CS150 - EE141/241A
Fall 2014
Digital Design and Integrated Circuits

Instructors:
John Wawrzynek and Vladimir Stojanovic

Lecture 7

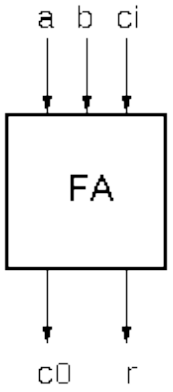
Outline

- Circuit Generators in Verilog
- Introduction to Logic Synthesis
- More on “always” blocks

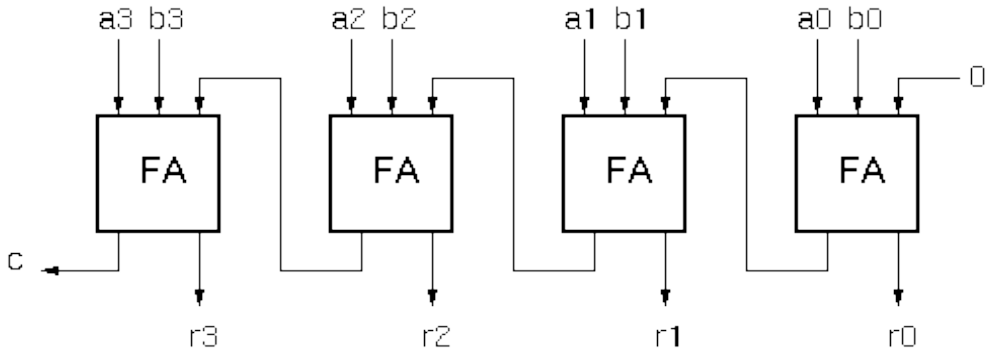
Verilog Circuit Generators

Review - Ripple Adder Example

```
module FullAdder(a, b, ci, r, co);
   input a, b, ci;
   output r, co;

   assign r = a ^ b ^ ci;
  assign co = a&ci + a&b + b&cin;

endmodule
```




```
module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;
  FullAdder
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),   .r(R[0]) ),
  add1(.a(A[1]), .b(B[1]), .ci(c1),   .co(c2),   .r(R[1]) ),
  add2(.a(A[2]), .b(B[2]), .ci(c2),   .co(c3),   .r(R[2]) ),
  add3(.a(A[3]), .b(B[3]), .ci(c3),   .co(R[4]), .r(R[3]) );
endmodule
```



Example - Ripple Adder Generator

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
module Adder(A, B, R);
  parameter N = 4;
  input [N-1:0] A;
  input [N-1:0] B;
  output [N:0] R;
  wire [N:0] C;

  genvar i;

  generate
     for (i=0; i<N; i=i+1) begin:bit
        FullAdder add(.a(A[i]), .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
    end
  endgenerate

  assign C[0] = 1'b0;
  assign R[N] = C[N];
endmodule
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

Replace all occurrences of "4" with "N".

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

```
Adder adder4 ( ... );

Adder #(.N(64))
adder64 ( ... );
```

Overwrite parameter N at instantiation.

More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;

  genvar i;

