Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science

6.111 - Introductory Digital Systems Laboratory

Problem Set 2 Solutions

Contact: Cemal Akcaba <akcabac@mit>

Problem 1: Counters

Part (a)

128 is the seventh power of 2. Hence, cascading two 74LS163s as you have done in the laboratory 1 would be sufficient.The third output bit of the higher order 74LS163 is the 14.4KHz clock. Figure 1 shows the wiring diagram.

Part(b)

We would like to divide our clock by 1843200. 1843200 in decimal is 1C2000 in hexadecimal. We need 21 bits to represent 0h1C2000, so we will need to count to a 21 bit number with our 163s. Therefore, we need at least **6** 74LS163s to produce a 1Hz clock.

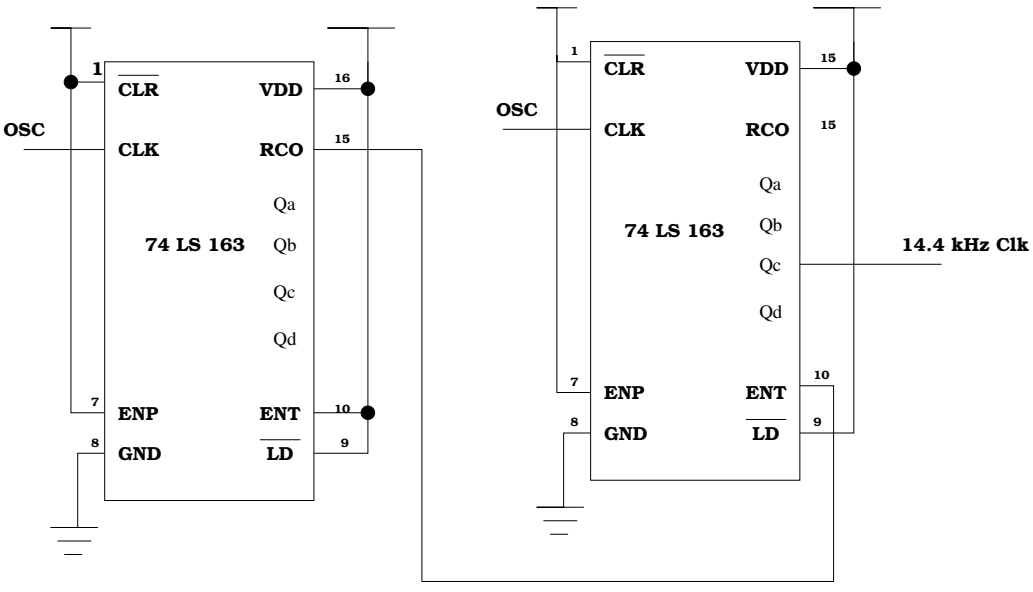


Figure 1: Cascading 74LS163s

Part(c)

74LS163 is a synchronous 4-bit counter, in which all output bits update at the rising edge clock. 74LS393 is a asynchronous 4-bit counter, in which lower order output bits update before the higher order bits. There are 4 parallel J-K registers within a 74LS163 that drive the output bits $Q_A..Q_D$. These parallel J-K registers are all clocked with the same clock signal, and hence all of them update at the same instant (as all the J-K registers have the same amount of clock to Q delay. In the architecture of 74LS393s, 4 **serial** T-registers are used. Hence, the higher order outputs can only change after the lower output bits have updated.

Part(d)

Verilog code for 1 Hz timer is shown below. As instructed in the problem set, your solution to this part should not be a 1Hz 50% duty cycle clock signal. Instead, we wanted you to create a pulse of one clock period width once with a frequency of 1 second.

```
module one_hz(clk, reset, one_hz_clk);
input clk;
input reset;
output one_hz_clk;
reg one_hz_clk;
reg [20:0] cnt;
//1.8432MHZ = 0h1C2000

always @(posedge clk)
  begin
  if(reset == 1)
   begin
    cnt <= 21'd0;
    one_hz_clk <= 1'b0;
   end
  else if(cnt == 21'd1843199)
// else if (cnt == 21'd0000002) => shorter cycle for test purposes.
   begin
     one_hz_clk <= 1'b1;
     cnt <= 21'd0;
   end
  else
   begin
     cnt <= cnt + 1;
     one_hz_clk <= 1'b0;
   end
end
endmodule
```

Problem 2:
Part(a)
State transition diagram.

