Tasks and Functions

- Both let you execute common procedures from different places in a description
- Both facilitate a readable style of code
- Tasks:
  - may have zero or more arguments of type input, output, or inout
  - do not return a value but can pass values through output and inout arguments
  - may contain timing control statements
  - may execute in non-zero simulation time
  - can enable other tasks (including itself) and functions
Tasks and Functions

- Functions:
  - must have at least one **input** argument
  - cannot have **output** or **inout** arguments
  - always return a single value
  - must not contain any timing control statements
  - always execute in 0 simulation time
  - can enable another function but not itself or any task
Tasks

- Declared within a module
- Must be named
- Called from a procedural statement
- Local variables can be declared
- Arguments of a task retain the type they hold in the environment that calls the task
- Arguments are passed by value
- When a task is called, its formal and actual arguments are associated in the order in which the task’s ports have been declared
Example

module bit_counter (data, count);
input [7:0] data;
output [3:0] count;
reg [3:0] count;

always @(data)
count_ones_in_data (data, count);

task count_ones_in_data;
input [7:0] a;
output [3:0] c;
reg [3:0] c;
reg [7:0] tmp;

begin c = 0; tmp = a;
while (tmp)
begin
    c = c + tmp[0];
    tmp = tmp >> 1;
end
endtask
endmodule
Functions

- May implement only combinational behavior
- Declared within a module
- Local variables can be declared
- Referenced in an expression (e.g., RHS of a continuous assignment statement)
- The value of a function is returned by its name
- Implicitly define an register variable having the same name, range, and type as the function itself; this variable must be assigned value within the function body
Example

module word_aligner (w_in, w_out);
input [7:0] w_in;
output [7:0] w_out;

assign w_out = align (w_in);

function [7:0] align;
input [7:0] word;
begin
    align = word;
    if (align != 0)
        while (align[7] == 0)
            align = align << 1;
end
endfunction
endmodule
Example - LFSR

- LFSR = Linear Feedback Shift Register
- Commonly used as binary pattern generator for self-testing circuits
- The following structure shows that
  - $C_N=1$, indicating $Y(0)$ is the input of the leftmost register (at stage N-1)
  - for $j=1,2,…,N-1$
    - if $C_j=1$, then exclusive-or of $Y[j]$ and $Y[0]$ forms the input of the register at stage j-1
    - otherwise, $Y[j]$ is the input of the register at stage j-1

![Diagram of Linear Feedback Shift Register](image)
module Autonomous_LFSR1 (Clock, Reset, Y);

parameter Length = 8;
parameter initial_state = 8'b1001_0001;
parameter [Length-1 :1] Tap_Coefficient = 7'b100_1111;
input Clock, Reset;
output [Length-1: 0] Y;
reg [Length-1: 0] Y;

always @(posedge Clock)
begin
  if (!Reset) Y = initial_state; // Arbitrary initial state
  else Y = LFSR_Value(Y); // Function call
end

function [Length-1: 0] LFSR_Value;
input [Length-1: 0] LFSR_state;
integer Cell_ptr;
begin
  for (Cell_ptr = Length -2; Cell_ptr >= 0; Cell_ptr = Cell_ptr -1)
    if (Tap_Coefficient [Cell_ptr + 1] == 1) // same as c in Figure
      LFSR_Value [Cell_ptr] = LFSR_state [Cell_ptr + 1] ^ LFSR_state [0];
    else
      LFSR_Value [Cell_ptr] = LFSR_state [Cell_ptr +1];
  LFSR_Value [Length - 1] = LFSR_state [0];
endfunction
endmodule

Figure 7.46 Data movement in a linear feedback shift register with modulo-2 (exclusive-or) addition.
Static vs. Dynamic Timing Analysis

- **Static timing analysis**
  - Consider all paths, including false paths which are never exercised

- **Dynamic timing analysis (simulation)**
  - If input stimulus set fails to exercise all functional paths, timing violations can be missed
  - Do not report timing on false paths
Example of Static Timing Analysis

- arrival time/required arrival time/slack
- slack = required arrival time – arrival time
Timing Checks

- Use timing checks to verify the timing of a design
- Timing checks
  - setup
  - hold
  - pulse width
  - clock period
  - skew
  - recovery
  - others
Systems Tasks for Timing Checks

- System tasks for timing checks can be invoked within a behavior in a testbench, or invoked within a `specify` block of a module.

