

Lecture-10 & 11: RTL Programming Statements

ECE-111 Advanced Digital Design Project

Vishal Karna

Winter 2022

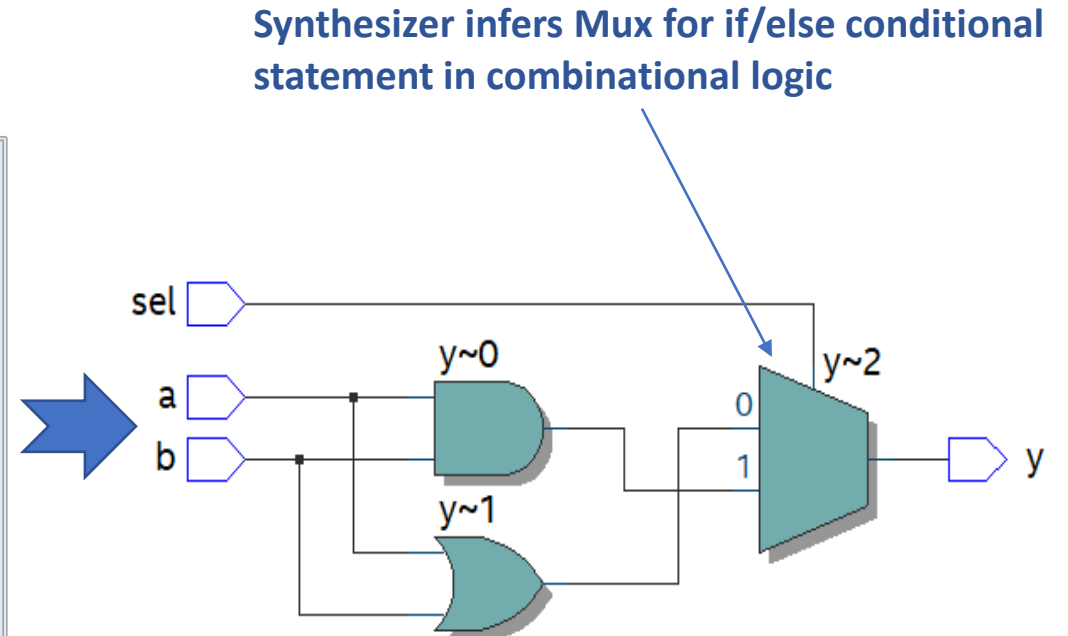
SystemVerilog RTL Programming Statements Summary

| 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 |

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
 - If there is no else branch, then a latch will be inferred to retain previous value
 - 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  
  end  
endmodule: ex1
```



if/else Conditional Statement

❑ else branch of if/else statement is optional

- If there is no else branch and if expression evaluates false ('0') or unknown ('X'), 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
```

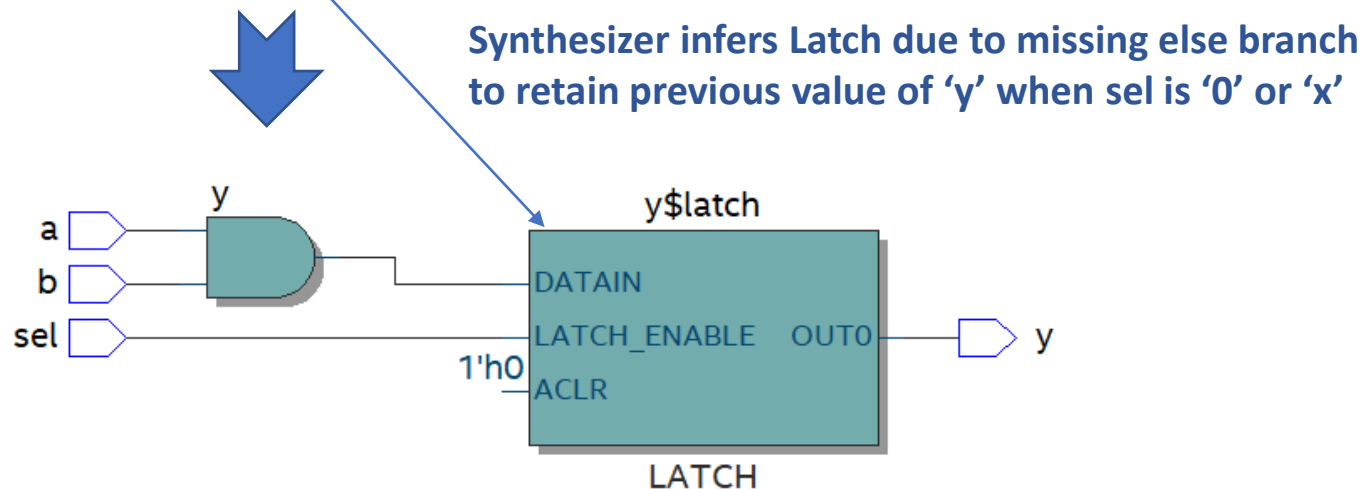
```
    if(sel == 1)
```

```
      y = a & b;
```

```
  end
```

```
endmodule: ex2
```

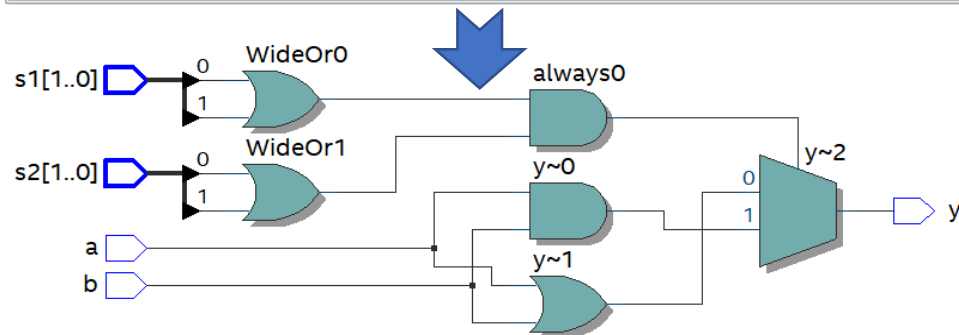
If sel is '0', there is no false branch to execute,
hence 'y' retains its previous value,
modeling the storage behavior of a latch



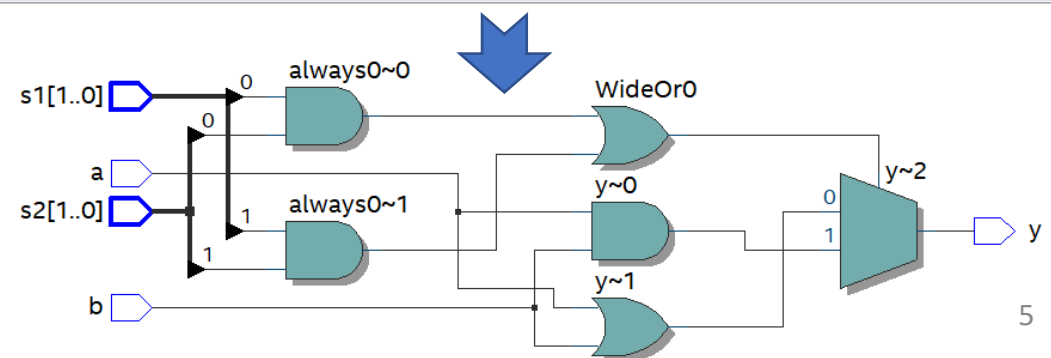
if/else Conditional Statement

- ❑ Using logical operators versus bitwise operators in if condition expression can result in a different circuit
 - Only use 1 bit vectors or use logical operators in if condition expression to return true/false
 - Do not perform true/false on vectors, since evaluating vectors as true or false could lead to design errors

```
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 condition  
      y = a & b; // 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 mis-match in simulation  
    end // vs post synthesis gate level model  
  endmodule: ex4 behavior
```



if/else Conditional Statements

❑ Chained if-else-if decisions can be specified

- if-else-if decisions are evaluated in the order it is specified
- Gives priority to if condition decisions listed first

```
module priority_encoder_4to2(  
  input logic [3:0] enc_in,  
  output logic [1:0] enc_out  
);  
always@(enc_in[0] or enc_in[1] or enc_in[2] or enc_in[3])  
begin  
  if(enc_in[0]) enc_out = 2'h0;  
  else if(enc_in[1]) enc_out = 2'h1;  
  else if(enc_in[2]) enc_out = 2'h2;  
  else if(enc_in[3]) enc_out = 2'h3;  
  else enc_out = 2'h0;  
end  
endmodule: priority_encoder_4to2
```

