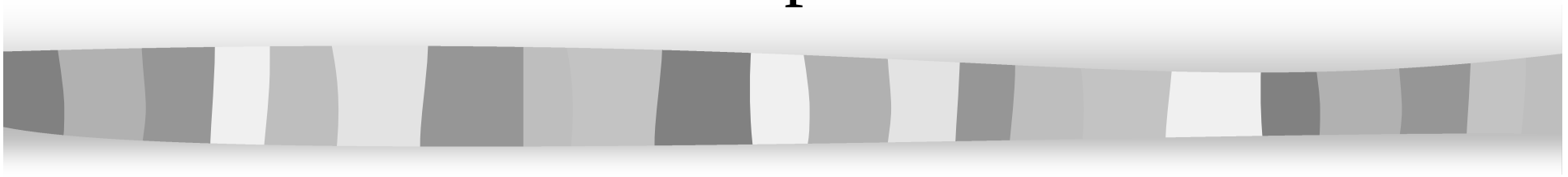
Behavioral Descriptions



Behaviors (Processes)

- Two constructs
  - **initial**: one-shot activity flow (not synthesizable but good for testbenches)
  - **always**: cyclic (repetitive) activity flow
- Use procedural statements that assign values to register variables (exception: **force**…**release**)

Behaviors

- Continuous assignments and primitives assign outputs whenever there are events on inputs
- Behaviors assign values when an assignment statement in the activity flow executes. (Input events on the RHS do not initiate activity – control must be passed to the statement.)

Behaviors

- Body may consist of a single statement or a block statement
- Block statement begins with **begin** and ends with **end**
- Behaviors are an elaborate form of continuous assignments or primitives but operate on registers rather than nets (exception: **force**…**release**)

initial

- Run once

```
…
initial                    // An “initial” behavior
 begin
    sig_a = 0;             // Procedural assignments
    sig_b = 1;             // execute sequentially.
    sig_c = 0;
 end
…
```

always

- Infinite loop until simulation stops

```
…
initial
 clock = 0;

always
 #10 clock = ~clock;

initial
 #100 $finish;
…
```

Structural vs. Behavioral Descriptions

```
module my_module(…);
  …
  assign …; // continuous assignment
  and (…);  // instantiation of a primitive
  adder_16 M(…);  // instantiation of a module

  always @ (…)
    begin … end

   initial
    begin … end

endmodule
```

Procedural Assignment

- Assign value to registers
- Blocking procedural assignment
  - Use “=”
  - An assignment is completed before the the next assignment starts

    a = 0; a = 1; c = a; // c = 1
- Non-blocking procedural assignment
  - Use “<=”
  - Assignments are executed in parallel

    a = 0; a <= 1; c = a; // c = 0

    d <= 0; d <= 1; // d = 0 or 1?

Examples

```
a = 1;
b = 0;
…
a <= b;          // Use b=0
b <= a;          // Use a=1
```

```
a = 1;
b = 0;
…
b <= a;          // Use a=1
a <= b;          // Use b=0
```

Examples

```
a = 1;
b = 0;
…
a = b;          // Use b=0
b = a;          // Use a=0
```

```
a = 1;
b = 0;
…
b = a;          // Use a=1
a = b;          // Use b=1
```

Procedural Assignments – Some Rules

- A register (net) variable can be referenced anywhere in a module
- A register variable can be assigned only within a procedural assignment, task or function
- A register variable cannot be **input** or **inout**
- A net variable may not be assigned within a behavior, task or function (exception: **force**…**release**)
- A net variable within a module must be driven by a primitive, continuous assignment, **force…release**, or module port

Procedural Continuous Assignment (PCA)

- **assign … deassign**
  - Assign value to register
  - Dynamic binding to target register
  - Override all procedural assignments to target register
  - Binding can not be removed until **deassign** or a new PCA is established
  - **deassign** is optional
  - Synthesis tools might not support it
  - Can be used to model level-sensitive behavior of combinational logic, transparent latches, and asynchronous control of sequential logic

Example

```
module mux4_PCA (a, b, c, d, select, y_out);
   input          a, b, c, d;
   input [1:0]    select;
   output         y_out;
   reg            y_out;

   always @ (select)
     if (select == 0) assign y_out = a; else
     if (select == 1) assign y_out = b; else
     if (select == 2) assign y_out = c; else
     if (select == 3) assign y_out = d; else
     assign y_out = 1'bx;
  endmodule
```

Alternative

```
module mux4_PCA (a, b, c, d, select, y_out);
   input          a, b, c, d;
   input [1:0]    select;
   output         y_out;
   reg            y_out;

   always @ (select or a or b or c or d)
     if (select == 0) y_out = a; else
     if (select == 1) y_out = b; else
     if (select == 2) y_out = c; else
     if (select == 3) y_out = d; else
     y_out = 1'bx;
  endmodule
```

Example

```
module Flop_PCA(preset, clear, q, qbar, clock, data);
 input preset, clear, clock, data;
 output q, qbar;
 reg q;

assign qbar = ~q;

always @ (negedge clock)
  q = data;

always @ (clear or preset)
  begin
    if (!clear) assign q = 0;
    else if (!preset) assign q = 1;
    else deassign q;
  end

endmodule
```

