Tasks and Functions

Verilog supports two types of sub-programs, *tasks* and *functions*.

- Tasks create a hierarchical organization of the procedural stmts within a behavior.
- Functions substitute for an expression.

Tasks are declared within a module and may be referenced only from within a behavior.

Task parameters are copied when the task is called, i.e., are passed by value.

See text for examples, and other rules.

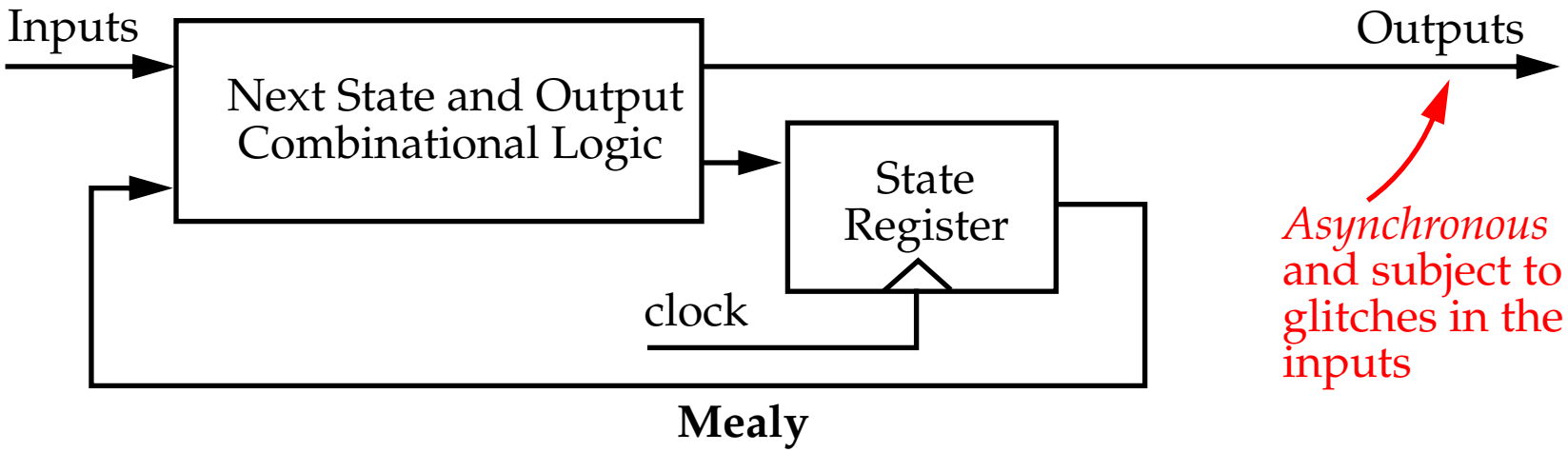
Functions may implement only combinational behavior.

No timing controls are permitted, it must have at least once **input** argument, **output** and **inout** arguments are not permitted.

