a clock cycle latency between assertion of *d_rdy* and a new *din* value being provided
- Continually send serial data until reset is asserted.
- Main clock is signal *clk*. The serial clock is *serclk* which has 1 clock pulse for every 4 pulses on *clk*.
  - New serial data should be provided for every pulse on *serclk*.
  - Both *clk* and *serclk* provided by testbench.

Zip Archive *serial.zip*

- Contains directory serial, which contains files *tbser.v, ser.v, deser.v* .
- Also contains a *Modelsim* golden waveform called *serial_vsim.wlf* and command file *serial_wave.do* To view this waveform do:
  - qhsim –view serial_vsim.wlf –do "do serial_wave.do"
  - Shows all signals in *tbser.v* from golden simulation.
- The file qhsim_gold_log.txt contains the golden output
  - Testbench just sends 32 bytes to serializer/deserializer
  - Each time a new byte comes out of the deserializer, it is printed to screen.
- Synopsys script file *ser.script* for testing if verilog code is synthesizeable.
  - Your synthesized gate level code must produce same result as RTL code

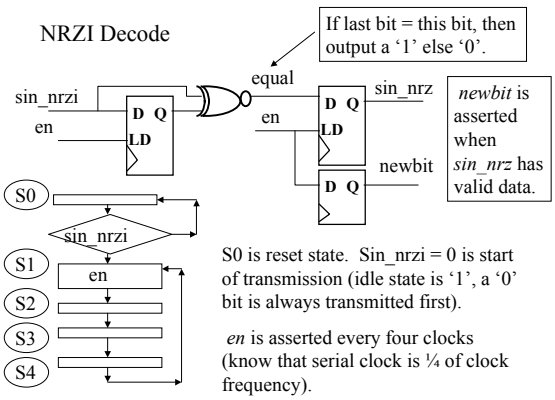
DESerializer Operation

Understanding the DESerializer operation may help with implementation of the serializer.



Load asserted every 8 bits shifted in so that shift register value transferred to output reg.

NRZI Decode



S0 is reset state. Sin_nrzi = 0 is start of transmission (idle state is '1', a '0' bit is always transmitted first).

en is asserted every four clocks (know that serial clock is ¼ of clock frequency).

Finite State Machine for *desnrz*

```
module desnrz (sout,newbit,clk, reset,sin);
output sout,newbit;
input clk,reset,sin;
reg [3:0] _sin,en,sout,newbit;
reg [2:0] state,nstate;
`define S0    'b000
`define S1    'b001
`define S2    'b010
`define S3    'b011
`define S4    'b100
`define S5    'b101
`define S6    'b110
`define S7    'b111
```

reg declaration required anytime a signal is assigned a value from an assignment statement in a procedure block.

Does not imply that a 'register' will be synthesized

State definitions

Procedural block for FSM state storage

Triggered on rising edge of clock, so outputs will have a rising-edge DFF synthesized.

```
always @(posedge clk) begin
  if (reset)   begin
     state <= `S0;
     l_sin <= 1;
     sout <= 1;
   end
  else state <= nstate;

  newbit <= 0;
  if (en) begin
    if (l_sin != sin) sout = 0;
      else sout <= 1;
    newbit <= 1;
    l_sin <= sin;
  end
end
```

Synchronous reset

en asserted by FSM logic every 4 clocks since we know serial clock is ¼ clock freq.

l_sin is last serial input. If last serial input not equal to current serial input, then was a '0' value. If the last serial bit is equal to current bit, then a '1'. The newbit asserted to indicate a valid serial output bit.

Combinational Block

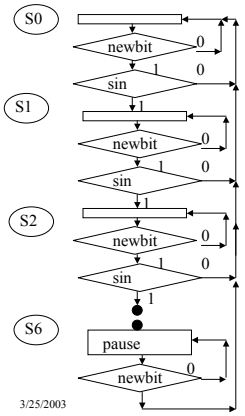
Triggered on any changes to *state* or *sin*

```
always @(state or sin) begin
 nstate = state;
 en = 0;
 case (state)
  `S0: // wait for start edge
        if (!sin) nstate = `S1;
  `S1: begin
         en = 1; nstate = `S2;
       end
  `S2: nstate = `S3;
  `S3: nstate = `S4;
  `S4: nstate = `S1;

   default:  nstate = `S0;
  endcase
 end
endmodule
```

Default output assignments (en negated, stay in same state)

Need *begin/end* if more than one statement in block.



Bit De-stuffing

pause asserted when six '1' bits detected. The pause signal used to halt shift register so that the '0' bit which was stuffed is not shifted into register.

