EEE3096S: Embedded Systems II

LECTURE 28:

STATE MACHINES

Presented by:

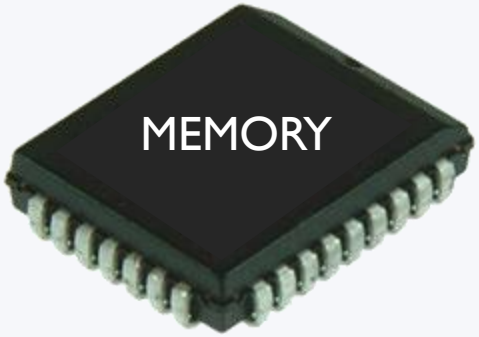
STANLEY MBEWE

Electrical Engineering
University of Cape Town

OUTLINE OF LECTURE

- Review of HDL Memory Design problem
- Implementing a Finite State Machine (FSM)
- Example state machine in Verilog
- Testbench for a state machine

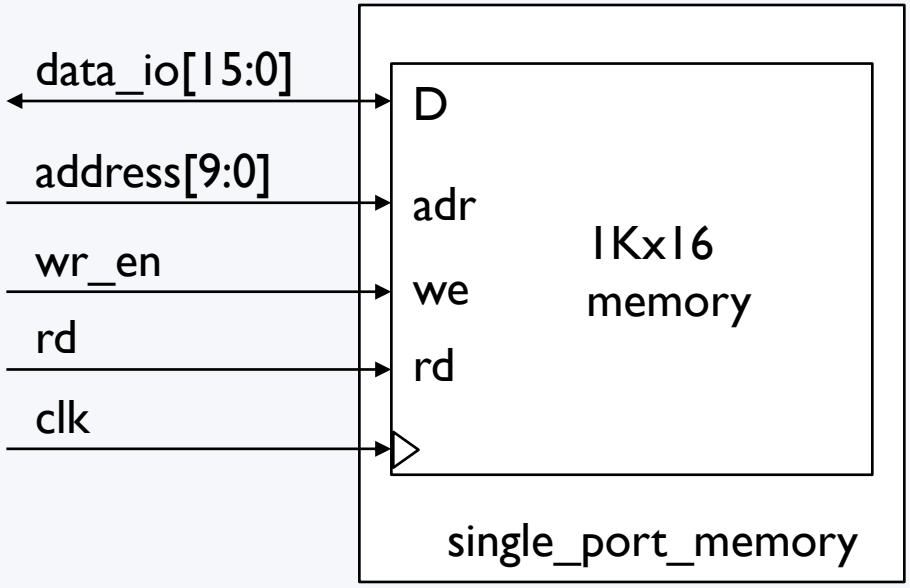
Memory in HDL

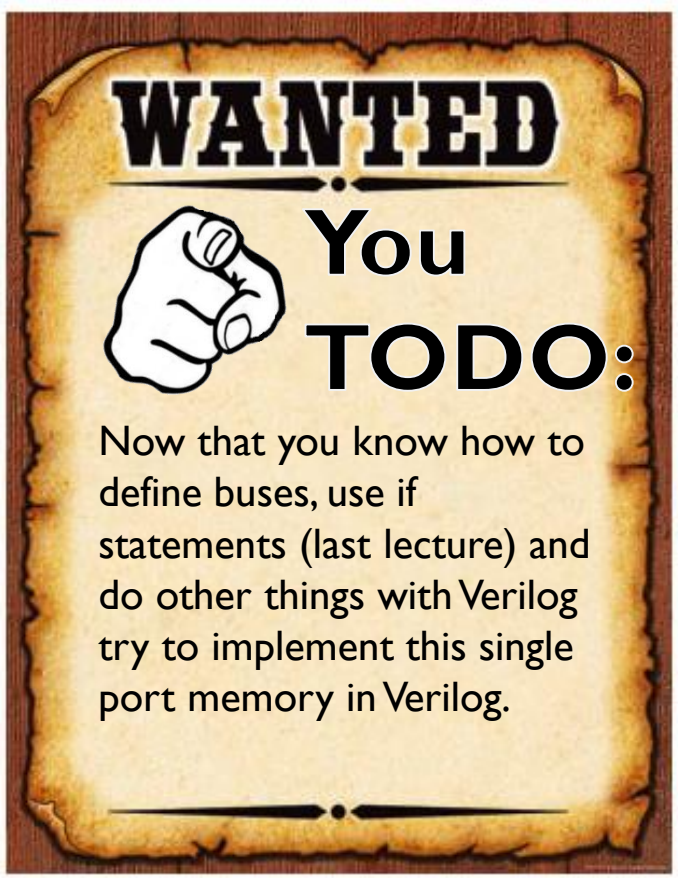


CLASS ACTIVITY: HDL MEMORY

Verilog Activity 2:

For this class activity consider this single port memory component below…





Operation: As you know from previous lectures, with this type of memory design you can only read or write at one time. The design is such that it does not do anything unless the wr_en is set (causing data to be sent to memory) or rd is set and data is read from the given address. If both we_en and rd are low or they are both high then nothing happens.

