College of Engineering and Applied Sciences

ECE 3570 Introduction to Computer Architecture

Project Assignment #5

Team Members

Raj Basnet

15 April 2020

Introduction

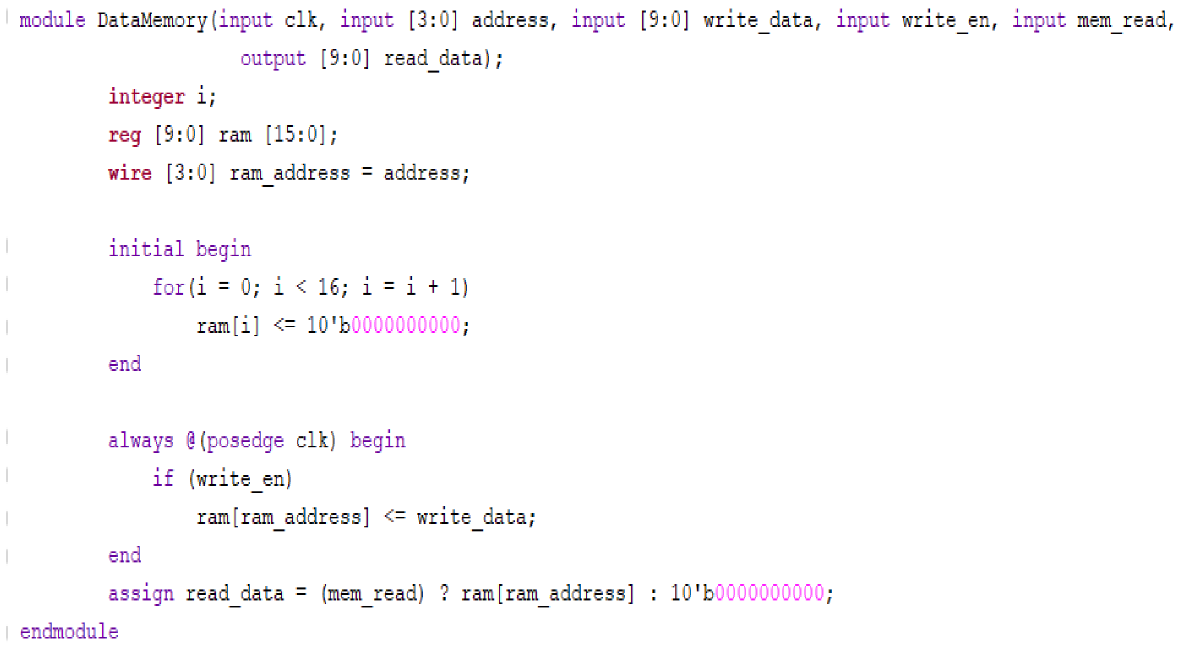
The main objective of the lab is to design a cache memory for a deeper understanding of the data cache implementation and design tradeoffs. The project will provide an insight about implementing the own data cache and memory using certain cache and block size. The project also helps to better understand the cache design and integration challenges. Furthermore, it will also enhance the learning of Verilog hardware description language.

Data Cache Design, Data Memory and Cache Controller components

1. Data Memory

Data Memory consists of the read and write data to be used by the 10-bit CPU. It also contains the RAM memory locations to store the values to be saved in the memory address and 16 such memory address or RAM locations are created. It has proper control signals for load and store data from or to memory locations.

Data Memory Verilog Code



```verilog
module DataMemory(input clk, input [3:0] address, input [9:0] write_data, input write_en, input mem_read,
                  output [9:0] read_data);
        integer i;
        reg [9:0] ram [15:0];
        wire [3:0] ram_address = address;

        initial begin
            for(i = 0; i < 16; i = i + 1)
                ram[i] <= 10'b0000000000;
        end

        always @(posedge clk) begin
            if (write_en)
                ram[ram_address] <= write_data;
        end
        assign read_data = (mem_read) ? ram[ram_address] : 10'b0000000000;
endmodule
```

