DESIGNING SPI INTERFACE FOR SRAM

Meera Ramesh

Dept. of Computer Engineering

Charles W. Davidson College of Engineering

San Jose State University

Sai Navya Panditi

Dept. of Computer Engineering

Charles W. Davidson College of Engineering

San Jose State University

***Abstract*— The project report is regarding the design and implementation of an SPI interface for a SRAM. The implementation involves transmitter and receiver architecture for SPI interface. This report also focuses on the block diagram and counter-decoder representation of controller unit. The whole timing diagram is achieved through simulation which is done using the Verilog Hardware Description language (HDL).**

I. INTRODUCTION

This project is regarding the design and implementation of SPI interface for an SRAM module with an integrated transmitter and receiver architecture as per the given sequential logic circuit and implementation of the design in Verilog.

The designing starts with collecting the required logic blocks and by setting a data flow in line with the required specifications.

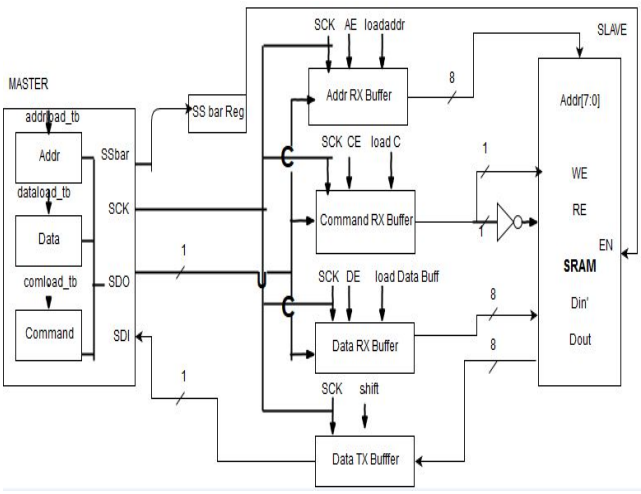
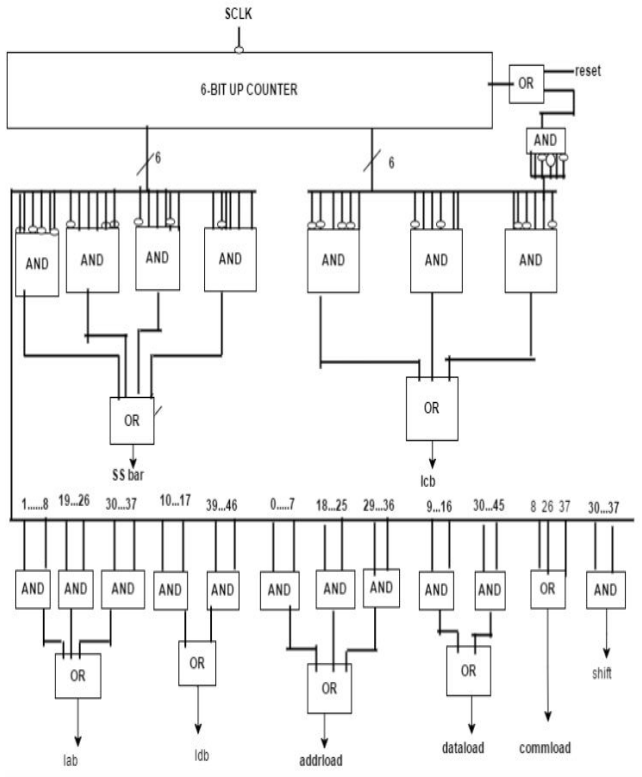


Fig 1: Data Path Block Diagram

The next step involves constructing the timing diagram to indicate the precise movements of the data from a logic block to a different logic block. And any modification within the logic block will be recorded within the respective timing diagram.

The integration of a controller into the data-path brings forth control signals which governs the flow of the data. By these signals, buffers will be enabled and data is transferred.



