VHDL
Modeling Behavior
from Synthesis Perspective
- Part B -

EL 310
Erkay Savaş
Sabancı University

The Wait Statement

- Syntax
  - `wait until condition;`
- Different forms
  - `wait until(clk'event and clk = '1');`
  - `wait until(rising_edge(clk));`
  - `wait until(falling_edge(clk));`
- It implies synchronous logic.
  - Consider a circuit where signal values are stored in edge-triggered flip-flops and their values are updated in the clock edge.
  - Therefore, all signals whose values are controlled by the **wait** statement will be synthesized into edge-triggered flip-flops.

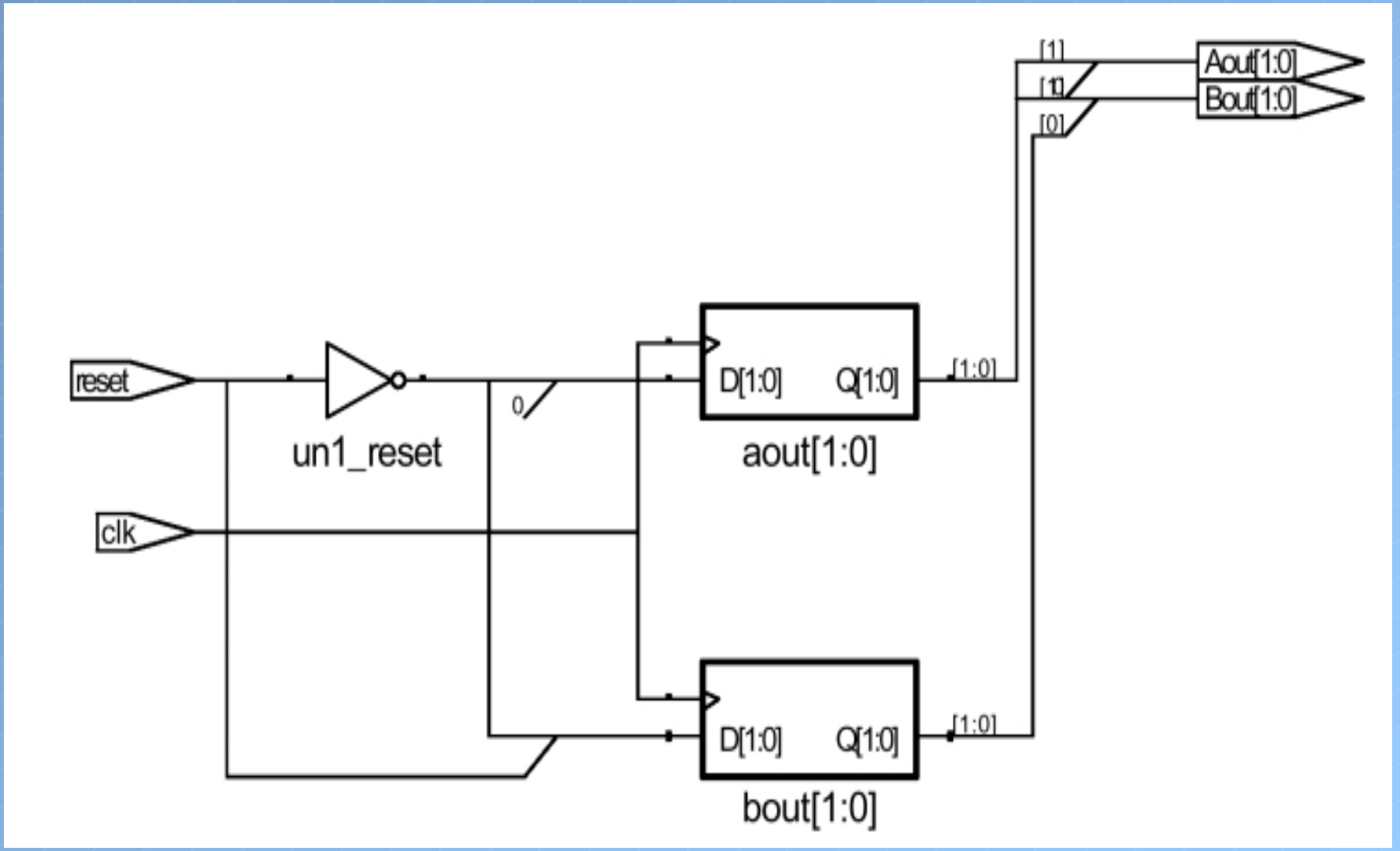
Wait vs. If-Then-Endif Statement

- Only one wait statement is permitted in a process and it must be the first statement.
  - Then, an edge-triggered flip-flop will be inferred for every signal in the process.
- We do not always want to generate flip-flops for every signal
  - Then we must use `if-then-endif` construct.
  - Edge detection expression limits the flip-flop inference to signals assigned in the body of `then` statement.
  - The remaining statements will lead to combinational logic.

