

Verilog Coding

Module #3

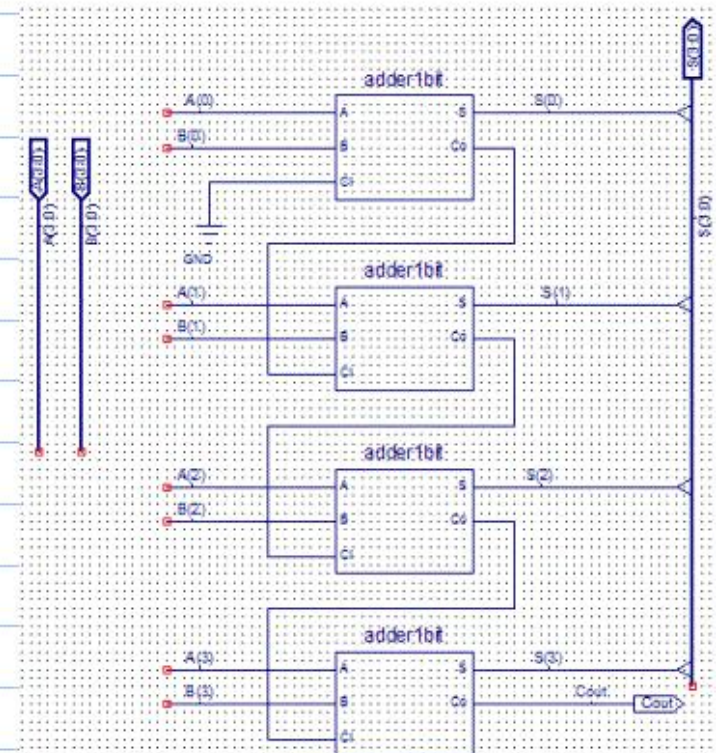
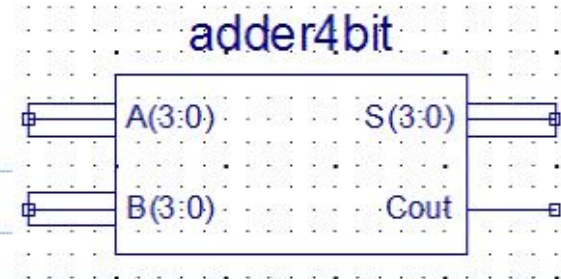
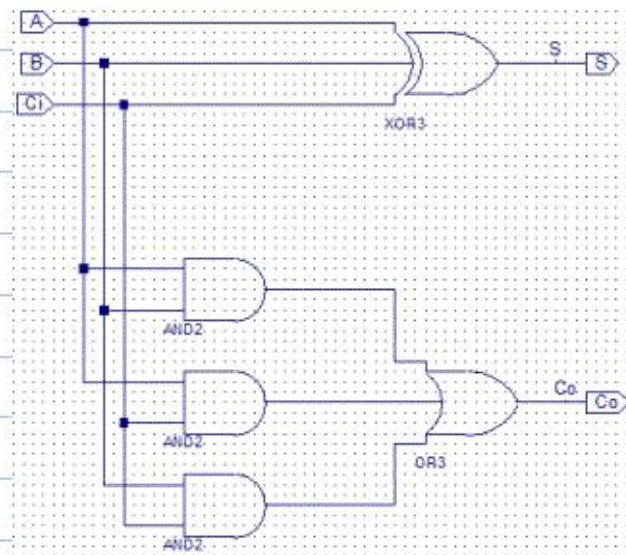
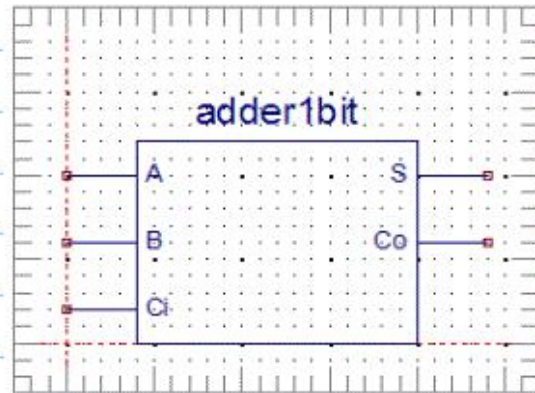
Structural vs. Behavioral coding

