CECS 360
Verification

Modeling with a High Level Language

The Device Under Test

- In 460 we are required to design the Transmit portion of a UART
  - One of the functions that must be performed is to receive the byte from the processor and format it to be shifted out over the TX line
  - There is a defined algorithm for formatting the data
- This presentation shows how to create the expected vectors using a C program and then how to use the vectors to verify the design

The Device Under Test

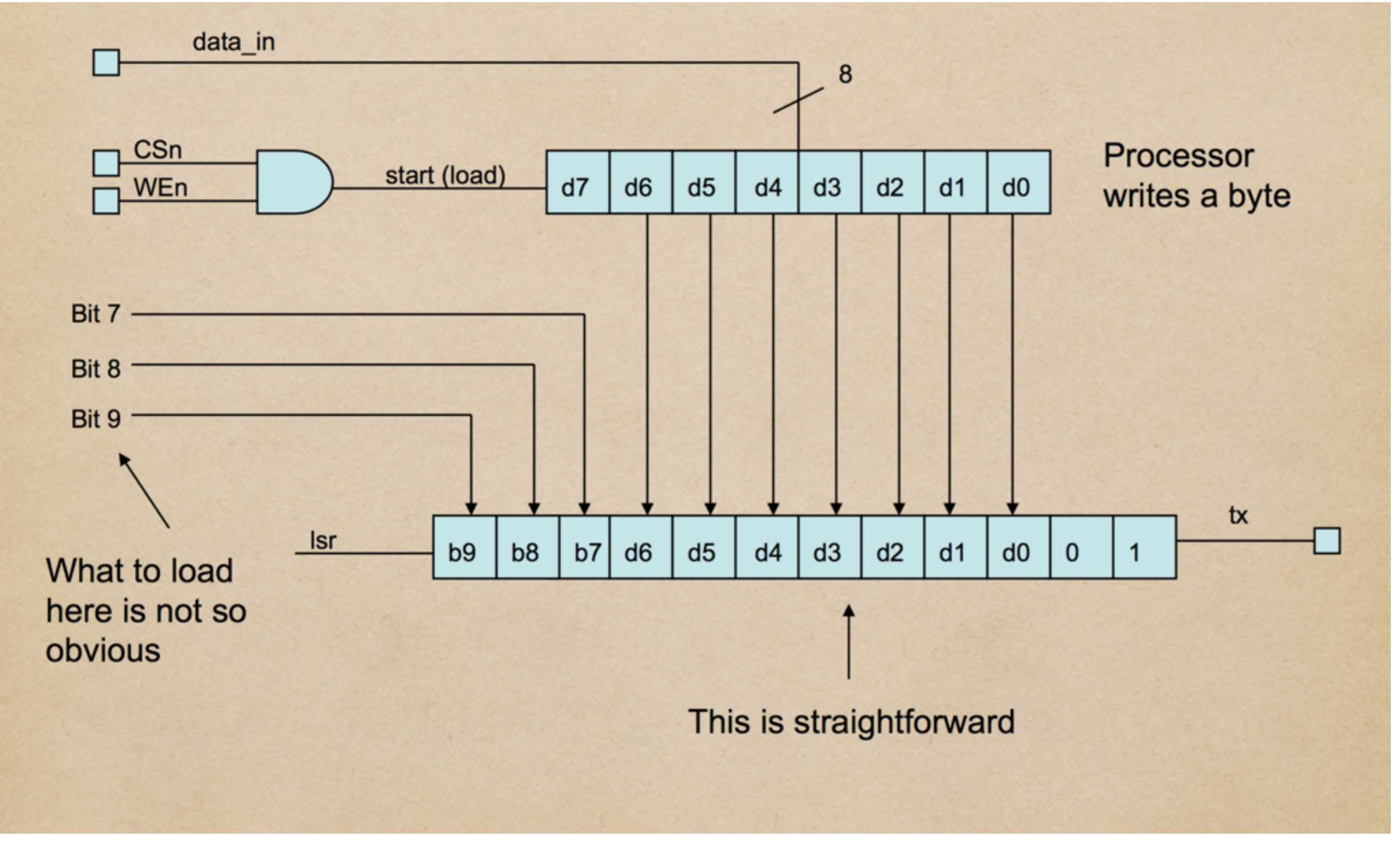
```
// transmit UART

module transmit (clk, rstb, port_id, port_out, read_strobe, write_strobe, tx,
                 bit8, parity_en, odd_n_even,shift_reg, done, shift);

input          clk;              // 50MHz clock - posedge used
input          rstb;             // low active asynchronous reset (synchronously released)
input    [7:0] port_id;          // eight bit address out of Picoblaze Processor
input    [7:0] port_out;         // eight bit data out of Picoblase Processor
input          read_strobe;      // high active read strobe from Picoblaze Processor
input          write_strobe;     // high active write strobe from Picoblaze Processor
input          bit8;             // one bit 1: 8 bits 0: 7 bits
input          parity_en;        // one bit parity enable 1: enabled 0: disabled
input          odd_n_even;       // one bit parity select 1: odd 0: even
input          done;
input          shift;

output         tx;               // transmit out to UART interface
output [10:0]  shift_reg;        // shift register to move the data out tx
```

