Digital Design with Verilog

Verilog

Lecture 10: Behavioral Modeling – Part 2

Learning Objectives

- Conditional Statements/Multiway Branching
  - if-else
  - case
- Loops
  - While, forever, for, repeat
- Miscellaneous
  - Synchronous/Asynchronous resets

Conditional Programming Constructs

- Procedural blocks provide the conditional programming constructs such as if-else decisions, case statements, and loops.

- These constructs are only available within a procedural block.

- They can be used to model both combinational and sequential logic.

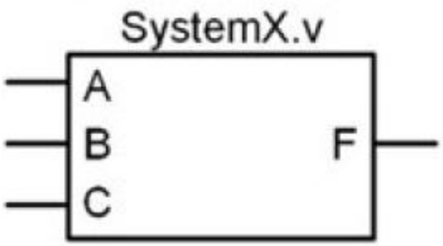
if-else Statements

- An if-else statement provides a way to make conditional signal assignments based on Boolean conditions.
- The if portion of statement is followed by a Boolean condition that if evaluated TRUE will cause the signal assignment listed after it to be performed.
- If the Boolean condition is evaluated FALSE, the statements listed after the else portion are executed.
- If multiple statements are to be executed in either the if or else portion, then the statement group keywords begin/end need to be used.

if-else Statements

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;
//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;
//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

if-else Statements



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX
  (output reg F,
   input  wire A, B, C);

  always @ (A, B, C)
    begin
      if      (A==1'b0 && B==1'b0 && C==1'b0)
        F = 1'b1;
      else if (A==1'b0 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else if (A==1'b1 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else
        F = 1'b0;
    end

endmodule
```

Multiway Branching

- The case statement
  - This compares 0,1,x and z of the condition, bit by bit with the different case options.

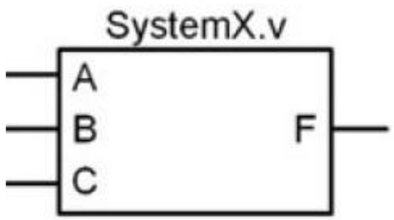
  - If width of condition and a case option are unequal, they are zero filled to match the bid width of the widest of both.

```
case (<input_name>)
    input_val_1 : statement_1
    input_val_2 : statement_2
                          :
    input_val_n : statement_n
    default     : default_statement
endcase
```

Multiway Branching

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
//Switch based on concatenation of control signals
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

Case Statement



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX
   (output reg F,
    input  wire A, B, C);

   always @ (A, B, C)
     begin
       case ( {A,B,C} )
            3'b000   : F = 1'b1;
            3'b001   : F = 1'b0;
            3'b010   : F = 1'b1;
            3'b011   : F = 1'b0;
            3'b100   : F = 1'b0;
            3'b101   : F = 1'b0;
            3'b110   : F = 1'b1;
            3'b111   : F = 1'b0;
            default : F = 1'bX;
       endcase
     end

endmodule
```

```
case ( {A,B,C} )
   3'b000   : F = 1'b1;
   3'b010   : F = 1'b1;
   3'b110   : F = 1'b1;
   default : F = 1'b0;
endcase
```

```
case ( {A,B,C} )
    3'b000, 3'b010, 3'b111  : F = 1'b1;
    default                 : F = 1'b0;
endcase
```

The casex and casez statements

- casez does not compare z-values in the condition and case options. All bit positions with z may be represented as ? in that position.

- casex does not compare both x and z-values in the condition and case options.

casez

- Wildcard Case Statement: casez

- **casez** allows “Z” and “?” to be treated as don’t care values in either the *case expression* and/or the *case item* when doing case comparison.

```
always @(irq) begin
  {int2, int1, int0} = 3'b000;
  casez (irq)
    3'b1?? : int2 = 1'b1;
    3'b?1? : int1 = 1'b1;
    3'b??1 : int0 = 1'b1;
    default: {int2, int1, int0} = 3'b000;
  endcase
end
```

casex

- Even wilder: casex

- **casex** allows “Z”, “?”, and “X” to be treated as don’t care values in either the *case expression* and/or the *case item* when doing case comparison.

```
reg [3:0] encoding;
integer state;
casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

Loops

Loops

- A loop within Verilog provides a mechanism to perform repetitive assignments infinitely.

- This is useful in test benches for creating stimulus such as clocks or other periodic signals.

- There are four types of looping statements in Verilog:

    while,

    for,

    repeat, and

    forever.

While

- A while loop provides a looping structure with a Boolean condition that controls its execution.

- The loop will only execute if the Boolean condition is evaluated true.

- Syntax:

