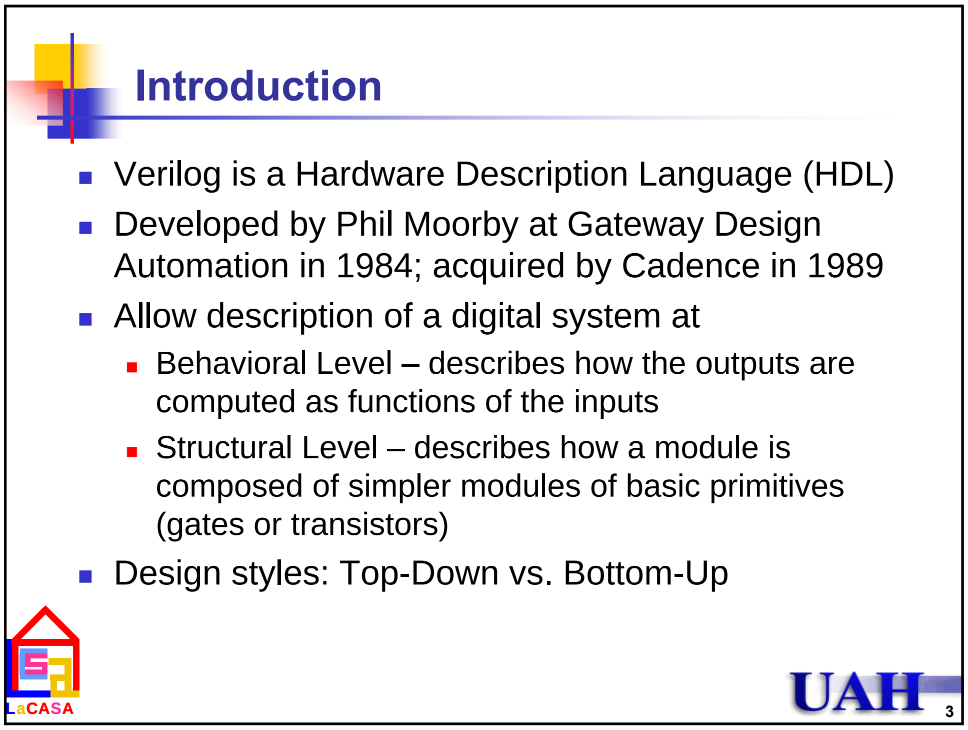
Verilog Tutorial

Aleksandar Milenković

The LaCASA Laboratory
Electrical and Computer Engineering Department
The University of Alabama in Huntsville
Email: milenka@ece.uah.edu
Web: http://www.ece.uah.edu/~milenka
http://www.ece.uah.edu/~lacasa

Outline

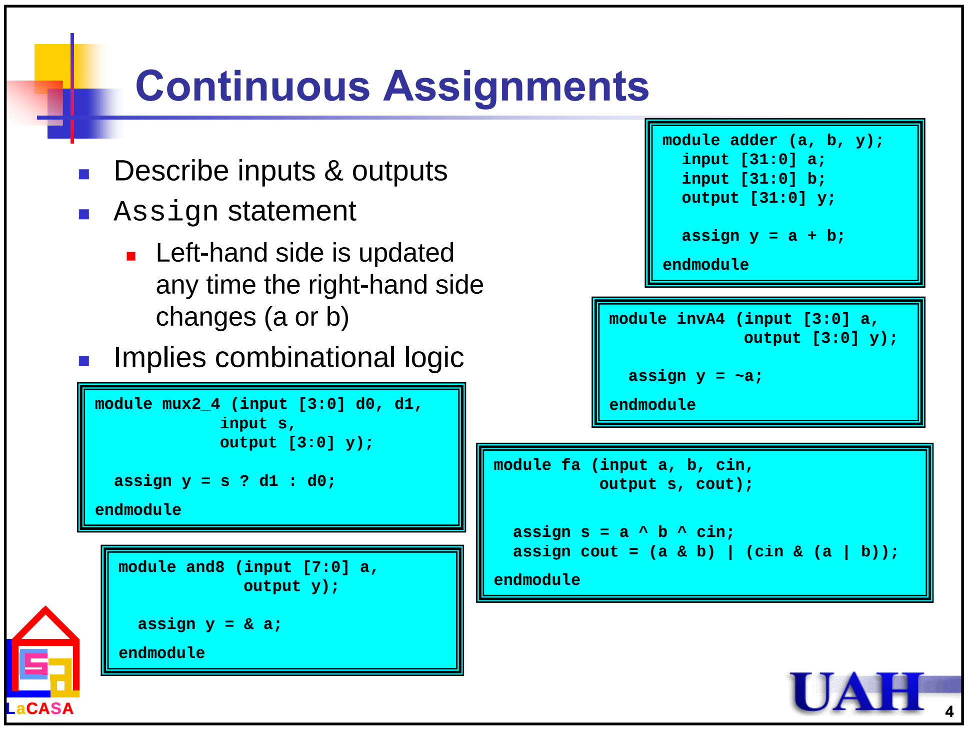
- Introduction
- Verilog (with focus on synthesis)
  - Continuous Assignments (`assign`)
  - Hierarchy
  - `Always` Blocks



