Lecture 10: RTL Programming Statements

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

Some slides are courtesy of Prof. Lin or Prof. Karna ECE111 courses

Important announcements (Details in Canvas)

- **This class on Tues 10/29 will be virtual – posted online (Not in person)**
- **Class on Thurs 10/31 will be in person as usual so please be here in the class**
- **Prof office hours next week:** Thurs 10/31 from 12-2pm or by email appointment
- **TA office hours:** are now MWF 9-11am for Fall'24
  - Zoom Meeting ID: 948 6397 0932; Passcode 004453
  - https://ucsd.zoom.us/j/94863970932?pwd=iXcQXbLYjOaAQTdOJlrmglCYMSyeir.1
- **TA Discussion session:** Wed 4-4:50PM (in person) Location: CNTRE 214
- **Oct 29:** Homework 5 will be posted on Canvas
  - Due on Wed Nov 6, **11/06/24**
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 4 overview

- Design of a synthesizable SystemVerilog model of a Carry Look Ahead Adder, and a synthesizable SystemVerilog code for a Booth Multiplier.
- You will learn how to:
  - Create synthesizable SystemVerilog code
  - Better learn how to use testbenches
  - Design functional SystemVerilog code that can compile post synthesis.
- There will be two parts for this homework:
  - **Homework-5a**: Developing a Synthesizable SystemVerilog Model for a of an N-bit Carry Look Ahead Adder
  - **Homework-5b**: Developing a synthesizable SystemVerilog model of an N-bit Booth Multiplier.

Recap

Brief Summary on initial and always Procedural Blocks

Initial Procedural Blocks

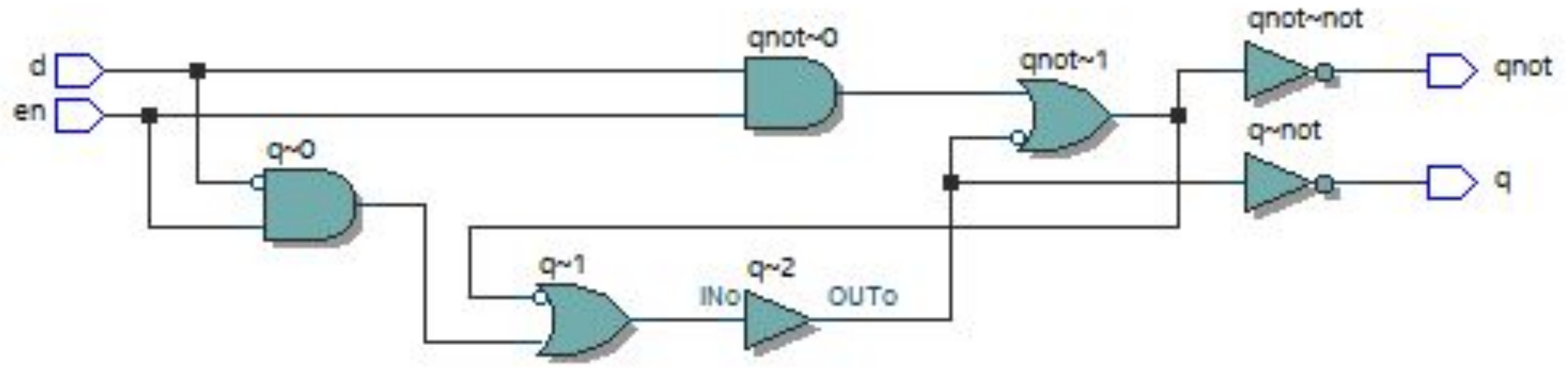
- **Not synthesizable** and does not generate hardware logic
- Used for **simulation purpose**
- Starts **execution from beginning** of a simulation at time 0
- **Executes only once** during simulation and it terminates when all statements within it are executed
- **Any number** of initial blocks can be defined within a module
- **Multiple** initial blocks executes **concurrently**

Always Procedural Blocks

- **Synthesizable** and it can generate hardware logic
- Used for **specifying design behavior** (RTL code)
- Starts **execution from beginning** of a simulation at time 0
- **Executes repeatedly**. Its activity shall cease only when the simulation is terminated
- **Any number** of always blocks can be defined within a module
- **Multiple** always blocks executes **concurrently**

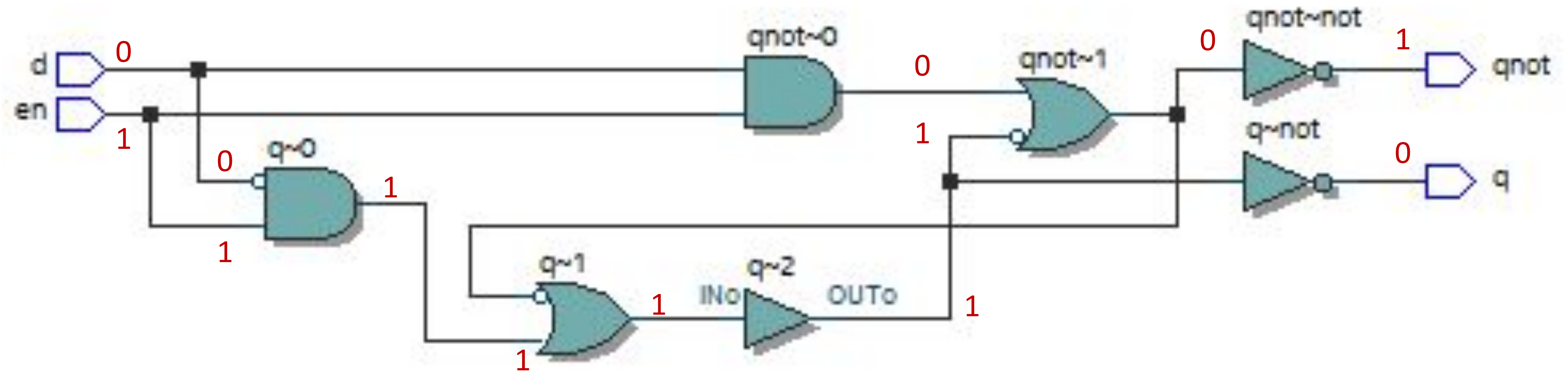
Can always_comb model a latch

```
module latch_comb(
 input logic [3:0] d, en,
 output logic q, qbar);
 always_comb begin
    q <= ~(qbar | (en & ~d);
    qbar <= ~(q | (en &d));
 end
endmodule
```