  generate for (i=0; i<SIZE; i=i+1) begin:bit
    assign bin[i] = ^gray[SIZE-1:i];
  end endgenerate
endmodule
```

variable exists only in the specification - not in the final circuit.

Keywords that denotes synthesis-time operations

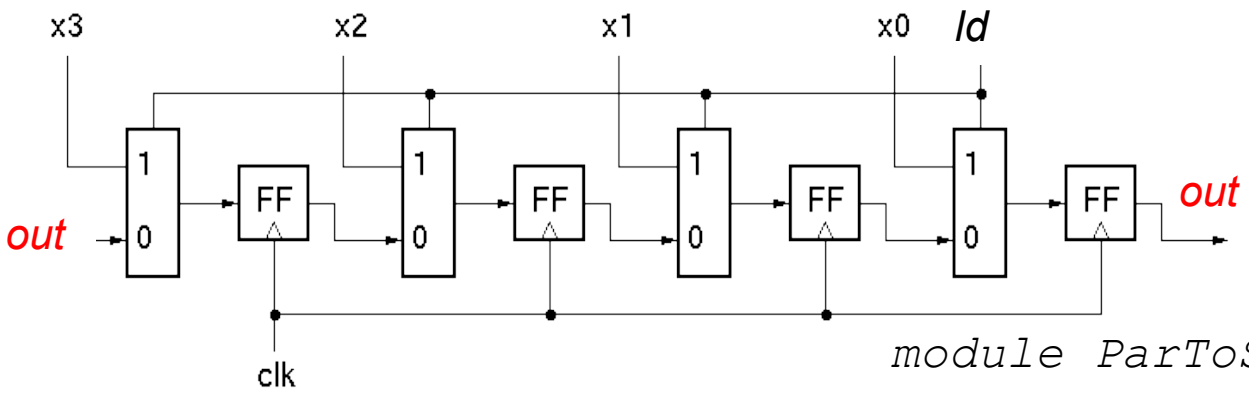
For-loop creates instances of assignments

Loop must have constant bounds

generate if-else-if based on an expression that is deterministic at the time the design is synthesized.

generate case : selecting case expression must be deterministic at the time the design is synthesized.

Example - Parallel to Serial Converter




```
module ParToSer(ld, X, out, clk);