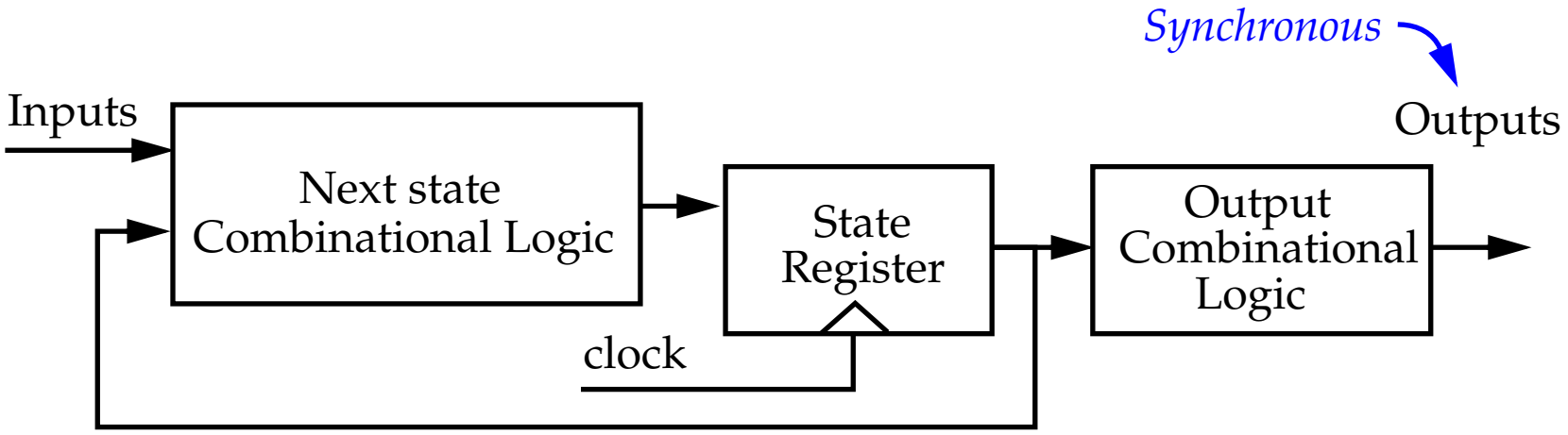
The definition implicitly defines an internal **reg** variable with the same name, range and type as the function itself, that is assigned to within the function (see text).

Behavioral Models of FSMs

Two basic forms of Finite State Machines



Mealy



Moore

Behavioral Models of FSMs

There are two descriptive styles of FSMs.

- *Explicit*: declares a state register to encode the machine's state. A behavior **explicitly** assigns values to the state register to govern the state transitions.
- *Implicit*: uses multiple event controls within a cyclic behavior to implicitly describe an evolution of states.

Explicit FSMs, several styles are possible:

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

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

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

Explicit FSMs

A second style replaces the continuous assignment generating the *next_state* with asynchronous (combinational) behavior:

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

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

   always @ ( state or the_inputs )
      // decode next_state with case or if stmt

   always @ (negedge reset or posedge clk)
      if (reset == 1'b0) state <= start_state;
      else state <= next_state; //Non-blocking or procedural assignment
endmodule
```

This latter style can exploit the **case** stmt and other procedural constructs for descriptions that are complex.

Note that in both styles, the outputs are *asynchronous*.

Explicit FSMs

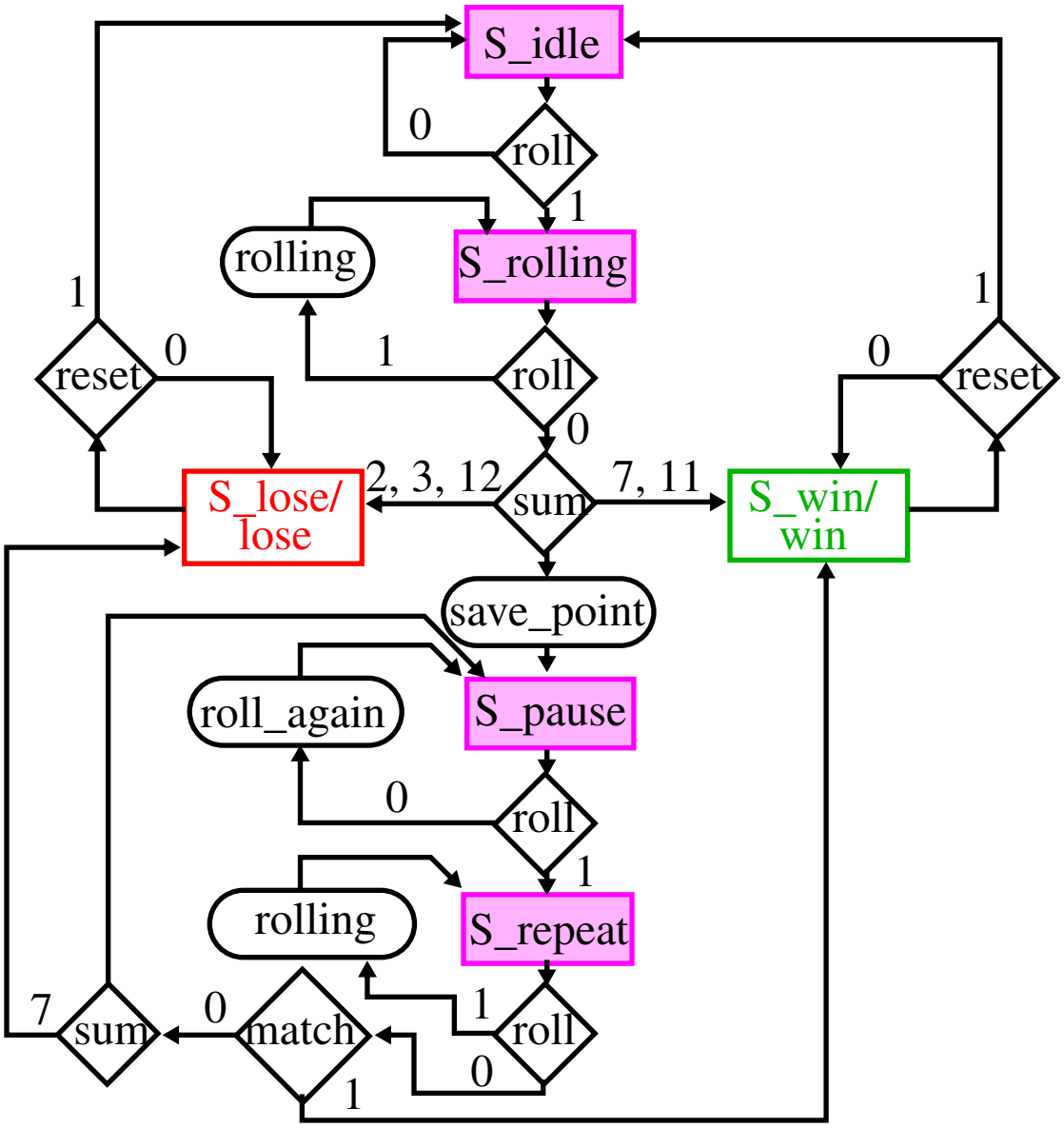
It may be desired to *register* the outputs, and make them *synchronous*:

```
module FSM_style3 (...)
   input ...;
   output ...;
   parameter size = ...;
   reg [size-1 : 0] state, next_state;

   always @ ( state or the_inputs )
      // decode next_state with case or if stmt

   always @ (negedge reset or posedge clk)
      if (reset == 1'b0) state <= start_state;
      else begin
         state <= next_state;
         outputs <= some_value (inputs, next_state);
      end
endmodule
```

State machines can be represented in

- Tabular format (state transition table)
- Graphical format (state transition graph)
- Algorithmic state machine (ASM) chart

Explicit FSMs: Craps example



Explicit FSMs: Craps example

Player roles the die with 3 possible outcomes:

- Sum is 7 or 11, player wins
- Sum is 2, 3, or 12, player loses.
- Otherwise, sum is declared to be the player's *point*.

To win, player must roll repeatedly until the point is made, but before rolling a 7, i.e., if 7 rolled before the point, the player loses.

In our machine, a *rolling unit* generates random values

At the rising edge of clock, with *roll* asserted, the *rolling unit* generates 2 values *D_left* and *D_right*.

When *roll* is de-asserted, the *scoring unit* computes the *sum*, *D_left* and *D_right*.

Then *win*, *lose* or *roll_again* may be asserted, depending on the result.

Reset must be asserted before another player can play.

Explicit FSMs: Craps example

```
module Crap_shoot
  (clk, reset, point, roll, win, match, lose, roll_again, rolling, blank, D_left, D_right,
    sum)
  input clk, reset, roll;
  output win, lose, match, roll_gain, rolling, blank;
  output [3:0] point;
  output [2:0] D_left, D_right;
  output [3:0] sum;
  parameter S_idle = 0;
  parameter S_rolling = 1;
  parameter S_pause = 2;
  parameter S_repeat = 3;
  parameter S_lose = 4;
  parameter S_win = 5;

  wire match, rolling, roll_again, win, lose, save_point;
  reg [2:0] D_left, D_right;
  wire [3:0] sum = D_left + D_right;
  reg [2:0] state, next_state;
  reg [3:0] point;
```

Explicit FSMs: Craps example