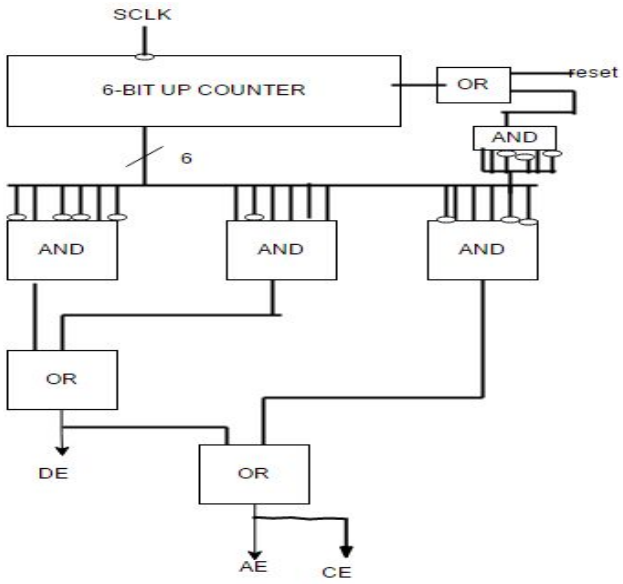


Fig 2: Counter-decoder representation of the controller unit

The above figure indicates the Counter-decoder representation of the controller unit. Controller includes six bit up-counter and decoders to get address, data and command enable signals to load data into the address buffer, data buffer and command buffer. The counters work at the negative edge of clock and are used to generate signals to load data into the address buffer,command buffer and data buffer. They also generate signals which enable providing address, data and command from the buffers to the SRAM. Additionally, counter has a reset input signal, thus once reset signal is received by counter, the counter is reset.

II. SIMULATION

In the simulation part there are three ways in which data transfer takes place. First is a write transfer, then a read transfer and again a write transfer. In the below section, we have explained what happens during a write transfer and a read transfer.

Write transfer

Address, data and command are transferred from the Test bench via SDO to the address, data and command buffers when the control signal is provided by the controller and when ssbar is low. The address followed by the command, then data will be transferred via SDO.

The address, data and command are provided from these buffers to the SRAM at the positive edge of SCK and the data is written to the SRAM.

Read Transfer:

Address and command are transferred from the Test bench via SDO to the address and command buffers when the control signals are provided by the controller and when ssbar is low. The address followed by the command will be transferred via SDO.

The address and command are provided from these buffers to the SRAM at the positive edge of SCK and the SRAM provides the data at the next positive edge of SCK. This data is saved in a 8-bit shift register. In the following negative edge the shift register starts transferring data 1 bit at a time to the SDI of the Master.

For write or read transfer, to transfer address, data and command to the buffers from the test bench, shift registers are employed. These shift registers send the data to SDO. They shift the data to the left because MSB is transferred first from the SDO.

Icarus simulator was used to design the modules and GTKwave was used to view the timing diagram.

MODULES

A. Controller

The functions of the controller can be described with the help of Counter-decoder representation of the controller unit. The 6-bit counter implicitly receives the reset signal in cycle 50. The logic diagram with gates was decoded to obtain the expressions. Given below are the expressions which are used to obtain the control signals.

Expression for counter-decoder scheme at negative edge of the clock

```
ssbar <= (~a[5]&a[4]&~a[3]&~a[2]&a[1]&~a[0]) |
(~a[5]&a[4]&a[3]&a[2]&~a[1]&~a[0])           |
(~a[5]&a[4]&a[3]&a[2]&~a[1]&a[0])            |
(a[5]&~a[4]&a[3]&a[2]&a[1]&a[0]);

lcb <= (~a[5]&~a[4]&a[3]&~a[2]&~a[1]&a[0]) |
(~a[5]&a[4]&a[3]&~a[2]&a[1]&a[0])          |
```

```
(a[5]&~a[4]&~a[3]&a[2]&a[1]&~a[0]);

dataload <= (~a[5]&~a[4]&~a[3]&~a[2]&~a[1]&~a[0])
| (~a[5]&a[4]&~a[3]&~a[2]&a[1]&~a[0]);

ae
<=(~temp[5]&temp[4]&~temp[3]&~temp[2]&temp[1]
&~temp[0])
|(temp[5]&~temp[4]&temp[3]&temp[2]&temp[1]&tem
p[0])                                          |
(~temp[5]&temp[4]&temp[3]&temp[2]&~temp[1]&~te
mp[0]);

ce
<=(~temp[5]&temp[4]&~temp[3]&~temp[2]&temp[1]
&~temp[0])
|(temp[5]&~temp[4]&temp[3]&temp[2]&temp[1]&tem
p[0])                                          |
(~temp[5]&temp[4]&temp[3]&temp[2]&~temp[1]&~te
mp[0]);

de
<=(~temp[5]&temp[4]&~temp[3]&~temp[2]&temp[1]
&~temp[0])                                     |
(temp[5]&~temp[4]&temp[3]&temp[2]&temp[1]&tem
p[0]);

resetext <= (a[5] & a[4] & ~a[3] & ~a[2] & a[1] &
~a[0]);
```

B. SPI Interface

1. **Shift Register**

There are two shift registers employed to shift the address and data to the left one bit at a time to the buffers via the SDO port. Address shift register is enabled when the addrload signal goes high and data shift register is enabled when the dataload signal goes high.