   input [3:0] X;
   input ld, clk;
   output out;

   reg [3:0] Q;
   wire [3:0] NS;

   assign NS =
     (ld) ? X : {Q[0], Q[3:1]};

   always @ (posedge clk)
      Q <= NS;

   assign out = Q[0];
endmodule
```

Specifies the muxing with "rotation"

forces Q register (flip-flops) to be rewritten every cycle

connect output

Parameterized Version

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
parameter N = 4;
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

```
ParToSer #(.N(8))
   ps8 ( ... );

ParToSer #(.N(64))
ps64 ( ... );
```

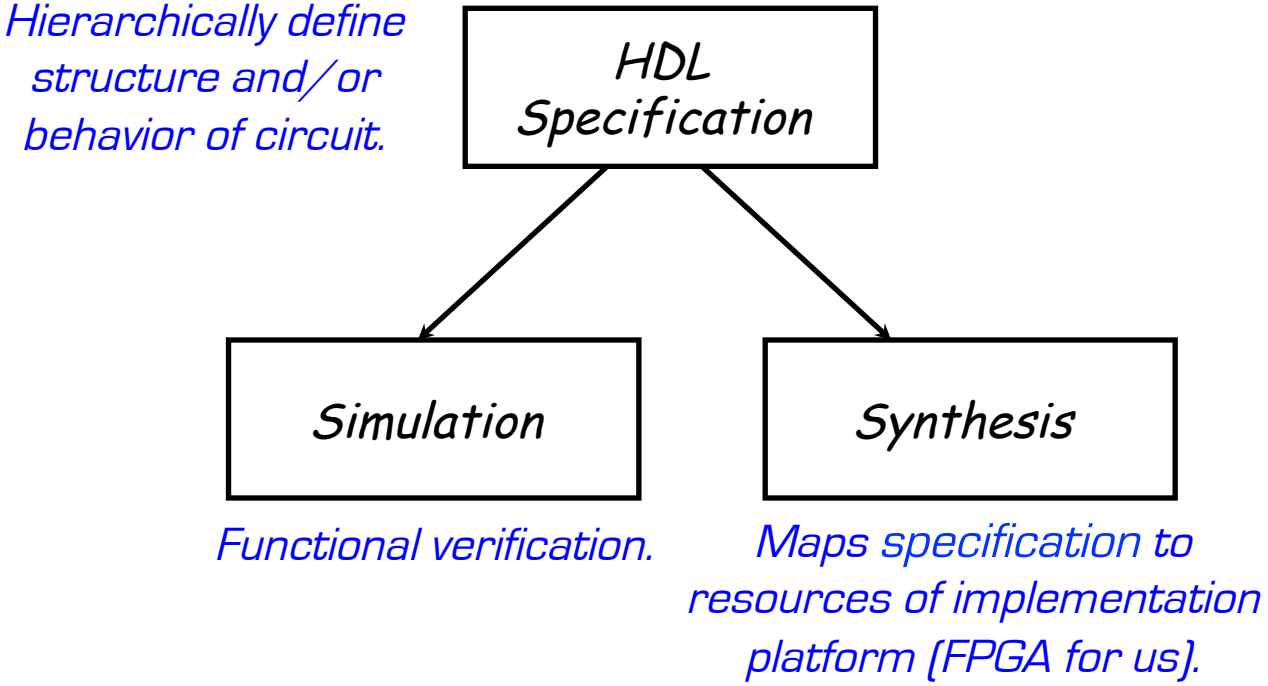
Overwrite parameter N at instantiation.