```
// Rolling Unit
   always @( posedge clk or posedge reset )
      if (reset) begin D_left <= 1; D_right <= 1; end
      else begin
         if (D_left < 6) D_left <= D_left + 1; else D_left <= 1;
         if (D_left == 6 && D_right < 6) D_right <= D_right + 1; else
            if (D_left == 6 && D_right == 6) D_right <= 1;
      end

// Scoring Unit
   assign match = (sum == point);
   assign roll_again = (state == S_pause && !roll);
   assign rolling = ((state == S_rolling && roll) || (state == S_repeat && roll));
   assign save_point = ((state == S_rolling) && !roll &&
      sum != 2 &&
      sum != 3 &&
      sum != 12 &&
      sum != 7 &&
      sum != 11);
   assign win = (state == S_win);
   assign lose = (state == S_lose);
   assign blank = (point < 2);
```

Explicit FSMs: Craps example

```
// Control Unit
   always @( posedge save_point or posedge reset )
      if (reset) point <= 0;
      else point <= sum;

   always @(posedge clk or posedge reset)
      if (reset) state <= S_idle;
      else state <= next_state;

   always @(state or sum or roll or match)
      case (state)
         S_idle: if (roll) next_state <= S_rolling; else next_state <= S_idle;
         S_rolling: if (roll) next_state <= S_rolling; else
                        if (sum == 2 || sum == 3 || sum == 12) next_state <= S_lose;
                        else
                           if (sum == 7 || sum == 11) next_state <= S_win;
                           else next_state <= S_pause;
         S_pause: if (roll) next_state <= S_repeat; else next_state <= S_pause;
         S_repeat: if (roll) next_state <= S_repeat; else
                        if (match) next_state <= S_win; else
                           if (sum == 7) next_state <= S_lose; else next_state <= S_pause;
         S_win: next_state <= S_win;
         S_lose: next_state <= S_lose;
      endcase endmodule
```

Implicit FSMs

Here, the state of the machine is *not* explicitly-declared in state registers.
Instead, the state is implied by the evolution of the activity flow.

It is more abstract and can require less code.

However, it is only good for machines in which a given state can be reached from only one other state.

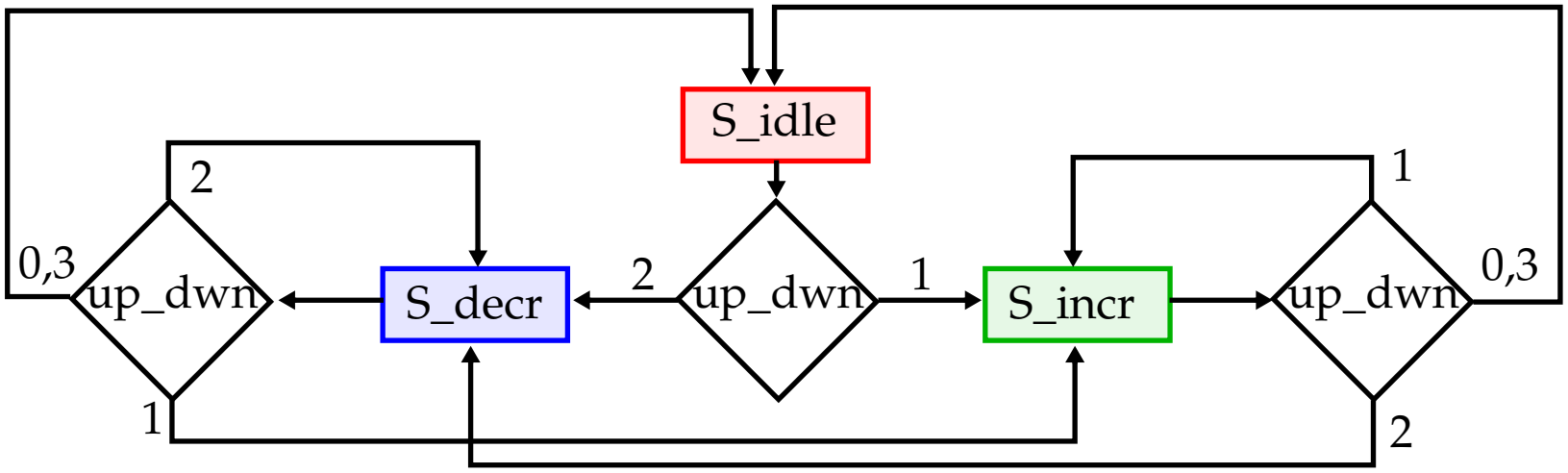
Also, FSMs with multiple event controls make reset control more complicated because it needs to be possible to reset from every state.

Consider the alternatives for a state machine for an Up Down counter.

- Up_Down_Implicit1: *count* is the output, no state register declared.
- Up_Down_Implicit2: same except state logic and combinational logic is in separate event control blocks.
- Up_Down_Explicit: state transitions are explicitly enumerated by the **case** statement. Here, the size of the FSM (not shown) depends on the word length of *count*.

Implicit FSMs

ASM for Up Down Counter




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

   always @(negedge clk or negedge reset)
      if (reset == 0) count = 3'b0;
      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
```

