FYS4220/9220

UNIVERSITY
OF OSLO

Summary of FPGA & VHDL

Lecture #6

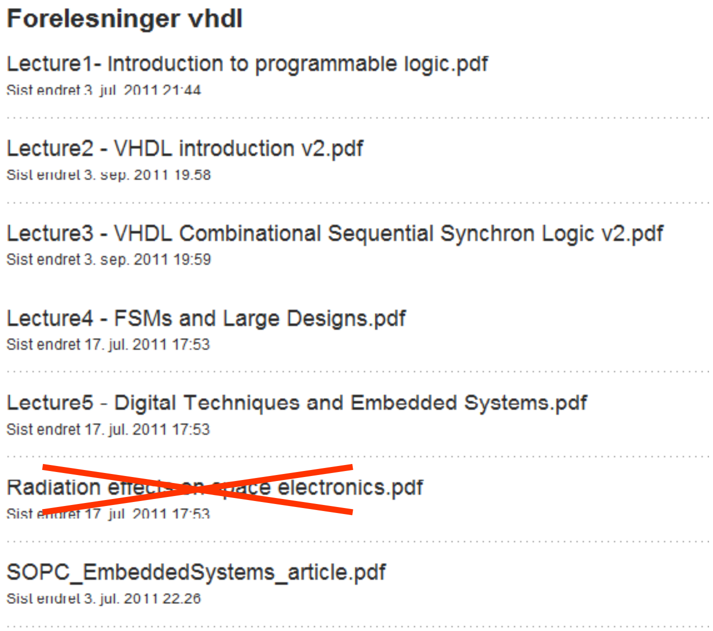
Jan Kenneth Bekkeng, University of Oslo - Department of Physics

16.11.2011

Curriculum (VHDL & FPGA part)

Curriculum (Syllabus) defined by:

- Lectures



Forelesninger vhdl

Lecture1- Introduction to programmable logic.pdf
Sist endret 3. jul. 2011 21:44

Lecture2 - VHDL introduction v2.pdf
Sist endret 3. sep. 2011 19.58

Lecture3 - VHDL Combinational Sequential Synchron Logic v2.pdf
Sist endret 3. sep. 2011 19:59

Lecture4 - FSMs and Large Designs.pdf
Sist endret 17. jul. 2011 17:53

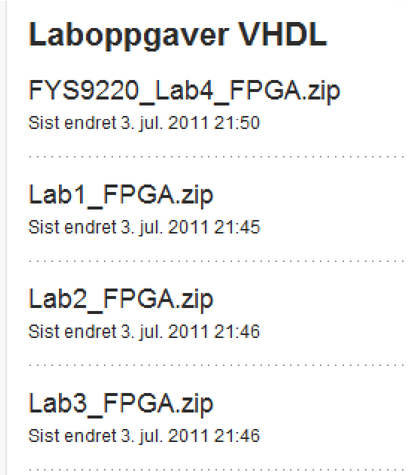
Lecture5 - Digital Techniques and Embedded Systems.pdf
Sist endret 17. jul. 2011 17:53

~~Radiation effects on space electronics.pdf~~
~~Sist endret 17. jul. 2011 17:53~~

SOPC_EmbeddedSystems_article.pdf
Sist endret 3. jul. 2011 22.26

Lecture6: Summary ….

