VHDL
Modeling Behavior from Synthesis Perspective
- Part A -

EL 310

Erkay Savaş

Sabancı University

Motivation

- Synthesis of VHDL models with <u>process</u> construct.
- Processes provide powerful modeling abstraction for simulation purposes.
  - modeling complex digital system behavior that cannot be captured with only CSA statements.
- Increasing the level of modeling abstraction makes hardware inference process harder.
  - More than one hardware implementation is possible
  - Inference process has more work to do. Relationship between the language constructs and the real systems is no longer explicit.
  - Simple coding guidelines in order to ease the synthesis compiler's task.

Synthesis with Processes

- Process construct models have similarities to high-level programming languages
  - sequential style of high-level programming languages may lead to excessive logic and often long signal paths.
- Good rule of thumb:
  - Avoid long processes
  - promote concurrency within models through the use of multiple processes and CSA statements.
  - After all, processes are concurrent to other processes and to CSA statements.

Recall

- Combinational logic
  - The value of an output signal is defined for every combination of values of input signals
  - "Previous" values do not have to be remembered.
  - Make sure that every time a process is executed, each output should be assigned a value.
- Sequential logic
  - The values of an output signal may depend on "previous" values stored in the circuit
  - **`if-then-else`** constructs leads to a latch element if **`else`** branch does not assign a value to one of the output signals
  - Next issue: will edge-sensitive flip-flop or latch be inferred?
  - `wait until (falling_edge(clk));`

One Old Example

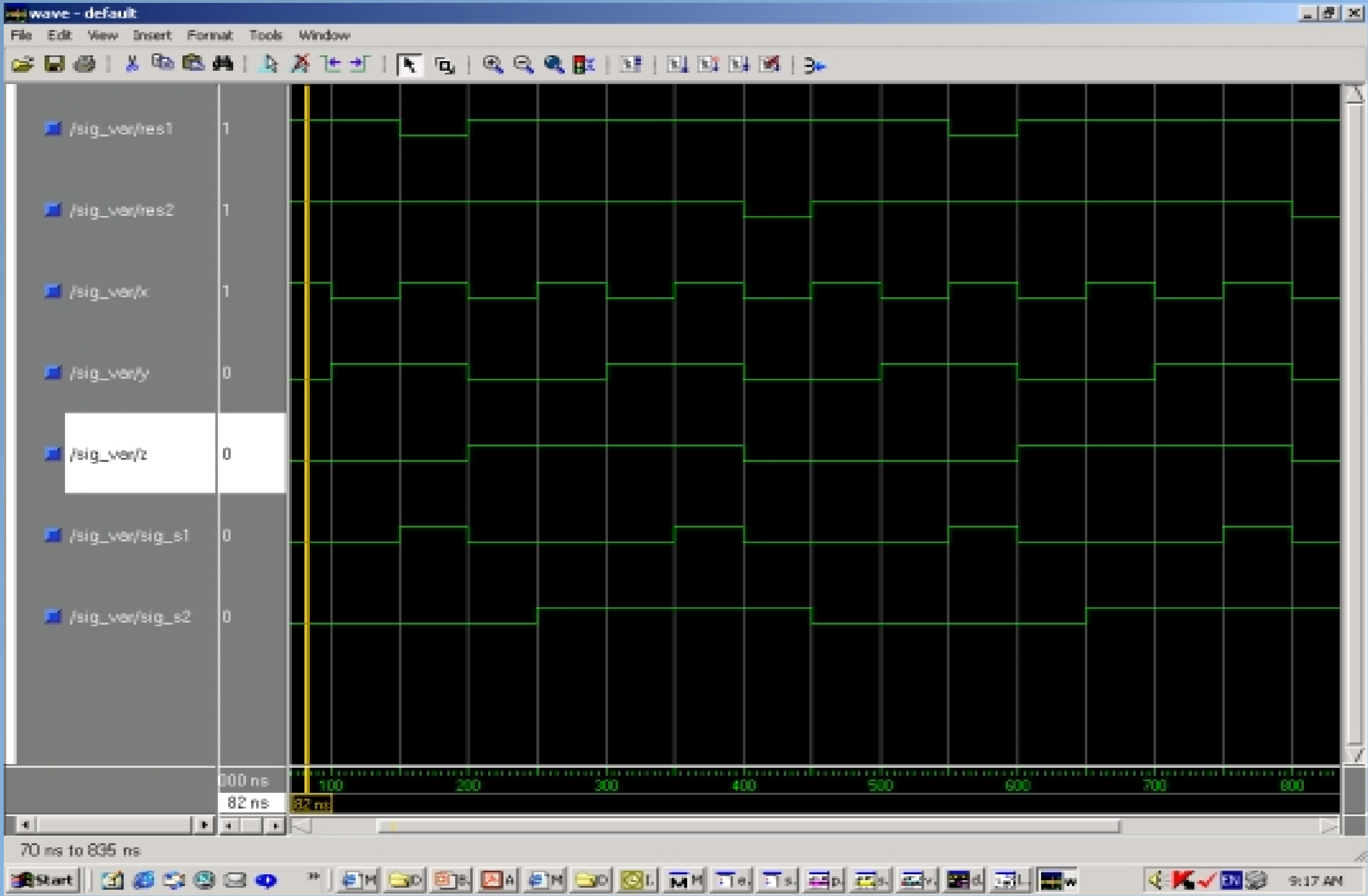
```
library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
port(x,y,z: in std_logic;
     res1, res2: out std_logic);
end entity sig_var;

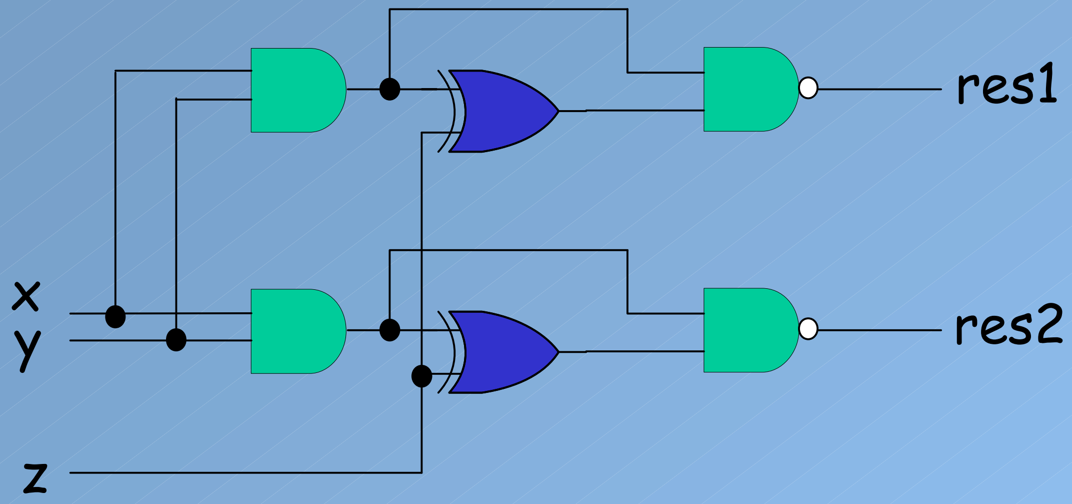
architecture behavioral of sig_var is
  signal sig_s1, sig_s2: std_logic;
begin

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 nand var_s2;
end process;
proc2:process(x,y,z) is -- process 1 using signals
begin
  sig_s1 := x and y;
  sig_s2 := sig_s1 xor z;
  res1 <= sig_s1 nand sig_s2;
end process;
end architecture behavioral;
```

Simulation Results

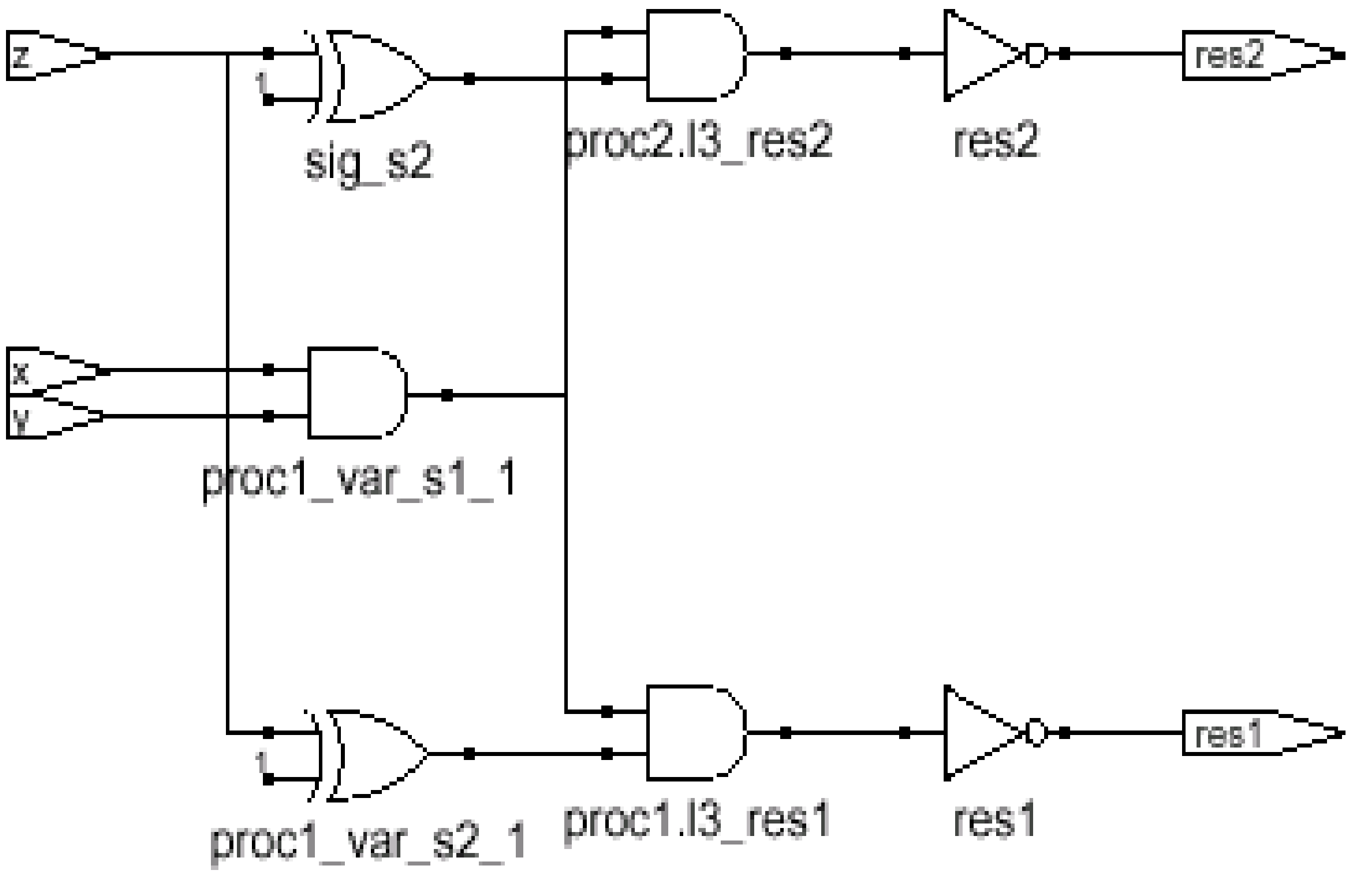


Example: The Synthesized Logic



- Two processes result in identical logic for `res1` and `res2`.
- Both circuit are combinational
- combinational logic is a natural result of using "variables" that are assigned immediately with values evaluated at the RHS of assignment statements.

