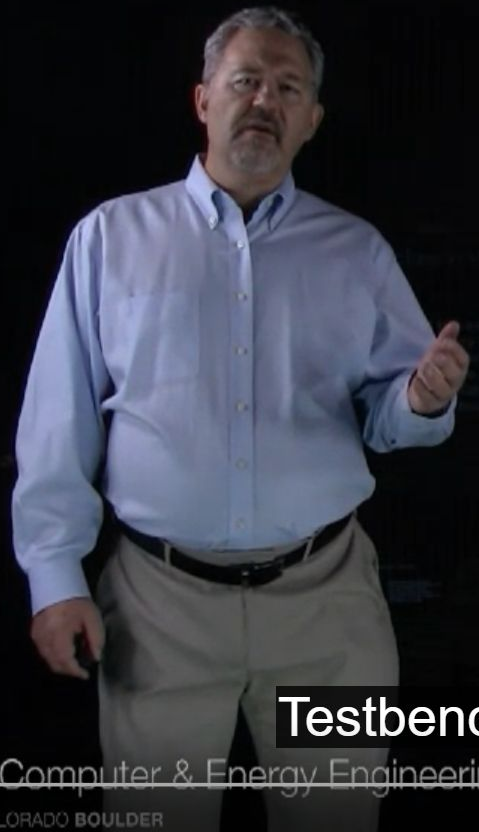
Testbenches in Verilog II

Professor Tim Scherr

University of Colorado **Boulder**

Testbenches in Verilog II

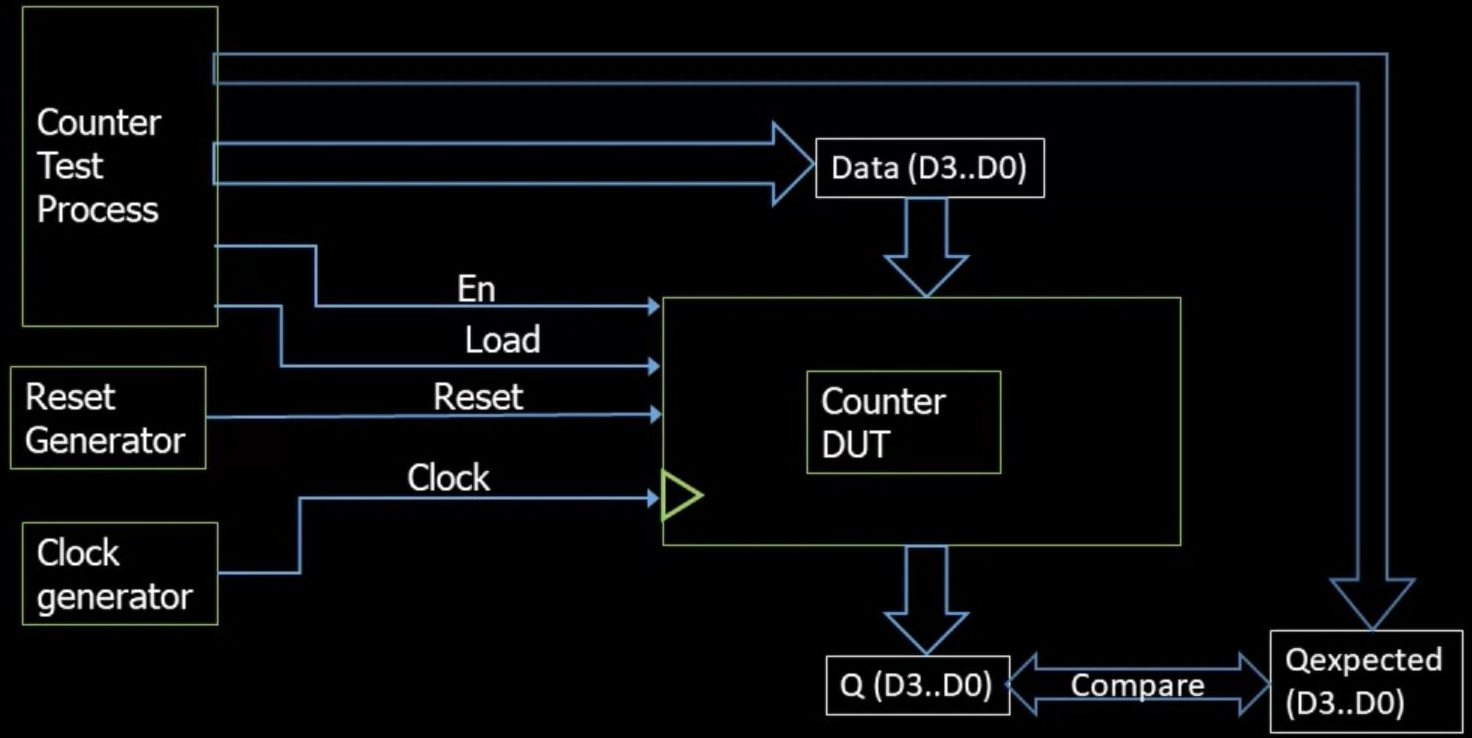


In this video, you will learn:

- **How to write simple testbenches for synchronous circuits**
- **How to use external signal generators to create stimulus for synchronous circuits.**

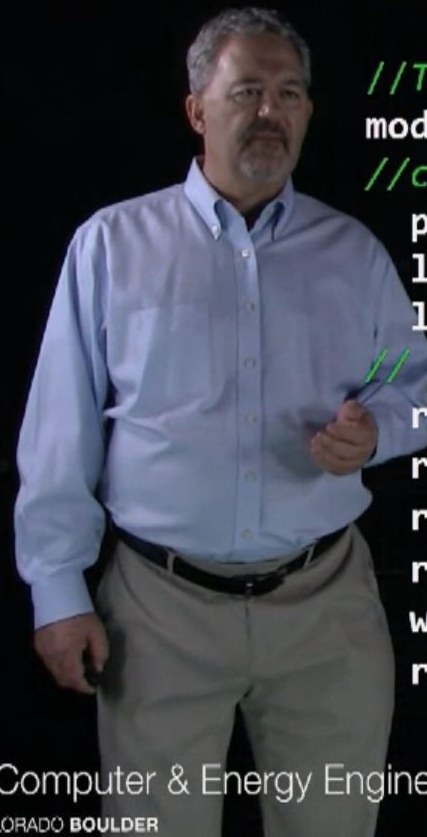
Testbenches are a very rich topic.