Can always_comb model a latch

```
module latch_comb(
 input logic [3:0] d, en,
 output logic q, qbar);
 always_comb begin
    q <= ~(qbar | (en & ~d);
    qbar <= ~(q | (en &d));
 end
endmodule
```

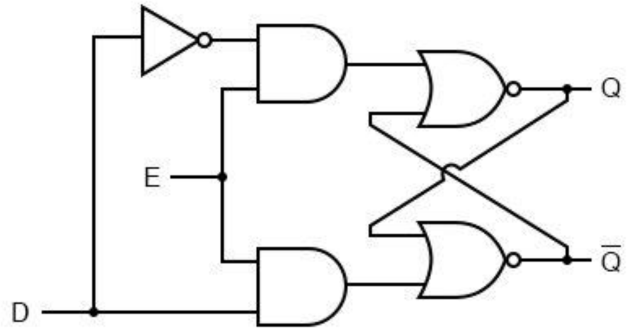


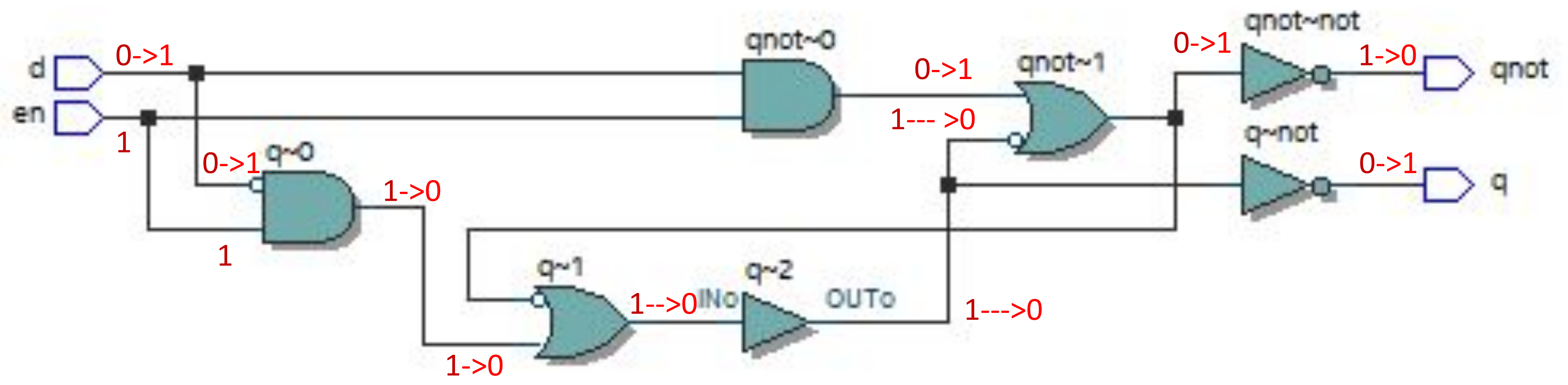
Can always_comb model a latch

```
module latch_comb(
 input logic [3:0] d, en,
 output logic q, qbar);
 always_comb begin
    q <= ~(qbar | (en & ~d);
    qbar <= ~(q | (en &d));
 end
endmodule
```

What's the problem here?

How do they compare?





RTL Programming Statements

SystemVerilog RTL Programming Statements Summary

Category	RTL Programming Statement	Synthesizable	Usage
Decision Statements	if/else	Synthesizable	RTL Design and Testbench code
	case (case, case/inside, casex, casez, unique/priority case)	Synthesizable	RTL Design and Testbench code
	generate if/else, generate case	Synthesizable	RTL Design and Testbench code
Looping Statements	for	Synthesizable	RTL Design and Testbench code
	repeat	Synthesizable	RTL Design and Testbench code
	while	Non-Synthesizable	Testbench code
	do/while	Non-Synthesizable	Testbench code
	foreach	Non-Synthesizable	Testbench code
	forever	Non-Synthesizable	Testbench code
	generate for	Synthesizable	RTL Design and Testbench code
Jump Statements	continue	Synthesizable	Mostly used in Testbench code
	break	Synthesizable	Mostly used in Testbench code
	disable	Non-Synthesizable	Testbench code