Synplify Pro RTL View



res1 = res2 = z + (xy)'

Example: The Synthesized Logic

- Recall that
  - the values of sig_s1 and sig_s2 used when the process proc2 is executed are the ones when the process is invoked,
  - Not the values are newly evaluated within the procedure
  - This would cause inference of storage elements.
- However, compilers generally optimize this sequence to produce combinational logic.
  - Thus, signals and variables behave identically with respect to the synthesis in this case.
  - Dependency in the sequential execution of signals will increase the critical path.

Sensitivity List Mismatch

- The signal `sig_s1` is not in the sensitivity list.
  - Therefore, an event on it (caused by another process perhaps) will not lead to the execution of process `proc2`.
  - However, an event on `sig_s1` will cause the re-computation of the output values in the synthesized circuit.
- Simulation semantics may not match the behavior of the synthesized circuit.

Data Dependency

- Independent of whether variables and signals are used, process statements introduce a data dependency between the process statements
- ```
  s1 <= ...
  s2 <= ... s1 ...
  s3 <= ... s2 ...
  s4 <= ... s2 ...
  ```
  and so on.
- <u>Result</u>: long chain of dependency to calculate `s4`.
- <u>Consequence</u>: a long signal path in the synthesized logic.
  ```

Synthesis of Conditional Statements

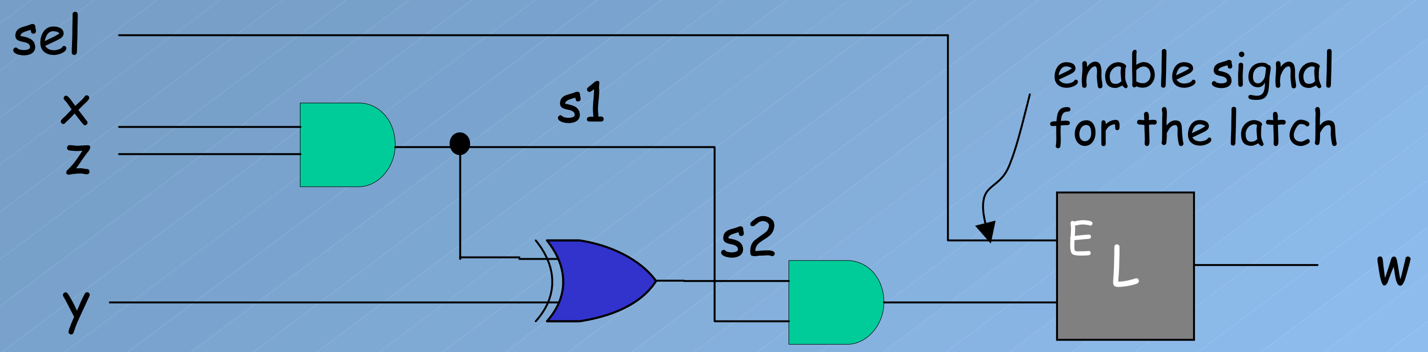
- `If-Then-Else` and `If-Then-Elsif` statements
  - All Boolean valued expressions are evaluated sequentially until the first true expression is encountered.

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

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

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

Example: Latch Inference



- If sel = '1' the new value of the signal w is computed as shown in the code.
- Otherwise it will keeps its previous value stored in the latch.
- <u>Rule</u>: to avoid unnecessary latched inferred, the signal values should be assigned a value in every branch of conditional statements (both in `then` and `else` statements).
- remember that we have to make sure that all output signals get a value in every execution of process.

Alternative to Avoid Latches

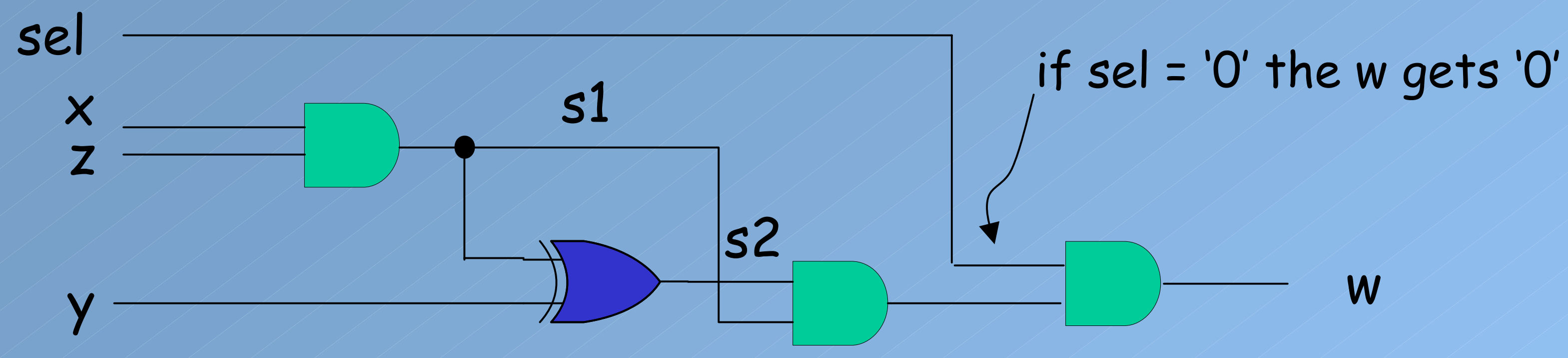
- Assign a default value to the signal prior to the if statement

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

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

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

Avoid Latches by Default Value



Efficiency

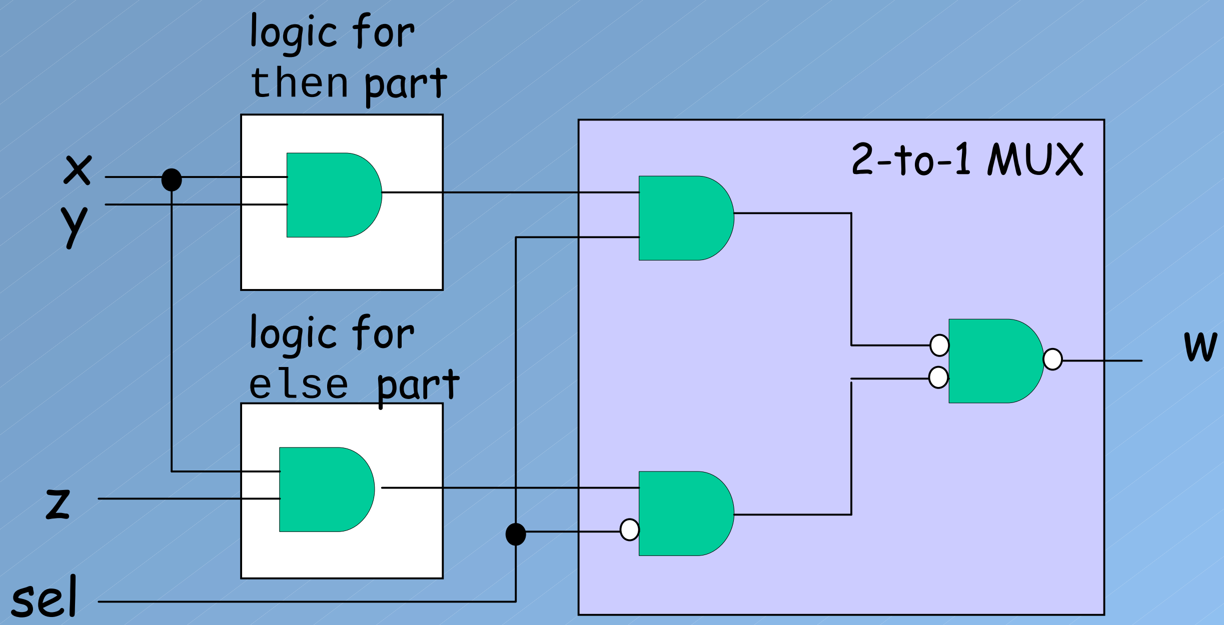
- Avoiding unnecessary latches is one aspect of effective design for combinational circuit synthesis
- Efficiency in terms of speed and/or size is the other.

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

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

architecture behavioral of inference is
variable s1, s2:std_logic;
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;
```

Example: Efficiency



- The general principle:
- combinational logic is generated for each branch of an `if-then-else` construct.
- A multiplexor is generated to select the outcome

More Efficient Design - 1

- a small rearrangement will simplify the design

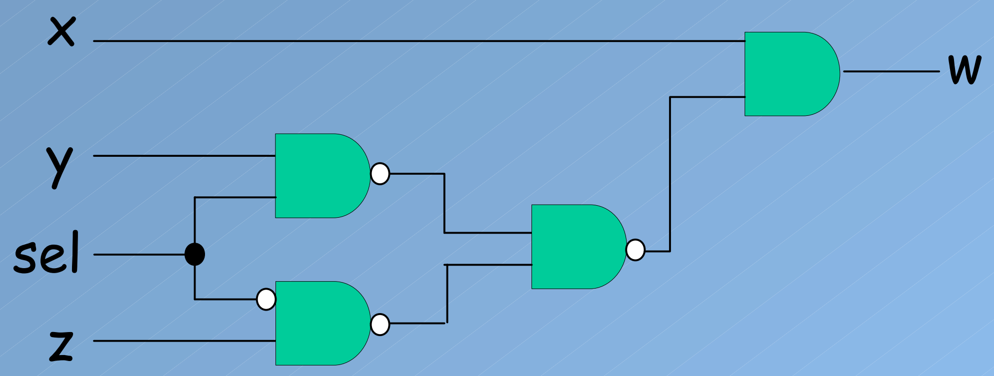
```
library IEEE;
use IEEE.std_logic_1164.all;

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

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

end architecture;
```

More Efficient Design - 2



- Coding styles help us to control how much hardware is generated.

- Golden rule: move complex or hardware intensive operations that are replicated in **then** and **else** branches out of the **if-then-else** statement.