Synchronous Logic with Wait Statement

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

entity edge is
port (reset, clk: in std_logic;
      Aout, Bout: out integer range 0 to 3);
end entity edge;

architecture behavioral of edge is
begin
  process
  begin
    wait until (rising_edge(clk));
    if reset = '1' then
      Aout <= 0;
      Bout <= 1;
    else
      Aout <= 2;
      Bout <= 2;
    end if;
 end process;

end architecture behavioral;
```

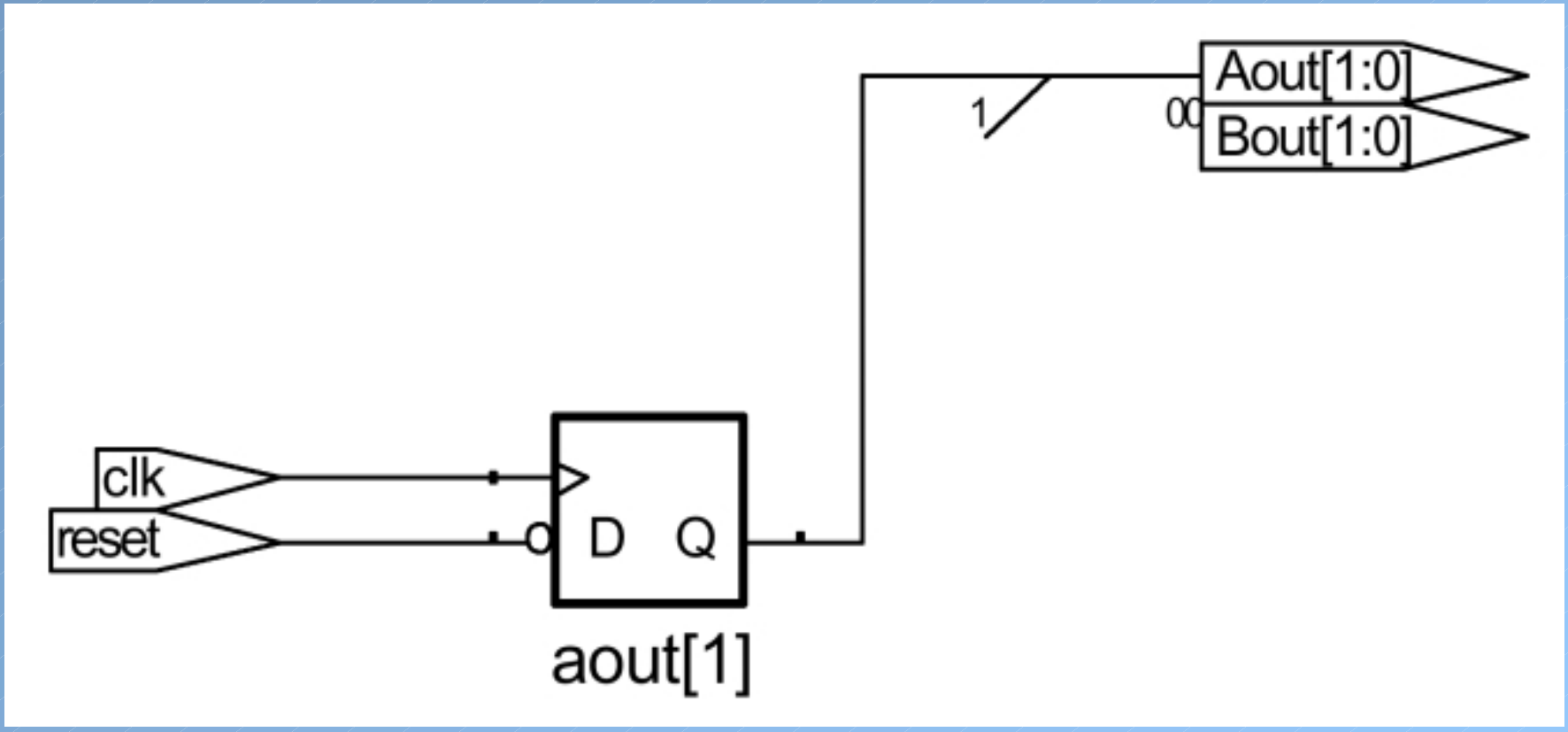
Synchronous Logic with Wait Statement



Example in the Textbook

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

entity edge is
port (reset, clk: in std_logic;
      Aout, Bout: out integer range 0 to 3);
end entity edge;

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;
```