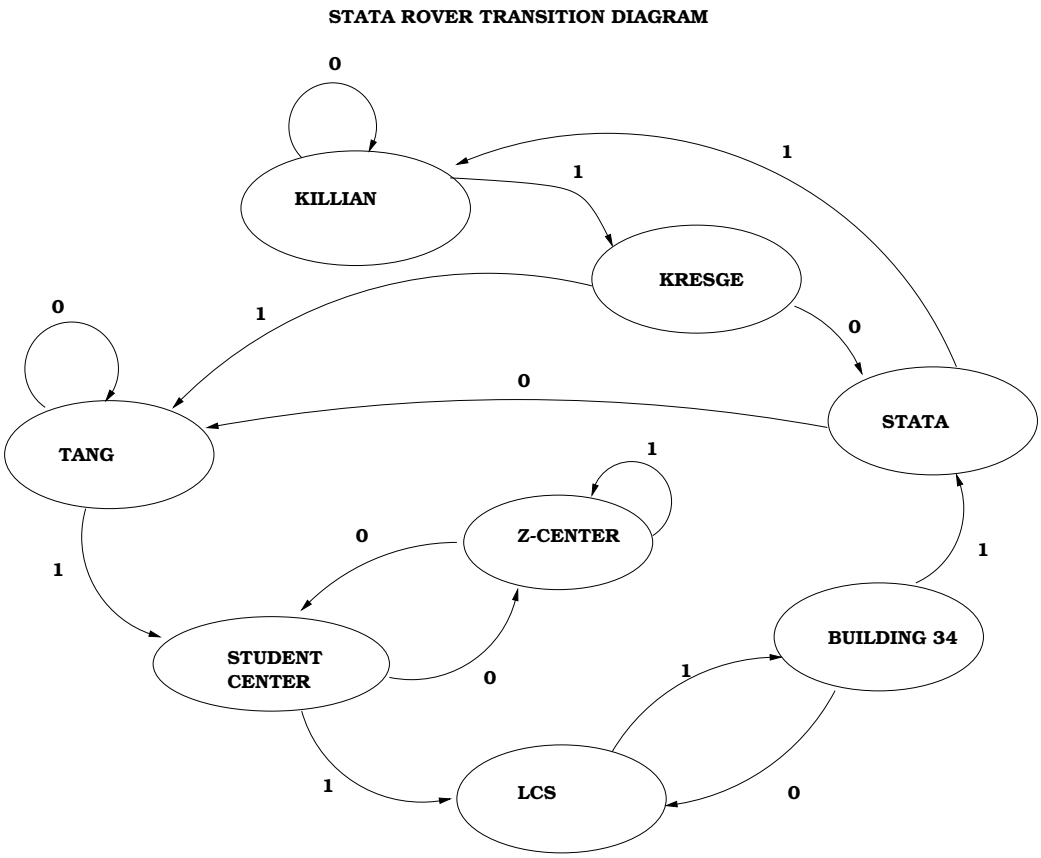


Figure 2: Transition diagram

Part(b)
Assuming the rover will start from Killian:
(Killian)->Kresge, Stata, Tang, Tang, Student Center

Part(c)
Z-Center. There is no arrow with an input '1' to Z-Center from other states. So starting from any other state, with all '1's as inputs the rover cannot ever reach the Z-Center.