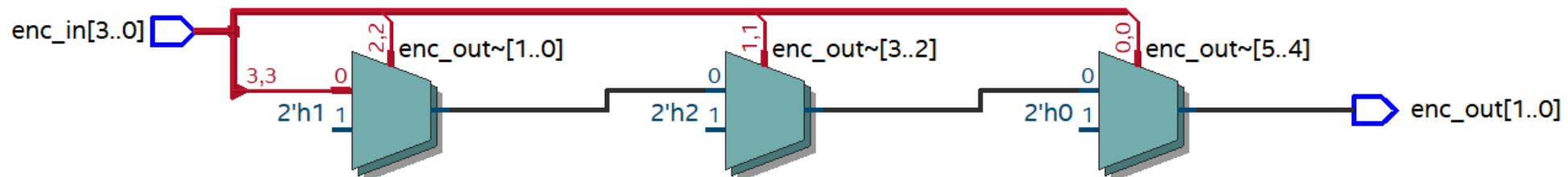
If enc_in[0] and enc_in[1] both becomes '1' same time, enc_in[0] branch gets executed first due to its order in which it is specified.

Chained if-else-if results in a series of mux connected back to back

If default else branch is not specified then synthesizer will infer a mux to retain previous value when enc_in value is either 4'b000 or 4'bZZZZ or 4'bXXXX



Priority encoder is implemented as series of mux, where output of mux is input to the next stage mux.

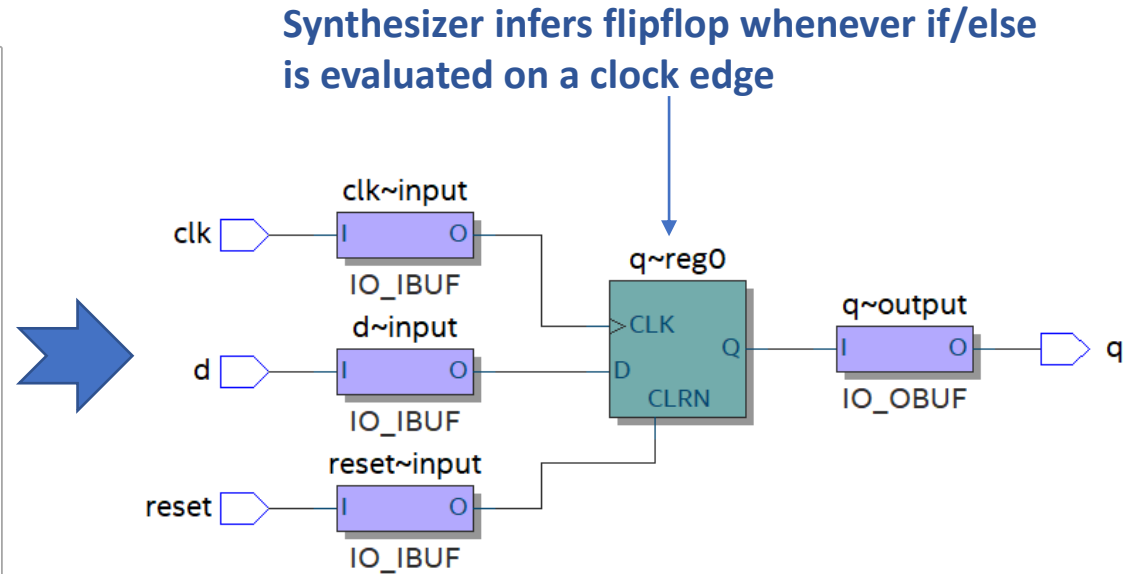


if/else Conditional Statement

- ❑ if/else evaluated on a clock edge behaves as a flip-flop

```
module dff(  
  input logic clk, reset, d,  
  output logic q  
);  
always_ff@(posedge clk or negedge reset)  
begin  
  if(!reset)   
    q <= 0;  
  else  
    q <= d;  
end  
endmodule: dff
```


At same time if there is
posedge of clk and negedge of
reset, then if(!reset) branch will
execute first since this
condition is described first



For loop

- ❑ **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)
 - **Examples :**
 - for (**int** i=0; i<8; i=i+1)  **int 'i'** locally declared as part of initial statement, is local to **for loop** and hence 'i' cannot be accessed outside for loop
 - **logic**[31:0] address;
for (address=0; address<32'h4000; address = address+16)

Example of Shift Register using For Loop-Blocking Assignment

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  
        assign dout = q[7];  
    end  
endmodule
```

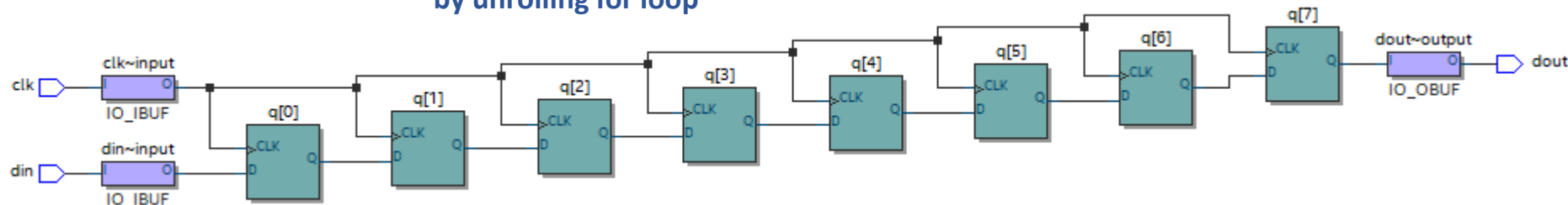
Synthesis compiler will
unroll For loop and
replicate 7 chained
registers

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



Iteration variable
'i' declared inside
for loop, hence
called automatic
variable.
Scope of 'i' is
within for loop
only. Once for
loop terminates,
'i' is no longer
accessible

Example of For Loop replicating 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

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

For SIZE=4,
Equivalent
logic

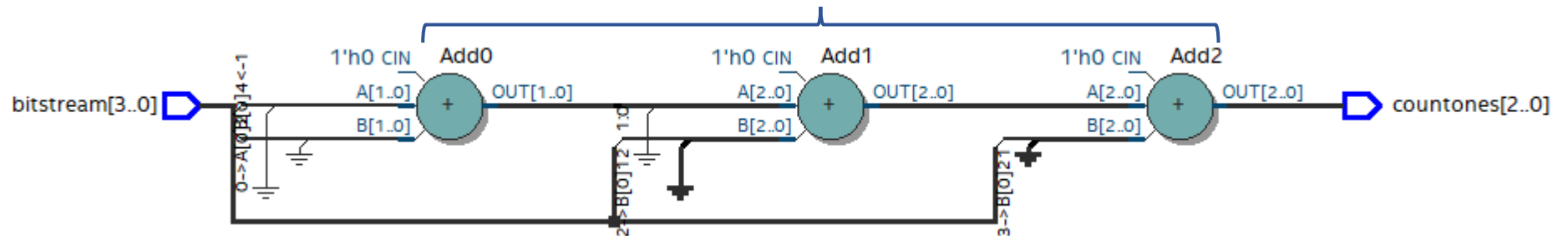
Count ones in input bit stream using For Loop

```
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



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



repeat loop

- ❑ A **repeat** loop will blindly execute block of code a set number of times
 - As with **For** loops, a repeat loop is synthesizable if the bounds of the loop is a fixed value
 - **repeat** loop's index can never be used inside the loop.
 - In Synthesizable code, **repeat** loops should only be used to expand replicated code !!
 - More often, **repeat** loops are used in testbenches.