Figure 1: Verilog Code for Data Memory

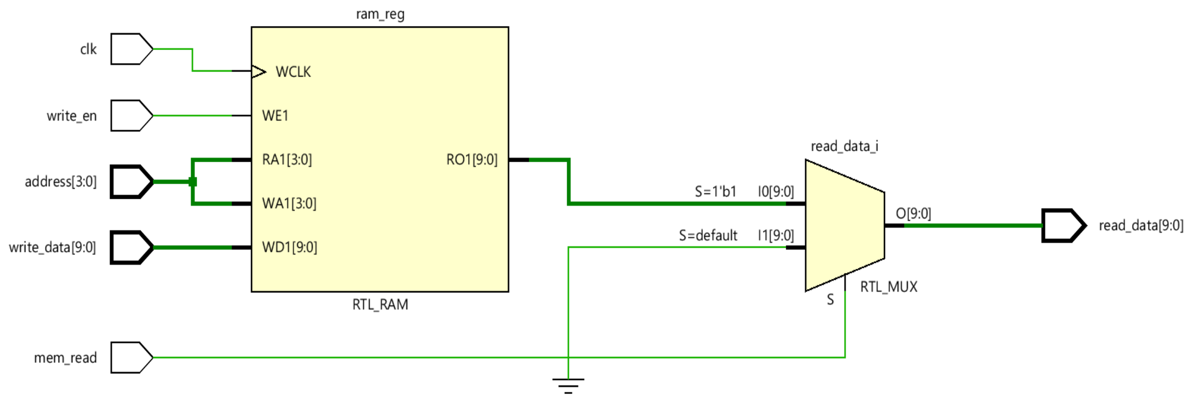
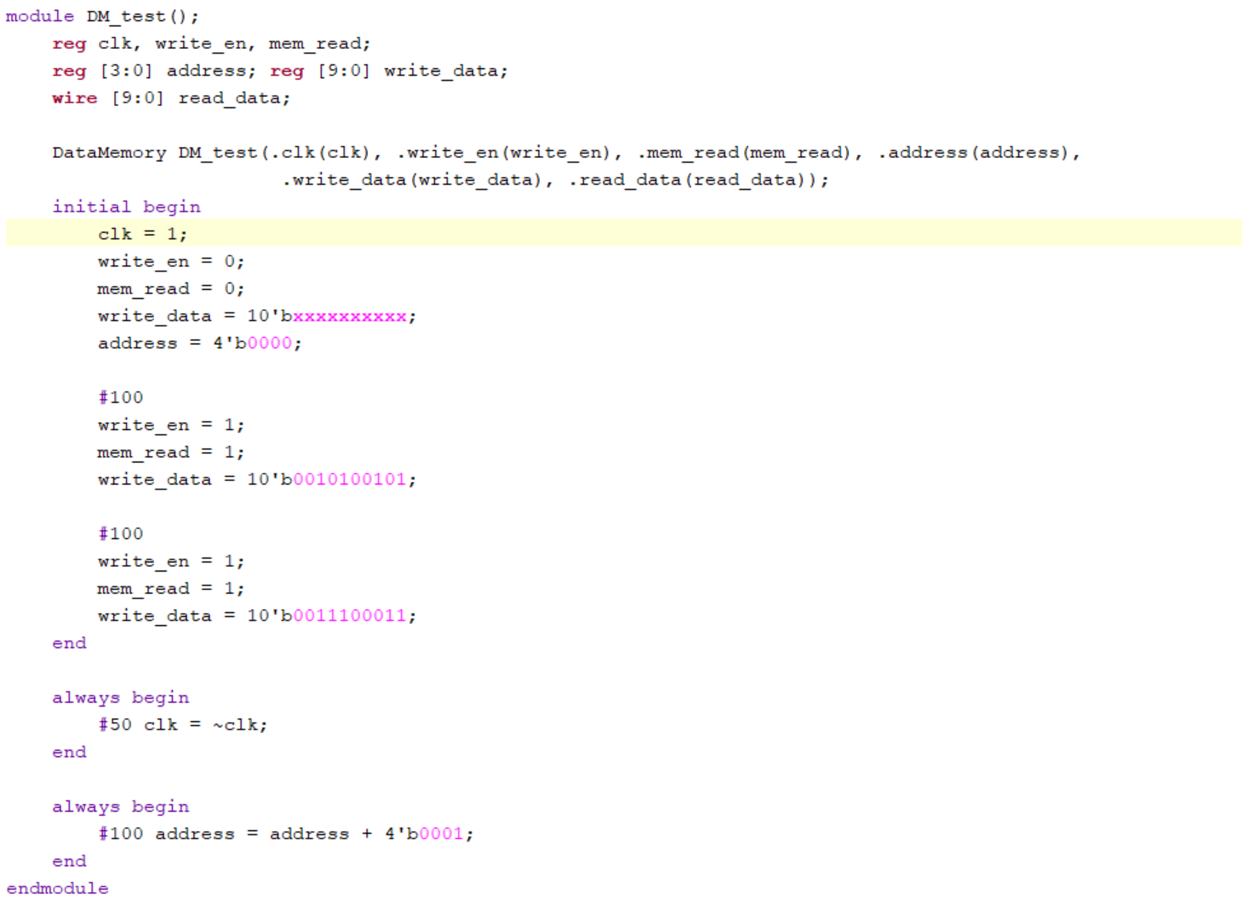


Figure 2: Schematic Diagram for Data Memory



```
module DM_test();
    reg clk, write_en, mem_read;
    reg [3:0] address; reg [9:0] write_data;
    wire [9:0] read_data;

    DataMemory DM_test(.clk(clk), .write_en(write_en), .mem_read(mem_read), .address(address),
                    .write_data(write_data), .read_data(read_data));
    initial begin
        clk = 1;
        write_en = 0;
        mem_read = 0;
        write_data = 10'bxxxxxxxxxx;
        address = 4'b0000;

        #100
        write_en = 1;
        mem_read = 1;
        write_data = 10'b0010100101;

        #100
        write_en = 1;
        mem_read = 1;
        write_data = 10'b0011100011;
    end

    always begin
        #50 clk = ~clk;
    end

    always begin
        #100 address = address + 4'b0001;
    end
endmodule
```