Multiple Levels of Nesting

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

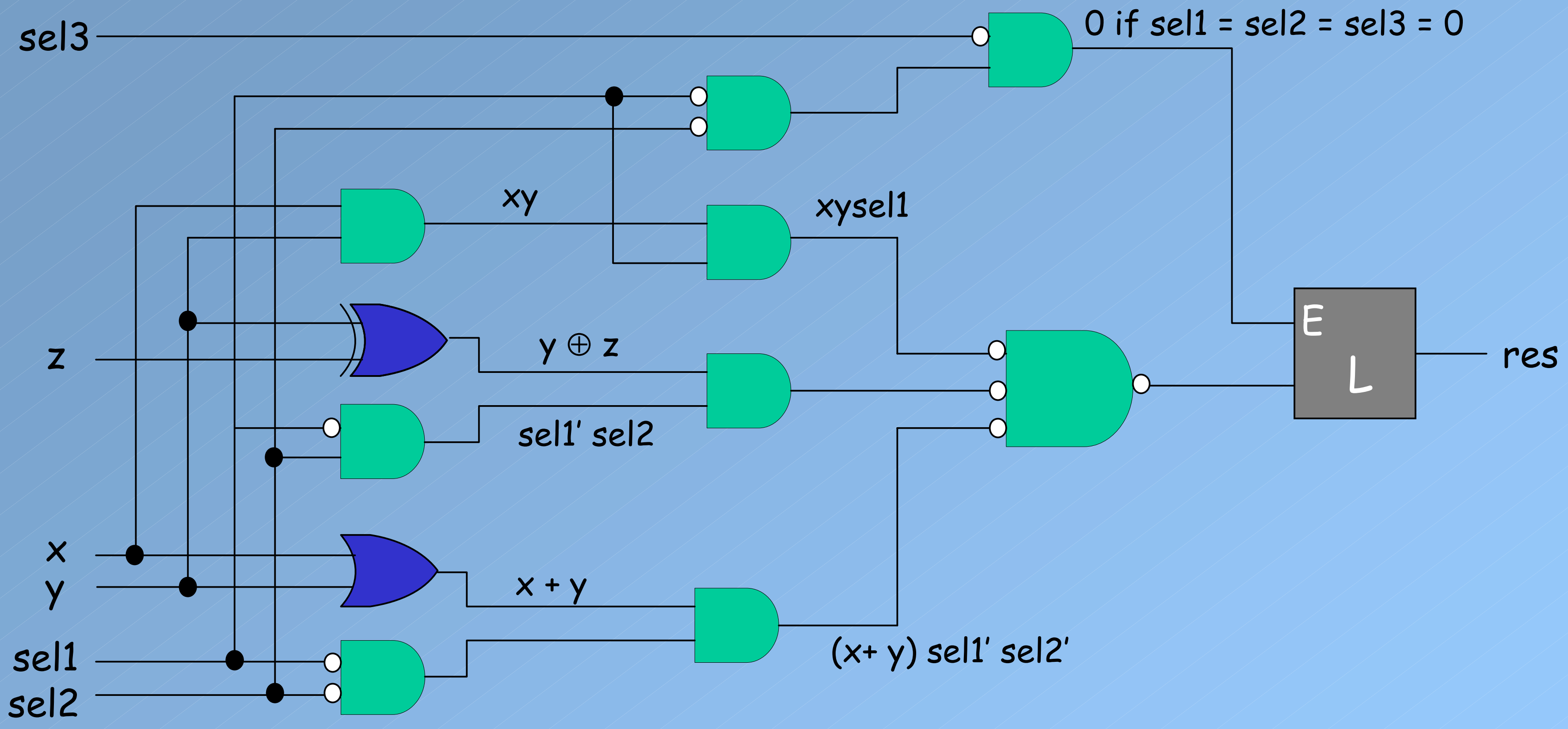
entity nested is
port(sel1, sel2, sel3, x,y,z: in std_logic;
     res: out std_logic);
end entity nested;

architecture behavioral of nested is
begin
process(x, y, z, sel1, sel2, sel3) is
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;

end architecture;
```

Question: Is latch inferred?

Example: Multiple Levels of Nesting

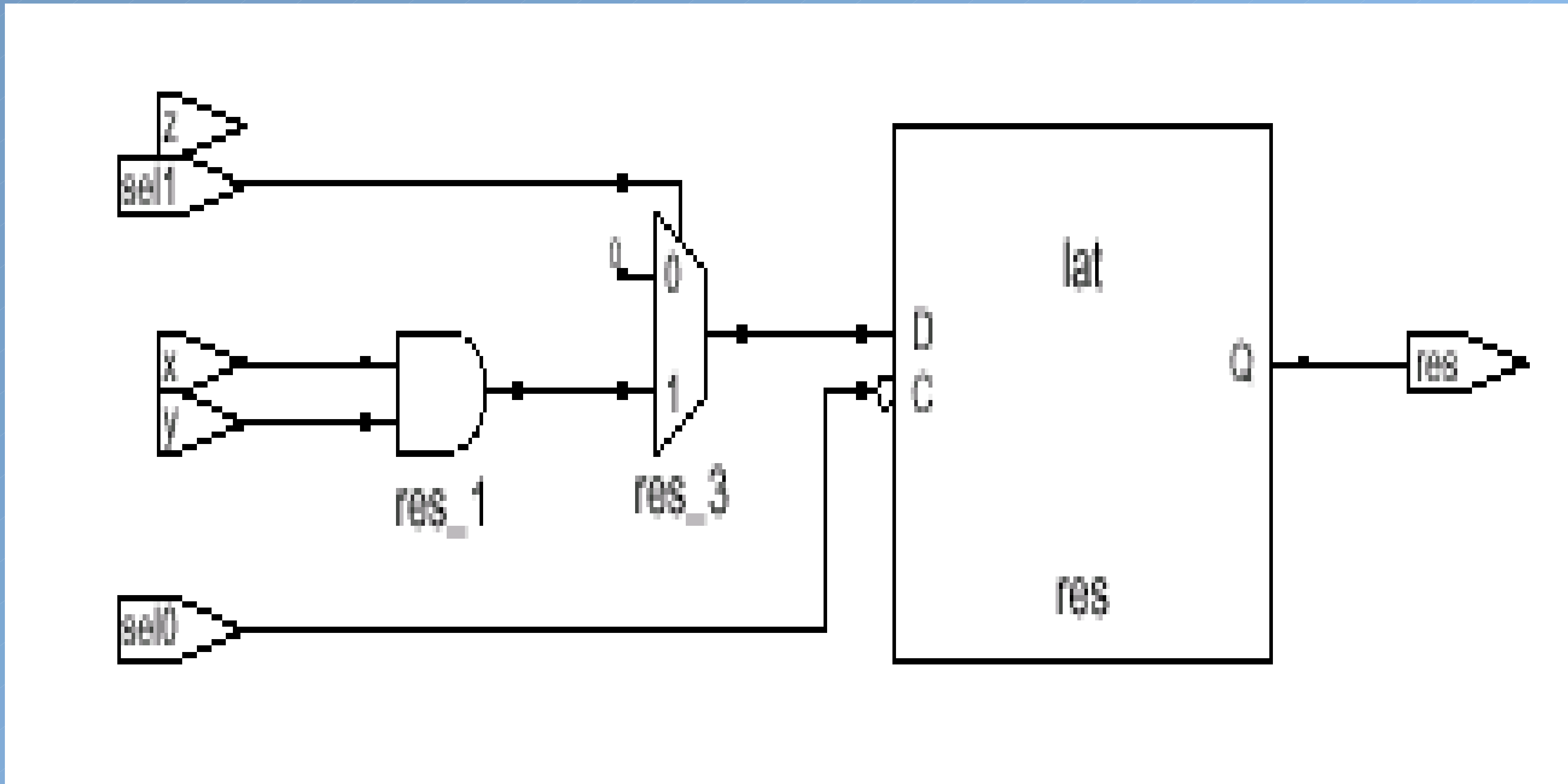


- Priority logic is implemented.
- What is the highest priority?

Nested If Statements

```
library IEEE;
use IEEE.std_logic_1164.all;
entity nested_ifs is
port(sel0, sel1, x,y,z: in std_logic;
     res: out std_logic);
end entity nested_ifs;
architecture behavioral of nested_ifs is
begin
process(x, y, z, sel0, sel1) is
begin
  if (sel0 = '0') then
    if (sel1 = '1') then
        res <= x and y;
    else                        -- included to avoid latch inference
        res <= '0';
    end if;
  end if;
end process;
end architecture;
```

Nested If Statements

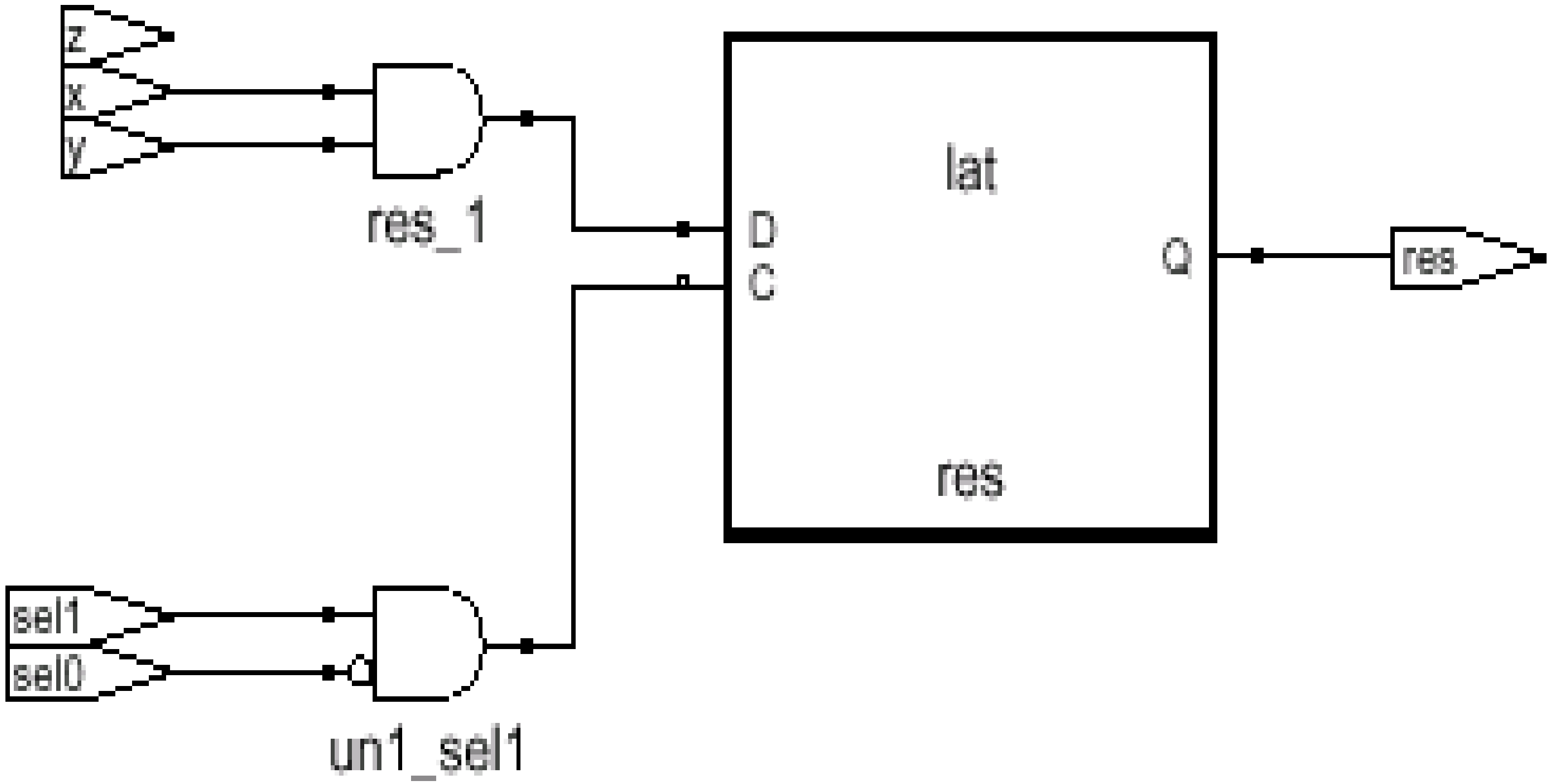


- Latch is inferred because of the outer if-then-else statement

Nested If Statements

