

Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 12 (and 13)

Sequential Code

Revision 1

Book Contents

Part I: Digital Circuits Review

1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

Part II: VHDL

5. Introduction to VHDL
6. Code Structure and Composition
7. Predefined Data Types
8. User-Defined Data Types
9. Operators and Attributes
10. Concurrent Code
11. Concurrent Code – Practice
12. Sequential Code
13. Sequential Code – Practice
14. Packages and Subprograms
15. The Case of State Machines
16. The Case of State Machines – Practice
17. Additional Design Examples
18. Intr. to Simulation with Testbenches

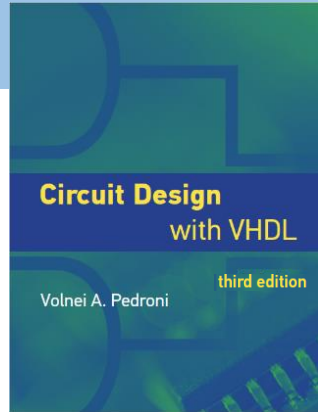
Appendices

- A. Vivado Tutorial
- B. Quartus Prime Tutorial
- C. ModelSim Tutorial
- D. Simulation Analysis and Recommendations
- E. Using Seven-Segment Displays with VHDL
- F. Serial Peripheral Interface
- G. I2C (Inter Integrated Circuits) Interface
- H. Alphanumeric LCD
- I. VGA Video Interface
- J. DVI Video Interface
- K. TMDS Link
- L. Using Phase-Locked Loops with VHDL
- M. List of Enumerated Examples and Exercises

VHDL for Synthesis Slides

| Chapter | Title |
|---------|--------------------------------|
| 5 | Introduction to VHDL |
| 6 | Code Structure and Composition |
| 7 | Predefined Data Types |
| 8 | User-Defined Data Types |
| 9 | Operators and Attributes |
| 10 | Concurrent Code |
| 11 | Concurrent Code – Practice |
| 12 | Sequential Code |
| 13 | Sequential Code – Practice |
| 14 | Packages and Subprograms |





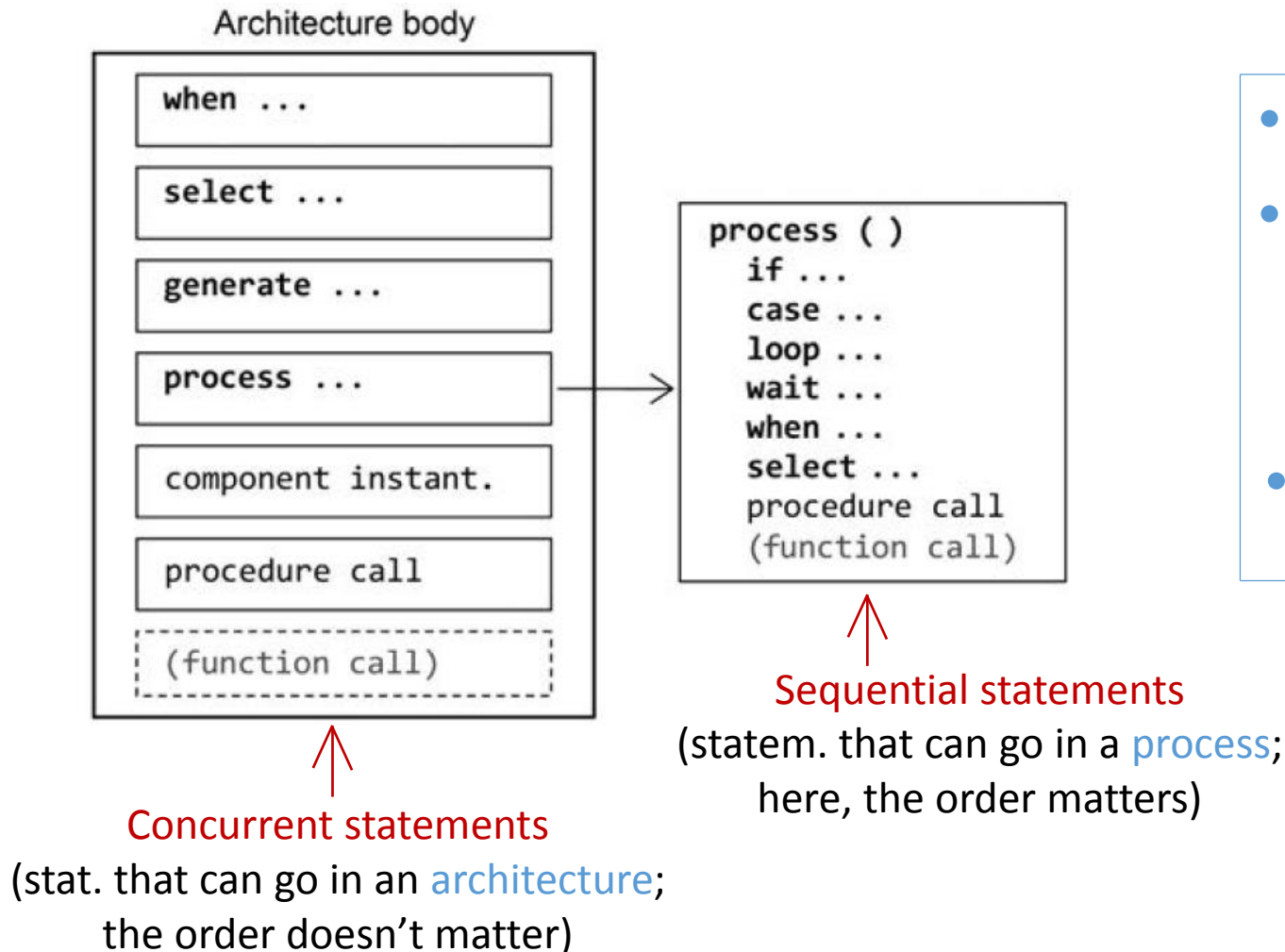
Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal versus Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

1. Sequential statements

1. Sequential statements

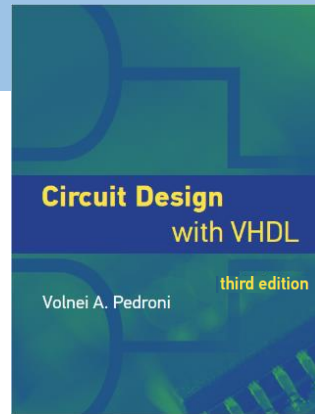


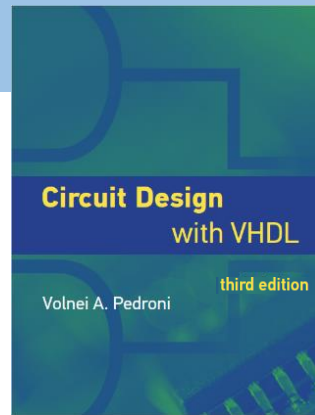
- Some statements are concurrent *and* sequential
- Statements left out:
 - block* → because of its little use
 - assert* → doesn't generate hardware (will be seen in ch. 14)
- *Function call* included for clarity (it's not a statement but rather part of an expression)

1. Sequential statements

- Used inside `process` and subprograms (i.e., `function`^(*) and `procedure`)
- Can implement both `sequential` and `combinational` circuits
- Sequential code allows the use of `variables` (a major feature of VHDL)

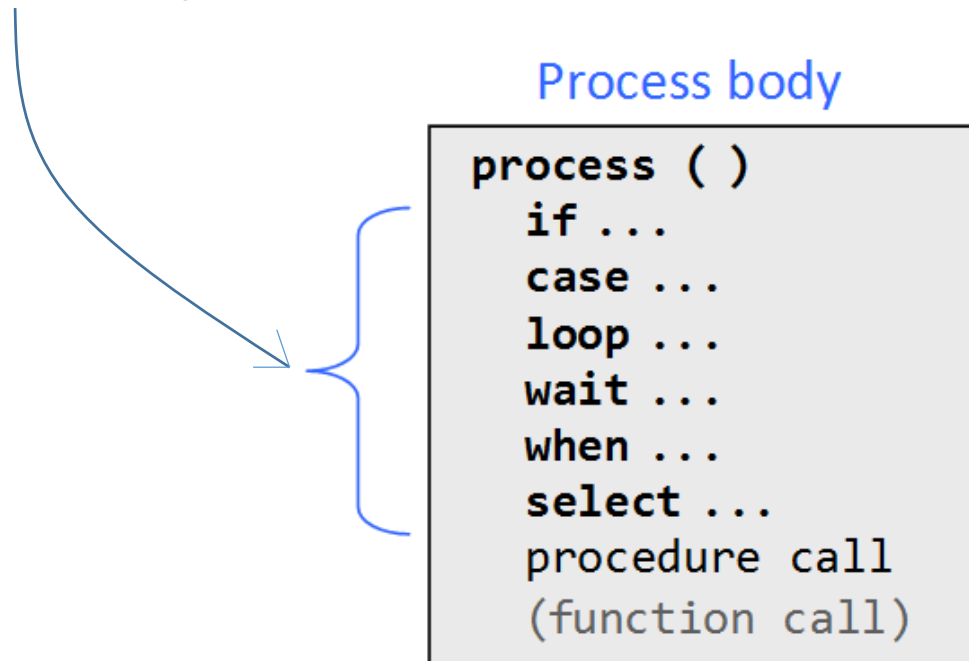
^(*) Exception: `wait` cannot be used in `function`



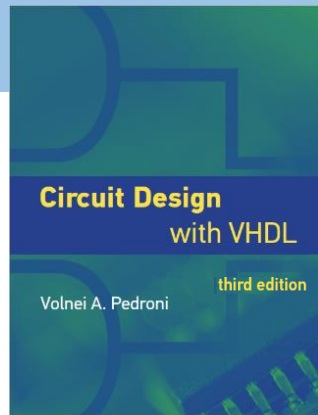


1. Sequential statements

- Used inside `process` and subprograms (i.e., `function`^(*) and `procedure`)
- Can implement both `sequential` and `combinational` circuits
- Sequential code allows the use of `variables` (a major feature of VHDL)
- **Fundamental sequential statements**



^(*) Exception: `wait` cannot be used in `function`

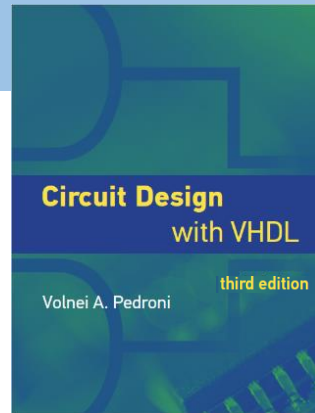


Chapters 12-13

Sequential Code

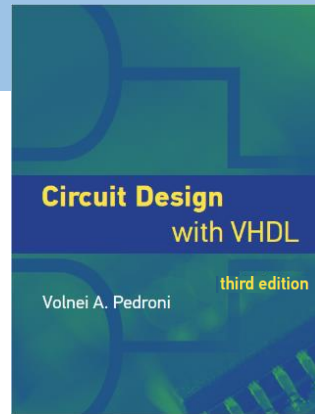
1. Sequential statements
- ➔ 2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

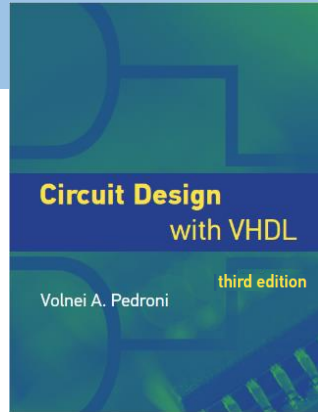
2. Detecting clock transitions (before we start ...)



2. Detecting clock transitions (before we start ...)

- Clock is a **major** component of sequential circuits
(By the way, are all sequential circuits clocked?)





2. Detecting clock transitions (before we start ...)

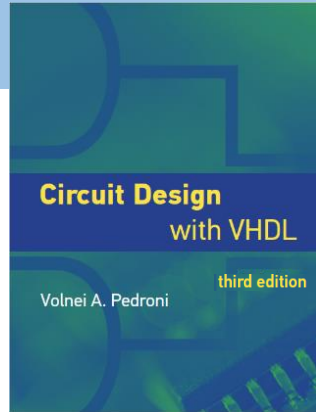
- Clock is a **major** component of sequential circuits
(By the way, are all sequential circuits clocked?)
- Clock transitions can be detected as follows:

Option 1:

```
if clk'event and clk='1' then ...    --positive clock transition
if clk'event and clk='0' then ...    --negative clock transition
```

Option 2 (recommended):

```
if rising_edge(clk) then ...         --positive clock transition
if falling_edge(clk) then ...        --negative clock transition
```



2. Detecting clock transitions (before we start ...)

- Clock is a **major** component of sequential circuits
(By the way, are all sequential circuits clocked?)

