Bridge of Life Education

Verilog Critical Concepts Explained

7-8-2023

Jiin Lai

Topics

1. Scheduling Semantics - Blocking / Non-blocking
2. RTL Coding Styles That Yield Simulation and Synthesis Mismatches

Level of Proficiency in Verilog Design

1. Code matches simulation result (Test-bench + RTL)
2. Pre-synthesis (RTL) matches Post-synthesis (Gate-level)
3. Design Quality (PPA – Power/Performance/Area)
4. System/Application Level Optimization

IEEE Std 1364-1995 –Scheduling Semantics Blocking / Non-blocking

Two coding guidelines

1. Guideline: Use blocking assignments in always blocks that are written to generate combinational logic.

2. Guideline: Use nonblocking assignments in always blocks that are written to generate sequential logic.

Blocking Assignment

Execution of blocking assignments can be viewed as a one-step process:
1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

Nonblocking Assignment

Execution of nonblocking assignments can be viewed as a two-step process:

1. Evaluate the RHS of nonblocking statements at the beginning of the time step.
2. Update the LHS of nonblocking statements at the end of the time step.

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

Stratified Event Queue

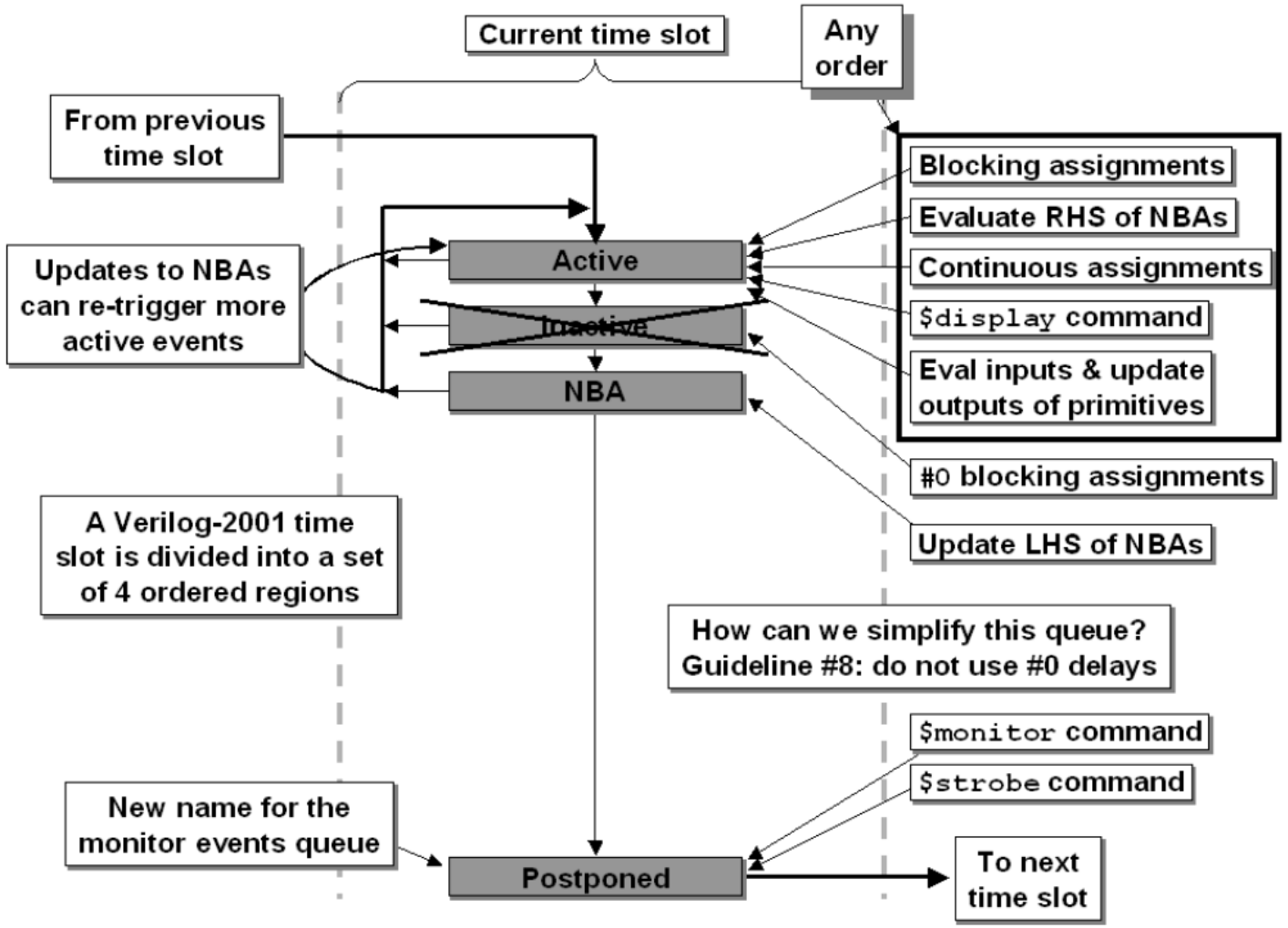
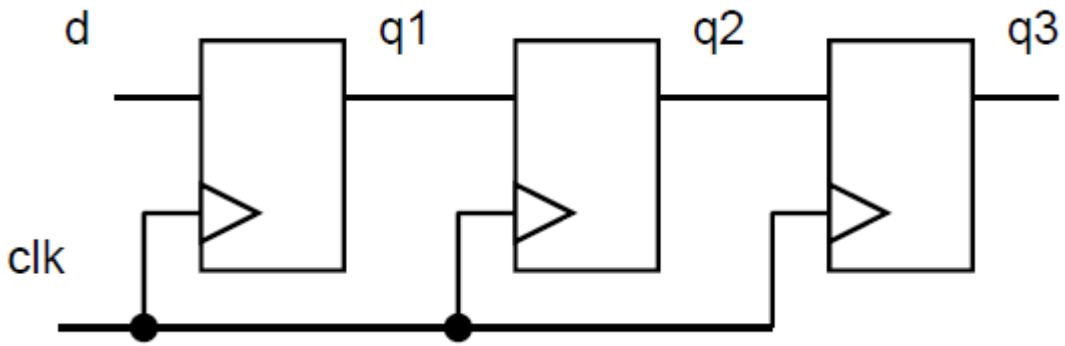