```
while (<boolean_condition>)
   begin
    statement_1
    statement_2
             :
    statement_n
   end
```

While

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;
initial
begin
count = 0;
while (count < 128) //Execute loop till count is 127.
//exit at count 128
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

For loop

- The loop will execute if a Boolean condition associated with the loop variable is TRUE.

- Syntax:

```
for (<initial_assignment>; <Boolean_condition>; <step_assignment>)
  begin
   statement_1
   statement_2
         :
   statement_n
  end
```

For loop

```
integer count;
initial
for ( count=0; count < 128; count = count + 1)
$display("Count = %d", count);

//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1]; //Integer array state with elements
0:31
integer i;
initial
begin
for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
state[i] = 0;
for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
state[i] = 1;
end
```

Repeat Loop

- A repeat loop provides a looping structure that will execute a fixed number of times.

- Syntax:

```
repeat (<number_of_loops>)
  begin
    statement_1
    statement_2
         :
    statement_n
  end
```

Repeat Loop

```
//Illustration 1 : increment and display count from 0 to 127
integer count;
initial
begin
count = 0;
repeat(128)
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

Forever loop

- A forever loop within an initial block provides identical behavior as an always loop without a sensitivity loop.

- Syntax:

```
forever
  begin
    statement_1
    statement_2
         :
    statement_n
  end
```

