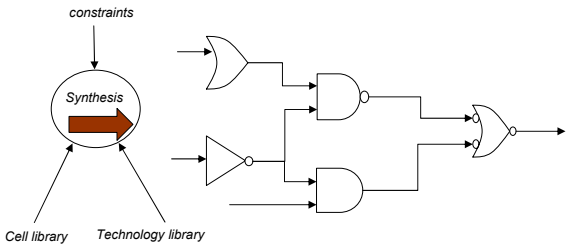
Modeling Complex Behaviors: Synthesis

VHDL Synthesis

VHDL Model

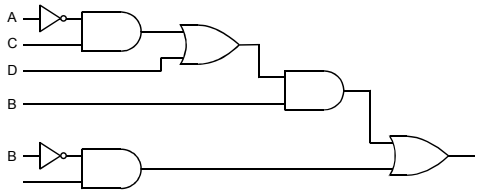
```
entity my_ckt is
port(x, y:in bit;
       z: out bit)
end entity my_ckt;
architecture behavioral of my_ckt is
begin
--
-- process code here
--
end architecture behavioral;
```



- Inferring hardware from sequential code in processes
  - Multiple alternative implementations
  - Metrics for selecting the right implementation
- Coding style can “help” the compiler

Language Directed View

- Inferring combinational logic
  - Every combination of inputs produces an output
  - Every execution path performs an assignment to every signal



- Inferring sequential logic
  - A signal is not assigned in some execution path

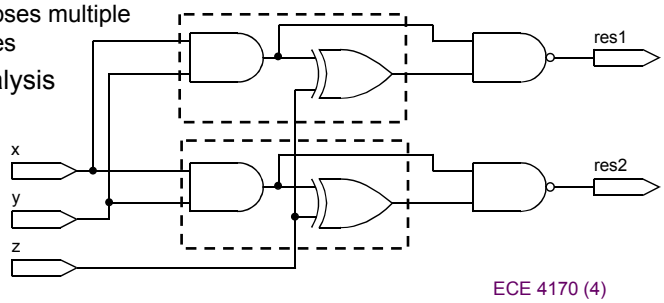
 remember the value, i.e., storage

Simple Assignment Statements

```
architecture behavior of stover is
signal sig_s1, sig_s2: std_logic;
begin
proc1: process (x, y, z) is -- Process 1 using variables
variable var_s1, var_s2: std_logic;
begin
var_s1:= x and y;
var_s2:= var_s1 xor z;
res1 <= var_s1 and var_s2;
end process proc1;
```

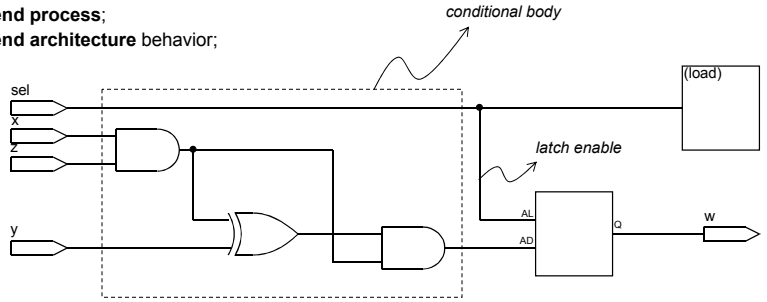
```
proc2: process (x, y, z) -- Process 2 using
signals

begin
sig_s1 <= x and y;
sig_s2 <= sig_s1 xor z;
res2 <= sig_s1 and sig_s2;
end process proc2;

end architecture behavior;
```

- Simulation mismatch
  - synthesis collapses multiple simulation cycles
- Dependence analysis