Counter Test Bench



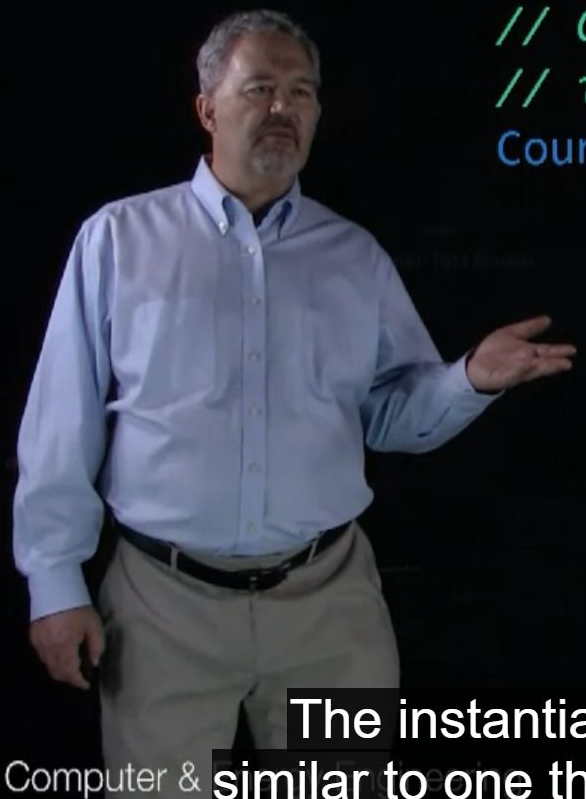
An additional complication comes from creating a testbench for

Counter Test Bench



```
timescale 1 ns / 1 ps    // set timescale
                         // to nanoseconds, ps precision
//Testbench entity declaration
module Counter_tb ();  // top level, no external ports
//constant declarations
  parameter delay = 10;  //ns  defines the wait period.
  localparam n = 4;  // width of counter in bits
  localparam T = 20;  // clock period
//  signal declarations
  reg clock = 0; //clock if needed, from generator model
  reg reset = 0;    // reset if needed
  reg [n-1:0] data_tb = 4'b0000;  // data input stimulus
  reg load = 0, en = 0;  // input stimulus
  wire [3:0] q;  // output to check
  reg [n-1:0] checkcount= 4'b0000;// variable to compare
                                  // to count
```

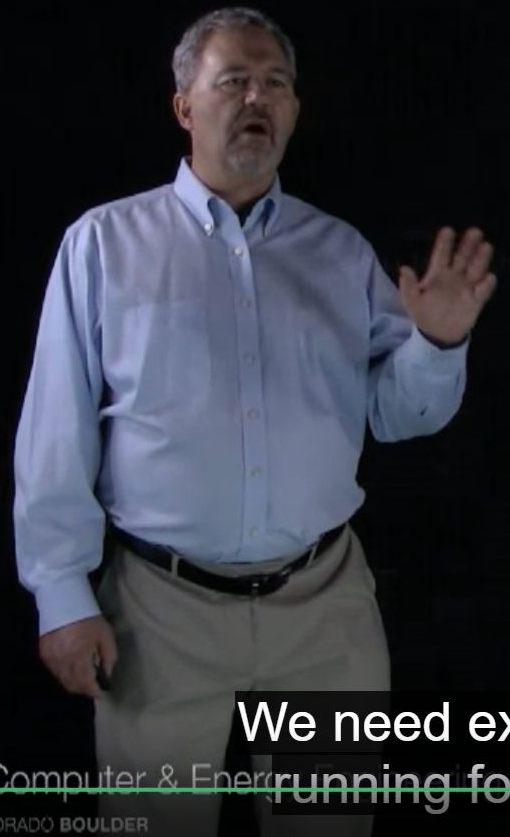
Counter Test Bench



```
// Component Instances
// instantiate the device under test
Counter DUT (     // Device under Test
        // Inputs
        .d(data_tb),
        .clk(clock),
        .reset(reset),
        .load(load),
        .en(en),
          // Outputs
        .q(q)
```

The instantiation of the counter is similar to one that we introduced before.

Counter Test Bench

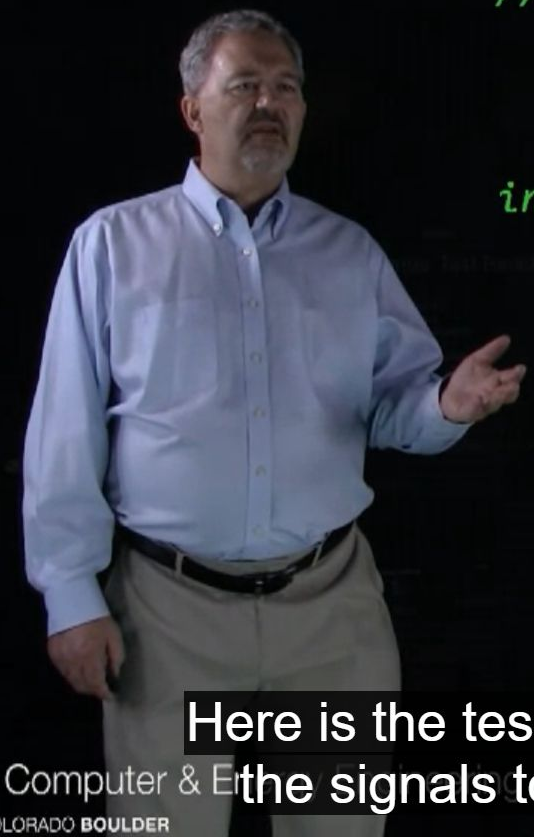


```
// External Device Simulation Processes
// clock driver
  always
  begin
    clock = 1'b1;
    #(T/2);
    clock = 1'b0;
    #(T/2);
  end

// reset driver
  initial
  begin
    reset = 1'b1;
    #(T/2);
    reset = 1'b0;
```

We need external generator models running for the clock in the reset.

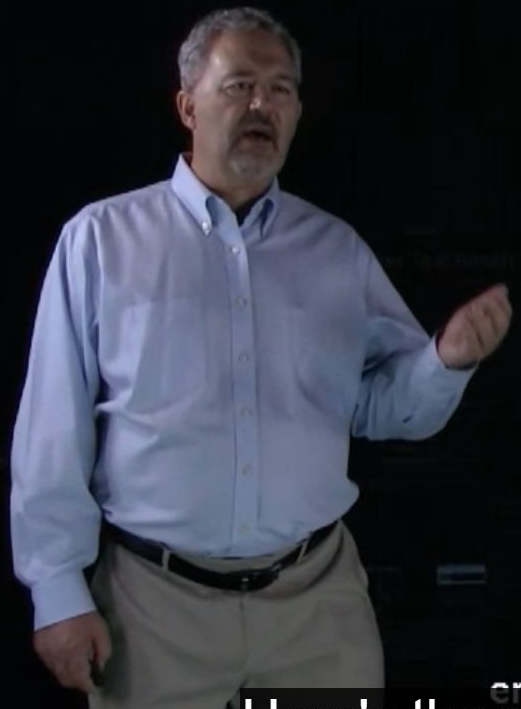
Counter Test Bench



```
// Test Process
  initial    // test generation process
  begin
    @(negedge reset) // wait for reset
inactive
    @(negedge clock) // wait for one clock
  // Test load
    load = 1'b1;
    en = 1'b0;
    data_tb = 4'b1010;
    @(negedge clock) // wait for one clock
    if (q != 4'b1010)
      $display("Load failure %b", q);
```