Forever loop

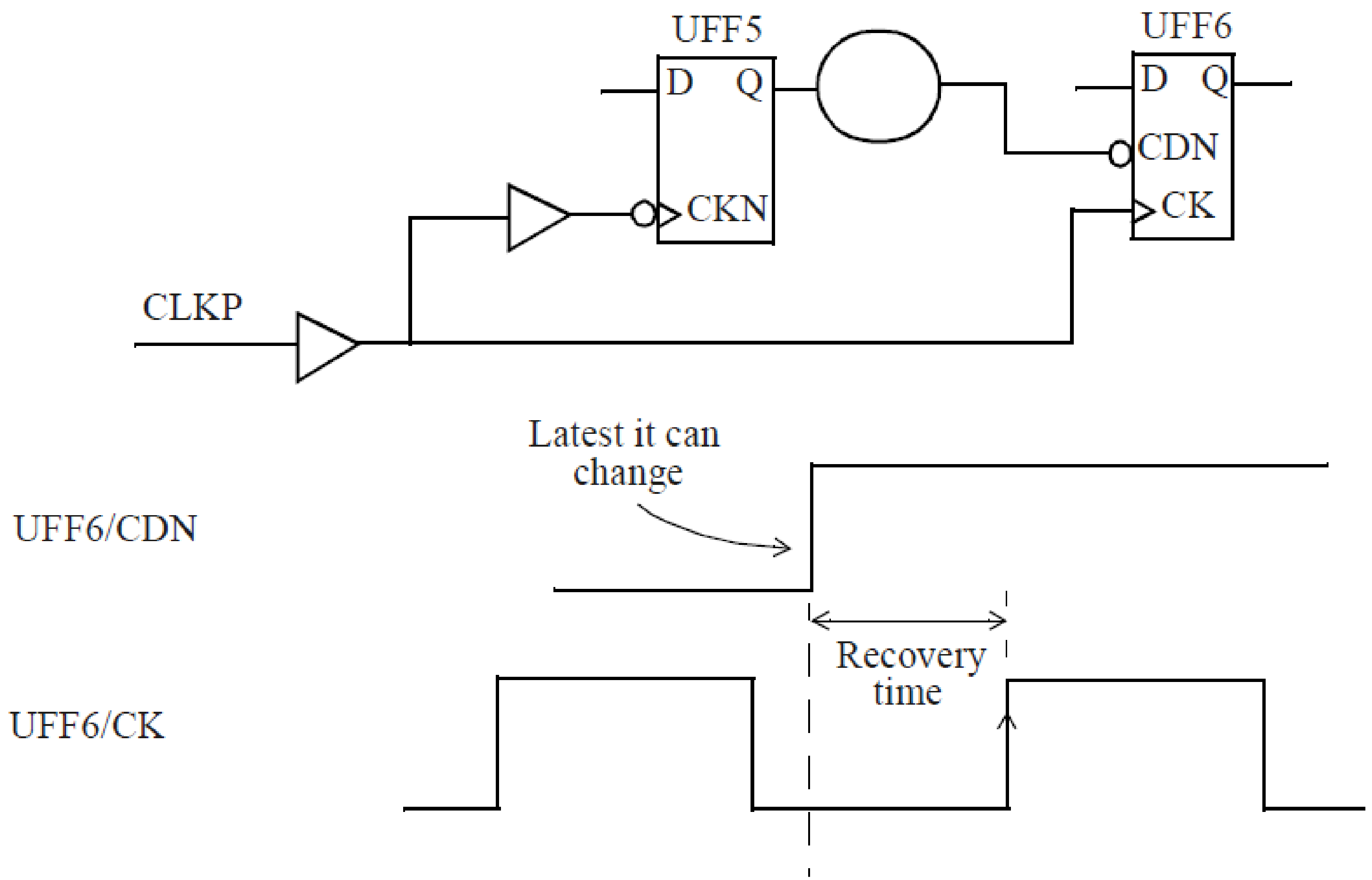
```
//Example 1: Clock generation
//Use forever loop instead of always block
reg clock;
initial
begin
clock = 1'b0;
forever #10 clock = ~clock; //Clock with period of 20 units
end
//Example 2: Synchronize two register values at every positive edge of
//clock
reg clock;
reg x, y;
initial
forever @(posedge clock) x = y;
```

Miscellaneous

Recovery Timing Check

- A **recovery timing check** ensures
  - that there is a minimum amount of time between the asynchronous signal becoming inactive and the next active clock edge.
- In other words, this check ensures that after the asynchronous signal becomes inactive, there is adequate time to recover so that the next active clock edge can be effective.