Conditional Statements

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
if (sel = '1')then
s1:= x and z;
s2:= s1 xor y;
w <= s2 and s1;          -- w gets a value only conditionally
end if;                  -- hence a latch is inferred
end process;
end architecture behavior;
```



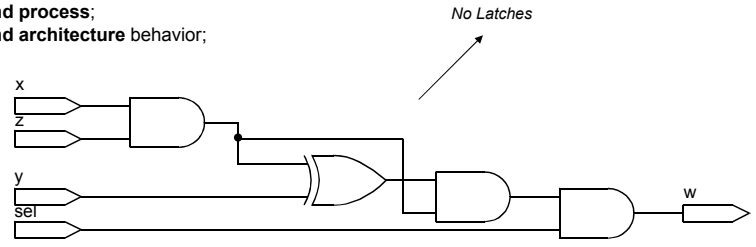
Avoiding Latch Inference

- Ensure every path computes a value for every signal
  - Presence of the else branch
  - Nested structures
  - if-then-elsif structure
- Use initial values
- Basic principle: ensure that every combination of input signal values leads to a computation of a value for every output signal

  ➡ combinational logic

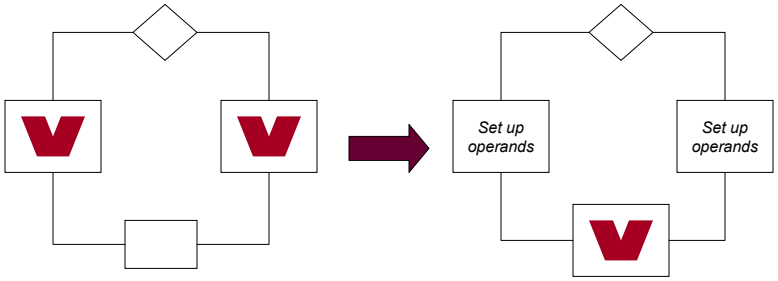
Revisit the Example

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
w <= '0';      -- output signal set to a default value to avoid latch inference
if (sel = '1')then
s1:= x and z;              -- body generates combinational logic
s2:= s1 xor y;
w <= s2 and s1;
 end if;

end process;
end architecture behavior;
```



Efficiency Considerations

- Now that we can control latch inferencing what about circuit size and speed?
- Move common operations (hardware) out of the branches
  - Good programming practice in general
  - Trade multiplexors for more expensive hardware

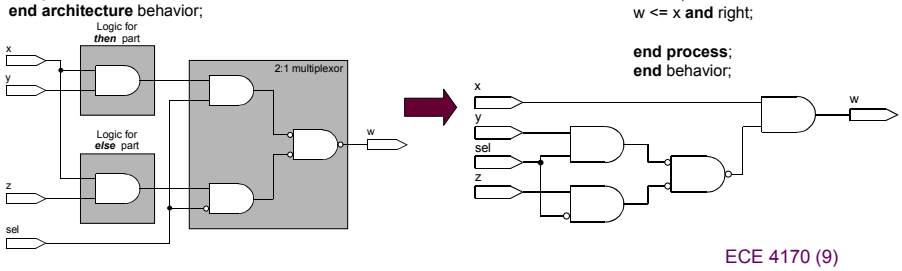


Efficiency Considerations

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inference is
port (sel : in std_logic;
x, y, z: in std_logic;
w: out std_logic );
end entity inference;

architecture behavior of inference is
begin
process (x, y, z, sel) is
begin
if (sel = '1')then
w <= x and y;
 else
w <= x and z;
 end if;
end process;
end architecture behavior;
```

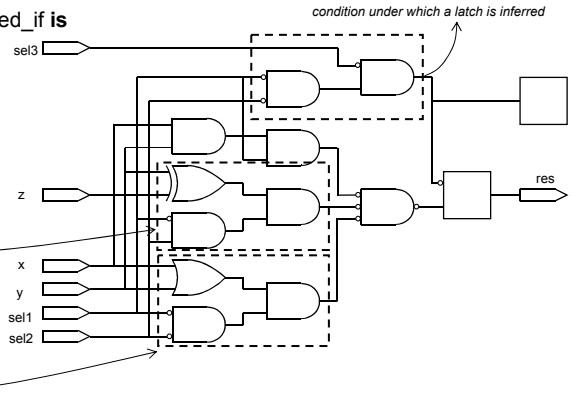
- Eliminated the latches
- Computation (logical AND) preceded by operand selection (mux)

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inference is
port (sel : in std_logic;
x, y, z: in std_logic;
w: out std_logic );
end inference;

architecture behavior of inference is
begin
process (x, y, z, sel)
variable right: std_logic;
begin
if (sel = '1')then
right:= y;
 else
right:= z;
 end if;
w <= x and right;

end process;
end behavior;
```