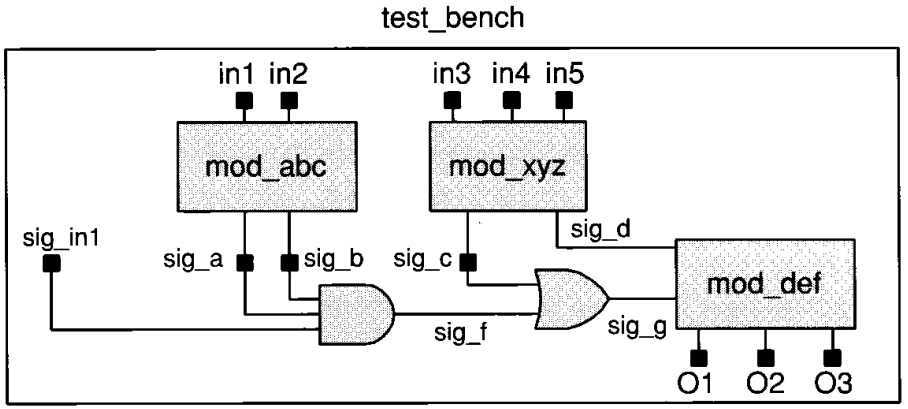
Procedural Continuous Assignment (PCA)

- **force … release**
  - Assign value to net or register
  - Dynamic binding to target net or register
  - Overwrite primitive and continuous assignment to a net, and override procedural assignment and **assign … deassign** to a register
  - Binding can not be removed until **release**
  - Synthesis tools might not support it
  - Can be used with hierarchical de-referencing in testbenches

Example

- …

```
force sig_a = 1;
force sig_b = 1;
force sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
// Insert code to construct tests
release sig_a;
release sig_b;
release sig_c;
…
```



Assignment Modes

Variable type	Output of primitive	Continuous assignment	Procedural assignment	**assign … deassign**	**force … release**
Net	Yes	Yes	No	No	Yes
Register	Comb - No Seq - Yes	No	Yes	Yes	Yes

Procedural Timing Controls

- Mechanisms
  - delay control operator (#)
  - event control operator (@)
  - event **or**
  - named events
  - **wait** construct

Delay Control Operator (#)

- Suspend the activity flow at the location of the operator
- Examples:

```
…
always
 begin
   #0 clock = 0;
   #50 clock = 1;
   #50;
 end
…
```

```
…
always
 begin
   #clock_period/2;
   clock = ~clock;
 end
…
```

Event Control Operator (@)

- Synchronize execution to an event
- Example 1:

```
…
@ signal_1 a=b;
…
```

- Example 2:

```
…
@ (event_a) begin
  …
  @ (event_b) begin
  …
  end
end
…
```

posedge and **negedge**

- **posedge**: 0->1, 0->x, x->1
- **negedge**: 1->0, 1->x, x->0
- Example:
  - **always** @ (**posedge** clock) #10 b = a;

Event **or**

```
…
always @ (set or reset or posedge clk)
  begin
     if (reset == 0) q = 0;
     else if (set == 0) q = 1;
     else if (clk == 1) q = data;
  end
…
```

The above behavior does not correctly model a positive edge-triggered D flip-flop with asynchronous (active-low) set and reset.

Fix

```
…
always @ (negedge set or negedge reset or posedge clk)
  begin
     if (reset == 0) q = 0;
     else if (set == 0) q = 1;
     else if (clk == 1) q = data;
  end
…
```

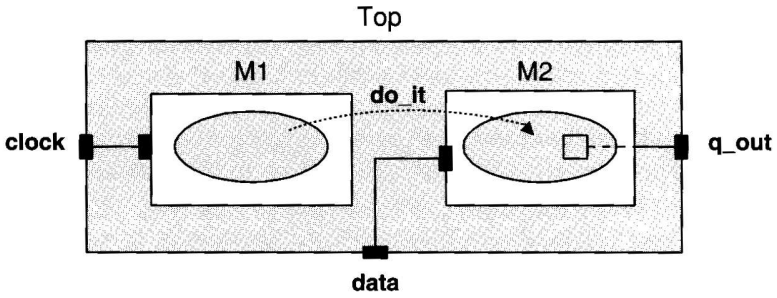
Named Events

- Synchronization within and between modules
- A named event can be declared only in a module, with keyword **event**; it can then be referenced within that module directly, or in other modules by hierarchically de-referencing the name of the event.
- The occurrence of an event is explicitly determined by a procedural statement using the event-trigger operator, "->".
- Example:

```
…
  event up_edge;

  always @ (posedge clk)
    -> up_edge;

  always @ (up_edge)
    …
…
```

Example

```
module top (clk, data, q);
 input clk, data;
 output q;

 talker M1 (clk);
 receiver M2 (q, data);
endmodule
```




```
module talker (clk);
 input clk;
 event do_it;
 always @ (posedge clk)
    -> do_it;
endmodule

module receiver (q, data);
 input data;
 output q;
 reg q;
 always @ (top.M1.do_it)
    q = data;
endmodule
```

wait

- Suspend activity flow until expression following **wait** is true
- Can model level-sensitive behavior
- Example:

  wait (enable) a = b;