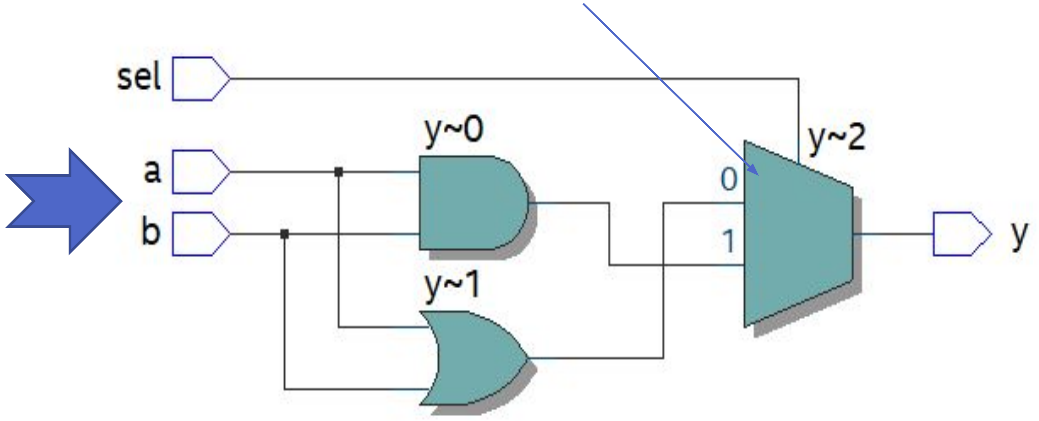
Decision Statements

if/else Conditional Statement

- If/else statement evaluates an expression and executes one of the two possible branches
  - If expression is True ('1'), then all statements within if branch will be executed
  - If expression is False('0', 'X' or 'Z'), then all statements within else branch will be executed
  - Multiple statements can be specified within true and false branch

```
module ex1(
    input logic a, b, sel,
    output logic y);


always@(a,b, sel) begin
    if(sel == 1)
        y = a & b;  // statement executed if sel is '1'
    else
        y = a | b;  // statement executed if sel is either '0', 'X' or 'Z'
end
endmodule: ex1
```

Synthesizer infers a Mux for if/else conditional statement in combinational logic

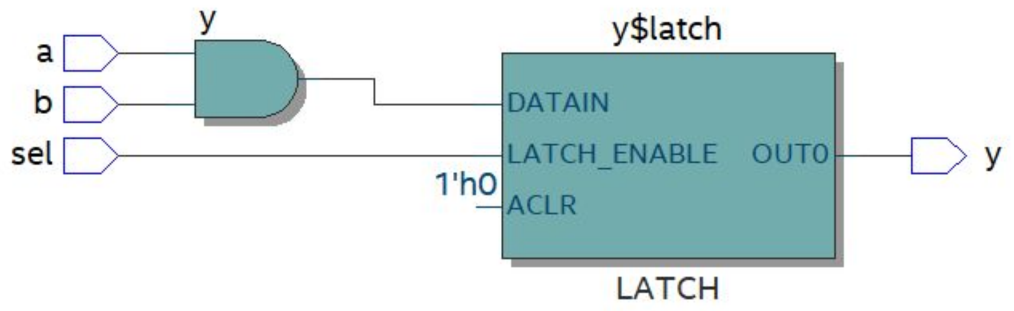


if/else Conditional Statement

- If there is no else branch, then a latch will be inferred to retain previous value

```
module ex2(
 input logic a, b, sel,
 output logic y);

 always@(a,b,sel) begin
  if(sel == 1)
    y = a & b;
  end
endmodule: ex2
```

Synthesizer infers a Latch due to missing else branch to retain previous value of 'y' when sel is '0' or 'x'

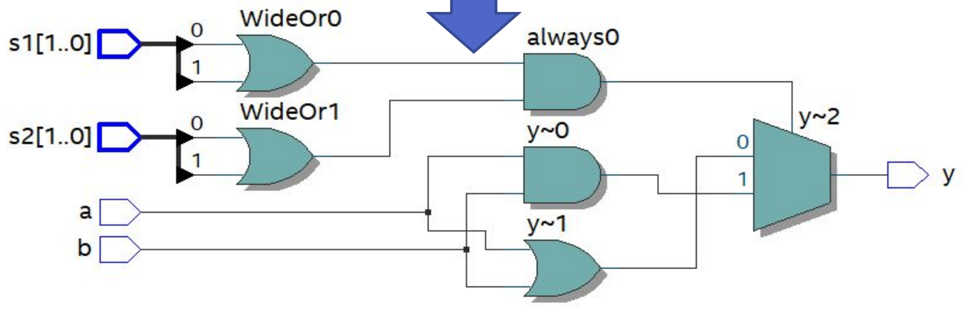


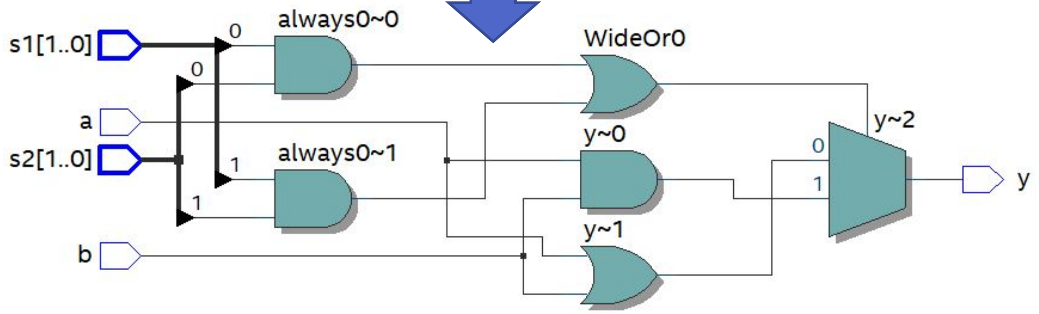
if/else Conditional Statement

- Using **logical versus bitwise** operators in if condition expression can result in **different circuits**
  - Only use 1-bit vectors or use logical operators in if condition expression to return true/false
  - Do not perform true/false on vectors

```
module ex3(
 input logic a, b,
 input logic[1:0] s1, s2,
 output logic y);

 always@(a,b) begin
   if(s1 && s2)          // Use logical operators in if
     y = a & b;          condition expression
   else
     y = a | b;
  end
endmodule: ex3
```

```
module ex4(
 input logic a, b,
 input logic[1:0] s1, s2,
 output logic y);

 always@(a,b) begin
   if(s1 & s2)           // using bitwise operator can lead
     y = a & b;          into design bugs if any bit is either
   else                  'X' or 'Z' and it will cause else branch to
     y = a | b;          execute. Results in mismatch between
  end                    simulation and synthesis behavior
endmodule: ex4
```





case Conditional Statement

- **Case statement** provides a concise way to represent series of decisions choices
- SystemVerilog case has implied "break" statement
- Used for developing **mux, decoder, encoders, next state logic** in FSM
- case items are **not-necessarily non-overlapping**
- Syntax:

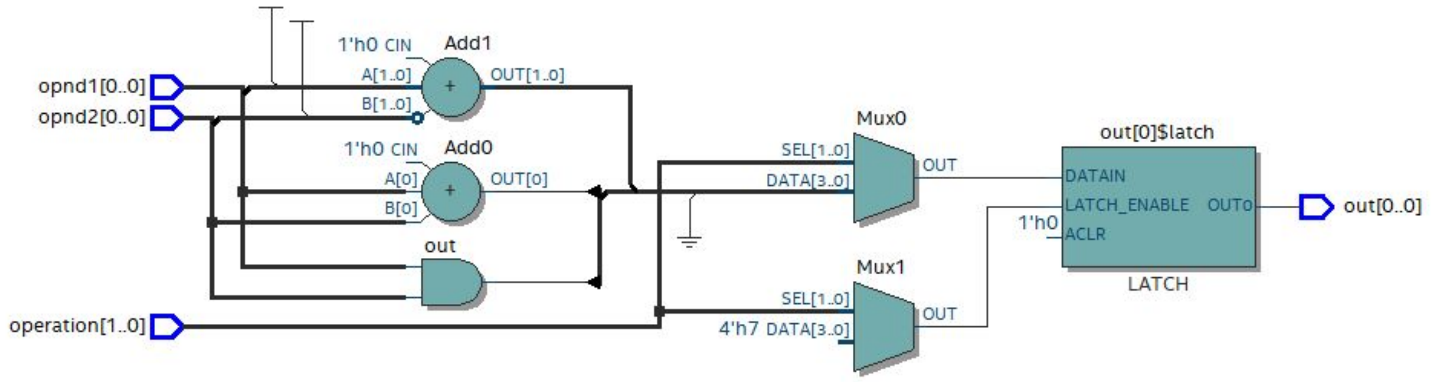
```
case(case_expression)
        case_item1 : <statement1>;
        case_item2 : begin
          statement2a;
          statement2b;
        end
      default : case_item_statement5;
   endcase
```

```
module alu #(parameter N=1) (
  input logic[N-1:0] opnd1, opnd2,
  input logic[1:0] operation,
  output logic[N-1:0] out);

  always_comb
  begin
    case(operation)
      2'b00: out = opnd1 + opnd2;
      2'b01: out = opnd1 - opnd2;
      2'b10: out = opnd1 & opnd2;
      2'b11: out = opnd1 | opnd2;
      default: out ='X;
    endcase
  end
endmodule: alu
```

case Statement – Incomplete case

- Incomplete case items will result in a latch inference

```
module alu #(parameter N=1) (
  input logic[N-1:0] opnd1, opnd2,
  input logic[1:0] operation,
  output logic[N-1:0] out);

  always@(operation, opnd1, opnd2)
  begin
    case(operation)
      2'b00: out = opnd1 + opnd2;
      2'b01: out = opnd1 - opnd2;
      2'b10: out = opnd1 & opnd2;
    endcase
  end
endmodule: alu
```

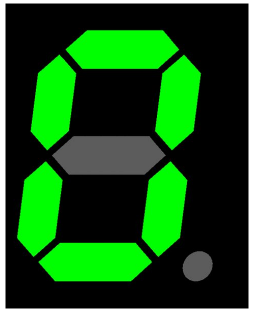


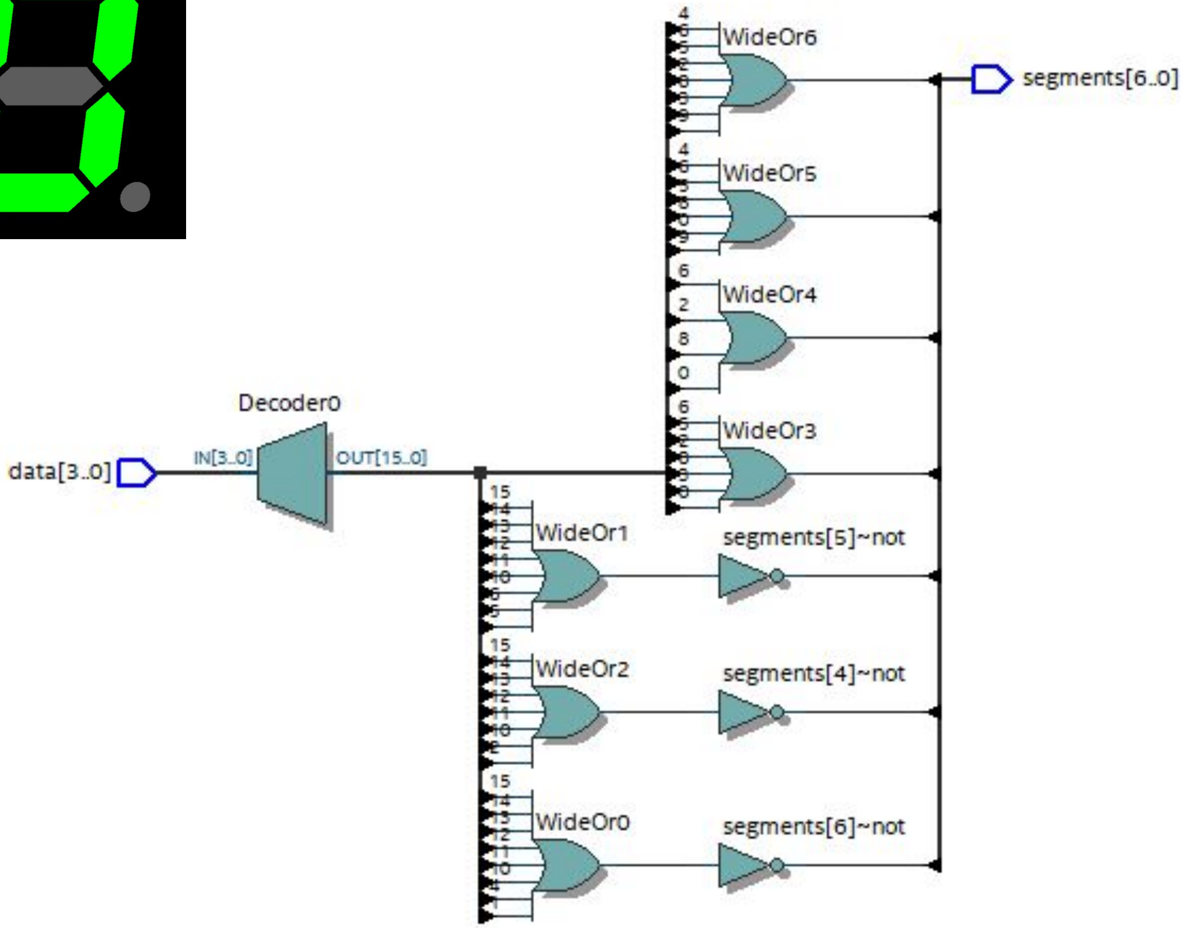
Synthesis tool inferred a latch due to missing case items for "operation" values 2'b11 or missing default statement.