Here is the testbench code that generates the signals to test the load function of

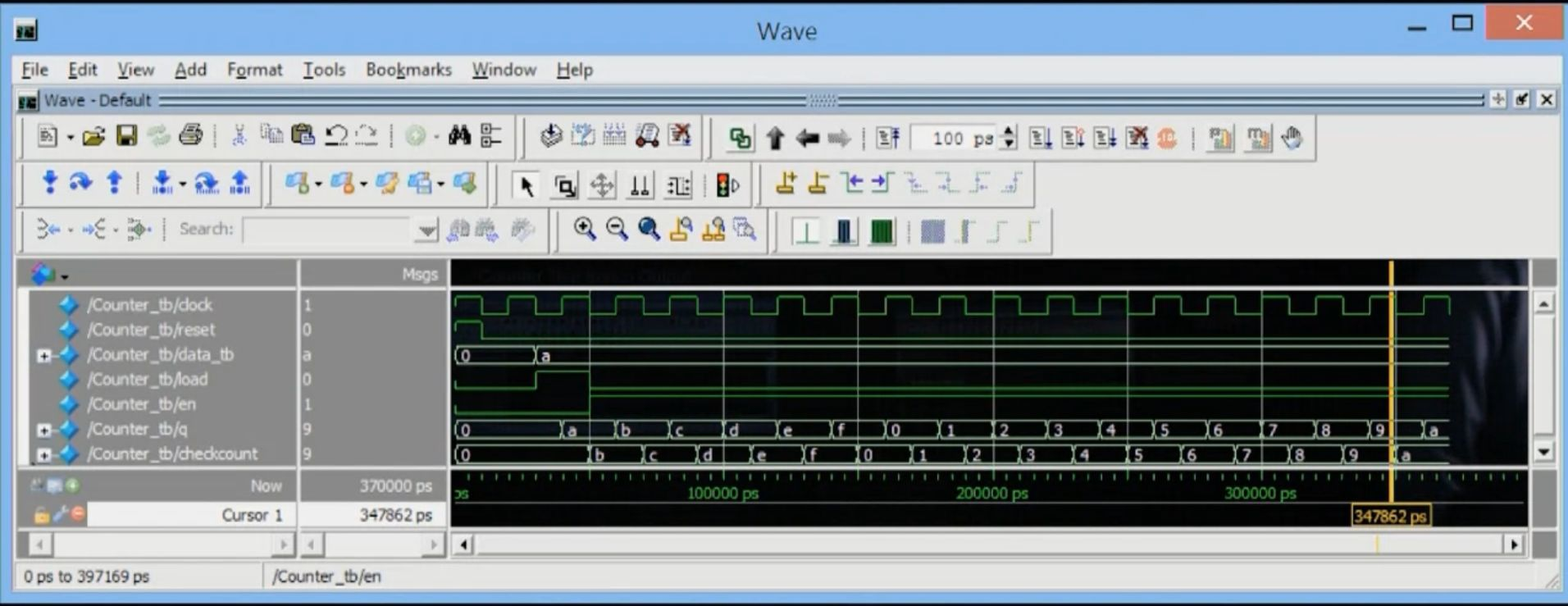
Counter Test Bench



```
// Test Process, continued
   // test count
    checkcount = 4'b1010;  // compare variable
    load = 1'b0;
    en = 1'b1;
    repeat (2**n)
    begin
      checkcount = checkcount + 1;   // count
      @(negedge clock) // wait for one clock
      if (q != checkcount)
        $display("Count failure at time %g/t at
                  count %b", $time, q);
    end
    $stop;   // end simulation
  end
endmodule
```

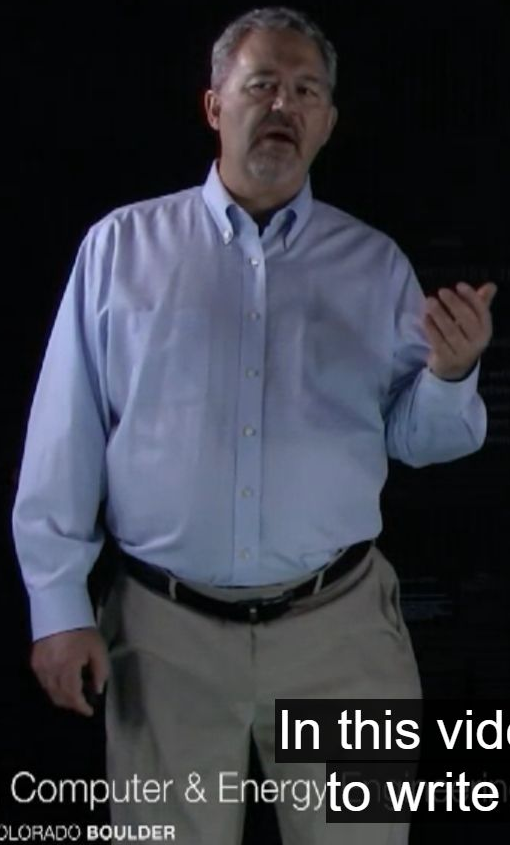
Here's the section of code that tests the counting function of the counter.

Counter Test Bench Output



In the counter testbench output waveform, we see the load test at the left and

Summary – Testbenches in Verilog II



In this video, you have learned:

- **How to write simple testbenches for synchronous circuits**
- **How to use external signal generators to create stimulus for synchronous circuits.**

In this video you have learned how to write simple testbenches for

References

[1] Pong P. Chu, "Regular Sequential Circuit" in *Embedded SOPC Design with NIOS II Processor and Verilog Examples*, Hoboken, NJ, Wiley, 2012, ch. 5, sec. 5.2, pp. 107-110

[2] D. Smith, "Test Harnesses" in *HDL Chip Design, A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog,* Madison, AL, Doone Publications, 1996, ch. 11, pp. 323-344.

Be Boulder.