Introduction

- Verilog is a Hardware Description Language (HDL)
- Developed by Phil Moorby at Gateway Design Automation in 1984; acquired by Cadence in 1989
- Allow description of a digital system at
  - Behavioral Level – describes how the outputs are computed as functions of the inputs
  - Structural Level – describes how a module is composed of simpler modules of basic primitives (gates or transistors)
- Design styles: Top-Down vs. Bottom-Up

LaCASA UAH 3



Continuous Assignments

- Describe inputs & outputs
- `Assign` statement
  - Left-hand side is updated any time the right-hand side changes (a or b)
- Implies combinational logic

```
module adder (a, b, y);
  input [31:0] a;
  input [31:0] b;
  output [31:0] y;

  assign y = a + b;
endmodule
```

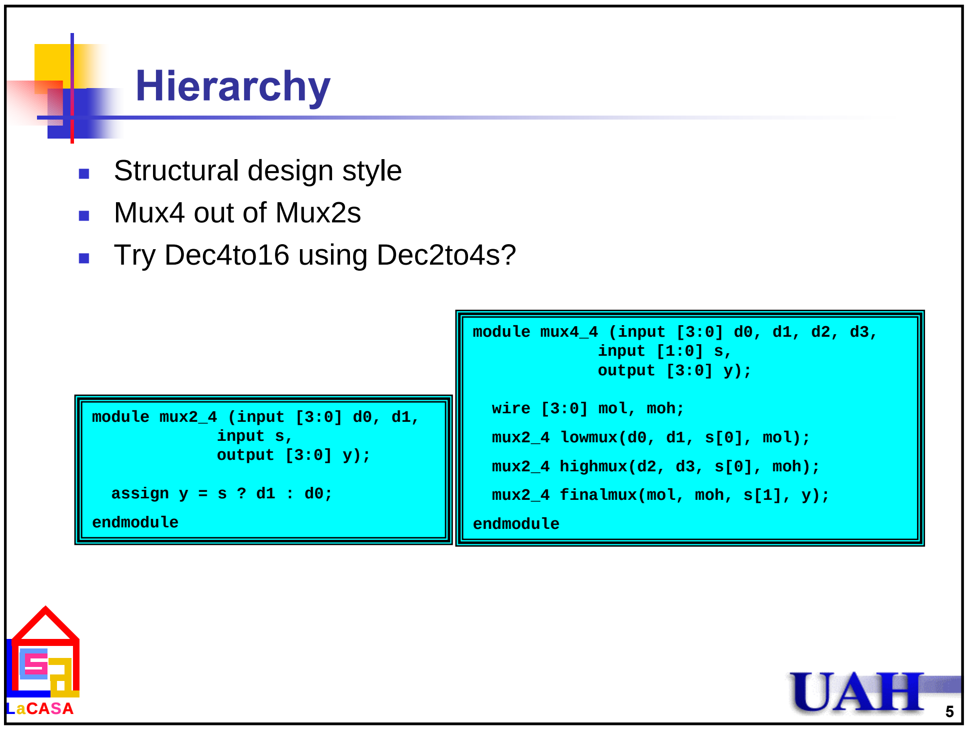
```
module invA4 (input [3:0] a,
              output [3:0] y);

  assign y = ~a;
endmodule
```

```
module mux2_4 (input [3:0] d0, d1,
               input s,
               output [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

