Lecture 11: RTL Programming Statements II

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

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

Recap

SystemVerilog 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: Generate

Generate Statements

- SystemVerilog provides generate statement to create multiple instantiations of module or code within module

- There are two types of generate statements :
  1. generate conditionals (decision)
     - generate if-else
     - generate case
  2. generate for loops (looping)

generate if-else syntax

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

Generate Statement Rules

- 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

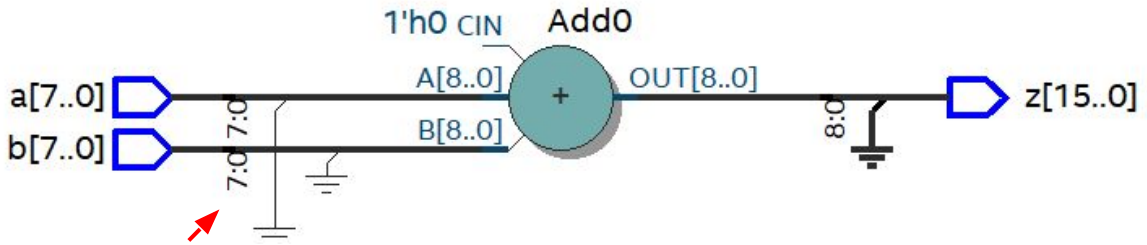
- Items that are not permitted in a generate statement include:
  - port declarations
  - constant declarations
  - specify blocks

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

For OP_TYPE=0, synthesizer generates adder logic as shown here



Looping Statements (generate for, repeat, while, do-while, etc.)

genvar and generate for

- **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 for loop syntax

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

Generate OR NOT Generate

```
module A();
..
endmodule;

module B();
      parameter NUM_OF_A_MODULES = 2; // should be overriden from higher hierarchy
      genvar i;
      for (i=0 i<NUM_OF_A_MODULES; i=i+1)
      { A A_inst(); }
endmodule;
```

- `generate for` loops let you access (using hierarchical names) the internal signals of the **instances generated**. The regular `for` loop doesn't let you do that.
- One cannot use regular for loop here because we are changing the generation variable every time, and can change the parameter.

Simulation speed

```
// faster :: 1 always block, simulator can optimize the for loop
always @(posedge sysclk) begin
  for (i = 0; i < 3 ; i = i + 1) begin
    temp[i] <= 1'b0;
  end
end

// slower :: creates 4 always blocks, harder for the simulator to optimize
genvar i;
generate // optional if > *-2001
for (i = 0; i < 3 ; i = i + 1) begin
    always @(posedge sysclk) begin
        temp[i] <= 1'b0;
    end
end
endgenerate // match generate
```

repeat Loops

- A **repeat** loop will 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**
  - In **synthesizable** code, **repeat** loops should only be used to expand replicated code
  - More often, **repeat** loops are used in **testbenches**
  - **repeat** loop's index can never be used inside the loop

- **Syntax :**

  repeat(<*iteration_index*>) begin

  <one or more statements>

  end

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

Same as

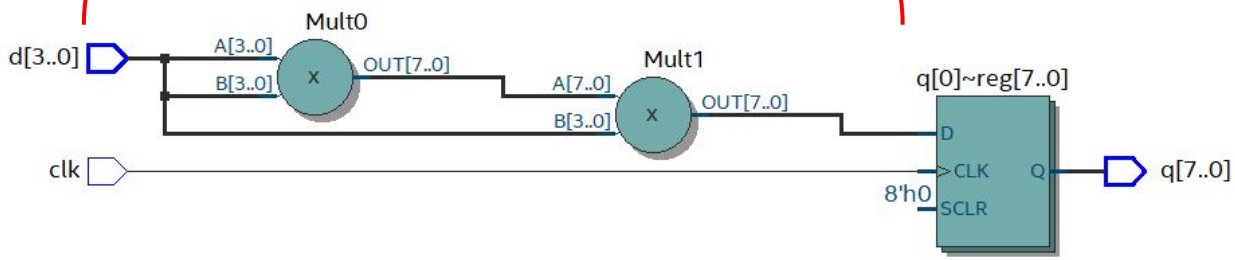
```
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
#10 clock = !clock;
```

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



while Loops

- 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
  - **while** loops are typically used in **testbench** code (non-synthesizable); in synthesizable code, **for** and **repeat** loops are more preferred