- `$setup` (data_event, ref_event, limit);
  - violation is reported if the period that elapses from data_event to ref_event is less than limit
  - e.g., `$setup` (data, `posedge` clk, 5);

- `$hold` (ref_event, data_event, limit);
  - Violation is reported if the stable period of data_event is less than limit after ref_event
  - e.g., `$hold` (`posedge` clk, data, 2);

- `$setuphold` (data_event, ref_event, s_limit, h_limit)
  - `$setuphold` is the combination of `$setup` and `$hold`
  - e.g., `$setuphold` (data, `posedge` clk, 5, 2);
Systems Tasks for Timing Checks

- **period** (ref_event, limit);
  - violation is reported if the time between two consecutive ref-event
    is less than limit
  - e.g., $period (posedge clk, 16);

- **width** (ref_event, limit);
  - violation is reported if the period that elapses between ref_event
    and the next opposite transition is less than limit
  - e.g., $width (posedge clk, 8);

- **skew** (event_1, event_2, limit);
  - violation is reported if the time between event_1 and event_2
    exceeds limit
  - e.g., $skew (posedge clk1, posedge clk2, 3);

- **recovery** (ref_event, data_event, limit);
  - violation is reported if the time between ref_event and data_event
    exceeds limit
  - e.g., $recovery (posedge set, data, 5)
Systems Tasks for Timing Checks

- **$nochange (posedge clk, data, -1, 2)**
  - checks whether data is stable in the interval (-1,2) relative to posedge clk

- **edge**
  - the edges 01, 10, 0x, x1, 1x, x0 can be used with the specifier edge
  - **$setuphold** (data, edge 01 clk, 5, 2)

- **&&&**
  - **$setup** (data, posedge clk &&& (!reset), 5)
Notifiers in Timing Checks

- When a timing check violation occurs, Verilog reports a violation and the output gets the new value.
- The normal behavior should be for the output to become undefined when a timing violation occurs.
- Example: (an additional port is specified in the sequential UDP which will force the output to an undefined value whenever the notifier register toggles)

```
module dff (data, clock, q);
  input data, clock;
  output q;
  ...
  udp_dff (q, data, clock, notifier);
  specify
    $setup (data, posedge clock, 12, notifier);
    $hold (posedge clock, data, 5, notifier);
    $width (posedge clock, 25, notifier);
  endspecify
endmodule
```
Variable Scope Revisited

- The scope of variables declared within a `begin...end` block is local to the block.
- If a block is named, the variables declared within it can be hierarchically de-referenced from any location in the design – e.g., X.Y.k

```verilog
module X(...);
    ...
    begin: Y
        reg k;
        ...
    end
    ...
endmodule
```
Finite State Machines

Figure 7.56 The generic structure of Mealy (a) and Moore (b) finite state machines.
Descriptive Styles

- **Explicit style**
  - Declare a state register to encode the state

- **Implicit style**
  - Use multiple event controls to describe an evolution of states
  - More abstract and less code than explicit style
  - Description of reset behavior could be more complicated than explicit style
  - Only suitable when a given state can be reached from only one other state
Explicit Style 1

module FSM_style1 (...);
    input ...;
    output ...;
    parameter size = ...;
    reg [size-1:0] state;
    wire [size-1:0] next_state;

    assign the_outputs = ... // a function of state and inputs
                        // or a function of state

    assign next_state = ... // a function of state and inputs

    always @ (nedge reset or posedge clk)
        if (reset == 1'b0) state = start_state; else
        state <= next_state;
endmodule
Explicit Style 2

module FSM_style2 (...);
  input ...;
  output ...;
  parameter size = ...;
  reg [size-1:0] state, next_state;