- **Syntax :**

```
repeat(<iteration_index>) begin
    <one or more statements>
end
```

- **Example of repeat loop in testbench (non-synthesizable code) :**

```
module testbench;
logic clock;
initial begin
    repeat(5) begin
        #10 clock = !clock;
    end
end
endmodule: testbench
```

Same as

```
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
```

Example of Repeat loop in Synthesizable Code

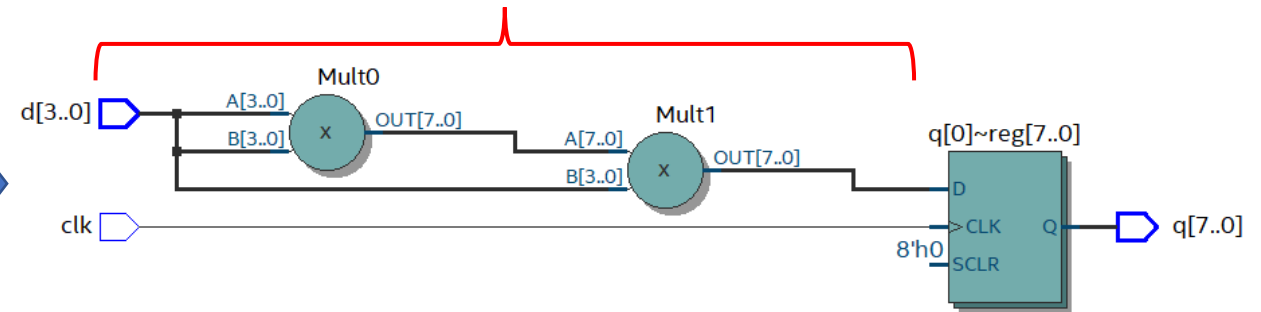
Compute Exponential d^E using repeat loop

```
module exponential
#(parameter E = 3, parameter N = 4, parameter M = N * 2)(
  input logic clk,
  input logic [N-1:0] d,
  output logic [M-1:0] q);

always_ff@(posedge clk) begin
  logic [M-1:0] q_temp;
  if(E==0)
    q <= 1; // if E==0, d0 will result in value '1'
  else begin // else multiply 'd' value by 'E' num of times
    q_temp = d;
    repeat(E-1) begin
      q_temp = q_temp * d;
    end
    q <= q_temp; // final multiplication output stored in D-FF
  end
end
endmodule: exponential
```



For E=3, Two chained multipliers are inferred by synthesizer to perform d^3 and inferred D-Flipflop to store output exponential result



Note : Total combinational delay of chained multiplier logic should be within 1 clock cycle in order for valid and stable result to be registered in output flipflops. Some synthesis compilers can do register retiming, to insert or move registers to create pipeline within combinational logic

while loop

- ❑ A **while** loop executes programming statement(s) until the end expressions becomes false
 - As with **For** loops, a repeat loop is synthesizable if the bounds of the loop is a fixed value
 - End expression of **while** loop is tested at the top of the loop
 - If end expression is **False** then statement(s) within **while** loop are not executed
 - If end expression is **True** then statement(s) within **while** loop are executed, then it returns at the top of the loop and tests the end expression again
 - In Synthesizable code, **while** loop usage is not common instead **for** and **repeat** loops are used !!
 - More often, **while** loops are used in testbench code (non-synthesizable)

- **Syntax & Example :**
 while(*<end_expression>*) **begin**
 <one or more statements>
 end

```
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;  
        while(index < SIZE) begin  
            countones = countones + bitstream[index];  
            index = index + 1;  
        end  
    end  
endmodule: count_ones
```

- This implementation using while loop will not synthesize since end expression has non-constant loop condition (**index** is non-constant and changing inside while loop).

- Synthesizer cannot statically determine how many times loop will execute and therefore cannot roll out the loop.

do-while loop

- ❑ A **do-while** executes programming statement(s) until the end expressions becomes false
 - Similar to while loop except, the end expression is tested at the bottom of the loop
 - Statement(s) within the **do-while** loop will executed at least once when the loop is first entered
 - If the end expression if False when loop reaches the bottom, the loop exits
 - If end expression is True then loop returns back to the top and executes statement(s) within **do-while** loop
 - In Synthesizable code, **do-while** usage is not common instead **for** and **repeat** loops are used !!
 - More often, **do-while** loops are used in testbench code (non-synthesizable)

- **Syntax and Example :**

```
do begin
    <one or more statements>
end while(<end_expression>);
```

```
module testbench;
initial begin
    int count_value;
    do begin
        $display("count value = %d\n", count_value);
        count_value = count_value + 1;
    end while(count_value < 16)
end
endmodule: testbench
```

print count value and
increment count value at
least once and then
until (count_value < 16)
condition is true, keep
printing and incrementing
count value else exit the
loop

forever loop

- ❑ A **forever** loop executes programming statement(s) inside the loop continuously and will never stop running
 - Forever loop has indefinite iteration, hence **not synthesizable** and mostly used in testbench code
 - Forever loops are similar to for loops and while loops except forever loop will never stop running and while/for loops have a limit

- **Syntax & Example :**

forever begin

<one or more statements>

end

```
module clock_gen_testbench (  
  logic clock = 1'b0;
```

```
  initial
```

```
  begin
```

```
    forever
```

```
      #10 clock = !clock;
```

```
  end
```

```
endmodule: clock_gen_testbench
```

} forever will run continuously and will generate posedge and negedge of clock every 10 timeunits

foreach loop

- ❑ A **foreach** loop iterates through all the dimensions of **unpacked array**
 - **foreach** will automatically declare its loop control variables, starting and ending indices of the array and direction of indexing (incrementing or decrementing)
 - Used for testbench code (**Non-synthesizable code**)
 - **Syntax :**

```
foreach(<iteration_index>) begin
    <one or more statements>
end
```
 - **Example :**

```
module testbench;
byte mem[0:4];
initial begin
    byte counter = 100;
    foreach(mem[idx]) begin
        mem[idx] = counter++;
    end
    for(int i = 0; i < $size(mem); i++) begin
        $display("mem[%0d]: %0d", i, mem[i]);
    end
end
endmodule: testbench
```

idx variable is automatically declared, initialized and incremented

Simulation Output:

```
mem[0]: 100
mem[1]: 101
mem[2]: 102
mem[3]: 103
mem[4]: 104
```

