SPI Interface with SRAM

Pooja Prakashchand

SJSU Id: 010814258

SaiKruthi Vemuru

SJSU Id: 010826621

Abstract – This report demonstrates the SPI Interface for a SRAM. The SPI Interface is implemented for a slave, which reads data from the SDI port at the positive edge of the clock and sends the data out from the SDO port at the negative edge of the clock. Data-path and controller design is implemented using Hardware Description language - Verilog. We used ModelSim and Icarus tools to simulate and check the results.

1. **INTRODUCTION**

We developed an architecture that performs two main functions reading and writing data into SRAM using SPI interface. SRAM acts as a slave which receives data from the master. SPI Interface has a transmitter and receiver to transmit and receive data to and from the SRAM.

Transmitter is a parallel in and serial out shift register, which takes in the entire word coming out of SRAM and sends bit by bit to the master through the SDO port of the SPI Interface at negative edge.

Receiver is a serial in and parallel out shift register, which takes the data, bit by bit through the SDI port of SPI Interface and gives out the entire word to the SRAM.

According to the SPI protocol the master sends the address bits first followed by the one-bit command (enables SRAM for read or write operation) which is then followed by the data bits if the data is to be written to the SRAM.

Receiver is divided into three parts namely two shift registers and one buffer to store and give the address, data and command to the SRAM separately.

1. **COMPONENTS USED**

The following components are used to perform read and write operations to the SRAM:

1. *8-bit shift registers* – used by transmitter and receiver modules
2. *4-bit down counters* – used by the shift registers to send or receive data
3. *D-Flip flop* – used to store the address till it is provided to the SRAM
4. *SRAM memory* – used to write or read data
5. *5-bit up counter* – used by the controller to generate states
6. *Controller* – used to generate control signals which activates the respective shift registers (transmitter register or receiver register) and counters in accordance with the operation to be performed i.e. read or write operation.

The design is implemented using Verilog and the simulation is done using Icarus and ModelSim with the test bench module.

1. **PROJECT DESCRIPTION**

The figure below shows the complete architecture of SPI interface with SRAM.

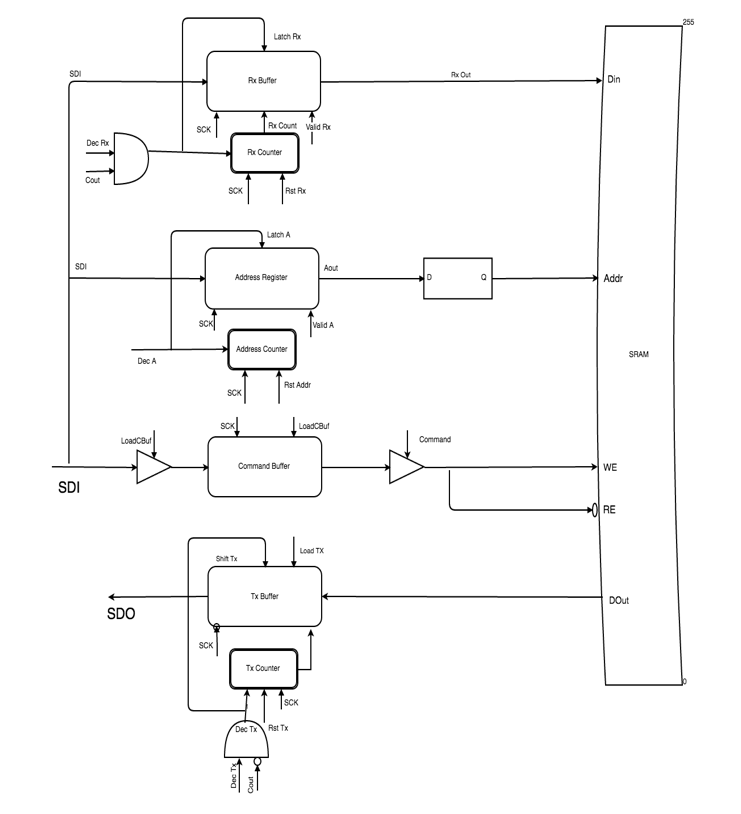


Figure 1: SPI Interface with SRAM

The detailed explanation of SPI interface, transmitter, receiver, SRAM, controller, read data path and write data path are given below.

1. *SPI Interface*

Serial peripheral interface is serial bus interface which is used mainly for communication between processor and slow memory devices. The communication between master and slave takes place using four signals. There is no acknowledgement in serial communication and the master sends out data irrespective of whether slave is present or not. The master sends following three signals which are received by slave:

* SS- active low slave select signal to activate slave and initiate data transfer process
* SCK- SPI clock sent to slave to synchronize the slave with master
* MOSI-Master out slave in which transmits the actual address, command and data bits in case of write operation.

The master receives the following from the slave:

* MISO- Master in slave out receive the data bits sent by slave in case of write operation.

SPI operates in four different modes which decides the clock edge at which the data is sent or received by the master and slave. The architecture designed operates at mode 0 in which the master sends data at negative edge and receives data at positive edge.

In the architecture designed the address register, data register and command buffer receives MOSI signal at positive edge of SCK. Transmitter register sends MISO at negative edge of SCK.

1. *Receiver*

Receiver consists of 8-bit serial in ad parallel out shift register and 4-bit down counter. Address bits and Data bits are latched separately using two shift registers, whose output is then given to the SRAM. The operation of the receiver is explained below:

* The shift register starts latching the bits once the latch signal goes high.
* The data is given out in the cycle where the valid signal goes high.
* The receiver uses a down counter to keep track of number of bits latched.
* The counter is reset and loaded with the value equal to the maximum number of bits to be latched which is 8 in our case as the word size is 8-bit.
* The counter starts decrementing once the decrement signal goes high.
* The latch signal goes high for the same number of cycles as the decrement signal.

First the 8 address bits are sent and hence the address register and the counter gets activated for 8 cycles. Next the command bit is sent which is stored in the command buffer. The tri-state buffer in the path is activated when the input should be given to the command buffer. If it is a write operation, then the command bit is followed by the 8 data bits and hence the data register and the counter gets activated for the next 8 cycles.

Hence the output of the address register, command buffer and data register are given to the SRAM in case of write operation. In case of read operation, the output of address register and command buffer are given to SRAM.

1. *Transmitter*

Transmitter circuit consists of 8-bit parallel in and serial out shift register and 4 bit down counter to keep track of number of bits shifted out of the transmitter. The transmitter operation is explained in detail below:

* The 8-bit data is loaded into the shift register once the load signal goes high.
* The transmitter starts shifting data from the next cycle if the shift signal goes high in the next cycle.
* The shift signal stays high for 8 cycles to shift out the 8 data bits.
* The decrement signal and shift signal stays high for same number of cycles.

After the address and command are given to SRAM, data is obtained at the output port which is then shifted out bit by bit using transmitter. Transmitter circuit operates at negative edge of the clock.

1. *SRAM*

We have configured 8x256 SRAM with an 8-bit address. The SRAM has the following inputs:

* Enable port: Slave select signal is given as input to this port.
* Address: The address coming out from the address register is given as input to this port.
* Data in: The data coming out from the receiver register is given as input to this port.
* RE: The command buffer output is given to the not gate. The output of the not gate is given as input to this port.
* WE: The command buffer output is given as input to this port.

The SRAM has one output port through which data comes out in case of read operation.

1. *Read Data path*

As shown in figure, the read data path consists of address register, command buffer, SRAM and transmitter.

The address and command are given to the SRAM which gives data out. The data is transmitted out to the master bit by bit through the transmitter.