- **Syntax:**

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

Count ones using while 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;

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

endmodule: count_ones
```

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 be executed **at least once** when the loop is first entered
  - **do-while** loops are typically used in **testbench** code (non-synthesizable)

- **Syntax:**

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

Print count_value at least once and until (count_value < 16)

```
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
```

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 is mostly used in testbench code

- Syntax:

```
forever begin
   <one or more statements>
end
```

Forever loop to generate posedge and negedge of clock every 10 timeunits

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

  initial
    begin
      forever
        #10 clock = !clock;
    end

endmodule: clock_gen_testbench`
```

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)
  - Not synthesizable and is mostly used in testbench code

- Syntax:

```
foreach(<variable>[<iteration_index>]) begin
   <one or more statements>
end
```

Foreach loop example to initialize a memory array

```
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
```

Simulation Output:

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

Tasks

Tasks

- Task encapsulates one ore more programming statements which can be called from different parts 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
```

Task Rules

- Tasks can:
  - be static or automatic
  - contain both non-blocking and blocking assignment statements
  - contain any time controlled statements such as **#**, **@**, or **wait, posedge, negedge**, etc
  - call another tasks and functions
  - have zero or multiple inputs and outputs specified in its argument list
  - take, drive and source global variables, when no local variables are used
  - be used in both synthesizable and non-synthesizable code

Tasks vs. functions

Functions

- Cannot contain time-controlling statements (functions execute in ”zero time”)
- Can call other functions but cannot call a task
- Must have at least one input argument
- Returns a single value
- Allows blocking statements only

Tasks

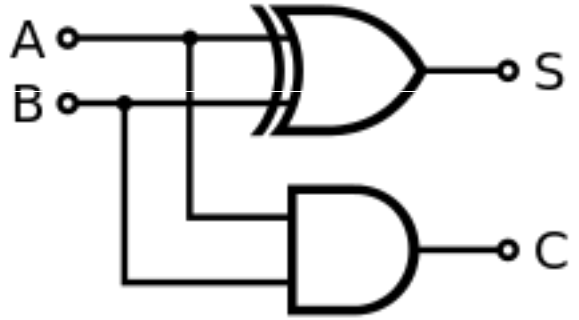
- Can contain time-controlling statements
- Can call other tasks and functions
- Arguments are optional
- Does not return values but can affect multiple values
- Allows blocking and non-blocking statements

Adders

1-bit Half and Full Adder

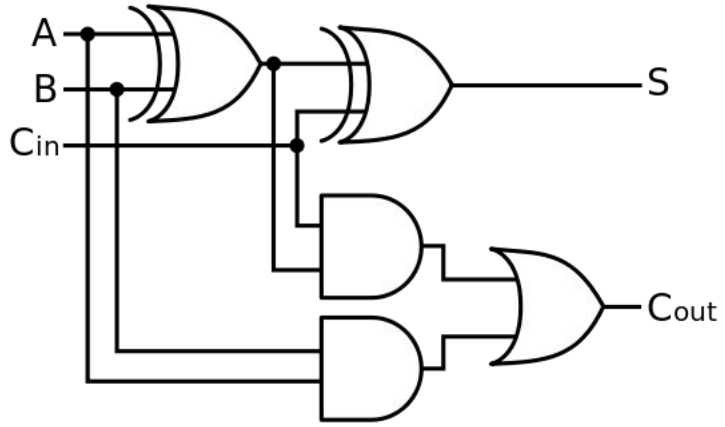
Half adder

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Full adder

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

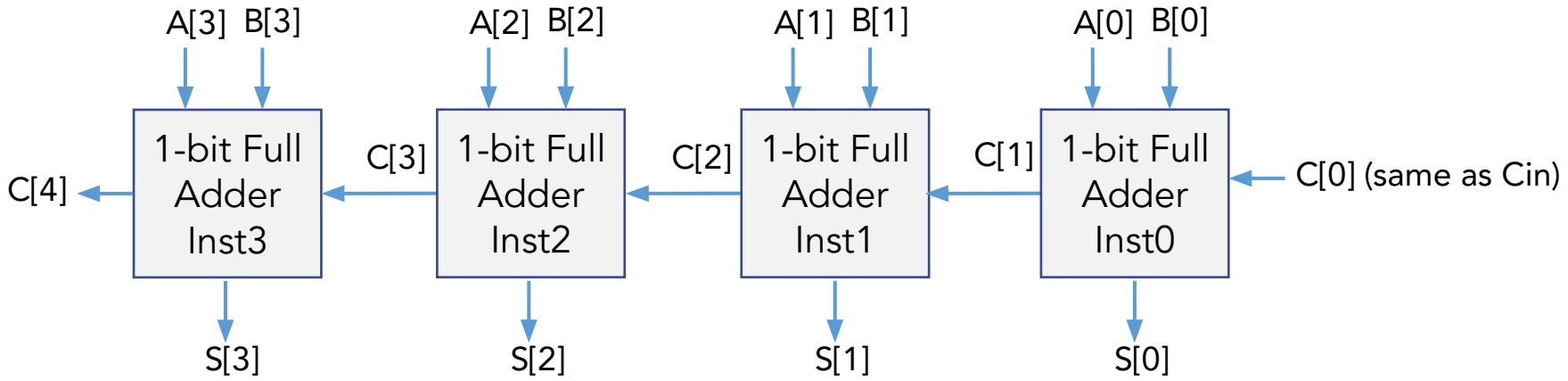


Wikipedia

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.

4-bit Ripple Carry Adder:



Ripple Carry Adder using Generate Statements

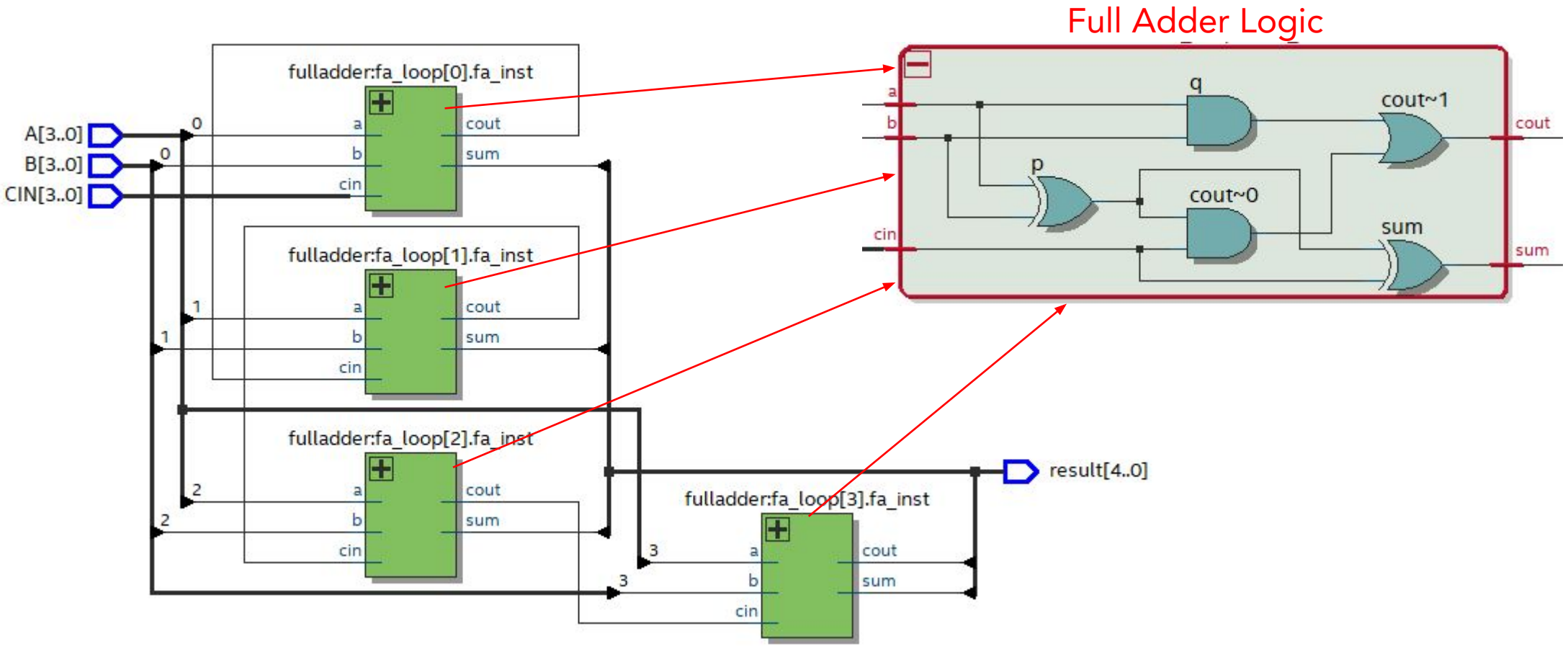
```
module ripple_carry_adder  #(parameter N = 4)(
  input logic[N-1:0] A, B,
  input logic CIN,
  output logic[N:0] result);
  logic[N:0] l_carry;
  logic[N-1:0] l_sum;
  // assign Carry in to first full adder carryin
  assign l_carry[0] = CIN;