- Laboratory exercises + documentation



Laboppgaver VHDL

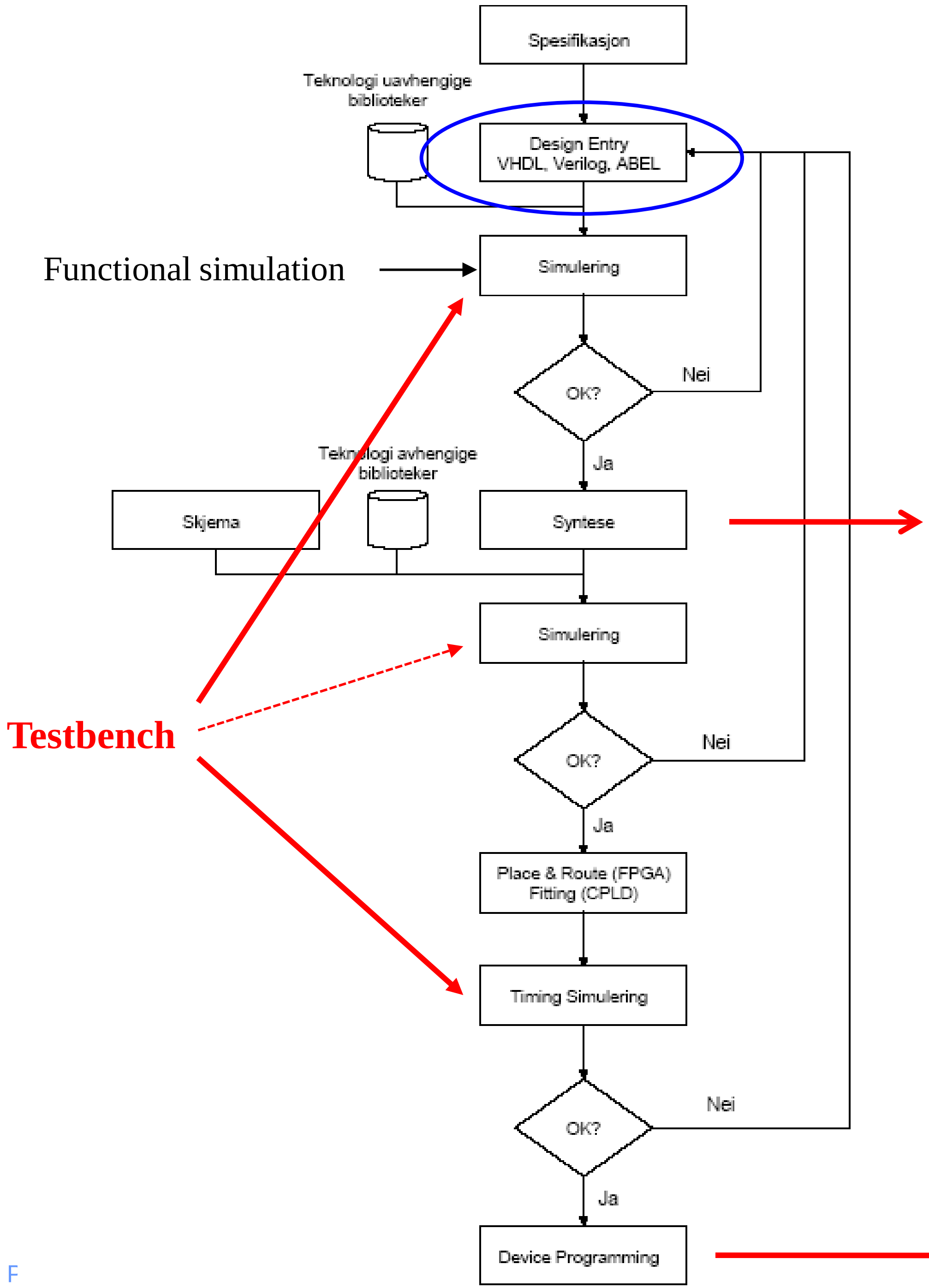
FYS9220_Lab4_FPGA.zip
Sist endret 3. jul. 2011 21:50