Hint: a *inout* defines a tristate, can be either input or output.

STEPS TO UNDERTAKE FOR CLASS ACTIVITY #2

- Design your module (you already know the ports but write out the interface)
- Understand how it is going to work
- Implement the one functionality to write (then maybe test it, e.g. dry run on paper)
- Implement the other functionality to read (then test it on paper)
- Consider handling exception/illegal states
- If there is time plan and develop test benches

Later, once you've had a change to try this on your own I'l release my sample solution and we will discuss it in next lecture

DEFINING THE MODULE'S PORT INTERFACE

```
//-----------------------------------------//
// Single-port Memory                      //
//-----------------------------------------//

module single_port_memory (
 // ------ system inputs -------------
 clk,  // technically this is synchronous memory
       // because a clock is used
 reset,
 // ---------- inouts ---------------
 data_io,
 // ---------- inputs ---------------
 address,
 wr_en,
 rd
 );
```

UNDERSTAND HOW IT IS GOING TO WORK…

- So, any operations will be ignored (dataio should be Z) if there is no data being loaded in or out of the memory device.
- If wr_en is set, then
  - data should be copied from dataio to the right memory location
- If rd is set, then
  - Get data from memory and copy it to local register, delay a moment while this happens
  - Then link dataio to this copy once you've got it out of memory
  - Possible add a dataread line for safety
- So, we may need:
  - word buffer for the read and
  - a read ready flag (these will be internal registers)

DEFINING THE MODULE'S OPERATION

```
module single_port_memory (…);
…
 // Internal data
 reg [15:0] memory[0:1023]; //wanted 1K of 16bit words
 reg [15:0] dat_out;   // stores data to be output
                              (more reliable)
 reg rd_d1; // delay mechanism
```

DEFINING THE MODULE'S OPERATION

```
module single_port_memory (…);
…
// ---------  implementation ---------
// Memory access statemachine (synchronized memory)
 always @( posedge clk )
  begin
   if (wr_en) // is write enabled?
     memory[address] <= data_io; // if so write to the memory
   dat_out <= memory[address];   // get the memory at the
                        // address (save an op, don't need else)
   rd_d1 <= rd;         // say data is ready to be latched,
                        // implement a delay due to read time
  end
 // only latch to the dataio when read is complete otherwise
 // connect to Z which is ofcourse overridden if another part
 // connects to high or low voltages.
 assign data_io = rd_d1 ? dat_out : 16'bz;
endmodule
```

COMPLETE SIMPLE PORT MEMORY SOLUTION

```
module single_port_memory (
 clk,  // technically this is synchronous memory because a clock is used
reset,  data_io,  address,  wr_en,  rd   );
 input clk, reset;            // system clock and reset
 inout [15:0] data_io;  // tristate input/output port
 input [9:0] address;  // address of memory to access
 input wr_en,rd;
 // Internal data
 reg [15:0] memory[0:1023]; // wanted 1K of 16 bit words
 reg [15:0] dat_out;   // stores data to be output (more reliable)
 reg rd_d1; // delay device
 // ---------  implementation ---------
always @( posedge clk )  begin
   if (wr_en) // is write enabled?
     memory[address] <= data_io; // if so write to the memory
   dat_out <= memory[address];   // get the memory at the address
   rd_d1 <= rd;                  // say data is ready to be latched
  end
 // only latch to the dataio when read is complete otherwise connect to Z.
 assign data_io = rd_d1 ? dat_out : 16'bz;
endmodule
```

IMPLEMENTING A FSM

We will cover this only very briefly, if you take EEE4120F you will get much more into FSMs and DFG etc/ on FPGAs.