  // Instantiate Full Adder for 'N' instances
  genvar i;
  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
```

genvar 'i' controls generate for loop iteration

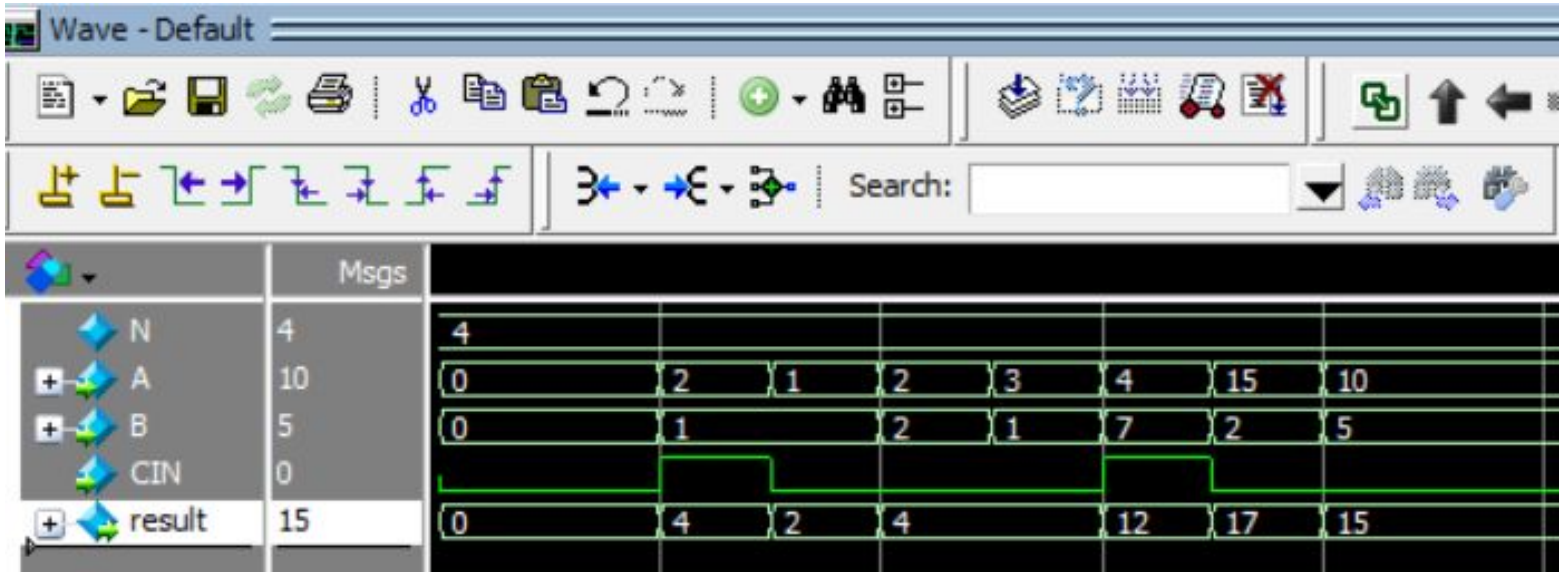
generate for loop statement will create 'N' number of full adder module instances