The Device Under Test



The Device Under Test

- The functionality of the bits to be loaded, {b9,b8,b7} can be captured in the form of a truth table
- The LHS represents the configuration and the RHS represents the bit values along with the number of bits to transmit

bit8	parity_en	odd_n_even	b9	b8	b7	cnt
0	0	0	1	1	1	9
0	0	1	1	1	1	9
0	1	0	1	1	EP	10
0	1	1	1	1	OP	10
1	0	0	1	1	d7	10
1	0	1	1	1	d7	10
1	1	0	1	EP	d7	11
1	1	1	1	OP	d7	11

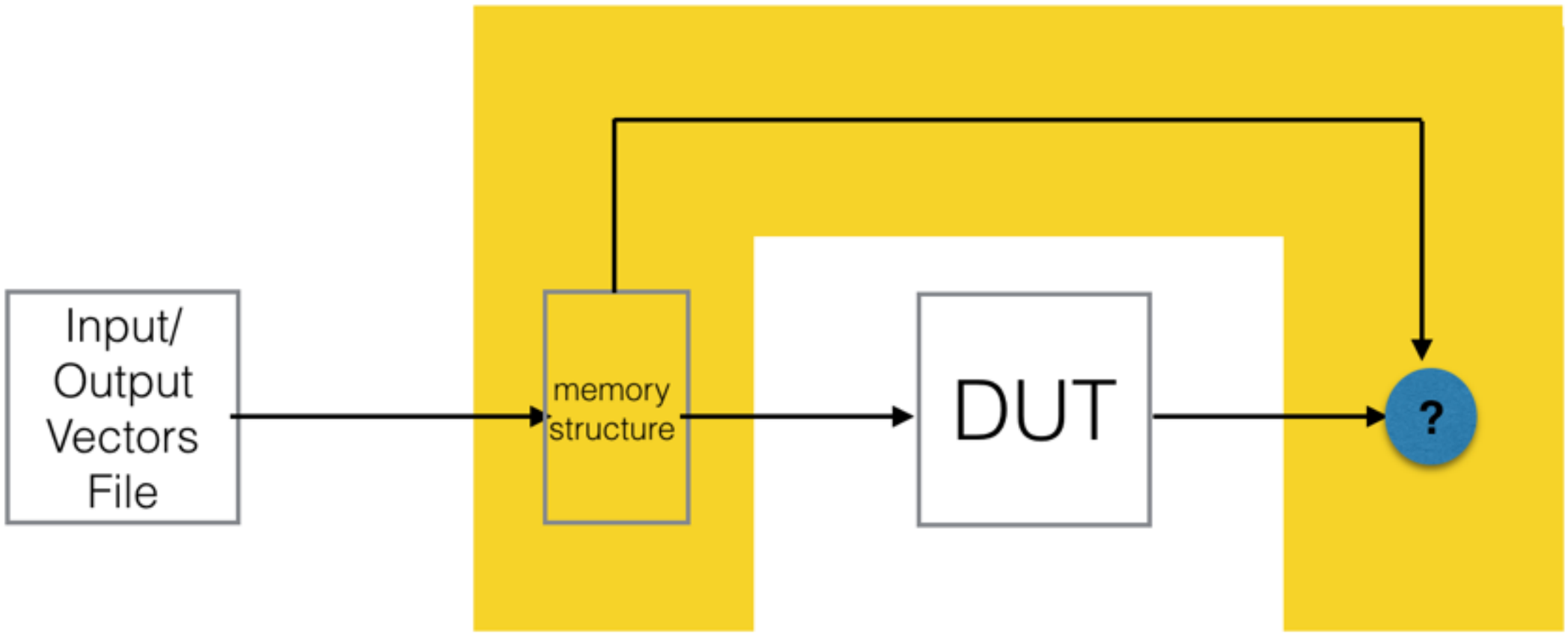
The Issue

- When presented with an algorithm to implement in Verilog sometimes the design of the test bench becomes rather complicated

- It is sometimes advantageous to let the complexity be assigned to the generation of the stimulus and let the test bench be simple in its implementation

- The test bench then applies the stimulus vectors as inputs to the design and then once the stimulus has propagated to the outputs to use another vector serve as the expected results. The test bench then compares the outputs of the design to the expected vector and determines if it is correct or not

The Device Under Test



cpp Code

```
//
//  main.cpp
//  460 Test Code
//
//  Created by John Tramel on 10/9/15.
//  Copyright © 2015 John Tramel. All rights reserved.
//

#include <stdio.h>

int loop;
short buf[16] = {123, 222, 17, 29, 31, 22, 84, 231, 12, 45, 85, 64, 77, 85, 66, 126};
#define bool int

// prototypes

bool getParity(unsigned int n);

// main loop

int main(int argc, const char * argv[]) {

    short buildit;
    short keepit;
    int i;

// insert code here...
printf("Hello, World!\n");
for (loop = 0; loop < 8; loop++)
{
    switch (loop) {
        case 0:  // 7N1
            for (i = 0; i < 16; i++) {
                buildit = buf[i] & 0x7F;  // grab 7 bits
                buildit = buildit << 2;
                buildit = buildit | 0xE01;
                printf(" %d %X %X\n", loop, buf[i],buildit);
            }
            break;