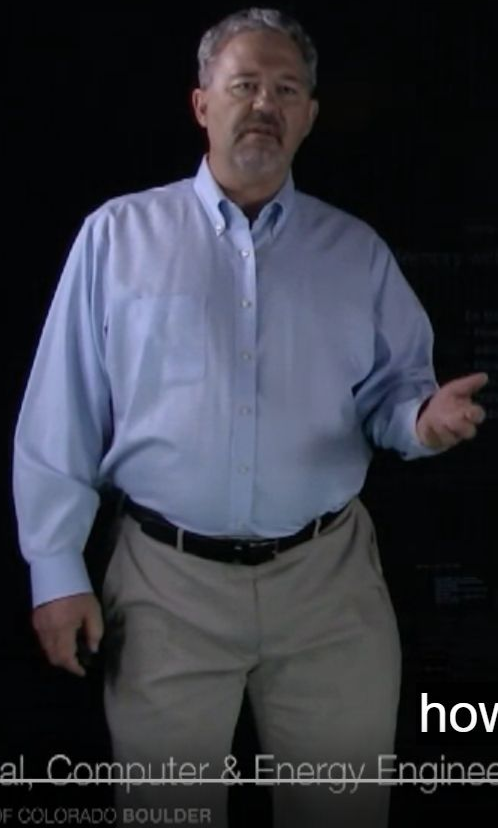
University of Colorado **Boulder**

Memory with Verilog

Professor Tim Scherr

University of Colorado **Boulder**

Memory with Verilog

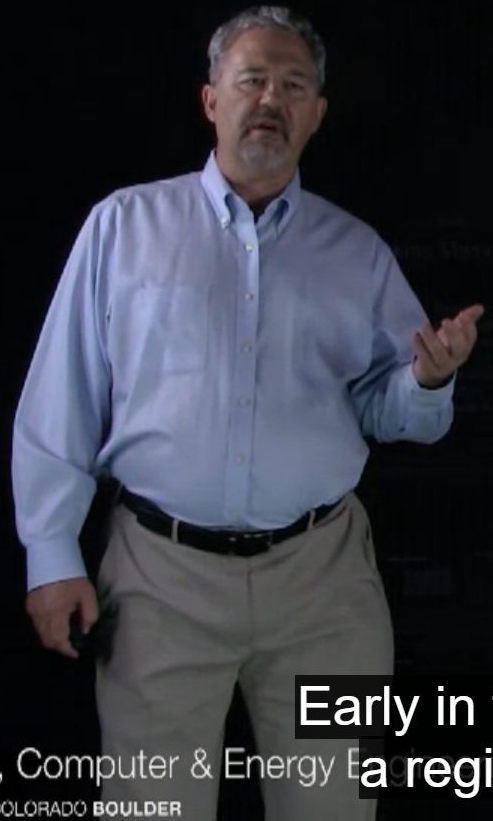


In this video, you will learn:

- **How to create memory devices using Verilog**
- **How to initialize a memory using file input**
- **How to create a simple look up table in Verilog**

how to initialize a memory
using the file input,

Making Memories

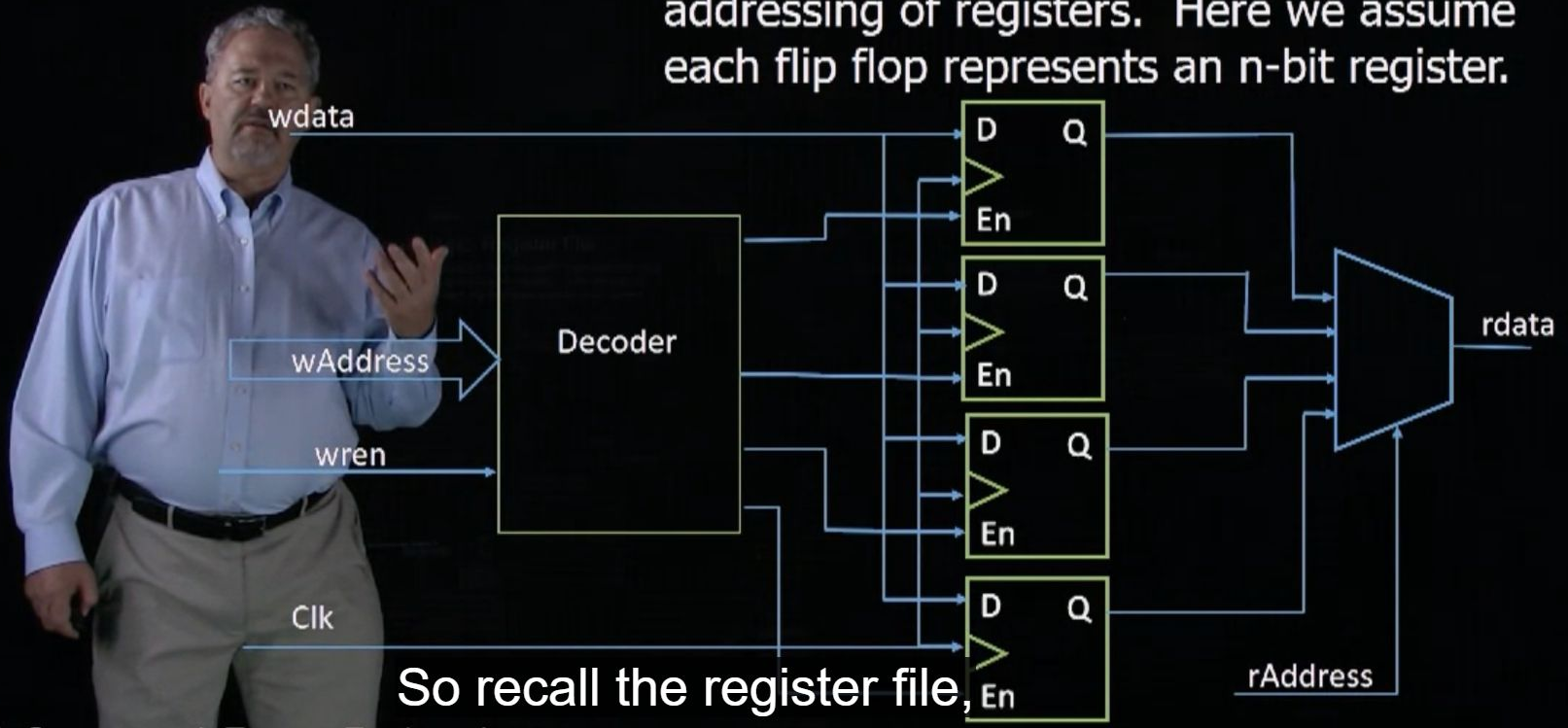


Memories are a common element in most digital systems. Earlier in this course we described a register file circuit, in which individual registers were enabled for access by decoding an address. We will now extend this to include RAM and ROM memories.

Early in this course, we described a register file circuit in which

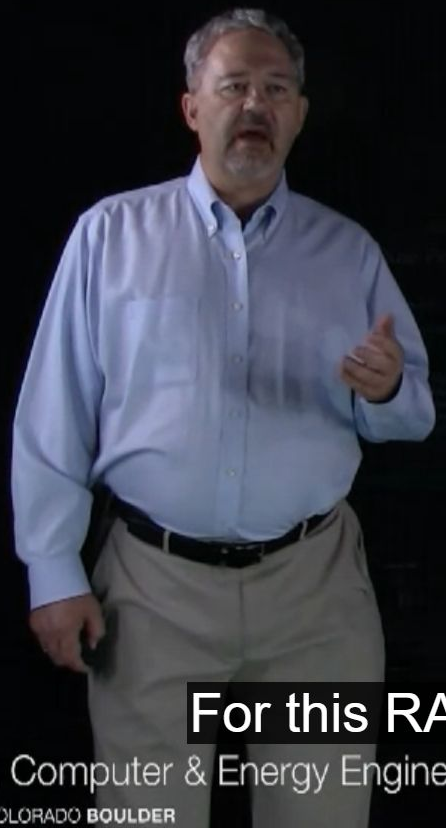
Synchronous Logic: Register File

Register Files are useful constructs that allow addressing of registers. Here we assume each flip flop represents an n-bit register.



