Lab 6: SPI SRAM module

T 2pm-5pm

Zichen Huang, Jack Landers

Introduction

In this lab, we are going to build and test a bidirectional SPI SRAM chip model which includes a secondary SPI interface for connection to a microprocessor. SPI is a serial communications port that uses 4 wires for full communications. Our main focus will be on developing a version of the Microchip 23A640 64-Kbit SPI Bus Low-Power Serial SRAM.

To kick off, we need to wrap our heads around the 23A640 datasheet in detail. It's crucial because it'll guide us on what features are essential and what we can skip. The aim is to create a Verilog implementation of this chip that's precise and compatible with the provided testbench.

Our first challenge is to create an asynchronous SRAM module with 8192, 8-bit data words, using a single, bidirectional data bus. After that, we need to create an SPI interface module and our goal here is to ensure the module can handle both read and write operations smoothly. The final hurdle involves integrating the SPI interface with the SRAM module within the M23A640 module. It's crucial for the seamless functioning of the individual components and to meet the 23A640 datasheet specifications.

Procedure

At the beginning of the lab, we need to read both datasheets very carefully. From the datasheet, we know that although we have a 16-bit address, our SRAM remains 13 bits, which means some of the bits are “don’t care” bits. This is pretty necessary for us to understand before getting into the different parts of the lab.

Part 1

From the datasheet, we know that the SRAM module includes 8192, 8-bit data words. Also, this is an asynchronous SRAM, meaning that it has no clocks. Besides, it uses a single, bidirectional data bus. So the inputs of the SRAM would be address, write_enable, and write_enable. An additional 8-bit bidirectional data bus is used as well. For storage, we need to create 8192 slots of 8-bit memory for storing all the data.

Since we want to write the data each time we receive the write_enable signal, we did “memory[address] <= data” at the positive edge of the write_enable signal. We did almost the same thing for reading the data, however, since we want the data output to be high impedance when read_enable is disabled, so else-if condition is used here.

For the testbench, we simply tested each condition that we could have when actually implementing the SRAM, and the module managed to pass all the tests.

Part 2

Here from the datasheet, we know that the SPI interface needs to connect to the SRAM in the future and it needs to perform the write and read operations. However, those operations are

separate and won't work at the same time, so I decided to use a state machine here. The basic idea of the state machine is that it's set to IDLE initially. And the other states are READ and WRITE. In the IDLE state, the module will try to read the received operation code and determine which is the next state. After getting to the next state, either READ or WRITE, it will receive the data and perform the corresponding operations. Here we only have reading and writing. After finishing the operations, it will go back to the IDLE state and wait for the next operation code.

Also, two internal wires, data_received and data_to_send are created for storing all the data before sending to or receiving them from the SRAM. Also, according to the datasheet, the MISO output should always remain high impedance unless we are reading the data from the datasheet. So in the IDLE state, we only have these codes:

```
state <= IDLE;
bit_count <= 0;
miso <= 1'bZ;
```

bit_count is a counter used for counting how many bits of data we have received or sent in order to determine when to transmit from one state to another.

In the WRITE state, the code is:

```
state <= WRITE;
data_received <= (data_received << 1) | mosi;
if (bit_count == 7) begin
    state <= IDLE; // Move to IDLE state after WRITE is complete
    bit_count <= 0; // Reset bit counter for READ operation
End
bit_count <= bit_count + 1;
```

So here we are collecting the data one bit at a time from the MOSI input. And it will return to the IDLE state by default after receiving all the data.

In the READ state, the code is:

```
state <= READ;
miso <= data_to_send[7 - bit_count]; // Send MSB first
if (bit_count == 7) begin
        state <= IDLE; // Return to IDLE after completing READ
        bit_count <= 0;
end
bit_count <= bit_count + 1;
```

The logic is pretty code to the READ state, however, here we need to get the data first and send them one by one to the master. It will return to the IDLE state by default as well.

In the testbench, I tested both the write and read functions as well as some corner cases that might not work well. After some debugging, we managed to get it running.

Part 3

Part 3 of this lab mainly focuses on combining the modules of Part 1 and Part 2 to make a functional SPI module. However, we need to some modifications to our codes from Part 1 and Part 2 here.