assign the_outputs = ... // a function of state and inputs
  // or a function of state

always @ (state or the_inputs)
  begin
    // decode for next_state with case or if statement
  end

always @ (negedge reset or posedge clk)
  if (reset == 1'b0) state = start_state; else
    state <= next_state;
endmodule
Explicit Style 3

module FSM_style 3 (...);
  input ...;
  output ...;
  parameter size = ...;
  reg [size-1:0] state, next_state;

always @ (state or the_inputs)
  begin
    // decode for next_state with case or if statement
  end

always @ (negedge reset or posedge clk)
  if (reset == 1'b0) state = start_state; else begin
    state <= next_state;
    the_outputs <= ... // synchronous outputs which depend on
    // inputs and state for a synchronous Mealy machine
  end
endmodule
Example: Speed Controller

- **low**
  - **stopped**
    - **b = 1**
    - **b = 1**
    - **a = 1, b = 0**
  - **a = 1, b = 0**

- **medium**
  - **high**
    - **a = 1, b = 0**
  - **b = 1**

- **a: accelerator**
- **b: brake**

- **speed**
  - **accelerator**
  - **brake**
  - **clock**
// Explicit FSM style
module speed_machine_1 (clock, accelerator, brake, speed);

input clock, accelerator, brake;
output [1:0] speed;
reg [1:0] state, next_state;

parameter stopped = 2’b00;
parameter s_slow = 2’b01;
parameter s_medium = 2’b10;
parameter s_high = 2’b11;

assign speed = state;

always @(posedge clock )
state <= next_state;

always @ ( state or accelerator or brake )
if ( brake == 1’b1 )
case ( state )
  stopped: next_state <= stopped;
  s_low: next_state <= stopped;
  s_medium: next_state <= s_low;
  s_high: next_state <= s_medium;
  default: next_state <= stopped;
endcase
else if ( accelerator == 1’b1 )
case ( state )
  stopped: next_state <= s_low;
  s_low: next_state <= s_medium;
  s_medium: next_state <= s_high;
  s_high: next_state <= s_high;
  default: next_state <= stopped;
endcase
else next_state <= state;
endmodule
```verilog
module speed_machine_2 (clock, accelerator, brake, speed);
    // Implicit FSM
    // Style: Decode the input, decode the state
    // Model has implied states

    input clock, accelerator, brake;
    output [1:0] speed;
    reg [1:0] speed;

    `define stopped 2'b00
    `define low 2'b01
    `define medium 2'b10
    `define high 2'b11

    always @(posedge clock)
        if (brake == 1'b1)
            case (speed)
                `stopped: speed <= `stopped;
                `low: speed <= `low;
                `medium: speed <= `medium;
                `high: speed <= `high;
                default: speed <= `stopped;
            endcase
        else if (accelerator == 1'b1)
            case (speed)
                `stopped: speed <= `low;
                `low: speed <= `medium;
                `medium: speed <= `high;
                `high: speed <= `high;
                default: speed <= `stopped;
            endcase
        else
            speed <= speed;
endmodule
```
Another Implicit Style

```verilog
module speed_machine_3 (clock, accelerator, brake, speed);
    // Style: case of state and inputs. Model has implied states

    input clock, accelerator, brake;
    output [1:0] speed;
    reg [1:0] speed;

    `define stopped 2'b00
    `define low 2'b01
    `define medium 2'b10
    `define high 2'b11

    always @ (posedge clock)
        case (speed)
            `stopped: if (brake == 1'b1) speed <= `stopped;
            else if (accelerator == 1'b1) speed <= `low;
            `low: if (brake == 1'b1) speed <= `low;
                else if (accelerator == 1'b1) speed <= `medium;
            `medium: if (brake == 1'b1) speed <= `low;
                else if (accelerator == 1'b1) speed <= `high;
            `high: if (brake == 1'b1) speed <= `medium;
                default: speed <= `stopped;
        endcase
endmodule
```
Up-Down Counter (Explicit Style)

```verilog
module Up_Down_Explicit (count, up_dwn, clock, reset_);
    output [2:0] count;
    input [1:0] up_dwn;
    input clock, reset_
    reg [2:0] count, next_count;

    always @(negedge clock or negedge reset_)
        if (reset_ == 0) count = 3'b0; else count = next_count;

    always @ (count or up_dwn)
        begin
            case (count)
                0: case (up_dwn)
                    0, 3: next_count = 0;
                    1: next_count = 1;
                    2: next_count = 3'b111;
                    default next_count = 0; endcase
                1: case (up_dwn)
                    0, 3: next_count = 1;
                    1: next_count = 2;
                    2: next_count = 0;
                    default next_count = 1; endcase
                2: case (up_dwn)
                    0, 3: next_count = 2;
                    1: next_count = 3;
                    2: next_count = 1;
                    default next_count = 2; endcase
                3: case (up_dwn)
                    0, 3: next_count = 3;
                    1: next_count = 4;
                    2: next_count = 2;
                    default next_count = 3; endcase
                4, 5, 6, 7: if (up_dwn == 0 || up_dwn == 3) next_count = count;
                                else if (up_dwn == 1) next_count = count + 1;
                                else if (up_dwn == 2) next_count = count -1;
                                else next_count = 0;
            endcase
        end
    endmodule
```
Implicit Styles