Note : if always_comb is used, synthesizer will generate compile time error to let designer know that latch will be inferred

7-Segment LED Display using case Statement

```
module seven_seg(
 input  logic [3:0] data,
 output logic [6:0] segments);
  always_comb
    case (data)
     0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       7'b111_1001;
      4: segments =       7'b011_0011;
      5: segments =       7'b101_1011;
      6: segments =       7'b101_1111;
      7: segments =       7'b111_0000;
      8: segments =       7'b111_1111;
      9: segments =       7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```





https://commons.wikimedia.org/

case Inside Statement

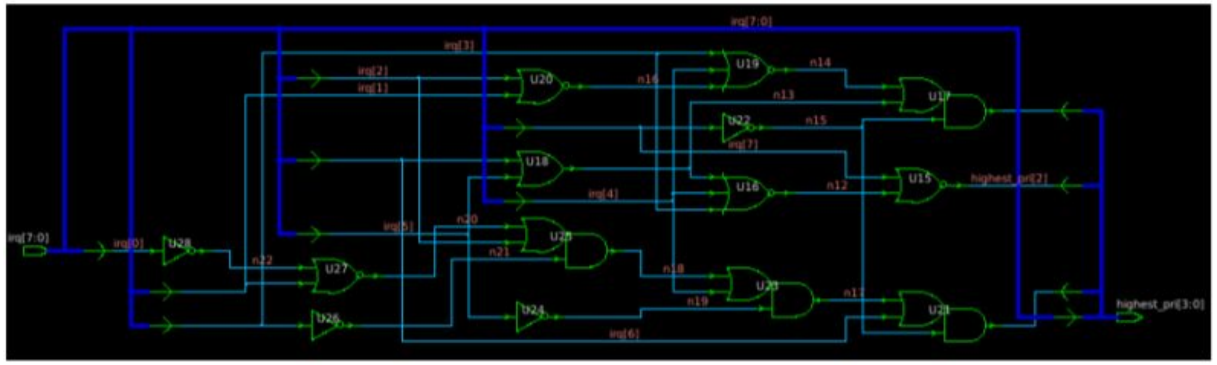
- Case inside statement is similar to case statement except it uses ==? wildcard case equality operator
  - Any bit in a case item that is set to X or Z or ?, that bit is ignored when case expression is compared with the case item

```
module case_inside(
 input logic [3:0] sel,
 output logic [3:0] y);
 always_comb begin
  case(sel) inside
    4'b1???: y = 4'b1000;  // this branch will be selected if “sel” MSB is ‘1’, regardless of any values in rest of the “sel” bits
    4'b01??: y = 4'b0100; // this branch will be selected if “sel” MSB is ‘01’, regardless of any values in rest of the “sel” bits
    4’b011?: y = 4’b1100;
    4'b001?: y = 4'b0010; // this branch will be selected if “sel” MSB is ‘001’, regardless of any values in rest of the “sel” bits
    4'b0001: y = 4'b0001;
    default: y = 4'b0000;
   endcase
  end
endmodule
```

casez Statement

- With a casez statement, any case item bits that are specified with the characters z, Z or ? are treated as don't care bits
  - For example: 2b1? can match the case expressions: 2b10, 2b11, or 2b1z
- casez has overlapping case items. If more than one case item matches a case expression, the first matching case item has priority

```
module priority_encoder_casez (
  input        [7:0] irq,          //interrupt requests
  output logic [3:0] highest_pri); //encoded highest pritority interrupt
  always_comb begin
    priority casez (irq)
      8'b1???????  :  highest_pri = 4'h8; //interrupt 8
      8'b?1??????  :  highest_pri = 4'h7; //interrupt 7
      8'b??1?????  :  highest_pri = 4'h6; //interrupt 6
      8'b???1????  :  highest_pri = 4'h5; //interrupt 5
      8'b????1???  :  highest_pri = 4'h4; //interrupt 4
      8'b?????1??  :  highest_pri = 4'h3; //interrupt 3
      8'b??????1?  :  highest_pri = 4'h2; //interrupt 2
      8'b???????1  :  highest_pri = 4'h1; //interrupt 1
      default      :  highest_pri = 4'h0; //no interrupts
    endcase
  end
endmodule
```



Simultaneous interrupt requests may be asserted, but returns only the highest priority request.

casex Statement

- With a casex statement, any case item bits that are specified with the characters x, X, z, Z or ? are treated as don’t care bits
    - For example: 2b1? can match the case expressions: 2b10, 2b11, 2b1x, or 2b1z or 2b1?