Lab1_FPGA.zip
Sist endret 3. jul. 2011 21:45

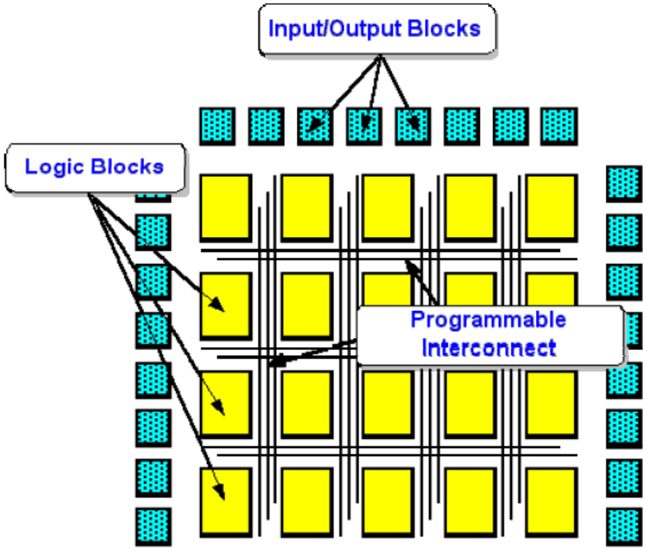
Lab2_FPGA.zip
Sist endret 3. jul. 2011 21:46