```
library IEEE;
use IEEE.std_logic_1164.all;
entity nested_ifs is
port(sel0, sel1, x,y,z: in std_logic;
res: out std_logic);
end entity nested_ifs;
architecture behavioral of nested_ifs is
begin
process(x, y, z, sel0, sel1) is
begin
  if (sel0 = '0') then
    if (sel1 = '1') then
        res <= x and y;
    end if;
  end if;
end process;
end architecture;
```

Nested If Statements



Remove the inner `if-then-else` from the outer one in order to infer combinational logic

Don’t Cares in Conditional Expressions

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

entity sig_var is
port(sel: in std_logic_vector(1 downto 0);
     x,y,z: in std_logic;
     w: out std_logic);
end entity sig_var;

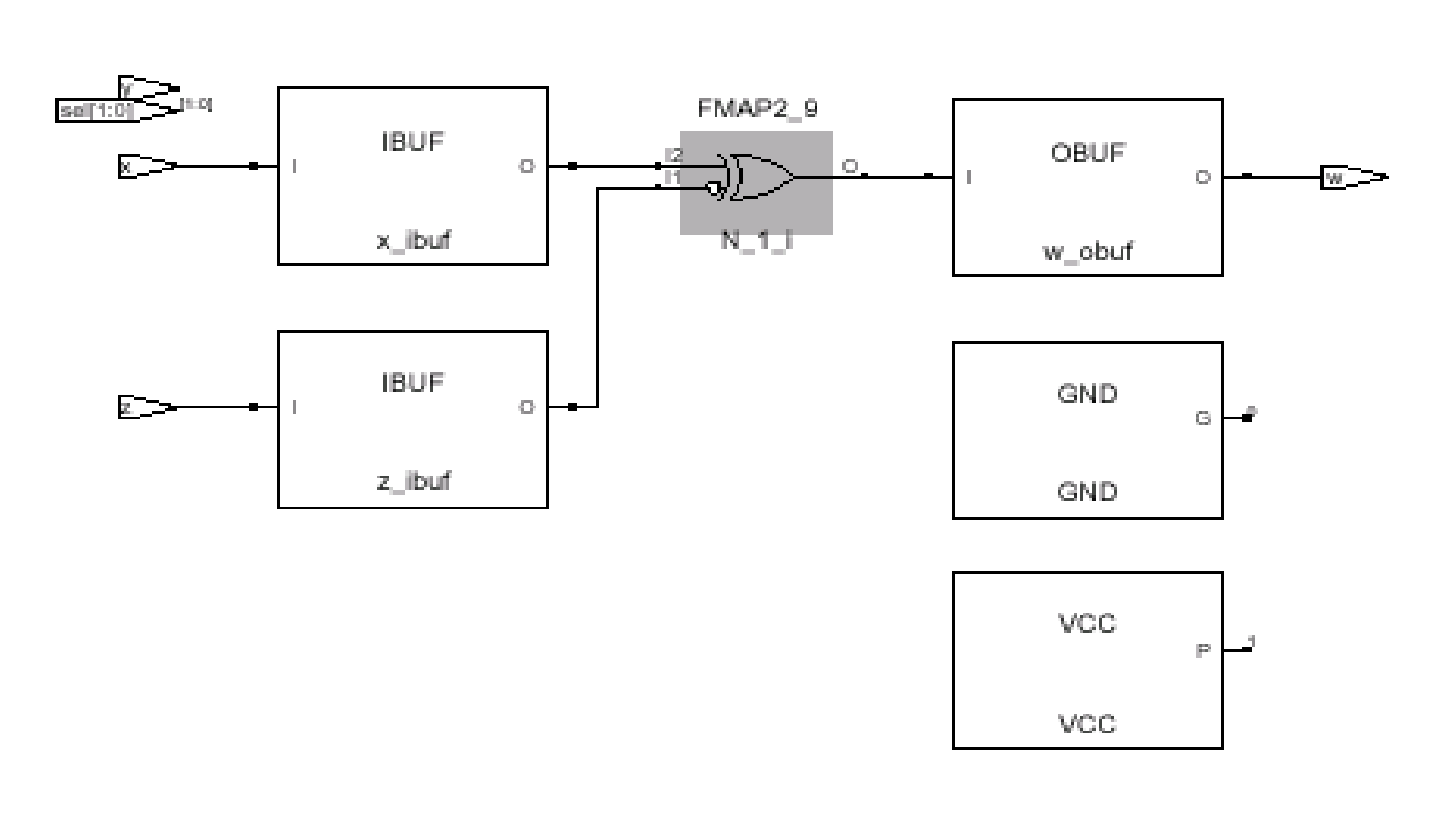
architecture behavioral of sig_var is
begin
process(x, y, z) is
begin
  if (sel = ‘-0’) then
    w <= x xnor y;
  else
    w <= x xnor z;
  end if;
end process;

end architecture;
```

comparisons with don’t cares in conditional expressions return always FALSE.

Don’t Cares in Conditional Expressions

Synthesis compiler generates a logic in which then branch is never taken



Case Statement

- `Case` statement is similar to `if-then-else` constructs
  - but identifies mutually exclusive blocks of code.
  - Only one branch of a `case` statement can be true
  - and collectively all branches cover all possible values of select expression.
  - The `others` clause helps cover all possible values of select expression
  - Selecting one of the several possible alternatives suggests that a multiplexor is inferred.
  - Select expression provides the control signals for multiplexor.

Case Statement: Example

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

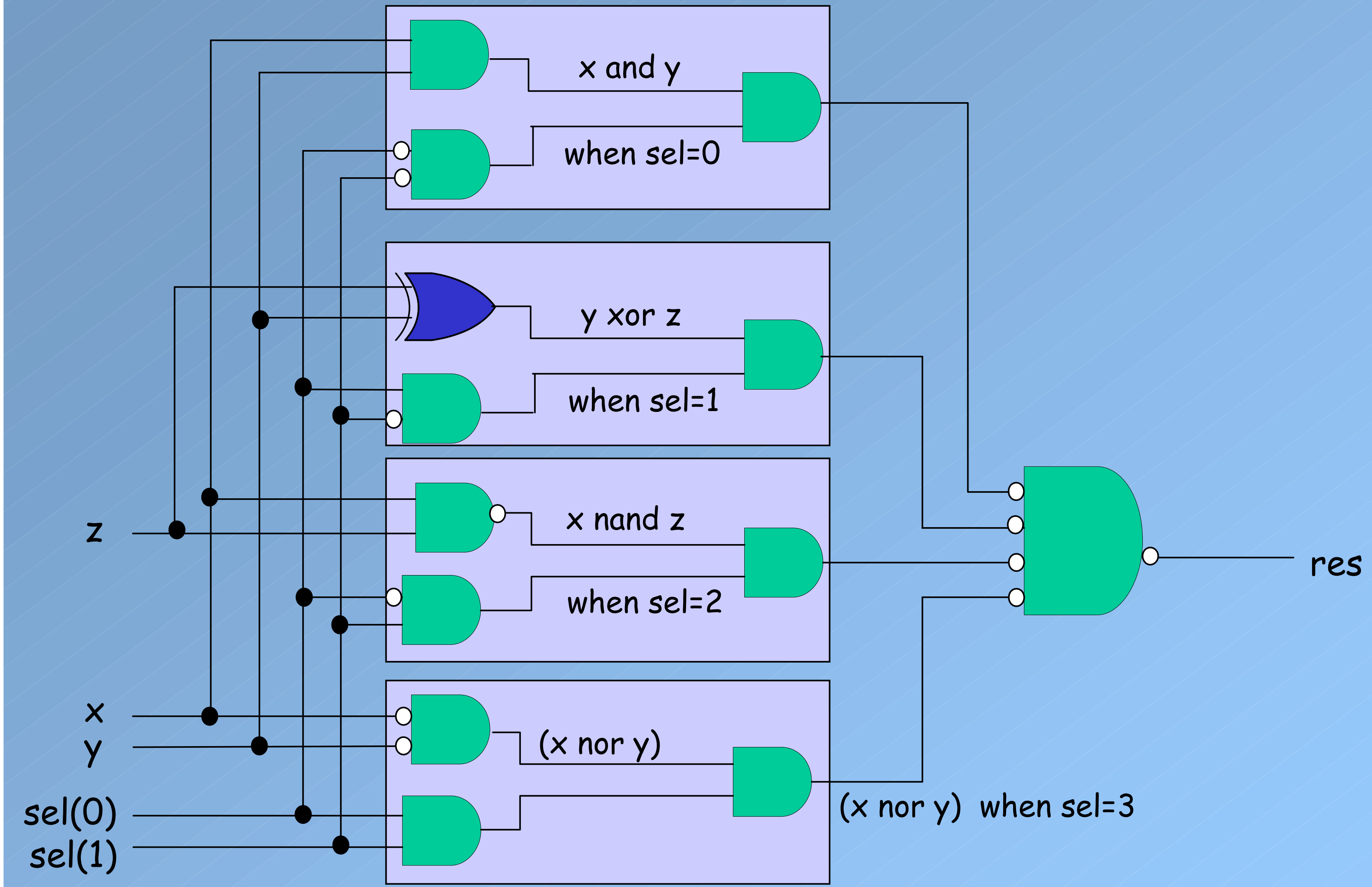
entity case_ex is
port(sel: in integer range 0 to 3;
     x,y,z: in std_logic;
     res: out std_logic);
end entity case_ex;

architecture behavioral 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 => res <= x nor z;
  end case;
end process;

end architecture;
```

A combinational logic is inferred when output signals receive a value in all branches of the case

Case Statement: Example



`sel: in integer range 0 to 3;` is mapped to a two-bit signal

Latch Inference from Case Statement

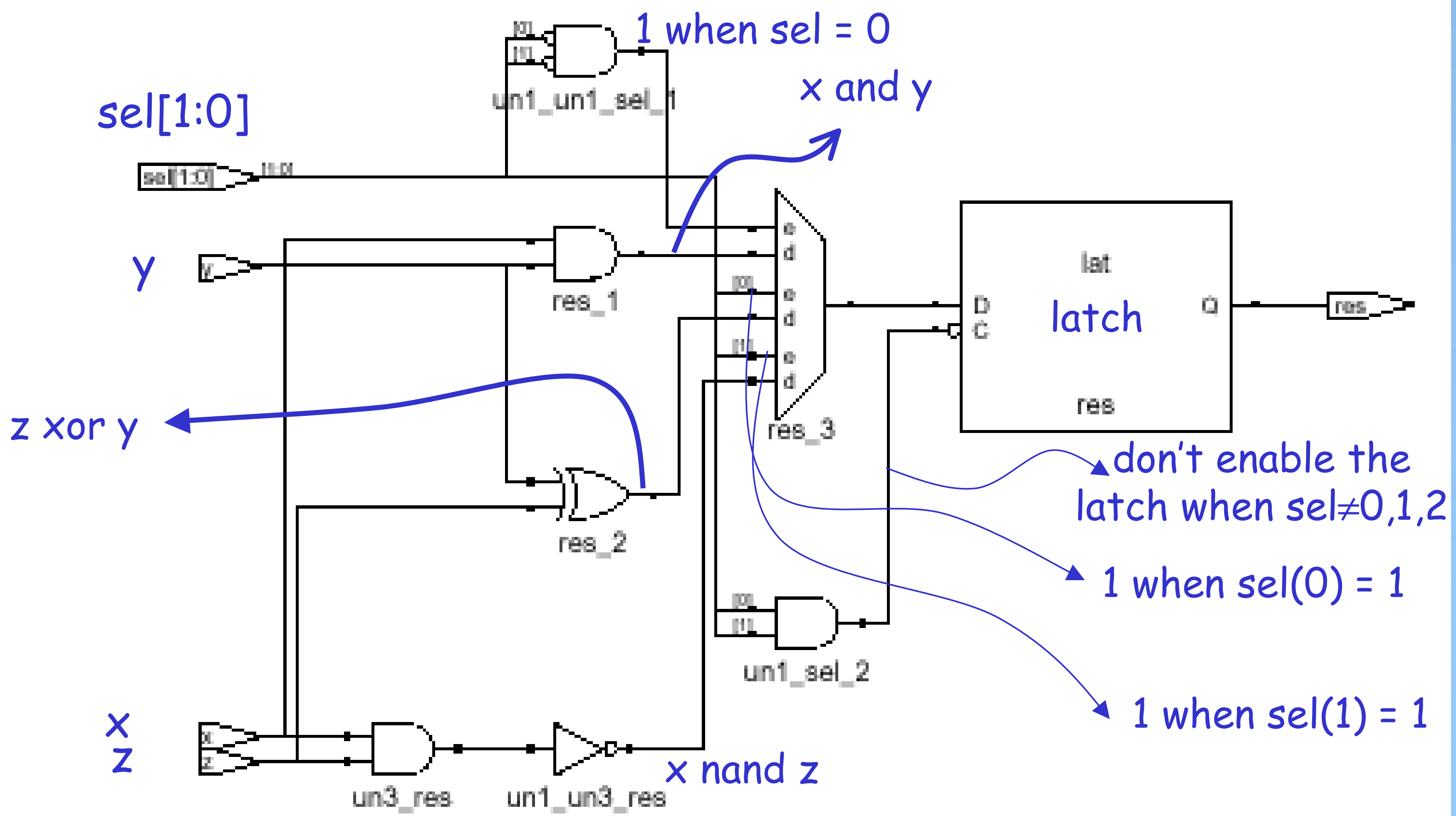
```
library IEEE;
use IEEE.std_logic_1164.all;