Recovery Timing Check



Recovery Timing Check

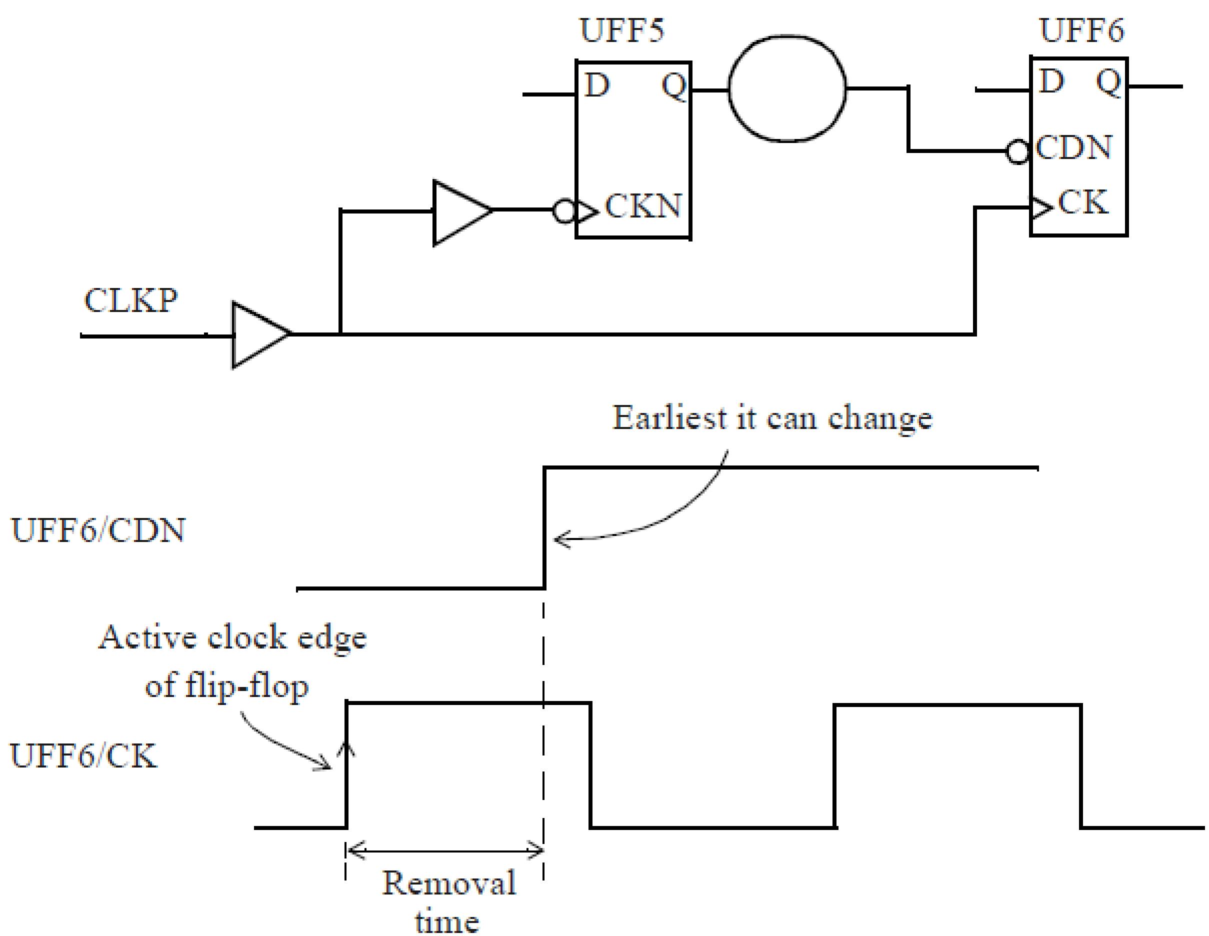
Removal Timing Check

- A **removal timing check** ensures
  - that there is adequate time between an active clock edge and the release of an asynchronous control signal.

- The check ensures
  - that the active clock edge has no effect because the asynchronous control signal remains active until *removal time* after the active clock edge.

- In other words, the asynchronous control signal is released (becomes inactive) well after the active clock edge so that the clock edge can have no effect.

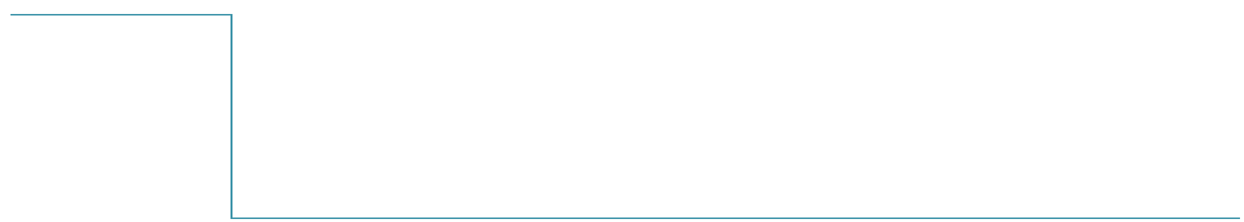
Removal Timing Check



Removal timing check.

