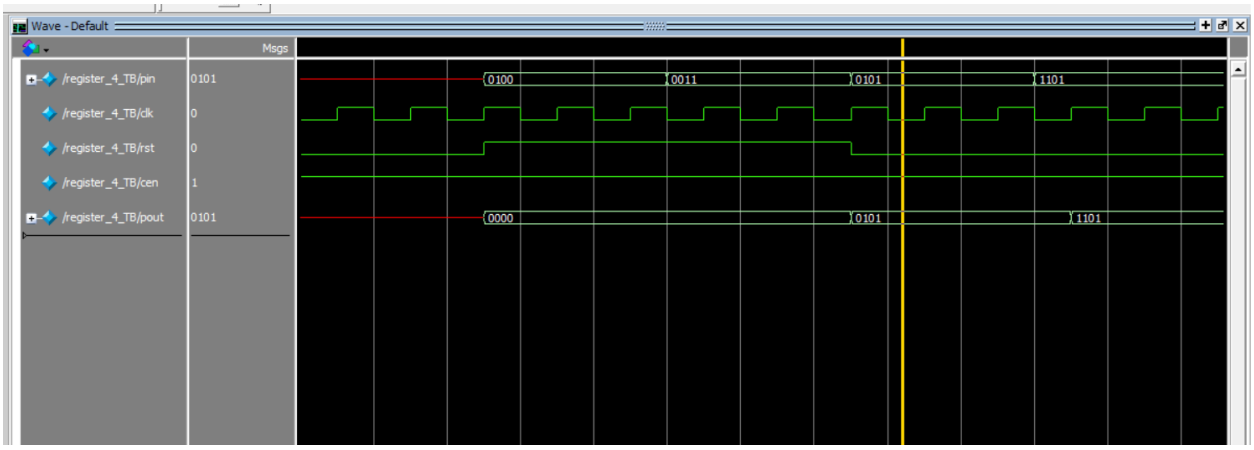
Amir Moumeni Zadeh – SID: 810101529 – CA#4 Report