Intra-Assignment Delay

- Postpone assignment, not evaluation
- Examples (blocking assignments)

```
…
a = #5 b;
c = d;
…
```

```
…
e = @ (bus) f;
g = h;
…
```

Intra-Assignment Delay

- Examples (non-blocking assignments)

```
…
initial begin
 @ (posedge clk)
   g <= @ (bus) k;
   p <= q;
end
…
```

```
…
always begin
 @ (posedge clock)
   g <= @ (bus) k;
end
…
```

Blocking vs. Non-Blocking Assignments

```
// blocking
a = #10 1;
b = #2 0;
c = #3 1;
```

```
// non-blocking
a <= #10 1;
b <= #2 0;
c <= #3 1;
```

Examples

```
assign y = a | b;
```

```
always @ (a or b)
  y = a | b;
```

```
assign #5 y = a | b;
```

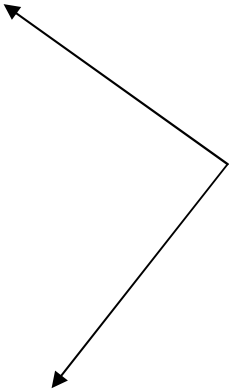
```
always @ (a or b)
  #5 y = a | b;
```

```
always @ (a or b)
  y = #5 a | b;
```

```
always @ (a or b)
  y <= #5 a | b;
```

Repeated Intra-Assignment Delay

reg_a = **repeat** (5) @ (**negedge** clock) reg_b;



equivalent

```
begin
  temp = reg_b;
  @ (negedge clock);
  @ (negedge clock);
  @ (negedge clock);
  @ (negedge clock);
  @ (negedge clock);
  reg_a = temp;
end
```

Example

```
module repeater;
   reg clock;
   reg reg_a, reg_b;

   initial
      clock = 0;

   initial begin
      #5   reg_a = 1;
      #10  reg_a = 0;
      #5   reg_a = 1;
      #20  reg_a = 0;
   end

   always
      #5 clock = ~ clock;

   initial
      #100 $finish;
   initial
      begin
         #10 reg_b = repeat (5) @ (posedge clock) reg_a;
      end
endmodule
```

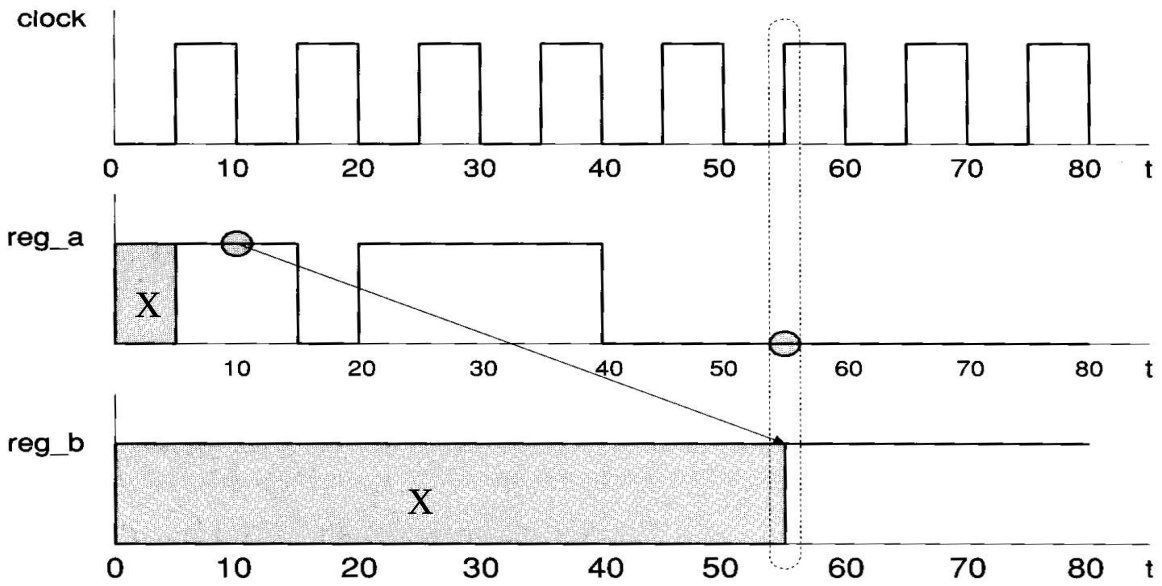


Figure 7.20 Waveforms for repeated intra-assignment delay.

