Simple Behavioral Model: the always block

always block

- *Always waiting for a change to a trigger signal*
- *Then executes the body*

```
module and_gate (out, in1, in2);
  input   in1, in2;
  output  out;
  reg     out;

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

Not a real register!!
A Verilog register
Needed because of assignment in always block

Specifies when block is executed
I.e., triggered by which signals

`always` Block

Procedure that describes the function of a circuit

- Can contain many statements including `if`, `for`, `while`, `case`
- Statements in the `always` block are executed *sequentially*

  (Continuous assignments <= are executed in *parallel*)
- Entire block is executed at once
- *Final* result describes the function of the circuit for current set of inputs

  intermediate assignments don't matter, only the final result

`begin/end` used to group statements

"Complete" Assignments

If an `always` block executes, and a variable is *not* assigned

- Variable keeps its old value (think implicit state!)
- *NOT* combinational logic ⇒ latch is inserted (implied memory)

This is usually *not* what you want: dangerous for the novice!

Any variable assigned in an `always` block should be assigned for any (and every!) execution of the block

Incomplete Triggers

Leaving out an input trigger usually results in a sequential circuit

Example: Output of this "and" gate depends on the input history

```
module and_gate (out, in1, in2);
  input       in1, in2;
  output      out;
  reg         out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

Verilog if

Same as C if statement

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2’b00) Y = A;
    else if (sel == 2’b01) Y = B;
    else if (sel == 2’b10) Y = C;
    else if (sel == 2’b11) Y = D;

endmodule
```

Verilog if

Another way

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;     // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;               // target of assignment

  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0) Y = A;
      else             Y = B;
    else
      if (sel[1] == 0) Y = C;
      else             Y = D;
endmodule
```

Verilog case

Sequential execution of cases

- Default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;       // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                 // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2’b00: Y = A;
      2’b01: Y = B;
      2’b10: Y = C;
      2’b11: Y = D;
    endcase
endmodule
```

Conditions tested in top to bottom order

Verilog case

- Without the default case, this example would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

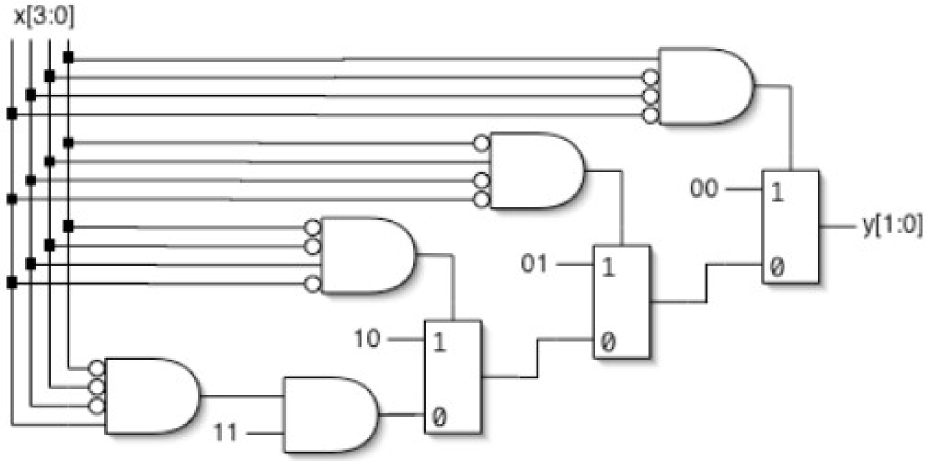
```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

  always @(A)
    case (A)
      8’b00000001: Y = 0;
      8’b00000010: Y = 1;
      8’b00000100: Y = 2;
      8’b00001000: Y = 3;
      8’b00010000: Y = 4;
      8’b00100000: Y = 5;
      8’b01000000: Y = 6;
      8’b10000000: Y = 7;
      default:     Y = 3’bX;      // Don’t care when input is not 1-hot
    endcase
endmodule
```

Encoder Example

- Nested IF-ELSE might lead to “priority logic”
  - Example: 4-to-2 encoder

