Parallel versus serial execution

- **assign** statements are implicitly parallel
  - “=” means continuous assignment
  - Example
    ```verilog
generate
assign E = A & D;
assign A = B & C;
```
  - A and E change if B changes

- **always** blocks execute in parallel
  - **always @(posedge clock)**

- Procedural block internals not necessarily parallel
  - “=” is a blocking assignment (sequential)
  - “<=” is a nonblocking assignment (parallel)
  - Examples of procedures: **always**, **function**, etc.
@ inside always blocks

🌟 Nested @ can halt execution
   ➤ Says “wait for a trigger”
   ➤ Can use for simple state machines
      ➣ When next state doesn’t depend on input

```verilog
always @(posedge clk) begin
    state = 4’b0001;
    @(posedge clk)
    state = 4’b0010;
    @(posedge clk)
    state = 4’b0100;
    @(posedge clk)
    state = 4’b1000;
end
```
Assignments

Be careful with $\textit{always}$ assignments

always @(c or x) begin
  if (c) begin
    value = x;
    end
  end
  y = value;
end

always @(c or x) begin
  value = x;
  if (c) begin
    value = 0;
    end
  end
  y = value;
end

always @(c or x) begin
  if (c)
    value = 0;
  else if (x)
    value = 1;
  end
end

always @(a or b)
  f = a & b & c;
end

Which statements generate a latch?
Assignments

+ Blocking versus non-blocking

```verilog
reg B, C, D;

always @(posedge clk)
begin
    B = A;
    C = B;
    D = C;
end
```

```verilog
reg B, C, D;

always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```
Finite state machines

 FSM model

 ![Diagram of FSM model]

 Implementing FSMs

 - Combinational logic: Use function or always
 - Registers: always triggered off posedge clock
Example: Moore machine

✧ Reduce 1s
  ➤ Change the first 1 to 0

module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  parameter zero = 0, one1 = 1, two1s = 2; // states
  reg out;
  reg [1:0] state; // state register
  reg [1:0] next_state; // always block needs reg

  // Implement the state register
  always @(posedge clk)
    if (reset) state = zero; // can put in next state logic
    else state = next_state;

Continued next page
Moore reduce 1s (cont’d)

```verilog
always @(in or state) // combinational logic
  case (state)
    zero: begin // last input was a zero
      out = 0;
      if (in) next_state = one1;
      else    next_state = zero;
    end
    one1: begin // we’ve seen one 1
      out = 0;
      if (in) next_state = twols;
      else    next_state = zero;
    end
    twols: begin // we’ve seen at least 2 ones
      out = 1;
      if (in) next_state = twols;
      else    next_state = zero;
    end
    default: begin // in case we reach a bad state
      next_state = zero;
      out = 0;
    endcase
  endcase
endmodule

must include all signals that are input to state and output equations
```

CSE467 Verilog FSMs
Another way

module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  parameter zero = 0, one1 = 1, twols = 2; // states
  reg [1:0] state; // state register
  assign out = reduce_output(state);
  always @(posedge clk)
    if (reset) state = zero;
    else state = next_state(in, state);

  function reduce_output;
    input [1:0] state;
    case (state)
      zero: reduce_output = 0;
      one1: reduce_output = 0;
      twols: reduce_output = 1;
      default: reduce_output = 0;
    endcase
  endfunction

Continued next page
Another way (con’t)

```verilog
function [1:0] next_state;
    input in;
    input [1:0] state;
    case (state)
        zero:
            if (in) next_state = one1;
            else next_state = zero;
        one1:
            if (in) next_state = twols;
            else next_state = zero;
        twols:
            if (in) next_state = twols;
            else next_state = zero;
        default:
            next_state = zero;
    endcase
endfunction
endmodule
```
Example: Mealy machine reduce 1s

module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;
    parameter zero = 0, one = 1; // states
    reg state; // state register
    assign out = reduce_output(in, state);

    always @(posedge clk)
        if (reset) state = zero;
        else state = next_state(in, state);

    function reduce_output;
        input in, state;
        case (state)
            zero: reduce_output = 0;
            one: if (~in) reduce_output = 0;
                else reduce_output = 1;
        endcase
    endfunction

Continued next page
function next_state;
    input in, state;
    case (state)
        zero:
        if (~in) next_state = zero;
        else    next_state = one;
        one:
        if (~in) next_state = zero;
        else    next_state = one;
    endcase
endfunction
Mealy machine reduce 1s again...

Notice the next state is the input...

```verilog
module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;
    reg state; // state register
    assign out = state & in;

    always @(posedge clk)
        if (reset) state = 0;
        else    state = in;
endmodule
```

![Diagram of Mealy machine reduce 1s again...](https://via.placeholder.com/150)
If you use *always* blocks...

- Use at least 2 of them
  - Avoid embedding output logic into state block
    - Use an output *always*
    - Use a state *always*

- Not an issue with Mealy machines
  - Must use 2 *always* with Mealy
    - Because output can change at any time
1-always Moore FSM (*not recommended!*)

- Reduce 1s again

```verilog
module reduce (clk, reset, in, out);
    input clk, reset, in;
    output out;
    reg out;
    reg [1:0] state; // state register
    parameter zero = 0, one1 = 1, two1s = 2;
```

*Continued next page*
always @(posedge clk)
case (state)
  zero: begin
    out = 0;
    if (in) state = one1;
    else state = zero;
  end
  one1: begin
    if (in)
      state = twols;
    out = 1;
  end else begin
    state = zero;
    out = 0;
  end
  twols: begin
    if (in)
      state = twols;
    out = 1;
  end else begin
    state = zero;
    out = 0;
  end
  default: begin
    state = zero;
    out = 0;
  end
endcase
endmodule

Confusing: the output logic is buried in the state logic

All outputs are registered

...and compiler might give you an extra register...
Synchronous Mealy FSMs

- Register the output

```verilog
always @(posedge clk) begin
    state <= f(state, inputs);
    Mealy_out <= f(state, inputs);
end
```
Parameters customize module instantiation

- Can “tweak” modules for particular applications
  - Example: Variable bit widths

module comparator (a, b, c);
    parameter width = 8;
    input [width-1:0] a, b;
    output c;
    assign c = (a == b);
endmodule

... comparator #(32) c32(w, x, p);
... comparator #(4) c4(y, z, q);
...
Parameters can choose logic functionality

Example: Choose adder or subtractor

```verilog
'define ADD 0
'define SUB 1
module addsub (a, b, c);
  parameter width = 8;
  parameter op = `ADD;
  input [width-1:0] a, b;
  output [width-1:0] c;
  assign c = (op == `ADD) ? (a + b) : (a - b);
endmodule

...addsub #(32,`SUB) c32(x, y, z);
...
Parameters cannot generate logic

- Can evaluate constant-valued expressions
- Cannot evaluate nonconstant-valued expressions

This code won’t work

```verilog
module add_or_sub (a, b, op, c);
    parameter width = 8;
    parameter add = 1'b1;
    input [width-1:0] a, b;
    input op;
    output [width-1:0] c;
    assign c = (op == add) ? (a + b) : (a - b);
endmodule
```