2. **Receive Buffer**

There are three receive buffers. addrbuff, databuff and commbuff which are used to receive the data 1 bit at a time from SDO. When the ae, de and ce signal goes high , the addrbuff and databuff transmit 8 bit data to the SRAM while the commbuff transmits 1 bit command to the SRAM. When the commbuff has a value 1, the command to the SRAM is taken as write and read otherwise.

3. **Transmit Buffer**

Dout is a transmit buffer which works as a left shift register. It receives 8 bit data from the SRAM. When the shift signal goes high it shifts 1 bit of data to the SDI of the Master.

C. SRAM

SRAM is configured as a 256X8 register with a unidirectional port. It works at the positive edge of the clock. It should receive address, data and command at a positive edge of the clock at the same time from addrbuff , databuff and commbuff if data is to be written to the SRAM. If data is to be read from the SRAM, the address and command are provided from addrbuff and commbuff at a positive edge and data is output from the SRAM to dout at the next positive edge of clock.

III. VERIFICATION

Observations

To verify the functionality of the code for SPI interface to read or write data from the SRAM using a controller, test bench was written in Verilog.

Control signals

- **addrload**, **dataload** and **commload** signals are provided from the controller to the test bench(master). These signals determine when address,data and command should be loaded into the registers, which will be sent over SDO.
- **ssbar** is low implies the data is sent out from SDO and when ssbar is high the SRAM is enabled. ssbar going high is taken as the enable signal for the SRAM.
- **lab** is high implies data is sent from SDO to the address buffer.
- **lcb** is high implies data is sent from SDO to the command buffer.
- **ldb** is high implies data is sent from SDO to the data buffer.
- **ae** , **de** and **ce** are high implies the data from the data buffer is written to the address specified by the address buffer. Similarly when ae and ce go high the data is read from the SRAM at address given by the address buffer to the dout register.
- **shift** goes high implies the data from dout register is given 1 bit at a time to the SDI of the master.

- **Commandbuf** register is 1 for write and 0 for read.

There are three transfers for verification. First Master **writes** a data to the SRAM (Slave). Second, master **reads** a data from SRAM. Third, master again **writes** data to the SRAM. The detailed explanation for each of these three phases is explained below and can be verified from the timing diagram.

➢ **Write**

Address is sent out from SDO of the master from cycles 1 to 8 when lab is high. Command is sent in cycle 9 when lcb is high. Data is sent from cycles 10 to 17 when ldb is high. In the next positive edge (cycle 19) when ssbar is high, the SRAM is enabled and the address, command to write and data are given parallely from the buffers to the SRAM. The data 35 is written to the SRAM at address 63 in cycle 19 at the positive edge.

➢ **Read**

Address is sent out from SDO of the master from cycles 19 to 26 when lab is high. Command is sent in cycle 27 when lcb is high. In the next positive edge (28) when ssbar is high, the SRAM is enabled and the address, command to read are given parallely from the buffers to the SRAM. The SRAM will provide data to the dout at the positive edge of 29 from address 50.

Shift goes high from cycle 30 to 37 and data from dout is sent to SDI of master.

➢ **Write**

Address is sent out from SDO of the master from cycles 30 to 37 when lab is high. Command is sent in cycle 38 when lcb is high. Data is sent from cycles 39 to 47 when ldb is high. In the next positive edge (cycle 48) when ssbar is high, the SRAM is enabled and the address, command to write and data are given parallely from the buffers to the SRAM. The data 60 is written to the SRAM at address 50 in cycle 48 at the positive edge.

SPI can transmit data via SDO at the negative edge and can sample the data it receives via SDI in the positive edge. We can make this observation from cycle 30 to cycle 37, where address is sent out via SDO of the master at the negative edge and it receives data at the SDI of the master in the negative edge.