```
module fa (input a, b, cin,
           output s, cout);

  assign s = a ^ b ^ cin;
  assign cout = (a & b) | (cin & (a | b));
endmodule
```

```
module and8 (input [7:0] a,
             output y);

  assign y = & a;
endmodule
```

LaCASA UAH 4



Hierarchy

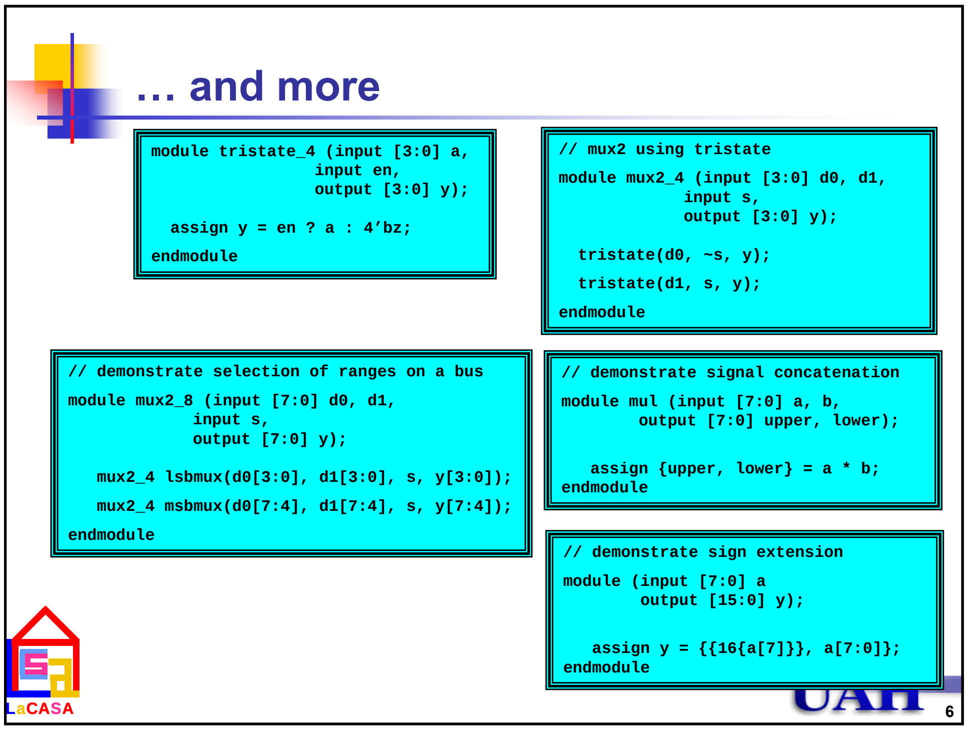
- Structural design style
- Mux4 out of Mux2s
- Try Dec4to16 using Dec2to4s?

```
module mux2_4 (input [3:0] d0, d1,
               input s,
               output [3:0] y);

  assign y = s ? d1 : d0;

endmodule
```

```
module mux4_4 (input [3:0] d0, d1, d2, d3,
               input [1:0] s,
               output [3:0] y);

  wire [3:0] mol, moh;

  mux2_4 lowmux(d0, d1, s[0], mol);

  mux2_4 highmux(d2, d3, s[0], moh);

  mux2_4 finalmux(mol, moh, s[1], y);

endmodule
```

LaCASA

UAH



… and more

```
module tristate_4 (input [3:0] a,
                   input en,
                   output [3:0] y);

  assign y = en ? a : 4'bz;

endmodule
```

```
// mux2 using tristate

module mux2_4 (input [3:0] d0, d1,
               input s,
               output [3:0] y);

  tristate(d0, ~s, y);

  tristate(d1, s, y);

endmodule
```

```
// demonstrate selection of ranges on a bus

module mux2_8 (input [7:0] d0, d1,
               input s,
               output [7:0] y);

   mux2_4 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);

   mux2_4 msbmux(d0[7:4], d1[7:4], s, y[7:4]);

endmodule
```

```
// demonstrate signal concatenation

module mul (input [7:0] a, b,
        output [7:0] upper, lower);

   assign {upper, lower} = a * b;
endmodule
```

```
// demonstrate sign extension

module (input [7:0] a
        output [15:0] y);


   assign y = {{16{a[7]}}, a[7:0]};
endmodule
```

LaCASA

UAH

Always Blocks

- **`Always`** blocks are reevaluated only when signals in the header (called a *sensitivity list*) change
- Depending on the form, can imply either sequential or combinational circuits

LaCASA

UAH

Combinational Logic

```
module  mux2( input d0, d1, s,
              output y);
   reg y;

   always @(s or d0 or d1)
     begin : MUX
      case(s)
        1'b0 : y = d0;
        1'b1 : y = d1;
      endcase
     end
endmodule
```