Nested Constructs

```
architecture behavior of nested_if is
begin
process (x,y,z,sel1,sel2,sel3)
begin
if (sel1 = '1')then
 res <= x and y;
elsif (sel2 ='1') then
res <= y xor z;
elsif (sel3 ='1') then
res <= x or y;
 end if;
end process;
```



- Latch inference due to the absence of the last "else"
- Latch inference due to placement in higher level blocks
- Priority ordering of the computations

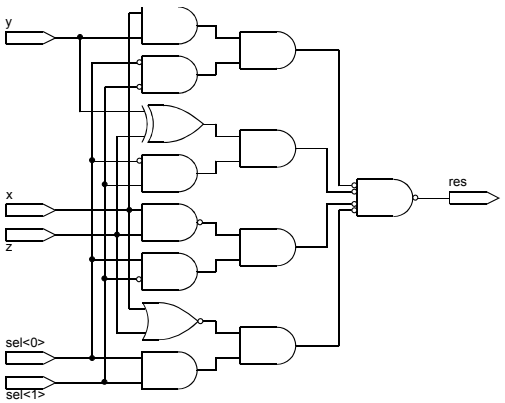
Comparison Logic and Effects

```
architecture behavior of stover is
begin
process (x, y, z) is
begin
if (sel = "– 0") then          -- the symbol "–" represents the don't care
w <= x nor y;                  -- for std_logic types
else
w <= x nor z;
end if;

end process;
end architecture behavior;
```

- Comparison always returns false - the "then" branch is never taken
- Synthesis returns the combinational logic for the "else" branch – a single NOR gate

CASE Statements

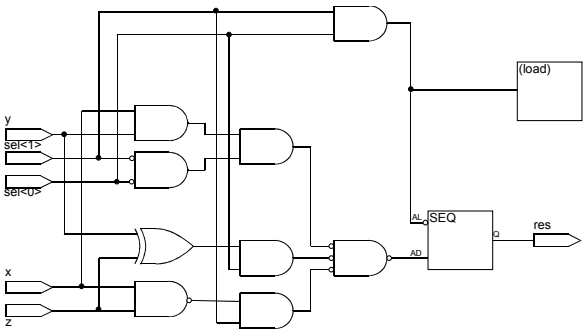
```
architecture behavior of case_st is
begin
process (x,y,z,sel) is
begin
case sel is
when 0 => res <= x and y;
when 1 => res <= y xor z;
when 2 => res <= x nand z;
when others => res <= x nor z;
end case;
end process;
end architecture behavior;
```



- Synthesis of a multiplexor
- Latch inference intuition applies
- Difference with nested *if-then-elsif* → latter synthesizes priority logic

CASE Statements (cont.)

```
architecture behavior of case_ex is
begin
process (x,y,z,sel) is
begin
case sel is
when 0 => res <= x and y;
when 1 => res <= y xor z;
when 2 => res <= x nand z;
when others => null;
-- in this case the value of res
-- remains unaltered
end case;
end process;
end architecture behavior;
```



- Use of the “null” statement and latch inference

Loop Statements

- How much hardware should be generated?
- Most commonly supported is the *for loop*
  - Number of iterations is known a priori
  - Loop is unrolled and optimized as a sequence of sequential statements

```
for N in 3 downto 1 loop
shift_reg (n) <= shift_reg (n-1);
end loop;
```

```
shift_reg (3) <= shift_reg (2);
shift_reg (2) <= shift_reg (1);
shift_reg (1) <= shift_reg (0);
```

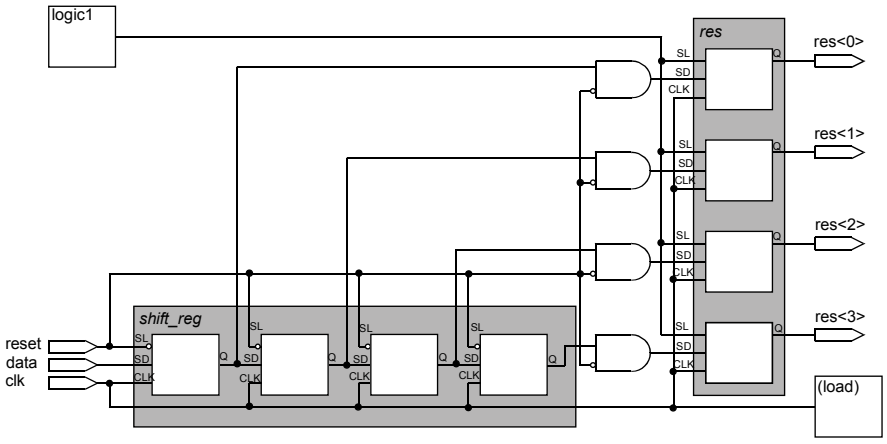
- Dependencies within and across iterations
  - Cross iteration dependencies synthesize to long signal paths
- Constraints on type of loop index