- Clock transitions can be detected as follows:

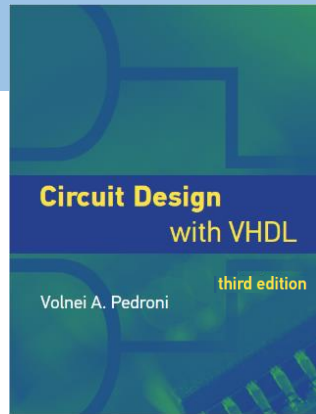
Option 1:

```
if clk'event and clk='1' then ...    --positive clock transition
if clk'event and clk='0' then ...    --negative clock transition
```

Option 2 (recommended):

```
if rising_edge(clk) then ...         --positive clock transition
if falling_edge(clk) then ...        --negative clock transition
```

- Option 2 uses the `'event` attribute plus the `To_X01` function (sec. 7.10.4), exploring all `std_ulogic` values



2. Detecting clock transitions (before we start ...)

- Clock is a **major** component of sequential circuits
(By the way, are all sequential circuits clocked?)

- Clock transitions can be detected as follows:

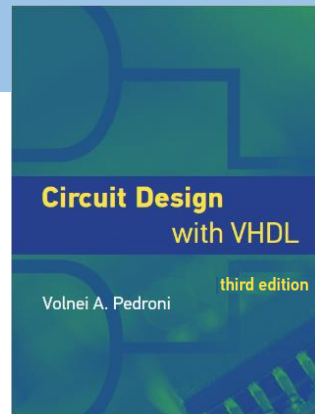
Option 1:

```
if clk'event and clk='1' then ...    --positive clock transition
if clk'event and clk='0' then ...    --negative clock transition
```

Option 2 (recommended):

```
if rising_edge(clk) then ...         --positive clock transition
if falling_edge(clk) then ...        --negative clock transition
```

- Option 2 uses the 'event attribute plus the *To_X01* function (sec. 7.10.4), exploring all *std_ulogic* values
- A good design uses only values '0' and '1' for clock, but option 2 makes up for bad practices (like the use of 'L' and 'H')



2. Detecting clock transitions (before we start ...)

- Clock is a **major** component of sequential circuits
(By the way, are all sequential circuits clocked?)

- Clock transitions can be detected as follows:

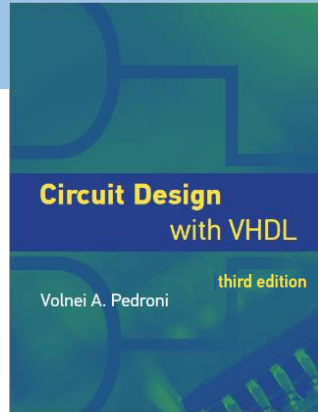
Option 1:

```
if clk'event and clk='1' then ...    --positive clock transition
if clk'event and clk='0' then ...    --negative clock transition
```

Option 2 (recommended):

```
if rising_edge(clk) then ...         --positive clock transition
if falling_edge(clk) then ...        --negative clock transition
```

- Option 2 uses the `'event` attribute plus the `To_X01` function (sec. 7.10.4), exploring all `std_ulogic` values
- A good design uses only values `'0'` and `'1'` for clock, but option 2 makes up for bad practices (like the use of `'L'` and `'H'`)
- Option 2 is predefined for `std_ulogic/std_logic`, `bit`, and `boolean`

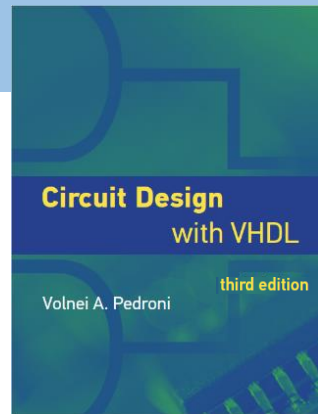


Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
- ➔ 3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

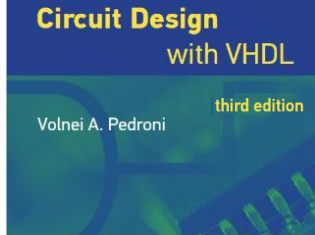
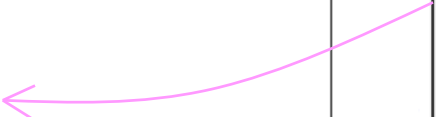
3. The *process* statement

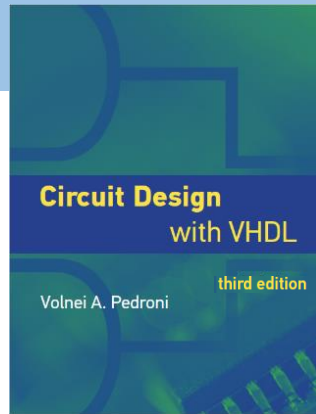


3. The *process* statement

```
[label:] process [sensitivity_list] [is]  
    [declarative_part]  
begin  
    sequential_statement_part  
end process [label];
```

```
process ( )  
    if ...  
    case ...  
    loop ...  
    wait ...  
    when ...  
    select ...  
    procedure call  
    (function call)
```



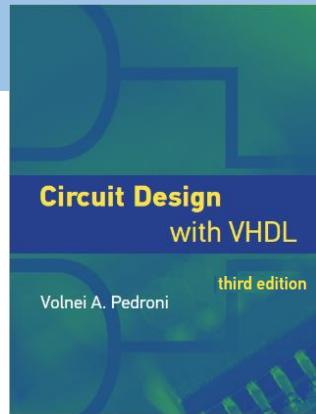


3. The *process* statement

```
[label:] process [sensitivity_list] [is]
  [declarative_part]
begin
  sequential_statement_part
end process [label];
```

```
process ( )
  if ...
  case ...
  loop ...
  wait ...
  when ...
  select ...
  procedure call
  (function call)
```

- The **declarative part** can contain declarations of *type*, *constant*, *variable*, ... (declaration of *signal* is **forbidden**)
- The **sequential statements part** can contain only **sequential statements** (listed above), plus **operators** (because these can go in any code)

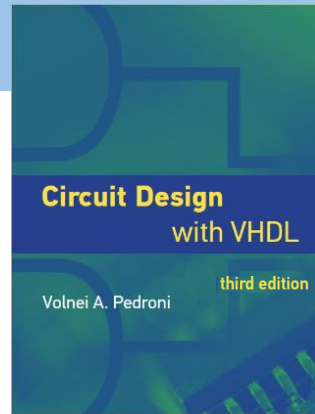


3. The *process* statement

```
[label:] process [sensitivity_list] [is]
  [declarative_part]
begin
  sequential_statement_part
end process [label];
```

```
process ( )
  if ...
  case ...
  loop ...
  wait ...
  when ...
  select ...
  procedure call
  (function call)
```

- The **declarative part** can contain declarations of *type*, *constant*, *variable*, ... (declaration of *signal* is **forbidden**)
- The **sequential statements part** can contain only **sequential statements** (listed above), plus **operators** (because these can go in any code)
- The process **wakes** (executes) when a signal in the sensitivity list changes

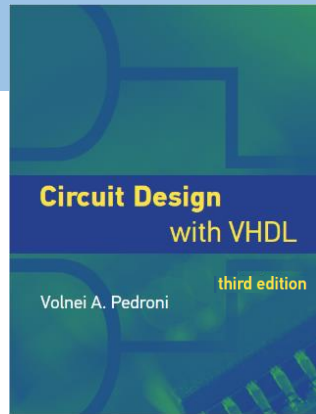


3. The *process* statement

```
[label:] process [sensitivity_list] [is]
  [declarative_part]
begin
  sequential_statement_part
end process [label];
```

```
process ( )
  if ...
  case ...
  loop ...
  wait ...
  when ...
  select ...
  procedure call
  (function call)
```

- The **declarative part** can contain declarations of *type*, *constant*, *variable*, ... (declaration of *signal* is **forbidden**)
- The **sequential statements part** can contain only **sequential statements** (listed above), plus **operators** (because these can go in any code)
- The process **wakes** (executes) when a signal in the sensitivity list changes
- The process suspends automatically when it reaches the **end process** keywords, **even if the awaking condition is still true**



3. The *process* statement

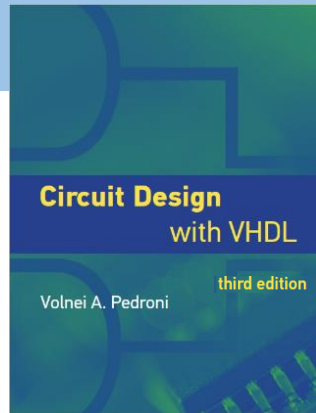
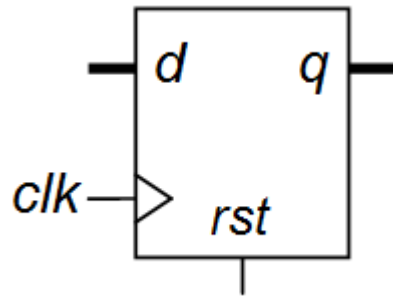
```
[label:] process [sensitivity_list] [is]
  [declarative_part]
begin
  sequential_statement_part
end process [label];
```

```
process ( )
  if ...
  case ...
  loop ...
  wait ...
  when ...
  select ...
  procedure call
  (function call)
```

- The **declarative part** can contain declarations of *type*, *constant*, *variable*, ... (declaration of *signal* is **forbidden**)
- The **sequential statements part** can contain only **sequential statements** (listed above), plus **operators** (because these can go in any code)
- The process **wakes** (executes) when a signal in the sensitivity list changes
- The process suspends automatically when it reaches the **end process** keywords, **even if the awaking condition is still true**
- If the process contains a (explicit) *wait* statement, the sensitivity list must be omitted (that *wait* statement is then responsible for waking the process)

3. The *process* statement

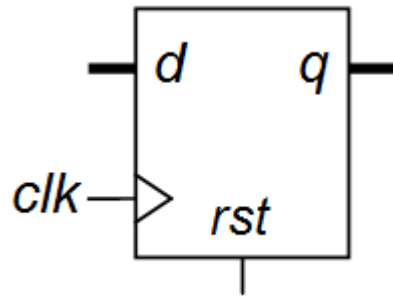
Example: DFF with asynchronous reset
(without wait)



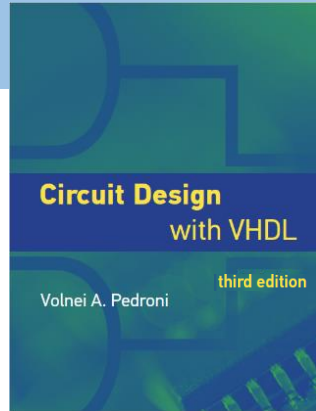
3. The *process* statement

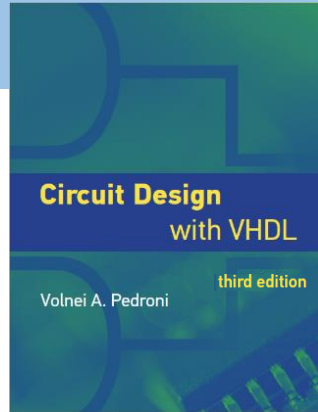
Example: DFF with asynchronous reset

(without wait)



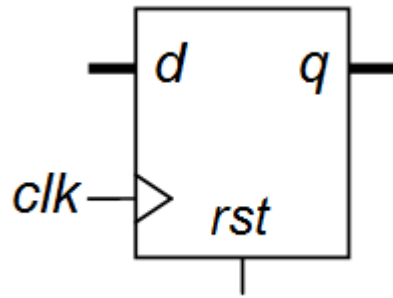
```
process (clk, rst)
begin
    if rst then
        q <= (others => '0');
    elsif rising_edge(clk) then
        q <= d;
    end if;
end process;
```