Part(d)

Verilog code for StataRover.v

```
module stataRover (clk, fsmreset, fsminput, state);
// System Clk
input clk;
// Global Reset signal
input fsmreset;
input fsminput;
//
output[2:0] state;
// internal state
reg [2:0] state;
reg [2:0] nextstate;

// State declarations.
parameter KILLIAN = 0;
parameter KRESGE = 1;
parameter TANG = 2;
parameter ZCENTER = 3;
parameter STUDENTCENTER = 4;
parameter BUILDING34 = 5;
parameter LCS = 6;
parameter STATACENTER= 7;

always @ (posedge clk)
  begin
   if (fsmreset) state<= KILLIAN;
   else state<=nextstate;
  end
always @ (state or fsminput) begin
  nextstate = 3'b000;
  case (state)
     KILLIAN: if(fsminput) nextstate = KRESGE;
     else nextstate = KILLIAN;

     KRESGE: if(fsminput) nextstate = TANG;
     else nextstate = STATACENTER;

     TANG: if(fsminput) nextstate = STUDENTCENTER;
     else nextstate = TANG;

     ZCENTER: if(fsminput) nextstate = ZCENTER;
     else nextstate = KILLIAN;

     STUDENTCENTER: if(fsminput) nextstate = LCS;
     else nextstate = ZCENTER;

     BUILDING34: if(fsminput) nextstate = STATACENTER;
     else nextstate = LCS;


     LCS: if(fsminput) nextstate = BUILDING34;
     else nextstate = LCS;


     STATACENTER: if(fsminput) nextstate = KILLIAN;
     else nextstate = TANG;
   endcase // case(state)
  end // always @ (state or fsminput)
endmodule //
```

Problem 3:

The code for top level module(top.v) and the tesbench(tbRover.v). For simulation purposes, a 9.2 kHz enable signal was used instead of a 1 Hz enable signal.

top.v

```
module top(clk, reset, fsminput, fsmreset, state);
 input clk, reset, fsminput, fsmreset;
 output[2:0] state;
 wire one_hz_clk;
 one_hz one_hz1(clk, reset, one_hz_clk);
 stataRover rover1(one_hz_clk, fsmreset, fsminput, state);
endmodule
```

tbRover.v

```
'timescale 1ns/10ps
module tbRover;
 reg clk;
 reg reset;
 reg fsminput;
 reg fsmreset;
 wire [2:0] fsm_state;
 top top1(.clk(clk), .reset(reset), .fsminput(fsminput), .fsmreset(fsmreset), .state(fsm_state));

 initial
  begin
   clk = 0;
   reset = 0;
   fsminput = 0;
   fsmreset = 0;
   // #20
   #1084
   reset = 1;
   fsmreset =1;
   // #20
   #1084
   reset = 0;
   fsmreset = 0;
   // #40
   #2168
   fsminput = 1;
   // #20
   #1084
   fsminput = 0;
   // #100
   #5420
   fsminput = 1;
 end

 always #271 clk= ~clk;
endmodule
```

The screenshot of the simulation.