Examples: mux and parity generator

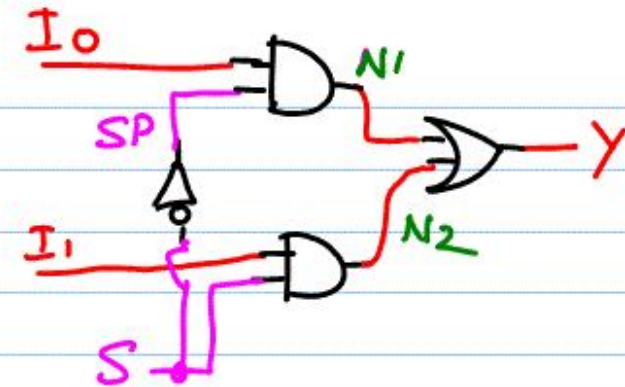
By
Gandhi Puvvada



Xilinx™ Schematic Entry Tutorial



Primitive Gate Instantiation



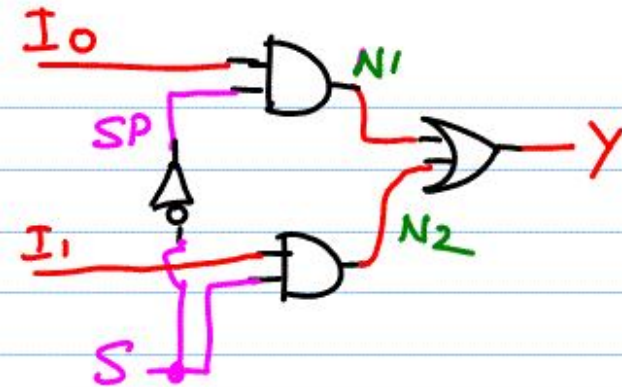
```

not gate1 (SP, S);
and gate2 (N1, I0, SP);
and gate3 (N2, I1, S);
or gate4 (Y, N1, N2);
  
```

wire N1, N2, SP;

assign (continuous)

Assignment Statements



assign $SP = \sim S;$

assign $N1 = I0 \& SP;$

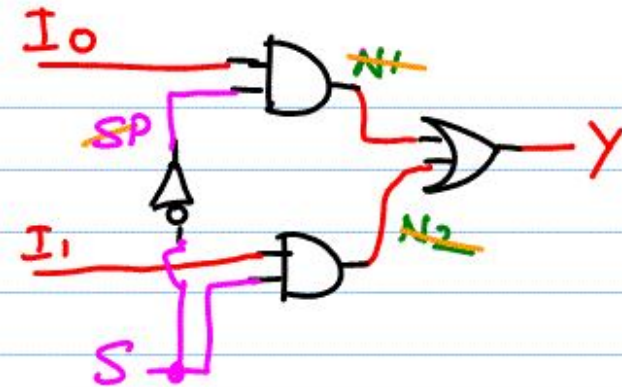
assign $N2 = I1 \& S;$

assign $Y = N1 \mid N2;$

wire $N1, N2, SP;$

assign (continuous)

Assignment Statement



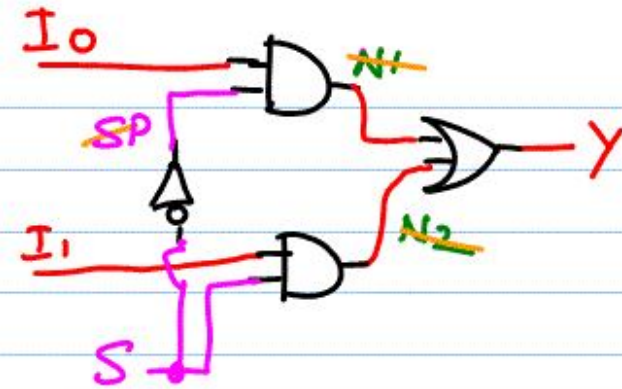
assign Y =

$$\left((I_0 \& (\sim S)) \mid \right. \\ \left. (I_1 \& S) \right) ;$$

~~wire N1, N2, SP;~~

assign (continuous)