Figure 3: Verilog testbench for data memory

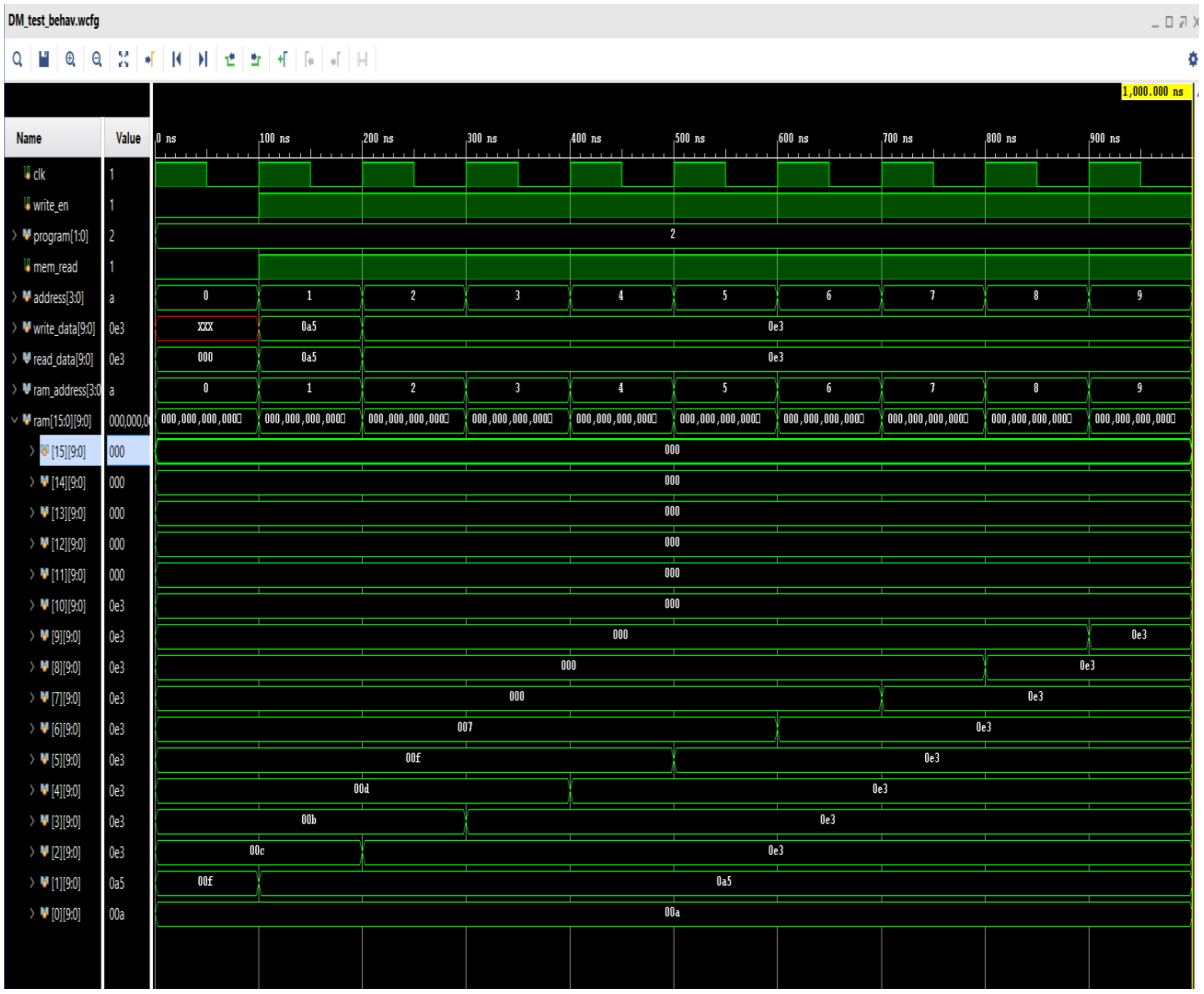
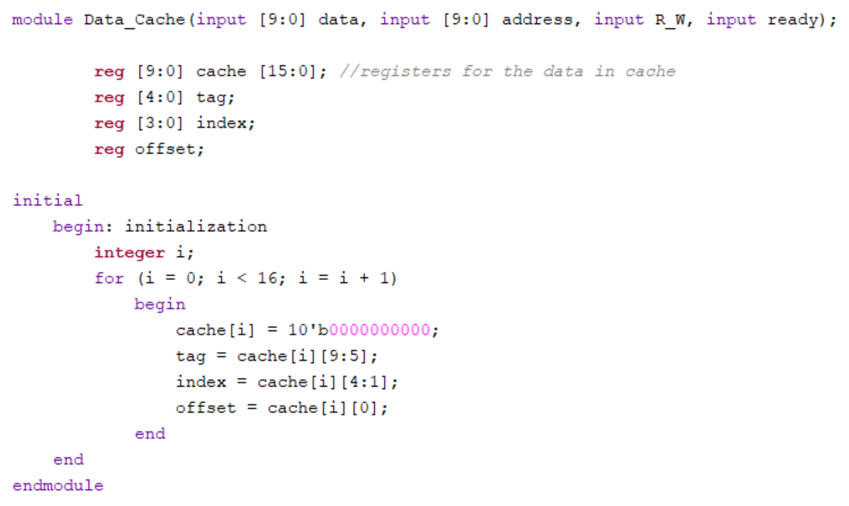


Figure 4: Timing Diagram for Data Memory

2. Data Cache

The data cache is composed of 16 blocks using direct-mapped cache. The block size for the data cache is 20 bits. The cache can process only one memory at a time. The cache consists of one data port, one address port, one read/write signal and one ready signal. The total cache size is 2^{10} bits because it consists of 10 bits of the address. Since each block consists of maximum 2 words, there is only 1 offset bit. For the 16 blocks of block size, it will have 4 index bits and then 5 tag bits (Total address size = 10 bits).

Verilog code for data cache



```
module Data_Cache(input [9:0] data, input [9:0] address, input R_W, input ready);

        reg [9:0] cache [15:0]; //registers for the data in cache
        reg [4:0] tag;
        reg [3:0] index;
        reg offset;

initial
    begin: initialization
        integer i;
        for (i = 0; i < 16; i = i + 1)
            begin
                cache[i] = 10'b0000000000;
                tag = cache[i][9:5];
                index = cache[i][4:1];
                offset = cache[i][0];
            end
    end
endmodule
```

Figure 5: Verilog code for data cache memory

3. **Cache Controller**

The cache controller consists of a finite state machine with one clock signal, one ready signal, a 10-bit data port, a 10-bit address port and a 10-bit output port. The cache controller acts to control and manage the co-ordination between data memory and cache memory. For a cache hit, the cache request, access and data output should be done within a same cycle. On the cache miss, the block will be fetched from the main memory using 2 extra cycles. The cache controller connects the data cache with data memory to implement the data cache for 10-bit CPU.