```
module ParToSer(ld, X, out, CLK);
  input [N-1:0] X;
  input ld, clk;
  output out;
  reg out;
  reg [N-1:0] Q;
  wire [N-1:0] NS;

  assign NS =
    (ld) ? X : {Q[0], Q[N-1:1]};

  always @ (posedge clk)
     Q <= NS;

  assign out = Q[0];
endmodule
```

Replace all occurrences of "3" with "N-1".

Intro to Logic Synthesis

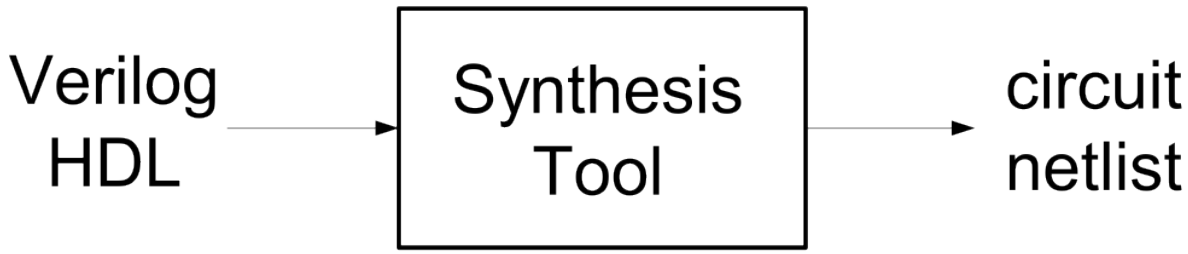
EECS150 Design Methodology



Note: This in not the entire story. Other tools are often used analyze HDL specifications and synthesis results. More on this later.

Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).



- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

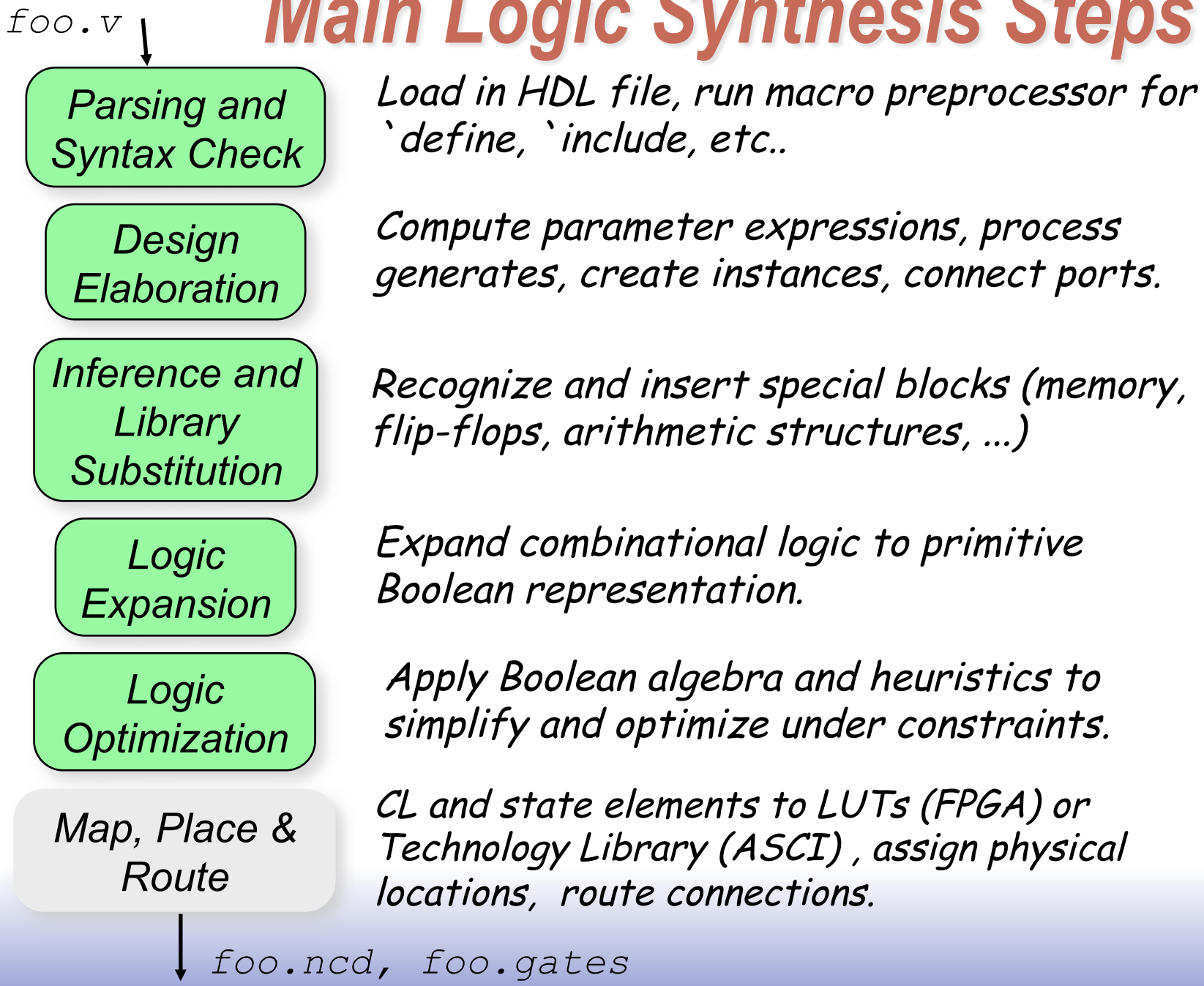
Why Logic Synthesis?

1. Automatically manages many details of the design process:
   - ⇒ Fewer bugs
   - ⇒ Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

Why Not Logic Synthesis?

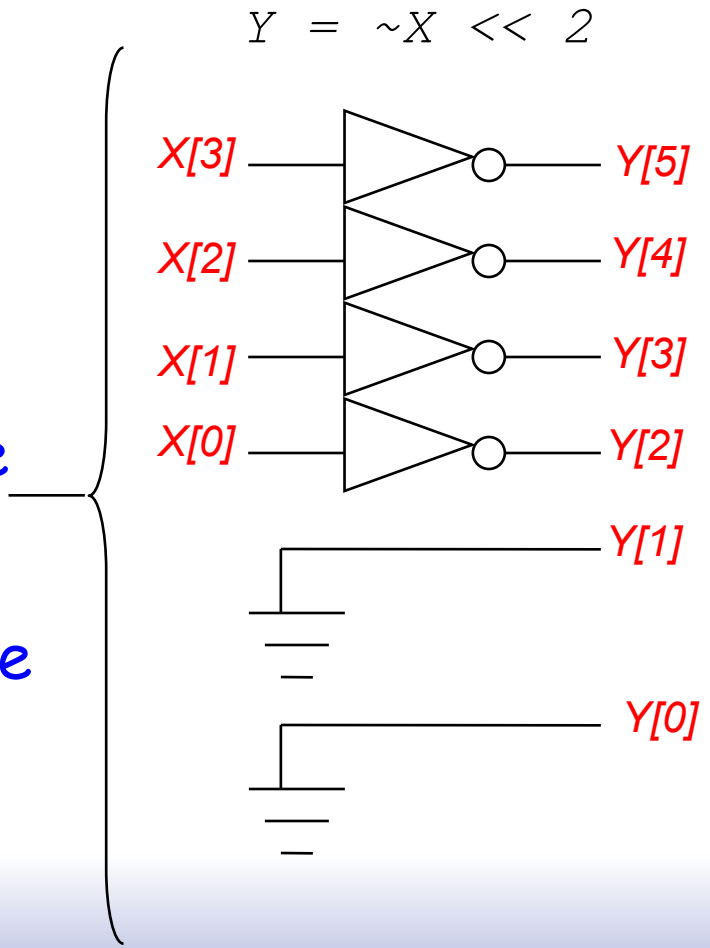
1. *May lead to non-optimal designs in some cases.*

Main Logic Synthesis Steps



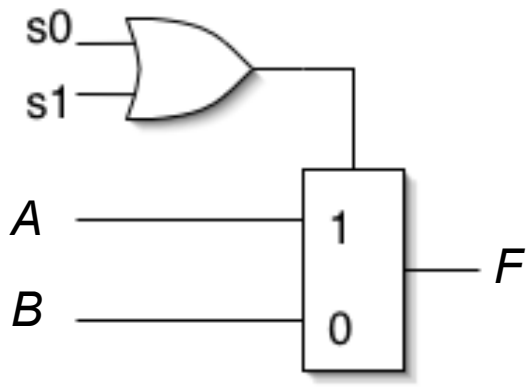
Operators and Synthesis

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts, a whole different story --- shifter
- Conditional expression generates logic or MUX



Simple Synthesis Example