Lab3_FPGA.zip
Sist endret 3. jul. 2011 21:46

Design flow

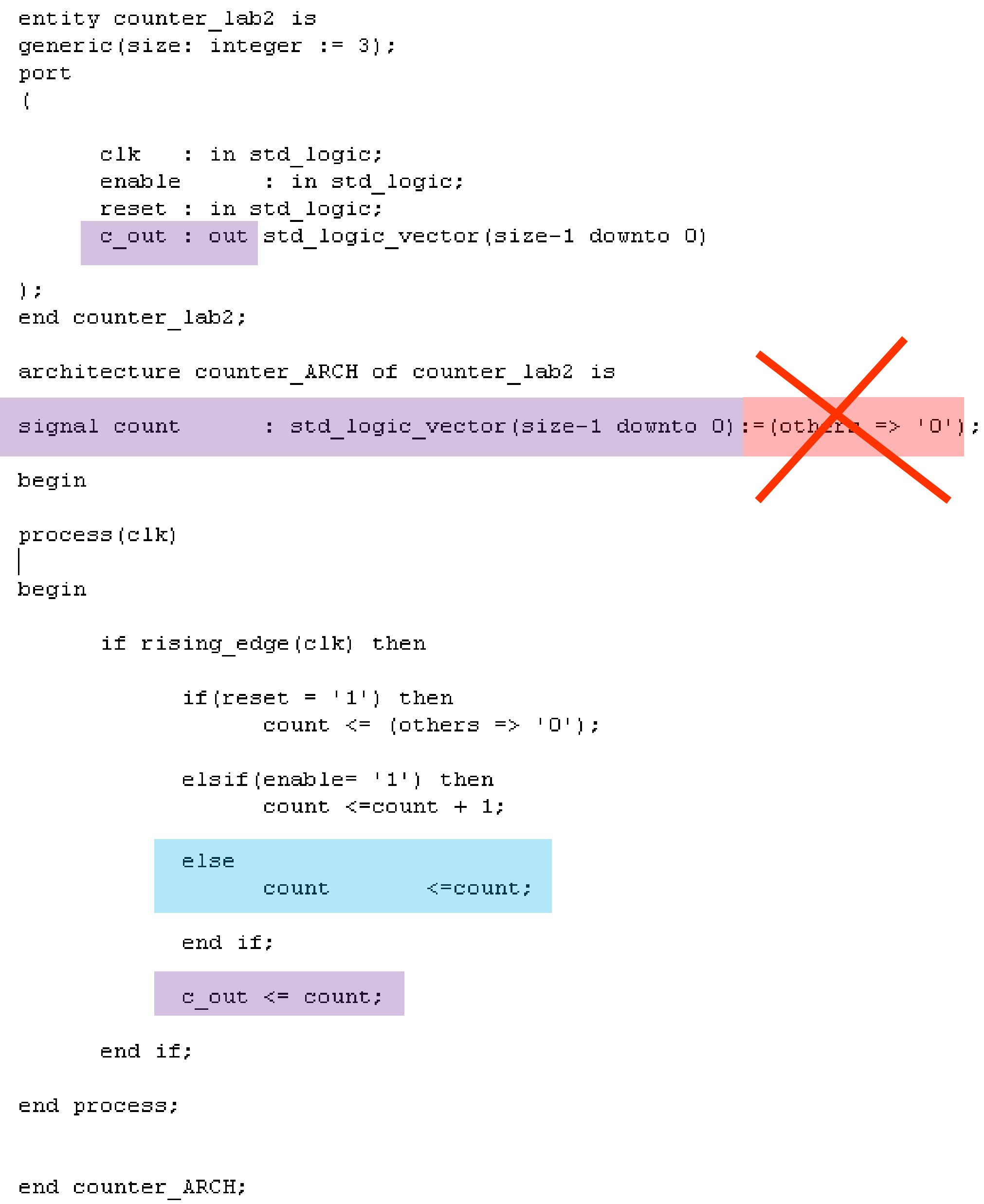


Boolean equations/Netlist



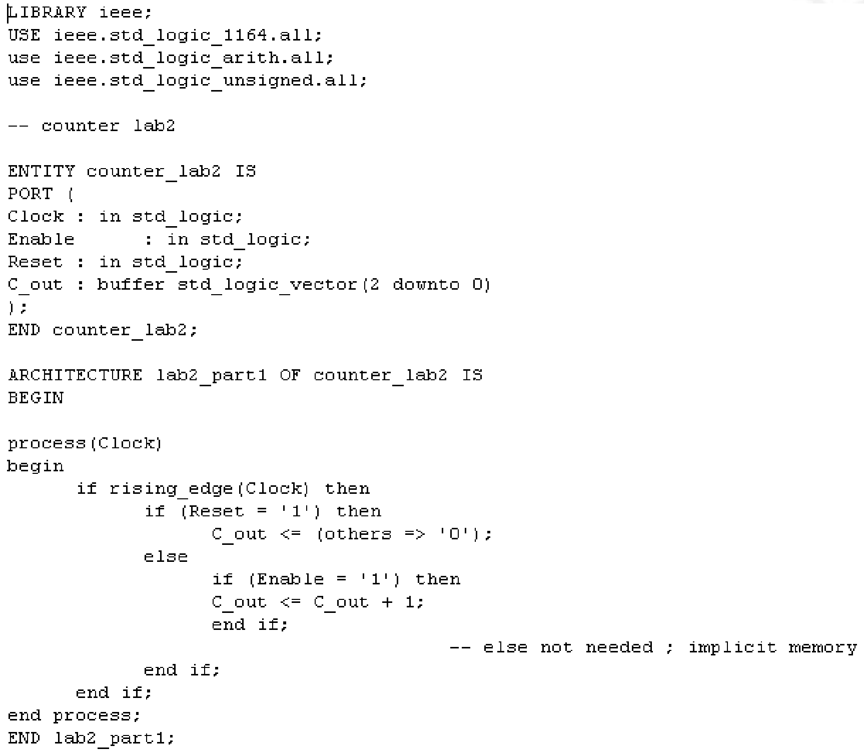
INF3430-Kretsteknologier

”101001110001”



```
entity counter_lab2 is
generic(size: integer := 3);
port
(

      clk   : in std_logic;
      enable       : in std_logic;
      reset : in std_logic;
      c_out : out std_logic_vector(size-1 downto 0)

);
end counter_lab2;

architecture counter_ARCH of counter_lab2 is

signal count        : std_logic_vector(size-1 downto 0):=(others => '0');

begin

process(clk)

begin

      if rising_edge(clk) then

            if(reset = '1') then
                  count <= (others => '0');

            elsif(enable= '1') then
                  count <=count + 1;

            else
                  count        <=count;

            end if;

            c_out <= count;

      end if;

end process;


end counter_ARCH;
```