```
module  mux2( input d0, d1, s,
              output y);
   reg y;

   always @(*)
     begin : MUX
      case(s)
        1'b0 : y = d0;
        1'b1 : y = d1;
      endcase
     end
endmodule
```

Warning: All the signals on the left side of assignments in `always` blocks must be declared as `reg`. However, declaring a signal as `reg` does not mean the signal is actually a register.

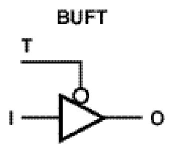
LaCASA

UAH

Tri-State

```
module three_st (T, I, O);
  input T, I;
  output O;
  reg O;

  always @(T or I)
    begin
      if (~T)
        O = I;
    else
      O = 1'bZ;
    end
endmodule
```

```
module three_st (T, I, O);
  input T, I;
  output O;

  assign O = (~T) ? I: 1'bZ;
endmodule
```



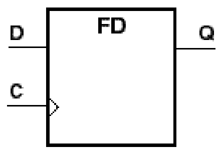
Registers

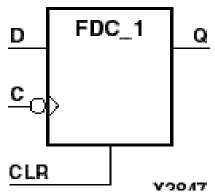
D-FF with Positive Clock

```
module flop (C, D, Q);
  input C, D;
  output Q;
  reg Q;

  always @(posedge C)
    begin
      Q = D;
    end
endmodule
```

D-FF with Negative-edge Clock and Async. Clear

```
module flop (C, D, CLR, Q);
  input C, D, CLR;
  output Q;
  reg Q;

  always @(negedge C or posedge CLR)
    begin
      if (CLR)
        Q = 1'b0;
      else
        Q = D;
    end
endmodule
```

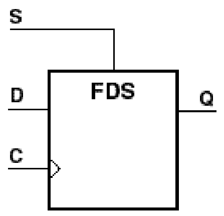




DFFs

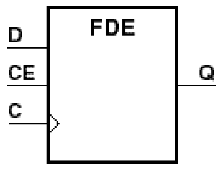
D-FF with Positive-Edge Clock and Synchronous Set

```
module flop (C, D, S, Q);
  input C, D, S;
  output Q;
  reg Q;

  always @(posedge C)
    begin
      if (S)
        Q = 1'b1;
      else
        Q = D;
    end
endmodule
```



D-FF with Positive-Edge Clock and Clock Enable

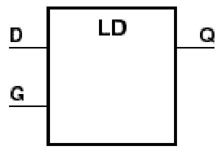
```
module flop (C, D, CE, Q);
  input C, D, CE;
  output Q;
  reg Q;


  always @(posedge C)
    begin
      if (CE)
        Q = D;
    end
endmodule
```



Latches

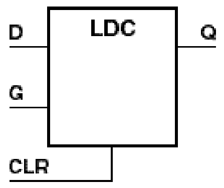
Latch with Positive Gate

```
module latch (G, D, Q);
  input G, D;
  output Q;
  reg Q;


always @(G or D)
    begin
      if (G)
        Q = D;
  end
endmodule
```



Latch with Positive Gate and Asynchronous Clear

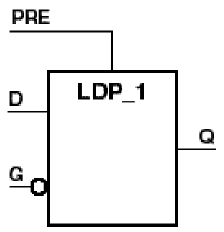
```
module latch (G, D, CLR, Q);
  input G, D, CLR;
  output Q;
  reg Q;



  always @(G or D or CLR)
    begin
      if (CLR)
        Q = 1'b0;
      else if (G)
        Q = D;
    end
endmodule
```



4-bit Latch

4-bit Latch with Positive Gate Inverted Gate and Asynchronous Preset

```
module latch (G, D, PRE, Q);
  input G, PRE;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] Q;