So recall the register file,

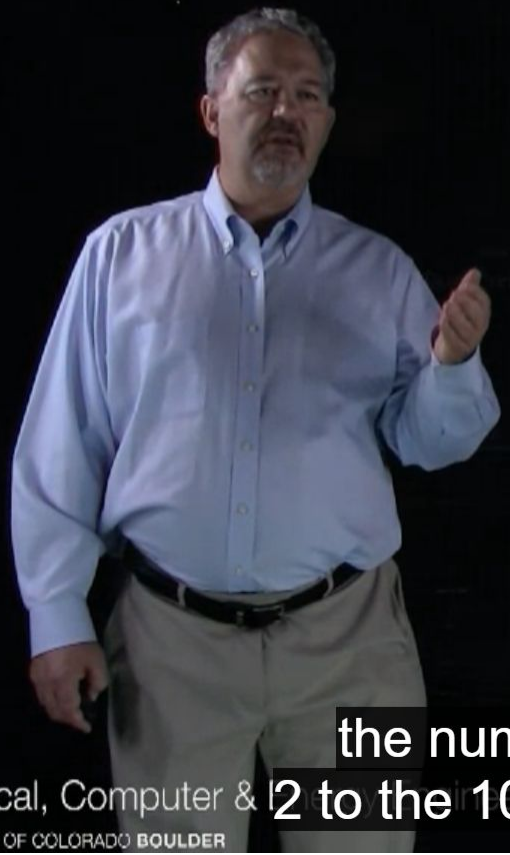
Dual Port RAM



```
module DPRAM
#(
  parameter Data_width = 8,  //# of bits in word
            Addr_width = 10  // # of address bits
)
(  //ports
  input wire clk,
  input wire we,
  input wire [(Addr_width-1):0] w_addr, r_addr,
  input wire [(Data_width-1):0] d,
  output wire [(Data_width-1):0] q
);
```

For this RAM model the data bus with and

Dual Port RAM

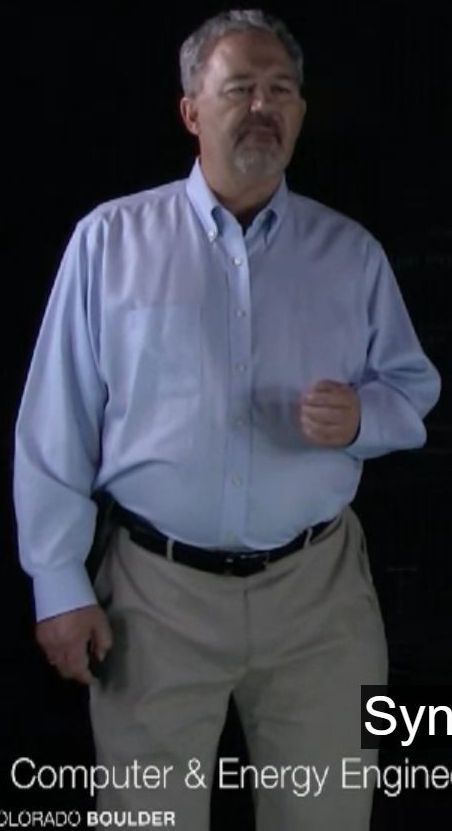


```
// signal declarations
  reg [Data_width-1:0] ram [2**Addr_width-1:0];
    // 2 dimensional array for RAM storage
  reg [Data_width-1:0] data_reg;
    // read output reg.

// RAM initialization from an external file
initial
  $readmemh("initalRAM.txt", ram);
```

the number of register locations is
2 to the 10th making a 1024 by 8 RAM.

Dual Port RAM



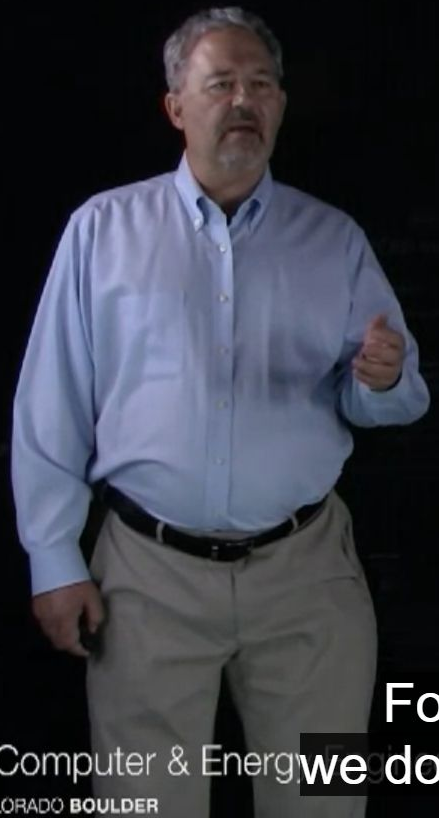
```
// body
//  write operation
  always @(posedge clk)
  begin
    if (we)
      ram[w_addr] <= d;     // write data
    data_reg <= ram[r_addr]; // read data to reg
  end

// read operation
  assign q = data_reg;

endmodule
```

Synchronous with the rising edge of the clock,

ROM memory

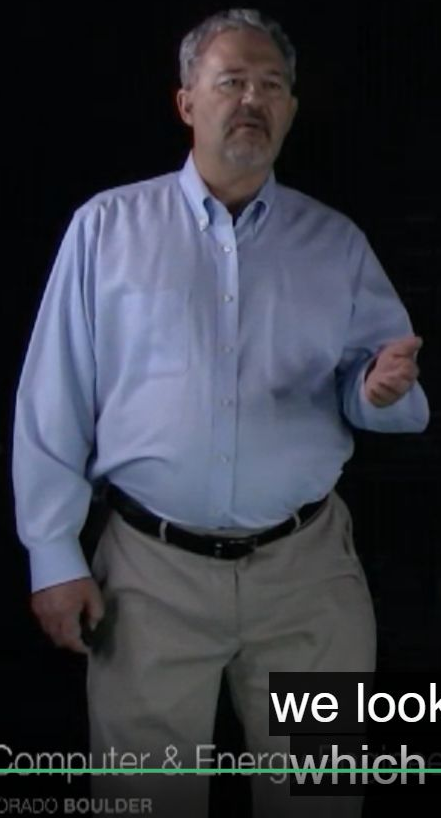


```
module ROM
#(
  parameter Data_width = 8,  //# of bits in word
            Addr_width = 3  // # of address bits
  )
  (  //ports
    input wire clk,
    input wire [Addr_width-1:0] addr,
    output wire [Data_width-1:0] data
  );

// signal declarations
reg [Data_width-1:0] rom_data, data_reg;
```