The logic of the SPI module is inherited from the Part 2 module, which uses a state machine. The biggest changes are made for reading the operational code and the address for reading and writing. In the IDLE state, we added sections for reading the operational code from the MOSI input. After determining the operation needed, it will move to the corresponding state.

In the WRITE state, for example, codes are modified to read the address and send the data to the corresponding SRAM slot after receiving all the data needed:

```
bit_count = bit_count + 1;
miso = 1'bZ;
if(bit_count < 6'd16)
        WriteAddress <= (WriteAddress << 1) | mosi;          //Read the address
if(bit_count == 6'd16)
        address <= WriteAddress[12:0];
if(bit_count >= 6'd16 && bit_count < 6'd24)
        data_received <= (data_received << 1) | mosi;        //Read the data
if(bit_count == 6'd23)
        prewe = 1;
if(bit_count == 6'd24)begin
        opcode <= 7'b0;
        we = 1;
        prewe = 0;
        state = IDLE;
        bit_count <= 6'b0;
End
miso = 1'bZ;
```

The counter is also modified to load all the data. A pre-write-enable indictor is added for enabling the data bus. Similar modifications are also done for the READ state.

We didn't modify the SRAM and the M23A640 module is added for using the SPI interface.

Results

• What problems did you encounter while testing your steps yourself?
When I tested the steps by myself, I found it couldn't write and read the data from the SRAM in the cycle when write_enable and read_enable are triggered. So I added codes for triggering the inout wire and prepared everything ready for reading and writing one clock cycle ahead of time.
• Did any problems arise when demonstrating for the TA? What were they? Explain your thoughts on how/why these test cases escaped your own testing.

When I demoed my part 2 to the TA, we found that the MISO was not working properly. Then I went back and checked my codes and found that I didn't count the delay of the state machine which affects the last output of the MISO.
I guess the reason why this escaped my own testing was because I didn't perform a full check on my MISO and I forgot some corner cases.

Final results from Wolfe's testbench:

```
Chronologic VCS simulator copyright 1991-2019
Contains Synopsys proprietary information.
Compiler version P-2019.06-SP2-1_Full64; Runtime version P-2019.06-SP2-1_Full64;
  Dec  4 00:14 2023
Beginning write-read test
Passed write-read test
Beginning write-read test 2
Passed write-read test 2
Beginning broken write test
Passed broken write test
Beginning alias test
Passed alias test
***** - ALL TESTS PASSED - *****
$finish called from file "tbserial.v", line 27.
$finish at simulation time                20825
          V C S   S i m u l a t i o n   R e p o r t
Time: 20825
CPU Time:      0.180 seconds;       Data structure size:   0.0Mb
```

Part 1

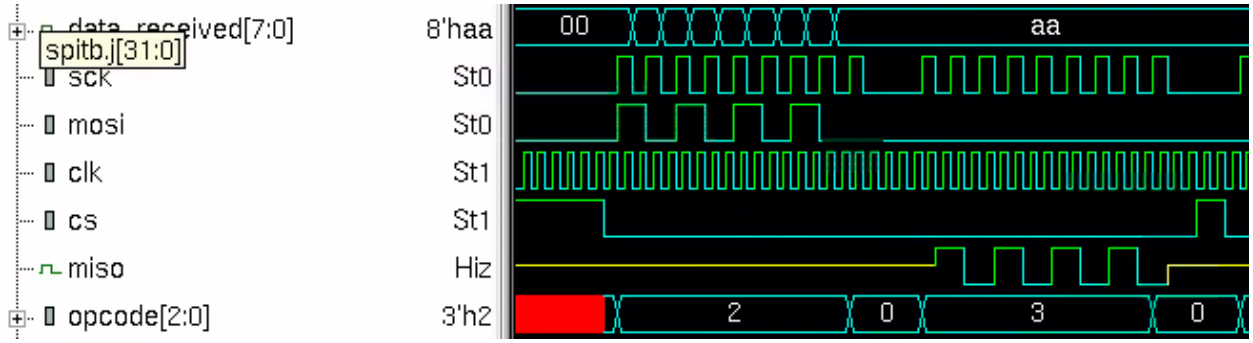
```
Read data at address 0000000000001: 10101010
Read data at address 0000000000010: 11001100
Read data at address 0000000000100: zzzzzzzz
Read data at address 1111111111111: 11111100
```

According to my testbench enclosed below in the appendix, address 000000000000001 is supposed to have data 10101010 and address 000000000000010 is also supposed to have data 11001100. For address 000000000000100, the data was written when write_enable is low, so it should output high impedance. For address 1111111111111111, it's corner situation test and it returns the correct data.