entity case_ex is
port(sel: in integer range 0 to 3;
     x,y,z: in std_logic;
     res: out std_logic);
end entity case_ex;

architecture behavioral 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;
```

Latch Inference from Case Statement



Avoiding Latch Using Default Value

```
entity case_ex is
port(sel: in integer range 0 to 3;
     x,y,z: in std_logic;
     res: out std_logic);
end entity case_ex;

architecture behavioral of case_ex is
begin
process(x, y, z, sel) is
begin
  res <= '0';
  case sel is
    when 0 => res <= x and y;
    when 1 => res <= y xor z;
    when 2 => res <= x nand z;
    when others => null;
  end case;
end process;

end architecture;
```

Any `case` statement has an equivalent `if-then-elsif` form.
Question: what is the difference between the two?

Using don't cares in Case Statements

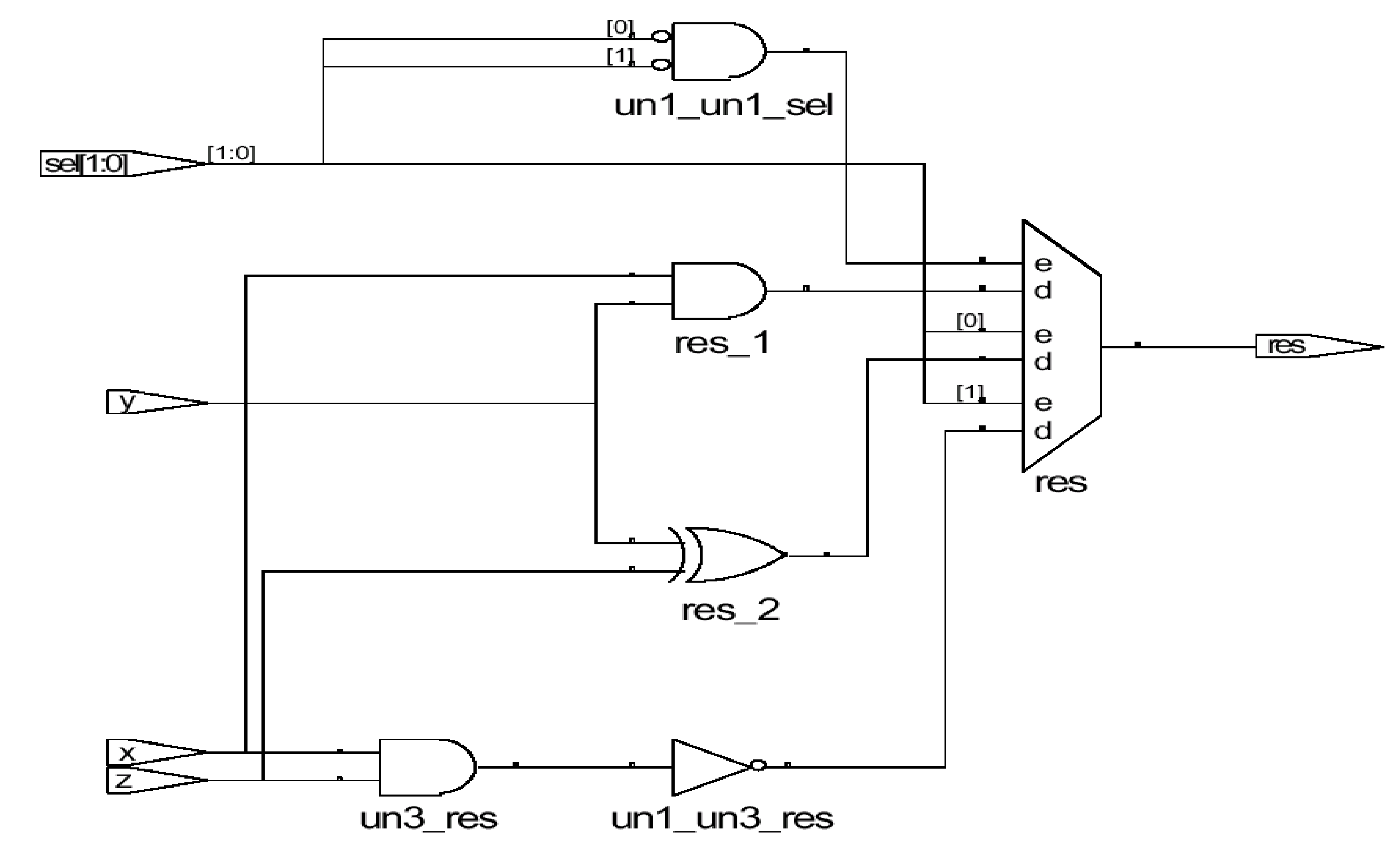
```
entity case_ex is
port(sel: in integer range 0 to 3;
     x,y,z: in std_logic;
     res: out std_logic);
end entity case_ex;

architecture behavioral 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 => res <= '0';
    when others => res <= '-';
  end case;
end process;

end architecture;
```

Use don't cares in **when others** to let the synthesis tool to optimize the circuit.

Using `don't cares` *in* `Case` *Statements*



circuit output res is not defined for sel = 3

Loop Statements

- For-Loop
  - Most common construct for loops supported by synthesis compilers
  - At compile-time, it is possible to know when the loops ends; the logic is inferred accordingly.
  - With while-loops, when the loop ends may be data dependent.
  - Therefore, a state machine controller is synthesized to cycle datapath a data-dependent number of times.
  - For-loops, on the other hand, is easy to synthesize. since the number of iterations is known.

For Loops

- Example:
  - ```
    for N in 3 downto 1 loop
          shift_reg(N) <= shift_reg(N-1);
    end loop;
    ```
  - The loop can easily be replaced by sequential code
    ```
    shift_reg(3) <= shift_reg(2);
    shift_reg(2) <= shift_reg(1);
    shift_reg(1) <= shift_reg(0);
    ```
  - This technique is known as <u>loop unrolling</u>, and it is also being commonly used in conventional languages for optimization purposes.
  - In VHDL synthesis, loop unrolling is also used as an optimization technique.
    ```

Loop Unrolling: Example

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

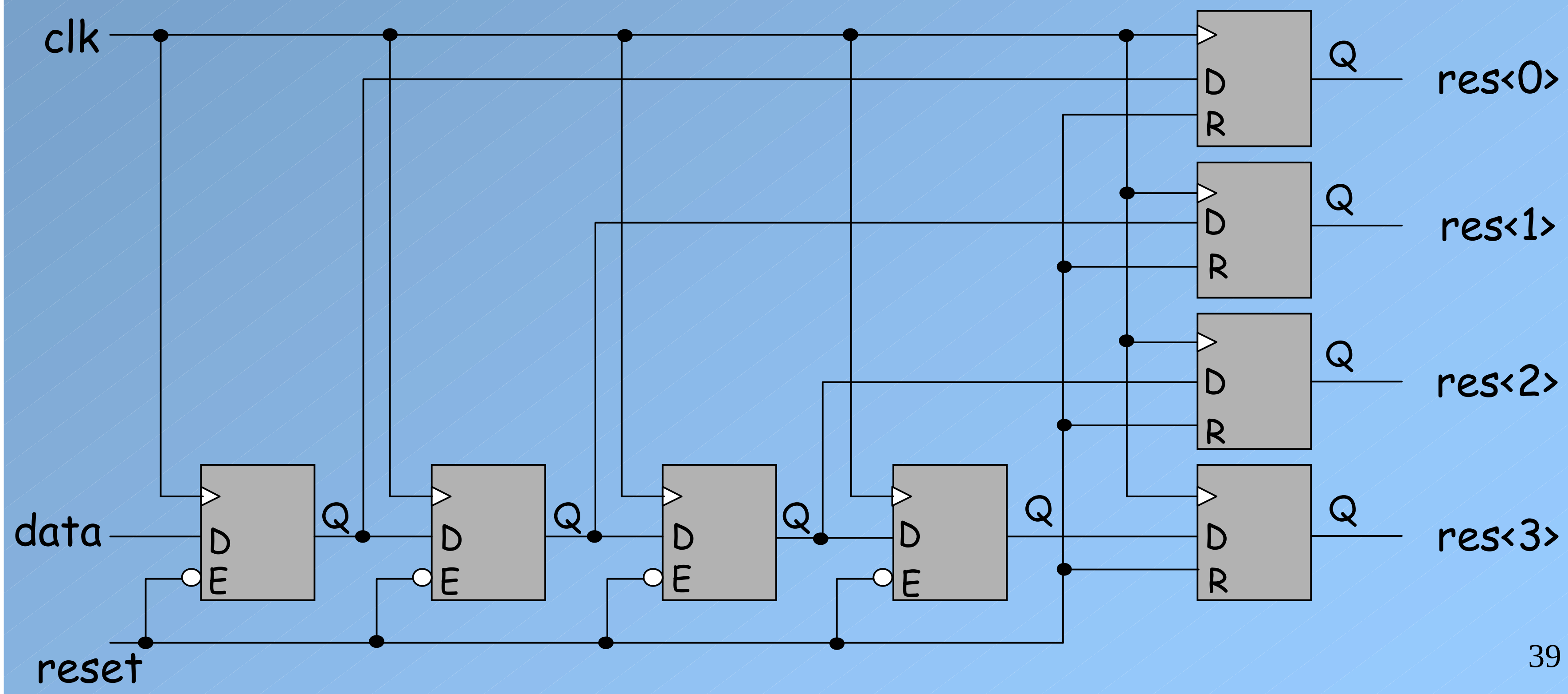
entity iteration is
port(clk, reset, data: in std_logic;
     res: out std_logic_vector(3 downto 0));
end entity iteration;

architecture behavioral 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;
```

Loop Unrolling: Example

- Latch inference
  - Because of **if**(rising_edge(clk)) **then** with no else, a storage element will be inferred.
  - Bu due to the call to function rising_edge() a flip-flop instead of a latch is inferred.



While Loop

```
entity iteration is
port(clk, reset, data: in std_logic;
     res: out std_logic_vector(3 downto 0))
end entity iteration;
architecture behavioral of iteration is
  signal shift_reg: std_logic_vector(3 downto 0);
begin
process(clk, reset, data) is
  variable N: integer;
begin
  if(rising_edge(clk)) then
    if(reset = '1') then res <= "0000";
    else
      N := 3;
      while N > 0 loop
        shift_reg(N) <= shift_reg(N-1); N := N - 1;
      end loop;
      shift_reg(0) <= data; res <= shift_reg;
    end if;
  end if;
end process;
end architecture;
```

Exit Statement

```
sum := 1; j := 0;
L2: loop;
    j := j+21;
    sum := sum *10;
    exit when sum > 100;
end loop L2;
```

- The `exit` statement can be used only inside a loop.
- It causes execution to jump out of the innermost loop or the loop whose label is specified.
- **exit** [loop-label][**when** condition];

```
sum := 1; j := 0;
L3: loop;
    j := j+21;
    sum := sum *10;
    if sum > 100 then
       exit L3;
    end if;
end loop L3;
```

Next Statement

- A sequential statement that can be used only inside a loop.
- Syntax: **next** `[loop-label][`**when** `condition];`
- It results in skipping the remaining statements in the current iteration of the loop;
- execution resumes with the first statement in the next iteration of this loop, if one exists.