Synchronous Reset / Asynchronous Reset

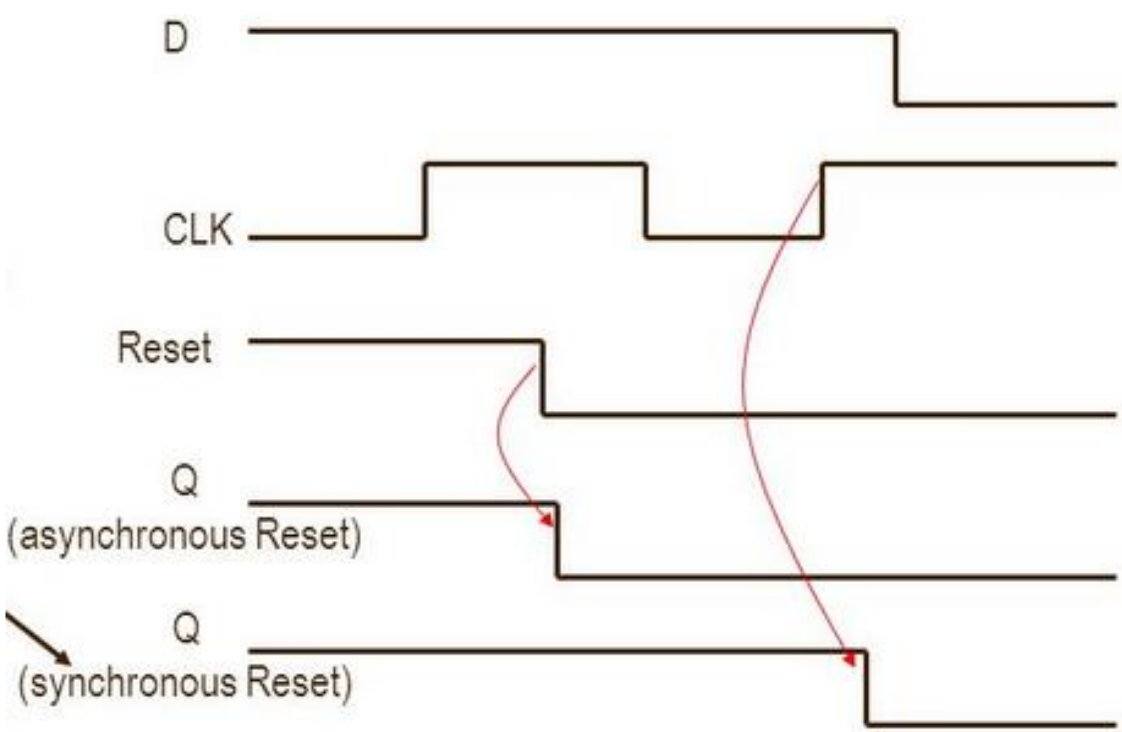
- Why is reset used ?
  - Reset is used to initialize the hardware design of a system and to make system to known state from unknown state.



Exemplary Reset Signal

Synchronous Reset / Asynchronous Reset

- Synchronous reset : A synchronous reset will reset the circuit at active edge of clock.
- Asynchronous reset : Asynchronous reset will reset the circuit asynchronously i.e. no matter with the clock.



Synchronous Reset – D Flip-Flop

```
module dff_sync_reset (
data    , // Data Input
clk     , // Clock Input
reset   , // Reset input
q         // Q output
);
//-----------Input Ports---------------
input data, clk, reset ;

//-----------Output Ports---------------
output q;

//------------Internal Variables--------
reg q;

//-------------Code Starts Here---------
always @ ( posedge clk)
if (~reset) begin
  q <= 1'b0;
end  else begin
  q <= data;
end

endmodule //End Of Module dff_sync_reset
```

Asynchronous Reset – D Flip-Flop

```
module dff_async_reset (
data   , // Data Input
clk     , // Clock Input
reset , // Reset input
q            // Q output
);
//-----------Input Ports---------------
input data, clk, reset ;

//-----------Output Ports---------------
output q;

//------------Internal Variables--------
reg q;

//-------------Code Starts Here---------
always @ ( posedge clk or negedge reset)
if (~reset) begin
  q <= 1'b0;
end  else begin
  q <= data;
end

endmodule //End Of Module dff_async_reset
```

References

- Chapter 7, Verilog HDL by Samir Palnitkar, Second Edition.

- Disclaimer: “I don’t claim the ownership of all the slides, some of the material is picked up from various publicly available sources on the internet”.

Thank you