Part 2

```
Test passed for vector aa
Test passed for vector 55
Test passed for vector ff
Test passed for vector 00
```

According to my testbench, I connect to the internal wire of the SPI interface which stores the received data and the data to send. They also return the correct answers. The MISO also works according to the waveform.



It’s the first case in which write aa to the SPI interface first and then read it.

Conclusion

This is a very time-consuming lab, I spend a lot of time debugging. What I felt is that when it gets to the logical level of the chips, you always have to consider delays and potential conflicts first. It’s not like writing Python codes which just run line by line and do a lot of things automatically. In Verilog, if you want to do something, usually you have to deal with the clock and other logic. For example, in the beginning, I thought everything was supposed to be run at the rising edge of the SPI clock, however, operations are actually finished at the negative edge of the clock.

Appendix

SRAM:

```
module sram (address, re, we, data);

input [12:0]address;
input re, we;
inout [7:0]data;

reg [7:0]memory[8191:0];

reg [7:0]data_out;

assign data = (re) ? data_out : 8'bz; // Bidirectional data bus

always @(posedge we) begin
        memory[address] <= data;
end

always @(*) begin
        if(re)
                data_out = memory[address];
        else
                data_out = 8'bz;
end

endmodule
```

SRAM TESTBENCH:

```
module sramtb();

reg [12:0]address;
reg re, we;
wire [7:0]data;

reg [7:0]data_in;

sram dut(address, re, we, data);

assign data = (we) ? data_in : 8'bz; // Connect bidirectional data bus
```

```
initial begin

    // Reset signals
    address = 0;
    we = 0;
    re = 0;
    data_in = 0;

    // Allow some time for initial conditions to settle
    #10;

    // Write some data to the SRAM
    address = 13'b0000000000001;
    data_in = 8'b10101010;
    we = 1;
    #10;
    we = 0;
    #10;

    address = 13'b0000000000010;
    data_in = 8'b11001100;
    we = 1;
    #10;
            data_in = 8'b11111111; //Test if the data only changes when we is asserted
            #10;
    we = 0;
    #10;
            data_in = 8'b01010101;
            #10;
            address = 13'b0000000000100; //Test if the data writes when we is low
            #10;

    // Read back the data and check
    address = 13'b0000000000001;
    re = 1;
    #10;
    $display("Read data at address 0000000000001: %b", data);
    re = 0;
    #10;
```

```
        address = 13'b0000000000010;
        re = 1;
        #10;
        $display("Read data at address 0000000000010: %b", data);
        re = 0;
        #10;

                //Test the output of the data when we and re are both low
                address = 13'b0000000000100;
                #10;
        $display("Read data at address 0000000000100: %b", data);
                #10;

                //Test the last spot of the memory
                address = 13'b1111111111111;
                #10
                data_in = 8'b11111100;
                we = 1;
                #10
                re = 0;
                we = 1;
                #10
                $display("Read data at address 1111111111111: %b", data);
                #10;

        // End of test
        $finish;
end

endmodule
```