```
for j in 10 downto 5 loop
  if sum < total_sum then
    sum := sum + 2;
  elsif sum = total_sum then
    next;
  else
    null;
  end if;
  k := k + 1;
end loop;
```

Next Statement

- The next statement can also cause an inner loop to be exited.

```
L4: for k in 10 downto 1 loop
  -- statements section 1
  L5: loop
    -- statements section 2
    next L4 when WR_DONE = '1';
    -- statements section 3
  end loop L5;
  -- statements section 4
end loop L4;
```

- When `WR_DONE = '1'` becomes true, statements section 3 and 4 are skipped, and execution jumps to the beginning of the next iteration of loop L4.
- Notice that loop L5 is terminated by `next` statement.

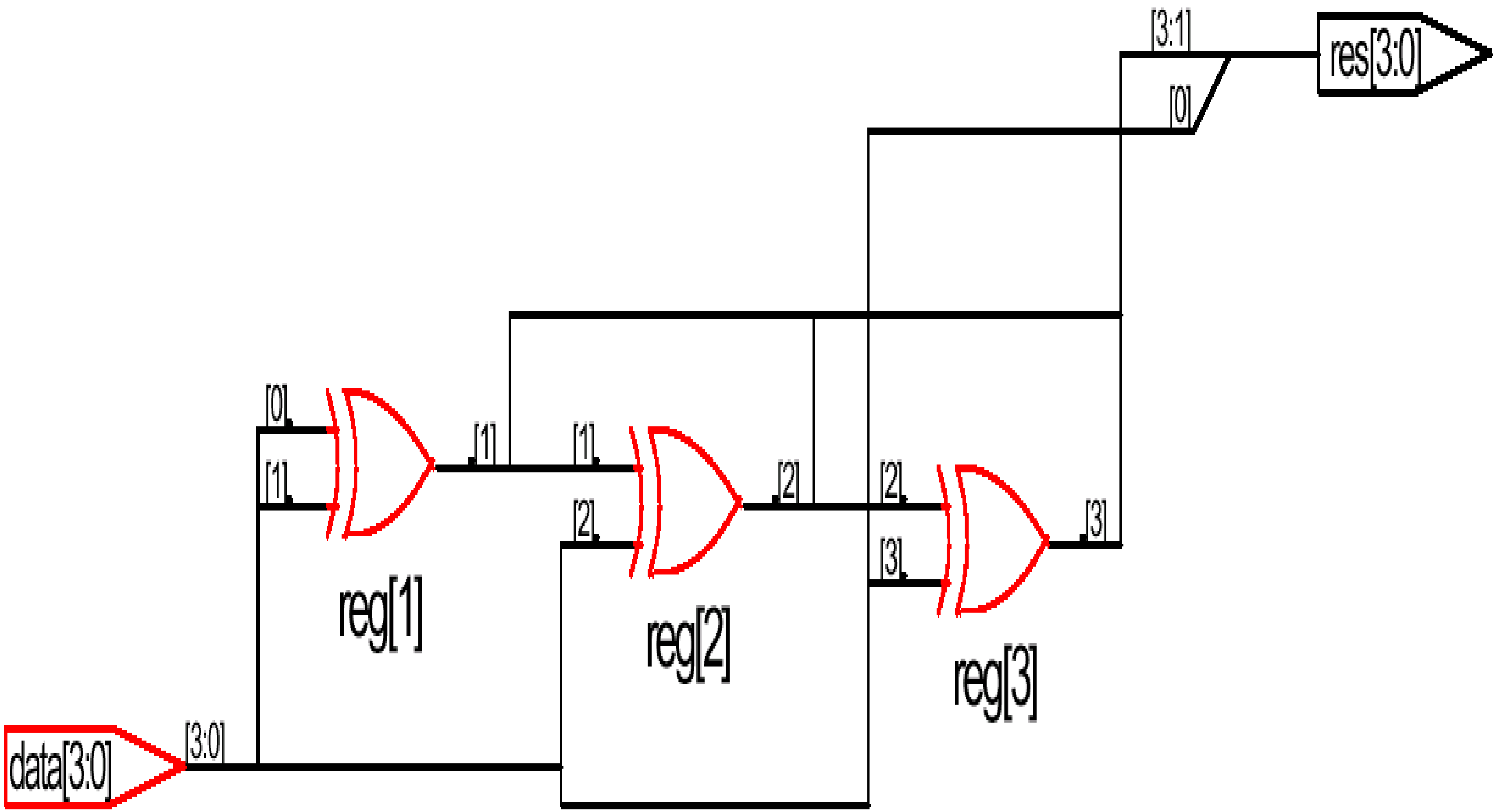
Loops

- Some synthesis compilers forces the type of the loop index to integers or only a subsets of array types.
- The dependencies across iterations of a loop will lead to long signal paths.
- Example:
  - ```
    reg(0)<= datain(0);
    for N in 1 to 3 loop
        reg(N) <= reg(N-1) xor datain(N);
    end loop;
    ```
  - When you unroll
    ```
    reg(0)<= datain(0);
    reg(1)<= reg(0) xor datain(1);
    reg(2)<= reg(1) xor datain(2);
    reg(3)<= reg(2) xor datain(3);
    ```
    ```

Loops: Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity iteration is
port(data: in std_logic_vector(3 downto 0);
     res: out std_logic_vector(3 downto 0));
end entity iteration;
architecture behavioral of iteration is
  signal reg: std_logic_vector(3 downto 0);
begin
  forl: process(data) is
  begin
    reg(0) <= data(0);
    for N in 1 to 3 loop
      reg(N) <= reg(N-1) xor data(N);
    end loop;
  res <= reg;
end process;
end architecture;
```

Loops: Example



Sensitivity List in Synthesis

- Process is executed when there is an event on any signal in the sensitivity list.
  - When a signal is not in the sensitivity list, an event on this signal does not cause the execution of the process.
  - However, when the VHDL code is synthesized in to a logic circuit, a change on any signal will lead to the re-calculation of the other signals that depends on this signal.
  - Therefore, simulation results after synthesis may not match exactly to those before the synthesis.
  - Lesson: Include all signals of process in the sensitivity list when performing simulations.

Variables

- Variables are language objects that is useful in describing the behavior of the circuit.
  - They are typically used to transfer values between statements in a process
  - Compiler may collapse variable assignment statements and eliminate some of them.
  - When it is not possible to do so, then a variable is synthesized to a wire connecting gates.

Inferring Latches

```
...
L1: if(s1 = '1') then
...
L2: if(s2 = '1') then
          Aout <= '1';
    else
          Aout <= '0';
    end if;
end if;
...
```

- Question: Is a latch for Aout inferred?

Buffer and Inout Entity Modes

- A signal can serve as both a driver and as an output signal
  - ```
    Aout <= in1 and in2;
    Bout <= Aout xor in3;
    ```
- In this case, `Aout` signal must be defined as either `buffer` or `inout`;
- It is important for hierarchical models
- For the time being, try to avoid situations like this, using the following:
  - ```
    VarAout := in1 and in2;
    Bout <= VarAout xor in3;
    Aout <= VarAout;
    ```

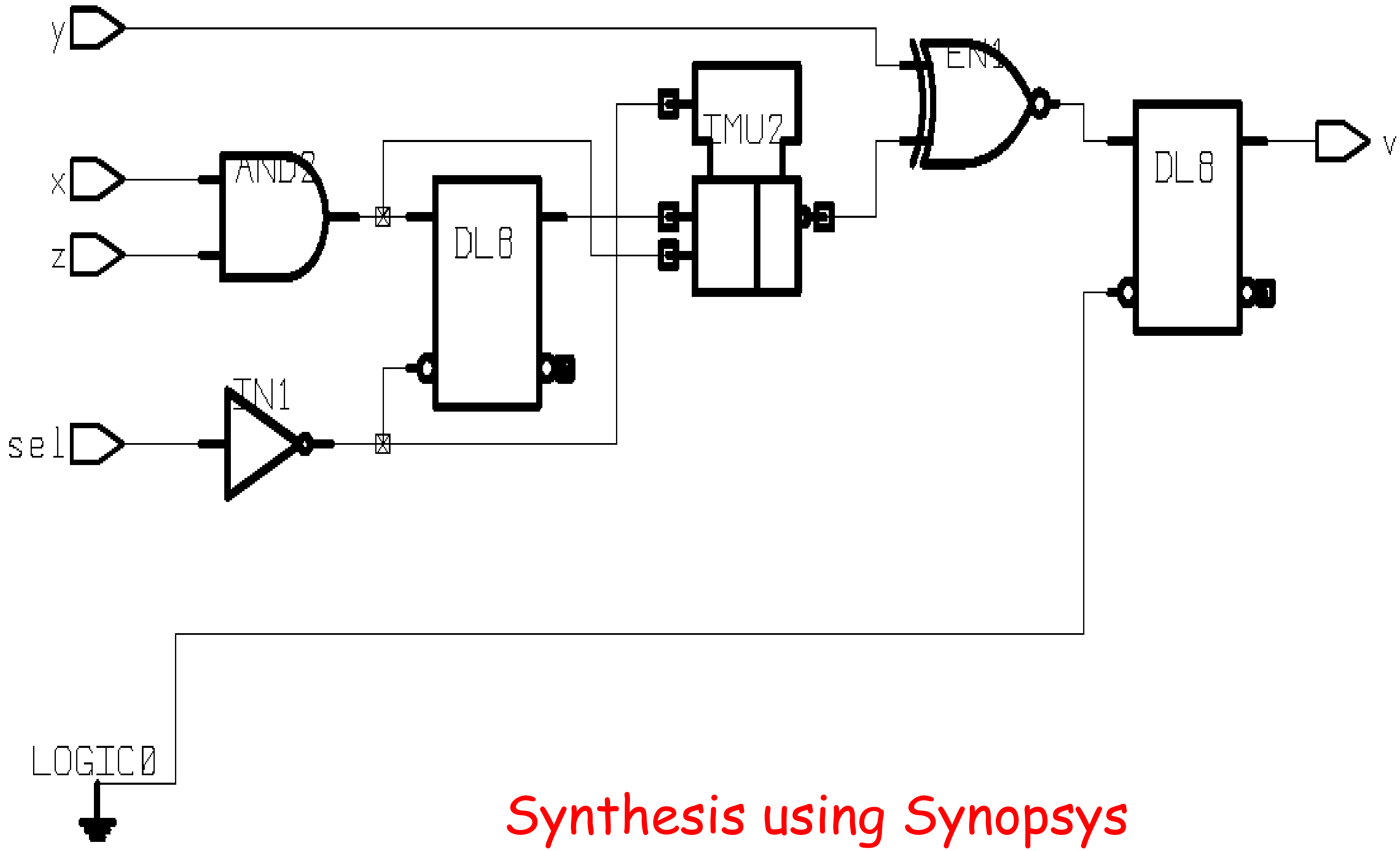
Inference Using Signals vs. Variables

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sigvar is
port(sel: in std_logic; x, y, z: in std_logic; v: out std_logic);
end entity sigvar;
architecture behavioral of sigvar is
signal sig_s1: std_logic;
begin
process(x, y, z, sel) is
variable var_s1: std_logic;
begin
  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;
```