```
module ex_casex(
 input logic [3:0] sel,
 output logic [3:0] y);
 always_comb begin
  casex(sel)
    4’b1xxx: y = 4'b1000;  // process this branch if “sel” MSB is ‘1’ regardless of other bits are X or Z
    4'b01??: y = 4'b0100;
    4'b001?: y = 4'b0010;
    4'b0001: y = 4'b0001;
    default: y = 4'b0000;
   endcase
  end
endmodule
```

case Statement Modifiers

- SystemVerilog introduced two statement **modifiers**
  - **priority** and **unique**
  - Both give information to synthesis to **aid optimization**
  - Both are **assertions** (simulation error reporting mechanisms)

case Statement Modifier: unique

```
module unique_case(
 input logic a,b,c
 output logic [1:0] sel);

 always_comb begin

  unique case(sel) inside
   2'b00 : out = a;
   2'b01 : out = b;
   2'b10 : out = c;
endcase

 end
endmodule
```

unique modified before case indicates to synthesizer that case statement can be considered as complete even thought only three of the four possible values of 2-bit "sel" are specified in case items

<u>Unique indicates to synthesis</u>

- All possible values of case expression are in the case items
- Each case item is unique and only one match should occur.
- **No overlapping case items** and hence case items can be evaluated in parallel.
- It produces **parallel decoding** which may be smaller/faster
- Also known as parallel_case and it indicates that no priority logic is necessary, slow priority encoders removed from designs

<u>Unique indicates to simulation</u>

- At simulation run time, **a match must be found in case items**
- At run time, only one match will be found in case items

case Statement Modifier: priority

```
module priority_case(
 input logic [3:0] sel,
 output logic [3:0] y);
 always_comb begin

 priority casez(sel)
    4'b1???: y = 4'b1000;
    4'b111?: y = 4'b0100;
    4'b001?: y = 4'b0010;
    4'b0001: y = 4'b0001;
   endcase

 end
endmodule
```

priority modifier will indicate to synthesize compiler if sel value is say, 4'b1110 treat first case item 4'b1??? as highest priority and in this case assign Y = 4'b1000

Priority indicates to synthesis

- Priority statement indicates that each selection item in a series of decisions must be evaluated in the order in which they are listed, and all legal cases have been listed.
- A synthesis tool is free to optimize the logic assuming that all other unlisted conditions are don't cares
- All possible values for case expression are in case items
- It indicates that all *other* testable conditions are don't cares and may be used to simplify logic

For Loops and Functions

for Loops

- For loop in synthesizable code is used to replicate hardware logic
  - Does not work like **For** loop in traditional software programming languages such as C/C++
  - In SystemVerilog **For** loop only expands hardware logic
  - For synthesis compiler to **unroll the loop**, number of iterations a loop will execute must be a fixed number

- Syntax :

```
for(<index statement>;  <condition expression>;  <increment statement>) begin
   <one or more statements>
end
```

  - **index statement** : only executed once when the loop starts. May be assignment statement hence LHS is register type (reg, logic, int)
  - **condition expression** : evaluated before first pass of the loop. If true, statements within for loop is executed else loop exits
  - **increment statement** : executed at the end of the each pass of the loop. Condition expression is evaluated again. If true, loop is repeated otherwise exits. May be assignment statement hence LHS is register type (reg, logic, int)

For Loop Non-Blocking Assignment Example for Shift Registers

8-bit Shift Register using For Loop

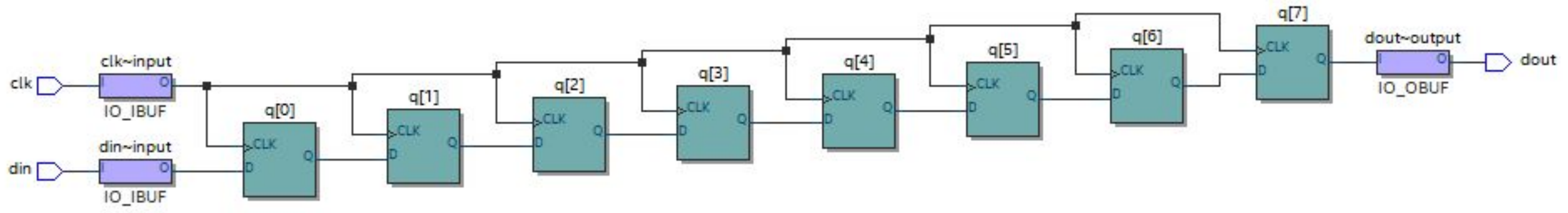
```
module shift_register (
  input logic clk, din,
  output logic dout);
logic [7:0] q;
always_ff@(posedge clk)
 begin
   q[0] <= din;
   for(int i=0; i<7; i=i+1) begin
     q[i+1] <= q[i];
   end
 end
 assign dout = q[7];
endmodule
```

Synthesis compiler will unroll the for loop and replicate 7 chained registers

8-bit Shift Register without using For Loop

```
module shift_register (
  input logic clk, din,
  output logic[7:0] dout);
logic [7:0] q;
always_ff@(posedge clk)
 begin
   q[0] <= d;
   q[1] <= q[0];
   q[2] <= q[1];
   q[3] <= q[2];
   q[4] <= q[3];
   q[5] <= q[4];
   q[6] <= q[5];
   q[7] <= q[6];
 end
assign dout = q[7];
endmodule
```

Synthesis compiler will generate 8-bit shift register by unrolling for loop



For Loop Example for Combinational Logic

$clog2 return the ceiling of the log base 2 of the argument (the log rounded up to an integer value). Example :

  $clog2(4) returns 2
  $clog2(16) returns 4
  $clog2(7) returns 3

Count ones in input bit stream using For Loop

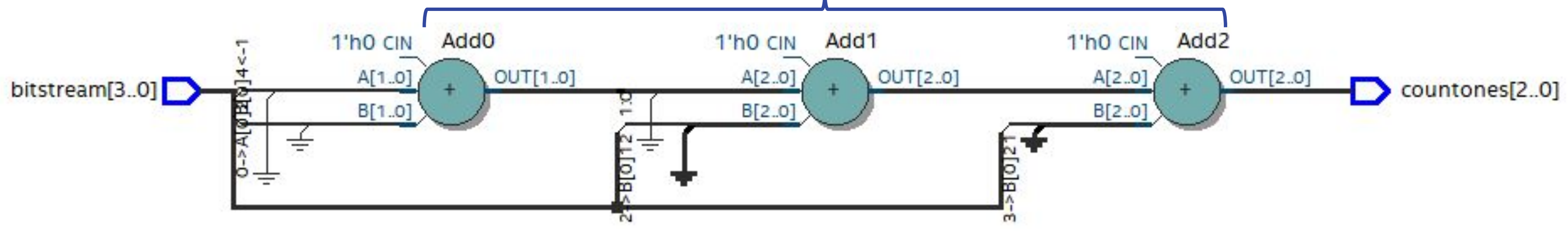
```
module count_ones #(parameter SIZE = 4) (
 input logic [SIZE-1:0] bitstream,
 output logic [$clog2(SIZE) : 0] countones);
 always_comb begin
   countones = 0;
   int index; // iteration variable declared outside for loop
   for(index = 0; index < SIZE; index++) begin
     countones = countones + bitstream[index];
   end
 end
endmodule: count_ones
```

Based on SIZE value, unrolling of for loop will generate that many adder logic