Loop Statements: Example

```
architecture behavior of iteration is
signal shift_reg : std_logic_vector(3 downto 0);
begin
process (clk, reset, data) is
        begin
        if (rising_edge(clk)) then
          if (reset = '1') then res <= "0000";
          else
                for n in 3 downto 1 loop
                        shift_reg(n) <= shift_reg(n-1);
                end loop;
                shift_reg(0) <= data;
                res <= shift_reg;
          end if;
        end if;
end process;
end architecture behavior;
```

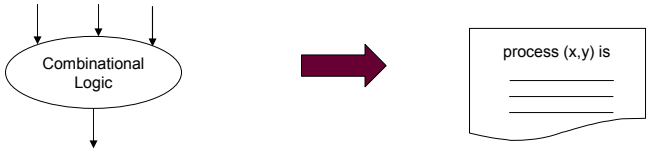
- Call to rising_edge(clk) produces edge triggered flip flops
- Loop leads to sequence of dependent assignment statements

Loop Statements: Example



Miscellaneous Issues

- Use of sensitivity lists: all signals are in the sensitivity list for synthesis



  - Pre-synthesis and post-synthesis simulation mismatches
- Optimizations involving variables
  - Eliminate intermediate variables
- Architecture level interpretation of inference rules
  - Latch inference occurs at outermost level of inferencing
- Use of *buffer* and *inout* modes.

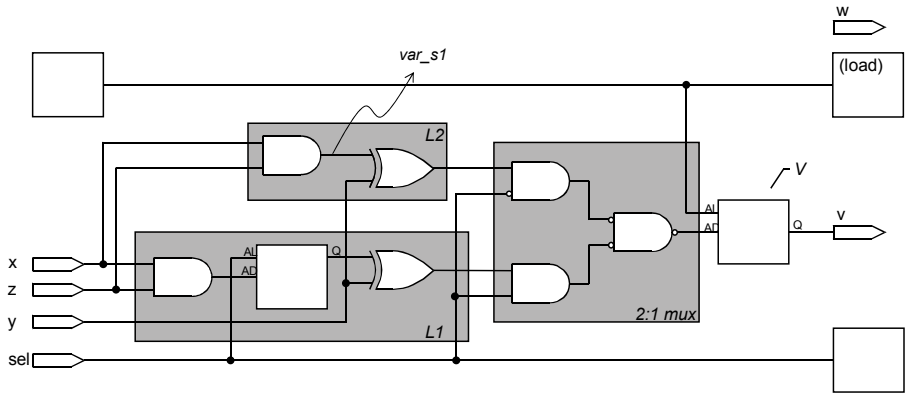
Inference Using Signals vs. Variables

```
library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
port (sel : in std_logic;
x, y, z: in std_logic;
v, w: out std_logic );
end entity sig_var;

architecture behavior of sig_var is
signal sig_s1 : std_logic;
begin
process (x, y, z, sel) is
variable var_s1: std_logic;
```

```
L1: if (sel = '1')then
sig_s1     <= x and z;
v          <= sig_s1 xor y;
end if;


L2: if (sel = '0') then
var_s1     := x and z;
v          <= var_s1 xor y;
end if;
end process;
end architecture behavior;
```

- Variable synthesized to a wire vs. signal synthesized to a latch
- Why is a latch inferred at all since all execution paths are covered?