```
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



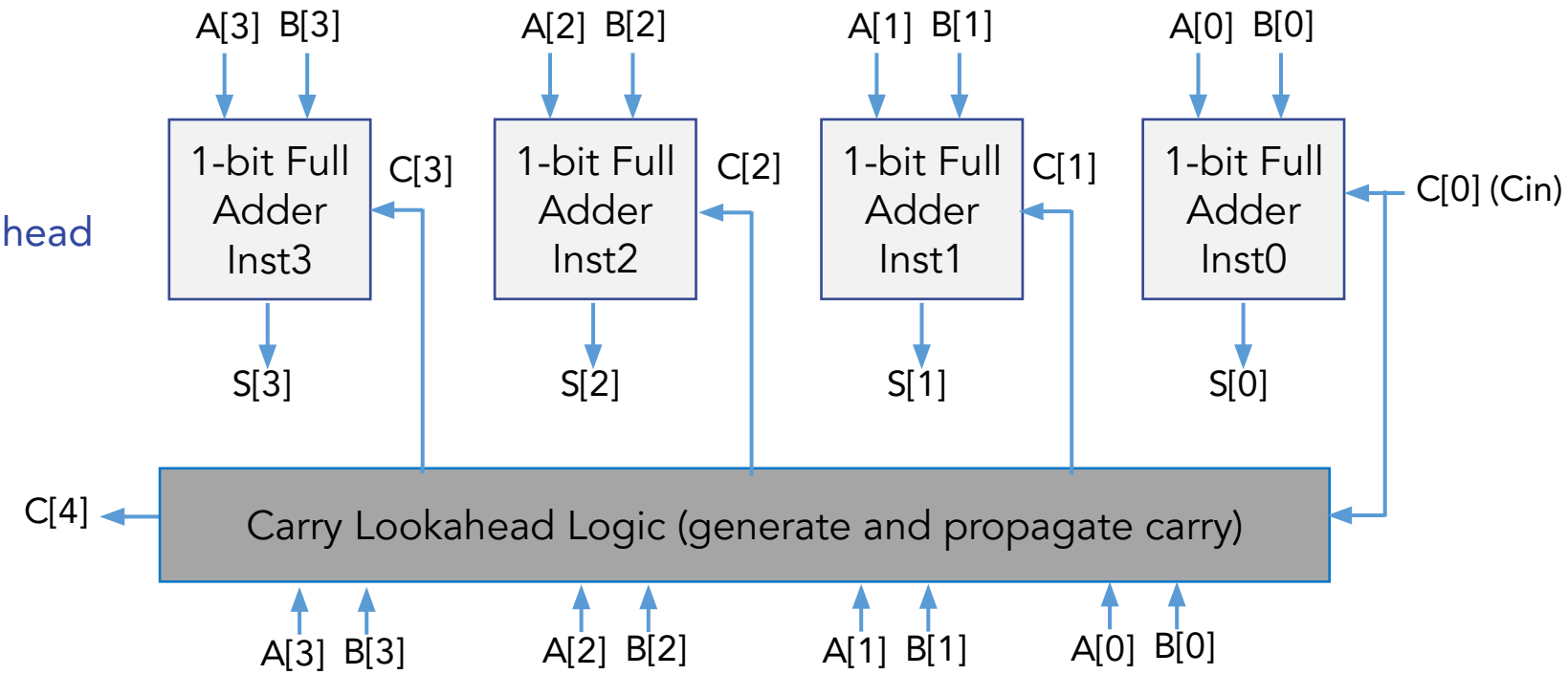
Ripple-Carry Adder Simulation Result



Carry Lookahead Adders (CLAs)

- A Carry Lookahead Adder is also made of a number of **full adders** but the **carry bit** for each full adder instance is **computed before** the full adder operation on inputs is performed
- Advantage: Faster than Ripple Carry Adder
- Disadvantage: More logic gates
- Many variants: Manchester, Kogge–Stone, and Brent–Kung

4-bit Carry Lookahead Adder:



Carry Lookahead Adder (CLA) Boolean Expressions

- **Carry Generate : G(i) = (A[i].B[i])** // Note : & is represented as .
- **Carry Propagate : P(i) = (A[i]+B[i])** // Note : | is represented as +
- **Generate and Propagate Carry : C[i+1] = G[i] + P[i].C[i]**
- **Carry inputs for each Full Adder Instance equations can be computed as follows:**