        case 1:  // 7N1
            for (i = 0; i < 16; i++) {
                buildit = buf[i] & 0x7F;  // grab 7 bits
                buildit = buildit << 2;
                buildit = buildit | 0xE01;
                printf("%d %X %X\n", loop, buf[i],buildit);
            }
            break;
        case 2:  // 7E1
            for (i = 0; i < 16; i++) {
                buildit = buf[i] & 0x7F;  // grab 7 bits
                keepit = buildit;
                buildit = buildit << 2;
                buildit = buildit | 0xC01;
                if (getParity(keepit))
                    buildit = buildit | 0x200;
                printf("%d %X %X\n", loop, buf[i],buildit);
            }
            break;
```

cpp Code

```cpp
// parity generator
// returns odd parity
// if odd number ones on input - returns a 1

bool getParity(unsigned int n)
{
    bool parity = 0;
//    printf("IN: %X ",n);
    while (n)
    {
        parity = !parity;
        n      = n & (n - 1);
    }
//    printf("Parity: %X\n",parity);
    return parity;
}
```

The Process

- Run the HLL program to generate an ASCII text file of the vectors (Binary/Hex)
- Use the file to load up the stimulus/expected vectors
- Run the test bench to verify your design

Memories in Verilog (QuickVerilog)

- A memory may be modeled in Verilog using procedural blocks
- A memory is defined as a two-dimensional array of registers
  - reg [15:0] mymem [0:1023];//1k x 16
    - [15:0] defines width and bit orientation
    - [0:1023] defines range of addresses
    - First address range value is the first address (applicable to system tasks that fill memory)
- Memory contents may be accessed on a word basis only (no doubly subscripted memory references)
- Contents must be copied to a register before bit access

```
reg [15:0] mymem [0:1023];
reg [15:0] word;
reg        thebit;
             .
             .
             .
word = mymem[100];
thebit = word[12];
```