case statement

❑ Case statement provides a concise way to represent series of decisions choices

- It is C like switch statement with exception that **case** does not support break to exit from branch
- SystemVerilog case has implied "break" statement
- case statements are 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; // If operation value is 'X' or 'Z' default branch is executed
  endcase
end
endmodule: alu
```

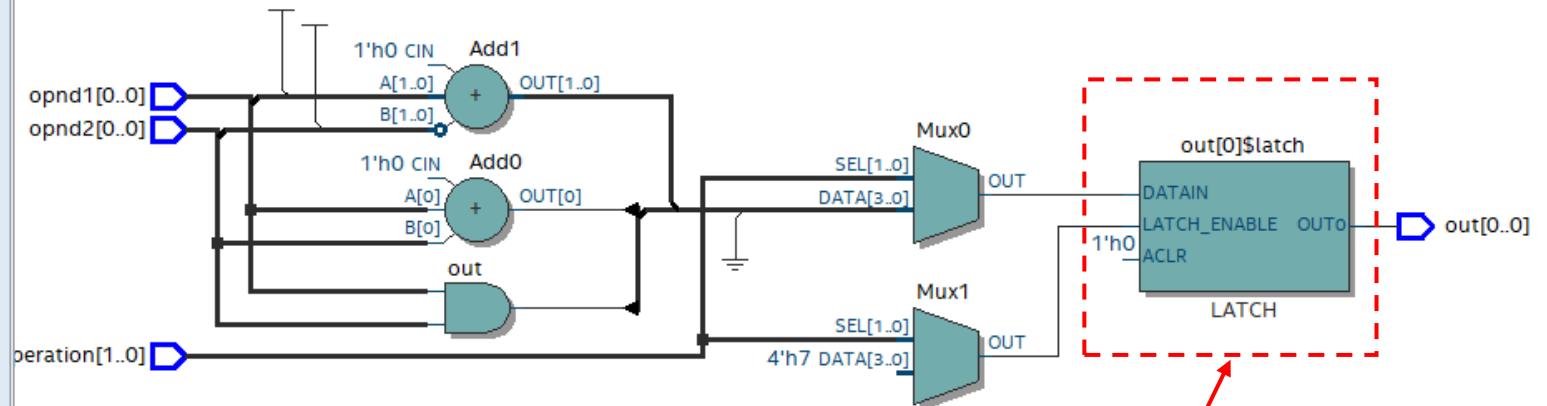
case expression use **===** **case equality operator** for exact match of 4 state values. If any bit of "**operation**" in case expression is '**X**' or '**Z**' it will select default branch.

case items are evaluated in order they are listed

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

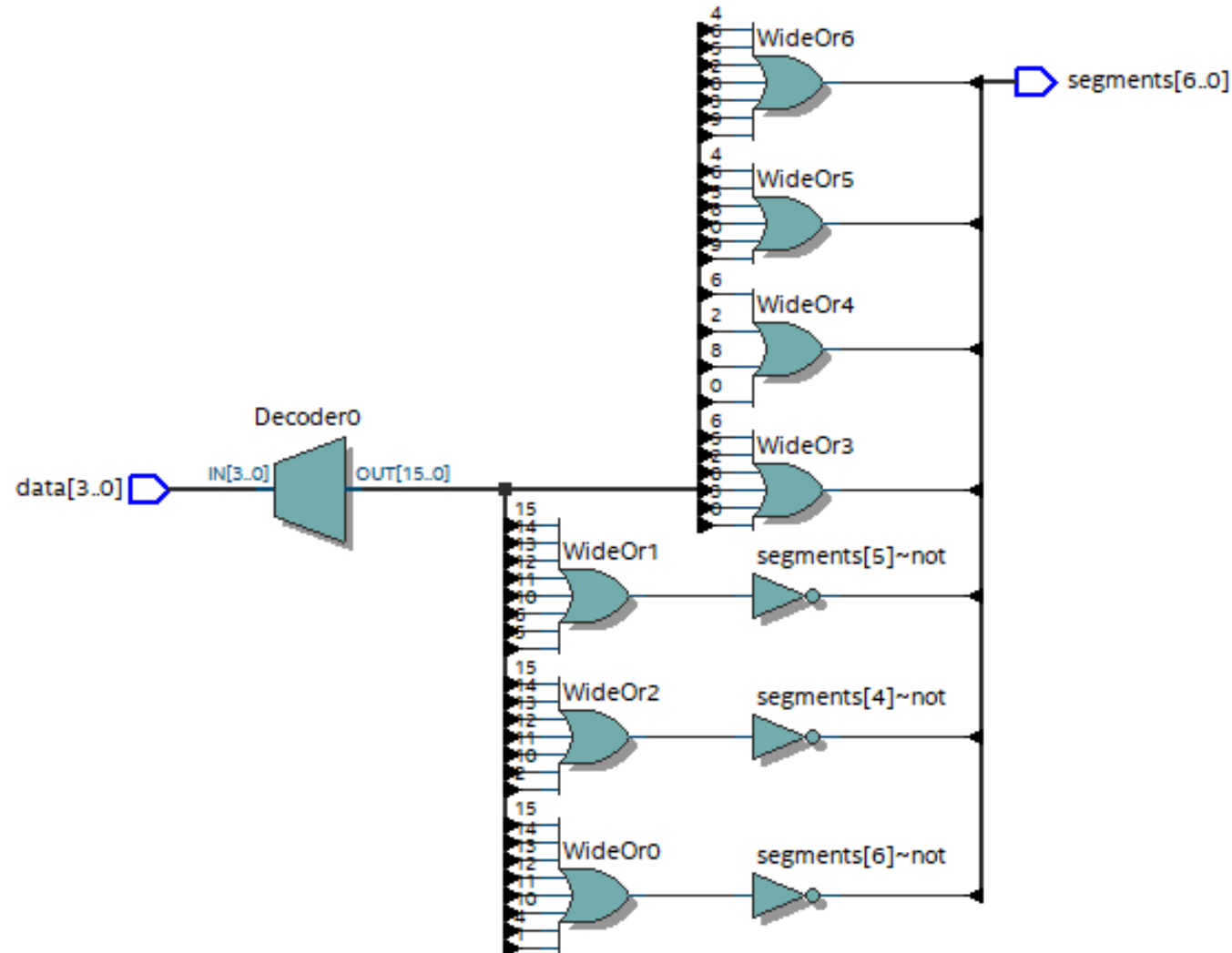
Seven Segment LED Example Use Case Statement

```
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
```

case statement translates into a more complex “multiplexor” similar to if-then-else

Case statement with “default” item expression are known as **full_case** and full case removes latches from my designs. !

Seven Segment LED Example Use Case Statement



case inside statement

- ❑ **Case inside statement is similar to case statement except it used ==? 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

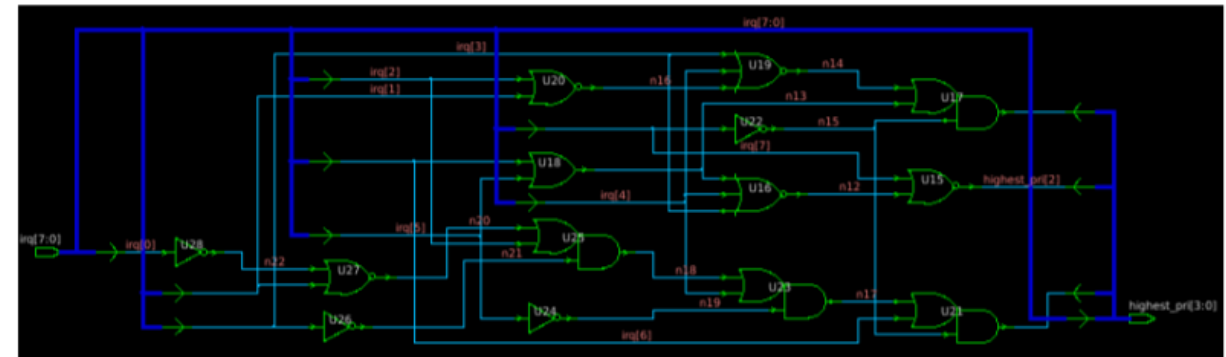
```
module case_inside(  
    input logic [3:0] sel,  
    output logic [3:0] y);  
    always_comb begin  
        case(sel) inside  
            4'b1???: y = 4'b1000; // this branch will be selected if "sel" MSB is '1', regardless of any values in rest of the "sel" bits  
            4'b01??: y = 4'b0100; // this branch will be selected if "sel" MSB is '01', regardless of any values in rest of the "sel" bits  
            4'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

❑ 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 priority 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

❑ 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'  
      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 : Unique and Priority

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

unique case

```
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 though only three of the four possible values of 2-bit "sel" are specified in case items

Unique indicates to synthesis

- 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

Unique indicates to simulation

- 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 modifiers : Unique and Priority

priority case

```
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

Functions

❑ Function is created when same operation is to be repeated and executed

- It enables reusability and make code more modular and maintainable

▪ Syntax :