Figure 1 - Verilog "stratified event queue"

Coding Guidelines

1. When modeling sequential logic, use nonblocking assignments.
2. When modeling latches, use nonblocking assignments.
3. When modeling combinational logic with an always block, use blocking assignments.
4. When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
5. Do not mix blocking and nonblocking assignments in the same always block.
6. Do not make assignments to the same variable from more than one always block.
7. Use $strobe to display values that have been assigned using nonblocking assignments.
8. Do not make assignments using #0 delays.

Pipeline Modeling



Bad blocking-assignment – synthesize to one flip-flops

```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

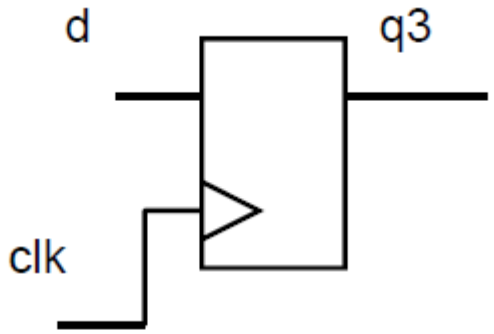


Figure 3 - Actual synthesized result!

Bad Blocking-assignment sequential coding – but it works

```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Bad blocking-assignment – synthesize correctly but simulation mismatch

```
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1=d;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;
endmodule
```

```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
  always @(posedge clk) q1=d;
endmodule
```

Using Nonblocking - 1

```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

Using Nonblocking - 2

```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

Using Nonblocking - 3

```
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1=d;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;
endmodule
```

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1<=d;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;
endmodule
```

Using Nonblocking - 4

```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
  always @(posedge clk) q1=d;
endmodule
```

```
module pipen4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;

  always @(posedge clk) q1<=d;
endmodule
```

Sequential Feedback – an LFSR example

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 = 1'b1;
      q2 = 1'b1;
      q1 = 1'b1;
    end
    else begin
      q3 = q2;
      q2 = n1;
      q1 = q3;
    end
endmodule
```

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

This works

LFSR – Non-blocking

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

```
module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

Guideline #1: When modeling sequential logic, use nonblocking assignments.
Guideline #2: When modeling latches, use nonblocking assignments.

Combinational Logic using Blocking Assignment

- What if using Non-Blocking assignment ?

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d or tmp1 or tmp2) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

Work but Multiple pass of always block

Efficient combinational logic coding using blocking assignment

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y    = tmp1 | tmp2;
  end
endmodule
```

Mixed Sequential & Combinational Logic – using Non-blocking

```
module nbex2 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= a ^ b;
endmodule
```

```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

  always @(a or b)
    y = a ^ b;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= y;
endmodule
```