Given below are the observation tables corresponding to the timing diagram and design from cycles 1 to 49.

Cycle	1	2	3	4	5	6	7	8
sdoM	0	0	1	1	1	1	1	1
sdoS	x	x	x	x	x	x	x	x
ssbar	0	0	0	0	0	0	0	0
dout	x	x	x	x	x	x	x	x
shift	0	0	0	0	0	0	0	0
addrbuf	x	x	x	x	x	x	x	x
databuf	x	x	x	x	x	x	x	x
commbuf	x	x	x	x	x	x	x	x
lab	1	1	1	1	1	1	1	1
ldb	0	0	0	0	0	0	0	0
lcb	0	0	0	0	0	0	0	0
ae	0	0	0	0	0	0	0	0
de	0	0	0	0	0	0	0	0
ce	0	0	0	0	0	0	0	0
addrReg_tb	126	252	248	240	224	192	128	0
commReg_tb	1	1	1	1	1	1	1	1
dataReg_tb	35	35	35	35	35	35	35	35
addrload_tb	1	1	1	1	1	1	1	1
commload_tb	0	0	0	0	0	0	0	1
Dataload_tb	0	0	0	0	0	0	0	0

Table 1: Cycle 1 to 8

Cycle	9	10	11	12	13	14	15	16
sdoM	1	0	0	1	0	0	0	1
sdoS	x	x	x	x	x	x	x	x
ssbar	0	0	0	0	0	0	0	0
dout	x	x	x	x	x	x	x	x
shift	0	0	0	0	0	0	0	0
addrbuf	63	63	63	63	63	63	63	63
databuf	x	x	x	x	x	x	x	x
commbuf	x	1	1	1	1	1	1	1
lab	0	0	0	0	0	0	0	0
ldb	0	1	1	1	1	1	1	1
lcb	1	0	0	0	0	0	0	0
ae	0	0	0	0	0	0	0	0
de	0	0	0	0	0	0	0	0
ce	0	0	0	0	0	0	0	0

addrReg_tb	0	0	0	0	0	0	0	0
commReg_tb	0	0	0	0	0	0	0	0
dataReg_tb	35	70	140	24	48	96	192	128
addrload_tb	0	0	0	0	0	0	0	0
commload_tb	0	0	0	0	0	0	0	0
Dataload_tb	1	1	1	1	1	1	1	1

Table 2: Cycle 9 to 16

Cycle	17	18	19	20	21	22	23	24
sdoM	1	1	0	0	1	1	0	0
sdoS	x	x	x	x	x	x	x	x
ssbar	0	1	0	0	0	0	0	0
dout	x	x	x	x	x	x	x	x
shift	0	0	0	0	0	0	0	0
addrbuf	63	63	63	63	63	63	63	55
databuf	x	35	35	35	35	35	35	35
commbuf	1	1	1	1	1	1	1	1
lab	0	0	1	1	1	1	1	1
ldb	1	0	0	0	0	0	0	0
lcb	0	0	0	0	0	0	0	0
ae	0	1	0	0	0	0	0	0
de	0	1	0	0	0	0	0	0
ce	0	1	0	0	0	0	0	0
addrReg_tb	0	0	100	200	144	32	64	128
commReg_tb	0	0	0	0	0	0	0	0
dataReg_tb	0	0	0	0	0	0	0	0
addrload_tb	0	1	1	1	1	1	1	1
commload_tb	0	0	0	0	0	0	0	0
Dataload_tb	0	0	0	0	0	0	0	0

Table 3: Cycle 17 to 24

Cycle	25	26	27	28	29	30	31	32
sdoM	1	0	0	0	0	0	0	1
sdoS	x	x	x	x	x	x	0	0
ssbar	0	0	0	1	1	0	0	0
dout	x	x	x	x	50	50	100	200
shift	0	0	0	0	0	1	1	1

addrbuf	51	51	50	50	50	50	50	50
databuf	35	35	35	35	35	35	35	35
commbuf	1	1	1	0	0	0	0	0
lab	1	1	0	0	0	1	1	1
ldb	0	0	0	0	0	0	0	0
lcb	0	0	1	0	0	0	0	0
ae	0	0	0	1	0	0	0	0
de	0	0	0	0	0	0	0	0
ce	0	0	0	1	0	0	0	0
addrReg_tb	0	0	0	0	0	100	200	144
commReg_tb	0	0	1	1	1	1	1	1
dataReg_tb	0	0	0	0	0	0	0	0
addrload_tb	1	0	0	0	1	1	1	1
commload_tb	0	1	0	0	0	0	0	0
Dataload_tb	0	0	0	0	0	0	0	0