```
module foo (A, B, s0, s1, F);
  input [3:0] A;
  input [3:0] B;
  input s0,s1;
  output [3:0] F;
  reg F;
  always @ (*)
    if (!s0 && s1 || s0) F=A; else F=B;
endmodule
```



Should expand if-else into 4-bit wide multiplexor and optimize the control logic and ultimately to a single LUT on an FPGA:

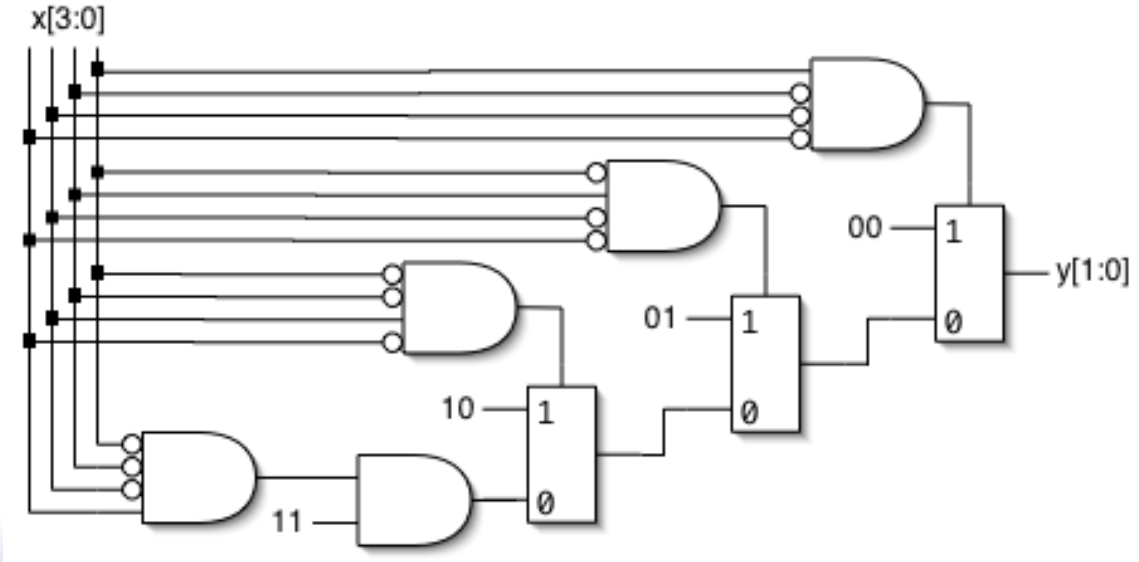
Encoder Example