Verilog Code for Cache Controller

```
module Cache_Controller(input clk, input [9:0] data, input [9:0] address, input R_W,
output [9:0] out, input ready);

   reg [9:0] temp_out;
   reg [3:0] index; // for keeping index of current address
   reg [4:0] tag;  // for keeping tag of current address
   reg dirty [15:0];
reg [4:0] prev_tag [15:0];

DataMemory ram();
Data_Cache cache(.ready(ready));

reg [1:0] state, next_state;
parameter [1:0] s1 = 2'b00, s2 = 2'b01, s3 = 2'b10, s4 = 2'b11;

/*always block statements to transfer between states*/
always @(posedge clk)
   if (ready == 0)
      state <= s1;
   else
      state <= next_state;

always @(state, ready, s1, s2, s3, s4)
begin
   case(state)
         s1: begin   /*In idle state, checks to see of the cache is ready or not*/
               if (ready == 1)
               begin
                  index = address[4:1];
                  tag = address[9:5];
                  next_state = s2;
               end
               else
               begin
                  next_state = s1;
               end
            end

         s2: begin   /*Check the dirty flag for specific memory address. Then, check
                  for a tag match. If tag matches, cahce hit data access and output
                  in same cycle and move to idle*/
               if(dirty[index] == 1)
                  begin
                     if (tag == prev_tag[index])
                        begin
                           if (R_W == 1)
                              //Read(load) from the memory and transfer to cache
                              begin
                                 cache.cache[index] = ram.ram[index];
                              end
                           else
                              //write(store) to the memory from cahce
                              begin
```

```
                    ram.ram[index] = cache.cache[index];
                  end
                next_state = s1;
              end
            else
              begin
                cache.tag = tag;
                cache.index = index;
                next_state = s3;
              end
          end
        else
          begin
            cache.tag = tag;
            cache.index = index;
            next_state <= s3;
          end
      end

    s3: begin    /*If no tag match or no dirty flag move to third state and save
            the previous tag in a register and move to fourth state*/
        prev_tag[index] = cache.tag;
        next_state <= s4;
      end

    s4: begin   //read/write
        if (R_W == 1)
          begin
          //Read the new data directly from the cache
            cache.cache[index] = data;
            dirty[index] = 1;
          end
        else
          begin
          //Write the new data into the memory and then write into cache
            ram.ram[index] = data;
            cache.cache[index] = ram.ram[index];
            dirty[index] = 1;
          end
        temp_out = cache.cache[index]; //temporary output value
        next_state <= s1; //Go to idle state
      end

    default:
        next_state <= s1;
endcase
```

```
    end

assign out = temp_out;

endmodule
```

Data Cache and Cache Controller testbench

```
module Cache_test();
    reg clk; reg R_W; reg ready;
    reg [9:0] data; reg [9:0] address;
    wire [9:0] out;

    Cache_Controller Cache_test(.clk(clk), .data(data), .address(address), .R_W(R_W), .out(out),
.ready(ready));

    initial
    begin
        clk = 1'b1;
        ready = 1'b0;

        #100

        address = 10'b0000000010;         // 0
        data =    10'b0011000011;      //0C3
        R_W = 1'b1;

        #25
        ready = 1'b1;

        #75
        address = 10'b1010100101;    //2A5  %  16
        data =    10'b0000100101;     //025    // 526421
        R_W = 1'b1;

        #100
        address = 10'b0000000010;         // 0
        data =    10'b0011000011;      //0C3
        R_W = 1'b0;

        #100
        address = 10'b1010100101;     //2A5  %  16
        data =    10'b0000100101;         //025
        R_W = 1'b0;

        #100
        address = 10'b0110101001;      //1A9  %  16
        data =    10'b1101101110;         // 26E
        R_W = 1'b1;

        #100
        address = 10'b0110101001;         // 1A9  % 16
        data =    10'b1101101110;         // 26E
        R_W = 1'b0;

        #100
        address = 10'b1010100101;         // 2A5  %  16
        data =    10'b0000100101;         // 025
        R_W = 1'b1;

        #100
        address = 10'b0000101100;         // 2C
        data =    10'b1111111111;         // 3FF
        R_W = 1'b1;

        #100
        address = 10'b0000101101;         // 2D
        data =    10'b0000000001;         // 1
        R_W = 1'b0;

        #100
        address = 10'b1010100101;         // 2A5
        data =    10'b0000000010;         // 2
        R_W = 1'b0;
    end

    always #12.5 clk = ~clk;

endmodule
```