Table 4: Cycle 25 to 32

Cycle	33	34	35	36	37	38	39	40
sdoM	1	0	0	1	0	1	0	0
sdoS	1	1	0	0	1	0	0	0
ssbar	0	0	0	0	0	0	0	0
dout	144	32	64	128	0	0	0	0
shift	1	1	1	1	1	0	0	0
addrbuf	50	50	50	50	50	50	50	50
databuf	35	35	35	35	35	35	35	35
commbuf	0	0	0	0	0	0	1	1
lab	1	1	1	1	1	0	0	0
ldb	0	0	0	0	0	0	1	1
lcb	0	0	0	0	0	1	0	0
ae	0	0	0	0	0	0	0	0
de	0	0	0	0	0	0	0	0l
ce	0	0	0	0	0	0	0	0
addrReg_tb	32	64	128	0	0	0	0	0
commReg_tb	1	1	1	1	1	0	0	0
dataReg_tb	0	0	0	0	0	0	120	240
addrload_tb	1	1	1	1	0	0	0	0
commload_tb	0	0	0	0	1	0	0	0
Dataload_tb	0	0	0	0	0	1	1	1

Table 5: Cycle 33 to 40

Cycle	41	42	43	44	45	46	47	48
sdoM	1	1	1	1	0	0	0	0
sdoS	0	0	0	0	0	0	0	0
ssbar	0	0	0	0	0	0	1	1
dout	0	0	0	0	0	0	0	0
shift	0	0	0	0	0	0	0	0
addrbuf	50	50	50	50	50	50	50	50
databuf	35	35	51	59	63	61	60	60
commbuf	1	1	1	1	1	1	1	1
lab	0	0	0	0	0	0	0	0
ldb	1	1	1	1	1	1	0	0
lcb	0	0	0	0	0	0	0	0
ae	0	0	0	0	0	0	1	1
de	0	0	0	0	0	0	1	1
ce	0	0	0	0	0	0	1	1
addrReg_tb	0	0	0	0	0	0	0	0
commReg_tb	0	0	0	0	0	0	0	0
dataReg_tb	224	192	128	0	0	0	0	0
addrload_tb	0	0	0	0	0	0	0	0
commload_tb	0	0	0	0	0	0	0	0
Dataload_tb	1	1	1	1	1	0	0	0

Table 6: Cycle 41 to 48

IV. CONCLUSION

We have successfully implemented the project. This project is been great learning experience and help us in learning the practical approach and application of Verilog HDL, designing and implementing SPI interface.

Difficulty faced

- As new to Verilog we need to start from basic, learning Verilog language and working of simulators.
- Faced some problem while using proper syntax for the operation and while instantiating modules and testing modules.
- Faced issues while designing the SPI interface. We had to make sure the controller sends the control signals at the right time for the desired operations to take place.
- Faced issues while sending address and data from the master via the 1 bit SDO register. We resolved it by using address decoders and data decoders.

ACKNOWLEDGMENT

Heartiest thanks to Dr. Ahmet Bindal (Professor, Computer Engineering Dept, SJSU). His constant support and motivation was the key to the success of this project.

REFERENCES

[1] Dr.Ahmet A Bindal, Fundamentals of Computer Design – A Micro-Architectural Analysis and Design Study, 5th ed.

[2] Verilog Notes- Dr.Ahmet A Bindal

External Links

- Asic-World - Extensive free online tutorial with many examples.
- GTK Wave Tutorial – Application tutorial.
- Icarus Verilog Tutorial – Application tutorial.