```
always @(x)
begin
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

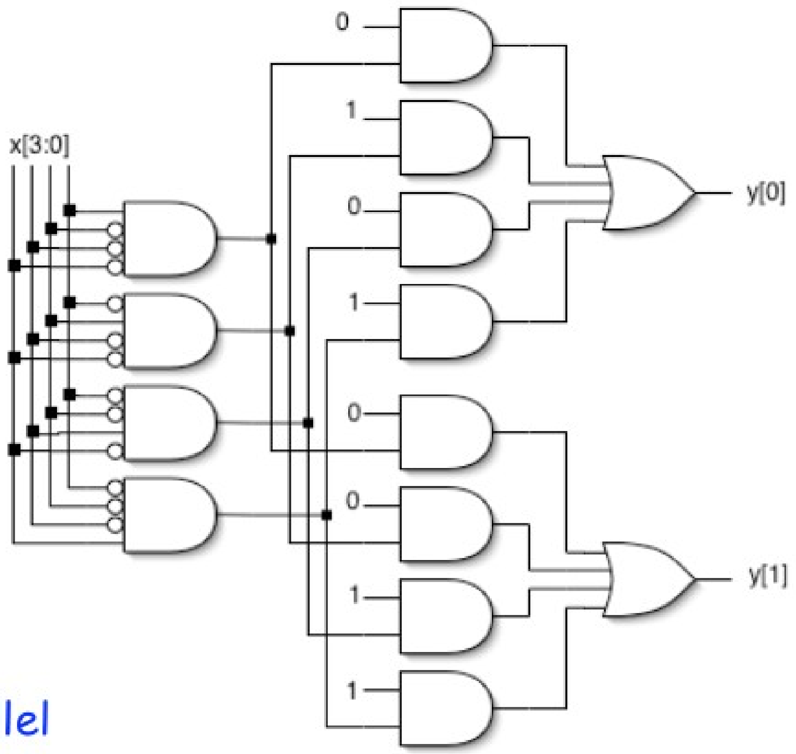
- This style of cascaded logic may adversely affect the performance of the circuit



Encoder Example (cont.)

- To avoid "priority logic" use the case construct:

```
always @(x)
begin
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```



- *All cases are matched in parallel*

Verilog case (cont)

Cases are executed sequentially

- Following implements a *priority* encoder

```
// Priority encoder
module encode (A, Y);
input  [7:0] A;          // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

  always @(A)
    case (1’b1)
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3’bX;  // Don’t care when input is all 0’s
    endcase
endmodule
```

Parallel Case

A priority encoder is more expensive than a simple encoder

- If we know the input is 1-hot, we can tell the synthesis tools
- **“parallel-case”** pragma (from "pragmatic") says the order of cases does not matter
- Pragmas or synthesis directives are specially formatted comments

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

  always @(A)
    case (1’b1)           // synthesis parallel-case
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3’bX;  // Don’t care when input is all 0’s
    endcase
endmodule
```

Verilog casex

- Like case, but cases can include 'X'
  - X bits not used when evaluating the cases
  - In other words, you don't care about those bits!

casex Example

```
// Priority encoder
module encode (A, valid, Y);
input  [7:0] A;                // 8-bit input vector
output [2:0] Y;                // 3-bit encoded output
output valid;                  // Asserted when an input is not all 0’s
reg    [2:0] Y;                // target of assignment
reg    valid;

  always @(A) begin
    valid = 1;
    casex (A)
      8’bXXXXXXX1: Y = 0;
      8’bXXXXXX10: Y = 1;
      8’bXXXXX100: Y = 2;
      8’bXXXX1000: Y = 3;
      8’bXXX10000: Y = 4;
      8’bXX100000: Y = 5;
      8’bX1000000: Y = 6;
      8’b10000000: Y = 7;
      default:  begin
         valid = 0;
         Y = 3’bX;  // Don’t care when input is all 0’s
      end
    endcase
  end
endmodule
```

Verilog for

for is similar to C

for statement is executed at compile time (like macro expansion)

```
// simple encoder
module encode (A, Y);
input  [7:0] A;            // 8-bit input vector
output [2:0] Y;            // 3-bit encoded output
reg    [2:0] Y;            // target of assignment

integer i;                 // Temporary variables for program only
reg [7:0] test;

  always @(A) begin
    test = 8b’00000001;
    Y = 3’bX;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = i;
       test = test << 1;
    end
  end
endmodule
```

for statements synthesize as cascaded combinational logic
⇒ Verilog unrolls the loop

Conway's Game of Life

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells,
each of which is in one of two possible states, alive or dead,
(or populated and unpopulated, respectively).

Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent.

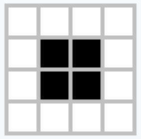
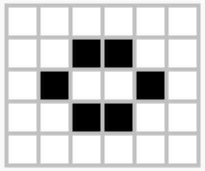
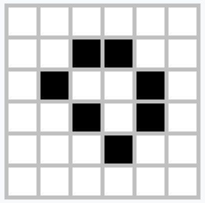
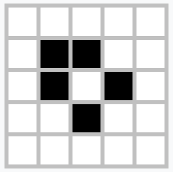
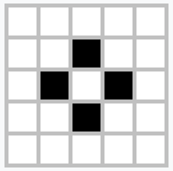
At each step in time, the following transitions occur:

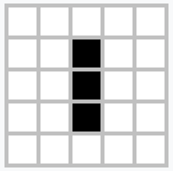
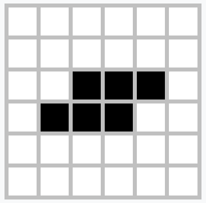
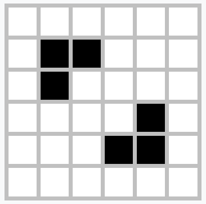
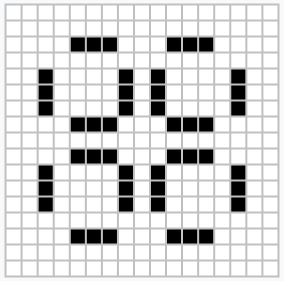
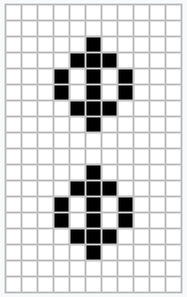
Any live cell with fewer than two live neighbours dies, as if by underpopulation.

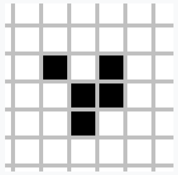
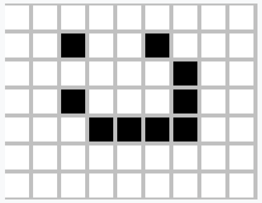
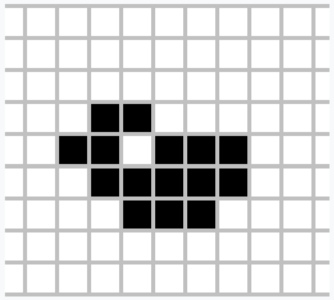
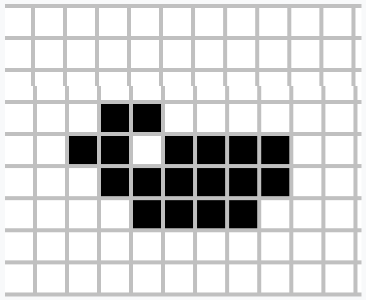
Any live cell with two or three live neighbours lives on to the next generation.

Any live cell with more than three live neighbours dies, as if by overpopulation.

Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Still lifes	
Block	
Bee-hive	
Loaf	
Boat	
Tub	

Oscillators	
Blinker (period 2)	
Toad (period 2)	
Beacon (period 2)	
Pulsar (period 3)	
Penta-decathlon (period 15)	

Spaceships	
Glider	
Light-weight spaceship (LWSS)	
Middle-weight spaceship (MWSS)	
Heavy-weight spaceship (HWSS)	

Another Behavioral Example

Computing Conway's Game of Life rule

```
module life (neighbours, self, out);
  input          self;
  input [7:0]    neighbours;
  output         out;
  reg            out;
  integer        count;
  integer        i;

  always @(neighbours or self) begin
    count = 0;
    for (i = 0; i<8; i = i+1) count = count + neighbours[i];
    Out = 0;
    out = out | (count == 3);
    out = out | ((self == 1) & (count == 2));
  end
endmodule
```

integers are temporary compiler variables

always block is executed instantaneously, if there are no delays only the final result is used