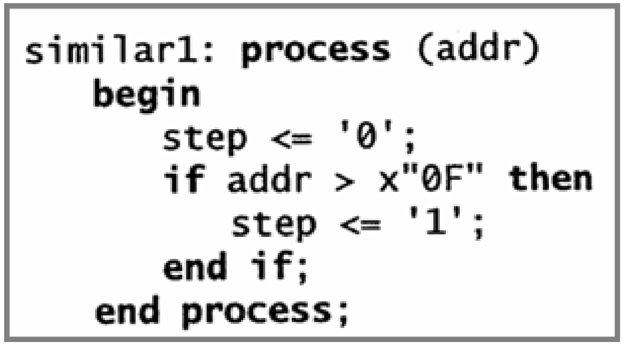
Improved code



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- counter lab2

ENTITY counter_lab2 IS
PORT (
Clock : in std_logic;
Enable      : in std_logic;
Reset : in std_logic;
C_out : buffer std_logic_vector(2 downto 0)
);
END counter_lab2;

ARCHITECTURE lab2_part1 OF counter_lab2 IS
BEGIN

process(Clock)
begin
      if rising_edge(Clock) then
            if (Reset = '1') then
                  C_out <= (others => '0');
            else
                  if (Enable = '1') then
                  C_out <= C_out + 1;
                  end if;
                                          -- else not needed ; implicit memory
            end if;
      end if;
end process;
END lab2_part1;
```

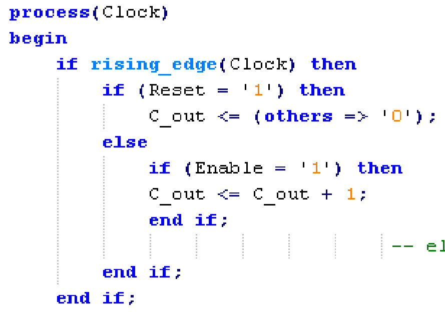
Coding for Synthesis

- Omit the **wait for XX ns** statement
- Omit the **... after XX ns** statement
- Omit initial values
  - Do not assign signals and variables initial values because initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design. For example, do not use initialization statements like the following: **variable SUM:INTEGER:~~=0;~~**
- Make sure that all outputs are defined in all branches of an *if statement*. If not it can creates latches
  - A good way to prevent this is to have default values for all outputs before the *if statements.*



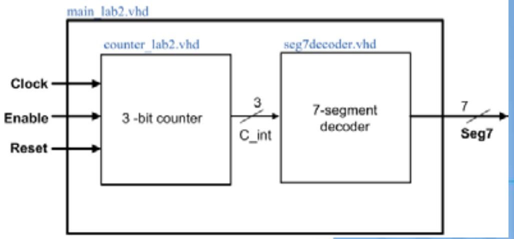
```
similar1: process (addr)
  begin
    step <= '0';
    if addr > x"0F" then
      step <= '1';
    end if;
  end process;
```

Common VHDL coding ”errors”