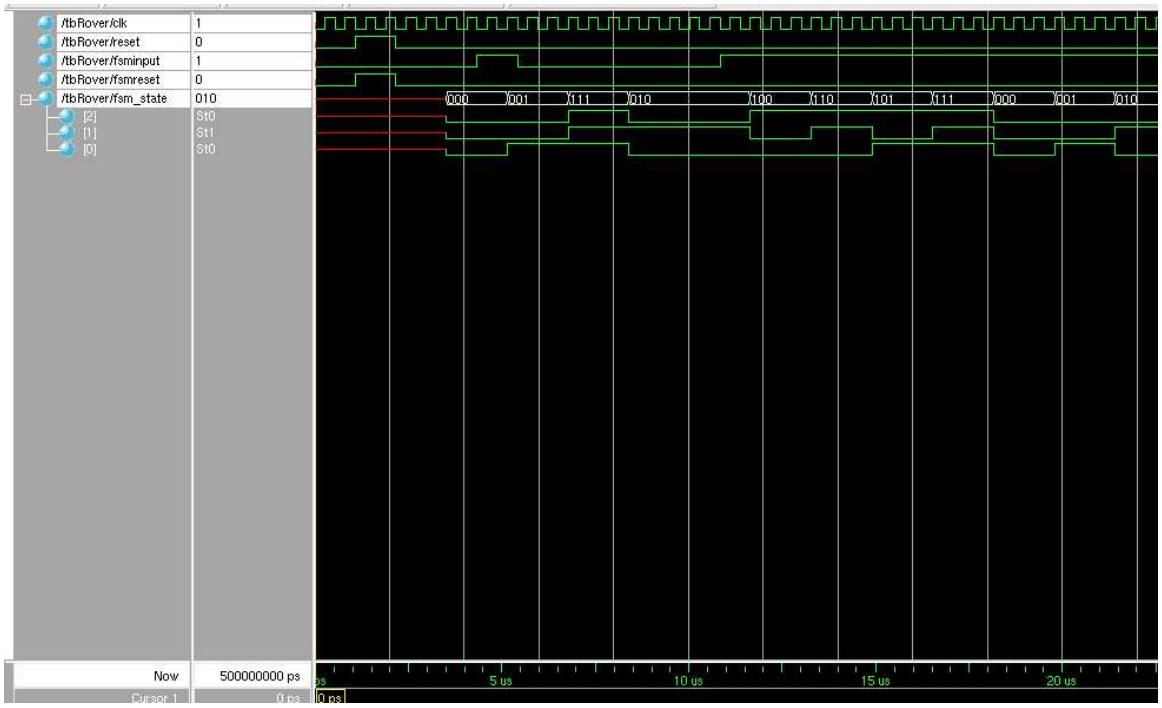
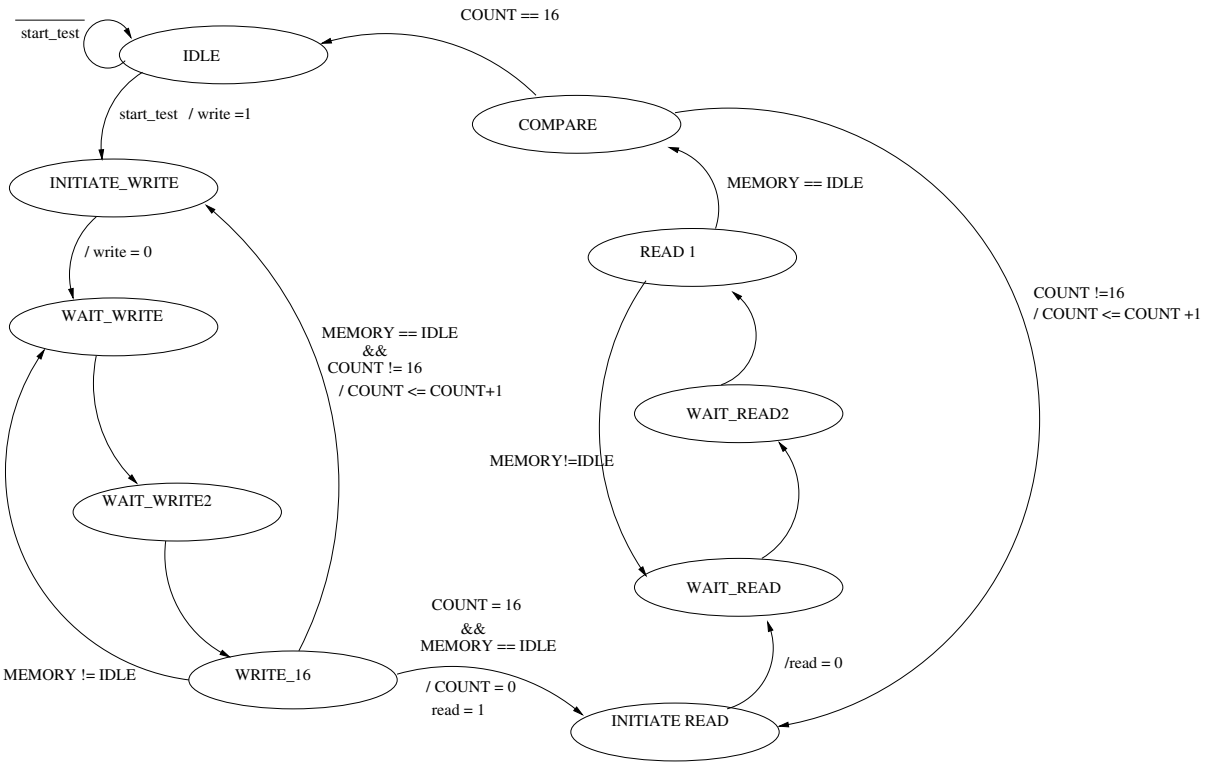