Verilog `while/repeat/forever` `(mainly for simulation)`

`while` (expression) statement

- Execute statement while expression is true

`repeat` (expression) statement

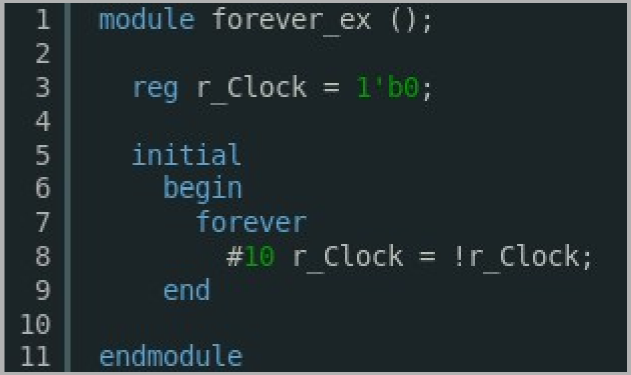
- Execute statement a fixed number of times

`forever` statement

- Execute statement forever

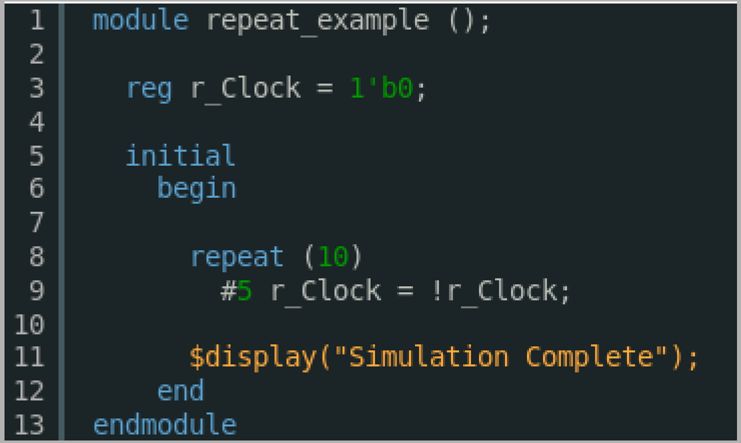
The keyword forever in Verilog creates a block of code that will run continuously.

Forever Loops should not be used in synthesizable code. They are intended for use in simulation test benches only.



```
module forever_ex ();

  reg r_Clock = 1'b0;

  initial
    begin
      forever
        #10 r_Clock = !r_Clock;
    end

endmodule
```

A repeat loop in Verilog will repeat a block of code some defined number of times.



```
module repeat_example ();

  reg r_Clock = 1'b0;

  initial
    begin

      repeat (10)
        #5 r_Clock = !r_Clock;

      $display("Simulation Complete");
    end
endmodule
```

It is very similar to a for loop, except that a repeat loop's index can never be used inside the loop. Repeat loops just blindly run the code as many times as you specify.

Repeat Loops can be used for synthesizable code, but be careful with them!.