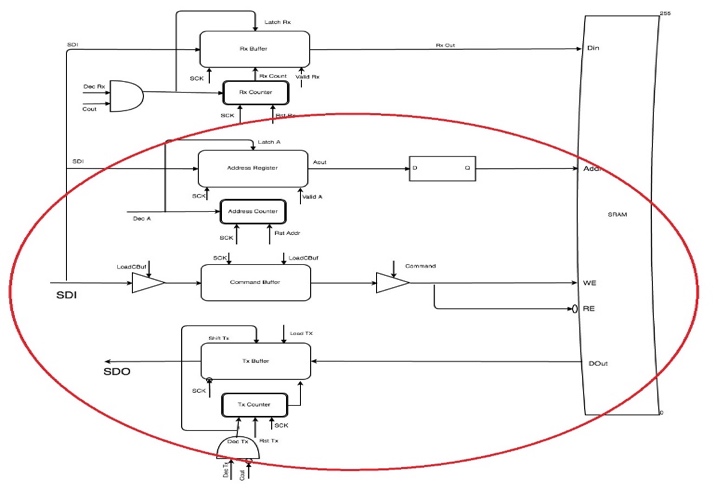
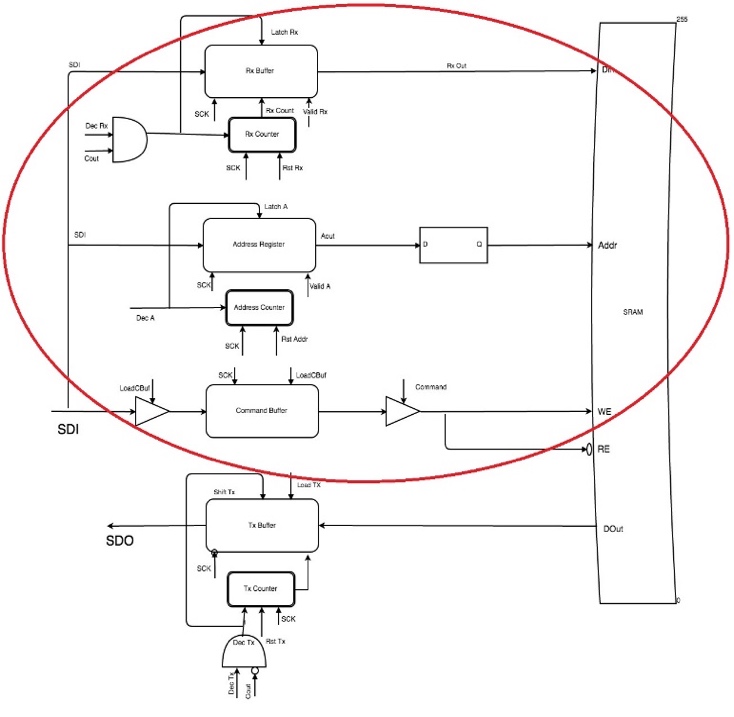


Figure 2: Shows the transmitter which is activated during read operation

1. *Write Data path*

As shown in the figure, the write data path consists of address register, command buffer, receiver register and SRAM.

The address, command and data are given to SRAM.

Figure 3: Shows the Receiver which is activated during write operation

1. Controller

The controller is designed using 5-bit up counter and decoders. The controller is designed in such a way that it activates the shift registers and counters according to the input coming in from the SDI port.

The 5-bit up counter starts incrementing once the active low slave select signal goes low.

* The controller stays in idle state until the active low slave select signal is high and the reset 5-bit counter signal is high.
* The controller transitions from idle state to Count down state as the SS signal goes low.
* It stays in the count down state until the Address count is greater than zero and decrement and latch signals are high.
* The controller moves to the next state as the valid signal goes high and shift register gives the address as the output. The load buffer signal also goes high in this cycle to store the command bit in the buffer.
* The command buffer output is checked and if it is one, it moves to count down receiver buffer state. It stays in this state till the decrement and latch signals are high.
* It moves to the next state where valid output signal goes high and gives the data to the SRAM as input.
* If the command buffer output is zero, it moves to Dout state. LoadTx buffer signal buffer goes high in this cycle and the data output from the SRAM is loaded into transmitter register.
* The transmitter goes to empty TX buffer state once the Shift and decrement signal goes high and shifts the data bit by bit.
* After the read or write operation, the counter is reset and slave select signal is made high.

Before starting any operation, the active low slave select signal and the reset signal should be made low.

The state diagram used to design the controller is as shown in the figure.

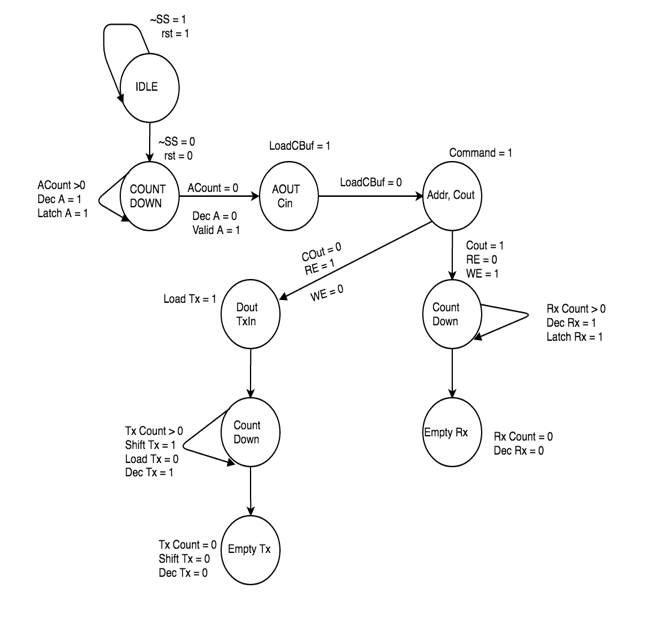


Figure 4: Represents the state diagram used to design the controller.

1. **VERIFICATION**

Verification is an important step to justify the results obtained.

We have done the verification of the design implemented using three test cases. The test cases are mentioned below in detail. We have modified the test bench according to the test case and verified the simulation output. By default, we have written 8’b0 to all locations in SRAM.

The verification results are shown in Figure.

The expected and obtained results are a match. Hence we can conclude that the architecture designed is working as expected.

|  |  |  |
| --- | --- | --- |
| Test Case 1 | Description | Expected Result |
| Read, Write, Read | Addr = 8’hfd  RE = 1  Addr = 8’hfd  WE = 1  DataIn = 8’hbd  Addr = 8’hfd  RE = 1  (Note: address, WE/RE, data is sent 1 bit at a time from SDI Port) | Data out = 8’h0  Data out = 8’hbd  (Note: data is sent out from SRAM 1 bit at a time from SDO Port) |
| Refer Figure 9, 10, 11 | | |

|  |  |  |
| --- | --- | --- |
| Test Case 2 | Description | Expected Result |
| Read, Write | Addr = 8’hfd  RE = 1  Addr = 8’hfd  WE = 1  DataIn = 8’hbd  (Note: address, WE/RE, data is sent 1 bit at a time from SDI Port) | Data out = 8’h0  (Note: data is sent out from SRAM 1 bit at a time from SDO Port) |
| Refer Figure 12, 13 | | |

|  |  |  |
| --- | --- | --- |
| Test Case 3 | Description | Expected Result |
| Write, Read | Addr = 8’hfd  WE = 1  DataIn = 8’hbd  Addr = 8’hfd  RE = 1  (Note: address, WE/RE, data is sent 1 bit at a time from SDI Port) | Data out = 8’hbd  (Note: data is sent out from SRAM 1 bit at a time from SDO Port) |
| Refer Figure 14, 15 | | |

1. **CONCLUSION**

This project demonstrated the working of SPI Interface for a SRAM. We designed the architecture, the state diagram, programmed the project in Verilog and used ModelSim and Icarus Verilog to simulate the results. This involved in-depth understanding of the working of SPI Interface how each logic works in the hardware. Trouble-shooting is one of the key aspects of the design process. Verification was achieved by cross checking the timing diagram as specified during the design process. This stabilizes the theoretical output and practical output and backtracking is easy if any errors occur.

# **Acknowledgment**

A special thanks to Dr. Ahmet Bindal (Professor, Computer Engineering Dept, SJSU). His clear explanation and guidance in making our concepts clear was the key to success of the project. Also, our TA Mr. Ramakrishna Melgiri helped us in all the queries we had and it helped us to complete the project in the given timeline.