For a read-only memory,
we don't need a write enable or

ROM Memory



```
// body
 always @(posedge clk)  // output register
   data_reg <= rom_data;

 always @*
case(addr)  // lookup table
      3'b000:  rom_data = 8'b1000_0000;
      3'b001:  rom_data = 8'b1010_1010;
      3'b010:  rom_data = 8'b0101_0101;
      3'b011:  rom_data = 8'b1000_0011;
      3'b100:  rom_data = 8'b0000_0000;
      3'b101:  rom_data = 8'b1001_1001;
      3'b110:  rom_data = 8'b1000_0001;
      3'b111:  rom_data = 8'b1111_0000;
   endcase

 assign data = data_reg;
endmodule
```

we look up the value of the ROM,
which is listed in the case table.

Summary – Testbenches in Verilog

In this video, you have learned:

- **How to create memory devices using Verilog**
- **How to initialize a memory using file input**
- **How to create a simple look up table in Verilog**

References

[1] D. Smith, "Test Harnesses" in *HDL Chip Design, A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog,* Madison, AL, Doone Publications, 1996, ch. 11, pp. 323-344.

[2] Pong P. Chu, "Regular Sequential Circuit" in *Embedded SOPC Design with NIOS II Processor and Verilog Examples*, Hoboken, NJ, Wiley, 2012, ch. 5, sec. 5.2, pp. 100-102

[3] Pong P. Chu, "Regular Sequential Circuit" in *Embedded SOPC Design with NIOS II Processor and Verilog Examples*, Hoboken, NJ, Wiley, 2012, ch. 5, sec. 5.7, pp. 124-131

Be Boulder.

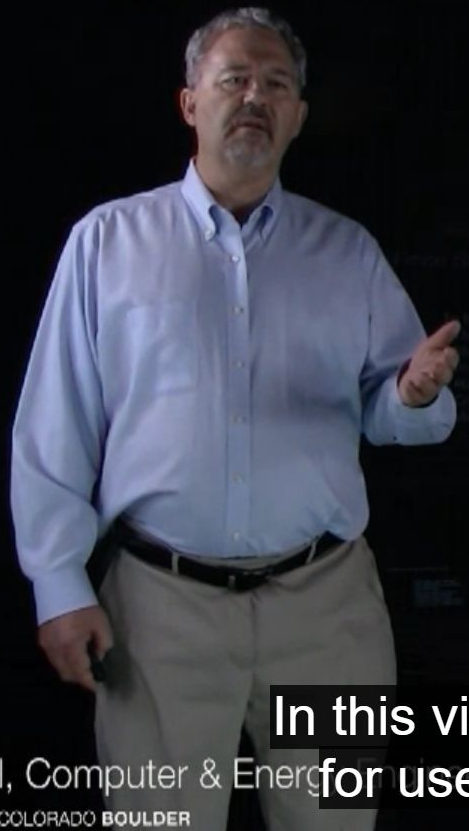
University of Colorado **Boulder**

Verilog Finite State Machines

Professor Tim Scherr

University of Colorado **Boulder**

Verilog Finite State Machines



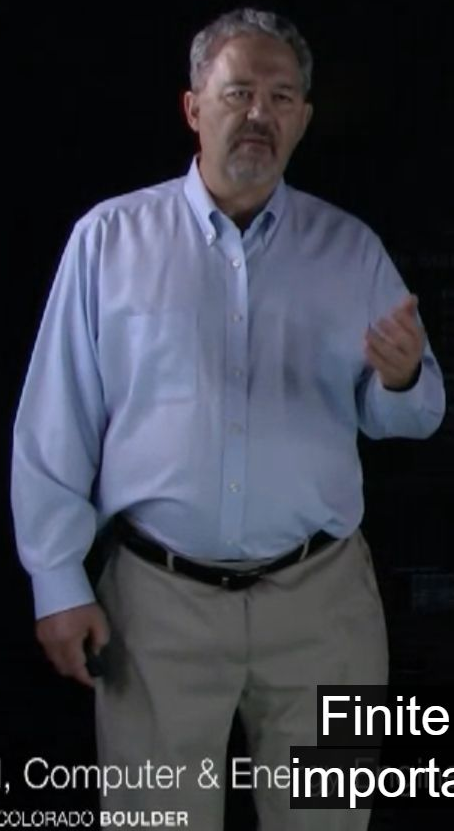
In this video, you will learn:

- **The rationale for use of Finite State Machines**
- **How to create Finite State Machines using Verilog**
- **Criteria for determining which state encoding formats to use**

In this video you'll learn the rationale for use of Finite State Machines,

Electrical, Computer & Energy
UNIVERSITY OF COLORADO BOULDER

Finite State Machines

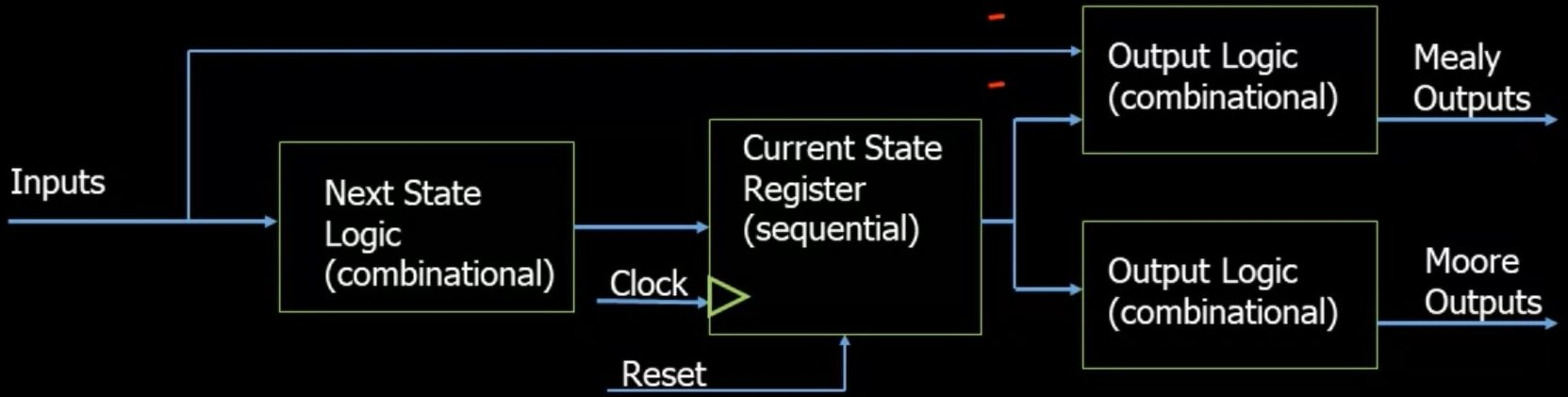


Finite State Machines (FSM) are a very important part of digital design (and software design, too!).
The state machine concept provides a highly **reliable**, **maintainable**, and **methodical** way to **design circuits** that **perform a sequence of operations** with great **predictability**.
State machines are always in a known state. Good Designers all use FSMs.