Inference Using Signals vs. Variables

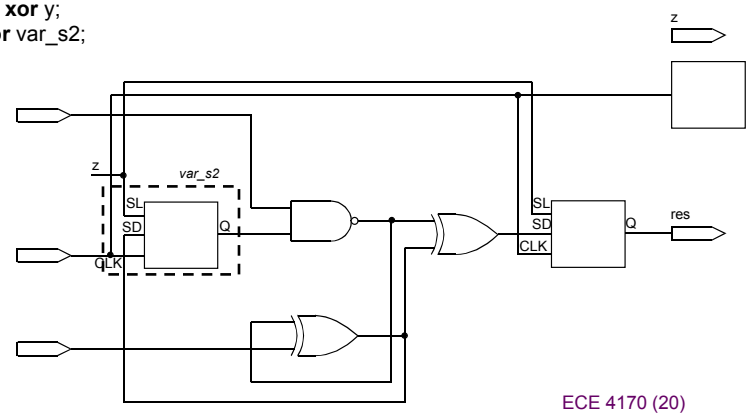


- Current value (variable) vs. previous value (signal)

Inferring Storage for Variables

```
architecture behavior of sig_var is
begin
process
variable var_s1, var_s2 :std_logic;
begin
wait until (rising_edge(clk));
var_s1 := x nand var_s2;
var_s2 := var_s1 xor y;
res <= var_s1 xor var_s2;
end process;
end behavior;
```

- Variables used before it is defined
  - Variables retain values across invocations
- There exists an execution sequence (first) where use precedes definition



Latch vs. Flip Flop Inference

- Predicates in conditional expressions lead to latch inference
  - *if (sel = '1') then...*
- Edge detection expressions lead to flip flop inference
  - *if (rising_edge(clk)) then...*
  - *if (clk'event and clk = '0') then..*
  - if (clk'lastvalue = '0' and clk = '1' and clk'event) ..

simulation semantics only

- Semantics must be consistent with available parts
  - Latches vs. flip flops
  - Trigger condition

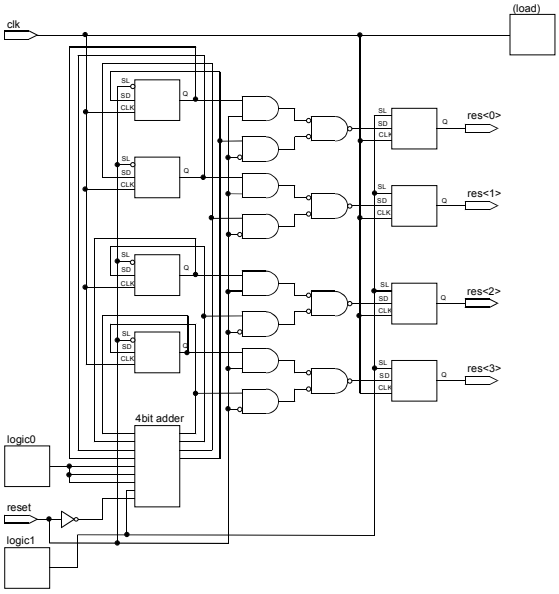
Example: Counter

```
entity counter is
port (clk, reset : in std_logic;
res : out unsigned (3 downto 0));
end entity counter;

architecture behavior of counter is
begin
process (clk, reset) is
variable var_count : unsigned (3 downto 0);
begin
if (rising_edge (clk)) then
if (reset = '1') then res <= "0000";
else
var_count := var_count + 1;
end if;
res <= var_count;
end if;
end process;
end architecture behavior;
```

edge trigger

Note variable is used before it is defined!

Example: Counter



Wait Statements

- Wait statements imply synchronous logic
- Only one wait statement permitted in a process and it must be the first statement in the process
- All signals in the body of the wait cause storage to be inferred
- Use *if-then-elsif-endif* constructs to trade-off combinational vs. sequential circuit inference