3. The *process* statement

Example: DFF with asynchronous reset
(without wait)



```
process (clk, rst)
begin
  if rst then
    q <= (others => '0');
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

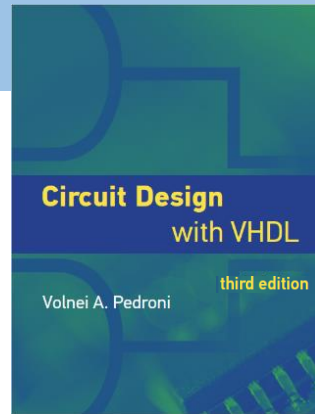
Example: A simpler DFF
(with wait)

```
process
begin
  wait until falling_edge(clk); --or wait until not clk
  q <= d;
end process;
```

3. The *process* statement

The *all* keyword:

- Recommended for the sensitivity list of combinational circuits

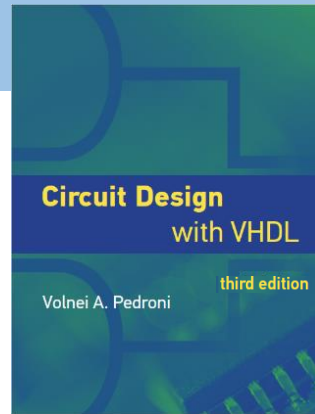


3. The *process* statement

The *all* keyword:

- Recommended for the sensitivity list of combinational circuits

Example: Consequences of incomplete sensitivity list



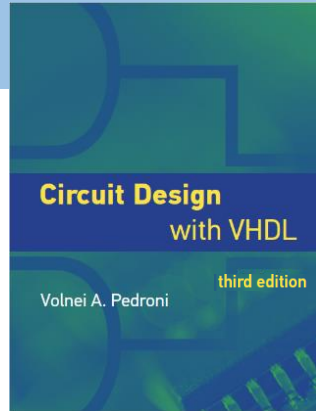
3. The *process* statement

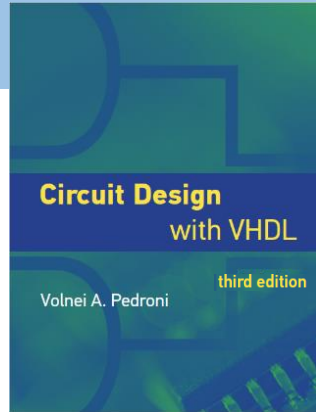
The *all* keyword:

- Recommended for the sensitivity list of combinational circuits

Example: Consequences of incomplete sensitivity list

Desired circuit:





3. The *process* statement

The *all* keyword:

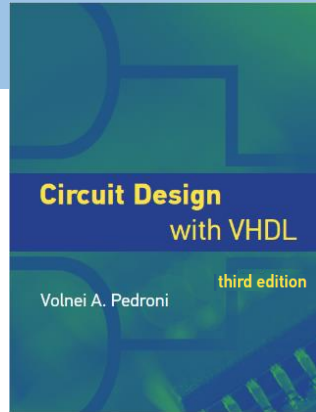
- Recommended for the sensitivity list of combinational circuits

Example: Consequences of incomplete sensitivity list

Desired circuit:



```
--Right code (if
--sequential):
process (all)
begin
    c <= a and b;
end process;
```



3. The *process* statement

The *all* keyword:

- Recommended for the sensitivity list of combinational circuits

Example: Consequences of incomplete sensitivity list

Desired circuit:



```
--Right code (if  
--sequential):  
process (all)  
begin  
    c <= a and b;  
end process;
```

Resulting circuit:

```
--Written code:  
process (a)  
begin  
    c <= a and b;  
end process;
```

3. The *process* statement

The *all* keyword:

- Recommended for the sensitivity list of combinational circuits

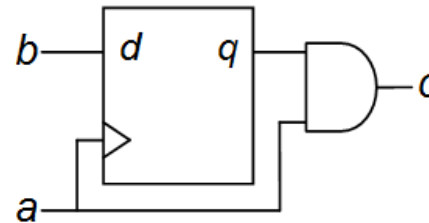
Example: Consequences of **incomplete sensitivity list**

Desired circuit:

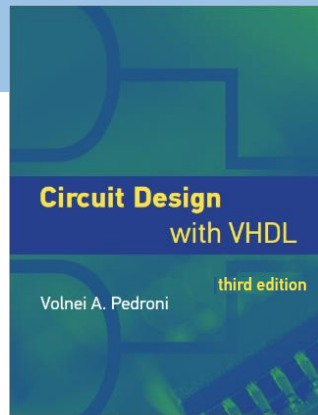


```
--Right code (if
--sequential):
process (all)
begin
    c <= a and b;
end process;
```

Resulting circuit:



```
--Written code: BAD!
process (a)
begin
    c <= a and b;
end process;
```

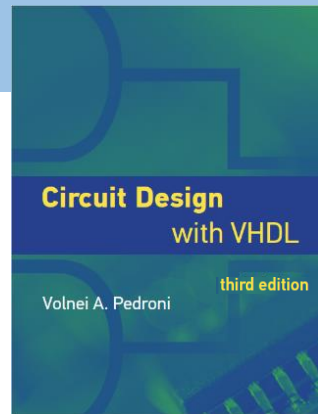


Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
- ➔ 4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

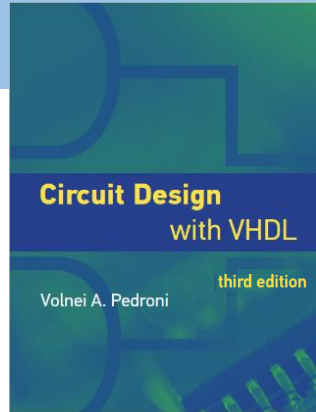
4. The *if* statement

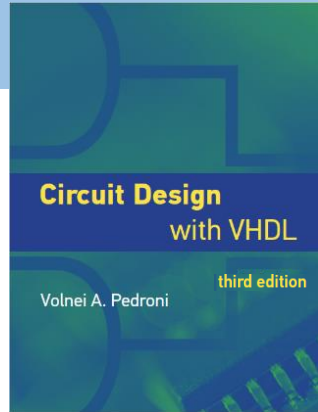


4. The *if* statement

```
[label:] if condition then  
    sequential_statements;  
[elsif  
    sequential_statements;  
[else  
    sequential_statements;  
end if [label];
```

- *if* is the sequential counterpart of the concurrent *when* statement

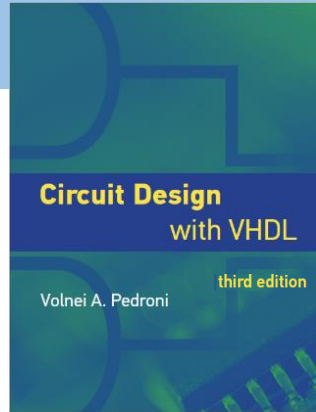




4. The *if* statement

```
[label:] if condition then  
    sequential_statements;  
[elsif  
    sequential_statements;  
[else  
    sequential_statements;  
end if [label];
```

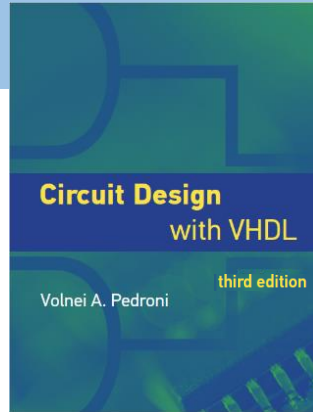
- *if* is the sequential counterpart of the concurrent *when* statement
- Like *when*, it has a priority-encoding nature, so use it only when there are few conditions to test



4. The *if* statement

```
[label:] if condition then  
    sequential_statements;  
[elsif  
    sequential_statements;]  
[else  
    sequential_statements;]  
end if [label];
```

- *if* is the sequential counterpart of the concurrent *when* statement
- Like *when*, it has a priority-encoding nature, so use it only when there are few conditions to test
- To enter truth tables in general (in sequential code), use *case* instead



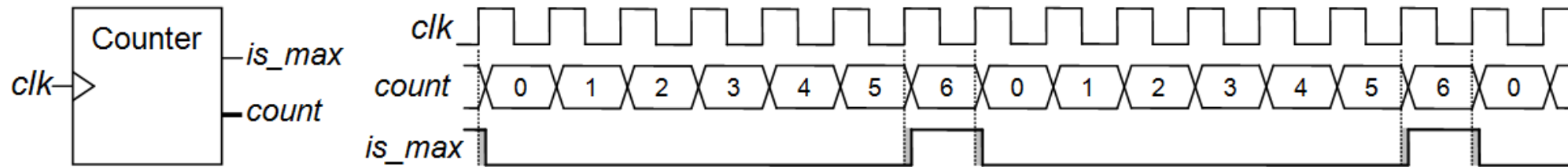
4. The *if* statement

```
[label:] if condition then  
    sequential_statements;  
[elsif  
    sequential_statements;  
[else  
    sequential_statements;  
end if [label];
```

- *if* is the sequential counterpart of the concurrent *when* statement
- Like *when*, it has a priority-encoding nature, so use it only when there are few conditions to test
- To enter truth tables in general (in sequential code), use *case* instead
- **IMP.:** Like *when*, incomplete in-out coverage is accepted by the compiler, but it will infer latches (when the output is unregistered)

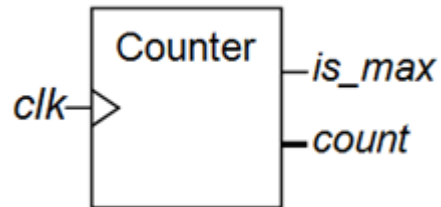
4. The *if* statement

Example:



4. The *if* statement

Example:



```

-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

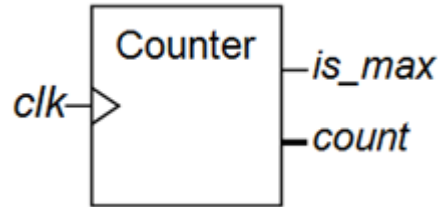
entity counter_with_is_max_flag is
  generic (
    MAX: natural := 6;
    BITS: natural := 3); --not independent (sec. 6.7)
  port (
    clk: in std_logic;
    count: out std_logic_vector(BITS-1 downto 0);
    is_max: out std_logic);
end entity;

architecture rtl of counter_with_is_max_flag is
  signal i: natural range 0 to MAX;
begin

```

4. The *if* statement

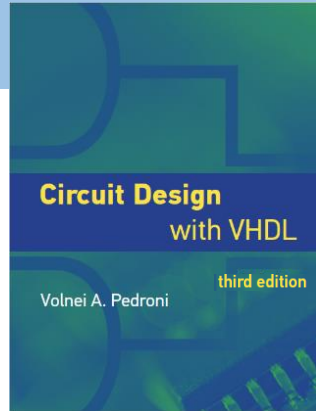
Example:



```

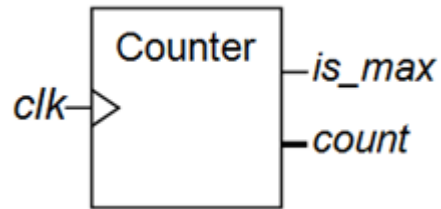
P1: process (clk)
begin
    if rising_edge(clk) then
        if i=MAX-1 then
            is_max <= '1';
            i <= i + 1;
        elsif i=MAX then
            is_max <= '0';
            i <= 0;
        else
            i <= i + 1;
        end if;
    end if;
end process;
count <= std_logic_vector(to_unsigned(i, BITS));

end architecture;
  
```



4. The *if* statement

Example:



```

P1: process (clk)
begin
    if rising_edge(clk) then
        if i=MAX-1 then
            is_max <= '1';
            i <= i + 1;
        elsif i=MAX then
            is_max <= '0';
            i <= 0;
        else
            i <= i + 1;
        end if;
    end if;
end process;
count <= std_logic_vector(to_
end architecture;
  
```

```

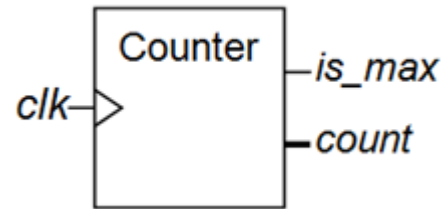
P2: process (clk)
begin
    if rising_edge(clk) then

        --Counter:
        if i /= MAX then
            i <= i + 1;
        else
            i <= 0;
        end if;
        count <= ... (same)