module Up_Down_Implicit1 (count, up_dwn, clock, reset_);
    output [2:0]         count;
    input  [1:0]         up_dwn;
    input   clock, reset :
    reg      [2:0]         count;

    always @ (negedge clock or negedge reset_)
        if (reset_ == 0) count = 3'b00; else
            if (up_dwn == 2'b00 || up_dwn == 2'b11) count = count; else
                if (up_dwn == 2'b01) count = count + 1; else
                    if (up_dwn == 2'b10) count = count -1;
    endmodule

module Up_Down_Implicit2 (count, up_dwn, clock, reset_);
    output [2:0]         count;
    input  [1:0]         up_dwn;
    input   clock, reset_:
    reg      [2:0]         count, next_count;

    always @ (negedge clock or negedge reset_)
        if (reset_ == 0) count = 3'b00; else count = next_count;

    always @ (count or up_dwn) begin
        if (up_dwn == 2'b00 || up_dwn == 2'b11) next_count = count; else
            if (up_dwn == 2'b01) next_count = count + 1; else
                if (up_dwn == 2'b10) next_count = count -1; else
                    next_count = count;
    end
    endmodule
Polling Circuit
State Transition Graph

Service request

Service code
Verilog Code

module polling (s_request, s_code, clk, rst);
    `define client1 2’b01
    `define client2 2’b10
    `define client3 2’b11
    `define none 2’b00
    input [3:1] s_request;
    input clk, rst; output [1:0] s_code;
    reg [1:0] next_client, present_client;
    always @ (posedge clk or posedge rst)
        begin if (rst)
            present_client = `none;
        else
            present_client = next_client;
        end
    assign s_code[1:0] = present_client;
    always @ (present_client or s_request)
        begin
            poll_for_clients (present_client, s_request, next_client);
        end
always @ (posedge clk or posedge rst)
    begin if (rst)
        present_client = `none;
    else
        present_client = next_client;
    end
assign s_code[1:0] = present_client;
always @ (present_client or s_request)
    begin
        poll_for_clients (present_client, s_request, next_client);
    end
endmodule

task poll_for_clients;
    input [1:0] present_client;
    input [3:1] s_request;
    output [1:0] next_client;
    reg [1:0] contender; integer N;
    begin: polling
        contender = `none;
        for (N = 3; N >= 1; N = N - 1)
            begin:
                if (s_request[N]) begin
                    if (present_client == N)
                        contender = present_client;
                    else begin
                        next_client = N;
                        disable polling; end
                    end
                end
                else begin
                    next_client = N;
                    disable polling; end
            end
        end
        if ((next_client == `none) &&
            (contender))
            next_client = contender; end
endtask
endmodule