Finite State machines are a very important part of digital design and

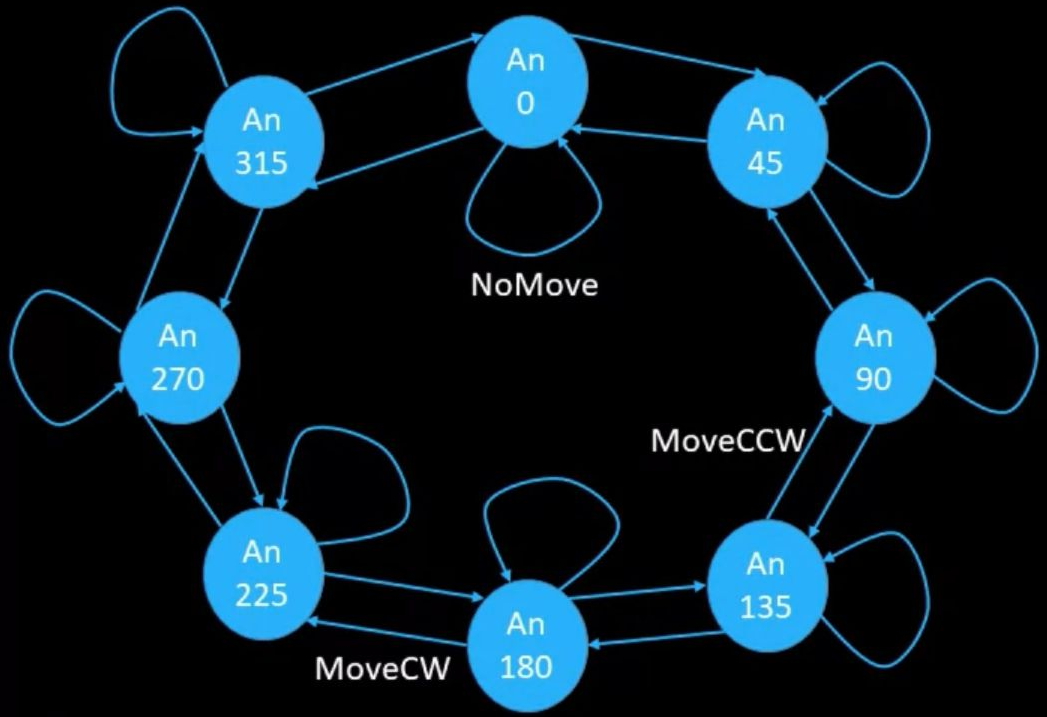
General FSM Block Diagram

FSMs are broadly categorized into 2 types: Moore or Mealy.
In Moore machines the output only depends on the state.
In Mealy machines the inputs and the state drive the output.



Immediately machines the inputs and the state drive the output,

Example of FSM State Diagram and State Table



Current State	Next State	Next State	Next State	Output	Output
	Input = Move CW	Input = Move CCW	Input = NoMove	Desired Position	PosError
An0	An45	An315	An0	Current State	Dpos - PhyPos
An45	An90	An0	An45	..	..
An90	An135	An45	An90	..	..
An135	An180	An90	An135	..	..
An180	An225	An135	An180	..	..
An225	An270	An180	An225	..	..
An270	An315	An225	An270	..	..
An315	An0	An270	An315	..	..

[1]

State Encoding

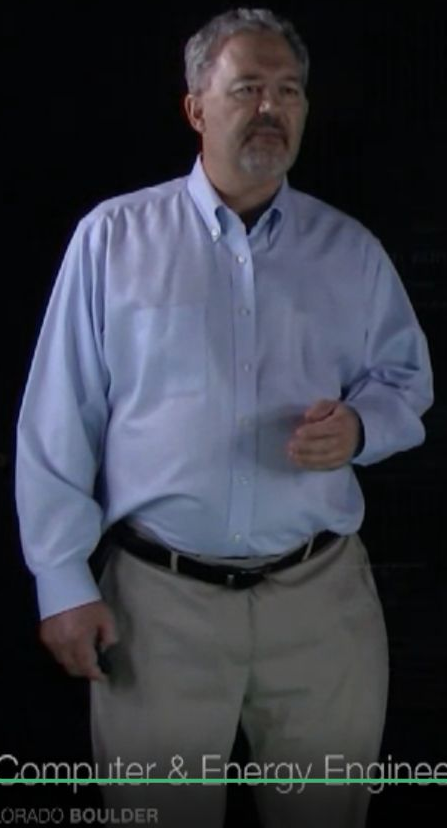
No.	Binary	Gray	Johnson	One-Hot
0	000	000	0000	00000001
1	001	001	0001	00000010
2	010	011	0011	00000100
3	011	010	0111	00001000
4	100	110	1111	00010000
5	101	111	1110	00100000
6	110	101	1100	01000000
7	111	100	1000	10000000

The current state usually represented by a collection of flip-flops.

Share

Press Esc to exit full screen

Implementation with Binary Encoding



```
module AngleFSM
#(  // Binary encoding of states
  parameter State_width = 3,
            An0 = 3'b000,
            An45 = 3'b001,
            An90 = 3'b010,
            An135 = 3'b011,
            An180 = 3'b100,
            An225 = 3'b101,
            An270 = 3'b110,
            An315 = 3'b111)
  (  //ports
    input wire clk, reset, MoveCW, MoveCCW,
    input wire [(State_width-1):0] PhysicalPosition,
    output wire [(State_width-1):0] DesiredPosition,
PosError
  );
  reg [(State_width-1):0] CurrentState, NextState;
```

Electrical, Computer & Energy Engineering

UNIVERSITY OF COLORADO BOULDER

[1]