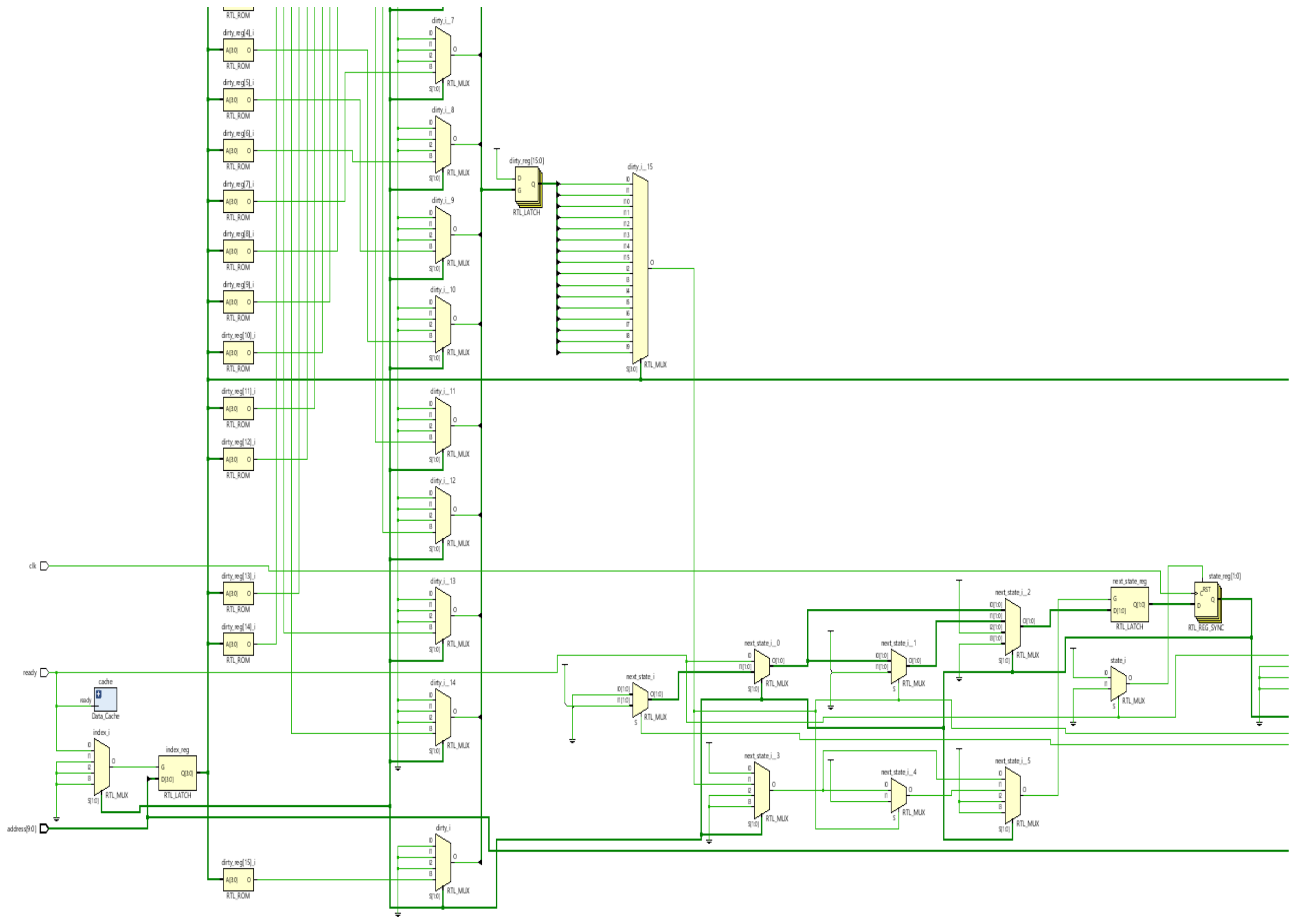


Figure 6: A part of the data cache and cache controller

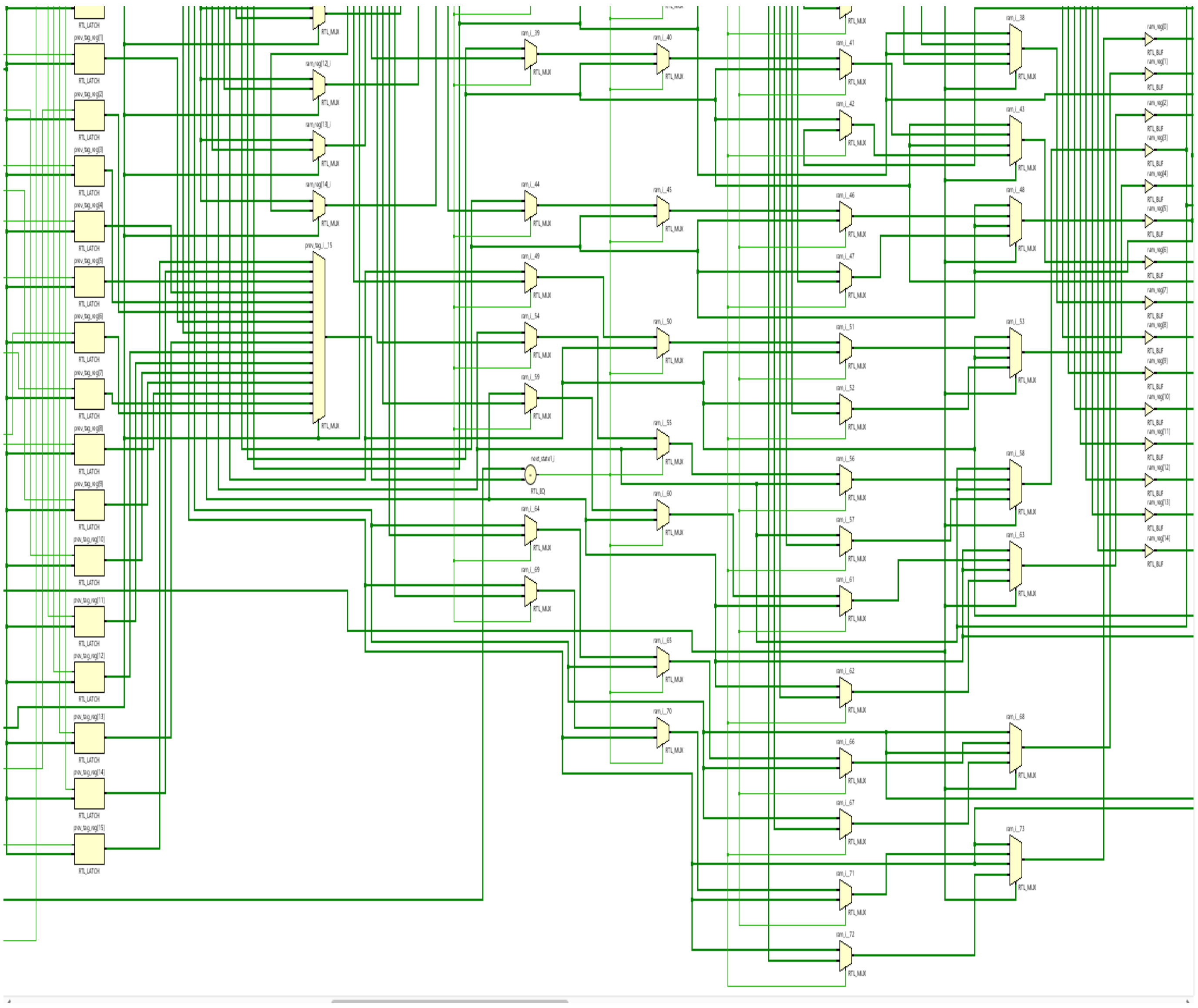


Figure 7: Another part of data cache and cache controller

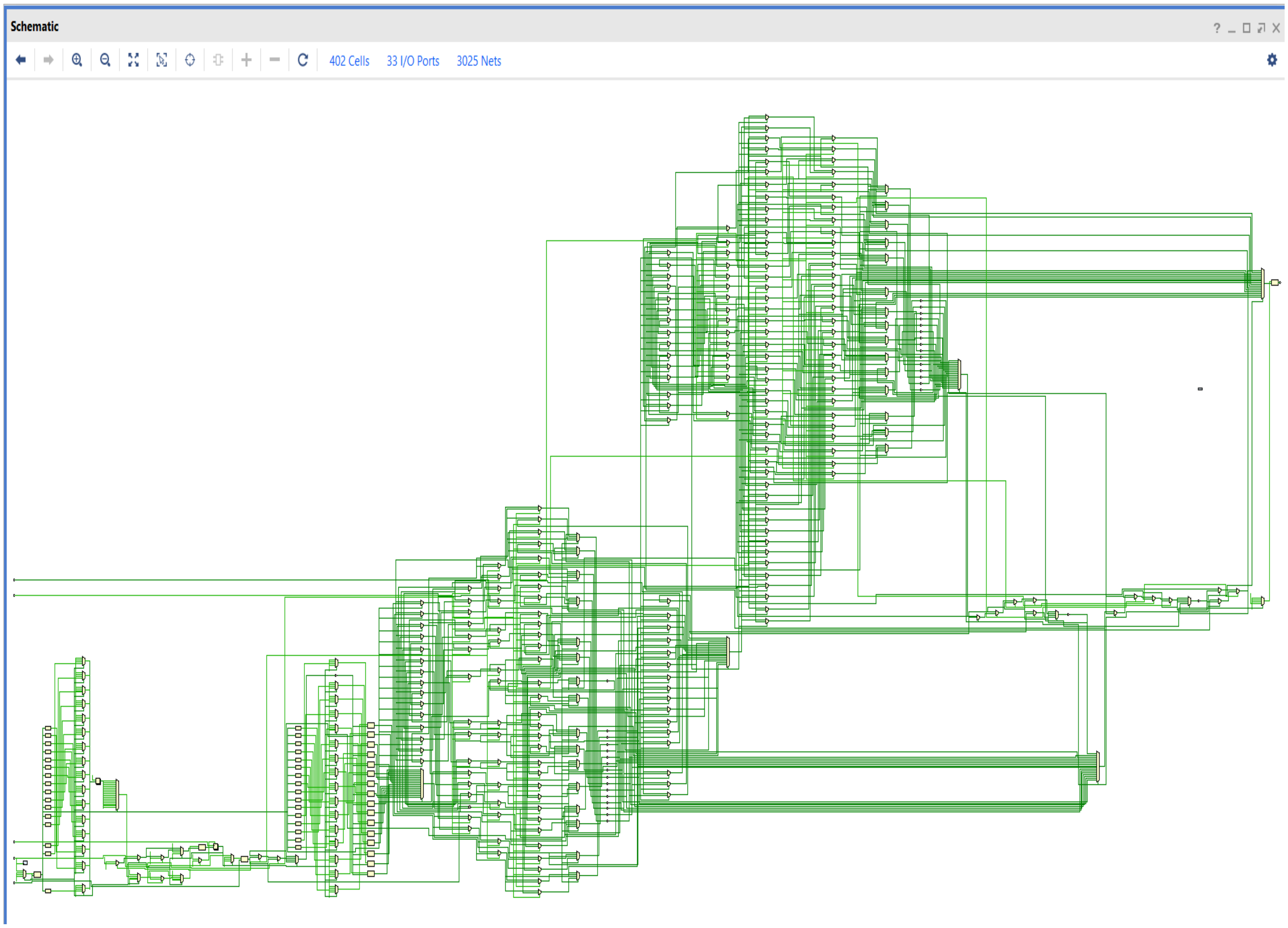


Figure 8: A complete schematic for data cache and cache controller

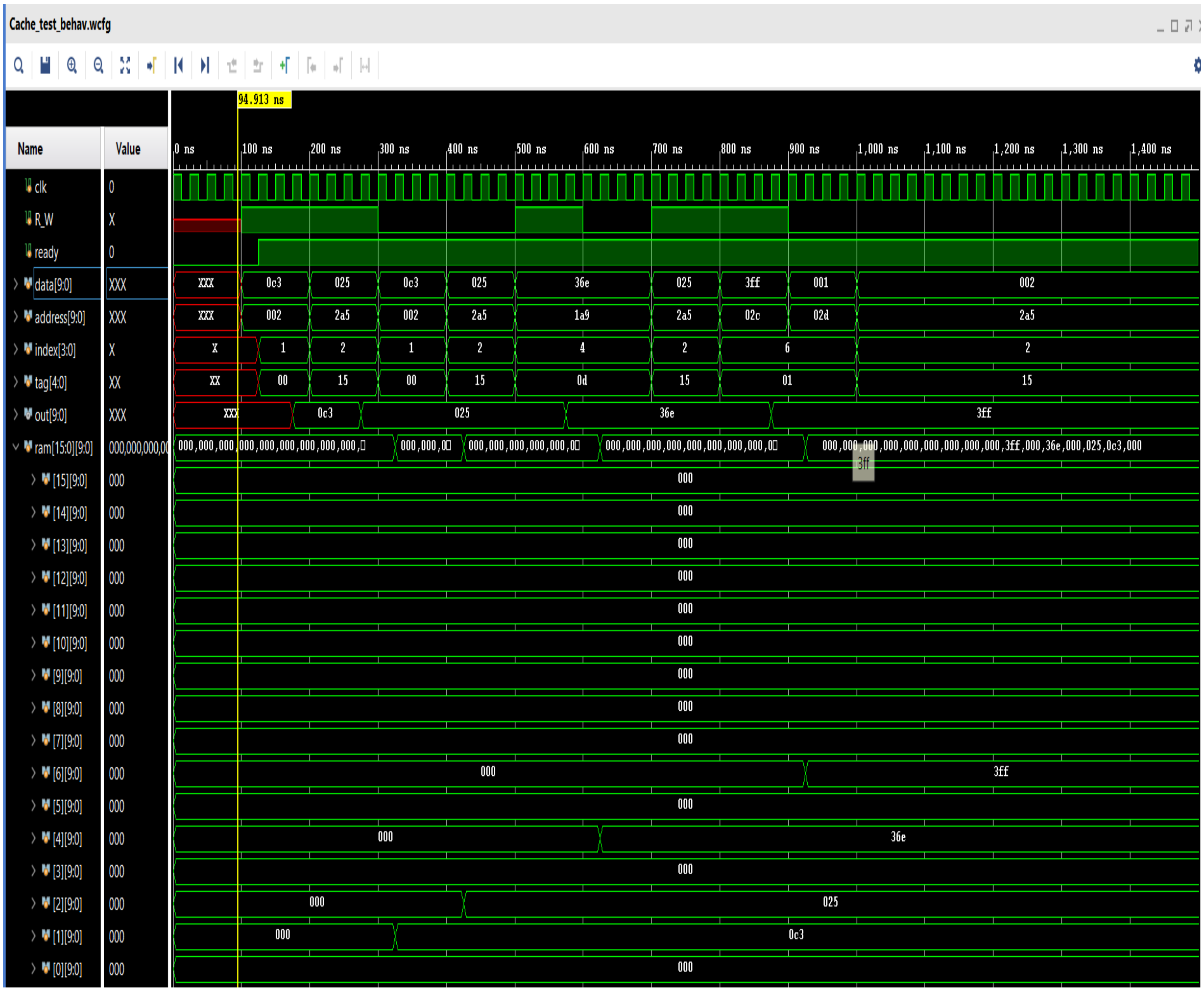


Figure 9: Timing diagram for data cache (first part)