Assignment Statement



assign $Y =$

$S ? I1 : I\emptyset ;$

~~wire N1, N2, SP ;~~

conditional operator $? :$

Sequential procedural **if** statement

Using **if** statement

```
req Y;
```

if (S)

$Y = I1;$

else

$Y = I0;$

if (S)

begin

$Y = I1;$

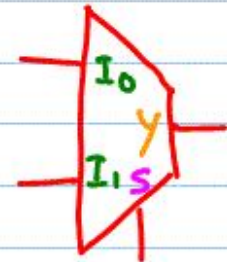
end

else

begin

$Y = I0;$

end



Sequential procedural **Case** statement

Using **Case** statement

reg Y;

Case (S)

1'b1: Y = I1;

1'b0: Y = I0;

endcase

Case(S)

1'b1:

begin

Y = I1;

end

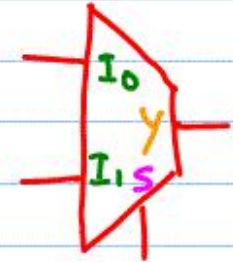
1'b0:

begin

Y = I0;

end

endcase



parity generator

Given 4-bit X
produce odd
parity

$X[3]$	$X[2]$	$X[1]$	$X[0]$	P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

$X[3]$	$X[2]$	$X[1]$	$X[0]$	P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

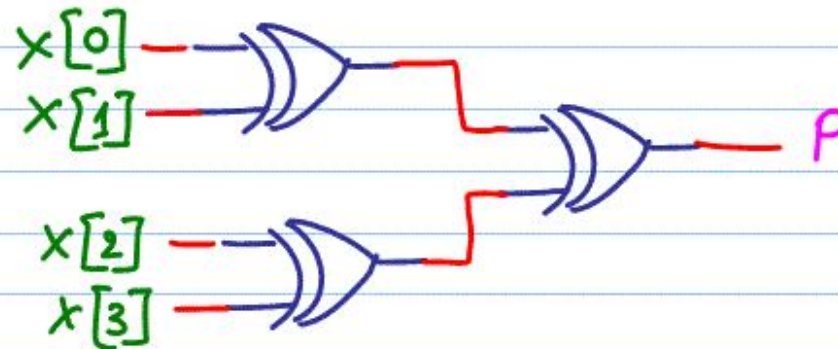
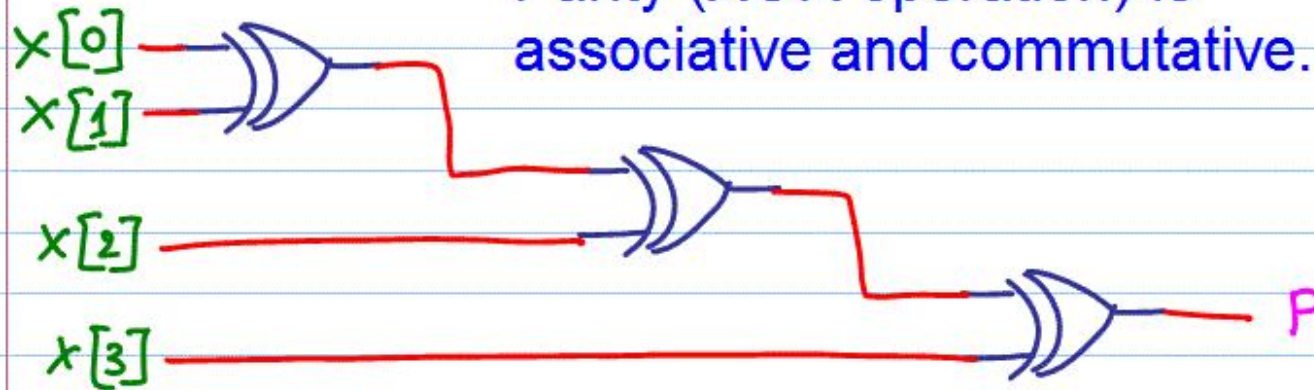
Case Statement with 16 case branches

$$2^8 = 256 \quad 2^{16} = 65,536$$

parity generator

Given 4-bit x
produce odd
parity

Parity (XOR operation) is
associative and commutative.