JOHN TRAMEL Mar 8, 2015, 3:11 PM

```
// reading memory

always @(*)
  word = mymem[address];   // where address is 0 .. 1023


// writing memory

always @(posedge clk)
  if (MemWrite) mymem[address] <= detain;
```

Memories in Verilog (QuickVerilog)

Initializing Memories

- Power on

```
initial
        for (i=0; i<1024; i=i+1)
                mymem[i] = 16'b0;          //fill w/ 0's
```

- Load contents from a text file

```
$readmemh("memfileh", mymem); //hex-ASCII file
                -or-
        $readmemb("memfileb", mymem); //bin-ASCII file
```

- Memory content files are always text files (ASCII)
- Memory contents
  - Hex: $readmemh
  - Binary: $readmemb
- Address references are always Hex
- Loading starts at the first memory address and proceeds until "@hexad" encountered
  - @1A // jump load pointer to address 1A

Memories in Verilog (QuickVerilog)

Sample Files

$readmemh

```
1A    2C    01
32    15    29
@10
14    17    26
@120
32    15    79
61    3B    CD
@3FD
FF    FF    FF
```

$readmemb

```
0001_1010 0010_1100 0000_0001
0011_0010 0001_0101 0010_1001
@10
0001_0100 0001_0111 0010_0110
@120
0011_0010 0001_0101 0111_1001
0110_0001 0011_1011 1100_1101
@3FD
1111_1111 1111_1111 1111_1111
```

Test Bench

```
`timescale 1ns/1ns

module TransTest_tb;

integer    i;
reg  [15:0] memory [0:511];
reg   [2:0] select;
reg   [7:0] datain;
reg  [11:0] expected;
reg         clk;
reg         rstb;
reg   [7:0] port_id;
reg   [7:0] port_out;
reg         read_strobe;
reg         write_strobe;
reg         bit8;
reg         parity_en;
reg         odd_n_even;
reg         done;
wire [10:0] shift_reg;

// instantiate the device under test

transmit transmit
            (
                .clk(clk),
                .rstb(rstb),
                .port_id(port_id),
                .port_out(port_out),
                .read_strobe(read_strobe),
                .write_strobe(write_strobe),
                .tx(tx),
                .shift(1'b0),

                .bit8(bit8),
                .parity_en(parity_en),
                .odd_n_even(odd_n_even),
                .shift_reg(shift_reg),
                .done(done)
            );

// generate the system clock

always #10 clk = ~clk;
```

Test Bench

```
// begin the verification sequence

initial begin

   $readmemh("vectors.txt",memory);

         clk = 0;
         rstb = 0;
         port_id = 0;
         port_out = 0;
         read_strobe = 0;
         write_strobe = 0;
         bit8 = 0;
         parity_en = 0;
         odd_n_even = 0;
         done = 0;

         #100
         rstb = 1;

      for (i=0; i < 384; i=i+3)
        begin
// grab stimulus from memory
        select = memory[i];
        datain = memory[i+1];
        expected = memory[i+2] & 11'h7FF;

        // assign stimulus
        {bit8,parity_en,odd_n_even} = select;

        // write the byte
        writeTrans(datain);

        // check the result
        checkTrans(expected);

        end
      end
```

Test Bench

```
// tasks to support the verification effort

// this task will program {eight, parity_en, oddhevenl} and write the data byte

task writeTrans;

input [7:0] writeData;

begin
  @(negedge clk)
  port_out = writeData;
  port_id = 8'h01;
  @(negedge clk)
  write_strobe = 1;
  @(negedge clk)
  write_strobe = 0;
 end
 endtask

 // this task will read the contents of the shift register and compare to expected

 task checkTrans;

 input [10:0] goodData;

 begin
   repeat (2) @(negedge clk) ;
        if (shift_reg != goodData)
           $display("Select %h Datain %h Expected %h Actual %h", select, datain, goodData, shift_reg);
        else
           $display("Select %h Datain %h Matched  %h Actual %h", select, datain, goodData, shift_reg);

        @(negedge clk)
        done = 1;
        @(negedge clk)
        done = 0;
end

endtask

endmodule
~
~
```