Nested IF-ELSE might lead to "priority logic"

Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

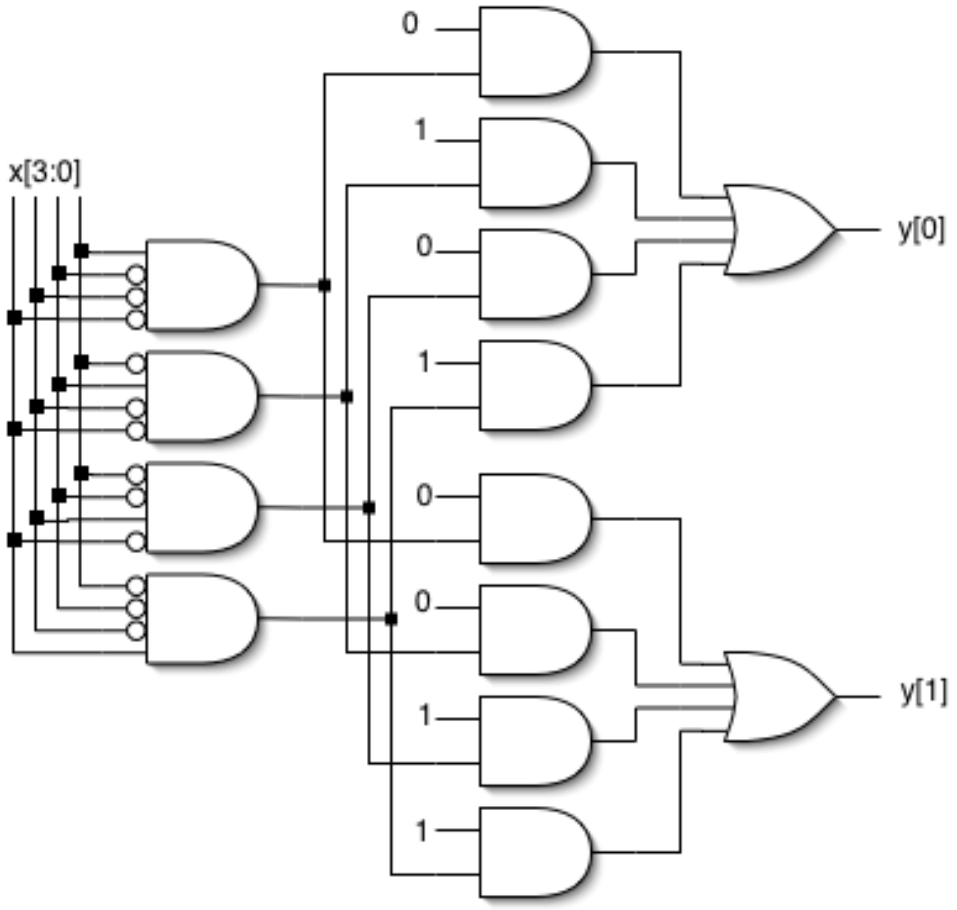
This style of cascaded logic may adversely affect the performance of the circuit.



Encoder Example (cont.)

To avoid "priority logic" use the case construct:

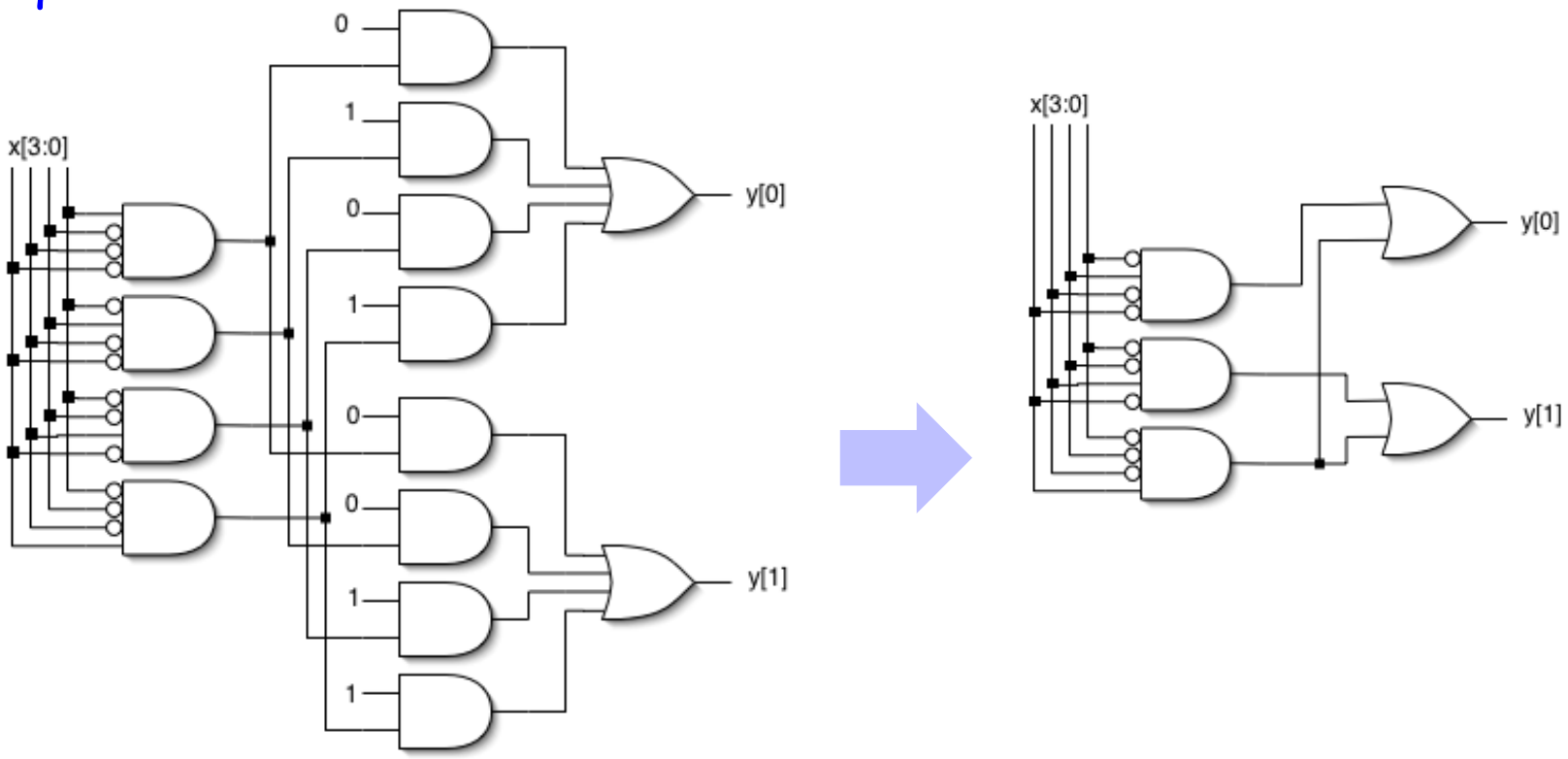
```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```