# **References**

[1] Dr. Ahmet Bindal – Fundamentals of Computer Architecture and Design, 2016

[2] Verilog Notes- Dr. Ahmet Bindal

# **Links**

[3] <http://www.asic-world.com/>

[4] <https://www.edaplayground.com/>

[5] http://stackoverflow.com/

# **APPENDIX**

# **SOURCE CODE**

module datapath(sck, sdi, rst, lA, dA);

input sdi, sck, rst, lA, dA;

wire [3:0] ACount, TXCount, RXCount;

wire decA, rstA, decRx, sdo;

wire [7:0] Aout, D, Q, dout, rxout, ParallelIn;

wire cin, LoadCBuf, cout, latchA, latchRx;

wire sigRE, command, sigWE ;

wire ss, decTx, rstRX, LoadTx, ShiftTx, rstTx, validA;

wire in1,in2,in3,in4,in5, validRx;

counter\_5bit counter\_5bit\_Inst( .in1(in1),

.in2(in2),

.in3(in3),

.in4(in4),

.in5(in5),

.sck(sck),

.rst(rst));

controller controller\_Inst( .decA(decA),

.rstA(rstA),

.latchA(latchA),

.validA(validA),

.loadCbuf(LoadCBuf),

.command(command),

.latchRx(latchRx),

.decRx(decRx),

.rstRx(rstRX),

.loadTx(LoadTx),

.decTx(decTx),

.shiftTx(ShiftTx),

.rstTx(rstTx),

.ss(ss),

.in1(in1),

.in2(in2),

.in3(in3),

.in4(in4),

.in5(in5),

.rst(rst),

.lA(lA),

.dA(dA),

.validRx(validRx),

.cout(cout));

counterRX\_4bit counterRX\_4bit\_Inst( .RXCount(RXCount),

.sck(sck),

.decRx(decRx),

.rstRX(rstRX));

rxbuf rxbuf\_Inst( .rxout(rxout),

.sdi(sdi),

.sck(sck),

.latchrxdata(latchRx),

.RxCount(RXCount),

.validRx(validRx));

counterAddr\_4bit counterAddr\_4bit\_Inst( .ACount(ACount),

.sck(sck),

.decA(decA),

.rstA(rstA));

addreg addreg\_Inst( .Aout(Aout),

.sdi(sdi),

.sck(sck),

.latchA(latchA),

.validA(validA),

.Acount(ACount));

dflipflop dflipflop\_Inst( .Q(Q),

.D(Aout),

.sck(sck));

triStatebufLoadCommand triStatebufLoadCommand\_Inst( .cin(cin),

.sdi(sdi),

.LoadCBuf(LoadCBuf));

commandBuf commandBuf\_Inst( .cout(cout),

.sck(sck),

.cin(cin),

.loadCBuf(LoadCBuf));

readWriteSig readWriteSig\_Inst( .sigWE(sigWE),

.sigRE(sigRE),

.cout(cout),

.command(command));

sram sram\_Inst( .dout(dout),

.din(rxout),

.addr(Q),

.sck(sck),

.re(sigRE),

.we(sigWE),

.ss(ss));

counterTX\_4bit counterTX\_4bit\_Inst( .TXCount(TXCount),

.sck(sck),

.decTx(decTx),

.rstTX(rstTx));

TxBuf TxBuf\_Inst( .sdo(sdo),

.sck(sck),

.ParallelIn(dout),

.LoadTx(LoadTx),

.ShiftTx(ShiftTx),

.TxCount(TXCount));

endmodule end

if (!ShiftIn) begin

ShiftRegOut <= 0;

end

end

module counter\_5bit(in1,in2,in3,in4,in5,sck, rst);

reg [4:0] out;

output reg in1,in2,in3,in4,in5;

input sck, rst;

always @(posedge sck)

begin

if(rst)

begin

out <= 5'b00000;

in1=0;in2=0;in3=0;in4=0;in5=0;

end

else

begin

case(out)

5'b00001 : begin in1=0;in2=0;in3=0;in4=0;in5=1; end

5'b00010 : begin in1=0;in2=0;in3=0;in4=1;in5=0; end

5'b00011 : begin in1=0;in2=0;in3=0;in4=1;in5=1; end

5'b00100 : begin in1=0;in2=0;in3=1;in4=0;in5=0; end

5'b00101 : begin in1=0;in2=0;in3=1;in4=0;in5=1; end

5'b00110 : begin in1=0;in2=0;in3=1;in4=1;in5=0; end

5'b00111 : begin in1=0;in2=0;in3=1;in4=1;in5=1; end

5'b01000 : begin in1=0;in2=1;in3=0;in4=0;in5=0; end

5'b01001 : begin in1=0;in2=1;in3=0;in4=0;in5=1; end

5'b01010 : begin in1=0;in2=1;in3=0;in4=1;in5=0; end

5'b01011 : begin in1=0;in2=1;in3=0;in4=1;in5=1; end

5'b01100 : begin in1=0;in2=1;in3=1;in4=0;in5=0; end

5'b01101 : begin in1=0;in2=1;in3=1;in4=0;in5=1; end

5'b01110 : begin in1=0;in2=1;in3=1;in4=1;in5=0; end

5'b01111 : begin in1=0;in2=1;in3=1;in4=1;in5=1; end

5'b10000 : begin in1=1;in2=0;in3=0;in4=0;in5=0; end

5'b10001 : begin in1=1;in2=0;in3=0;in4=0;in5=1; end

5'b10010 : begin in1=1;in2=0;in3=0;in4=1;in5=0; end

5'b10011 : begin in1=1;in2=0;in3=0;in4=1;in5=1; end

5'b10100 : begin in1=1;in2=0;in3=1;in4=0;in5=0; end //20

5'b10101 : begin in1=1;in2=0;in3=1;in4=0;in5=1; end

5'b10110 : begin in1=1;in2=0;in3=1;in4=1;in5=0; end

5'b10111 : begin in1=1;in2=0;in3=1;in4=1;in5=1; end

5'b11000 : begin in1=1;in2=1;in3=0;in4=0;in5=0; end

5'b11001 : begin in1=1;in2=1;in3=0;in4=0;in5=1; end

5'b11010 : begin in1=1;in2=1;in3=0;in4=1;in5=0; end

5'b11011 : begin in1=1;in2=1;in3=0;in4=1;in5=1; end

5'b11100 : begin in1=1;in2=1;in3=1;in4=0;in5=0; end

5'b11101 : begin in1=1;in2=1;in3=1;in4=0;in5=1; end

5'b11110 : begin in1=1;in2=1;in3=1;in4=1;in5=0; end

5'b11111 : begin in1=1;in2=1;in3=1;in4=1;in5=1; end

default: begin in1=0;in2=0;in3=0;in4=0;in5=1; out = 5'b00001; end

endcase

out = out + 1;

end

end

endmodule

module controller(decA, rstA, latchA, validA, loadCbuf, command, latchRx, validRx , decRx, rstRx, loadTx, decTx, shiftTx, rstTx, ss, in1,in2,in3,in4,in5, rst, lA, dA, cout);

output wire decA, rstA, latchA, validA, loadCbuf, command, latchRx, decRx, rstRx, ss, validRx, loadTx, decTx, shiftTx, rstTx;

input in1,in2,in3,in4,in5, rst, lA, dA, cout;

//output wire loadTx, decTx, rstTx, shiftTx;

//loadTx, decTx, rstTx, shiftTx,

and A2(ss1,~in1,~in2,~in3,~in4,in5);

and B2(ss2,~in1,~in2,~in3,in4,~in5);

and C2(ss3,~in1,~in2,~in3,in4,in5);

and D2(ss4,~in1,~in2,in3,~in4,~in5);

and E2(ss5,~in1,~in2,in3,~in4,in5);

and F2(ss6,~in1,~in2,in3,in4,~in5);

and G2(ss7,~in1,~in2,in3,in4,in5);

and H2(ss8,~in1,in2,~in3,~in4,~in5);

and I2(ss9,~in1,in2,~in3,~in4,in5);

and J2(ss10,~in1,in2,~in3,in4,~in5);

and K2(ss11,~in1,in2,~in3,in4,in5);

and L2(ss12,~in1,in2,in3,~in4,~in5);

and M2(ss13,~in1,in2,in3,~in4,in5);

and N2(ss14,~in1,in2,in3,in4,~in5);

and O2(ss15,~in1,in2,in3,in4,in5);

and P2(ss16,in1,~in2,~in3,~in4,~in5);

and R2(ss17,in1,~in2,~in3,~in4,in5);

and S2(ss18,in1,~in2,~in3,in4,~in5);

and T2(ss19,in1,~in2,~in3,in4,in5);