APPENDIX

1)Main Module Code

```
 module
spi(sdoS,clock,reset,sdoM,addrload,dataload,comload
);

  input reset,clock;
  output addrload;
  output dataload;
  output comload;

  input sdoM;

  output sdoS;

  reg ssbar,lcb,dataload,comload,addrload,shift,
  lab,ldb,de,ae,ce,commandbuf;
  reg [5:0] a;
  reg [5:0] temp;
  reg [7:0] addrbuf;
  reg [7:0] databuf;
  reg [2:0] addrcount;
  reg [2:0] datacount;
  reg [2:0] outputcount;
  reg [7:0] outputreg;
  reg [7:0] sram [255:0];

  reg [7:0] addr;
  reg comm;
  reg [7:0] din;
  reg [7:0] dout;
  reg regload;
  reg regdataload;
  reg resetext;
  reg sdoS;
  reg[7:0] i;

  initial
  begin
        a <= -1;
        temp <= 0;
        de <= 0;
        ae <= 0;
        ce <= 0;
addrcount <= 7;
datacount <= 7;
outputcount <= 7;
resetext <= 0;
sram[31] <= 31;
sram[50] <= 50;
end

always @(negedge clock)
begin
a=a+1;
if(reset==0 && resetext==0)
begin
ssbar <=
(~a[5]&a[4]&~a[3]&~a[2]&a[1]&~a[0]) |
(~a[5]&a[4]&a[3]&a[2]&~a[1]&~a[0]) |
(~a[5]&a[4]&a[3]&a[2]&~a[1]&a[0]) |
(a[5]&~a[4]&a[3]&a[2]&a[1]&a[0]);

lcb <=
(~a[5]&~a[4]&a[3]&~a[2]&~a[1]&a[0]) |
(~a[5]&a[4]&a[3]&~a[2]&a[1]&a[0]) |
(a[5]&~a[4]&~a[3]&a[2]&a[1]&~a[0]);

dataload <=
(~a[5]&~a[4]&~a[3]&~a[2]&~a[1]&~a[0]) |
(~a[5]&a[4]&~a[3]&~a[2]&a[1]&~a[0]);

ae <=
(~temp[5]&temp[4]&~temp[3]&~temp[2]&te
mp[1]&~temp[0])
|(temp[5]&~temp[4]&temp[3]&temp[2]&tem
p[1]&temp[0]) |
(~temp[5]&temp[4]&temp[3]&temp[2]&~te
mp[1]&~temp[0]);

ce <=
(~temp[5]&temp[4]&~temp[3]&~temp[2]&te
mp[1]&~temp[0])
|(temp[5]&~temp[4]&temp[3]&temp[2]&tem
p[1]&temp[0]) |
(~temp[5]&temp[4]&temp[3]&temp[2]&~te
mp[1]&~temp[0]);

de <=
(~temp[5]&temp[4]&~temp[3]&~temp[2]&te
mp[1]&~temp[0]) |
(temp[5]&~temp[4]&temp[3]&temp[2]&tem
p[1]&temp[0]);

resetext <= (a[5] & a[4] & ~a[3] & ~a[2] &
a[1] & ~a[0]);
```

```
        if((a>=1 && a<=8) || (a>=19 && a<=26) ||
        (a>=30 && a<=37)) lab <= 1;
        else lab<=0;
        if((a>=0 && a<=7) || (a>=18 && a<=25) ||
        (a>=29 && a<=36)) addrload <= 1;
        else addrload<=0;

        comload <=
        (~a[5]&~a[4]&a[3]&~a[2]&~a[1]&~a[0]) ||
        (~a[5]&a[4]&a[3]&~a[2]&a[1]&~a[0]) |
        (a[5]&~a[4]&~a[3]&a[2]&~a[1]&a[0]);
        /*8,26,37 */

        if((a>=10 && a<=17) || (a>=39 && a<=46))
        ldb <= 1;
        else ldb <= 0;

        if((a>=9 && a<=16) || (a>=38 && a<=45))
        dataload <= 1;
        else dataload <= 0;

        if(a>=30 && a<=37) shift <=1;
        else shift<=0;

        if(lab==1)
        begin
        addrbuf[addrcount] <= sdoM;
        addrcount <= addrcount - 1;
        end

        if(ldb==1)
        begin
        databuf[datacount] <= sdoM;
        datacount <= datacount - 1;
        end

        if(lcb==1)
        begin
        commandbuf <= sdoM;
        end

        if(shift==1)
        begin
        sdoS <= dout[7];
        dout <= dout << 1;
        end
    end
end

always @(posedge clock)
begin
temp=temp+1;
if(reset==0 && resetext==0)
    begin
    /* Reduced expression for counter-decoder
    scheme*/
    if(ssbar==1)
    begin

    if(ae==1 && ce==1)
    begin
    if(commandbuf==1 && de==1)
    begin
    sram[addrbuf] <= databuf;
    end
    else if(commandbuf==0)
    begin
    dout<=sram[addrbuf];
    end
    end
    end

    end
    end
    endmodule
```