SPI (Part 2):

```
module spi(rst, cs, sck, mosi, miso, data_received, data_to_send, opcode);
   input rst;         // Reset signal
   input cs;          // Chip Select (active low)
   input sck;         // SPI Clock
   input mosi;        // Master Out Slave In
   output reg miso;        // Master In Slave Out
   output reg [7:0] data_received; // Data received from master
   input [7:0] data_to_send;  // Data to send to master
```

```
        input [2:0] opcode;

// State definitions
parameter IDLE  = 2'b00;
parameter WRITE = 2'b01;
parameter READ  = 2'b10;

// State and bit count variables
reg [1:0] state = IDLE;
reg [2:0] bit_count = 0;

always @(posedge sck or posedge rst) begin
   if (rst) begin
                        state <= IDLE;
      bit_count <= 0;
      data_received <= 0;
      miso <= 1'bZ;
   end
      end

always @(negedge sck) begin
               if (!cs) begin
                        if (opcode == 2'b00) begin
                                 state <= IDLE;
                                 bit_count <= 0;
                                 miso <= 1'bZ;
                        end else if (opcode == 2'b10) begin
                                 state <= WRITE;
          data_received <= (data_received << 1) | mosi;
          if (bit_count == 7) begin
             state <= IDLE; // Move to IDLE state after WRITE is complete
             bit_count <= 0; // Reset bit counter for READ operation
          end
          bit_count <= bit_count + 1;
                        end else if (opcode == 2'b11) begin
                                 state <= READ;
          miso <= data_to_send[7 - bit_count]; // Send MSB first
          if (bit_count == 7) begin
             state <= IDLE; // Return to IDLE after completing READ
                                          bit_count <= 0;
```

```
            end
                                    bit_count <= bit_count + 1;
                            end
                    end else begin
            state <= IDLE;
            miso <= 1'bZ; // Ensure MISO is high impedance when not in use
            bit_count <= 0;
                    end
            end
endmodule
```

SPI TESTBENCH:

```
module spitb();
   reg clk;
   reg rst;
   reg cs;
   reg sck;
   reg mosi;
   reg [7:0] data_to_send;
         reg [2:0] opcode;

   wire miso;
   wire [7:0] data_received;

   spi dut(rst, cs, sck, mosi, miso, data_received, data_to_send, opcode);

   always #5 clk = ~clk; // 100MHz clock

   reg [7:0] test_vector[0:3];
   reg [7:0] transfer_data;
   integer i, j;

   initial begin
      // Initialize Inputs
      clk = 0;
      rst = 1; // Apply reset initially
      cs = 1; // Chip select is inactive initially
      sck = 0;
      mosi = 0;
      data_to_send = 8'h00; // Initialize data to send as 0
```

```
#20; rst = 0; // Release reset

// Initialize test vectors
test_vector[0] = 8'haa; // Test pattern 10101010
test_vector[1] = 8'h55; // Test pattern 01010101
test_vector[2] = 8'hff; // Test pattern 11111111
test_vector[3] = 8'h00; // Test pattern 00000000

// Wait for reset to complete
#40;

// Begin testing
for (i=0; i<4; i=i+1) begin
   cs = 0; // Activate chip select
                        opcode = 2'b00; // idle operation
                        #10
                        opcode = 2'b10; // write operation
   for (j=7; j>=0; j=j-1) begin
      mosi = test_vector[i][j];
      sck = 1;
      #10; // Full period for SCK high
      sck = 0;
                                #10;
   end
                        opcode = 2'b00;
   // Check data_received with test_vector
   if (data_received !== test_vector[i]) begin
      $display("Test failed for vector %h, received %h", test_vector[i], data_received);
   end else begin
      $display("Test passed for vector %h", test_vector[i]);
   end

                        sck = 1;
                        #10;
                        sck = 0;

                        #20
                        cs = 1;
```

```
    // Prepare for read operation
    data_to_send = test_vector[i];

    // Start read operation
    cs = 0; // Activate chip select

                        #20
                        opcode = 2'b11;
    for (j=7; j>=0; j=j-1) begin
                            sck = 1;
       #10; // Full period for SCK high
       sck = 0;
                                #10;
    end
                        opcode = 2'b00; // idle operation
                        sck = 1;
                        #10;
                        sck = 0;
                        #20
    cs = 1; // Deactivate chip select
    #20;

              end

              cs = 0;
              opcode = 2'b10;
              for (j = 7; j>=0; j=j-1) begin
                        mosi = test_vector[0][j];
                        sck = 1;
                        #10;
                        sck = 0;
                        #10;
              end
              opcode = 2'b11;

              data_to_send = test_vector[0];
for (j = 0; j<=7; j=j+1) begin
                 sck = 1;
    #10;
                        if (j > 2)
```

```
                                cs = 1;
        sck = 0;
                        #10;
    end
            cs = 1;
            opcode = 2'b10;
            for (j = 7; j>=0; j=j-1) begin
                    sck = 1;
                    #10;
                    sck = 0;
                    #10;
            end

    $finish; // End simulation
  end

endmodule
```