or Q2(ss20,ss1, ss2, ss3, ss4, ss5, ss6, ss7, ss8, ss9, ss10, ss11, ss12, ss13, ss14, ss15, ss16, ss17, ss18, ss19);

not ss181(ss, ss20);

and A(decA1,~in1,~in2,~in3,~in4,in5);

and B(decA2,~in1,~in2,~in3,in4,~in5);

and C(decA3,~in1,~in2,~in3,in4,in5);

and D(decA4,~in1,~in2,in3,~in4,~in5);

and E(decA5,~in1,~in2,in3,~in4,in5);

and F(decA6,~in1,~in2,in3,in4,~in5);

and G(decA7,~in1,~in2,in3,in4,in5);

and H(decA8,~in1,in2,~in3,~in4,~in5);

or one(decA,decA1,decA2,decA3,decA4,decA5,decA6,decA7,decA8, dA);//1

and two1(rstA1,~in1,in2,~in3,~in4,in5);//2

or two(rstA, rstA1, rst );//2

and A1(latchA1,~in1,~in2,~in3,~in4,in5);

and B1(latchA2,~in1,~in2,~in3,in4,~in5);

and C1(latchA3,~in1,~in2,~in3,in4,in5);

and D1(latchA4,~in1,~in2,in3,~in4,~in5);

and E1(latchA5,~in1,~in2,in3,~in4,in5);

and F1(latchA6,~in1,~in2,in3,in4,~in5);

and G1(latchA7,~in1,~in2,in3,in4,in5);

and H1(latchA8,~in1,in2,~in3,~in4,~in5);

or three(latchA,latchA1,latchA2,latchA3,latchA4,latchA5,latchA6,latchA7, latchA8, lA);//3

and four(validA,~in1,in2,~in3,~in4,in5);//4

and five(loadCbuf,~in1,in2,~in3,~in4,in5);//5

and six1(command1,~in1,in2,~in3,in4,~in5);

and six2(command2,~in1,in2,~in3,in4,in5);

and six3(command3,~in1,in2,in3,~in4,~in5);

and six4(command4,~in1,in2,in3,~in4,in5);

and six5(command5,~in1,in2,in3,in4,~in5);

and six6(command6,~in1,in2,in3,in4,in5);

and six7(command7,in1,~in2,~in3,~in4,~in5);

and six8(command8,in1,~in2,~in3,~in4,in5);

and six9(command9,in1,~in2,~in3,in4,~in5);

and six10(command10,in1,~in2,~in3,~in4,in5);

and six11(command11,in1,~in2,~in3,in4,~in5);

or six(command,command1,command2,command3,command4,command5,command6,command7, command8, command9, command10, command11);//6

and I(decRx1,~in1,in2,~in3,in4,~in5);

and J(decRx2,~in1,in2,~in3,in4,in5);

and K(decRx3,~in1,in2,in3,~in4,~in5);

and P(decRx4,~in1,in2,in3,~in4,in5);

and Q(decRx5,~in1,in2,in3,in4,~in5);

and R(decRx6,~in1,in2,in3,in4,in5);

and S(decRx7,in1,~in2,~in3,~in4,~in5);

and S4(decRx8,in1,~in2,~in3,~in4,in5);

or eight(decRx9, decRx1, decRx2, decRx3, decRx4, decRx5, decRx6, decRx7, decRx8);//8

and rxcheck(decRx, decRx9, cout);

and I5(latchRx1,~in1,in2,~in3,in4,~in5);

and J5(latchRx2,~in1,in2,~in3,in4,in5);

and K5(latchRx3,~in1,in2,in3,~in4,~in5);

and P5(latchRx4,~in1,in2,in3,~in4,in5);

and Q5(latchRx5,~in1,in2,in3,in4,~in5);

and R5(latchRx6,~in1,in2,in3,in4,in5);

and S5(latchRx7,in1,~in2,~in3,~in4,~in5);

and T5(latchRx8,in1,~in2,~in3,~in4,in5);

or nine(latchRx9, latchRx2, latchRx3, latchRx4, latchRx5, latchRx6, latchRx7, latchRx8,latchRx1 );//9

and latchRxCheck(latchRx,latchRx9, cout );

and validRx1(validRxTemp, in1,~in2,~in3,in4,~in5);

and validRx2(validRx, validRxTemp, cout );

and ten(rstRx1,in1,~in2,~in3,in4,~in5);

or ten1(rstRx, rstRx1, rst);//10

//TX

and loadTxcommand(loadTx,~in1,in2,~in3,in4,in5);

and II(decTx1,~in1,in2,in3,~in4,~in5);

and JJ(decTx2,~in1,in2,in3,~in4,in5);

and KK(decTx3,~in1,in2,in3,in4,~in5);

and PP(decTx4,~in1,in2,in3,in4,in5);

and QQ(decTx5,in1,~in2,~in3,~in4,~in5);

and RR(decTx6,in1,~in2,~in3,~in4,in5);

and SS(decTx7,in1,~in2,~in3,in4,~in5);

and TT(decTx8,in1,~in2,~in3,in4,in5);

or eight1(decTx9, decTx1, decTx2, decTx3, decTx4, decTx5, decTx6, decTx7, decTx8);//8

and DeCcheck1(decTx, decTx9, ~cout);

and I65(shiftTx1,~in1,in2,in3,~in4,~in5);

and J65(shiftTx2,~in1,in2,in3,~in4,in5);

and K65(shiftTx3,~in1,in2,in3,in4,~in5);

and P65(shiftTx4,~in1,in2,in3,in4,in5);

and Q65(shiftTx5,in1,~in2,~in3,~in4,~in5);

and R65(shiftTx6,in1,~in2,~in3,~in4,in5);

and S65(shiftTx7,in1,~in2,~in3,in4,~in5);

and T65(shiftTx8,in1,~in2,~in3,in4,in5);

or eight2(shiftTx9, shiftTx1, shiftTx2, shiftTx3, shiftTx4, shiftTx5, shiftTx6, shiftTx7, shiftTx8);//8

and shiftcheck1(shiftTx, shiftTx9, ~cout);

and KK3(rstTx1, in1,~in2,in3,~in4,~in5);

or ten12(rstTx, rstTx1, rst);

endmodule

module counterRX\_4bit(RXCount,sck,decRx,rstRX);

output reg [3:0] RXCount;

input sck,rstRX,decRx;

initial begin

RXCount <= 4'b1000;

end

always @(posedge sck)

begin

if(rstRX)

begin

RXCount <= 4'b1000;

end

else

begin

if(RXCount != 4'b0000 && decRx)

begin

if(!RXCount)

begin

RXCount <= 4'b1000;

end

RXCount <= RXCount - 1;

end

end

end

endmodule

module rxbuf(rxout, sdi, sck, latchrxdata, RxCount, validRx);

output[7:0] rxout;

input sdi, sck, latchrxdata, validRx;

input [3:0] RxCount;

reg [7:0] finalRotateRxData;

initial begin

finalRotateRxData <= 8'b00000000;

end

always @(posedge sck)

begin

if(latchrxdata)

begin

finalRotateRxData[RxCount-1] <= sdi;

end

end

assign rxout = (validRx) ? finalRotateRxData : 8'bz;

endmodule

module counterAddr\_4bit(ACount,sck,decA,rstA);

output reg [3:0] ACount;

input sck,rstA,decA;

initial begin

ACount <= 4'b1000;

end

always @(posedge sck)

begin

if(rstA)

begin

ACount <= 4'b1000;

end

else

begin

if(ACount != 4'b0000 && decA)

begin

if(!ACount)

begin

ACount <= 4'b1000;

end

ACount <= ACount - 1;

end

end

end

endmodule

module addreg(Aout, sdi, sck, latchA, validA, Acount);

output[7:0] Aout;

input sdi, sck, latchA, validA;

input [3:0] Acount;

reg [7:0] finalRotateAddr;

initial begin

finalRotateAddr = 8'b00000000;

end

always @(posedge sck)

begin

if(latchA)

begin

finalRotateAddr[Acount] <= sdi;

end

end

assign Aout = (validA) ? finalRotateAddr : Aout;

endmodule

module triStatebufLoadCommand (cin,sdi,LoadCBuf);

output cin;

input sdi;

input LoadCBuf;

wire sdi,LoadCBuf;

wire cin;

assign cin = (LoadCBuf) ? sdi : 1'bz;

endmodule

module commandBuf (cout, sck, cin, loadCBuf);

output cout;

input cin;

input sck, loadCBuf;

reg cout;

always @(posedge sck)

begin

if(loadCBuf)

cout = cin;

end

endmodule

module dflipflop(Q,D,sck);

input [7:0] D;

input sck;

output reg [7:0] Q;

always @(posedge sck)