Embedded Systems II

STATE MACHINES

- A state machine has:
  - Input events
  - Output events
  - Set of states
  - A function that maps
    (state,input) → (state,output)
  - A indication of the initial state
- A Finite State Machine (FSM) has a limited number of states

IMPLEMENTING A FSM WITH VERILOG

- The state machine needs a register to store its state
- It is sensitive to zero or more inputs, which can change state and/or produce an output

THE STATE REGISTER

- States could be numbered in sequence 0 .. 2^n-1 where n is the number of bits for the state.
- Or a different encoding / ordering scheme could be used to make state changes more robust, e.g.:
  - use of grey scale or ‘one hot’ encoding (where ‘one hot’ means there is just one pin in the state set at a time)

State	Binary	Gray	One Hot
0	3'b000	3'b000	8'b00000001
1	3'b001	3'b001	8'b00000010
2	3'b010	3'b011	8'b00000100
3	3'b011	3'b010	8'b00001000
4	3'b100	3'b110	8'b00010000
5	3'b101	3'b111	8'b00100000
6	3'b110	3'b101	8'b01000000
7	3'b111	3'b100	8'b10000000

THE STATES AND STATE CHANGES

- Need a means to force initial state at startup (i.e. reset state register)
  - Typically, a reset handler does this.
- Specifying and changing states
  - Usually, a case construct is used to define the states, but could use ifs and elses
- Need to decide if the state machine
  - is synchronous (clocked) or
  - asynchronous (activates whenever an input changes)
- May need recovery mechanism (e.g. watchdog or recovery default state)

EXAMPLE STATEMACHINE

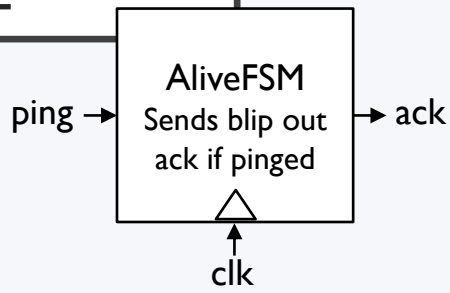
```
// Code your design here
module alivefms (input clk, input reset, input ping,
output reg ack,
       output reg[1:0] state);
 parameter [1:0] WAIT  = 2'b11;
 parameter [1:0] PING  = 2'b01;
 parameter [1:0] ACKN  = 2'b10;
 always @(posedge clk or posedge reset or
          posedge ping or negedge ping)
 begin
  if (reset)
  begin
     state <= WAIT;
     ack   <= 0;
  end else
  begin
    case(state)
      WAIT:    begin
               if (ping) state <= PING;
                   else  state <= WAIT;
               ack <= 0;
               end
      PING:    if (ping) state <= PING;
                   else  state <= ACKN;
      ACKN:    begin
                ack <= 1;
                state <= WAIT;
               end
      default: state <= WAIT;
    endcase
  end
end
endmodule
```

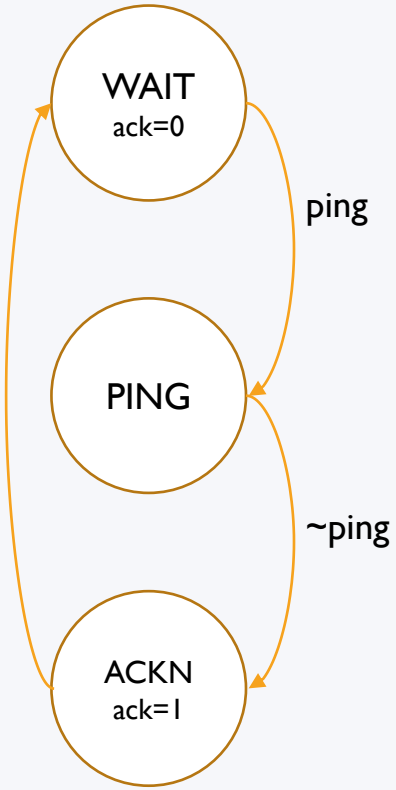
← Activation

← Start up

← States

← Recovery





By all means have a look at the approach, you

CREATE A TESTBENCH

```
// testbench for alivefsm
module alivefms_tb ();
reg  clk, reset, poke;
wire ack;
wire [1:0] state;
integer i;
  // instantiate the FSM
  alivefms alivefsm_tb (clk,reset,poke,ack,state);

initial  // method for testing the FSM:
begin
   // enable monitoring of wires of interest
  $monitor("reset=%d state=%d poke=%d ack=%d\n",
           reset,state,poke,ack);
   clk = 0; reset = 1; #5    // do the reset
   reset = 0; clk=0; #5
   poke = 1;  clk=1; #5   // poke the fsm
   poke = 0; #5           // release poke
   clk  = 0; #5           // needs a clk transition (i.e. get to ACK state)
   clk  = 1; #5
   if (ack == 1)          // check if worked as planned
      begin
        $display("SUCCESS\n");
      end
end
endmodule
```

← **poke is register in testbench to hook to ping input of FSM**

← **Instantiate aliveFSM**

Avail online at: https://www.edaplayground.com/x/25N5

Conclusion of HDL for EEE3096S !!