The timing diagram is composed of several test cases for cache memory for both read and write from and into the cache memory. The cache memory has four cycles for any cache miss as data need to be fetched from main memory. In the case of cache hit, the cache request, cache access and data output will be done in a single cycle. For this cache memory test case, the cache hit test case runs two times to compensate the cycle. During compulsory cache miss, the new data is either loaded using main memory to cache or it is written into the main memory from cache. For cache miss due to no match of tag (the dirty flag is on), the cache reads the new data using main memory or the old data in maim memory is replaced by new data (for write). For the cache hit, old data is read or written using only cache memory in a single cycle.

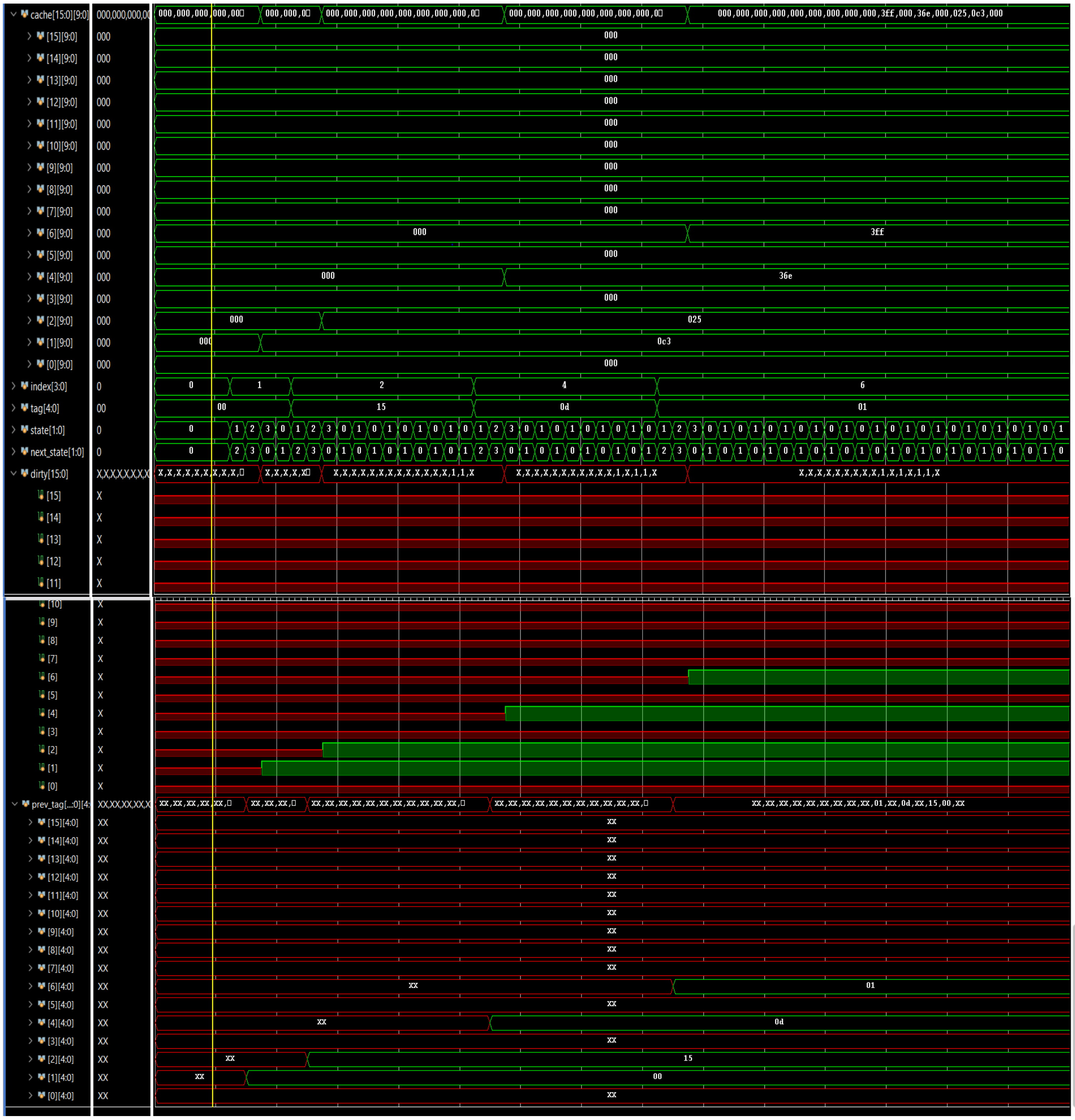


Figure 10: Timing Diagram for data cache (second part)