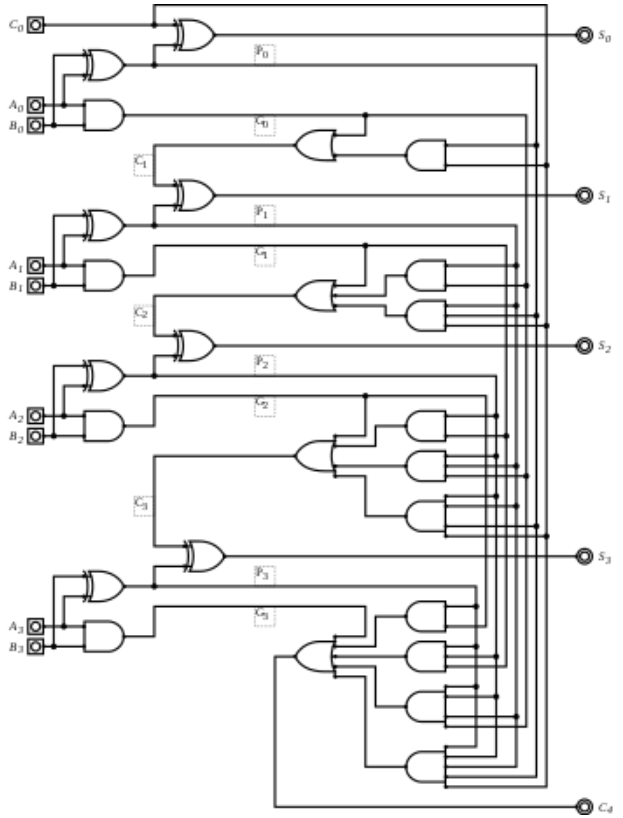
  C[1] = G[0] + P[0].C[0]
  = G[0] + P[0].Cin // Note : Replaced C[0] with Cin. C[0] is the input carry to first stage full adder

  C[2] = G[1] + P[1].C[1] // Note : C[1] can be replaced with G[0] + P[0].Cin
  = G[1] + P[1] (G[0] + P[0].Cin)
  = G[1] + P[1].G[0] + P[1].P[0].Cin

  C[3] = G[2] + P[2].C[2] // Note : C[2] can be replaced with G[1] + P[1].G[0] + P[1].P[0].Cin
  = G[2] + P[2] (G[1] + P[1].G[0] + P[1].P[0].Cin)
  = G[2] + P[2].G[1] + P[2].P[1].G[0] + P[2].P[1].P[0].Cin

  C[4] = G[3] + P[3].C[3] // Note : C[3] can be replaced with G[2] + P[2].G[1] + P[2].P[1].G[0] + P[2].P[1].P[0].Cin
  = G[3] + P[3] (G[2] + P[2].G[1] + P[2].P[1].G[0] + P[2].P[1].P[0].Cin)
  = G[3] + P[3].G[2] + P[3].P[2].G[1] + P[3].P[2].P[1].G[0] + P[3].P[2].P[1].P[0].Cin

4-bit Carry Lookahead Adder:



CLA Implementation Hint

```
generate for i=0 to i<N (where N = 4)
  fulladder fa_inst(
       ....
       ....
       ....
       .cout());  //empty parenthesis () after cout, means unconnected cout
  end: fa_loop


  // assign Carry in to first full adder carryin
  assign l_carry[0] = CIN;

  // generate carry : G(i)=A(i).B(i)
  // propogate carry : P(i)=A(i)+B(i)
  // Carry out: C(i+1)=G(i)+P(i).C(i)
  genvar j;
  generate
    for (j=0; j<N; j=j+1)
      begin : carry_gen_loop
        ....
        ....
        ....
      end : carry_gen_loop
  endgenerate

  // final result is concatenation of Final stage carry c[4] and sum S[3:0]
  assign result = {......};
```

// Note : . can be represented as &
// Note : + can be represented as |

Note : continuous assign statements are allowed within generate-for begin end body

Quiz

In this example, we create an N-bit shift register where each bit is a separate flip-flop. The `generate for` loop is used to instantiate these flip-flops. Explain why this could have not been possible with regular “for”

```
module shift_register #(
    parameter N = 8             // Number of bits in the shift register
) (
    input logic clk, input logic reset, input logic data_in, output logic [N-1:0] q_out
);

    // Generate block to create a chain of N flip-flops
    genvar i;
    generate
        for (i = 0; i < N; i++) begin : shift_chain
            if (i == 0) begin
                // First flip-flop, takes data_in as input
                always_ff @(posedge clk or posedge reset) begin
                    if (reset)
                        q_out[i] <= 1'b0;
                    else
                        q_out[i] <= data_in;
                end
            end
            else begin
                // Subsequent flip-flops, take the previous flip-flop's output as input
                always_ff @(posedge clk or posedge reset) begin
                    if (reset)
                        q_out[i] <= 1'b0;
                    else
                        q_out[i] <= q_out[i-1];
                end
            end
        end
    endgenerate

endmodule
```