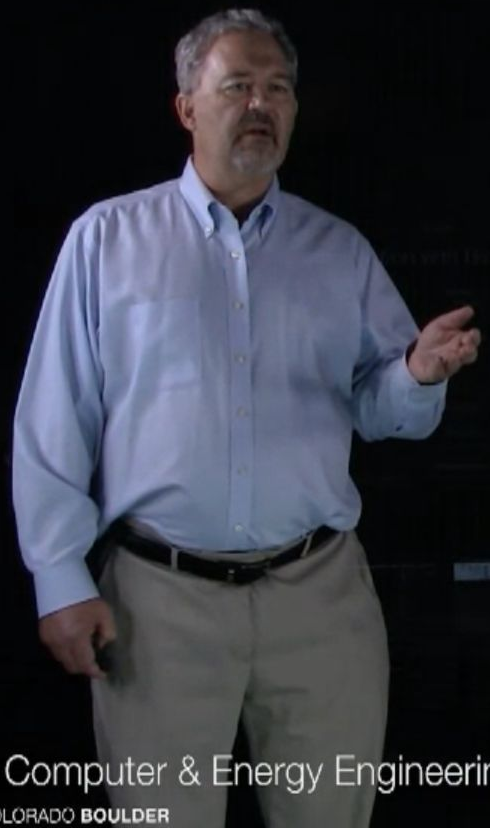
3:38 / 7:04

Implementation with Binary Encoding

```
// body of FSM is case statement
// Next State Logic
  always @(MoveCW or MoveCCW or
PhysicalPosition or CurrentState)
    begin: Combinational
      case (CurrentState)
        An0:
          if (MoveCW ==1)
            NextState = An45;
          else if (MoveCCW == 1)
            NextState = An315;
          else
            NextState = An0;
        An45:
          if (MoveCW ==1)
            NextState = An90;
          else if (MoveCCW == 1)
            NextState = An0;
          else
            NextState = An45;
...  //states An90 to An270 here
        An315:
          if (MoveCW ==1)
            NextState = An0;
          else if (MoveCCW == 1)
            NextState = An270;
          else
            NextState = An315;
        default:
          NextState = PhysicalPosition;
      endcase
    end
```

[1]

Implementation with Binary Encoding

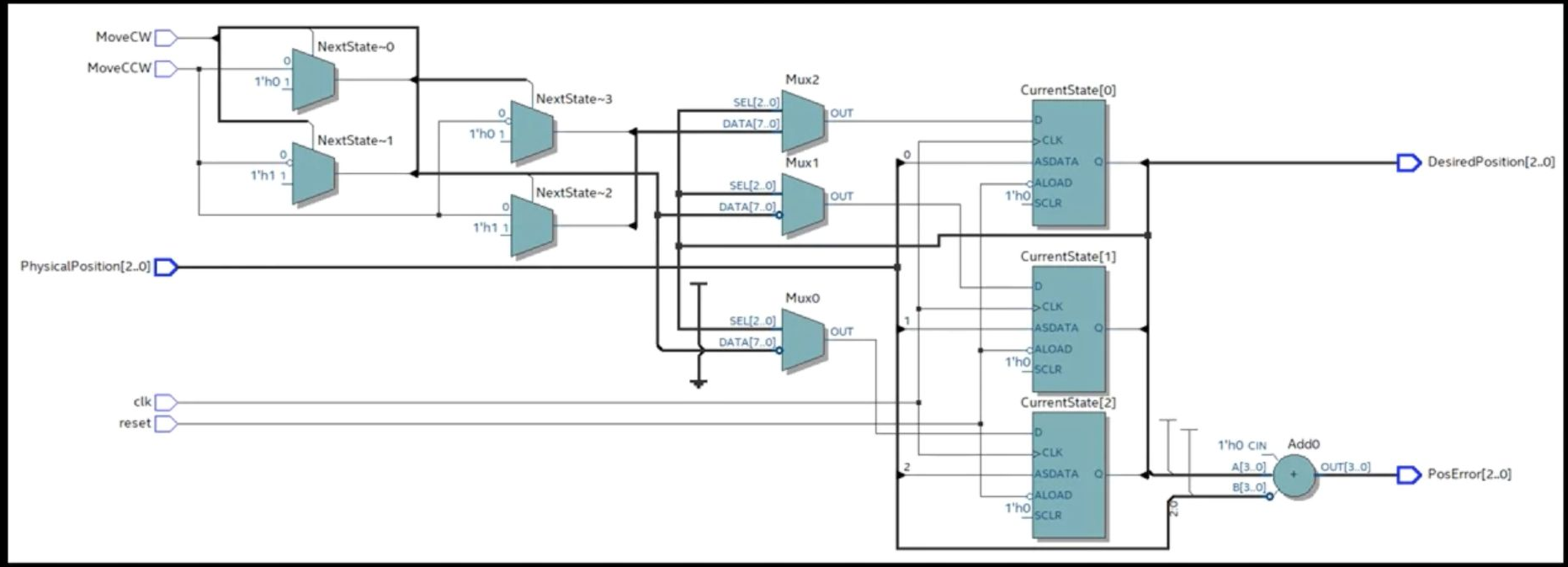


```
// Current State Register
  always @(posedge clk or negedge reset)
    begin: Sequential
      if (!reset)
        CurrentState = PhysicalPosition;
      else
        CurrentState = NextState;
    end

// Output Logic
// Moore Outputs
  assign DesiredPosition = CurrentState;
// Mealy Outputs
  assign PosError = DesiredPosition -
PhysicalPosition;

endmodule
```

Implementation with Binary Encoding

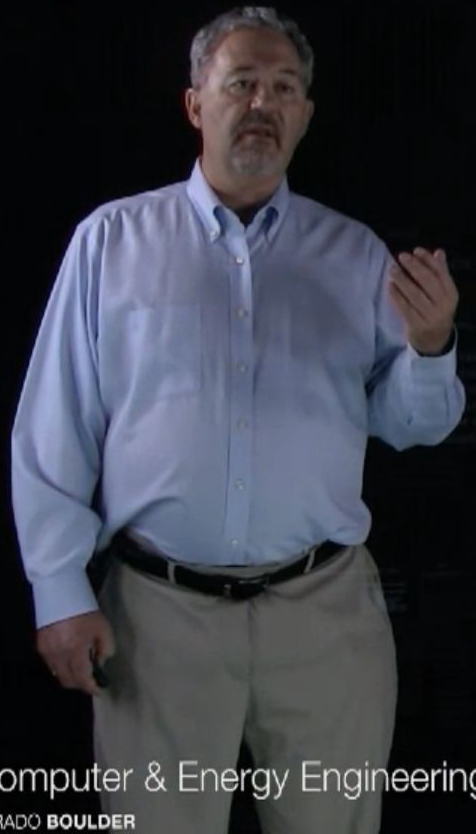


Schematic

State Encoding Comparison

Encoding	Registers (FFs)	Total Logic Cells FSM	Output Encoding Conversion Logic	Total Logic Cells
Binary	3	17	0	17
Gray	3	20	9	29
Johnson	4	36	9	51
One-Hot	8	107	33	146

Summary – State Machines in Verilog



In this video, you have learned:

- **The rationale for use of Finite State Machines**
- **How to create Finite State Machines using Verilog**
- **Criteria for determining which state encoding formats to use**

References

[1] D. Smith, "Modeling Finite State Machines" in *HDL Chip Design, A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog,* Madison, AL, Doone Publications, 1996, ch. 8, pp. 195-201.

Be Boulder.

University of Colorado **Boulder**