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

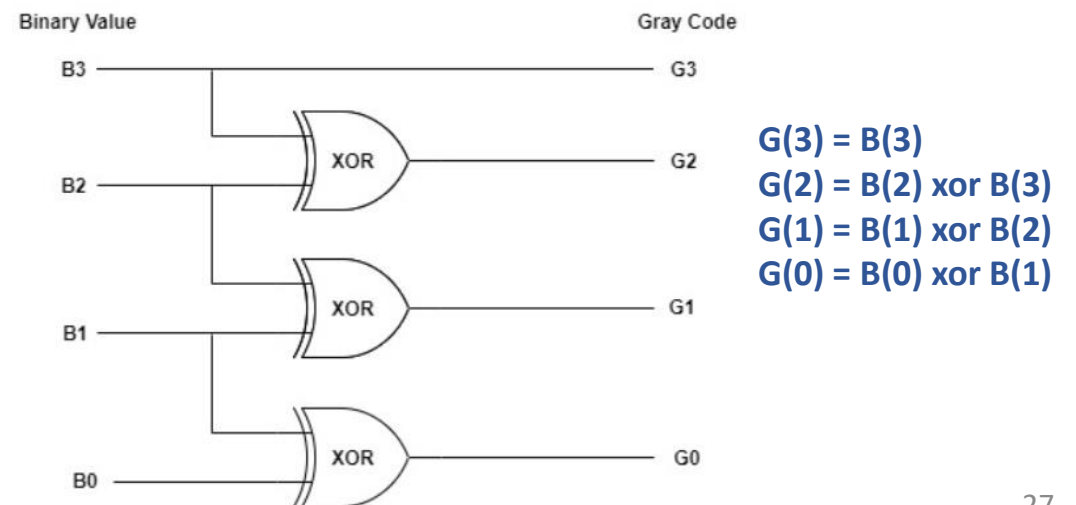
- When function is called, it executes programming statements and returns a value
- Functions can be declared within a module and an interface
- Function is used to create combinational logic
- Function can be called from **continuous assignment** statements, **always**, **initial** procedural blocks
- 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** and **inout** ports declared in its argument list

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
 - Gray code was originally designed to prevent spurious output from electromechanical switches
 - Gray codes are widely used to facilitate error correction in digital communication applications
- ❑ **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

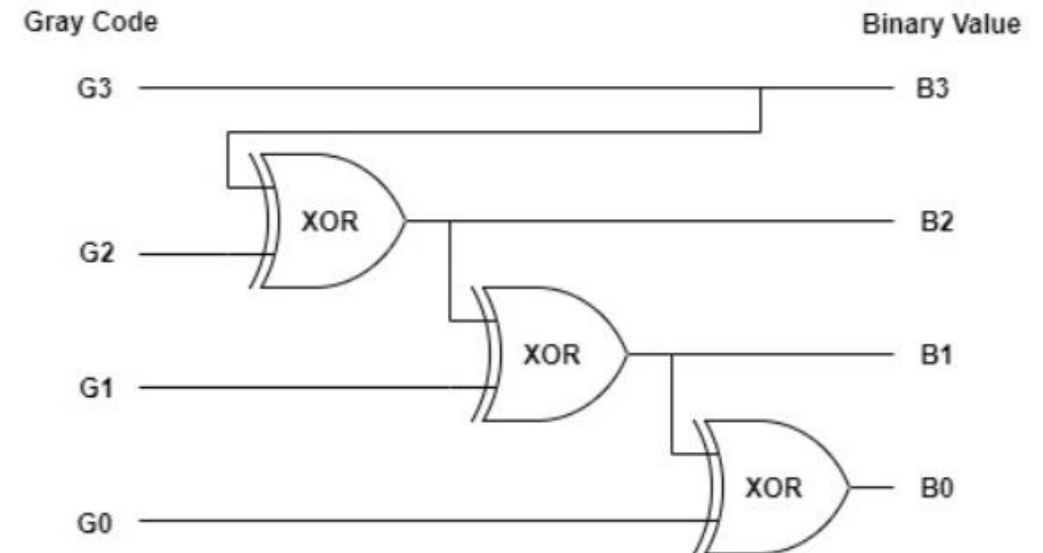


Gray Code to Binary Value Conversion

□ Gray Code to Binary Conversion :

- MSB of the binary code is always equal to the MSB of the given binary number.
- Other bits of the output binary code can be obtained by checking gray code bit at that index.
 - **Technique 1** : If current gray code bit is 0, then copy previous binary code bit, else copy invert of previous binary code bit. OR
 - **Technique 2** : Other bits of the output binary value can be obtained by XORing gray code bit at that index and binary bit at next index.

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



$$B(3) = G(3)$$

$$B(2) = G(2) \text{ xor } B(3)$$

$$B(1) = G(1) \text{ xor } B(2)$$

$$B(0) = G(0) \text{ xor } B(1)$$

Binary to Gray 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 default return type is logic if not specified

```
  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

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

// Store binary2gray output in a register

```
always_ff@(posedge clk or negedge rstn) begin
```

```
  if (!rstn) begin
```

```
    gray_value <= 0;
```

end

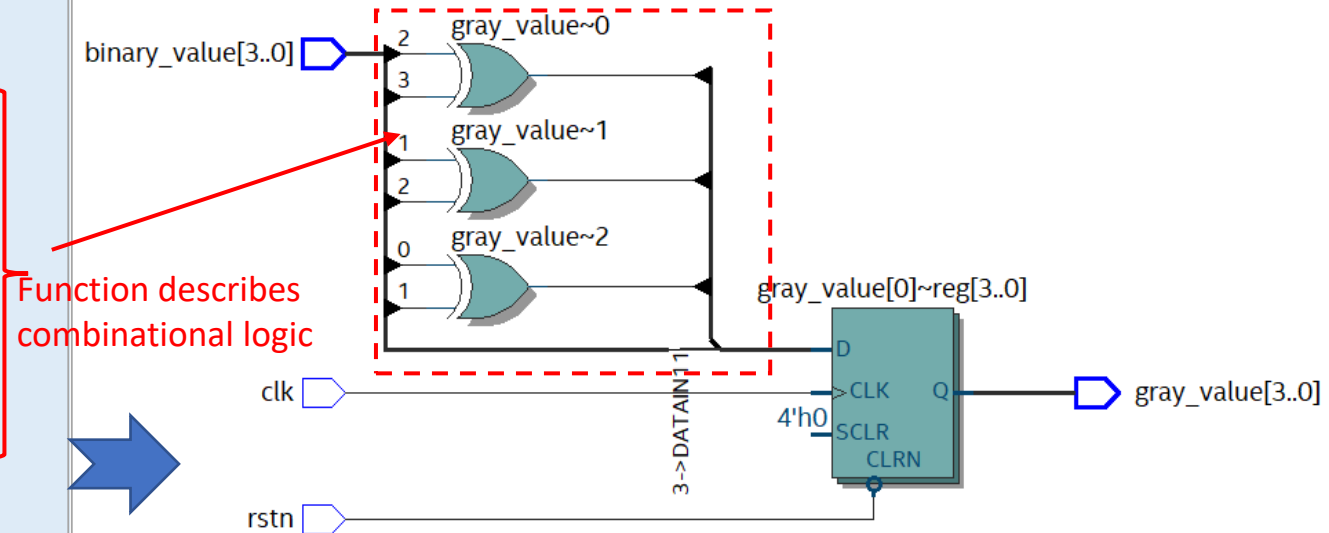
```
  else begin
```

```
    gray_value <= binary_to_gray(binary_value);
```

end

end

```
endmodule: binary_to_gray_conv
```



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 from one call to the next
 - Function name and inputs will retain their values 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 :

- Using keyword “return”
- 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 Functions

❑ Void functions 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

Function Rules

- ❑ Functions executes in zero simulation time, hence
 - Function definition cannot contain any time controlled statements such as **#**, **@**, or **wait**.
- ❑ Function definition cannot contain **non-blocking** assignment statements
- ❑ Functions can call other functions but cannot call tasks
- ❑ Functions can have any number of inputs but only one output (one return value)
- ❑ The order of inputs to a function dictates how it should be wired up when called
- ❑ Functions can drive global variables external to the function
- ❑ Variables declared inside a function are local to that function
- ❑ Functions can be ***automatic***
- ❑ Function return type defaults to one bit logic unless defined explicitly
- ❑ Function definition with return type must include either :
 - using **return** keyword or
 - an assignment of result value to the internal variable that has the same name as the function.

Tasks

❑ Task encapsulates one or more programming statements which can be called from different part of the code

- Syntactically similar to function however task do not have a return value
- **Syntax :** `task task_name(<optional input arguments>;`
 `begin`
 `<programming statements>`
 `end`
 `endtask`
- Tasks can be declared within a module and a interface
- Tasks can be used for modeling both combinational and sequential logic.
- Programming statements within a task can advance time
 - Tasks contain any time controlled statements such as **#**, **@**, or **wait**, **posedge**, **negedge**, etc
 - Synthesis compiler require task run in zero simulation time which is identical void functions

Binary to Gray Conversion using Task