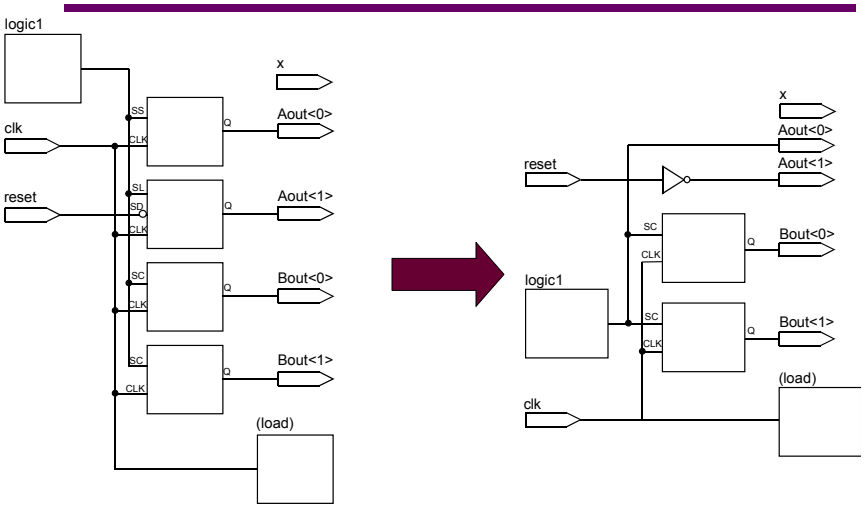
Controlling Inference: Example

```
architecture behavioral of edge is
begin
process
begin
wait until (rising_edge(clk));
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
Bout <= 0;
end process;
end architecture behavioral;
```

```
architecture behavioral of edge is
begin
process (reset, clk) is
begin
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
if (rising_edge(clk)) then
Bout <= 0;
end if;
end process;
end architecture behavioral;
```

- Control inference by use of the wait statement

Controlling Inference: Example



- Storage inference is a function of code structure

Simulation vs. Synthesis Sematics

Consistency with simulation semantics?

```
wait until rising_edge (clk);
sig_a <= sig_x and sig_y;
sig_b <= sig_a xor sig_c;
```

- What value of sig_a is used in the computation of sig_b?

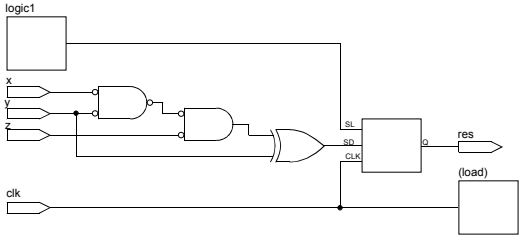
Consistency with simulation semantics?

```
process (x, y, z)
begin
L1: s1 <= x xor y;
L2: s2 <= s1 or z;
L3: w <= s1 nor s2;
end process;
```

- Storage is not inferred in this example!

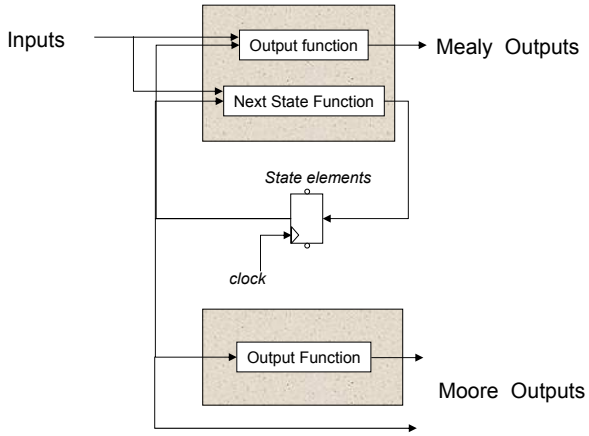
Variables Controlled by a Wait Statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sigvar is
port(y, z, x, clk : in std_logic;
    res: out std_logic);
end entity sigvar;