Simulation of Procedural Assignments

- At a given time step,
  - evaluate expressions on RHS
  - execute blocking assignments
  - execute non-blocking assignments that do not have intra-assignment timing control
  - execute past assignments which have been scheduled to execute in the current time step
  - execute **$monitor** (note: **$display** is executed when it is encountered, but is executed before non-blocking assignments)
  - Advance simulation time

$display vs. **$monitor**

```
initial begin
  a = 1;
  b = 0;
  a <= b;
  b <= a;
  $display (“a=%b b=%b”, a, b);
end
```

display: a=1 b=0

```
initial begin
  a = 1;
  b = 0;
  a <= b;
  b <= a;
  $monitor (“a=%b b=%b”, a, b);
end
```

monitor: a=0 b=1

Indeterminate Assignments

```
module ambiguity();
 reg a, b, c;
 ...
 always @ (a)
   c = a;

 always @(a)
   c = b;
 …
endmodule
```

No Ambiguity

```
module example;
 reg wave;
 reg [2:0] i;

 initial
   begin
     for (i=0; i<=5; i=i+1)
        wave <= #(i*10) i[0];
   end
endmodule
```

Activity Flow Control

- conditional operator (?…:)
- **case**, **casex**, **casez**
- **if … else**
- loops: **for**, **while**, **repeat**, **forever**
- **disable**
- **fork**…**join**

Conditional Operator

- ?…:
- Can be also used in a procedural statement to assign value to a register variable
- Example:

```
always @ (posedge clock)
  q = s ? a : b;
```

case

- Require complete bitwise match, so expression and case item must have the same bit length
- Example:

```
…
always @ (a or b or c or d or select)
 begin
   case (select)
     0: y = a;
     1: y = b;
     2: y = c;
     3: y = d;
     default: y = 1'bx;
   endcase
 end
…
```

casex and **casez**

- **casex** ignores bit positions that have “x” or “z”
- **casez** ignores bit positions that have “z”, and uses “?” as don’t cares
- Example:

```
…
always @ (decode)
  casez (word)
    16’b0000_????_????_????:;
```

if…**else**

- Numerical value of 0, or value x or z is logic false (non-zero numerical value is evaluated to true)
- **else** is paired with nearest **if** when ambiguous (use **begin**…**end** in nesting to clarify)

Examples

(a) **if** (a < b) c = d +1;

(b) **if** (a < b);

(c) **if** (k == 1)

begin : A_block

sum_out = sum_reg;

c_out = c_reg;

end

(d) **if** (a < b)

sum = sum +1;

else

sum = sum + 2;

(e) **if** (a == 1) sig_out = reg_a; **else**

if (a == 2) sig_out = reg_b; **else**

if (a == 3) sig_out = reg_c;

repeat

- Example:

```
…
word_address = 0;
repeat (memory_size)
  begin
    memory[word_address] = 0;
    word_address = word_address +1;
  end
...
```

for

```
…
reg [15:0] data;
integer k;

for (k = 4; k > 0; k = k – 1)
 begin
  data[k+10] = 0;
  data[k+2] = 1;
 end
…
```

```
…

reg [3:0] k;

for (k = 0; k <= 15; k = k + 1)

...
```

4-bit Carry Look-ahead Adder

```
module Add_prop_gen (sum, c_out, a, b, c_in); // 4-bit look-ahead carry adder
                                              // behavioral model
  output [3:0]    sum;
  output          c_out;
  input  [3:0]    a, b;
  input           c_in;

  reg    [3:0] carrychain;
  wire   [3:0] g = a & b; // carry generate, continuous assignment, bitwise and
  wire   [3:0] p = a ^ b; // carry propagate, continuous assignment, bitwise xor

  always @ (a or b or c_in)                    // event "or"
    begin : carry_generation                   // usage: block name
      integer i;
      #0 carrychain[0] = g[0] + (p[0] & c_in); // needed for simulation
      for(i = 1; i <= 3; i = i + 1)
        begin
          carrychain[i] = g[i] | (p[i] & carrychain[i-1]);
        end
    end

  wire [4:0] shiftedcarry = {carrychain, c_in} ; // concatenation
  wire [3:0] sum = p ^ shiftedcarry;             // summation
  wire c_out = shiftedcarry[4];                  // carry out, usage: bit select
endmodule
```