They should only be used to expand replicated code

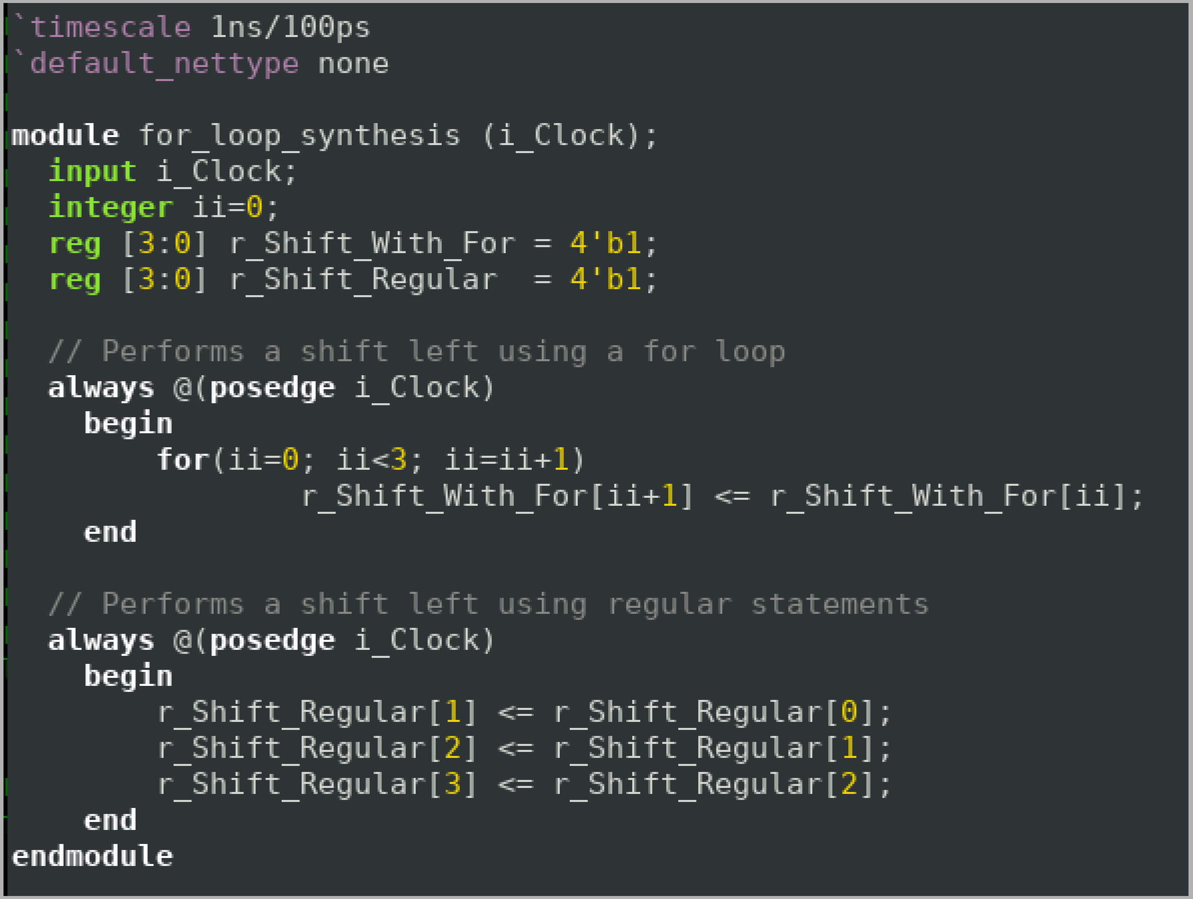
full-case and parallel-case

// synthesis parallel_case

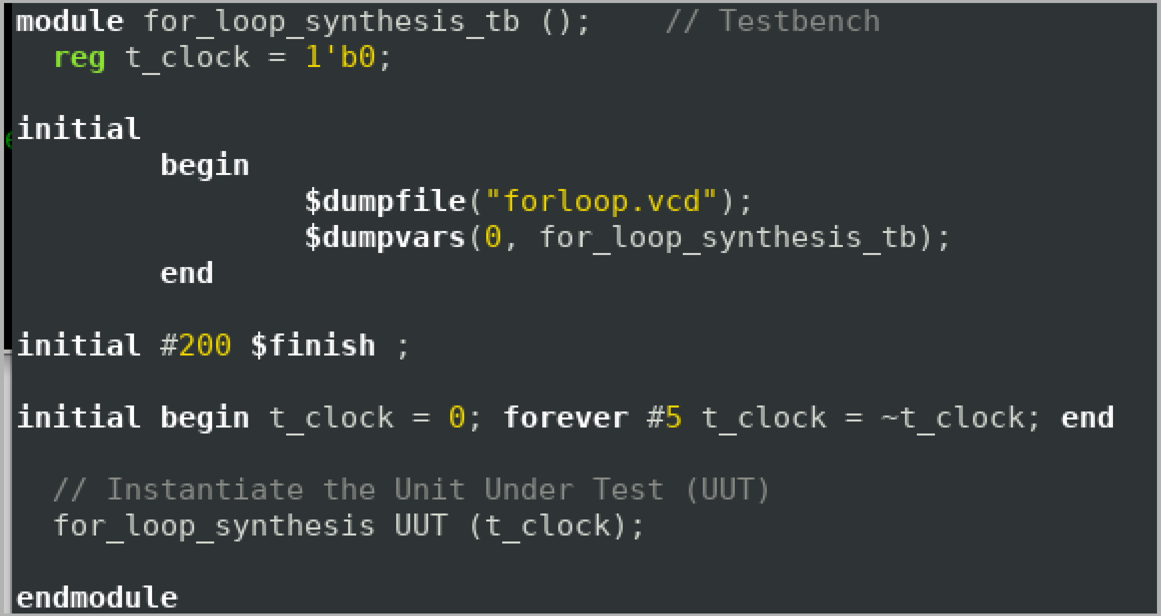
- Tells compiler that ordering of cases is not important
- That is, cases do not overlap
  - e. g., state machine - can't be in multiple states
- Gives cheaper implementation