```
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);
```

// Task to convert binary to gray value

```
task binary_to_gray(logic [N-1:0] value);
```

```
begin
```

```
  gray_value[N-1] = value[N-1];
```

```
  for(int i=N-1; i>0; i = i - 1)
```

```
    gray_value[i-1] = value[i] ^ value[i - 1];
```

```
end
```

**Direct driving of output signal
gray_value from task**

```
endtask
```

// Store binary2gray output in a register

```
always_ff@(posedge clk or negedge rstn) begin
```

```
  if (!rstn) begin
```

```
    gray_value <= 0;    call to a task with input value passed
```

```
  end
```

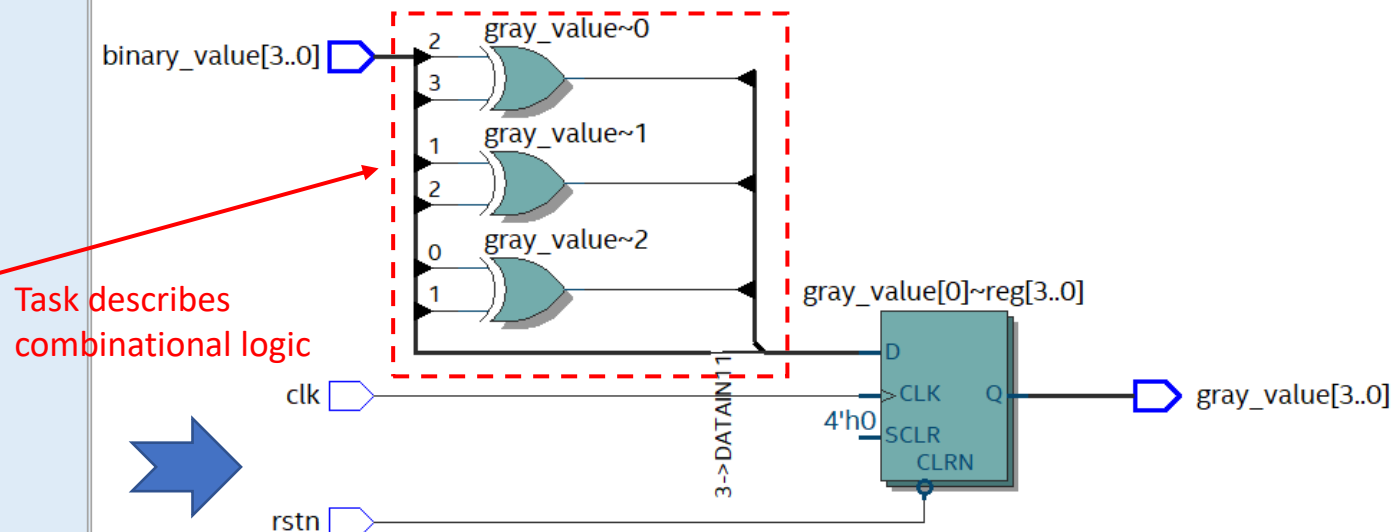
```
  else begin
```

```
    binary_to_gray(binary_value);
```

```
  end
```

```
end
```

```
endmodule: binary_to_gray_conv
```



Tasks Rules

- ☐ Tasks can contain both non-blocking and blocking assignment statements
- ☐ The task would return values in output arguments
- ☐ Tasks can call another tasks and functions
- ☐ A task can support multiple goals and can calculate multiple result values
 - Task can have multiple inputs and outputs specified in its argument list
- ☐ Tasks can take, drive and source global variables, when no local variables are used.
 - When local variables are used, output is assigned only at the end of task execution.
- ☐ A task must be specifically called with a statement, it cannot be used within an expression as a function can.
- ☐ Tasks can be used in both synthesizable and non-synthesizable code

Generate and Genvar

□ Generate

- SystemVerilog provides generate statement to create multiple instantiations of module or code within module
- There are two types of generate statements :
 - generate for loops
 - generate conditionals
 - generate if-else
 - generate case

generate for loop syntax

```
genvar i;  
generate  
  for (i = 0; i < N; i = i + 1) begin  
    ....  
    ....  
    ....  
  end  
endgenerate
```

generate if-else syntax

```
generate  
  if(<expression>) begin  
    ....  
  end  
  else if(<expression>) begin  
    ....  
  end  
endgenerate
```

□ Genvar

- **genvar** variable is a special integer variable to control and evaluate the generate loop during elaboration
- The value of a **genvar** variable can only be assigned a positive number, 0, X or Z,
- **genvar** variable can only be used during elaboration, and cannot be accessed during runtime
- **genvar** declaration can be inside or outside the generate region, and the same loop index variable can be used in multiple generate loops, as long as the loops don't nest

Generate Statement

☐ Permitted items in generate statements are :

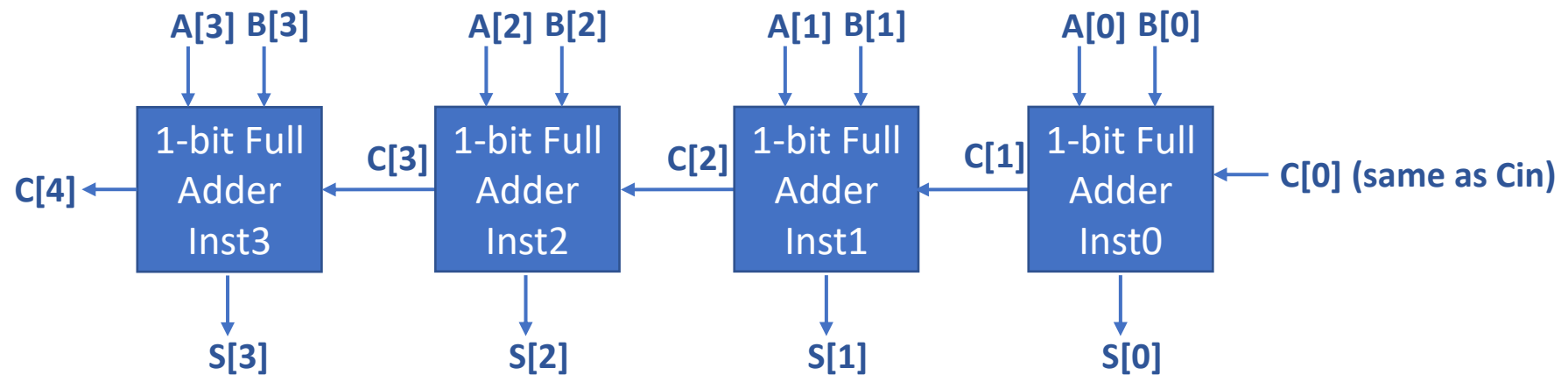
- Any number of module and primitive instances
- Any number of initial or always procedural blocks
- Any number of continuous assignments
- Any number of net and variable declarations
- Any number of parameter redefinitions
- Any number of task or function definitions

☐ Items that are not permitted in a generate statement include:

- port declarations
- constant declarations
- specify blocks.

Ripple Carry Adder

- ❑ **A Ripple Carry Adder is made of a number of full adders cascaded together.**
 - Since carry bit from previous full adder ripples (connected) to the next full adder, hence the name Ripple carry adder
 - It is used to add together two binary numbers using simple logic gates
 - Ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder.
 - The figure below shows 4 full-adders connected together to produce a 4-bit ripple carry adder.
 - only the first full adder may be replaced by a half adder (under the assumption that $C[0] = 0$).

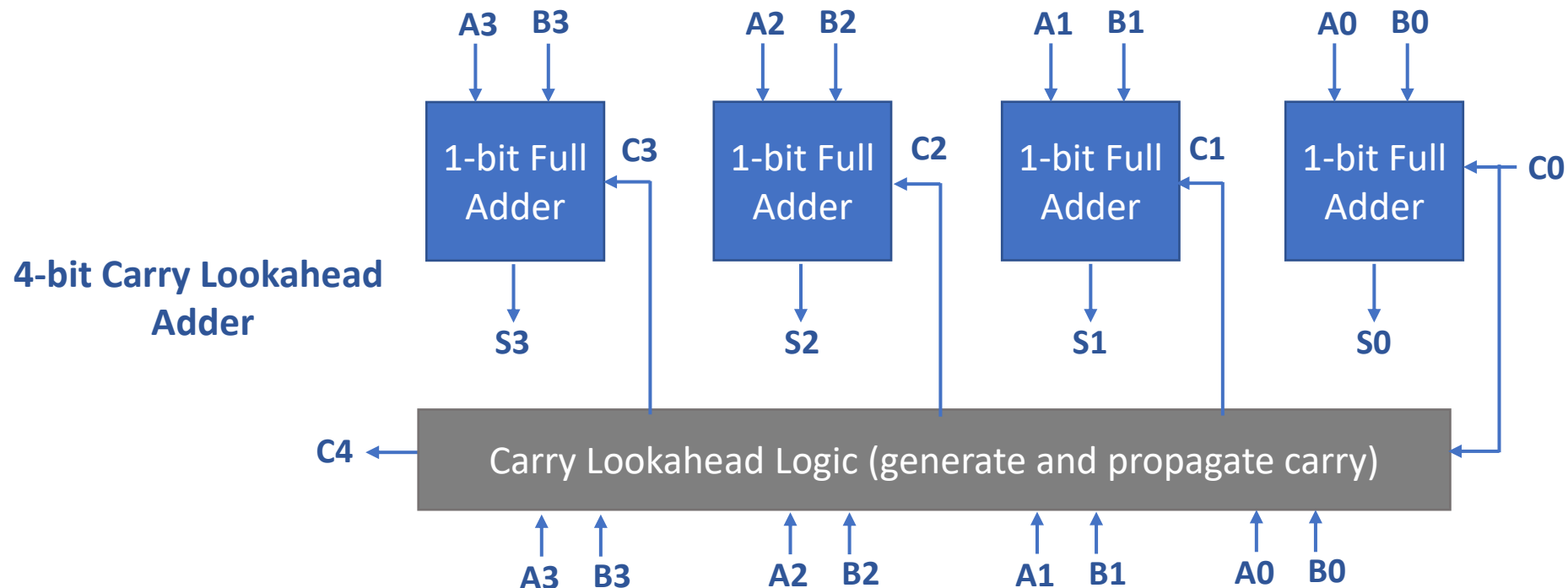


4-bit Ripple Carry Adder

Carry Lookahead Adder

❑ A Carry lookahead Adder is made of a number of full adders cascaded together.

- Carry lookahead adder is similar to ripple carry adder with the difference that it calculates the carry bit before the full adder is done with its operation.
- Advantage of carry lookahead adders is that it adds two numbers faster than ripple carry adder
- The drawback is that carry lookahead adder takes more logic.
- For faster performance when adding two number select carry lookahead adder implementation and for more lower resource usage select ripple carry adder implementation
- The figure below shows 4 full-adders connected together to produce a 4-bit lookahead carry adder.



Ripple Carry Adder using Generate