Example in the Textbook

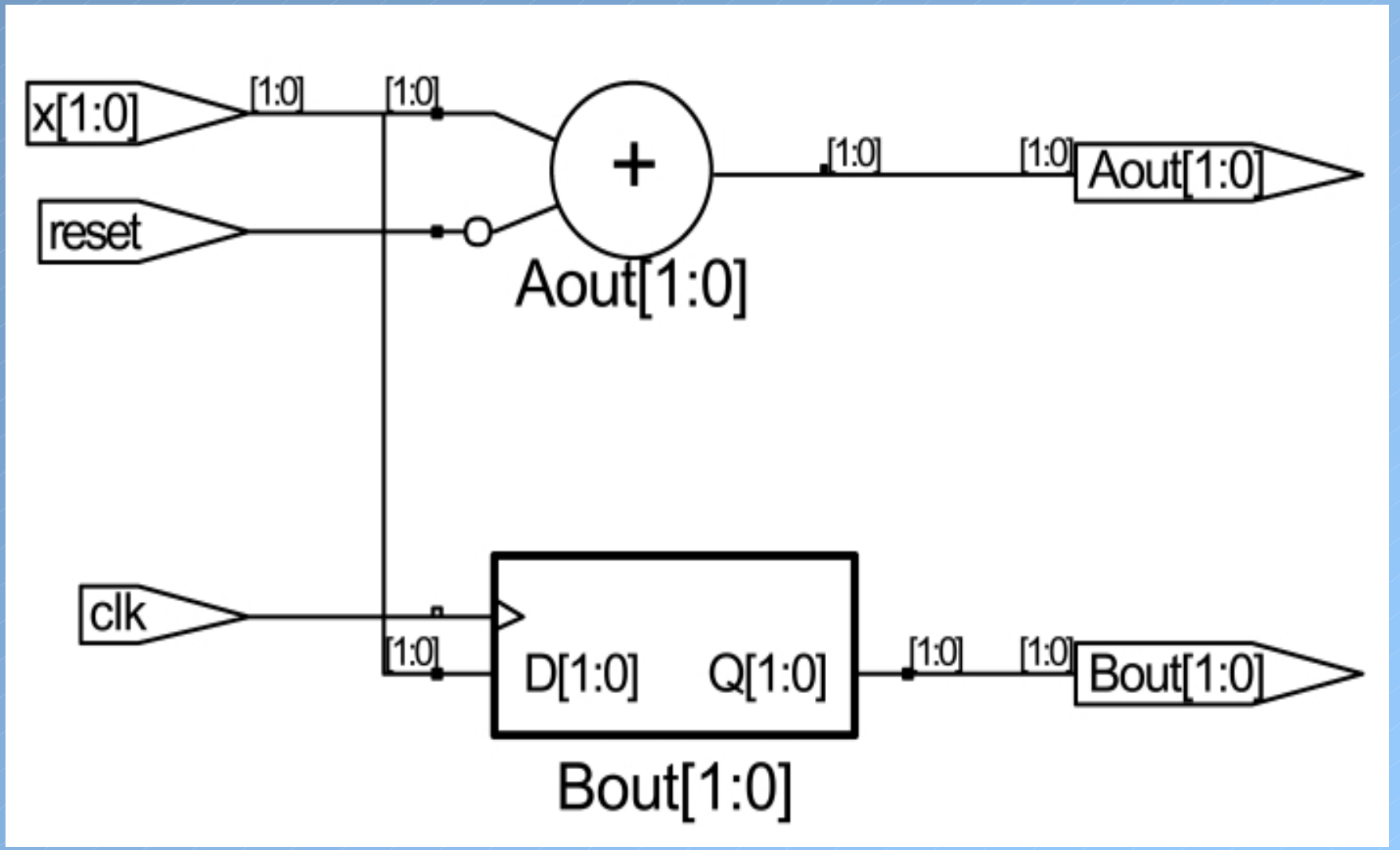


- Warning message gotten from the synthesis compiler:
- `All reachable assignments to bout(0) and bout(1) to assign '0', register removed by optimization.`

Synchronous Logic with If Statement

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

entity edge is
port (reset, clk: in std_logic;
      x: in integer range 0 to 3;
      Aout, Bout: out integer range 0 to 3);
end entity edge;

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

end architecture behavioral;
```

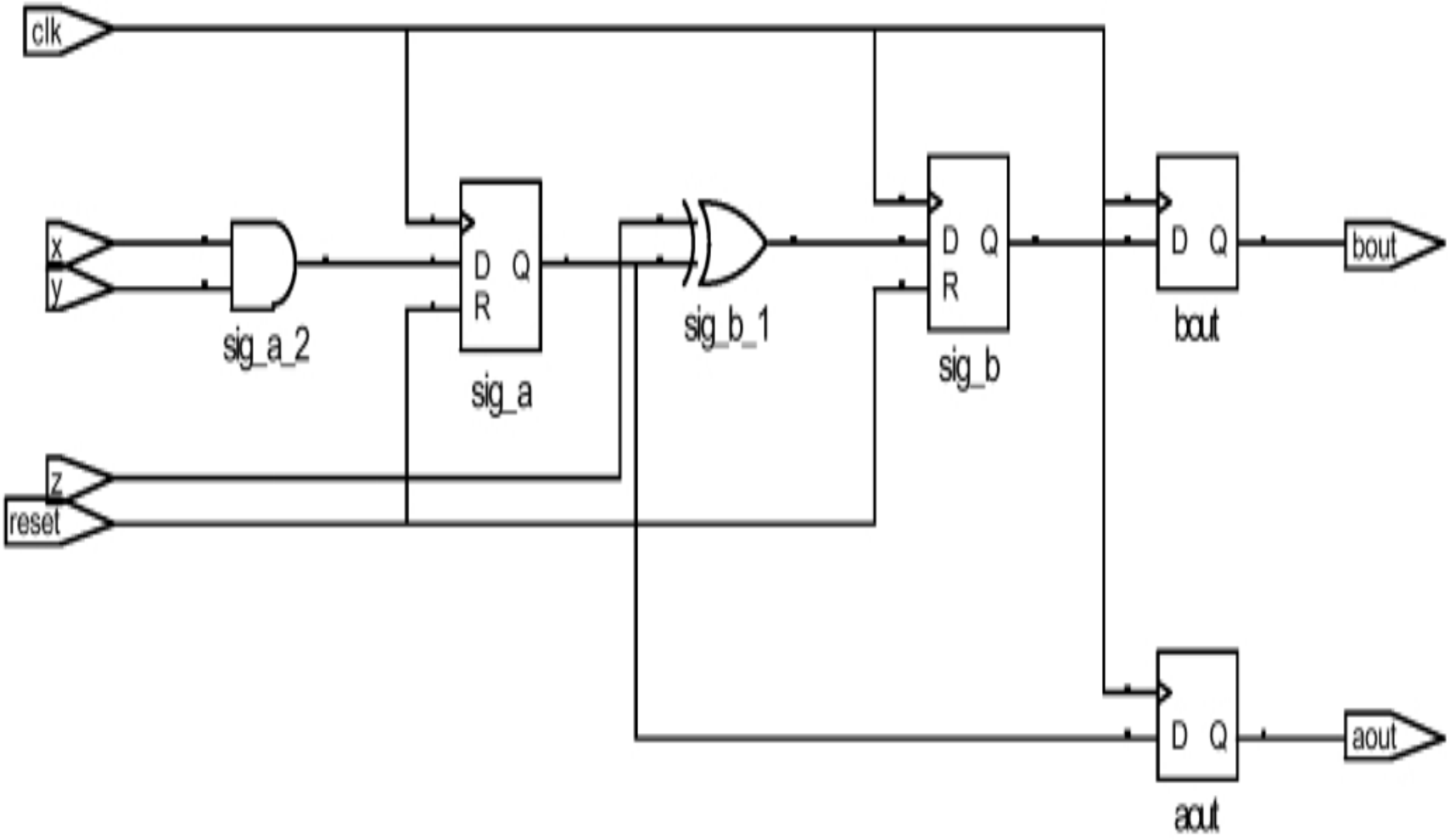
Synchronous Logic with If Statement



Synthesis vs. Simulation Semantics

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

entity semantics is
port (reset, clk, x, y, z: in std_logic;
      aout, bout: in std_logic);
end entity semantics;

architecture behavioral of semantics is
  signal sig_a, sig_b: std_logic;
begin
  process
  begin
    wait until (rising_edge(clk));
    if reset='1' then
      sig_a <= '0';
      sig_b <= '0';
    else
      sig_a <= x and y;
      sig_b <= sig_a xor z;
    end if;
    aout <= sig_a
    bout <= sig_b;
  end process;
end architecture behavioral;
```