  always @(G or D or PRE)
    begin
      if (PRE)
        Q = 4'b1111;
    else if (~G)
      Q = D;
    end
endmodule
```



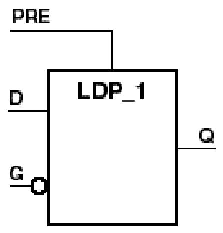
LaCASA

UAH

4-bit Register

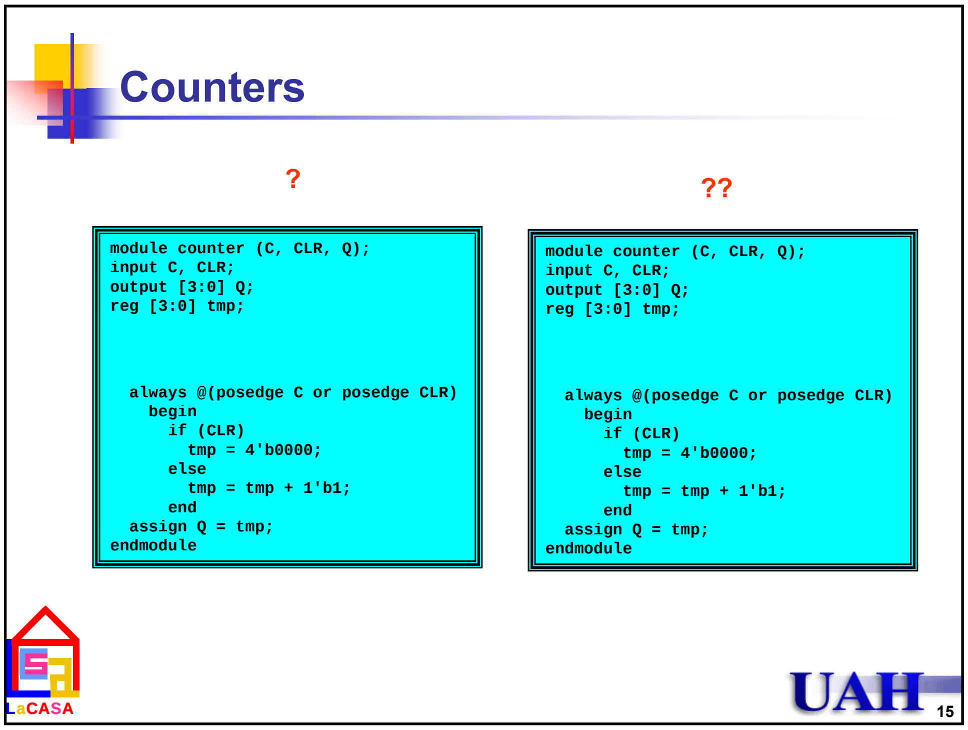
4-bit Register Positive-Edge Clock, Asynchronous Set and Clock Enable

```
module flop (C, D, CE, PRE, Q);
  input C, CE, PRE;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] Q;



  always @(posedge C or posedge PRE)
    begin
      if (PRE)
        Q = 4'b1111;
      else
        if (CE)
          Q = D;
    end
endmodule
```



LaCASA

UAH



Counters

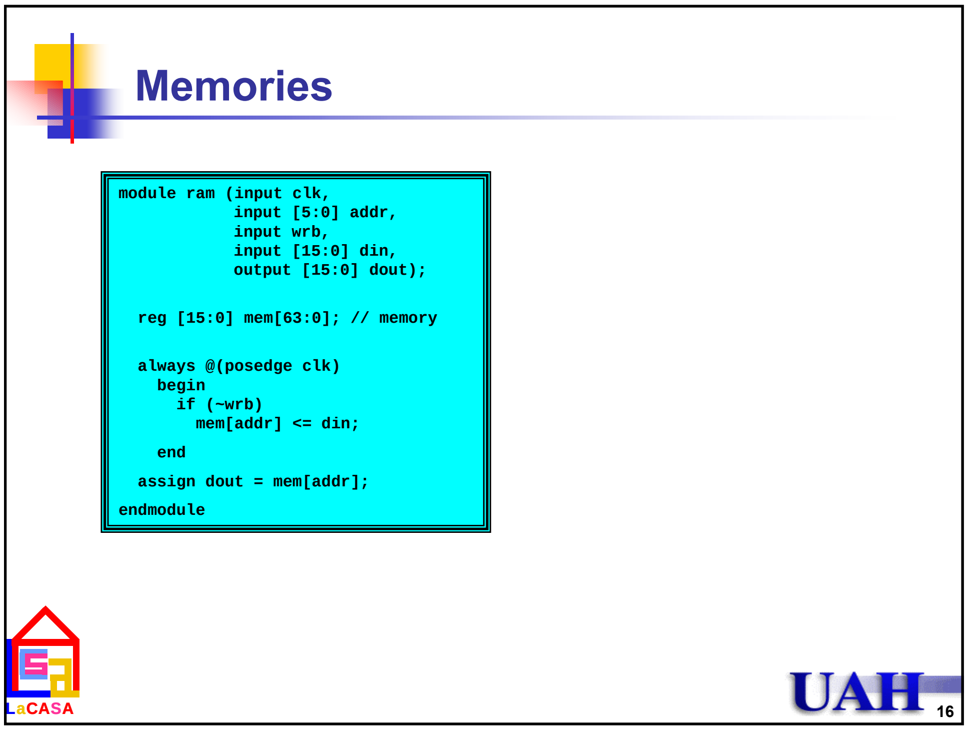
?