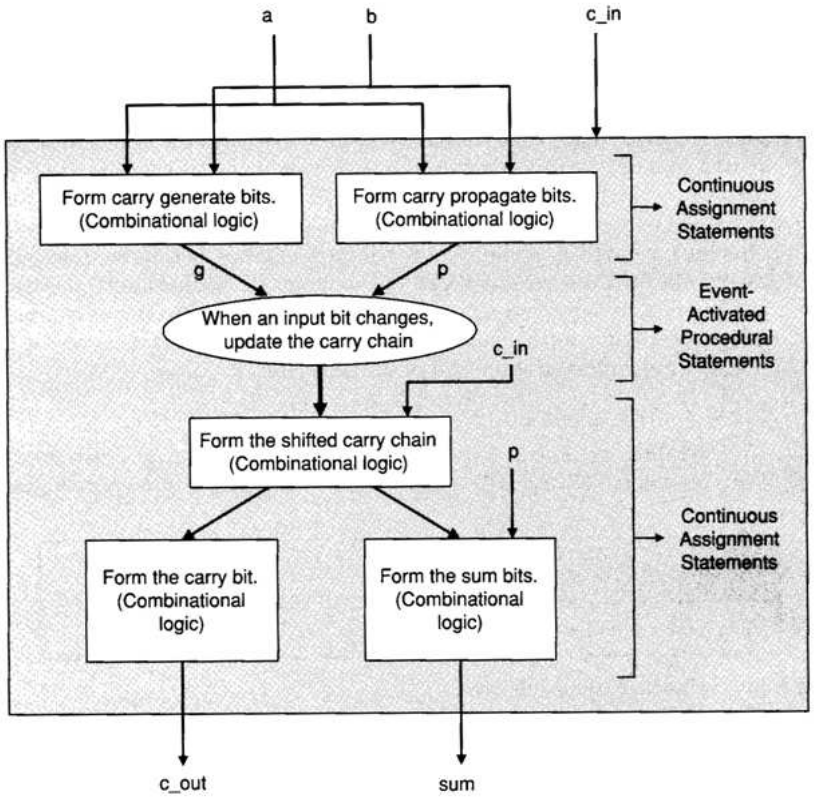


Figure 7.35 Organization of a carry look-ahead algorithm.

while

```
…
begin
  reg [7:0] temp;
  counter = 0;
  temp = a;

  while (temp)
    begin
      if (temp[0]) counter = counter + 1;
      // if statement can be replaced by
      // counter = counter + temp[0];
      temp = temp >> 1;
    end
end
…
```

forever

```
initial
 begin: clock_loop
     clock = 0;
     forever
        #10 clock = ~clock;
 end

initial
  #100 disable clock_loop;
```

disable

```
module find_first_one (a, trigger, i);
  input [15:0] a;
  input trigger;
  output [3:0] i;
  reg [3:0] i;

  always @ trigger
     for (i = 0; i < 16; i = i + 1)
        if (a[i] == 1) disable;

endmodule
```

Comparisons

- **always** vs. **forever**
- **disable** vs. **$finish**

Parallel Activity Flow: **fork** … **join**

- Create parallel threads of activity
- Not supported by synthesis tools
- Wait for completion of all threads between **fork** and **join**
- No ordering of the execution of the threads
- Ambiguity could exist due to interaction between threads

Example

```
fork
  #50 a = 1;
  #100 a = 0;
  #150 a = 1;
  #200 a = 0;
join
```



```
begin
  #50 a = 1;
  #50 a = 0;
  #50 a = 1;
  #50 a = 0;
end
```

Race Condition vs. No Race Condition

```
fork
  #150 a = b;
  #150 c = a;
join
```

```
fork
  a = #150 b;
  c = #150 a;
join
```

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
    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

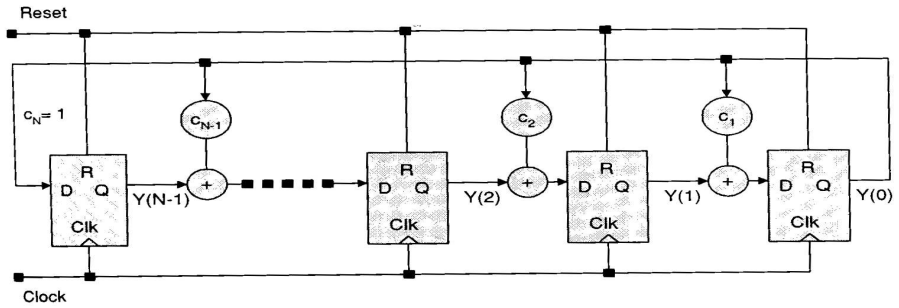


Figure 7.45 Linear feedback shift register with modulo-2 (exclusive-or) addition.

LFSR

```
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];
      end
   endfunction
endmodule
```

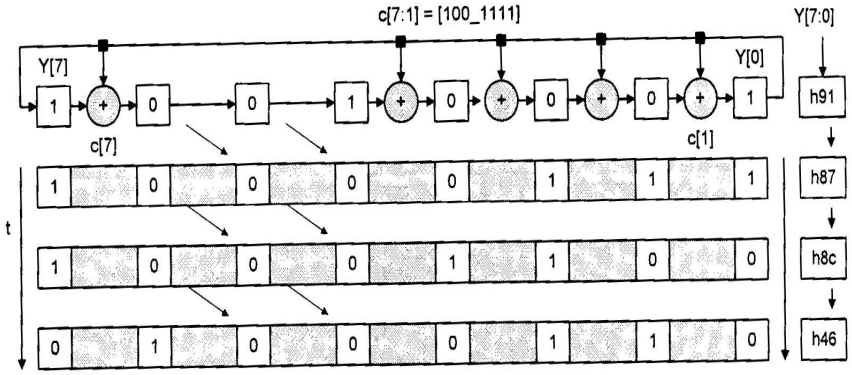
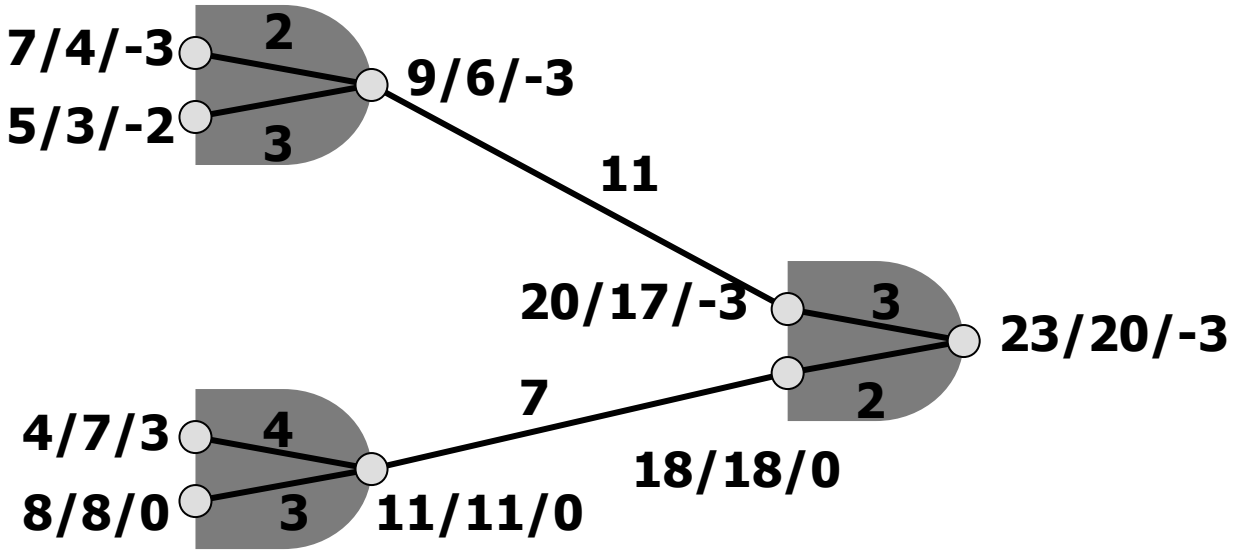