```
countones = countones + bitstream[0];
countones = countones + bitstream[1];
countones = countones + bitstream[2];
countones = countones + bitstream[3];
```

Equivalent logic for SIZE=4

Synthesis compiler unrolled For loop and generated three instances of "adder" logic since SIZE=4



Function

- Function is created when **same operation is to be repeated and executed**
  - It enables **reusability** and makes the code more modular and maintainable

  - Syntax :

```
function <optional datatype> function_name(<optional input arguments>);
  begin
    <programming statements>
  end
Endfunction
```

  - When function is called, it executes programming statements and returns a value
  - Functions can be declared within a **module** and an **interface**
  - Function is used to **create combinational logic**
  - Function can be called from **continuous assignment** statements, **always**, **initial** procedural blocks
  - Function definition can appear before or after the statement which calls function
  - Function is **synthesizable** with coding guidelines followed
  - Function can be used in **non-synthesizable code for testbench development**
  - Functions can have **input**, **output**, **inout** and **logic** ports declared in its argument list

Static and Automatic Functions

- Functions can be declared as **static** or **automatic**
  - If not specified, then **default is static**
- Static function **retains state** of any internal variables or storage **in simulation** from one call to the next
  - Function name and inputs will retain their values in simulation when the function exits
- Automatic function **allocates new storage for internal variables** each time the function is called

```
module ex_static_add_func(
  input logic[1:0] a, b, c,
  output logic[1:0] q);

 function logic[1:0] add3(input logic [1:0] x, y, z);
    logic [1:0] t;
    begin
      t = x + y;
      add3 = t + z;
     end
   endfunction

always_comb
   q = add3(a, b, c);
endmodule: ex_static_add_func
```

Variable "t" is shared across all invocations of "add3" since add3 is a static function !!!

```
module ex_automatic_add_func(
  input logic[1:0] a, b, c,
  output logic[1:0] q);

 function automatic logic[1:0] add3(input logic [1:0] x, y, z);
    logic [1:0] t;
    begin
      t = x + y;
      add3 = t + z;
     end
   endfunction

always_comb
   q = add3(a, b, c);
endmodule: ex_automatic_add_func
```

"automatic" ensures that all local variables are truly local. Each invocation of "add3" will use a different "t"!

Return on Functions

- Functions can **return values** using two approaches :
  1. Using keyword "**return**"
  2. Assigning value to **variable** with same **name** as function

```
module ex_static_add_func(
  input logic[1:0] a, b, c,
  output logic[1:0] q);

 function logic[1:0] add3(input logic [1:0] x, y, z);
    logic [1:0] t;
    begin
      t = x + y;
      return t + z;
     end
   endfunction

always_comb
   q = add3(a, b, c);
endmodule: ex_static_add_func
```

Returning value of t+z using "return" keyword instead of assigning to implicit variable add3

```
module ex_static_add_func(
  input logic[1:0] a, b, c,
  output logic[1:0] q);

 function logic[1:0] add3(input logic [1:0] x, y, z);
    logic [1:0] t;
    begin
      t = x + y;
      add3 = t + z;
     end
   endfunction

always_comb
   q = add3(a, b, c);
endmodule: ex_static_add_func
```

Returning value of t+z using by assigning result to implicitly declared variable name add3 which same as function name

void Function

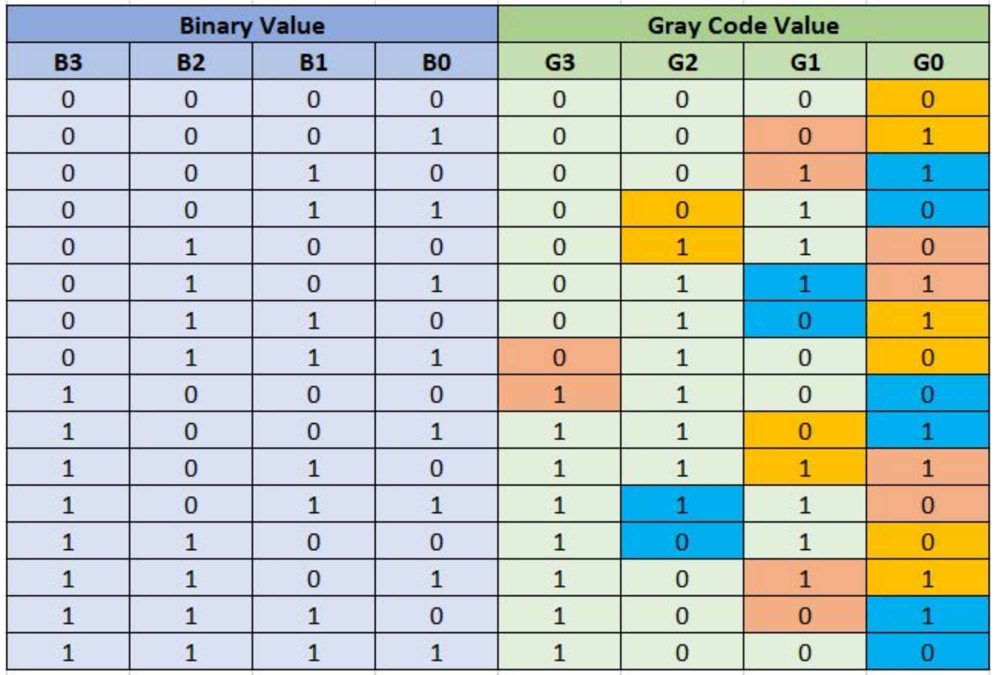
- Void function does not return value
  - It can return results by driving variable declared with output direction in its argument list
  - Function can have only one argument with output direction