Don’t Mixed Blocking & Non-Blocking

```
module ba_nba2 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q;

  always @(posedge clk or negedge rst_n) begin: ff
    reg tmp;
    if (!rst_n) q <= 1'b0;
    else begin
      tmp = a & b;
      q <= tmp;
    end
  end
endmodule
```

blocking / non-blocking assignment the same variable
- Error

```
module ba_nba6 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q, tmp;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q = 1'b0;  // blocking assignment to "q"
    else begin
      tmp = a & b;
      q <= tmp;            // nonblocking assignment to "q"
    end
endmodule
```

Multiple assignment to the same variable - Race

```
module badcode1 (q, d1, d2, clk, rst_n);
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d2;
endmodule
```

Nonblocking assignments are updated after $display

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial begin
    $strobe  ("\$strobe : a = %b", a);
    a = 0;
    a <= 1;
    $display ("\$display: a = %b", a);
    #1 $finish;
  end
endmodule
```

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

#0-delay assignment v.s. $strob/$monitor

```
module nb_schedule1;
  reg a, b;

  initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```

Multiple nonblocking assignment to the same variable
Nonblocking assignments shall be performed in the order the statements were executed.

```
initial begin
    a <= 0;
    a <= 1;
end
```

Case Study - 1

```
// Test bench
task tx;
  input value;

  data = value;
  @(posedge clk);
endtask

tx(8'h05);
tx(8'h06);
tx(8'h07);
```

data

clk

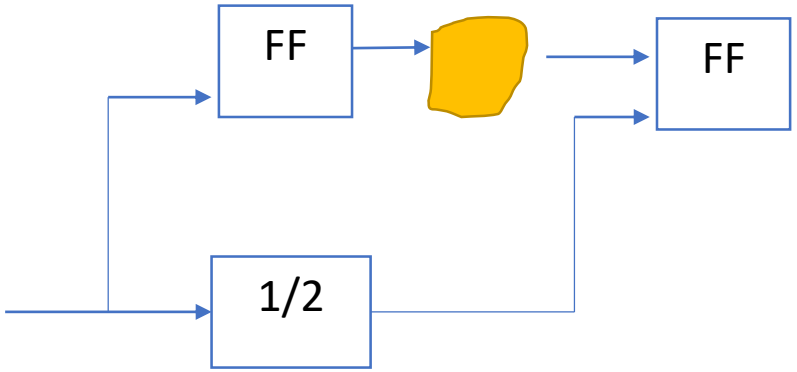
```
// Dut
reg  r_data;

@(posedge clk)
   r_data <= data;
```

data	05	06	07	
rdata		05	06	07
rdata		06	07	xx

desired

simulated

Case-2 : Generated clock




```
always @(posedge clk)
  if (!rstn) clk_divided2 <= 0;
  else       clk_divided2 <= ~clk_divided2;  // GOTCHA!
                                             // delay update to after delta

always @(posedge clk)
  if (!rstn) out1 <= 0;
  else       out1 <= in1;                    // delay update to after delta

always @(posedge clk_divided2)
  if (!rstn) out2 <= 0;
  else out2 <= out1;                         // race condition with out1
```

```
always @(posedge clk)
  if (!rstn) clk_divided2 = 0;
  else       clk_divided2 = ~clk_divided2;  // OK, immediate update


always @(posedge clk)
  if (!rstn) out1 <= 0;
  else       out1 <= in1;

always @(posedge clk_divided2)
  if (!rstn) out2 <= 0;
  else out2 <= out1;
```

RTL Coding Styles That Yield Simulation and Synthesis Mismatches

http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf

Incomplete sensitivity list

All synthesize a 2-input AND Gate

```
module code1a (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a or b)
    o = a & b;
endmodule
```

```
module code1b (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always @(a)
    o = a & b;
endmodule
```

```
module code1c (o, a, b);
  output o;
  input  a, b;
  reg    o;

  always
    o = a & b;
endmodule
```

Complete sensitivity list with mis-ordered assignment

```
module code2a (o, a, b, c, d);
  output o;
  input  a, b, c, d;
  reg    o, temp;

  always @(a or b or c or d) begin
    o    = a & b | temp;
    temp = c & d;
  end
endmodule
```

```
module code2b (o, a, b, c, d);
  output o;
  input  a, b, c, d;
  reg    o, temp;

  always @(a or b or c or d) begin
    temp = c & d;
    o    = a & b | temp;
  end
endmodule
```