Figure 7.46 Data movement in a linear feedback shift register with modulo-2 (exclusive-or) addition.

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)

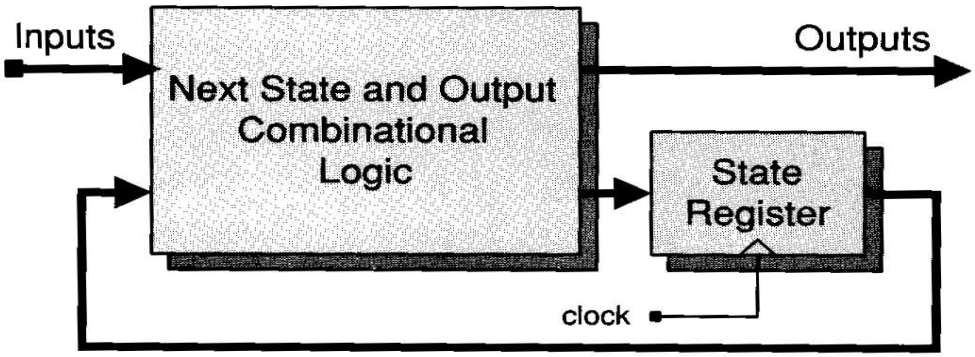
```
module dff (data, clock, q);
 input data, clock;
 output q;
 …
 udp_dff (q, data, clock, notifier);
 specify
  $setup (data, posedge clock, 12, notifier);
  $hold (posedge clock, data, 5, notifier);
  $width (posedge clock, 25, notifier);
 endspecify
endmodule
```

Variable Scope Revisited

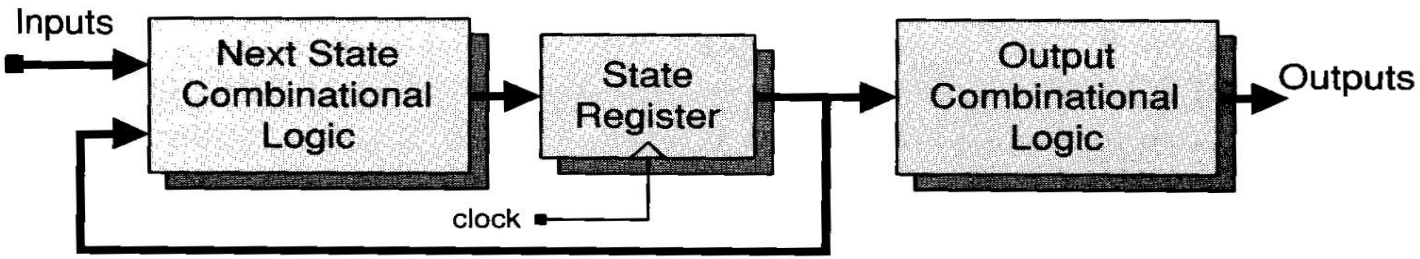
- The scope of variables declared within a **begin**…**end** block is local to the block
- If a block is named, the variables declared within it can be hierarchically de-referenced from any location in the design
  - e.g., X.Y.k

```
module X(...);
  ...
  begin: Y
    reg k;
    ...
  end
  ...
endmodule
```

Finite State Machines



(a)



(b)