Q = D;

endmodule

module readWriteSig(sigWE, sigRE, cout,command);

output sigRE, sigWE;

input cout;

input command;

wire cout,command;

wire sigRE, sigWE;

assign sigWE = (command) ? cout : 1'b0;

assign sigRE = (command) ? ~cout : 1'b0;

endmodule

module sram(dout, din, addr, sck, re, we, ss);

output [7:0] dout;

input [7:0] addr;

input [7:0] din;

input re, we, ss, sck;

reg[7:0] sram [0:255];

reg[7:0] dout;

wire[7:0] din;

reg[1:0] read\_write\_state;

wire en;

assign en = ~ss;

integer k;

initial

begin

for (k = 0; k < 256 - 1; k = k + 1)

begin

//sram[k] <= 8'b01110111;

sram[k] <= 8'b00000000;

end

end

always@(posedge sck)

begin

if(en)

begin

read\_write\_state = { we, re};

case(read\_write\_state)

2'b00 : dout = 8'bz;

2'b01 : dout = sram[addr];

2'b10 : sram[addr] = din;

2'b11 : $display("error");

//default : dout = sram[addr];

endcase

end

end

endmodule

module counterTX\_4bit(TXCount,sck,decTx,rstTX);

output reg [3:0] TXCount;

input sck,rstTX,decTx;

initial begin

TXCount <= 4'b1000;

end

always @(negedge sck)

begin

if(rstTX)

begin

TXCount <= 4'b1000;

end

else

begin

if (TXCount != 4'b0000 && decTx)

begin

TXCount <= TXCount - 1;

end

end

end

endmodule

module TxBuf(sdo, sck, ParallelIn, LoadTx, ShiftTx, TxCount);

output reg sdo;

input[7:0] ParallelIn;

input sck, LoadTx, ShiftTx;

input[3:0] TxCount;

integer i;

parameter s = 7;

reg [s:0]temp;

reg [s:0]tempR;

always@(negedge (sck))

begin

if (LoadTx)

begin

temp <= ParallelIn;

end

else

begin

for (i = s; i >= 0; i = i-1)

tempR[s-i] = temp[i];

end

assign sdo = (ShiftTx)? tempR[TxCount-1] : 1'bz;

end

endmodule

module testBench;

reg sck;

reg sdi;

reg clk;

reg rst;

reg latchA, dA;

datapath datapath\_Inst ( .sck(sck),

.sdi(sdi),

.rst(rst),

.lA(latchA),

.dA(dA));

initial begin

$dumpfile("TestFixture.vcd");

$dumpvars(9,testBench);

sdi = 1;

sck = 0;

clk = 0;

rst <= 1;

latchA = 1;

dA = 1;

end

always

#1 clk = ~clk;

always

#5 sck = ~sck;

initial

begin

rst <= 0;

latchA = 1;

dA = 1;

@ (negedge sck);

rst <= 0;

latchA = 0;

dA = 0;

//reading data

sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= 0; //command bit

#90 rst <= 1;

//writing data

#10 rst <= 0;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

#10 sdi <= $random;

**COMPLETE RESULT OF THE ARCHITECTURE**

(Write into SRAM and read from SRAM)

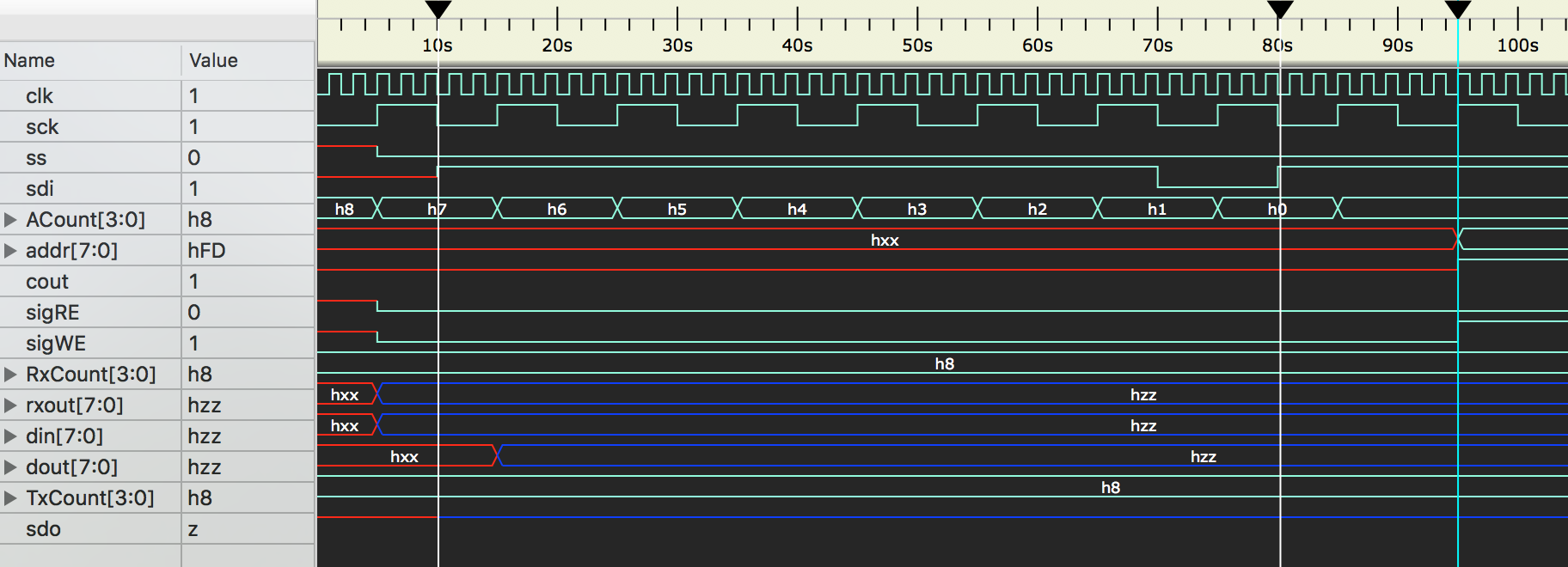


Figure 5: The address hFD and command WE = 1 is obtained from the SDI port for write operation