Function

Function

- Functions always synthesize to combinational logic

latch

```
module code3a (o, a, nrst, en);
  output o;
  input  a, nrst, en;
  reg    o;

  always @(a or nrst or en)
    if      (!nrst) o = 1'b0;
    else if (en)    o = a;
endmodule
```

3-input and gate

```
// Infers a latch with asynchronous low-true
//   nrst and transparent high latch enable "en"

module code3b (o, a, nrst, en);
  output o;
  input  a, nrst, en;
  reg    o;

  always @(a or nrst or en)
    o = latch(a, nrst, en);

  function latch;
    input a, nrst, en;
    if      (!nrst) latch = 1'b0;
    else if (en)    latch = a;
  endfunction
endmodule
```

CASE – Full Case // synopsys full_case

- inform the synthesis tool that the case statement is fully defined, and that the output assignments for all unused cases are “don't cares”.

```
// no full_case
// Decoder built from four 3-input and gates
//   and two inverters
module code4a (y, a, en);
  output [3:0] y;
  input  [1:0] a;
  input        en;
  reg    [3:0] y;

  always @(a or en) begin
    y = 4'h0;
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

```
// full_case example
// Decoder built from four 2-input nor gates
//   and two inverters
// The enable input is dangling (has been optimized away)
module code4b (y, a, en);
  output [3:0] y;
  input  [1:0] a;
  input        en;
  reg    [3:0] y;

  always @(a or en) begin
    y = 4'h0;
    case ({en,a}) // synopsys full_case
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

Case – Parallel Case // synopsys parallel_case

- all cases should be tested in parallel, even if there are overlapping cases which would normally cause a priority encoder to be inferred.
- When a design does have overlapping cases, the functionality between pre- and post-synthesis designs will be different.

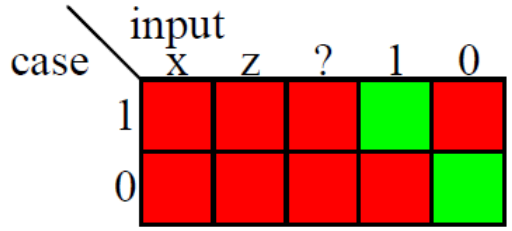
```
// no parallel_case
// Priority encoder - 2-input nand gate driving an
//   inverter (z-output) and also driving a
//   3-input and gate (y-output)
module code5a (y, z, a, b, c, d);
  output y, z;
  input  a, b, c, d;
  reg    y, z;

  always @(a or b or c or d) begin
    {y, z} = 2'b0;
    casez ({a, b, c, d})
      4'b11??: z = 1;
      4'b??11: y = 1;
    endcase
  end
endmodule
```

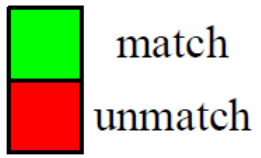
```
// parallel_case
// two parallel 2-input and gates
module code5b (y, z, a, b, c, d);
  output y, z;
  input  a, b, c, d;
  reg    y, z;

  always @(a or b or c or d) begin
    {y, z} = 2'b0;
    casez ({a, b, c, d}) // synopsys parallel_case
      4'b11??: z = 1;
      4'b??11: y = 1;
    endcase
  end
endmodule
```

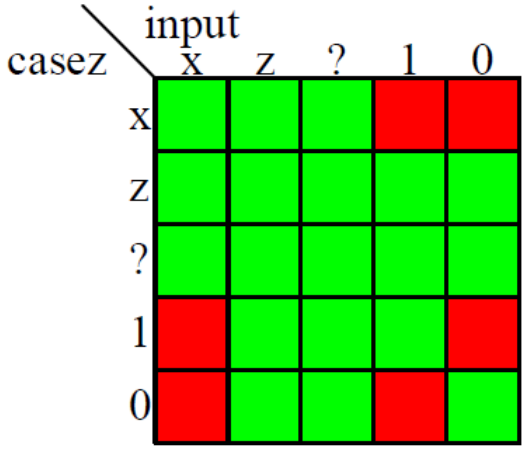
casex

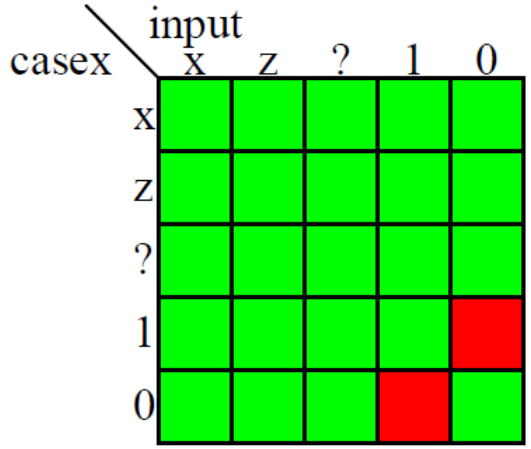
case, casez and casex

- case does not allow don't-care values (case: exact match (including x and z))
- casez allow both "z" and "?" values to be treated as don't-care values
- casex allows "z", "x" and "?" to be treated as don't-care values