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

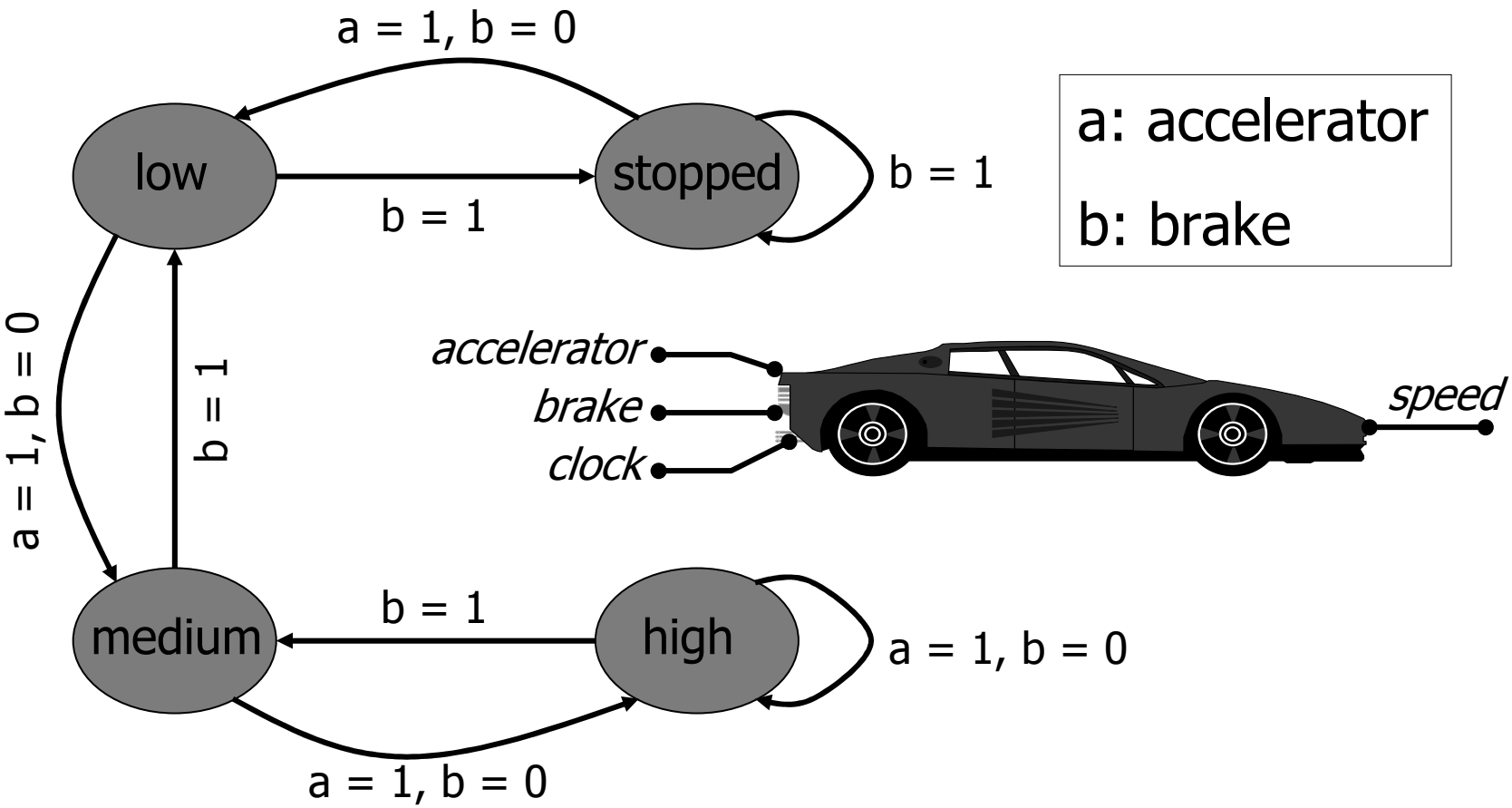
Explicit Style 2

```
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



Explicit Style

```
// 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
```

Implicit Style

```
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 <= `stopped;
            `medium:     speed <= `low;
             `high:      speed <= `medium;
            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

```
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)

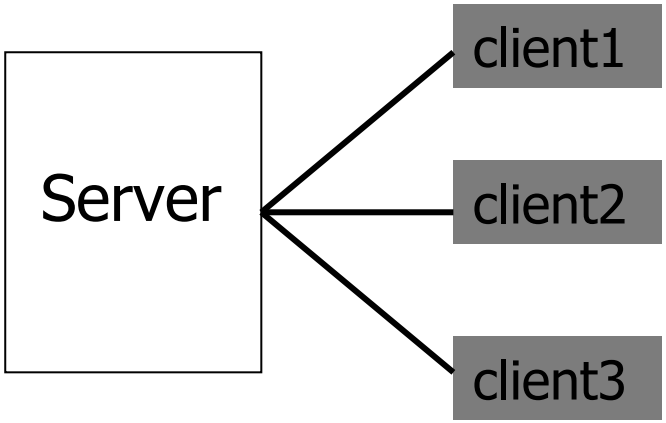
```
module Up_Down_Explicit (count, up_dwn, clock, reset_);
   output     [2:0]     count;
   input      [1:0]     up_dwn;
   input                clock, reset_;