// synthesis full_case

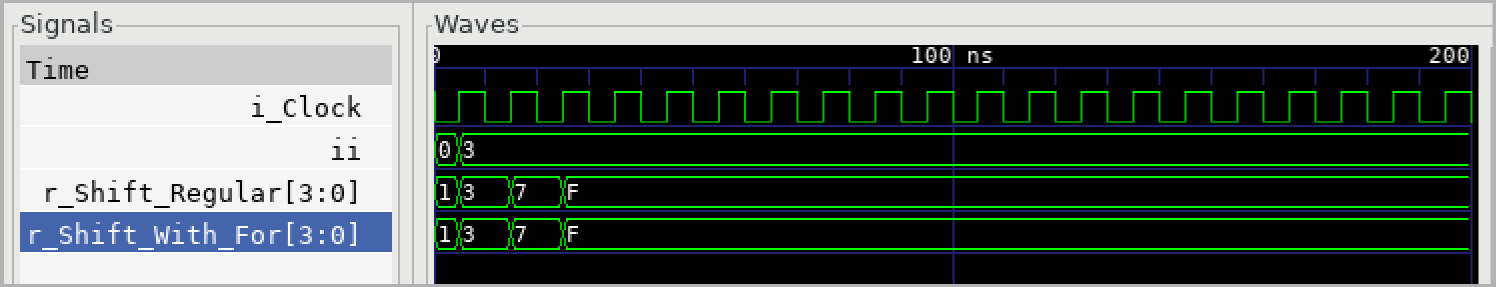
- Tells compiler that cases left out can be treated as don't cares
- Avoids incomplete specification and resulting latches



```
`timescale 1ns/100ps
`default_nettype none

module for_loop_synthesis (i_Clock);
  input i_Clock;
  integer ii=0;
  reg [3:0] r_Shift_With_For = 4'b1;
  reg [3:0] r_Shift_Regular  = 4'b1;

  // Performs a shift left using a for loop
  always @(posedge i_Clock)
    begin
        for(ii=0; ii<3; ii=ii+1)
                r_Shift_With_For[ii+1] <= r_Shift_With_For[ii];
    end

  // Performs a shift left using regular statements
  always @(posedge i_Clock)
    begin
        r_Shift_Regular[1] <= r_Shift_Regular[0];
        r_Shift_Regular[2] <= r_Shift_Regular[1];
        r_Shift_Regular[3] <= r_Shift_Regular[2];
    end
endmodule
```



```
module for_loop_synthesis_tb ();     // Testbench
  reg t_clock = 1'b0;

initial
        begin
                $dumpfile("forloop.vcd");
                $dumpvars(0, for_loop_synthesis_tb);
        end

initial #200 $finish ;

initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

  // Instantiate the Unit Under Test (UUT)
  for_loop_synthesis UUT (t_clock);

endmodule
```



Signals
Time
i_Clock
ii
r_Shift_Regular[3:0]
r_Shift_With_For[3:0]
Waves
0
100 ns
200
0
3
1
3
7
F
1
3
7
F