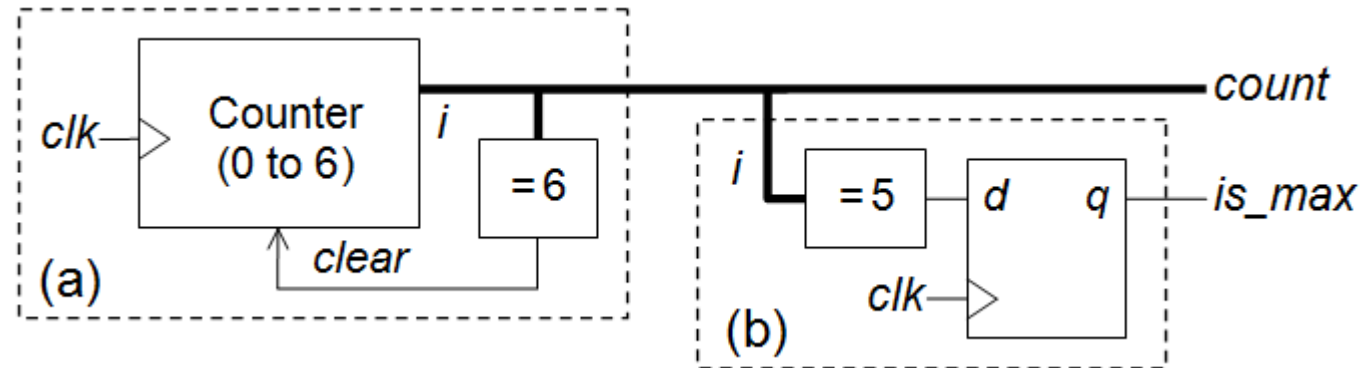
        --Flag generator:
        if i=MAX-1 then
            is_max <= '1';
        else
            is_max <= '0';
        end if;
    end if;
end process;
  
```

4. The *if* statement

Example:

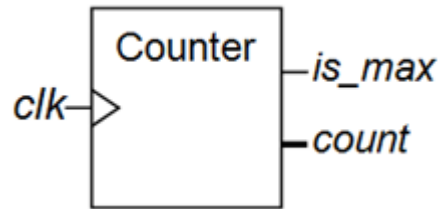


Result:

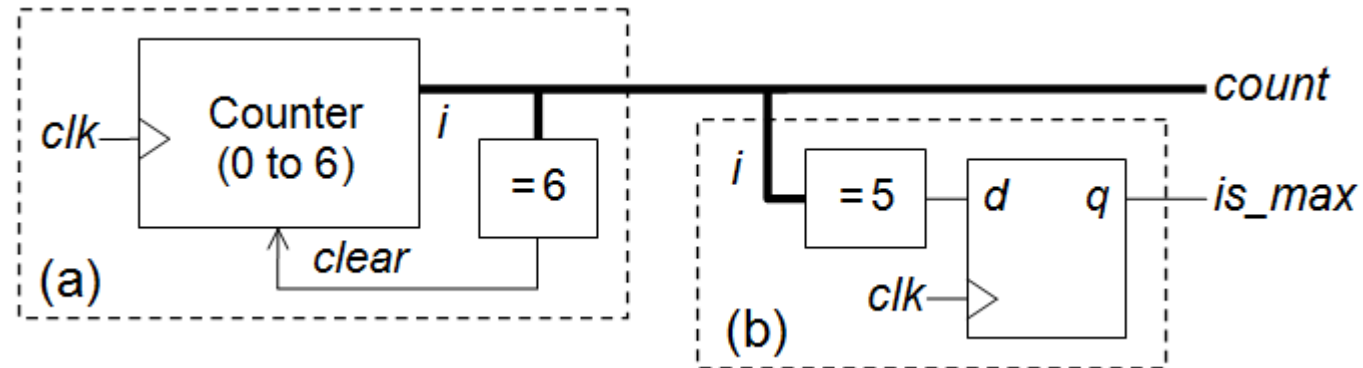


4. The *if* statement

Example:



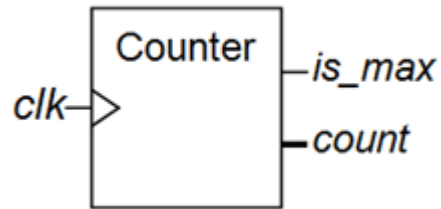
Result:



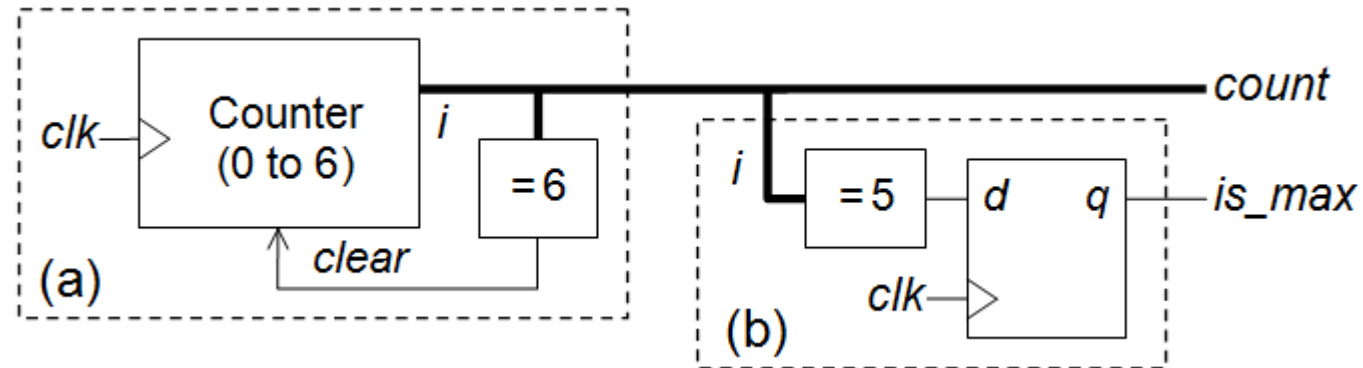
Could we have done better?

4. The *if* statement

Example:



Result:



Could we have done better?

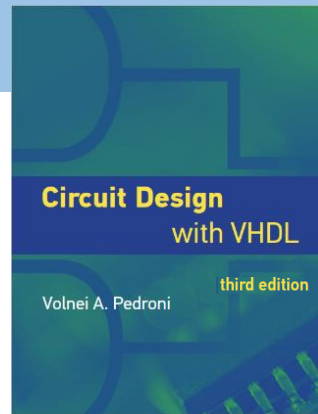
- In the example above, **some** optimization is possible (exercise 13.8)
- But in general, **left optimizations** (if any) are of **little impact**

Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
- ➡ 5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

5. The *case* statement

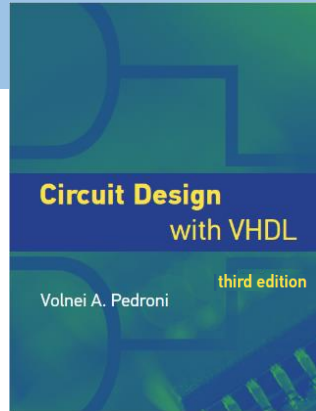


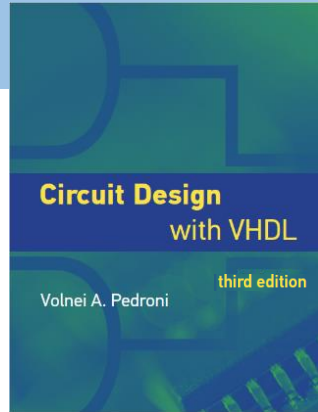
5. The *case* statement

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when others => assignments;  
end case;
```

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when choice => assignments;  
end case;
```

- *case* is the sequential counterpart of the concurrent *select* statement



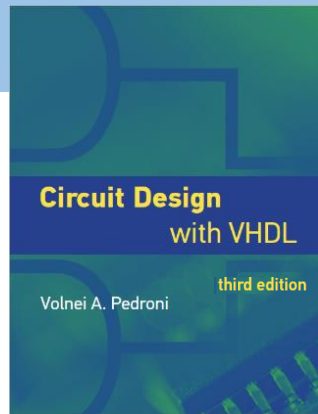


5. The *case* statement

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when others => assignments;  
end case;
```

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when choice => assignments;  
end case;
```

- *case* is the sequential counterpart of the concurrent *select* statement
- The pros and cons of the two constructions above are similar to those seen for *select*



5. The *case* statement

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when others => assignments;  
end case;
```

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when choice => assignments;  
end case;
```

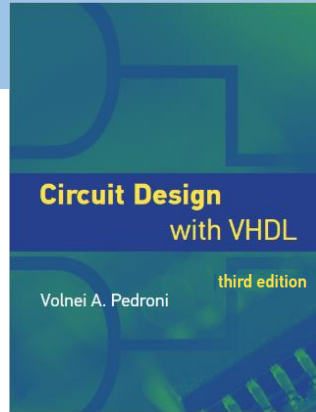
- *case* is the sequential counterpart of the concurrent *select* statement
- The pros and cons of the two constructions above are similar to those seen for *select*
- Like *select*, it is the proper option for entering *truth tables* in general (in sequential code)

5. The *case* statement

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when others => assignments;  
end case;
```

```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when choice => assignments;  
end case;
```

- *case* is the sequential counterpart of the concurrent *select* statement
- The pros and cons of the two constructions above are similar to those seen for *select*
- Like *select*, it is the proper option for entering *truth tables* in general (in sequential code)
- Like *select*, incomplete in-out coverage is **not** accepted by the compiler, guaranteeing a **latch-free** result



5. The *case* statement

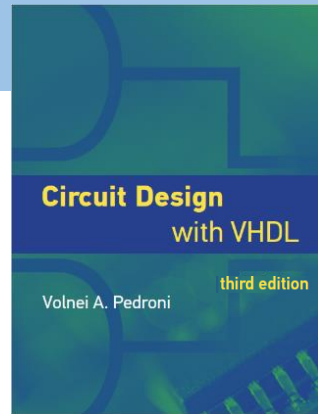
```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when others => assignments;  
end case;
```

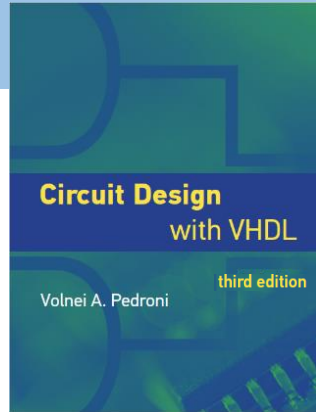
```
[label] case expression is  
    when choice => assignments;  
    when choice => assignments;  
    when choice => assignments;  
end case;
```

- *case* is the sequential counterpart of the concurrent *select* statement
- The pros and cons of the two constructions above are similar to those seen for *select*
- Like *select*, it is the proper option for entering *truth tables* in general (in sequential code)
- Like *select*, incomplete in-out coverage is **not** accepted by the compiler, guaranteeing a **latch-free** result
- Like *select*, the choice expressions can employ *to*, *downto*, *|* (means *or*), and *others*; and *equations* are allowed in the selection argument

5. The *case* statement

Example:





5. The *case* statement

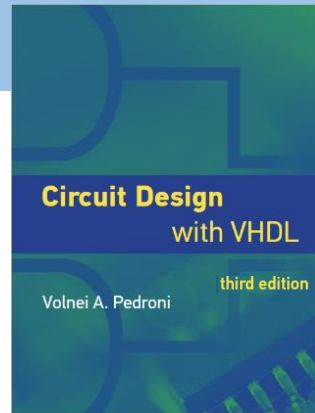
Example:

```
signal control: std_logic_vector(3 downto 0);  
...  
case control is  
  when "0000" => action <= A;  
  when "0001" | "1000" | "1111" => action <= B;  
  when others => action <= C;  
end case;
```

- SUV/SLV are the most common vector types in the industry
- Therefore, the syntax ending in “*when others*” is usually the only viable option