```
module ripple_carry_adder #(parameter N = 4)(  
  input logic[N-1:0] A, B,  
  input logic CIN,  
  output logic[N:0] result;  
  logic[N:0] l_carry;  
  logic[N-1:0] l_sum;  
  // assign Carry in to first full adder carryin  
  assign l_carry[0] = CIN;
```

// Instantiate Full Adder for 'N' instances

genvar i; ← genvar 'i' controls generate for loop iteration

generate

for(i=0; i<N; i=i+1) begin: fa_loop

fulladder fa_inst(

.a(A[i]),
.b(B[i]),
.cin(l_carry[i]),
.sum(l_sum[i]),
.cout(l_carry[i+1]));

end: fa_loop

endgenerate

// final result of addition and carry

assign result = {l_carry[N], l_sum};

endmodule: ripple_carry_adder

mandatory
to have
named for
loop inside
generate
body

generate for loop
statement will
create 'N' number
of full adder
module instances

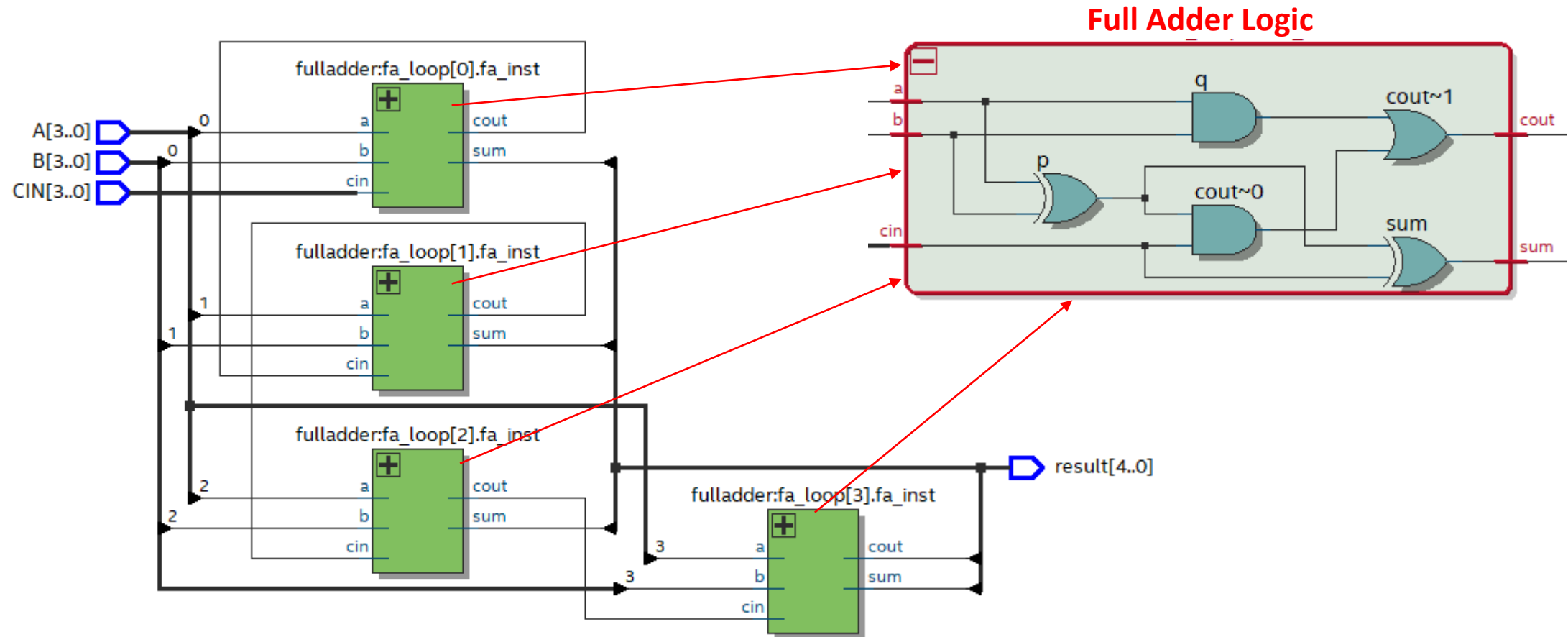
fulladder fa_inst0(
.a(A[0]),
.b(B[0]),
.cin(l_carry[0]),
.sum(l_sum[0]),
.cout(l_carry[1]);

fulladder fa_inst1(
.a(A[1]),
.b(B[1]),
.cin(l_carry[1]),
.sum(l_sum[1]),
.cout(l_carry[2]);

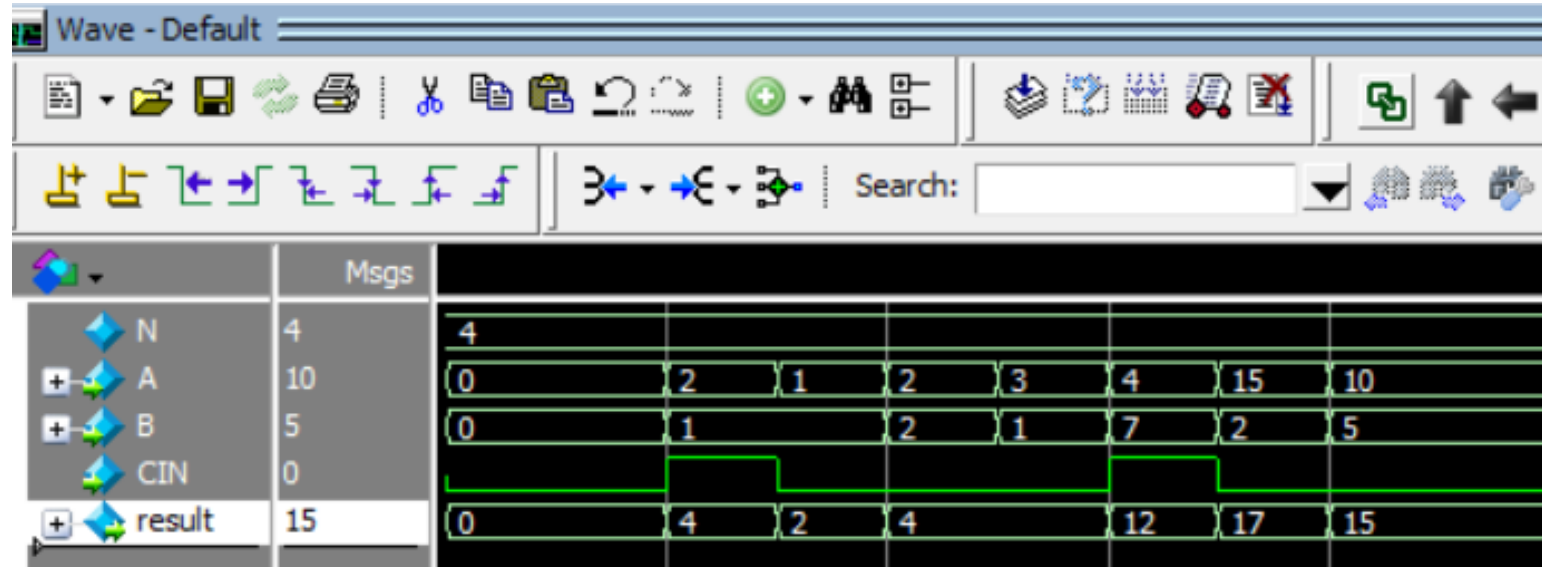
fulladder fa_inst2(
.a(A[2]),
.b(B[2]),
.cin(l_carry[2]),
.sum(l_sum[2]),
.cout(l_carry[3]);

fulladder fa_inst3(
.a(A[3]),
.b(B[3]),
.cin(l_carry[3]),
.sum(l_sum[3]),
.cout(l_carry[4]);

Ripple Carry Adder Post Synthesis Netlist



Ripple Carry Adder Simulation Result

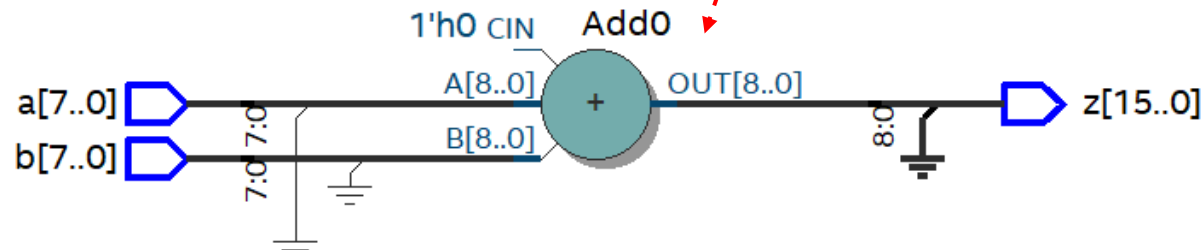


Generate if conditional Example