   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
```

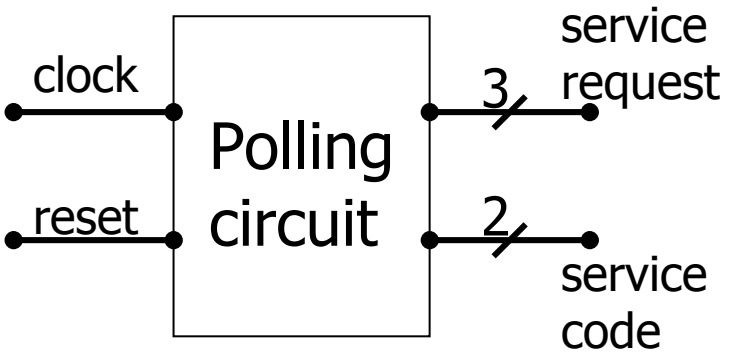
next_count = count;

Implicit Styles

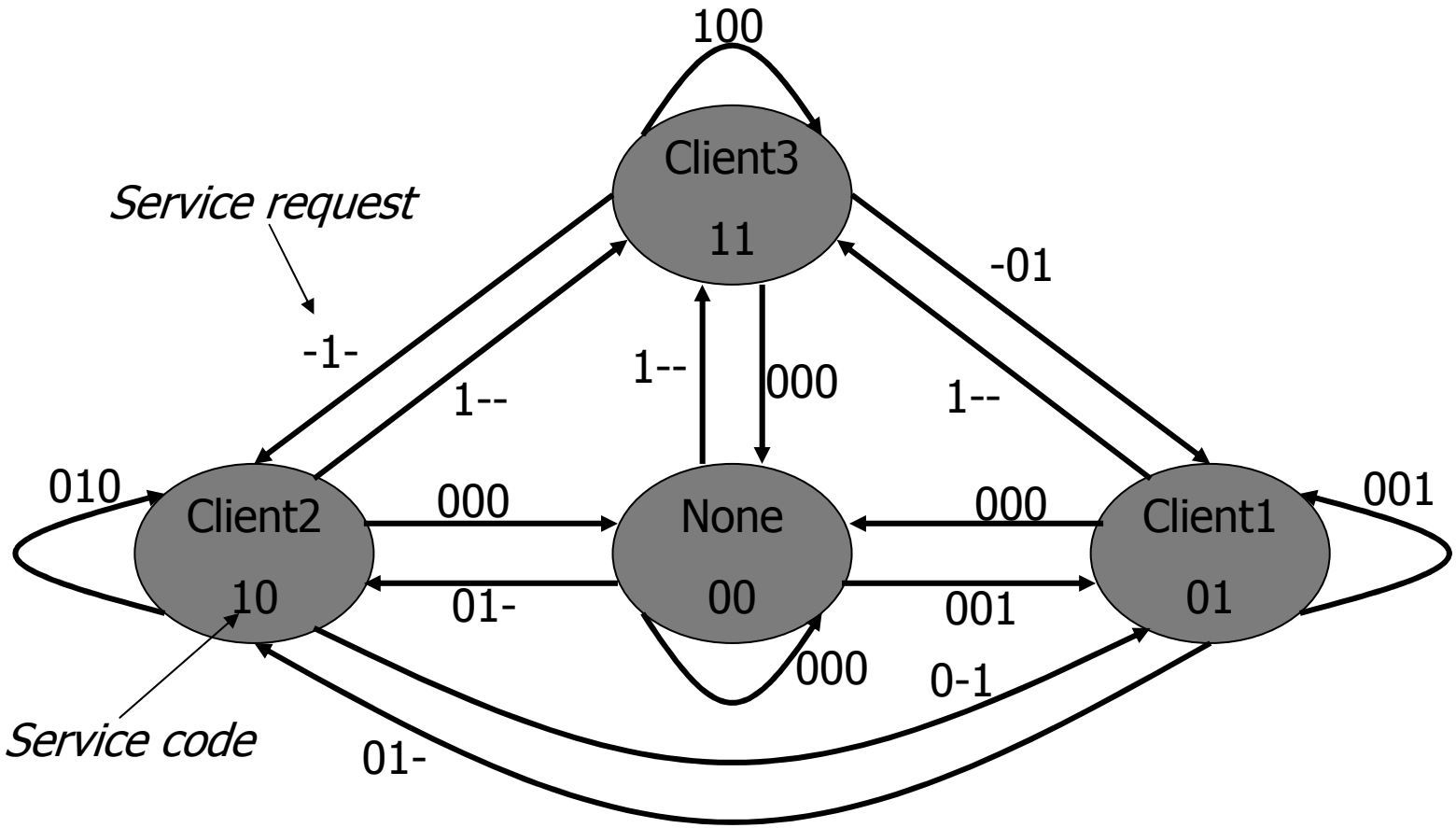
```
module Up_Down_Implicit1 (count, up_dwn, clock, reset_);
   output     [2:0]     count;
   input      [1:0]     up_dwn;
   input                clock, reset_;
   reg        [2:0]     count;

   always @ (negedge clock or negedge reset_)
      if (reset_ == 0) count = 3'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

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'b0; 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



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
```

```
   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
          if ((next_client == `none) &&
               (contender))
            next_client = contender; end
   endtask
endmodule
```