```
module ex_static_add_func(
  input logic[1:0] a, b, c,
  output logic[1:0] q);

 function void add3(input logic [1:0] x, y, z, output logic [1:0] sum);
    logic [1:0] t;
    begin
      t = x + y;
      sum = t + z;
     end
   endfunction

always_comb
   add3(a, b, c);
endmodule: ex_static_add_func
```

Returning value of t+z by driving output variable sum

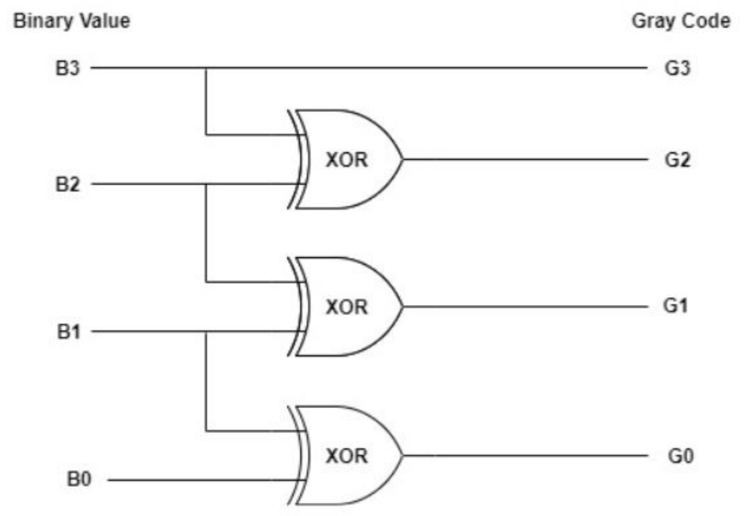
Binary to Gray Code Conversion

- Gray code named after Frank Gray, is an ordering of the binary numeral system such that two successive numbers differ in only one bit
- Binary to Gray conversion :
  - MSB of the gray code is always equal to the MSB of the given binary code.
  - Other bits of the output gray code can be obtained by XORing binary code bit at that index and previous index.



Binary Value				Gray Code Value			
B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

From previous to next value, only 1-bit changes at a time



G(3) = B(3)
G(2) = B(2) xor B(3)
G(1) = B(1) xor B(2)
G(0) = B(0) xor B(1)

Binary to Gray Code Conversion using Function

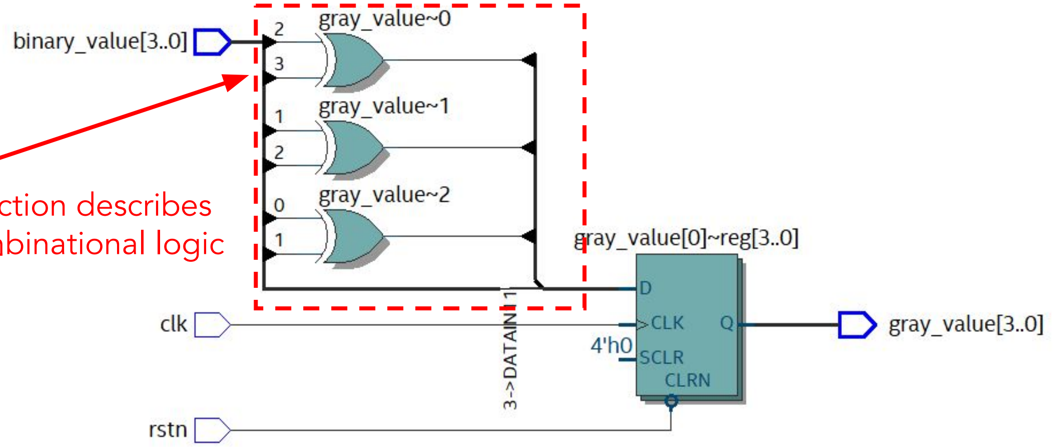
```
module binary_to_gray_conv  #(parameter N = 4)(
  input logic clk, rstn,
  input logic[N-1:0] binary_value,
  output logic[N-1:0] gray_value);

 // Function to convert binary to gray value
 function automatic [N-1:0] binary_to_gray(logic [N-1:0] value);
   begin
     binary_to_gray[N-1] = value[N-1];
     for(int i=N-1; i>0; i = i - 1)
       binary_to_gray[i-1] = value[i] ^ value[i - 1];
    end
 endfunction

 // Store binary2gray output in a register
 always_ff@(posedge clk or negedge rstn) begin
   if (!rstn) begin
     gray_value <= 0;
   end
   else begin
     gray_value <= binary_to_gray(binary_value);
   end
 end
endmodule: binary_to_gray_conv
```

default return type is logic if not specified

return value assigned to a automatic declared variable with same name as a function name

call to a function with input value passed

Function describes combinational logic


binary_value[3..0]
gray_value~0
gray_value~1
gray_value~2
gray_value[0]~reg[3..0]
clk
rstn
4'h0
3->DATAIN 1
D
CLK
Q
SCLR
CLRN
gray_value[3..0]