8-bit Shift register in *deser.v*

```
module des_shift (dout, sin, clk, reset, newbit,pause);
output [7:0] dout;
input clk, reset, newbit,pause, sin;
reg [7:0] dout;

 always @(posedge clk) begin

  if (reset)  dout <= 'b00000000;
  else if ((newbit) && (!pause)) begin
     dout[6:0] <= dout[7:1];  //right shift by 1
     dout[7] <= sin;
   end
 end
endmodule
```

Shift occurs if newbit available and not destuffing (pause == 0).

Synchronous reset

Data sent LSB first so shift data into MSB.

3-bit Counter register in *deser.v*

```
module descnt (dout, zero, clk, reset, newbit,pause);
output [2:0] dout;
output zero;
input clk,reset,newbit,pause;

reg [2:0] dout;
reg zero;

 assign zero = ~dout[2] & ~dout[1] & ~dout[0] ;

 always @(posedge clk) begin
   if (reset) dout <= 'b000;
   else if ((newbit) && (!pause)) dout <= dout + 1;
 end

endmodule
```

Assert *zero* when counter value = 0. This output used to control loading of output register.

Increment counter if newbit available and not destuffing.

8-bit Register in *deser.v*

```
module outreg (q,d,r,clk,ld);

output [7:0] q;
input [7:0] d;
input r,clk,ld;


reg [7:0] q;


always @(posedge clk) begin
  if (ld) q <= d;
  if (r) q <= 'h00;
end

endmodule
```

Hex formatting.

Note that synchronous reset takes precedence over synchronous load.

Deserializer module – connects other modules together

```
module deserializer (dout, clk, reset, sin);
output [7:0] dout;
input clk, reset, sin;

wire [2:0] bitcnt;
wire [7:0] sdout;
wire [7:0] dout;

 dff u_dff (lat_sin,sin,reset,clk);
 desnrz u_desnrz (sout_nrz,newbit, clk, reset,lat_sin);
 destuff u_destuff (sout,pause,newbit,sout_nrz,reset,clk);
 descnt u_descnt (bitcnt, zero, clk, reset, newbit,pause);
 des_shift u_shift (sdout, sout, clk, reset, newbit,pause);
 outreg u_outreg (dout,sdout,reset,clk,zero);

endmodule
```

Must explicitly declare the widths of any wires whose width is not 1. (default width is 1).

Asynchronous vs Synchronous Inputs

```
reg q;

always @(posedge clk) begin
  if (r) q <= 0;
   else q <= d;
```

Synchronous reset, high true

```
reg q;

always @(posedge clk or posedge r)
begin
 if ( r) then q <= 0;
 else q <= d;
end
```

Asynchronous reset –, high true.

Need 'posedge' on 'r' because Verilog syntax requires if any signals are edge-triggered in event list, all signals must be edge-triggered.

Style suggested by C. Cummings, SNUG 2002

tbser module in *tbser.v*

```
module tbser;
 reg clk,reset,start;
 wire [7:0] din;
 wire [4:0] addr;
 wire [7:0] dout;
 reg [7:0] last_dout;

 initial begin
   clk = 0;
   reset = 1;
   start = 0;
   last_dout = 'h00;
 end
```

Declaration of wires with non-default widths

Any block with 'initial' keyword only executed once.

tbser module in *tbser.v* cont.

Clock generation

```
always #(200/2) clk = ~clk;

serclkgen u_serclk (serclk, clk, reset);
serializer u_ser (sout, d_rdy, din, clk,serclk,reset, start);
deserializer u_des (dout, clk, reset, sout);
cnt5  u_cnt5 (addr, clk, reset, d_rdy);
rom u_rom (din, addr);
```

'serclkgen' module generates serial clock.

'rom', 'cnt5' used to generate 8-bit input values to serializer module ('rom' provides data values, 'cnt5' is 5-bit counter that provides address to 'rom' module. 'cnt5' incremented anytime that 'd_rdy' is asserted.

tbser module in *tbser.v* cont.

trace block prints *dout* value anytime it changes and *serclk* is asserted.

```
always @(posedge clk) begin : trace
   if (serclk == 1) begin
    if (last_dout != dout)  $display("Dout = %h ",dout);
    last_dout = dout;
   end
 end
```

Can name blocks (not required)

Print in hex format.

tbser module in *tbser.v* cont.

```
always  begin : stim
  @(posedge clk);
  @(posedge clk);
  reset = 0;
  @(posedge clk);
  start = 1;
  @(posedge clk);
  start = 0;
  while (addr == 0) begin
    @(posedge clk);
  end
  while (addr != 0) begin
    @(posedge clk);
  end
  while (addr == 0) begin
    @(posedge clk);
  end
  $finish;
 end
```

stim block provide stimulus for input signals.

Note use of @(posedge clk) -- waits until rising edge before continuing.