architecture behavioral of sigvar is
begin
 process is
variable a_var, b_var : std_logic;
begin
wait until (rising_edge(clk));
 a_var := x or y;
b_var := a_var nor z;
res <= b_var xor y;
end process;
end architecture behavioral;
```



- Variable dependencies collapsed
- If variables are “used before they are defined” then storage is inferred
- Signal dependencies are not collapsed, but rather storage is inferred for each signal

Synthesis of State Machines



- Combinatorial component and sequential component
- Asynchronous and synchronous components
- Control inference through use of wait and edge detection expressions

State Encodings

State	Sequential	Gray Code	One Hot
0	000	000	00000001
1	001	001	00000010
2	010	011	00000100
3	011	010	00001000
4	100	110	00010000
5	101	111	00100000
6	110	101	01000000
7	110	100	10000000

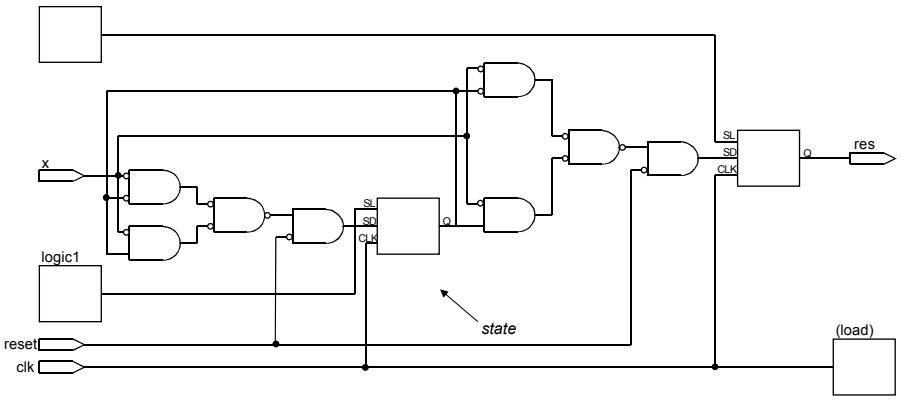
- Source-level constructs for user supplied encodings
- Goal: optimize area or speed
- What about illegal states?

Controlling Inference: Example

```
architecture behavioral of state_machine is
type statetype is (state0, state1);
signal state, next_state : statetype ;
begin
 process is
begin
wait until (rising_edge(clk)); -- rising edge
if reset = '1' then
res <= '0'; -- check for reset
state <= statetype'left; -- initialize state
else
case state is -- switch on current state
when state0 => -- set outputs and next state
if x = '0' then
  state <= state1;
  res <= '1';
else state <= state0;
  res <= '0';
end if
when state1 =>
  if x = '1' then
    state <= state0;
    res <= '0';
  else
    state <= state1;
    res <= '1';
  end if;
end case;
end if;
end process;
end architecture behavioral;
```

- Flips flops inferred for all signals in the body of the process

Controlling Inference: Example

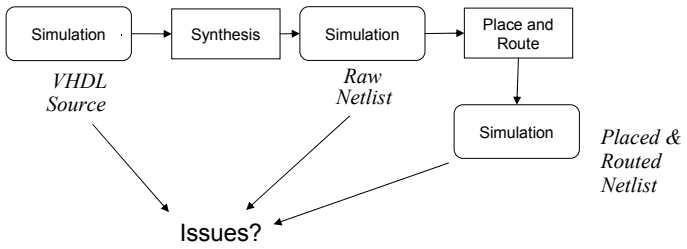


Controlling Inference: Example

```
architecture behavioral of state_machine is
type statetype is (state0, state1);
signal state, next_state : statetype;
begin
process (state, x) is
begin
 case state is -- switch on the current state
when state0 => -- set output and next state
   if x = '0' then
   next_state <= state1;
   res <= '1';
  else
   next_state <= state0;
   res <= '0';
 end if;
when state1 =>
  if x = '1' then
  next_state <= state0;
  res <= '0';
  else
   next_state <= state1;
   res <= '1';
  end if;
end case;
end process;

clk_process: process is
begin
wait until (rising_edge(clk);
  if reset = '1' then -- reset and initialize
    state <= statetype'left;
  else
    state <= next_state;
  end if;
end process clk_process;
end architecturebehavioral;
```

- Flip flops for the signal "res" are avoided

Simulation vs. Synthesis



- Design flow transforms a design to more detailed levels
- Simulation at each level to validate design
- Need to understand sources of mismatch between levels
- Performance issues differ at each level

Simulation vs. Synthesis

- Obvious source: design error
- Incomplete sensitivity lists

```
process (sel) is
begin
if (sel = '1' and En = '0') then
A <= 1;
else
A <= '0';
end if;
end process;
```

- Use of signals in a process