casex – do not use casex for RTL coding

- When input is in unknown state, the pre-synthesis casex simulation will treat the unknown input as a "don't care". The equivalent post-synthesis simulation will propagate ‘X’s through the gate-level model, if that condition is tested.

```
module code6 (memce0, memce1, cs, en, addr);
  output          memce0, memce1, cs;
  input           en;
  input  [31:30] addr;
  reg             memce0, memce1, cs;

  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casex ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule
```

If MSB goes unknown while en is asserted

casez – use sparingly and cautiously for RTL coding

- Short, concise and tabular method for coding useful structures, e.g. priority encoders, address decoder …

```
module code7 (memce0, memce1, cs, en, addr);
  output          memce0, memce1, cs;
  input           en;
  input  [31:30] addr;
  reg             memce0, memce1, cs;

  always @(addr or en) begin
    {memce0, memce1, cs} = 3'b0;
    casez ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs     = 1'b1;
    endcase
  end
endmodule
```

Assigning ‘X’

- Synthesis interprets as a “don’t care”
- FSM design assign ‘X” to unused states to help debugging
  - Default next state to ‘X’ prior to entering the case statement

3-to-1 multiplexers where s is never 2’b11.
If s=2’b11 happens, x propagate and get noticed earlier

```
// Note: the second example synthesizes to a smaller
//   and faster implementation than the first example.

module code8a (y, a, b, c, s);
  output      y;
  input       a, b, c;
  input [1:0] s;
  reg         y;

  always @(a or b or c or s) begin
    y = 1'bx;
    case (s)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
    endcase
  end
endmodule
```

```
module code8b (y, a, b, c, s);
  output      y;
  input       a, b, c;
  input [1:0] s;
  reg         y;

  always @(a or b or c or s)
    case (s)
      2'b00:          y = a;

 2'b01:          y = b;
      2'b10, 2'b11: y = c;
    endcase
endmodule
```

Translate_off/translate_on

- Used to display information about a design, But dangerous to model functionality
- The sequence “assert reset, assert set, remove reset, leaving set still asserted.” causes mismatch

```
// Generally good DFF with asynchronous set and reset
module code10a (q, d, clk, rstn, setn);
  output q;
  input  d, clk, rstn, setn;
  reg    q;

  always @(posedge clk or negedge rstn or negedge setn)
    if      (!rstn)  q <= 0;  // asynchronous reset
    else if (!setn)  q <= 1;  // asynchronous set
    else             q <= d;
endmodule
```

```
// synopsys translate_off
// Bad DFF with asynchronous set and reset.  This design
//   will not compile from Synopsys, and the design will
//   not simulate correctly.
module code10b (q, d, clk, rstn, setn);
  output q;
  input  d, clk, rstn, setn;
  reg    q;

  always @(posedge clk or rstn or setn)
    if      (!rstn)  q <= 0;  // asynchronous reset
    else if (!setn)  q <= 1;  // asynchronous set
    else             q <= d;

endmodule
```

Use translate_on/translate_off to model set/reset flip-flop

```
// Good DFF with asynchronous set and reset and self-
//   correcting
// set-reset assignment
module code10c (q, d, clk, rstn, setn);
  output q;
input  d, clk, rstn, setn;
  reg    q;

  always @(posedge clk or negedge rstn or negedge setn)
    if       (!rstn)  q <= 0;  // asynchronous reset
    else if (!setn)  q <= 1;  // asynchronous set
    else             q <= d;

  // synopsys translate_off
    always @(rstn or setn)
      if   (rstn && !setn) force q = 1;
      else                 release q;
  // synopsys translate_on

endmodule
```

Timing Delays

- An always block not schedule events in zero time could miss triggered events.
- The actual design is two inverter one is #25 delay, the other #40 delay.
- But simulation if in change in 40 ns, then mismatch

```
module code11 (out1, out2, in);
  output out1, out2;
  input  in;
  reg    out1, out2;

  always @(in) begin
    #25 out1 = ~in;
    #40 out2 = ~in;
  end
endmodule
```