2)Test bench code

```
`include "verilogHW2.v"
`timescale 1s/1s

module spitest;
reg reset_tb,clock_tb;
wire addrload_tb;
wire dataload_tb;
wire commload_tb;
reg [7:0] addrReg_tb;
reg [7:0] dataReg_tb;
reg commreg_tb;
wire sdiM;
reg sdo;
reg [3:0] count;
reg [3:0] countD;
spi i1
(.reset(reset_tb),.clock(clock_tb),.sdoM(sdo),.sdoS(sd
iM),.addrload(addrload_tb),.comload(commload_tb),.
dataload(dataload_tb));

initial
begin
$dumpfile("spi.vcd");
$dumpvars(0,spitest);
clock_tb <= 0;
reset_tb <=0;
addrReg_tb <= 63;
```

```
dataReg_tb <= 35;
commreg_tb <= 1;
count <= 8;
countD <= 8;
end

always #1 clock_tb = ~clock_tb;

always @(negedge clock_tb)
begin

if(addrload_tb==1)
begin
sdo <= addrReg_tb[7];
addrReg_tb <= addrReg_tb << 1;
count <= count - 1;
if(count==0)
begin
addrReg_tb <= addrReg_tb + 100;
count <= 7;
end
end

if(dataload_tb==1)
begin
sdo <= dataReg_tb[7];
dataReg_tb <= dataReg_tb << 1;
countD <= countD - 1;
if(countD==0)
begin
dataReg_tb <= dataReg_tb + 120;
countD <= 8;
end
end

if(commload_tb==1)
begin
sdo <= commreg_tb;
commreg_tb <= ~commreg_tb;
end

end
endmodule
```

OUTPUT WAVEFORM

Fig 3. output waveform from the simulator

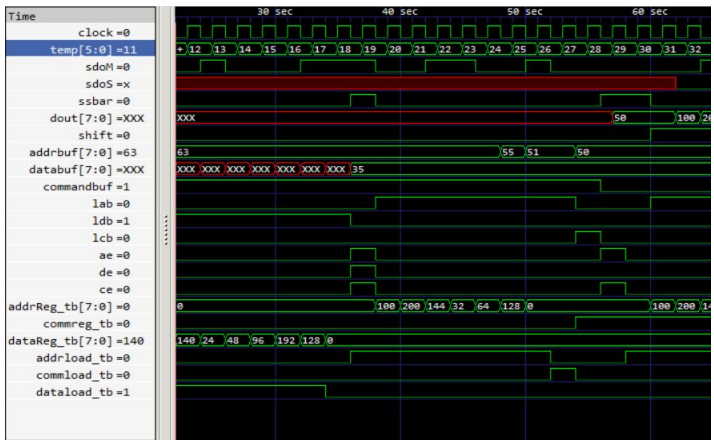


Fig 4. output waveform from the simulator

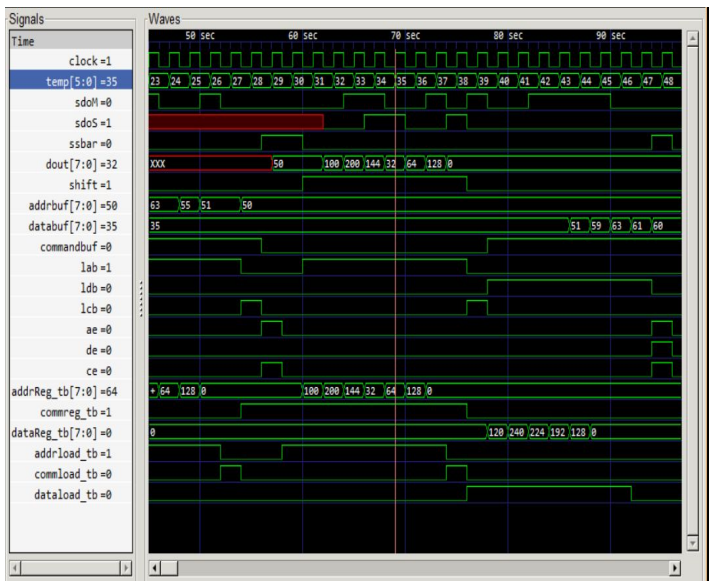


Fig 5. output waveform from the simulator