Synthesis vs. Simulation Semantics

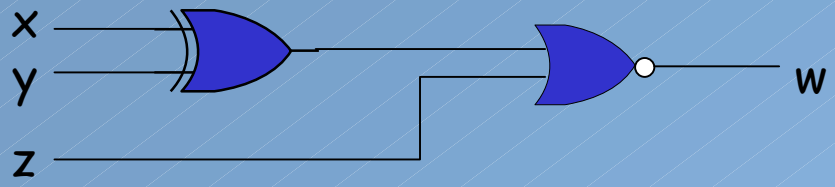


Simulation and Synthesis semantics match in this specific example

Synthesis vs. Simulation Semantics

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

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

architecture behavioral of semantics is
signal s1, s2: std_logic;
begin
  process(x, y, z) is
  begin
    L1: s1 <= x xor y;
    L2: s2 <= s1 or z;
    L3: w <= s1 nor s2;
  end process;
end architecture behavioral;
```

Synthesis vs. Simulation Semantics



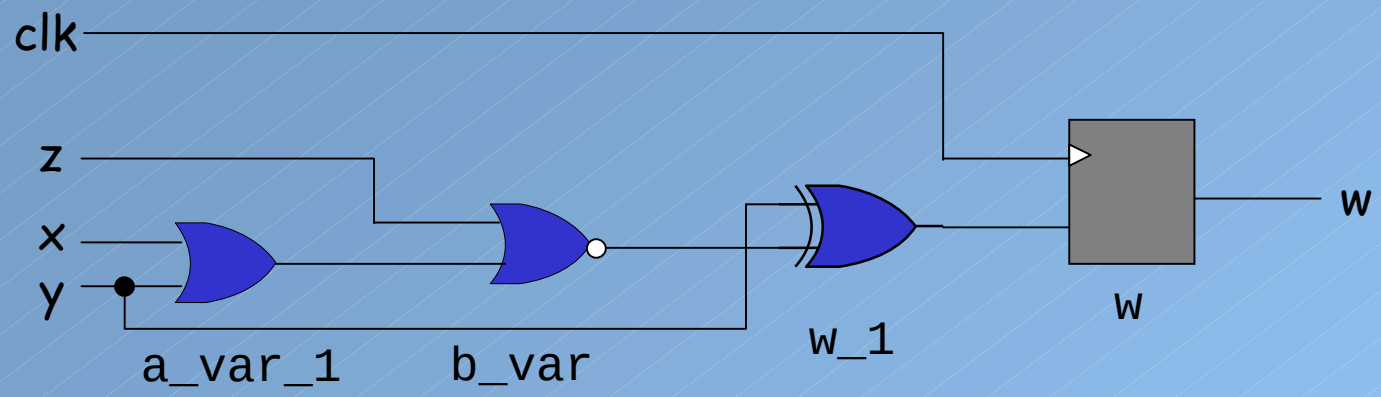
- $w = [(x \oplus y) + ((x \oplus y) + z)]' = [(x \oplus y) + z]'$

- Semantics of simulation and synthesis do not match
  - In behavioral simulation, the value of the signal s1 used in statement L2 is the one before this execution of process. This implies usage of a latch.
  - The synthesized logic is optimized and purely combinational

Variables with Wait Statement

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

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

architecture behavioral of var_wait is
begin
  process is
    variable a_var, b_var: std_logic;
  begin
    wait until (rising_edge(clk));
    L1: a_var := x or y;
    L2: b_var := a_var nor z;
    L3: w <= b_var xor y;
  end process;
end architecture behavioral;
```