5. The *case* statement

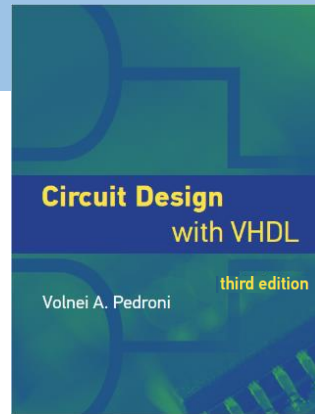
The *unaffected* keyword

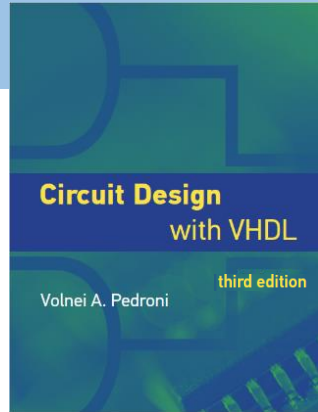


5. The *case* statement

The *unaffected* keyword

- Useful to complete the in-out coverage of *case*
- Use only in **registered** code (see why in **sec. 12.5**)





5. The *case* statement

The *unaffected* keyword

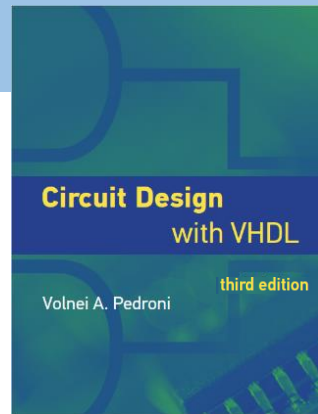
- Useful to complete the in-out coverage of *case*
- Use only in *registered* code (see why in *sec. 12.5*)

Example:

```
process (clk)
begin
    if rising_edge(clk) then
        case sel is
            when 0 => y <= a;
            when 1 => y <= b;
            when others => y <= unaffected;
        end case;
    end if;
end process;
```


5. The *case* statement

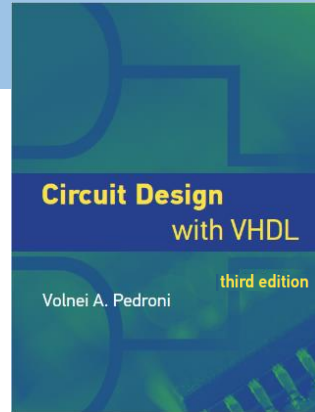
The matching *case?* version

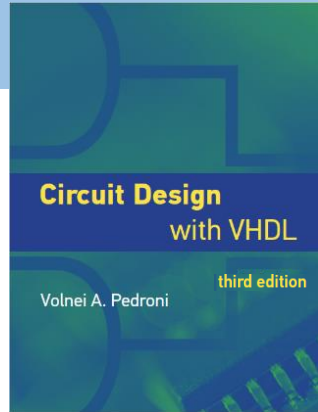


5. The *case* statement

The matching *case?* version

- It is the sequential counterpart of *select?*
- It can be used for “don’t care” values at the input





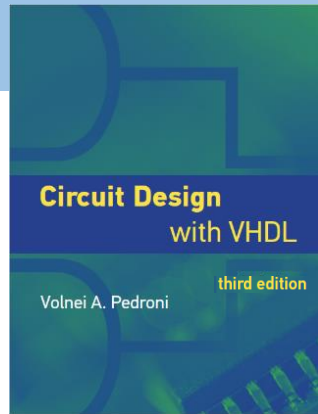
5. The *case* statement

The matching *case?* version

- It is the sequential counterpart of *select?*
- It can be used for “don’t care” values at the input

Example:

```
process (all)
begin
    case? x is
        when "1---" => y <= "1000";
        when "01--" => y <= "0100";
        when "001-" => y <= "0010";
        when "0001" => y <= "0001";
        when others => y <= "0000";
    end case?;
end process;
```

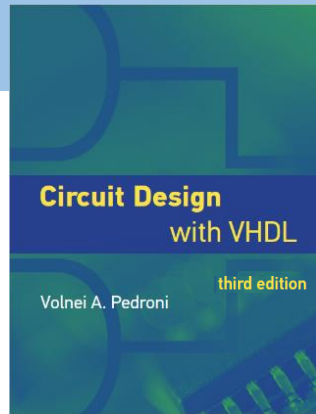


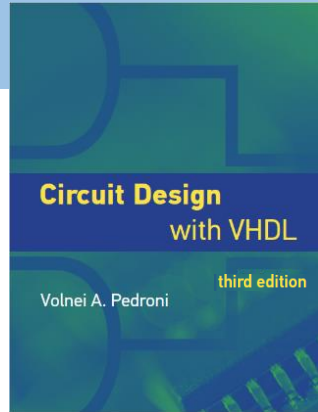
Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
- ➡ 6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

6. The *wait* statement





6. The *wait* statement

```
[label:] wait;
```

(for simulation)

```
[label:] wait until condition;
```

```
[label:] wait on sensitivity_list;
```

```
[label:] wait for time_expression;
```

(for simulation)

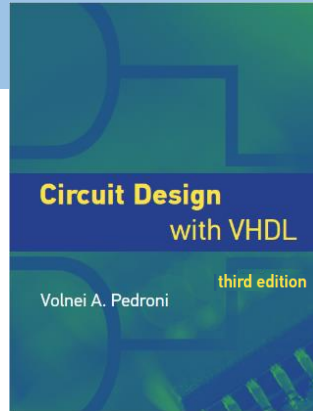
- Four versions, but only 2nd and 3rd are OK for synthesis
- Anyway, not of big help, so *wait* can be **ignored** for synthesis
- But important for simulation (chapter 18)

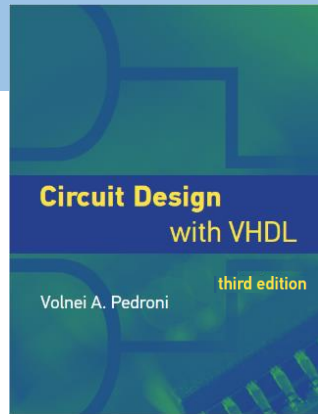
6. The *wait* statement

Example: --DFF built with “wait until”:

```
process
begin
    wait until not clk;
    if clr then
        q <= '0';
    else
        q <= d;
    end if;
end process;
```

(Notice the absence of sensitivity list when *wait* is used)



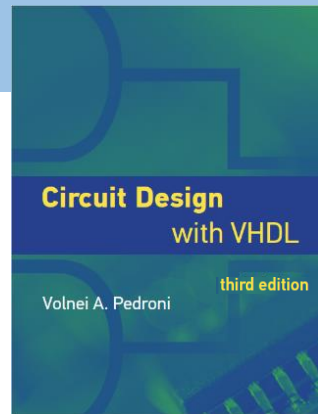


Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
- ➡ 7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

7. The *loop* statement



7. The *loop* statement

unconditional loop:

```
[label:] loop  
    sequential_statements;  
end loop [label];
```

while-loop:

```
[label:] while condition loop  
    sequential_statements;  
end loop [label];
```

for-loop:

```
[label:] for identifier in discrete_range loop  
    sequential_statements;  
end loop [label];
```

loop with exit:

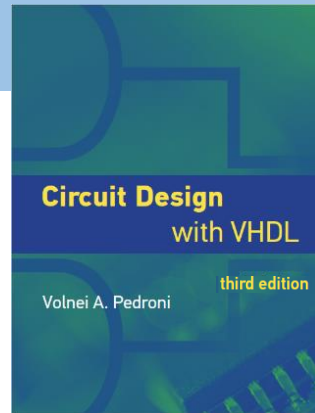
```
[loop_label:] [for identifier in discrete_range] loop  
    ...  
    [exit_label:] exit [loop_label] [when condition];  
    ...  
end loop [loop_label];
```

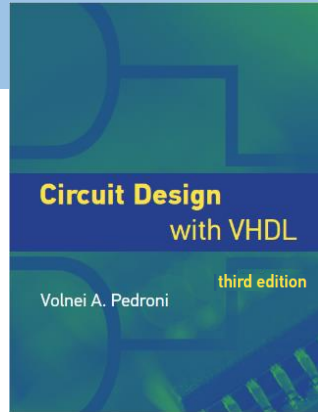
loop with next:

```
[loop_label:] [for identifier in discrete_range] loop  
    ...  
    [next_label:] next [loop_label] [when condition];  
    ...  
end loop [loop_label];
```

7. The *loop* statement

- *loop* is the sequential counterpart of the concurrent *generate* statement
- There are five options, *for-loop* being by far the most common
- Note that the versions with *exit* and *next* are indeed special cases of *for-loop*



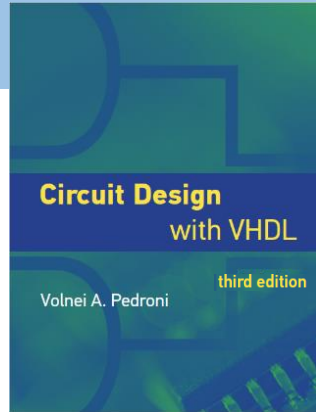


7. The *loop* statement

- *loop* is the sequential counterpart of the concurrent *generate* statement
- There are five options, *for-loop* being by far the most common
- Note that the versions with *exit* and *next* are indeed special cases of *for-loop*

Example with *for-loop*:

```
for i in 0 to 31 loop
  x(i) <= a(i) nand b(31-i);
end loop;
```



7. The *loop* statement

- *loop* is the sequential counterpart of the concurrent *generate* statement
- There are five options, *for-loop* being by far the most common
- Note that the versions with *exit* and *next* are indeed special cases of *for-loop*

Example with *for-loop*:

```
for i in 0 to 31 loop
    x(i) <= a(i) nand b(31-i);
end loop;
```

Example with *for-loop* + *exit*:

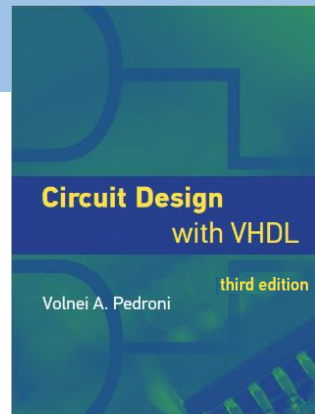
```
count := 0;
for i in data'range loop
    case data(i) is
        when '1' => count := count + 1;
        when others => exit;
    end case;
end loop;
leading_ones <= count;
```

Chapters 12-13

Sequential Code

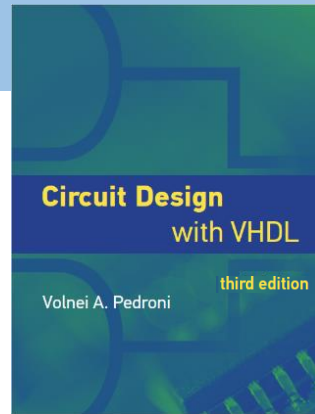
1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
- ➡ 8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

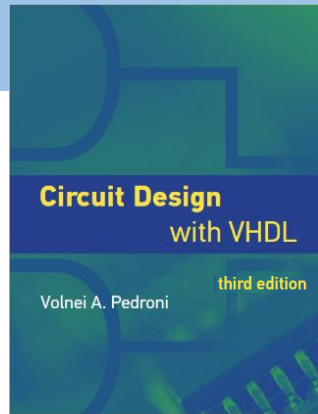
8. The sequential *when* and *select* statements



8. The sequential *when* and *select* statements

- Introduced in VHDL-2008
- Same syntax seen for the concurrent versions (chapter 10)





8. The sequential *when* and *select* statements

- Introduced in VHDL-2008
- Same syntax seen for the concurrent versions (chapter 10)

Example: --DFF with synchronous clear:

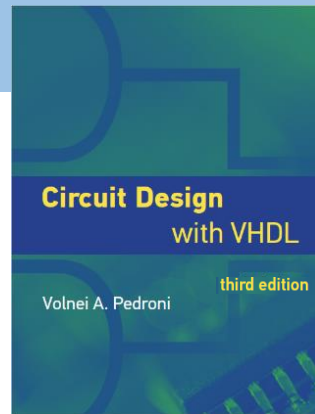
```
process (clk)
begin
    if rising_edge(clk) then
        q <= '0' when clear else d;
    end if;
end process;
```

Chapters 12-13

Sequential Code

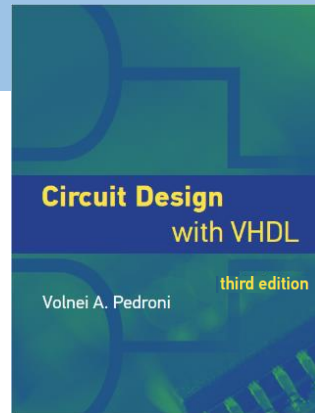
1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
- ➡ 9. *Signal versus Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

9. *Signal* versus *Variable*



9. *Signal* versus *Variable*

- Variables are a major feature of VHDL
- However, to write code efficiently, it's indispensable to know well their **properties** and their **differences** with respect to **signals**
- To help in that task, we propose the **6 rules** of table 12.1



9. *Signal* versus *Variable*

Table 12.1
Signal versus variable

| Rule/Feature | | Signal | Variable (*) |
|--------------|------------------------|---|---|
| 1 | Places of declaration | In entity, architecture, package, block, and generate (forbidden in sequential units) | Only in sequential units (i.e., process and subprograms) |
| 2 | Scope | Global (can be read and modified anywhere in the architecture statements region) | Local (can be read and modified only inside its own sequential unit) |
| 3 | Assignment symbol | <code><=</code> (example: <code>sig <= 255;</code>) | <code>:=</code> (example: <code>var := 255;</code>) |
| 4 | Multiple assignments | Allowed only inside sequential units, but be aware of combinational loops | Always fine (because they can only be used inside sequential units anyway) |
| 5 | Update | New value available only at the end of the process cycle (after suspension) | Updated immediately (new value ready to be used in the next line of code) |
| 6 | Inference of registers | Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal | Flip-flops are inferred when an assignment to a variable occurs at the transition of a signal and this variable affects some other signal |

(*) Shared variables not considered (should not be used for synthesis).