Data Cache Implementation for 10-bit CPU complete code and testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/09/2020 03:40:53 AM
// Design Name:
// Module Name: Data_Cache
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module Cache_Controller(input clk, input [9:0] data, input [9:0] address, input R_W, output
[9:0] out, input ready);

   reg [9:0] temp_out;
   reg [3:0] index; // for keeping index of current address
   reg [4:0] tag;  // for keeping tag of current address
   reg dirty [15:0];
   reg [4:0] prev_tag [15:0];

   DataMemory ram();
   Data_Cache cache(.ready(ready));

   reg [1:0] state, next_state;
   parameter [1:0] s1 = 2'b00, s2 = 2'b01, s3 = 2'b10, s4 = 2'b11;

   /*always block statements to transfer between states*/
   always @(posedge clk)
      if (ready == 0)
         state <= s1;
      else
         state <= next_state;

   always @(state, ready, s1, s2, s3, s4)
```

```
begin
   case(state)
         s1: begin   /*In idle state, checks to see of the cache is ready or not*/
               if (ready == 1)
               begin
                  index = address[4:1];
                  tag = address[9:5];
                  next_state = s2;
               end
               else
               begin
                  next_state = s1;
               end
            end

         s2: begin   /*Check the dirty flag for specific memory address. Then, check
                  for a tag match. If tag matches, cahce hit data access and output
                  in same cycle and move to idle*/
               if(dirty[index] == 1)
                  begin
                     if (tag == prev_tag[index])
                        begin
                           if (R_W == 1)
                              //Read(load) from the memory and transfer to cache
                              begin
                                 cache.cache[index] = ram.ram[index];
                              end
                           else
                              //write(store) to the memory from cahce
                              begin
                                 ram.ram[index] = cache.cache[index];
                              end
                           next_state = s1;
                        end
                     else
                        begin
                           cache.tag = tag;
                           cache.index = index;
                           next_state = s3;
                        end
                  end
               else
                  begin
                     cache.tag = tag;
                     cache.index = index;
                     next_state <= s3;
```

```
              end
          end

        s3: begin    /*If no tag match or no dirty flag move to third state and save
                the previous tag in a register and move to fourth state*/
            prev_tag[index] = cache.tag;
            next_state <= s4;
          end

        s4: begin   //read/write
            if (R_W == 1)
              begin
              //Read the new data directly from the cache
                cache.cache[index] = data;
                dirty[index] = 1;
              end
            else
              begin
              //Write the new data into the memory and then write into cache
                ram.ram[index] = data;
                cache.cache[index] = ram.ram[index];
                dirty[index] = 1;
              end
            temp_out = cache.cache[index]; //temporary output value
            next_state <= s1; //Go to idle state
          end

        default:
            next_state <= s1;
    endcase
  end

assign out = temp_out;

endmodule

module Data_Cache(input [9:0] data, input [9:0] address, input R_W, input ready);

    reg [9:0] cache [15:0]; //registers for the data in cache
    reg [4:0] tag;
    reg [3:0] index;
    reg offset;

initial
  begin: initialization
    integer i;
```

```
        for (i = 0; i < 16; i = i + 1)
            begin
                cache[i] = 10'b0000000000;
                tag = cache[i][9:5];
                index = cache[i][4:1];
                offset = cache[i][0];
            end
    end
endmodule

module DataMemory(input clk, input [3:0] address, input [9:0] write_data, input write_en, input
mem_read,
                    output [9:0] read_data);
        integer i;
        reg [9:0] ram [15:0];
        wire [3:0] ram_address = address;

        initial begin
            for(i = 0; i < 16; i = i + 1)
                ram[i] <= 10'b0000000000;
        end

        always @(posedge clk) begin
            if (write_en)
                ram[ram_address] <= write_data;
        end
        assign read_data = (mem_read) ? ram[ram_address] : 10'b0000000000;
endmodule
```

```
module Cache_test();
   reg clk; reg R_W; reg ready;
   reg [9:0] data; reg [9:0] address;
   wire [9:0] out;

   Cache_Controller Cache_test(.clk(clk), .data(data), .address(address), .R_W(R_W), .out(out),
.ready(ready));

   initial
   begin
      clk = 1'b1;
      ready = 1'b0;

      #100

      address = 10'b0000000010;        // 0
      data =    10'b0011000011;     //0C3
      R_W = 1'b1;

      #25
      ready = 1'b1;

      #75
      address = 10'b1010100101;   //2A5  %  16
      data =    10'b0000100101;    //025    // 526421
      R_W = 1'b1;

      #100
      address = 10'b0000000010;        // 0
      data =    10'b0011000011;     //0C3
      R_W = 1'b0;

      #100
      address = 10'b1010100101;    //2A5  %  16
      data =    10'b0000100101;        //025
      R_W = 1'b0;

      #100
      address = 10'b0110101001;     //1A9  %  16
      data =    10'b1101101110;        // 26E
      R_W = 1'b1;

      #100
      address = 10'b0110101001;        // 1A9  % 16
      data =    10'b1101101110;        // 26E
      R_W = 1'b0;
```

```
        #100
        address = 10'b1010100101;        // 2A5  %  16
        data =    10'b0000100101;        // 025
        R_W = 1'b1;

        #100
        address = 10'b0000101100;        // 2C
        data =    10'b1111111111;        // 3FF
        R_W = 1'b1;

        #100
        address = 10'b0000101101;        // 2D
        data =    10'b0000000001;        // 1
        R_W = 1'b0;

        #100
        address = 10'b1010100101;        // 2A5
        data =    10'b0000000010;        // 2
        R_W = 1'b0;
    end

    always #12.5 clk = ~clk;

endmodule
```