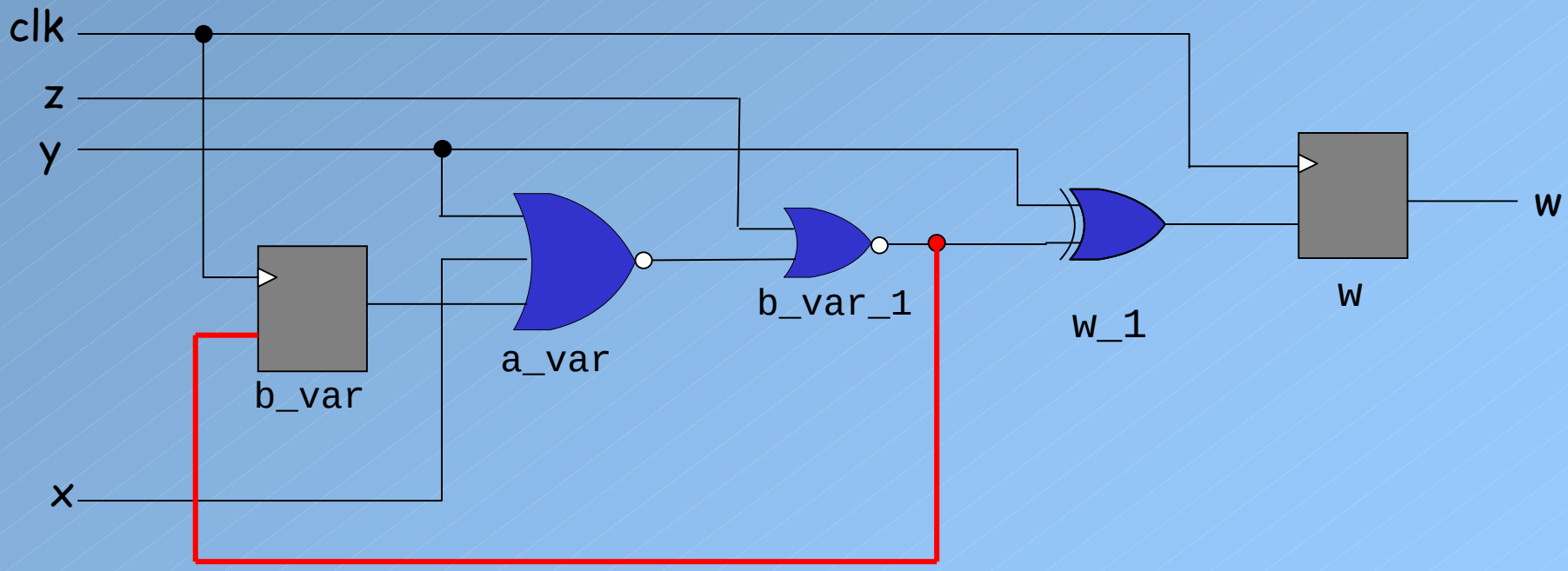
Variables with Wait Statement



- Unlike signals, variable assignments take affect immediately.
- Variables are usually collapsed into combinational logic unless a variable is used before it is defined.

Variables with Wait Statement

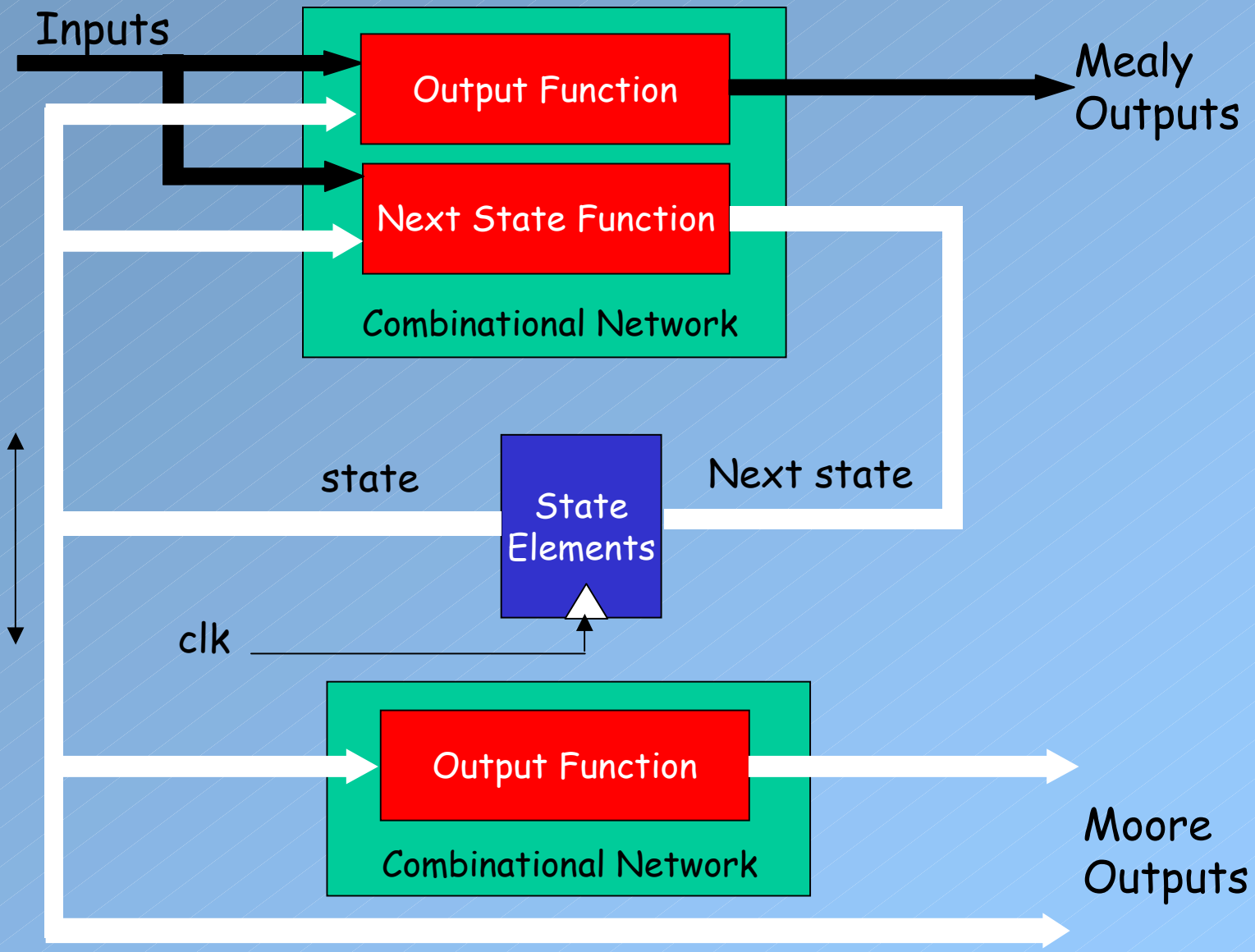
- Let us change the statement L1 in the previous example as follows:
  - `a_var := (x or y) nor b_var;`