Figure 3: Screenshot

Problem 4:

We are going to use a major-minor FSM scheme to implement this memory controller. Basically, we will write a 'major FSM' to control the multi-cycle access FSM shown in lecture 7. The major FSM will use the multi-cycle access FSM to write to the memory, and then to read back from the memory. When the multi-cycle access FSM is carrying out a read or write operation the major FSM will have to wait for the completion of the operation. There are many ways to this, if you draw the state transition diagram for the multi-cycle access FSM, you will realize that after a read or a write operation it goes back to the idle state, hence we can use the 'state' output of the multi-cycle access FSM to tell when a read or a write operation is completed. The state transition diagram for the 'major FSM' is as follows:



Note: This state diagram is different in terms of structure than the one implemented in the code for illustrative purposes. The outputs are updated during transitions in this diagram, but the outputs are updated within the states in the verilog code. (Now, what kind of state machine does the diagram represent? How about the code?)

Figure 4: Memory Tester State Transition Diagarm

memory_tester.v (major FSM):

```
// Module for testing a memory.
    module memory_tester(clk, reset, start_test, failure, success, read_data, ext_data, state, mem-
ory_state, count, failure_count);
    output [2:0] state; // state of the memory tester
    output [3:0] count; // the address for writing
    output [3:0] failure_count; // number of failures
    // state of the memory controller
    output[2:0] memory_state;
    output [7:0] read_data;
    // output signals
    output failure,success;


    input clk;
    input reset;
    // we are going to sit in the idle state until
    // this signal is asserted.
    input start_test;


    reg [3:0] failure_count;
    reg [3:0] failure_count_int;
    reg read, write, read_int, write_int;
    reg failure,success;
    reg failure_int;
    reg success_int;
    wire data_oen, data_sample, address_load;
    // state of the memory tester
    reg [3:0] state ,next;
    // address for reading/writing
    reg [3:0] count;
    reg [3:0] count_int;
    wire G_b, W_b;
    inout [7:0] ext_data;
    wire [12:0] ext_address;
    reg [7:0] write_data;
    reg [7:0] address;
    // we are going to use the simple multi-cycle read scheme shown
    // in the lecture. The state output of this module tells us
    // whether a read/write operation is complete.


memory_controller memory_controller1(
.clk(clk),.reset(reset),.G_b(G_b),.W_b(W_b),.address(address),ext_address(ext_address),.write_data(write_data),.read_data(read_data),
.ext_data(ext_data),.read(read),.write(write),.state(memory_state),.data_oen(data_oen),.address_load(addres_load),
.data_sample(data_sample));

    parameter IDLE = 0;
    parameter INITIATE_WRITE =1;
    parameter WRITE_WAIT = 2;
    parameter WRITE_WAIT2 = 3;
    parameter WRITE_16 = 4;
    parameter INITIATE_READ = 5;
    parameter WAIT_READ = 6;
    parameter WAIT_READ2 = 7;
    parameter READ_1 = 8;
    parameter COMPARE = 9;
    always @(posedge clk)
    begin
     if (!reset) state<= IDLE;
     else state <= next;
     failure <= failure_int;
     success <= success_int;
     // update the outputs
     failure_count <= failure_count_int;
     read <= read_int;
```

```
 write <= write_int;
 count <= count_int;
 address <= count;
 write_data <=count;
end // always @ (posedge clk)
always @ (state or start_test) begin
    failure_int = 0;
    success_int = 0;
    read_int = 0;
    write_int =0;
    case(state)
    IDLE:
     if (start_test) begin
     next = INITIATE_WRITE;
     write_int = 0;
     read_int = 0;
     count_int<=4'b000;
     failure_count_int <= 4'b0000;
    end
    else next=IDLE;

    INITIATE_WRITE: begin
    write_int =1;
    next = WRITE_WAIT;
    end

    WRITE_WAIT: begin
    write_int = 0;
    next = WRITE_WAIT2;
    end

    WRITE_WAIT2:begin
    write_int = 0;
    next = WRITE_16;
    end

    WRITE_16:
    if (memory_state == 4'b000) begin
      if (count == 4'b1111 ) begin
        next = INITIATE_READ;
        count_int <= 4'b0000;
      end
     else
       begin
        next = INITIATE_WRITE;
        count_int <= count + 1;
       end
    end
    else begin
     next = WRITE_WAIT;
    end // else (memory_state = 4'b000)

    INITIATE_READ:
    begin
    read_int = 1;
    write_int = 0;
    next = WAIT_READ;
    end

    WAIT_READ:
    begin
    next = WAIT_READ2;
    read_int = 0;
    end

    WAIT_READ2:
    begin
    next = READ_1;
```

```
    end

    READ_1:
    begin
    if (memory_state == 4'b0000)
    next = COMPARE;
    else
    next = WAIT_READ; // wait till read is done
    end

    COMPARE:
    begin
     if (address == read_data)
      success_int = 1;
    else begin
      failure_int = 1;
      failure_count_int <= failure_count_int+1;
    end
    if (count >= 4'b1111)
      next = IDLE;
    else begin
      next = INITIATE_READ;
      count_int = count + 1;
    end
    end

default: next = IDLE;
endcase // case(state)
end // always @ (state or start_test)
endmodule // memtest
```