SPI (Part 3):

```
module spi(csb, sck, mosi, miso);
  input csb;         // Chip Select (active low)
  input sck;        // SPI Clock
  input mosi;       // Master Out Slave In
  output reg miso;       // Master In Slave Out

        reg [7:0]opcode = 7'b0;                 // Instruction received from master
        reg [7:0]data_received = 7'b0;          // Data received from master
        wire [7:0]data_to_send;         // Data to send to master
        reg [15:0]ReadAddress;
        reg [15:0]WriteAddress;
        reg [12:0]address;              // Address for reading and writing data
        reg re, we;
        reg prewe;
        reg over;                               // Prepare for reading and writing
        wire [7:0]data;

        sram dut2(address, re, we, data);

  // State definitions
  parameter IDLE  = 2'b00;
```

```
    parameter WRITE = 2'b01;
    parameter READ  = 2'b10;

    // State and bit count variables
    reg [1:0] state = IDLE;
    reg [6:0] bit_count = 0;

    always @(negedge sck) begin
                if (!csb) begin
                        case (state)
                                IDLE: begin
                                        opcode <= (opcode << 1) | mosi;       //Read the instruction
                                        data_received <= 0;                             //Reset
everything
                                        ReadAddress <= 0;
                                        WriteAddress <= 0;
                                        bit_count <= 0;
                                        miso = 1'bZ;                                    //High
impedence for so
                                        re = 1'b0;
                                        we = 1'b0;
                                        over = 0;
                                        if(opcode == 8'b00000010)
                                                state = WRITE;
                                        else if(opcode == 8'b00000011)
                                                state = READ;
                                end
                                WRITE: begin
                                        bit_count = bit_count + 1;
                                        miso = 1'bZ;
                                        if(bit_count < 6'd16)
                                                WriteAddress <= (WriteAddress << 1) | mosi;
//Read the address
                                        if(bit_count == 6'd16)
                                                address <= WriteAddress[12:0];
                                        if(bit_count >= 6'd16 && bit_count < 6'd24)
                                                data_received <= (data_received << 1) | mosi;
//Read the data
                                        if(bit_count == 6'd23)
                                                prewe = 1;
```

```
                                        if(bit_count == 6'd24)begin
                                                opcode <= 7'b0;
                                                we = 1;
                                                prewe = 0;
                                                state = IDLE;
                                                bit_count <= 6'b0;
                                        end
                                        miso = 1'bZ;
                                end
                                READ: begin
                                        if      (bit_count < 6'd15)
                                                ReadAddress <= (ReadAddress << 1) | mosi;
//Read the address
                                        if      (bit_count == 6'd14)
                                                re = 1;
                                        if (bit_count > 6'd14 && bit_count < 6'd23 && !over)
begin
                                                miso <= data_to_send[22 - bit_count];
                                        end
                                        if (bit_count == 6'd23) begin
                                                opcode <= 7'b0;
                                                state = IDLE;
                                                bit_count <= 6'd0;
                                        end
                                        bit_count = bit_count + 1;
                                end
                        endcase
                end else begin
                        state <= IDLE;
                end
        end

        always @(posedge sck) begin
                if (state == READ && !csb) begin
                        if(ReadAddress > 16'h1fff)
                                over = 1;
                        else
                                address = ReadAddress[12:0];
                end
        end
```

```
        assign data = (we||prewe) ? data_received : 8'bZ;
        assign data_to_send = re ? data : 8'bZ;

endmodule
```

M23A640:

```
module M23A640(csb, so, holdb, sck, si);

input csb;
input holdb;
input sck;
input si;
output so;

spi dut1(csb, sck, si, so);

endmodule
```