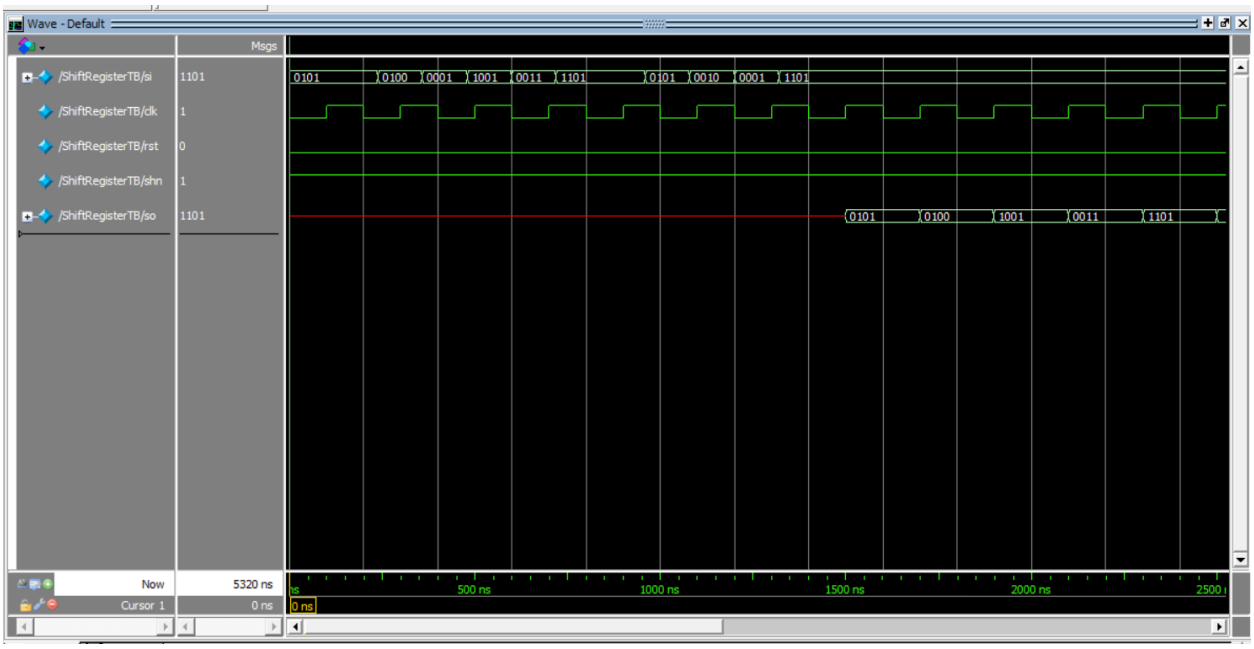


At positive edge of clock, when *rst* is *0* and *cen* is *1,* the value of *pin* goes into *pout.*

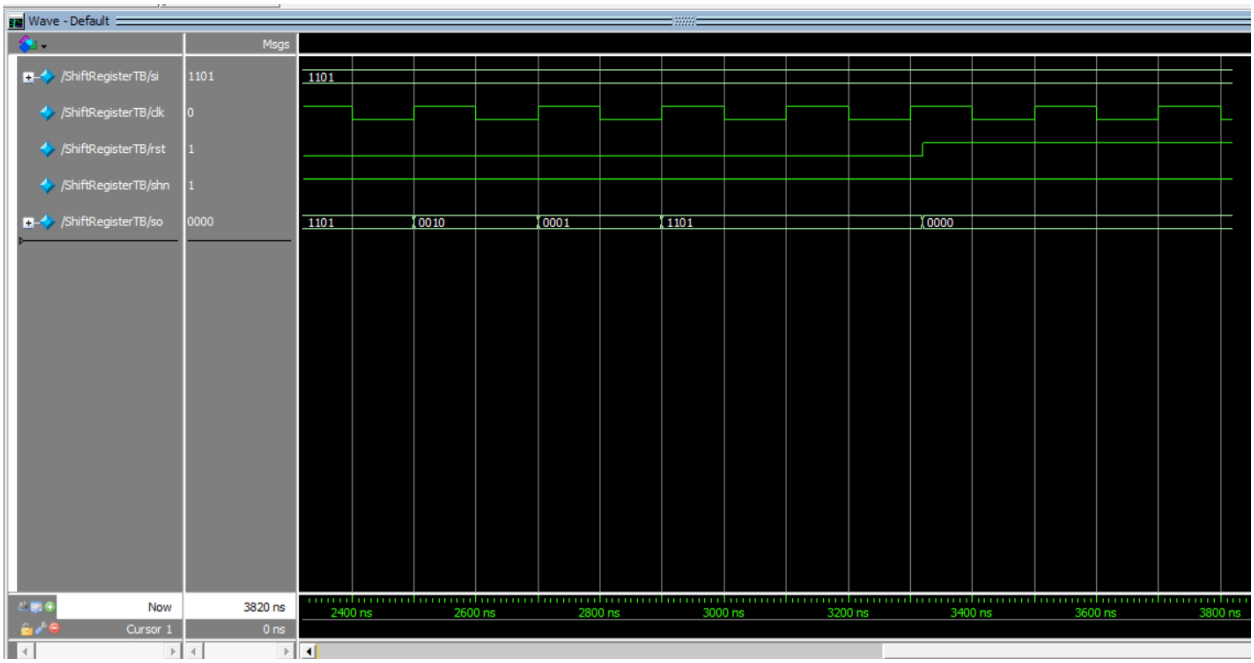
- When *rst* is active, *pout* gets 4'b0 no matter what is the input.
- When *cen* is inactive *pout* gets previous value of *pout.*

```
1   `timescale 1ns/1ns
2   module register_4(input clk, rst, cen, input [3:0] pin, output reg [3:0] pout);
3           always @(posedge clk, posedge rst) begin
4                   if (rst) pout <= 4'b0;
5                   else if (cen) pout <= pin;
6                   else pout <= pout;
7           end
8   endmodule
9
```

A shift-register is going to be designed.