```
process (x, y, z)
begin
L1: s1 <= x xor y;
L2: s2 <= s1 or z;
L3: w <= s1 nor s2;
end process;
```


Simulation vs. Synthesis

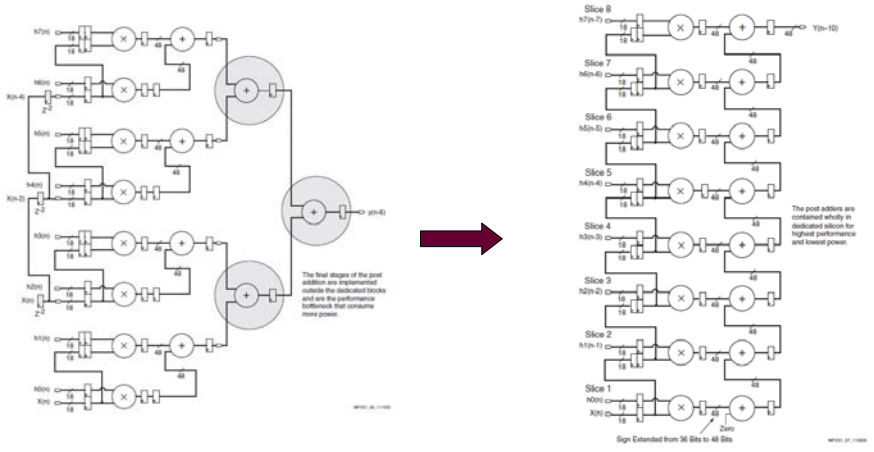
- Delay statements
- Speed
  - Overhead of maintaining and updating signal drivers
  - Use of processes and variables speed up simulation

Some Synthesis Guidelines: From Xilinx

- Use of resets
  - Asynchronous resets can prevent inference of optimized components
    - Optimized SRL library does not have reset
    - Building SRLs places reset on the critical path
    - Only synchronous resets available on some components
    - Mapping general logic to RAM
    - Efficiency of inference
  - Use synchronous reset as possible and necessary
  - Use of asynchronous resets results in synthesis of additional logic outside of primitives
    - Interferes with remainder of the design, e.g. during place and route
  - Optimizations for synchronous reset and logic synthesis

From http://www.xilinx.com/bvdocs/whitepapers/wp231.pdf

Some Synthesis Guidelines: From Xilinx



- Trading throughput for latency
  - Adder chains vs. adder trees

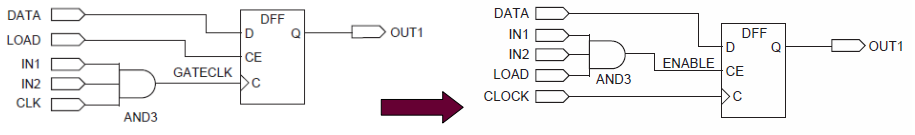
From http://www.xilinx.com/bvdocs/whitepapers/wp231.pdf

Some Synthesis Guidelines: From Xilinx

- Using registers
  - Pipelining for throughput
    - Can be coupled with retiming to further improve performance
    - What happens to our ability to verify circuits?
  - Register replication to handle high fan out
- Instantiation vs. inference
  - Some optimized behaviors cannot be inferred (in today's compilers)
  - Boundaries of optimization
    - "cone of influence" from instantiated components
  - Sometimes easier to instantiate elements than describe them effectively
    - e.g., Serdes

From http://www.xilinx.com/bvdocs/whitepapers/wp231.pdf

Some Synthesis Guidelines: From Xilinx



inadvisable *advisable*

- Clock gating vs. enables with global resources

From http://www.xilinx.com/bvdocs/whitepapers/wp231.pdf

Some Synthesis Guidelines: From Xilinx

- Nesting
  - Avoid deep nesting which leads to long signal paths
  - Can explicitly place registers in a loop to avoid long signals in unrolled loop
- Hierarchy
  - Boundaries may have to be preserved for verification purposes
  - Use global resources at the highest level of the hierarchy
- Check for additional resources in the document and Xilinx website for more info

From http://www.xilinx.com/bvdocs/whitepapers/wp231.pdf

Coding Styles

- Target independent vs. target dependent coding style
- Keep synthesizable blocks relatively small
- Separate combinational from clocked (registered) pieces of logic
  - Use nested if-then-elsif-endif structure in process followed by edge detected blocks of code
- Check http://toolbox.xilinx.com/docsan/xilinx7/books/data/docs/sim/sim0026_6.html

Summary

- Synthesis of behaviors encapsulated in processes
- Inference from sequential statements
- Latch inference vs. flip flop inference
- Effect of using variable vs. signals for synthesis
- Inference using *wait* vs. *if-then-else* statements
- Optimizations