```
module counter (C, CLR, Q);
input C, CLR;
output [3:0] Q;
reg [3:0] tmp;



  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
      else
        tmp = tmp + 1'b1;
      end
  assign Q = tmp;
endmodule
```

??

```
module counter (C, CLR, Q);
input C, CLR;
output [3:0] Q;
reg [3:0] tmp;



  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
      else
        tmp = tmp + 1'b1;
      end
  assign Q = tmp;
endmodule
```

LaCASA UAH 15



Memories

```
module ram (input clk,
            input [5:0] addr,
            input wrb,
            input [15:0] din,
            output [15:0] dout);


  reg [15:0] mem[63:0]; // memory


  always @(posedge clk)
    begin
      if (~wrb)
        mem[addr] <= din;

    end

  assign dout = mem[addr];

endmodule
```

LaCASA UAH 16

Blocking and Non-blocking Assignments

- Blocking assignments (use =)
  - A group of blocking assignments inside a begin-end block is evaluated sequentially
- Non-blocking assignments (use <=)
  - A group of non-blocking assignments are evaluated in parallel; all of the statements are evaluated before any of the left sides are updated.

LaCASA UAH 17

Shift Register

```
module shiftreg (input clk,
                 input sin,
                 output reg [3:0] q);

  always @(posedge clk)
    begin
       q[0] <= sin;

       q[1] <= q[0];

       q[2] <= q[1];

       q[3] <= q[2];

       // could be replaced by

       // q <= {q[2:0], sin}

    end

endmodule
```

```
// this is incorrect shift register

module shiftreg (input clk,
                 input sin,
                 output reg [3:0] q);

  always @(posedge clk)
    begin
       q[0] = sin;

       q[1] = q[0];

       q[2] = q[1];

       q[3] = q[2];

    end

endmodule
```

LaCASA UAH 18

State Machine

- Identify 3 portions
  - *State register*
  - *Next state logic*
  - *Output logic*
- State register
  - Resets asynchronously to the initial state
  - Otherwise advances to the next state
- Next logic
  - Computes the next state as a function of the current state and inputs
- Output logic
  - Computes the output as a function of the current state and the inputs

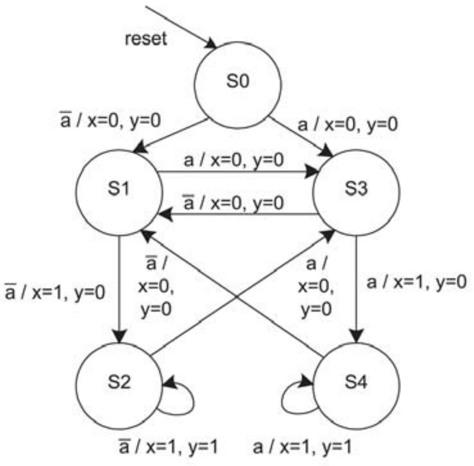


FIG A.5 History FSM state transition diagram

LaCASA

UAH

State Machine Description

```
module hFSM (input clk, reset, a,
             output x, y);

  reg[2:0] state, nextstate;

  parameter S0 = 3’b000;
  parameter S1 = 3’b010;
  parameter S2 = 3’b011;
  parameter S3 = 3’b100;
  parameter S4 = 3’b101;


  always @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;

  always @(*)
    case (state)
      S0: if (a) nextstate <= S3;
          else nextstate <= S1;
      S1: if (a) nextstate <= S3;
          else nextstate <= S2;
      S2: if (a) nextstate <= S3;
          else nextstate <= S2;
      S3: if (a) nextstate <= S4;
          else nextstate <= S1;
      S4: if (a) nextstate <= S4;
          else nextstate <= S1;
      default: nextstate <= S0;

   endcase
```

```
  //output logic
  assign x = (state[1] & ~a) |
             (state[2] & a);

  assign y = (state[1] & state[0] & ~a)
          | (state[2] & state[0] & a);


endmodule
```

LaCASA

UAH