memory_controller.v (multi-cycle access FSM)

```
// Verilog for Simple Multi-Cycle Access
// 6.111 Lecture 7
module memory_controller(clk, reset, G_b, W_b, address,
ext_address, write_data, read_data, ext_data, read,
write, state, data_oen, address_load, data_sample);

input clk, reset, read,write;
output G_b,W_b;
output [12:0] ext_address;
reg [12:0] ext_address;
input [12:0] address;
input [7:0] write_data;

output [7:0] read_data;
reg [7:0] read_data;
inout [7:0] ext_data;
reg [7:0] int_data;
output [2:0] state;
reg [2:0] state,next;
output data_oen, address_load, data_sample;
reg G_b, W_b, G_b_int, W_b_int, address_load,
data_oen, data_oen_int, data_sample;
wire [7:0] ext_data;

parameter IDLE = 0;
parameter write1 = 1;
parameter write2 = 2;
parameter write3 = 3;
parameter read1 = 4;
parameter read2 = 5;
parameter read3 = 6;

//Sequential always block for state assignment
assign ext_data = data_oen ? int_data : 8'bZ;
```

```
always @(posedge clk)
 begin
   if (!reset) state <= IDLE;
   else state <= next;
   G_b <= G_b_int;
   W_b <= W_b_int;
   data_oen <=data_oen_int;
   if (address_load) ext_address <= address;
   if (data_sample) read_data <= ext_data;
   if (address_load) int_data <= write_data;
  end // always @ (posedge clk)
   // note that address_load and data_sample are not
   // registered signals.
   // Combinational always block for next-stae
   // computation
always @ (state or read or write) begin
   W_b_int = 1;
   G_b_int = 1;
   address_load = 0;
   data_oen_int = 0;
   data_sample =0;
   case(state)
   IDLE:
      if (write) begin
      next = write1;
      address_load=1;
      data_oen_int=1;
      end
      else if (read) begin
      next = read1;
      address_load =1;
      G_b_int =0;
      end
      else next=IDLE;

   write1:
      begin
      next =write2;
      W_b_int =0;
      data_oen_int=1;
      end

   write2:
      begin
      next = write3;
      data_oen_int=1;
      end

   write3:
      begin
      next = IDLE;
      data_oen_int=0;
      end

   read1:
      begin
      next = read2;
      G_b_int = 0;
      data_sample = 1;
      end

   read2:
      begin
      next = read3;
      end

   read3:
      begin
```

```
      next = IDLE;
      end
      default: next=IDLE;
    endcase; // case(state)
 end // always @ (state or read or write)
endmodule // memtest
```