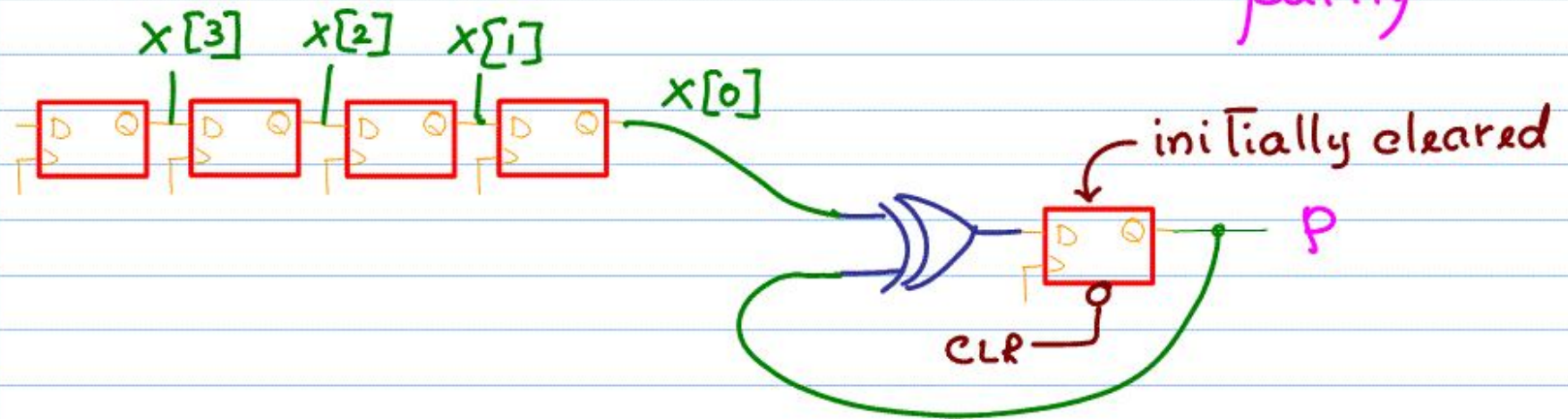


For 4-bit you
can design
at gate-level.

But if it is 64-bit
you want a better
method

parity generator

Given 4-bit x
produce odd
parity



SERIAL PARITY GENERATOR

parity generator

always @ (x)

begin

p = 1'b0;

for (i = 0; i <= 3; i = i + 1)

begin

p = p ^ x[i];

end

end

Given 4-bit x
produce odd
parity

integer i;

parity generator

always @ (x)

begin

$P = 1'b0;$

for ($i = 0; i \leq 3; i = i + 1$)

begin

$P = P \wedge x[i];$

end

end

Given 4-bit x
produce odd
parity

integer i;

Is this a combinational logic
or sequential logic?
Is "i" a counter?
Do all iterations take
place without
advancement
of time?

parity generator

always @ (x)

begin

ones_count = 3'b000;

for (i = 0; i <= 3; i = i + 1)

begin

if x[i] begin

ones_count = ones_count + 1;

end

end

p = ones_count[0];

end

Given 4-bit x
produce odd
parity

integer i;

reg [2:0] ones_count;

parity generator

Given 4-bit X
produce odd
parity

In the design on previous page

ones_count is
neither a **register**
nor a **counter**!

reg [2:0] ones_count;

ones_count = ones_count + 1;

Note: There no clock.

This is a combinational
logic!

Limit the usage of

- ① instantiations
- ② assign (continuous) statements

And start using

- ③ always blocks

To describe

Our design behaviorally.

Conclusion

Use **BEHAVIORAL CODING**
at **BLOCK** level
and **STRUCTURAL CODING**
at **TOP** level!