```
module generate_if_conditional #(parameter OP_TYPE = 0)(  
  input logic [7:0] a, b,  
  output logic [15:0] z);  
  
  generate  
    if(OP_TYPE == 0) begin  
      assign z = a + b;  
    end  
    else if(OP_TYPE == 1) begin  
      assign z = a - b;  
    end  
    else if(OP_TYPE == 2) begin  
      assign z = a ^ b;  
    end  
    else begin  
      assign z = a << 1;  
    end  
  endgenerate  
endmodule: generate_if_conditional
```

Based on OP_TYPE value set at elaboration time, only one the logic within if-else will be generated by Synthesis tool.

For OP_TYPE=0, synthesizer generated half adder logic as shown below



clock divide by 4 (Implementation-1)

```
module clock_divide_by_4 #(parameter N = 4)(  
  input logic clk, reset,  
  output logic clkout);  
  
  logic [$clog2(N)-1:0] cnt_value;  
  always_ff@(posedge clk or posedge reset)  
  begin  
    if (reset == 1) begin  
      cnt_value <= 0;  
      clkout <= 0;  
    end  
    else if(cnt_value == $clog2(N)-1) begin  
      cnt_value <= 0;  
      clkout <= ~clkout;  
    end  
    else  
      cnt_value <= cnt_value + 1;  
    end  
  end  
endmodule: clock_divide_by_4
```

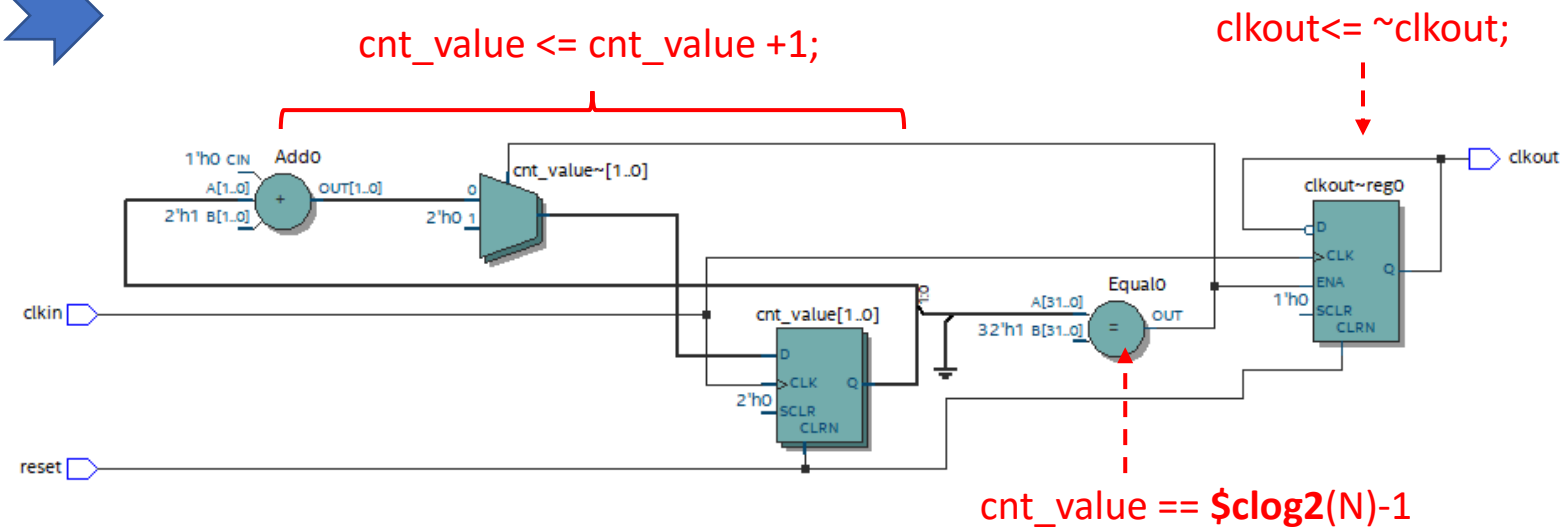
$\$clog2$ function returns the ceiling of the logarithm to the base 2.

Example :

for N=4, $\$clog2(4)$ will return 2

for N=6, $\$clog2(4)$ will return 3

For N=8, $\$clog2(4)$ will return 3



Question : If parameter N value is changed in SystemVerilog code to value 8, would above mentioned circuit work as divide by 8 ?

clock divide by 4 Example