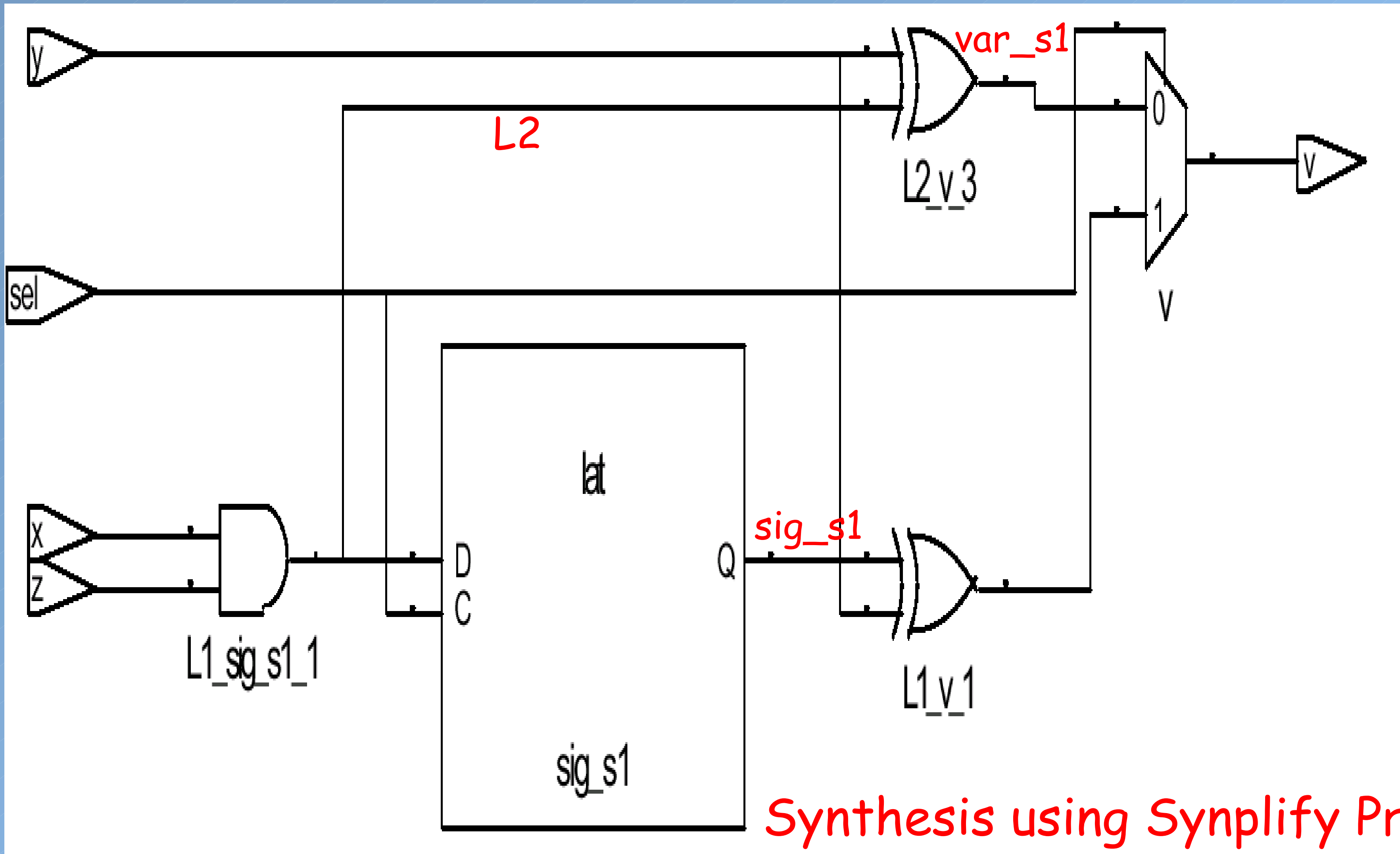
<u>Question</u>: Find the latches that will be inferred?

Inference Using Signals vs. Variables



Synthesis using Synopsys

Inference Using Signals vs. Variables



Synthesis using Synplify Pro

Latch Inference for Variables

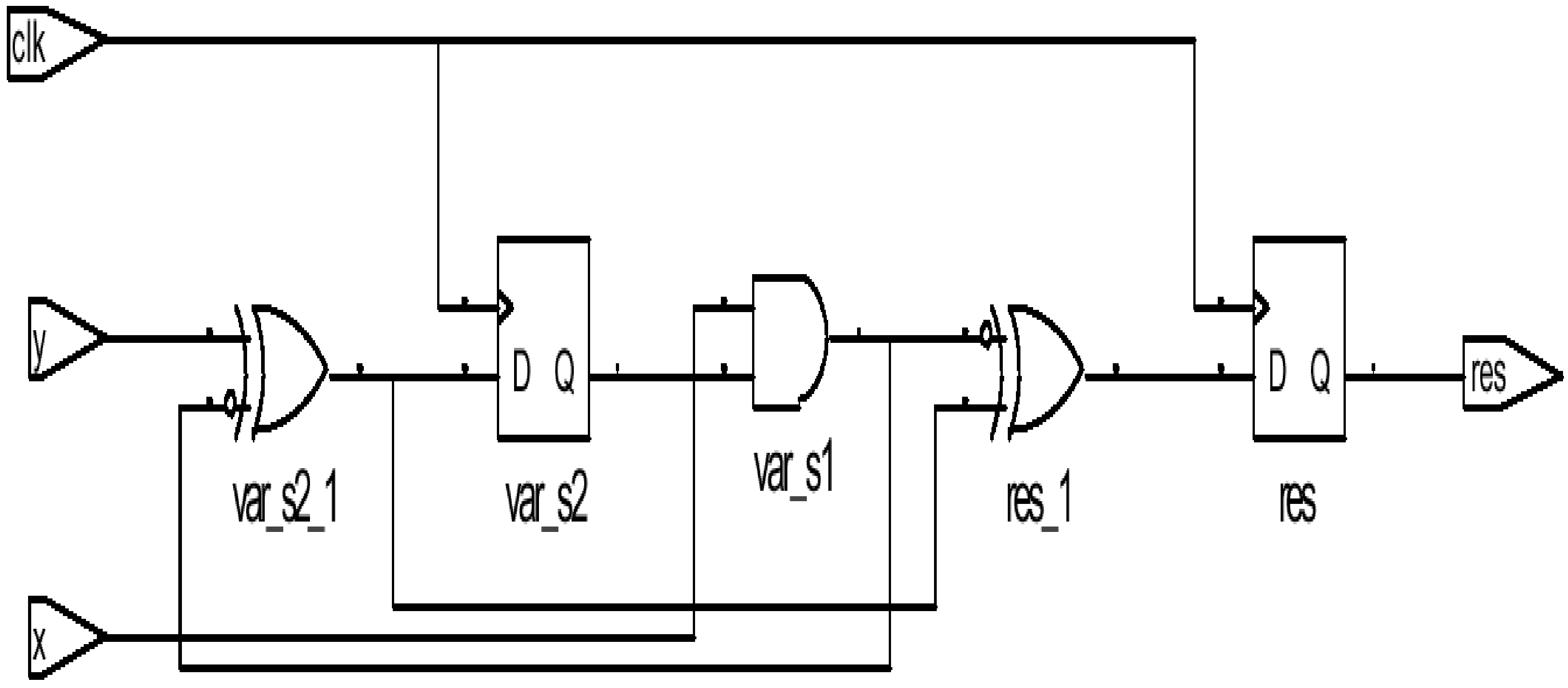
```
library IEEE;
use IEEE.std_logic_1164.all;
entity sigvar is
port(clk, x, y: in std_logic;
     res: out std_logic);
end entity sigvar;
architecture behavioral of sigvar is
begin
process
  variable var_s1, var_s2: std_logic;
begin
  wait until(rising_edge(clk));
  var_s1 := x nand var_s2;
  var_s2 := var_s1 xor y;
  res <= var_s1 xor var_S2;
end process;
end architecture;
```

<u>Question</u>: Find the latches that will be inferred

Latch Inference for Variables

- Signal `res`:
  - **`wait until`**`(rising_edge(clk));` statement causes a flip-flop.
  - When there is no rising_edge of the clock, the signal `res` should retain its value.
- Variable `var_s2`:
  - variable `var_s2` is used before it is defined.
  - Variables retain their values across process invocations.
  - Therefore, a flip-flop is inferred when a variable is used before it is defined.