Caveat

- We expect that level-sensitive or edge-sensitive expressions within the conditional part of the code determine whether latches or flip-flops are inferred.
- However, some synthesis compiler may still infer flip-flops regardless of whether the conditional expression is level- or edge-sensitive.

Synthesis of State Machines



Encoding of State Elements

- Function of number of states
  - For example, eight states ➔ 3 bits to represent each state uniquely.
  - Which state is numbered with what number?

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	111	100	10000000

most compact

FSM Compiler & Explorer in Synplify Pro

- FSM Compiler
  - automatically recognizes state machines in your design and optimizes them.
  - extracts the state machines as symbolic graphs, and then optimizes them by re-encoding the state representations
  - Optimization with respect to time and/or area.
  - and generating a better logic optimization starting point for the state machines.
- FSM Explorer
  - uses the state machines extracted by the FSM Compiler when it explores different encoding styles.

Encoding in Synplify Pro

- When FSM compiler is enabled
  - default encoding style automatically assigns encoding based on the number of states in the state machine as follows:
  - sequential for 0-4 states
  - One Hot for 5-24 states
  - gray for > 24 states
- Syn_state_directive allows to choose
  - sequential
  - gray
  - One Hot
  - Safe: default encoding & add reset logic to force the state machine to a known state if it reaches an invalid state.

State Machine Synthesis: Example

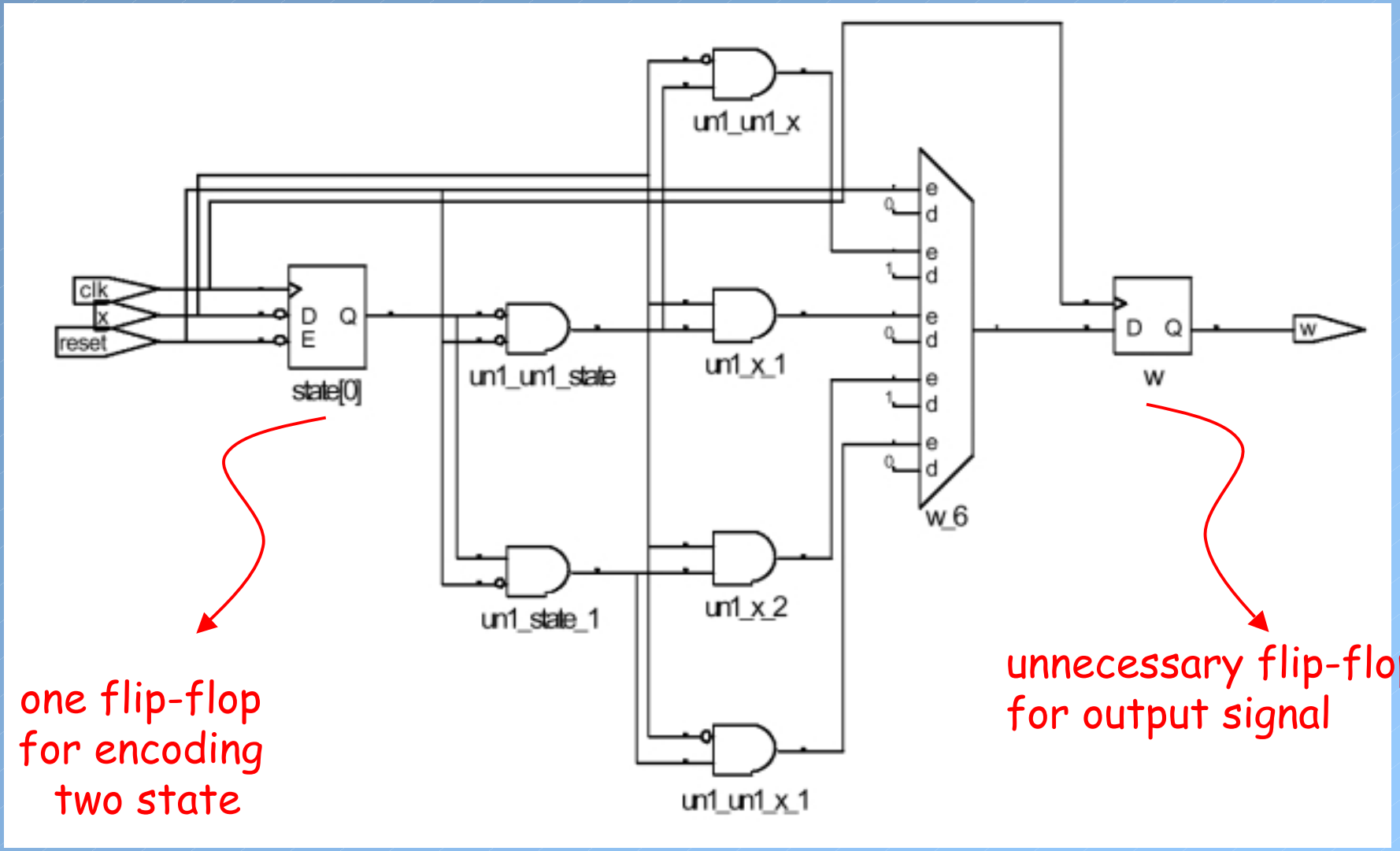
```
library IEEE;
use IEEE.std_logic_1164.all;

entity state_machine is
port (reset, x, clk: in std_logic;
      w: out std_logic);
end entity state_machine ;

architecture behavioral of state_machine is
  type state_type is (state0, state1);
  signal state, next_state: state_type;
begin
  process is
  begin
    wait until (rising_edge(clk));
    if reset = '1' then
      w <= '0';
    else
      ...
    end if;
  end process;
end architecture behavioral;
```