9. *Signal* versus *Variable*

Rule 1: Places of declaration

Signal:

- Declared (created) in the declarative part of entity, architecture, package, block, or generate
- Remember also that all circuit ports are signals

Variable:

- Declared (created) only in sequential units (i.e., process and subprograms)
- The only exception is for *shared variables*, but these should be avoided for synthesis

9. *Signal* versus *Variable*

Rule 1: Places of declaration

Signal:

- Declared (created) in the declarative part of entity, architecture, package, block, or generate
- Remember also that all circuit ports are signals

Variable:

- Declared (created) only in sequential units (i.e., process and subprograms)
- The only exception is for *shared variables*, but these should be avoided for synthesis

Rule 2: Scope (places of use)

Signal:

- Global (can be read and modified anywhere, in concurrent and sequential regions)

Variable:

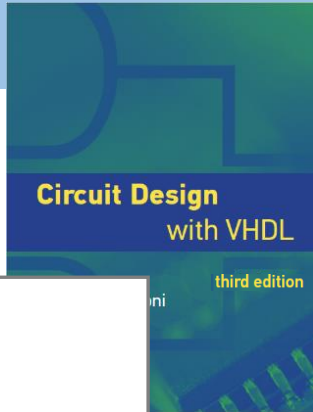
- Always local (can be read and modified only inside the sequential unit where it was created)
- To leave that unit, its value must be passed to a signal or must affect a signal
- The only exception is for shared variables (global), but these should be avoided for synthesis

9. *Signal* versus *Variable*

Rule 3: Assignment symbol

Signal: `<=` Example: `sig <= a + b;`

Variable: `:=` Example: `var := a + b;`



9. *Signal* versus *Variable*

Rule 3: Assignment symbol

Signal: `<=` Example: `sig <= a + b;`

Variable: `:=` Example: `var := a + b;`

Rule 4: Multiple assignments

Signal:

- In concurrent code, just one assignment is allowed
- In sequential code, multiple assignments are fine (last value prevails), but be aware of combinational loops (sec. 12.12)

Variable:

- Multiple assignments are always fine (because they can only occur in sequential code)
- Last value prevails

9. *Signal* versus *Variable*

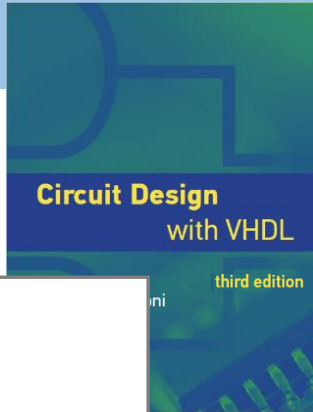
Rule 5: Update

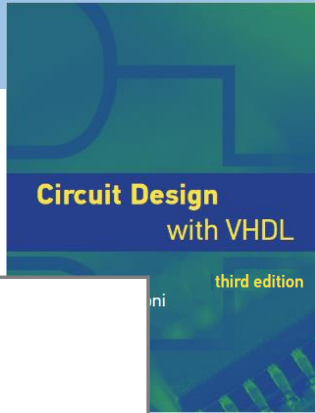
Signal:

- Updated only at the end of the process (when it suspends)

Variable:

- Updated immediately (new value ready to be used in the next line of code)





9. *Signal* versus *Variable*

Rule 5: Update

Signal:

- Updated only at the end of the process (when it suspends)

Variable:

- Updated immediately (new value ready to be used in the next line of code)

Rule 6: Inference of registers

Signal:

Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal

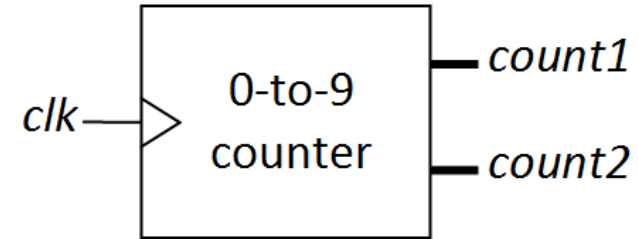
Variable:

- Flip-flops are inferred when an assignment to a variable occurs at the transition of a signal and this variable affects (directly or indirectly) some other signal

9. *Signal* versus *Variable*

Example:

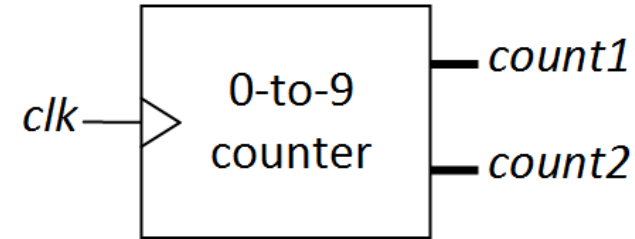
We want to build a 0-to-9 free-running counter



9. *Signal* versus *Variable*

Example:

We want to build a 0-to-9 free-running counter



```

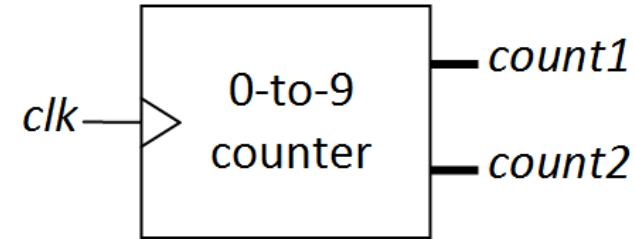
--count1 is a port (signal):
counter1: process (clk)
begin
    if rising_edge(clk) then
        count1 <= count1 + 1;
        if count1=10 then
            count1 <= 0;
        end if;
    end if;
end process counter1;
  
```

Counts from ? to ?

9. *Signal* versus *Variable*

Example:

We want to build a 0-to-9 free-running counter



```

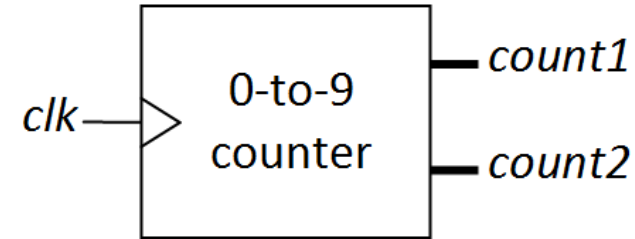
--count1 is a port (signal):
counter1: process (clk)
begin
    if rising_edge(clk) then
        count1 <= count1 + 1;
        if count1=10 then
            count1 <= 0;
        end if;
    end if;
end process counter1;
  
```

Counts from 0 to 10

9. *Signal* versus *Variable*

Example:

We want to build a 0-to-9 free-running counter



--count1 is a port (signal):

```

counter1: process (clk)
begin
  if rising_edge(clk) then
    count1 <= count1 + 1;
    if count1=10 then
      count1 <= 0;
    end if;
  end if;
end process counter1;
  
```

Counts from 0 to 10

--count2 is another port:

```

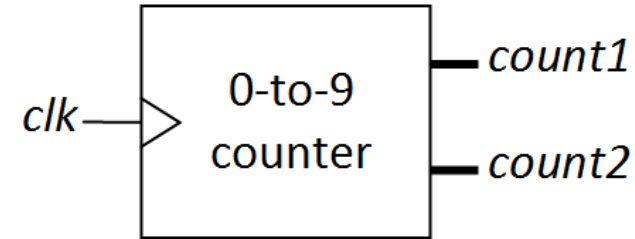
counter2: process (clk)
  variable var: natural range 0 to 10;
begin
  if rising_edge(clk) then
    var := var + 1;
    if var=10 then
      var := 0;
    end if;
  end if;
  count2 <= var;
end process counter2;
  
```

Counts from ? to ?

9. *Signal* versus *Variable*

Example:

We want to build a 0-to-9 free-running counter



--count1 is a port (signal):

```

counter1: process (clk)
begin
  if rising_edge(clk) then
    count1 <= count1 + 1;
    if count1=10 then
      count1 <= 0;
    end if;
  end if;
end process counter1;
  
```

Counts from 0 to 10

--count2 is another port:

```

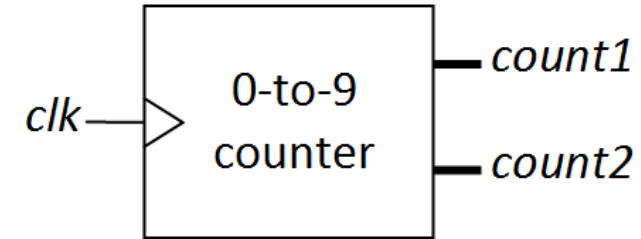
counter2: process (clk)
  variable var: natural range 0 to 10;
begin
  if rising_edge(clk) then
    var := var + 1;
    if var=10 then
      var := 0;
    end if;
  end if;
  count2 <= var;
end process counter2;
  
```

Counts from 0 to 9

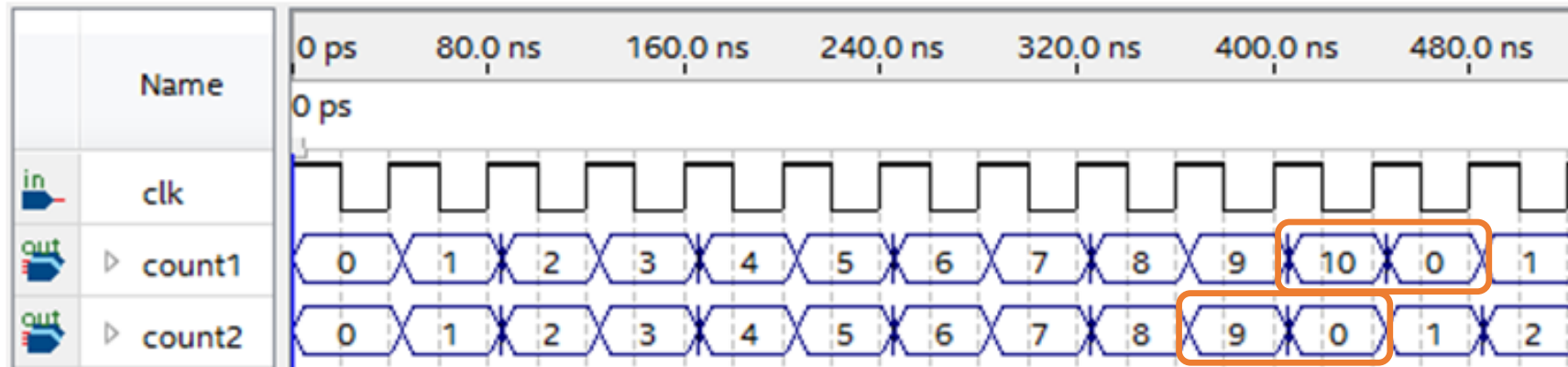
9. *Signal* versus *Variable*

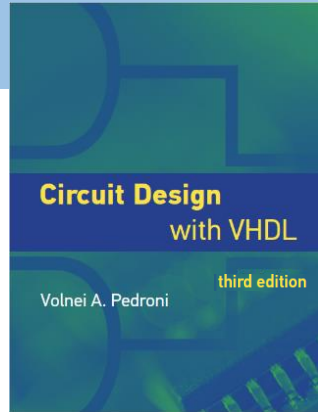
Example:

We want to build a 0-to-9 free-running counter



Simulation results:





We close this chapter with three special discussions:

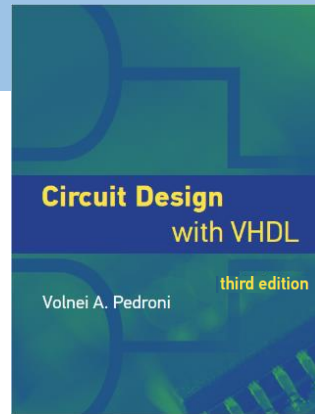
- 1) More on the “update rule” of signals and variables
- 2) More on the “inference of registers rule” of signals and variables
- 3) The danger of “combinational loops”

Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
- ➡ 10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables
12. Combinational loops

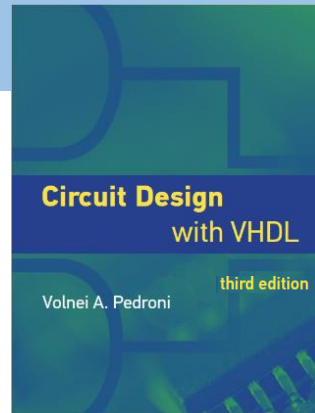
10. More on the **update rule** of signals and variables



10. More on the **update rule** of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately



10. More on the update rule of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Example (*count* is a signal):

| | <i>Modify-then-read</i> approach: | | <i>Read-then-modify</i> approach: |
|---|-----------------------------------|----|-----------------------------------|
| 1 | P1: process (clk) | 1 | P2: process (clk) |
| 2 | begin | 2 | begin |
| 3 | if rising_edge(clk) then | 3 | if rising_edge(clk) then |
| 4 | count <= count + 1; | 4 | if count /= 9 then |
| 5 | if count=9 then | 5 | count <= count + 1; |
| 6 | count <= 0; | 6 | else |
| 7 | end if; | 7 | count <= 0; |
| 8 | end if; | 8 | end if; |
| 9 | end process; | 9 | end if; |
| | Counts up to ? | 10 | end process; Counts up to ? |

10. More on the update rule of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Example (*count* is a signal):

| | <i>Modify-then-read</i> approach: | | <i>Read-then-modify</i> approach: |
|---|-----------------------------------|----|-----------------------------------|
| 1 | P1: process (clk) | 1 | P2: process (clk) |
| 2 | begin | 2 | begin |
| 3 | if rising_edge(clk) then | 3 | if rising_edge(clk) then |
| 4 | count <= count + 1; | 4 | if count /= 9 then |
| 5 | if count=9 then | 5 | count <= count + 1; |
| 6 | count <= 0; | 6 | else |
| 7 | end if; | 7 | count <= 0; |
| 8 | end if; | 8 | end if; |
| 9 | end process; | 9 | end if; |
| | Counts up to 9 | 10 | end process; Counts up to 9 |

10. More on the **update rule** of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Example (*i* is a variable):

| | <i>Modify-then-read</i> approach: | | <i>Read-then-modify</i> approach: |
|----|------------------------------------|----|-----------------------------------|
| 1 | P3: process (clk) | 1 | P4: process (clk) |
| 2 | variable i: natural range 0 to 10; | 2 | variable i: natural range 0 to 9; |
| 3 | begin | 3 | begin |
| 4 | if rising_edge(clk) then | 4 | if rising_edge(clk) then |
| 5 | i := i + 1; | 5 | if count /= 9 then |
| 6 | if i=10 then | 6 | i := i + 1; |
| 7 | i := 0; | 7 | else |
| 8 | end if; | 8 | i := 0; |
| 9 | end if; | 9 | end if; |
| 10 | count <= i; | 10 | end if; |
| 11 | end process; Counts up to ? | 11 | count <= i; Counts up to ? |
| | | 12 | end process; |

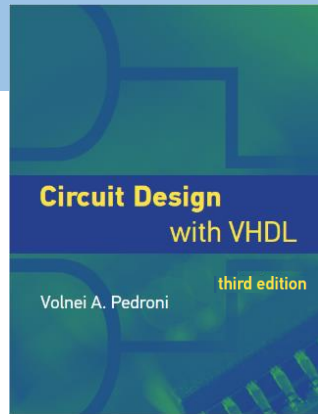
10. More on the **update rule** of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Example (*i* is a variable):

| | <i>Modify-then-read</i> approach: | | <i>Read-then-modify</i> approach: |
|----|------------------------------------|----|-----------------------------------|
| 1 | P3: process (clk) | 1 | P4: process (clk) |
| 2 | variable i: natural range 0 to 10; | 2 | variable i: natural range 0 to 9; |
| 3 | begin | 3 | begin |
| 4 | if rising_edge(clk) then | 4 | if rising_edge(clk) then |
| 5 | i := i + 1; | 5 | if count /= 9 then |
| 6 | if i=10 then | 6 | i := i + 1; |
| 7 | i := 0; | 7 | else |
| 8 | end if; | 8 | i := 0; |
| 9 | end if; | 9 | end if; |
| 10 | count <= i; | 10 | end if; |
| 11 | end process; Counts up to 9 | 11 | count <= i; Counts up to 9 |
| | | 12 | end process; |



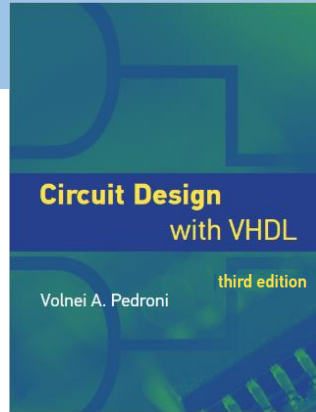
10. More on the **update rule** of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Conclusions (but see next recommendation):

- When using a **signal**: Prefer the *read-then-modify* approach (prevents comparison to the signal's *previous* value)
- When using a **variable**: Either approach is fine



10. More on the **update rule** of signals and variables

Rule 5: Update

- *Signals* are updated at the end of the process
- *Variables* are updated immediately

Conclusions (but see next recommendation):

- When using a *signal*: Prefer the *read-then-modify* approach (prevents comparison to the signal's *previous* value)
- When using a *variable*: Either approach is fine

Recommendation:

- Use *variables* (instead of *signals*) to implement *counters* and *loops*

Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
- ➡ 11. More on the inference of registers rule of signals and variables
12. Combinational loops

11. More on the inference of registers rule of signals and variables

11. More on the inference of registers rule of signals and variables

Rule 6: Inference of registers

- A **signal** (or **variable**, if it affects a signal) gets registered if an assignment is made to it at the transition of another signal

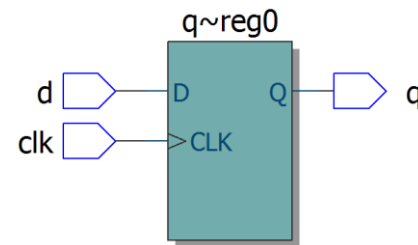
11. More on the inference of registers rule of signals and variables

Rule 6: Inference of registers

- A **signal** (or **variable**, if it affects a signal) gets registered if an assignment is made to it at the transition of another signal

Example: A value is assigned to a **signal** (*q*) at the transition of another **signal** (*clk*)

```
if rising_edge(clk) then  
    q <= d;  
end if;
```



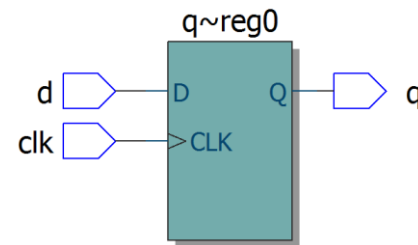
11. More on the inference of registers rule of signals and variables

Rule 6: Inference of registers

- A **signal** (or **variable**, if it affects a signal) gets registered if an assignment is made to it at the transition of another signal

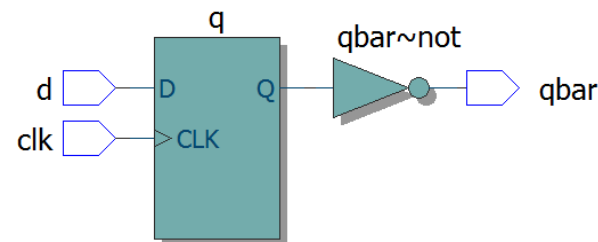
Example: A value is assigned to a **signal** (q) at the transition of another **signal** (clk)

```
if rising_edge(clk) then
  q <= d;
end if;
```



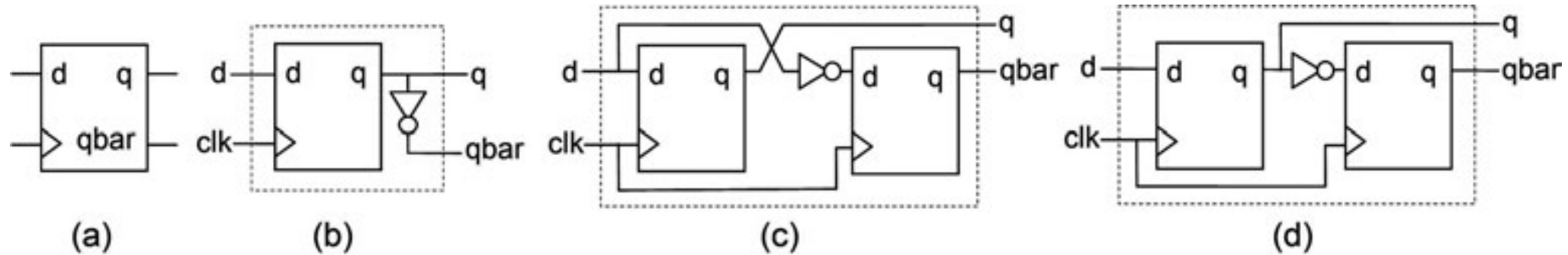
Example: A value is assigned to a **variable** (q) at the transition of a **signal** (clk), and this variable does affect a **signal** ($qbar$).

```
if rising_edge(clk) then
  q := d;
end if;
qbar <= not q;
```



11. More on the inference of registers rule of signals and variables

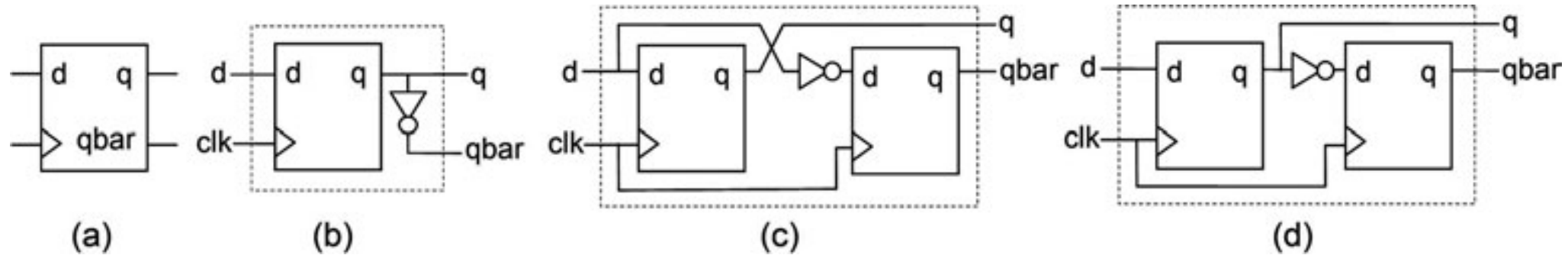
Another example:



- We want the DFF in (a), but in FPGAs we have to implement it as in (b)
- Which circuit does each code shown next produce?