Latch Inference for Variables



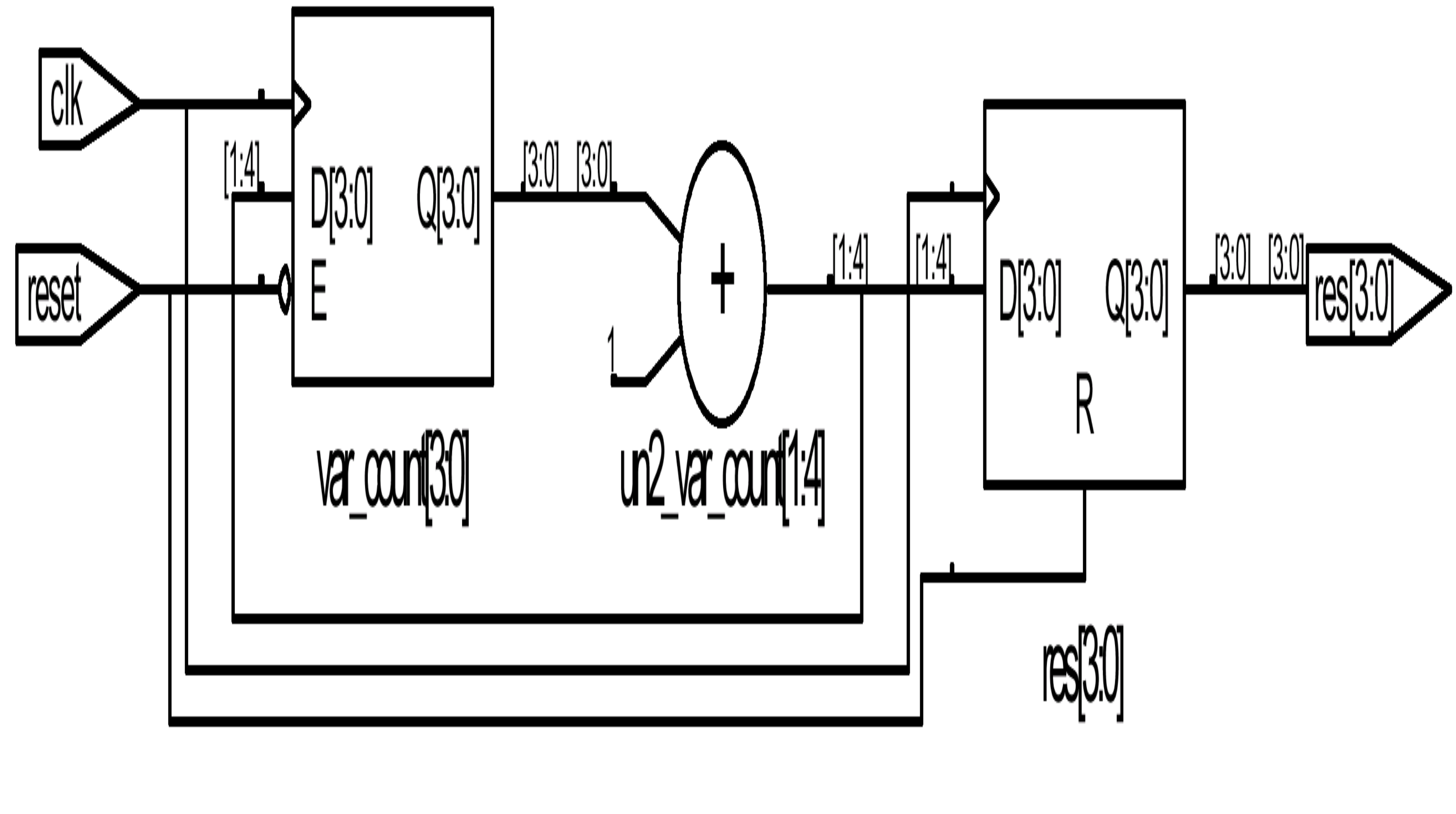
Latch and Flip-Flop Inference

- **`if`**, **`case`**, **`wait`**, conditional and selected signal assignment statements can be used to infer latches.
  - For example, "**`if`** `(sel = '1')` **`then`**" will lead to the inference of a latch.
- **`if`**`(rising_edge(clk))` **`then`** is used to infer an edge-triggered flip-flop.
  - `if(clk'event and clk = '1') then` may not detect a transition from '0' to '1'.
  - The attribute clk'last_value may be useful in detecting 0-to-1 transition.
  - However, it cannot be used in synthesis.
  - Use functions `rising_edge()` or `falling_edge()`.
  - The Xilinx XC4000 series FPGAs support both edge-triggered and level-sensitive devices.

Flip-Flops with Asynchronous Reset

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity counter_async is
port(clk, reset: in std_logic; res: out unsigned(3 downto 0));
end entity counter_async;
architecture behavioral of counter_async is
begin
process(clk, reset) is
  variable var_count: unsigned(3 downto 0);
begin
  if(reset='1') then res <= "0000";
  else
    if (rising_edge(clk)) then
      var_count := var_count + 1;
      res <= var_count;
    end if;
  end if;
end process;
end architecture;
```

Flip-Flops with Asynchronous Resets

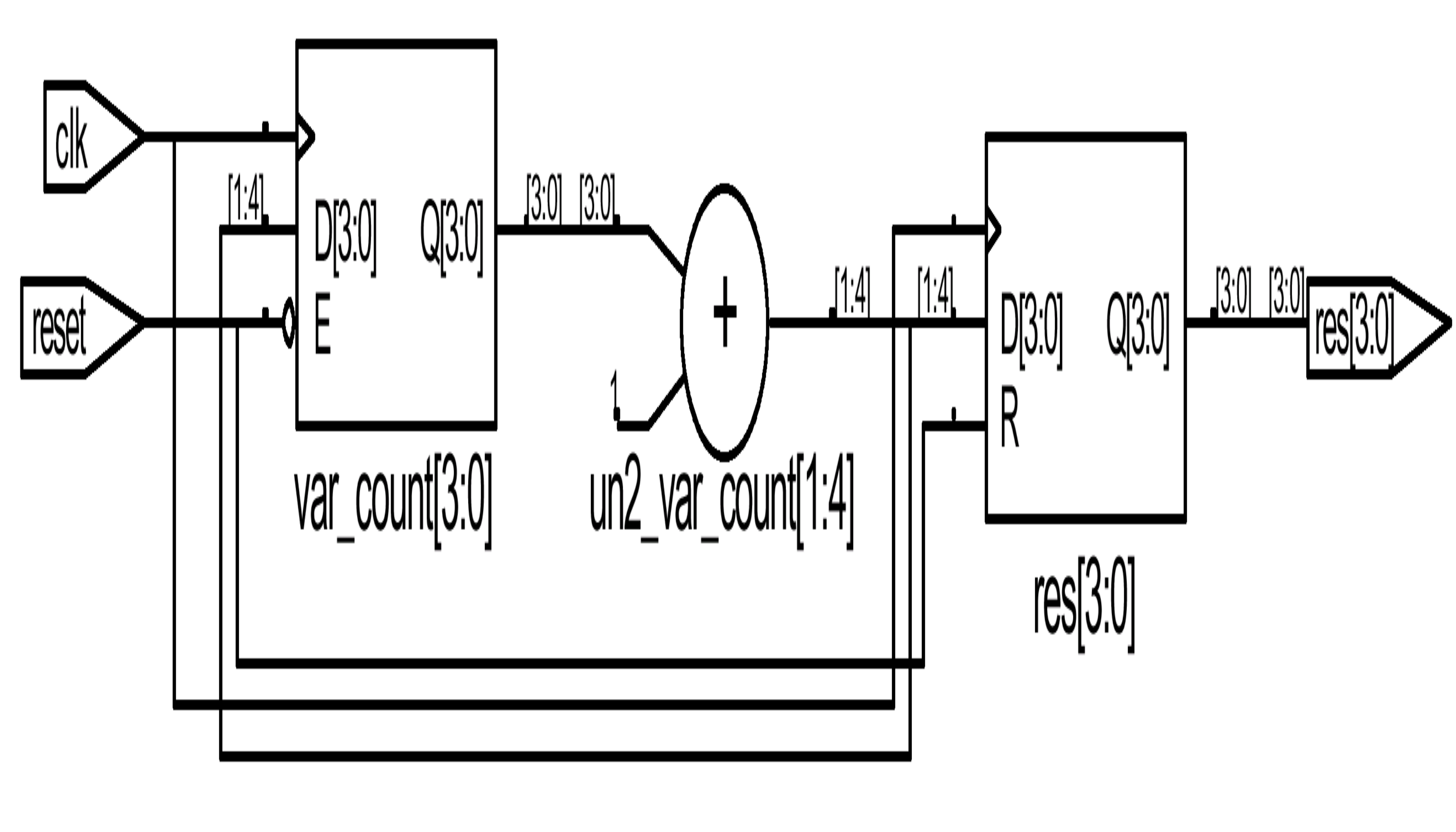


Asynchronous clear and set signal can be synthesized only if they exist in the target library.

Flip-Flops with Synchronous Resets

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity counter_async is
port(clk, reset: in std_logic; res: out unsigned(3 downto 0));
end entity counter_async;
architecture behavioral of counter_async 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;
      res <= var_count;
    end if;
  end if;
end process;
end architecture;
```

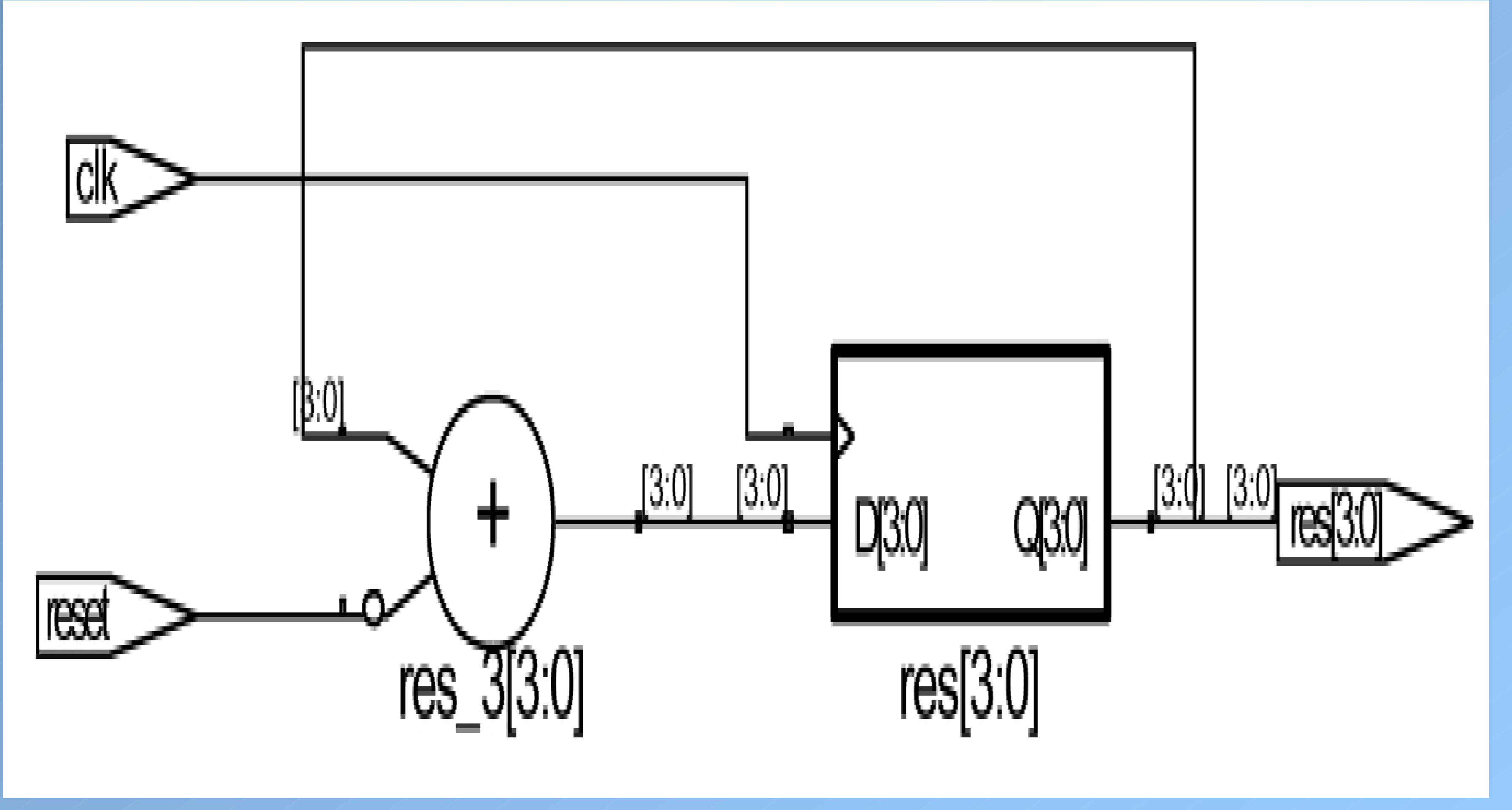
Flip-Flops with Synchronous Resets



Example in the Textbook

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity counter_async is
port(clk, reset: in std_logic; res: out unsigned(3 downto 0));
end entity counter_async;
architecture behavioral of counter_async 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;
```

Example in the Textbook



What is the reason for this difference?