tb_memory_tester.v (testbench)

```
// Testbench for memory_tester.
// This testbench will produce only 1 success.
// All the other reads will result in a failure.
module tb_memory_tester;

reg clk;
reg reset;
reg start_test;
wire success, failure;
wire [7:0] ext_data;
wire [12:0] ext_address;
wire [7:0] read_data;
wire [3:0] failure_count;
wire [3:0] state;
wire [2:0] mem_state;
wire [3:0] count;
reg enable_data;
reg [7:0] data_out;

assign ext_data = enable_data ? data_out:8'hZ;

memory_tester mtest1(clk, reset, start_test, failure, success, read_data,ext_data, state,mem_state,count,failure_count);

always #25 clk = ~clk ;

initial begin
 clk =0;
 reset = 0;
 start_test = 0;
 #75
 reset = 1;
 #50
 start_test = 1;
 data_out =0;
 enable_data =0;
 #50
 data_out=8'h0;
 #5700
 data_out = 8'h0;
 enable_data =1;
 end

endmodule
```

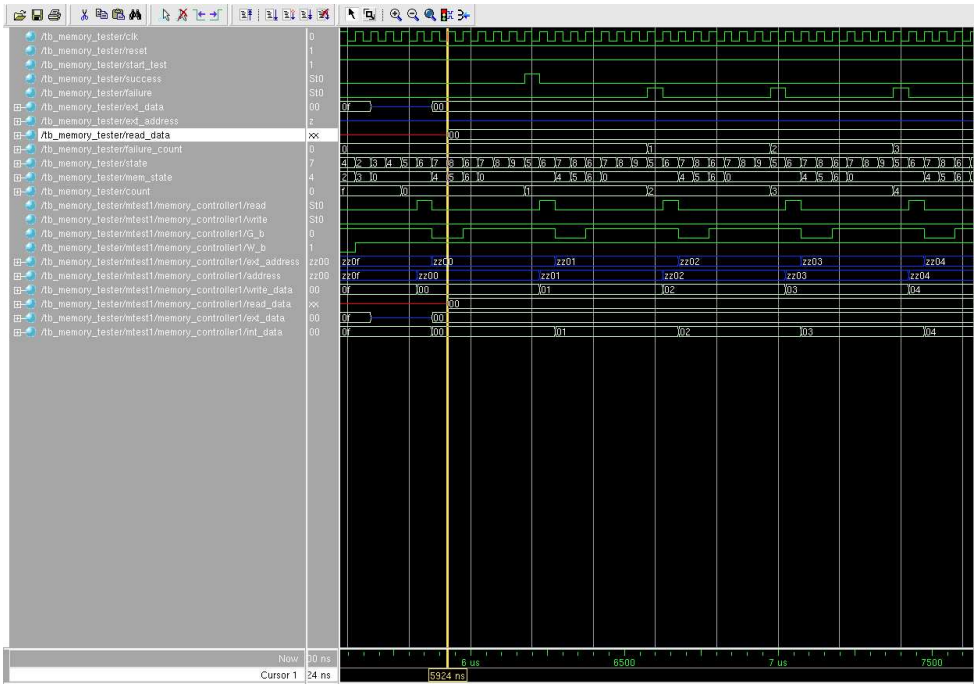


Figure 5: Screenshot of Memory Tester