State Machine Synthesis: Example

```
...
process
  begin
    wait until (rising_edge(clk));
    if reset = '1' then
      w <= '0';
    else
      case state is
         when state0 =>
            if x = '0' then state <= state1; w <= '1';
            else state <= state0; w <= '0';
            end if;
         when state1 =>
            if x = '1' then state <= state0; w <= '1';
            else state <= state1; w <= '0';
            end if;
      end case;
    end if;
  end process;
end architecture behavioral;
```

This is a Mealy machine where output w is a function of both the state and the input.

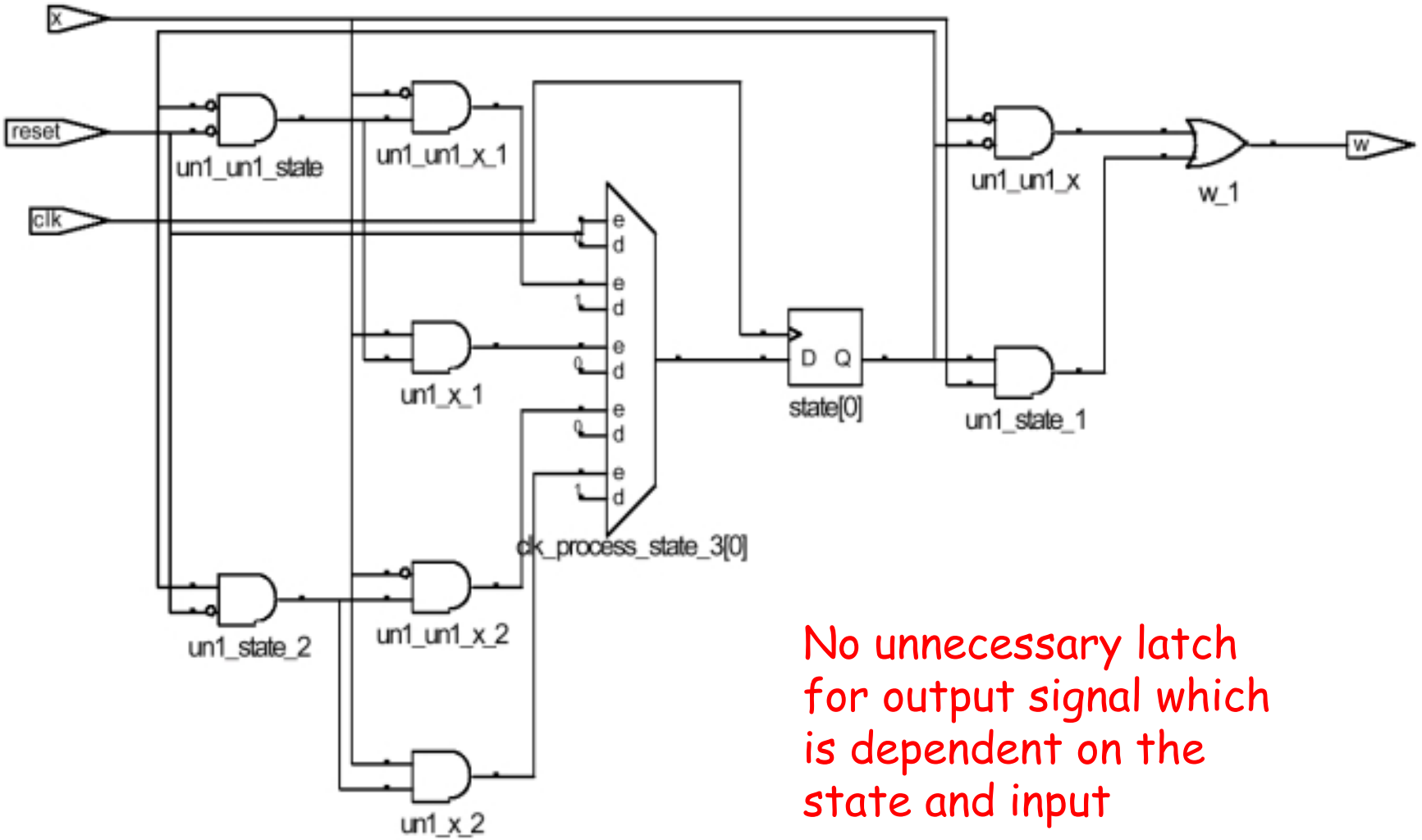
State Machine Synthesis: Example



State Machine Synthesis: Better VHDL

```
...
ns_process: process(state, x) is
begin
  case state is
    when state0 =>
      if x = '0' then next_state <= state1; w <= '1';
      else next_state <= state0; w <= 0;
      end if;
    when state1 =>
      if x = '0' then next_state <= state0; w <= '0';
      else next_state <= state1; w <= '1';
      end if;
  end case;
end process ns_process;


clk_process: process(reset, clk)
begin
  if (rising_edge(clk)) then
    if (reset = '1') then state <= state_type'left;
    else state <= next_state;
    end if;
  end if;
end process clk_process;

end architecture;
```