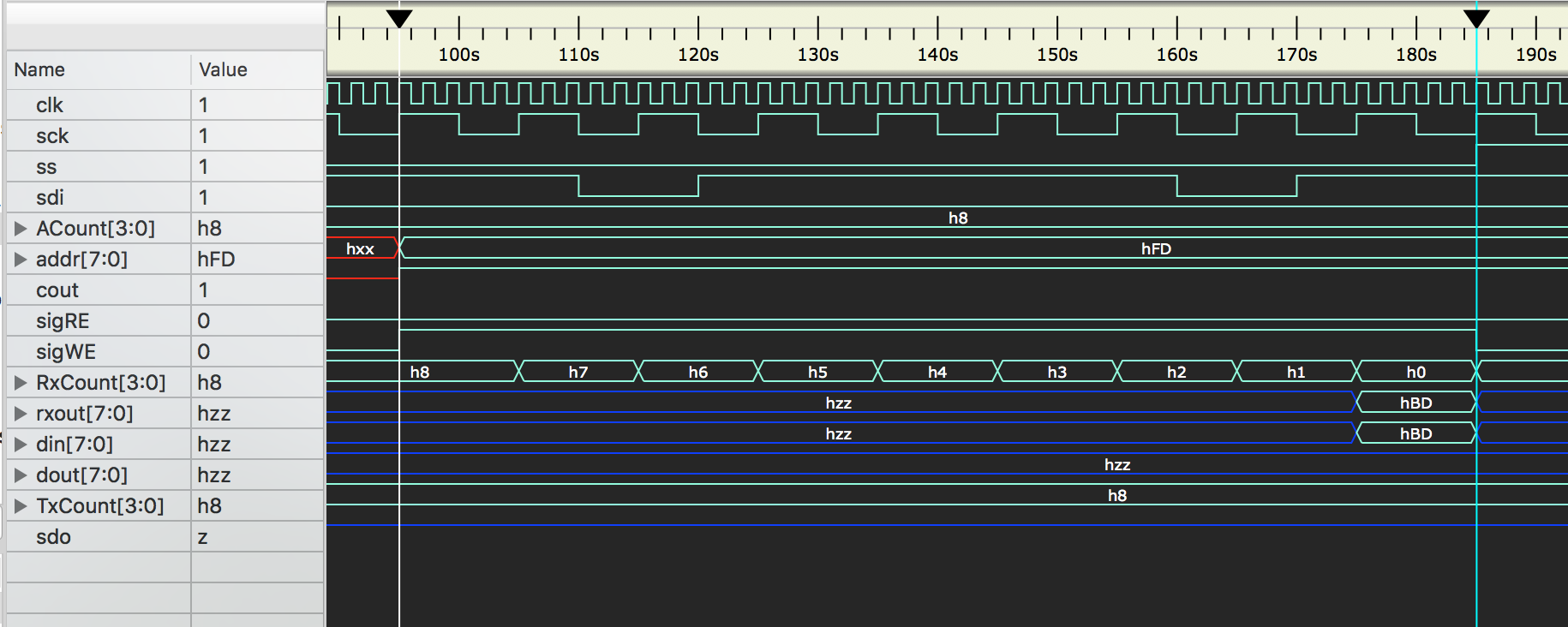


Figure 6: The data hBD is obtained from the SDI Port for write operation

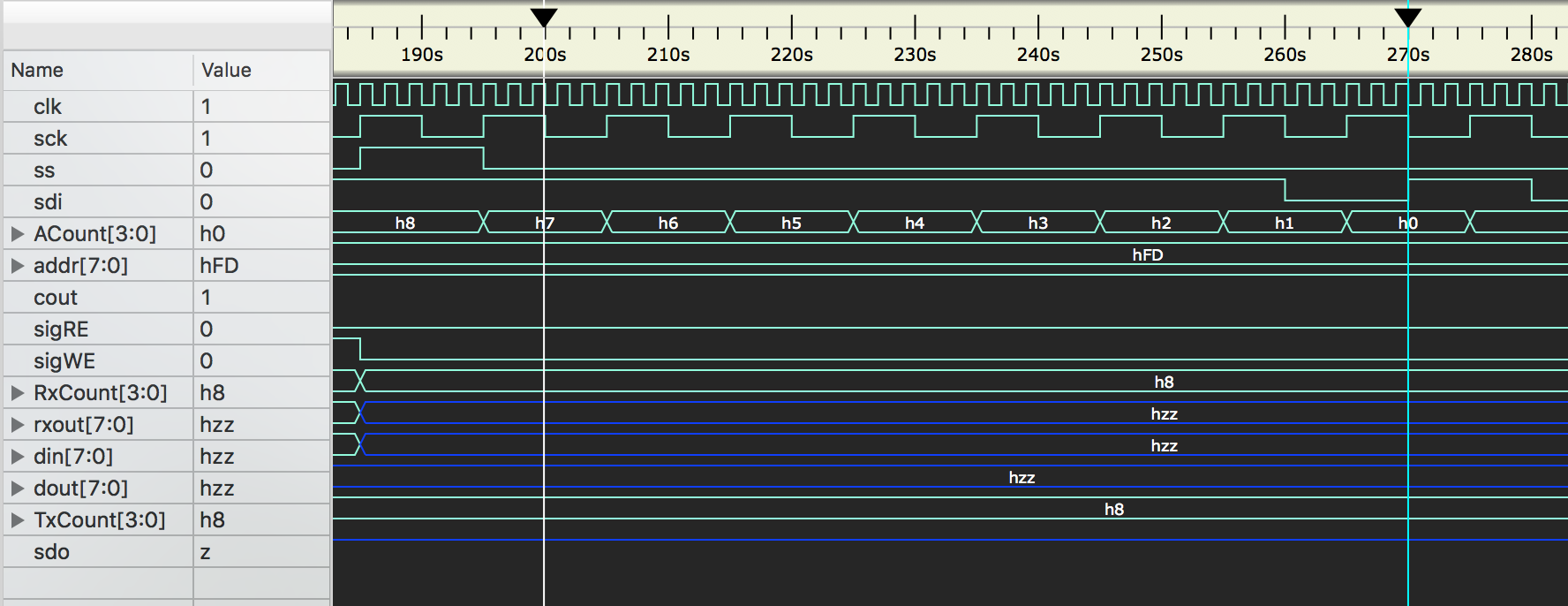


Figure 7: The Address hFD is obtained from the SDI Port for read operation

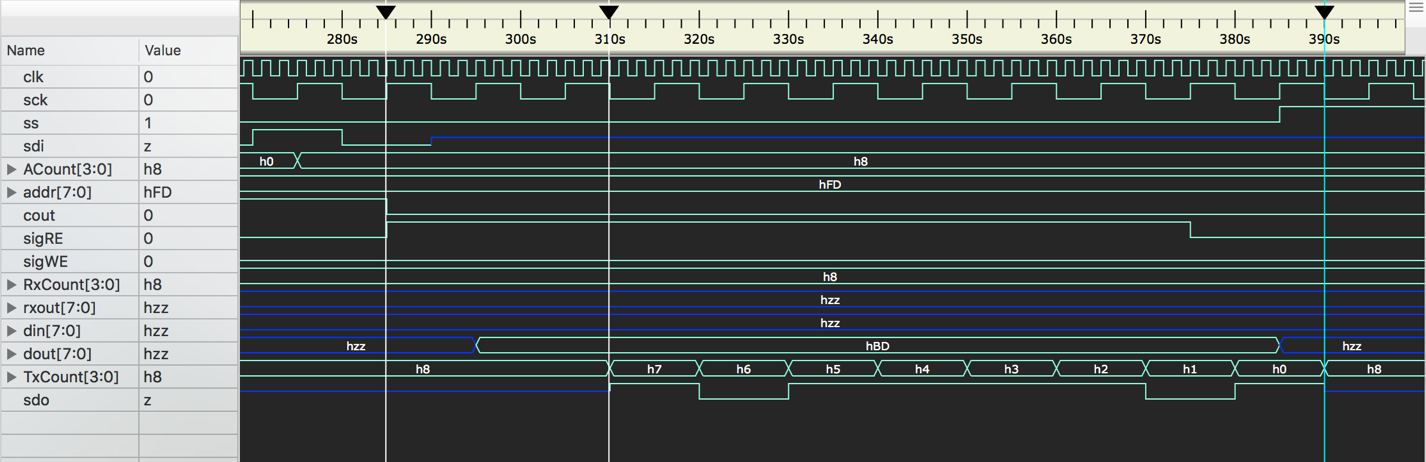


Figure 8: The data hBD is obtained from the SDO Port for read operation

**VERIFICATION RESULTS**

**TEST CASE 1:**

Read, write, Read