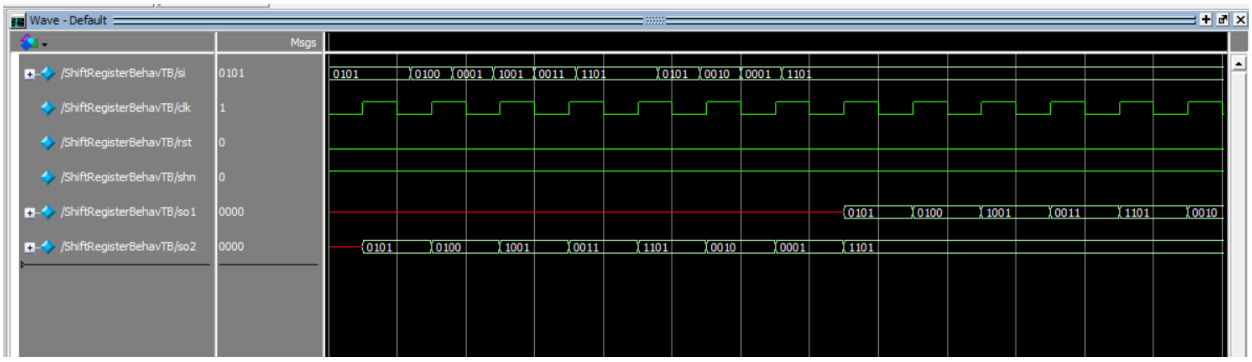
si gets in the data structure and shifts to right one place on every clock. So after 8 posedge of clock *si* appears on output which is the last character of data struture.



- When *rst* becomes 1, the output gets 0.
- When *shn* becomes inactive, the structure stops shifting as long as it becomes active again.

```
28  module ShiftRegister (input [3:0] si , input clk , rst, shn, output [3:0] so);
29          reg  [3:0] shift_reg [0:7] ;
30          genvar i;
31          generate for (i=0 ; i<8 ; i=i+1) begin : shift_stages
32                  if (i==0) begin
33                  register_4 shift_stage (clk , rst , shn , si , shift_reg[i]);
34                          end
35                  else begin
36                  register_4 shift_stage (clk , rst , shn , shift_reg [i-1] , shift_reg[i]);
37                          end
38          end
39  endgenerate
40  assign so = shift_reg[7];
41  endmodule
```

In part 3, we implement the same function using behavioural coding. In part a, we use a Non-blocking structure. It means that the input data shifts to the next stage on every clock. But in blocking structure immediately after data is on first stage, all stages get the same data which is much faster than the first one. About 8 clock cycles faster.



```
module ShiftRegNonBlock (input [3:0] si , input clk , rst, shn, output [3:0] so);
        reg  [3:0] shift_reg [0:7] ;
        always@(posedge clk , posedge rst) begin
        int i;
        for (i=0 ; i<8 ; i=i+1) begin : shift_stages
        if (rst)
                shift_reg[i] <= 4'b0;
        else if (i==0)
                if (shn)
                shift_reg[i] <= si;
                else shift_reg[i] <= shift_reg[i];
        else if (shn)
                shift_reg[i] <= shift_reg[i-1];
        else shift_reg[i] <= shift_reg[i];
end
end
assign so = shift_reg[7];
endmodule

module ShiftRegBlock (input [3:0] si , input clk , rst, shn, output [3:0] so);
        reg  [3:0] shift_reg [0:7] ;
        always@(posedge clk , posedge rst) begin
        int i;
        for (i=0 ; i<8 ; i=i+1) begin : shift_stages
        if (rst)
                for (i=0 ; i<8 ; i=i+1) begin
                        shift_reg[i] <= 4'b0;
                                        end
        else if (i==0)
                if (shn)
                for (i=0 ; i<8 ; i=i+1) begin
                        shift_reg[i] <= si;
                                        end
                else for (i=0 ; i<8 ; i=i+1) begin
                        shift_reg[i] <= shift_reg[i];
                                        end
        else if (shn)
                for (i=0 ; i<8 ; i=i+1) begin
                        shift_reg[i] <= shift_reg[i-1];
                                        end
        else for (i=0 ; i<8 ; i=i+1) begin
                        shift_reg[i] <= shift_reg[i];
                                        end
end
end
assign so = shift_reg[7];
endmodule
```