State Machine Synthesis: Better VHDL



No unnecessary latch for output signal which is dependent on the state and input

Another Example

```
entity sm1_2 is
  port (x, clk: in std_logic; z: out std_logic);
end entity sm1_2;

architecture behavioral of sm1_2 is
  subtype s_type is integer range 0 to 7;
  signal state, next_state: s_type;
  constant s0: s_type := 0;
  constant s1: s_type := 4;
  constant s2: s_type := 5;
  constant s3: s_type := 7;
  constant s4: s_type := 6;
  constant s5: s_type := 3;
  constant s6: s_type := 2;
begin

clk_process: process(clk)
begin
    if (rising_edge(clk)) then
      state <= next_state;
    end if;
end process clk_process;
```

Another Example

```
ns_process: process(state, x) is
begin
  z <= '0'; next_state <= s0;
  case state is
    when s0 =>
      if x = '0' then z <= '1'; next_state <= s1;
      else z <= '0'; next_state <= s2; end if;
    when s1 =>
      if x = '0' then z <= '1'; next_state <= s3;
      else z <= '0'; next_state <= s4; end if;
    when s2 =>
      if x = '0' then z <= '0'; next_state <= s4;
      else z <= '1'; next_state <= s4; end if;
    when s3 =>
      if x = '0' then z <= '0'; next_state <= s5;
      else z <= '1'; next_state <= s5; end if;
    when s4 =>
      if x = '0' then z <= '1'; next_state <= s5;
      else z <= '0'; next_state <= s6; end if;
    when s5 =>
      if x = '0' then z <= '0'; next_state <= s0;
      else z <= '1'; next_state <= s0; end if;
    when s6 =>
      if x = '0' then z <= '1'; next_state <= s0;
    when others => null;
  end case;
end process;
end architecture;
```

Null Statement

- a sequential statement that does not cause any action to take place; execution continues with the next statement.
  - In the previous example,
    ```
    when others => null;
    ```
    implies that no action needs to be performed when state is any other than specified in the case statement.
  - Recall that all choices must be covered in case statements.
  - Sometimes it is useful to explicitly say that no action needs to be performed
  - case or if statements.

Design Process

- Develop the VHDL models and use simulation to verify the functional correctness of the model.
- The model is then synthesized
- Synthesized model is simulated to verify the performance.
- Simulation of VHDL model and simulation model of the synthesized model may behave differently.
- This is what is called semantic mismatch.

Simulation vs. Synthesis

- Incomplete sensitivity list:
  - In VHDL model simulation, sensitivity list can include only a few signals and exclude the others.
  - Especially, if the process produces a combinational logic, then the circuit will be "sensitive" to a change on any of the signals that are manipulated in the process.
  -
```
process (sel) is
begin
   if(sel='1' and En='0') then A<='1';
   else A<= '0';
   end if;
end process;
```
  - The synthesized circuit will also be sensitive to the signal En.

Simulation vs. Synthesis

- Sequential signal assignments in a process
  - the code below show sequential behavior in VHDL simulation
  -
```
process (x, y, z)
begin
   L1: s1 <= x xor y;
   L2: s2 <= s1 or z;
   L3: w <= s1 nor s2;
end process;
```
  - However, synthesis compiler will generally optimize this sequence to produce combinational logic and avoid latches.

Simulation vs. Synthesis

- User specified delays may not match the actual delays in the synthesized circuit.
- Simulation Overhead:
  - Conditional and selected CSA can be modeled using process.
  - Since CSAs are always active, it has more simulation overhead.
  - CSAs, on the other hand, are better for synthesis.
- Speed:
  - Use of variables in process will result in faster simulation.
  - Use of process may obscure the concurrency within the process and may reduce the effectiveness of the inference mechanisms.

Synthesis Hints

- The while-loop statement may not be supported by some synthesis compiler, since iteration number is data dependent.
- All for loop indices must have statically determinable loop ranges.
- In order to avoid inference of an unnecessary latch for a signal, every execution path through the process must assign a value for that signal.
- A latch will be inferred for a variable if it is used before it is defined.
- Initialize your signals (if you must do so) explicitly under the control of reset signal. Otherwise, initialization may be ignored by the compiler.

Synthesis Hints

- Include all signals in a process in the sensitivity list of the process.
- The VHDL code should imply hardware. Avoid purely algorithmic descriptions of hardware.
- To avoid latches for a signal that appears in conditional statements, make sure that default values are assigned to this signals before a conditional block of code.
- Try to minimize dependencies between statements.
- Using don't care values to cover **when others** case in a **case** statement can enable the compiler to optimize the logic. Try to avoid setting signals to specific values "0000" or "1111" in `when others`.

Synthesis Hints

- If possible try to specify data ranges for signals.
- Minimize signal assignment statements in a process and use variables.
- Use `if-then-else` statements to infer flip-flops rather than `wait` statements. To infer flip-flop with a wait statement necessitates that wait statement must be the first and only wait statement in the process; thus leading to flip-flop inference for every signal in the process.
- Do not use `don't care` symbols in comparisons. Such comparisons will always return FALSE.
- Check vendor specific constraints on the permitted types and range for the for-loop index

Synthesis Hints

- Move common complex operations out of the branches of conditional statements.
- Using a `case` statement rather than `if-then-elsif` will produce less logic since priority logic will have to be generated for the latter.
- The choice of coding style must be guided by the building blocks that are available in the target technology. For example, if latches are not available, then level sensitive expressions will lead to the synthesis of latches in gate level equivalents. This will complicate the circuit and timing analysis.