Implicit FSMs

```
module Up_Down_Implicit2(count, up_dwn, clk, reset)
   output [2:0] count;
   input [1:0] up_dwn;
   input clk, reset;

   reg [2:0] count, next_count;

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

   always @(count or up_dwn)
      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;
endmodule
```

Explicit FSM Variant

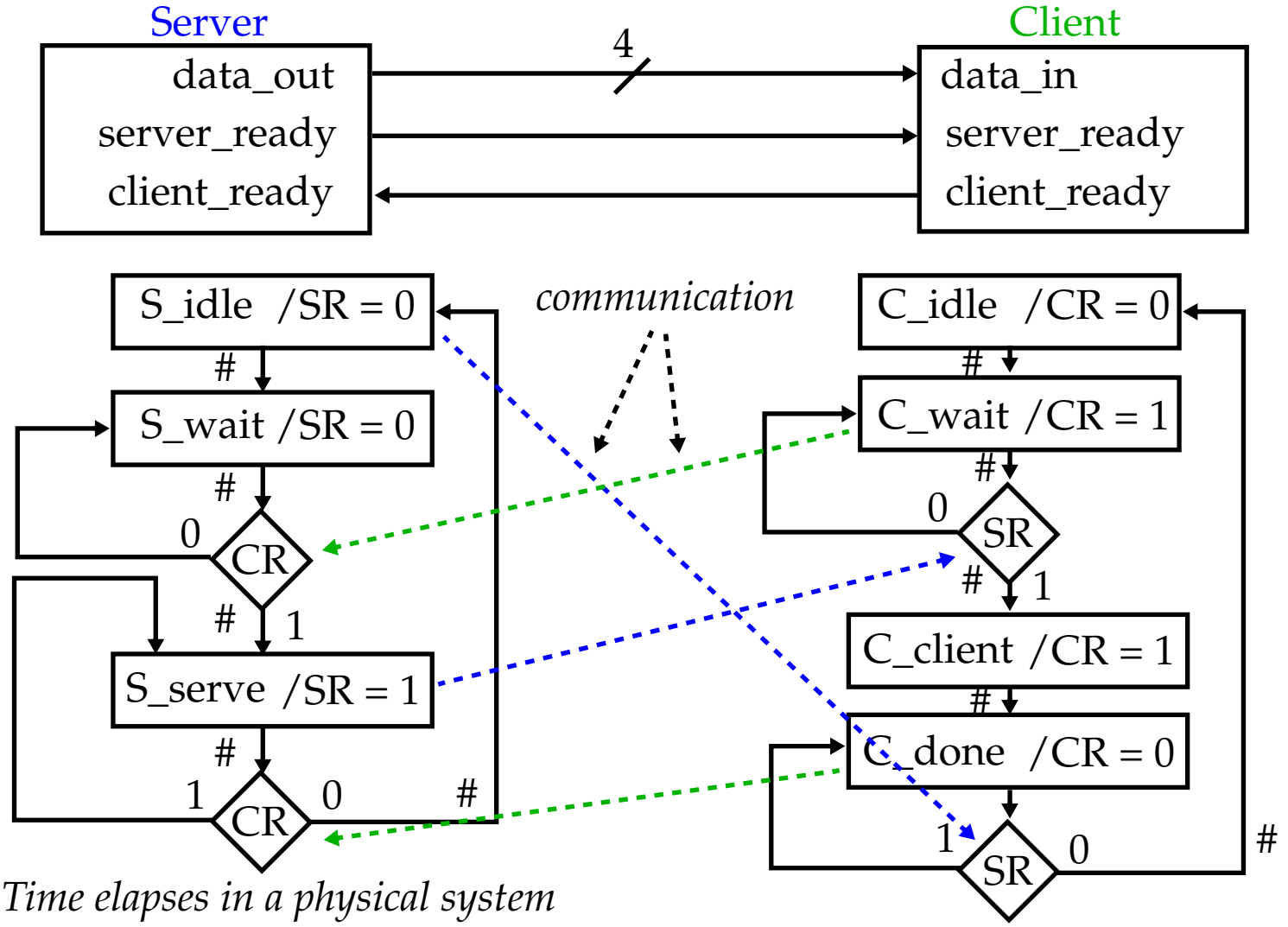
```
module Up_Down_Explicit(count, up_dwn, clk, reset)
   output [2:0] count;
   input [1:0] up_dwn;
   input clk, reset;

   reg [2:0] count, next_count;

   always @(negedge clk 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
```