All cases are matched in parallel.

Encoder Example (cont.)

This circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities:



A similar simplification would be applied to the if-else version also.

Always blocks, again

Combinational logic always blocks

Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

Sel case value 2'd2 omitted.

Out is not updated when select line has 2'd2.

Latch is added by tool to hold the last value of out under this condition.

Similar problem with if-else statements.

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```

Combinational logic always blocks (cont.)

To avoid synthesizing a latch in this case, add the missing select line:

```
2'd2: out = c;
```

Or, in general, use the "default" case:

```
default:  out = foo;
```

If you don't care about the assignment in a case (for instance you know that it will never come up) then you can assign the value "x" to the variable. Example:

```
default:  out = 1'bx;
```

The x is treated as a "don't care" for synthesis and will simplify the logic.

Be careful when assigning x (don't care). If this case were to come up, then the synthesized circuit and simulation may differ.

Incomplete Triggers

Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

in2 not in always sensitivity list.

A latched version of in2 is synthesized and used as input to the and-gate, so that the and-gate output is not always sensitive to in2.

Easy way to avoid incomplete triggers for combinational logic is with: `always @*`

Procedural Assignments

Verilog has two types of assignments within always blocks:

- Blocking procedural assignment "="
  - In simulation the RHS is executed and the assignment is completed before the next statement is executed. Example:

    Assume A holds the value 1 … A=2; B=A; A is left with 2, B with 2.
- Non-blocking procedural assignment "<="
  - In simulation the RHS is executed and all assignment take place at the same time (end of the current time step - not clock cycle). Example:

    Assume A holds the value 1 … A<=2; B<=A; A is left with 2, B with 1.

- *In synthesis the difference shows up primarily when inferring state elements:*

```
always @ (posedge clk) begin
 a = in;    b = a;
end
```

b stores in

```
always @ (posedge clk) begin
 a <= in;   b<= a;
end
```

b stores the old a

Procedural Assignments

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

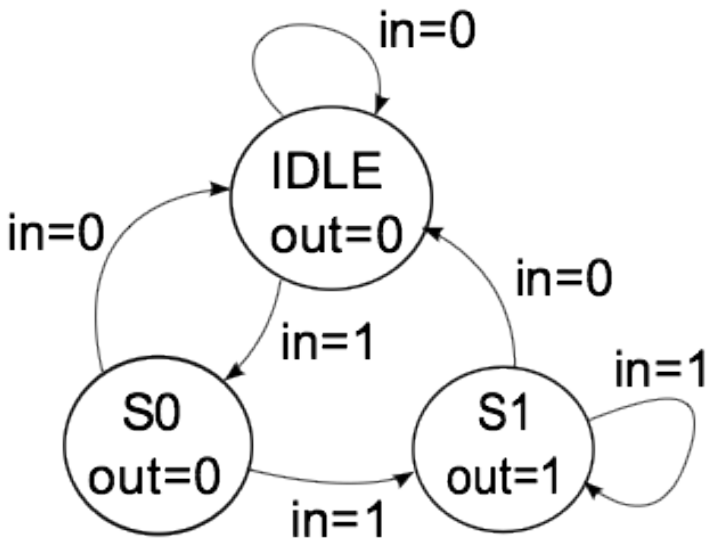
i. Use blocking assignments to model combinational logic within an always block (“=”).

ii. Use non-blocking assignments to implement sequential logic (“<=”).

iii. Do not mix blocking and non-blocking assignments in the same always block.

iv. Do not make assignments to the same variable from more than one always block.

FSM CL block rewritten




```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
    case (state)
     IDLE    : if (in == 1'b1) next_state = S0;
     S0      : if (in == 1'b1) next_state = S1;
     S1      : begin
                 out = 1'b1;
                 if (in == 1'b1) next_state = S1;
               end
     default: ;
    endcase
  end
endmodule
```

** for sensitivity list*

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.