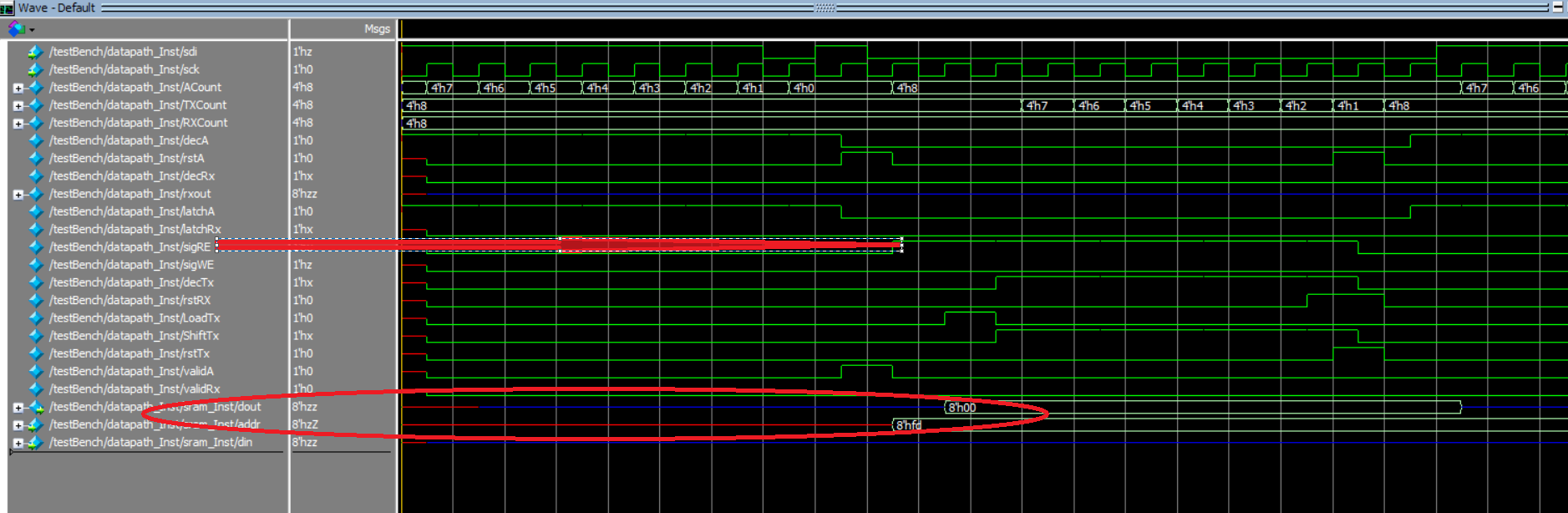


Figure 9: Address, command as expected are given to SRAM for READ

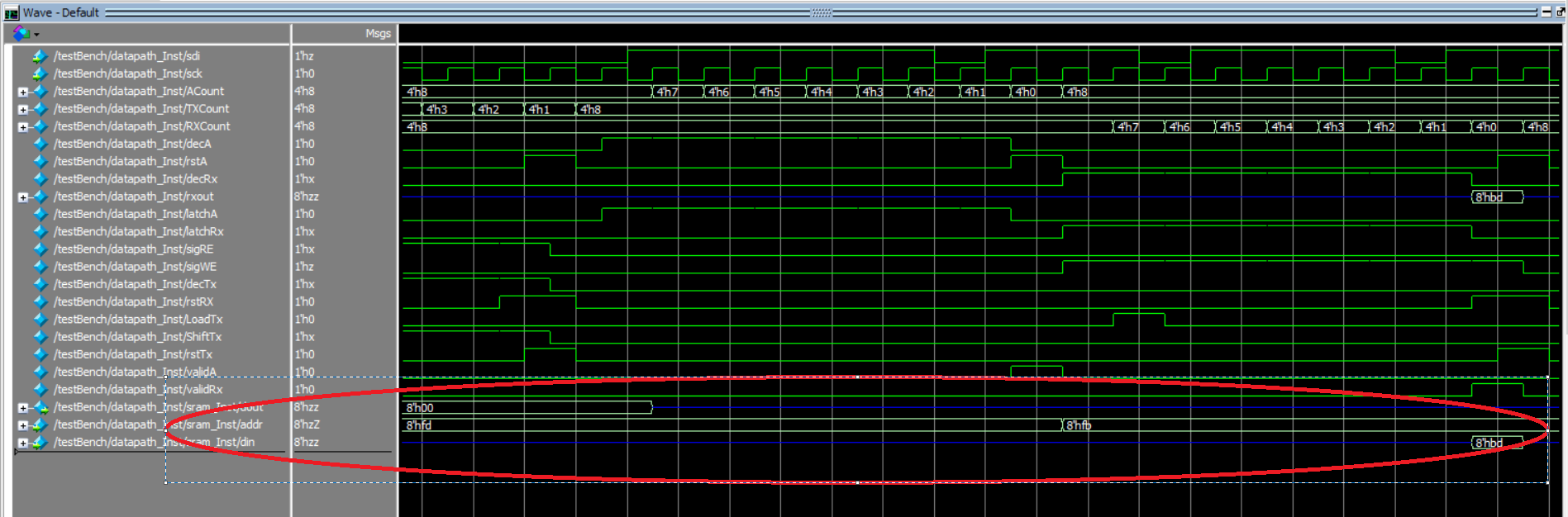


Figure 10: Address, command, data as expected are given to SRAM for Write

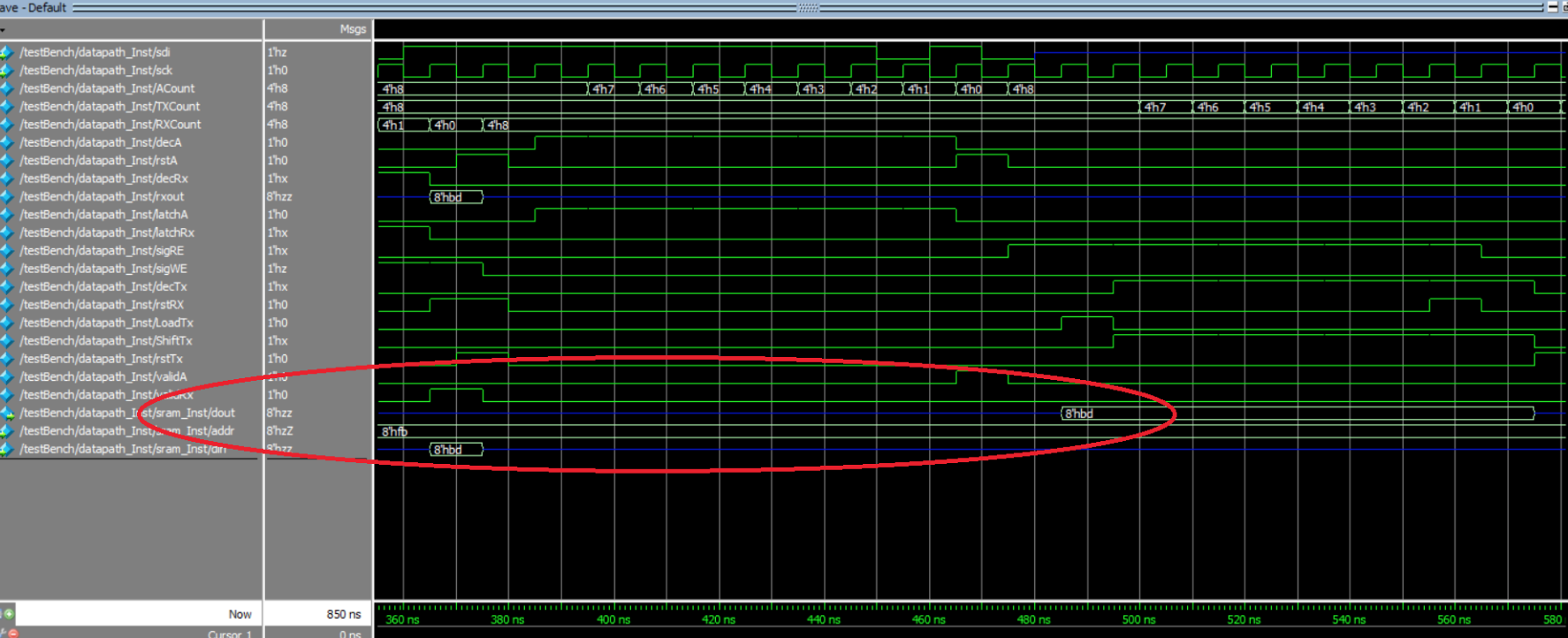


Figure 11: Address, command as expected are given to SRAM for READ

**TEST CASE 2:**

Read, write