11. More on the inference of registers rule of signals and variables

Another example:

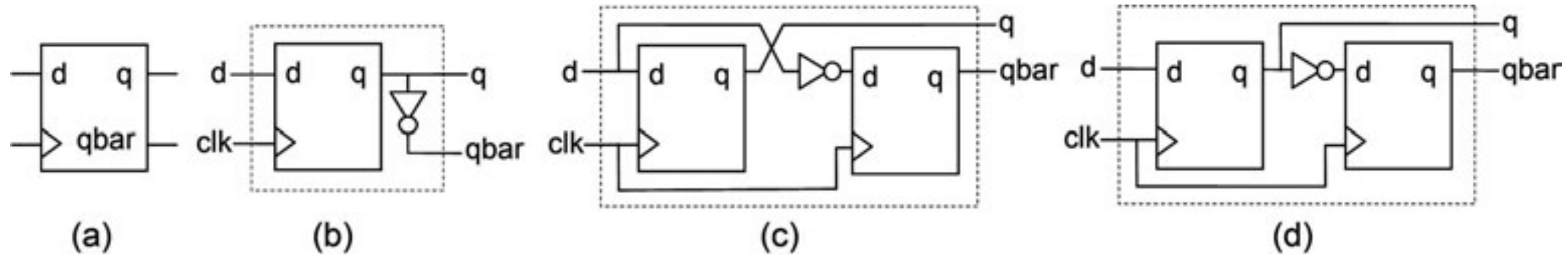


```
entity dff_with_qbar is
  port (
    d, clk: in std_logic;
    q, qbar: out std_logic);
end entity;
```

Notice that *q* and *qbar* are **ports** (therefore, **signals**)

11. More on the inference of registers rule of signals and variables

Another example:



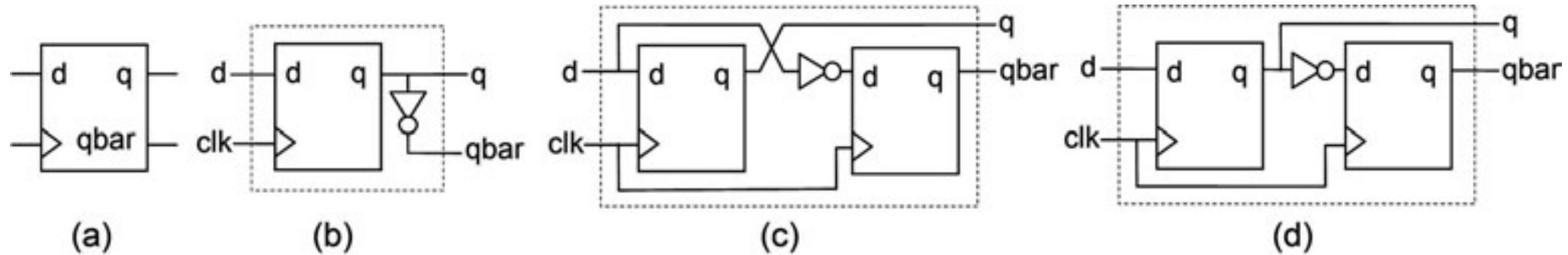
```
P1: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not d;
  end if;
end process;
```

```
P2: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not q;
  end if;
end process;
```

```
P3: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
qbar <= not q;
```

11. More on the inference of registers rule of signals and variables

Another example:



```
P1: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not d;
  end if;
end process;
```

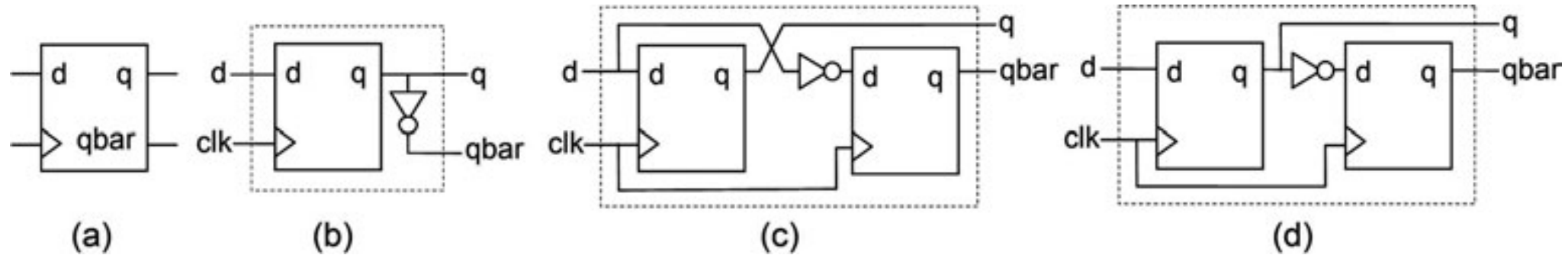
(c)

```
P2: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not q;
  end if;
end process;
```

```
P3: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
qbar <= not q;
```

11. More on the inference of registers rule of signals and variables

Another example:



```
P1: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not d;
  end if;
end process;
```

(c)

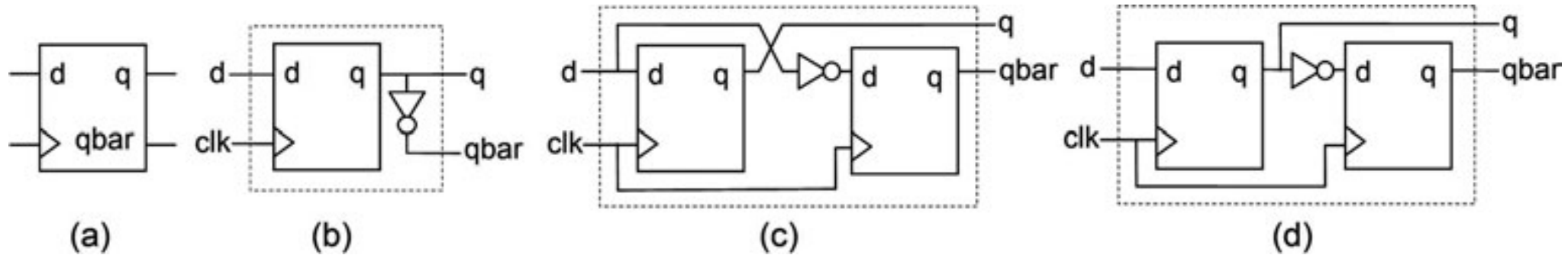
```
P2: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not q;
  end if;
end process;
```

(d)

```
P3: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
qbar <= not q;
```

11. More on the inference of registers rule of signals and variables

Another example:



```
P1: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not d;
  end if;
end process;
```

(c)

```
P2: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not q;
  end if;
end process;
```

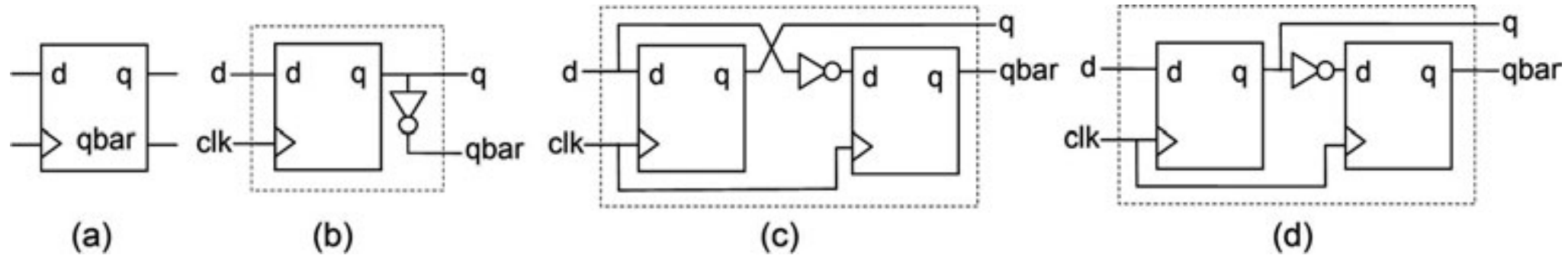
(d)

```
P3: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
qbar <= not q;
```

(b)

11. More on the inference of registers rule of signals and variables

Another example:



```
P1: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not d;
  end if;
end process;
```

(c)

```
P2: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
    qbar <= not q;
  end if;
end process;
```

(d)

```
P3: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
qbar <= not q;
```

(b)

- Is (c) functionally equivalent to (b)?
- What about (d)?

Chapters 12-13

Sequential Code

1. Sequential statements
2. Detecting clock transitions
3. The *process* statement
4. The *if* statement
5. The *case* statement
6. The *wait* statement
7. The *loop* statement
8. The sequential *when* and *select* statements
9. *Signal* versus *Variable*
10. More on the update rule of signals and variables
11. More on the inference of registers rule of signals and variables



12. Combinational loops

12. Combinational loops

12. Combinational loops

- Can occur when a *signal* (*) is used in a *loop* (sequential code) to try to build a combinational circuit

(*) It can occur occasionally also with variables; for example, when they are not initialized in the code.

12. Combinational loops

- Can occur when a *signal* (*) is used in a *loop* (sequential code) to try to build a combinational circuit

Example: Leading-ones counter

```
--With a variable (OK):
process (all)
    variable count: natural range 0 to BITS;
begin
    count := 0;
    for i in data'range loop
        case data(i) is
            when '1' => count := count + 1;
            when others => exit;
        end case;
    end loop;
    leading_ones <= count;
end process;
```

(*) It can occur occasionally also with variables; for example, when they are not initialized in the code.

12. Combinational loops

- Can occur when a *signal* (*) is used in a *loop* (sequential code) to try to build a combinational circuit

Example: Leading-ones counter

```
--With a signal (not OK):  
process (all)  
begin  
    leading_ones <= 0;  
    for i in data'range loop  
        case data(i) is  
            when '1' => leading_ones <= leading_ones + 1;  
            when others => exit;  
        end case;  
    end loop;  
end process;
```

(*) It can occur occasionally also with variables; for example, when they are not initialized in the code.

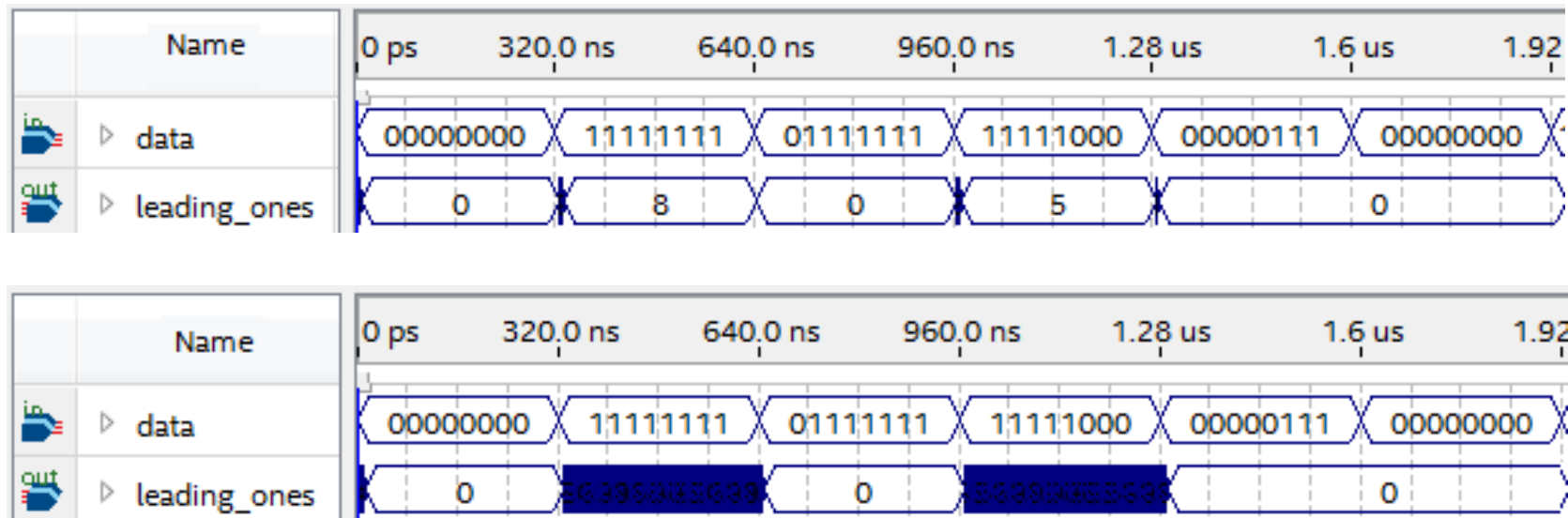
12. Combinational loops

- Can occur when a *signal* (*) is used in a *loop* (sequential code) to try to build a combinational circuit

Example: Leading-ones counter

(*) It can occur occasionally also with variables; for example, when they are not initialized in the code.

Simulation results (see details in section 12.12):



12. Combinational loops

- Can occur when a *signal* (*) is used in a *loop* (sequential code) to try to build a combinational circuit

Conclusion: Use always *variables* in *loop* statements

End of Chapter 12 (and 13)