Explicit FSM Variant

```
            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
```

FSM for Handshaking

Handshaking occurs between a "client" (receiver of service) and a "server" (provider of service).



#: *Time elapses in a physical system*

FSM for Handshaking

The *S_idle* and *C_idle* states are wait states where communication signals, *server_ready* (*SR*) and *client_ready* (*CR*), are not asserted.

S_idle corresponds to an interval of time required to prepare for service.

S_wait indicates server is awaiting a service request from a client.

C_idle corresponds to a period in which service is not desired.

The server remains in state *S_wait* until the client asserts *CR*.

When asserted, the server moves to state *S_serve* and asserts *SR*.

This action signals the client that the server has made the data available on the bus.

The client then enters state *C_client* and, after some interval, removes data from the bus, then de-asserts *CR*.

The server then returns to *S_idle* and de-asserts *SR*.

The client returns to *C_idle* on detecting this action.

FSM for Handshaking

```
module server(data_out, server_ready, client_ready);
   output [3:0] data_out;
   output server_ready;
   input client_ready;

   reg server_ready;
   reg [3:0] data_out;

   task pause;                          // To emulate the delay in a real system
      reg [3:0] delay;
      begin
         delay = $random; if (delay == 0) delay = 1;
         #delay;
      end endtask

   always
      begin
         server_ready = 0;
         pause;
         wait (client_ready)
         pause; data_out = $random;
         pause; server_ready = 1;
         wait (!client_ready)
         pause;
      end
   endmodule
```

FSM for Handshaking

```
module client(data_in, server_ready, client_ready);
   input [3:0] data_in;
   input server_ready;
   output client_ready;
   reg client_ready;
   reg [3:0] data_reg;
   task pause;                          // To emulate the delay in a real system
      reg [3:0] delay;
      begin
         delay = $random; if (delay == 0) delay = 1;
         #delay;
      end endtask
   always
      begin
         client_ready = 0;
         pause; client_ready = 1;
         forever begin
            wait (server_ready)
            pause; data_reg = data_in;
            pause; client_ready = 0;
            wait (!server_ready)
            pause; client_ready = 1;
      end
endmodule
```