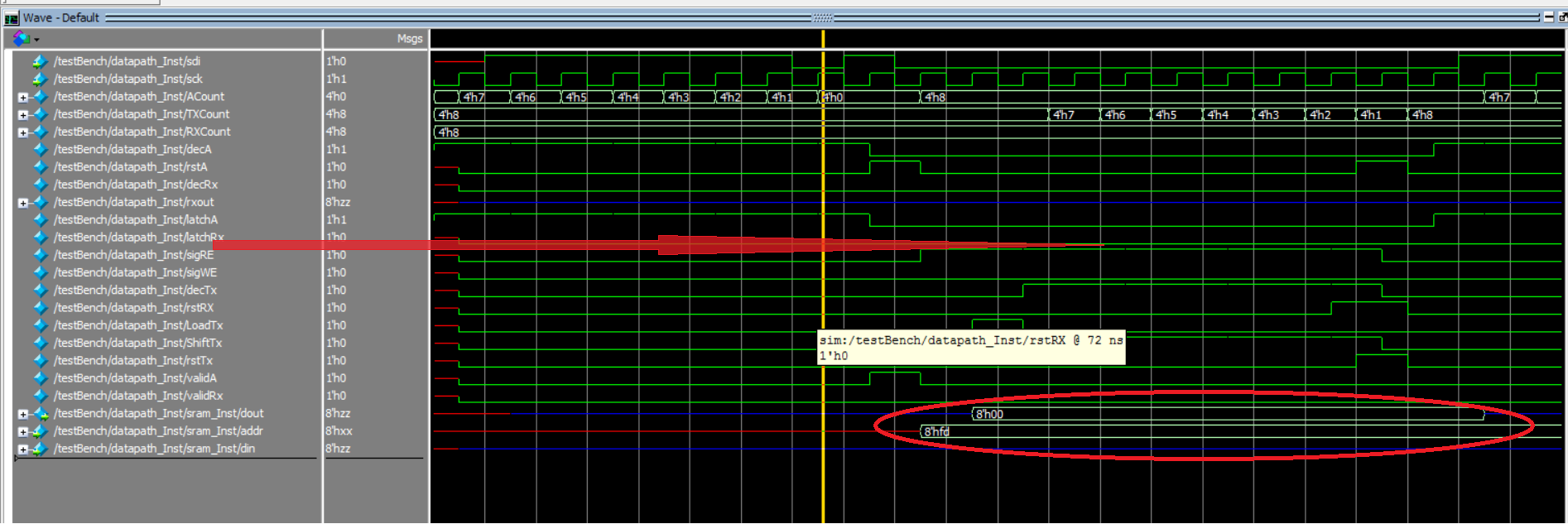


Figure 12: address, command given to SRAM for READ

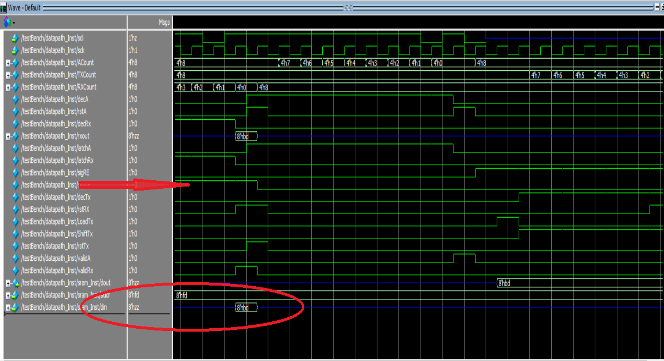


Figure 13: address, data, command given to SRAM for Write

TEST CASE 3:

Write, read

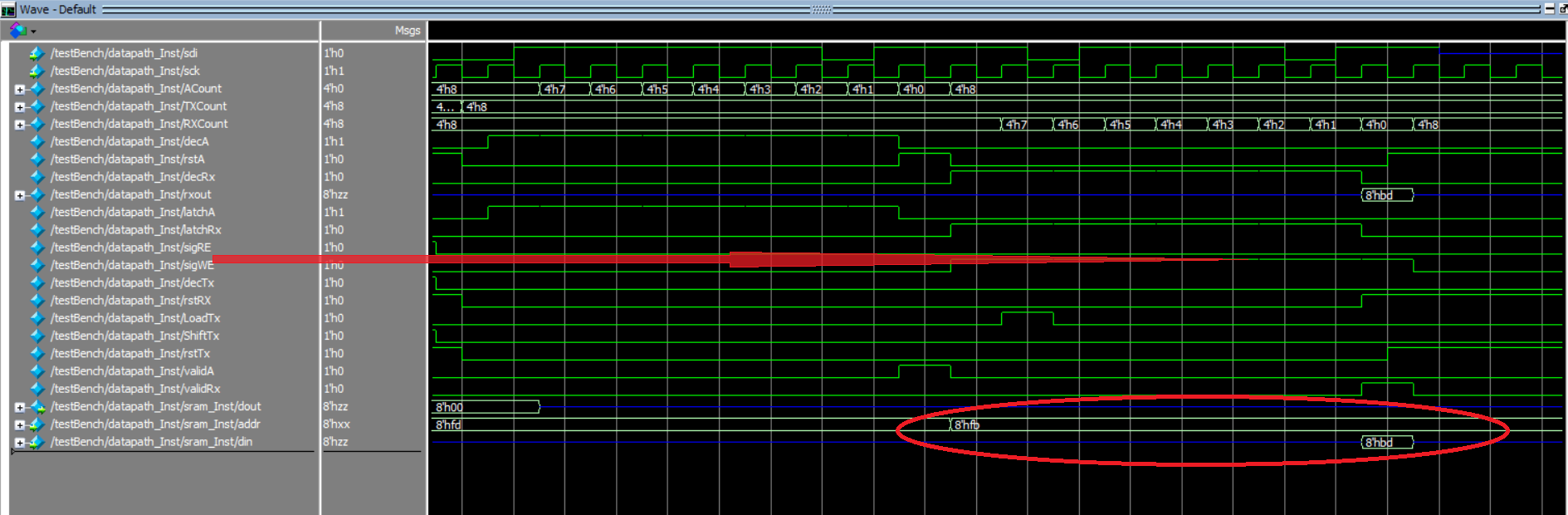


Figure 14: address, data, command given to SRAM for Write

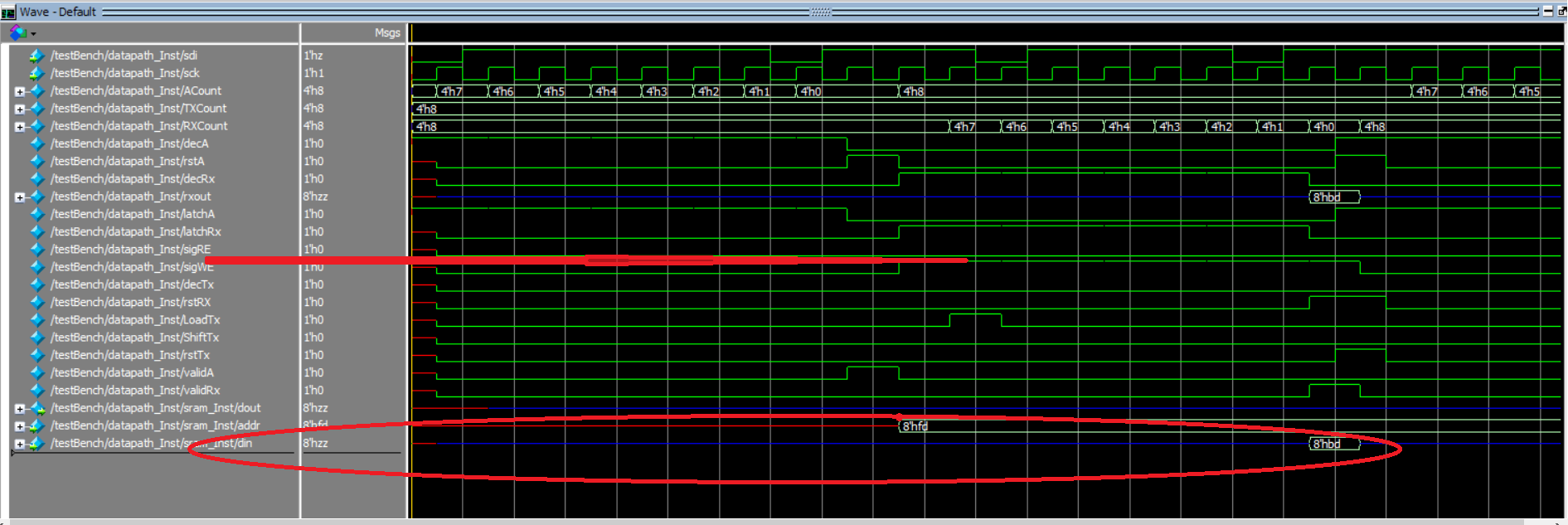


Figure 15: address, command given to SRAM for READ