```
process(Clock)
begin
    if rising_edge(Clock) then
        if (Reset = '1') then
            C_out <= (others => '0');
        else
            if (Enable = '1') then
            C_out <= C_out + 1;
            end if;
                                    -- el
        end if;
    end if;
```

- Missing indent (low readability of the code)
- Wrong sensitivity list
  - Too many/too few signals listed
  - Can create wrong behavior, e.g. in state machines, resulting in needless calls of the process or wrong VHDL description of the implemented circuit
- A mix of 1 process and 2 process FSM
- Declaration of unnecessary internal signals



- ASM chart different (e.g. simplified) compared to the VHDL code
  - The ASM chart is the documentation of your VHDL code, and must show all the states, the correct state transitions and the decisions (based on inputs).
  - Outputs can be given in the ASM chart and/or in a state-output table (e.g. give the most important outputs in the ASM chart to ensure correct coding)

Avoid latches

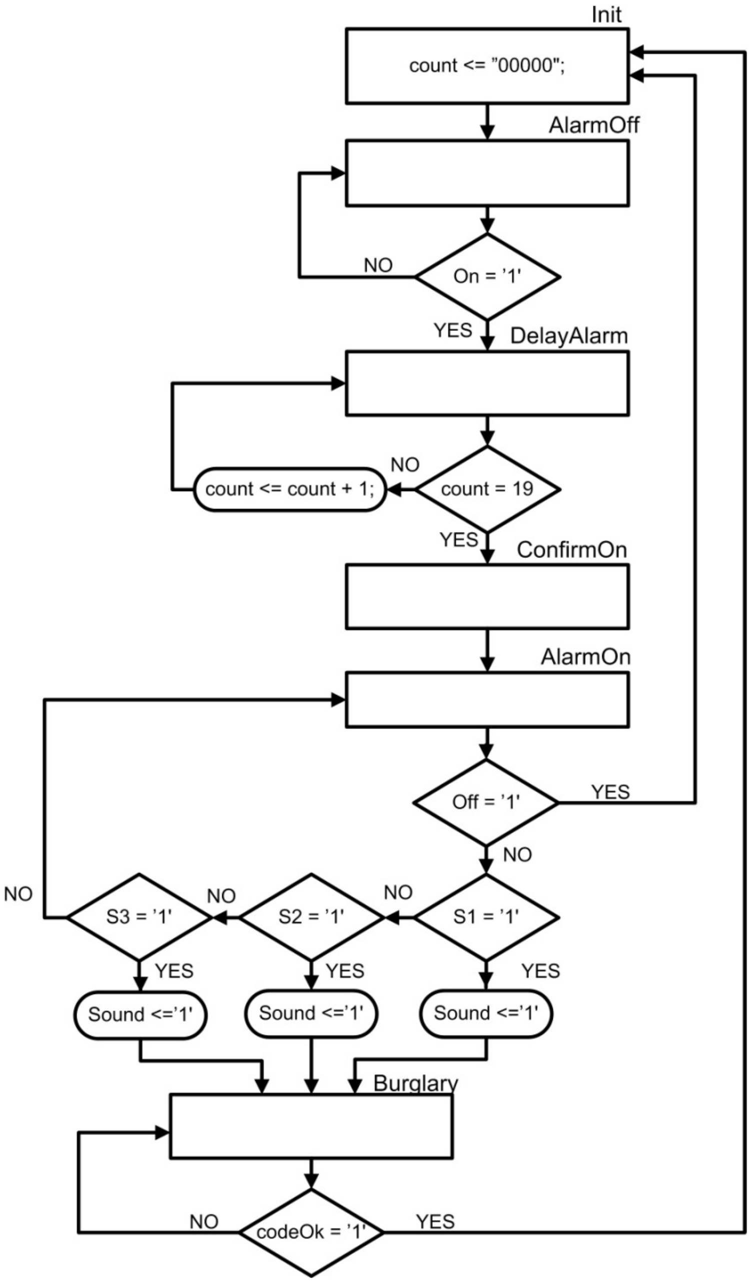
- Latches are created by "if" statements which are not completely specified.
- A Latch is created when an "else" statement is omitted, when values are not assigned a value, or when the "event" statement is missing.
- To avoid a Latch being developed assign an output for all possible input conditions.
  - Use an "else" statement instead of an "elsif" statement in the final branch of an "if" statement to avoid a latch.
  - Be sure to assign default values for all outputs at the beginning of a process.

```
-- VHDL Latch example
    process (enable, data_in)
    begin
        if enable = '1' then
        q <= data_in;
        end if;
    end process;
```

```
-- VHDL D flip-flop  example
process (clk) begin
        if (clk'event and clk = '1') then
                q <= d;
        end if;
end process;
end example;
```

Example of a good ASM chart:

- Yes/No (1/0) labels together with decision boxes
- Arrows to show the program flow
- State names in top right corner of the state boxes
- Selected outputs listed in conditional output boxes and inside the state boxes



State flow diagram and output tabell

Tilstandsdiagram

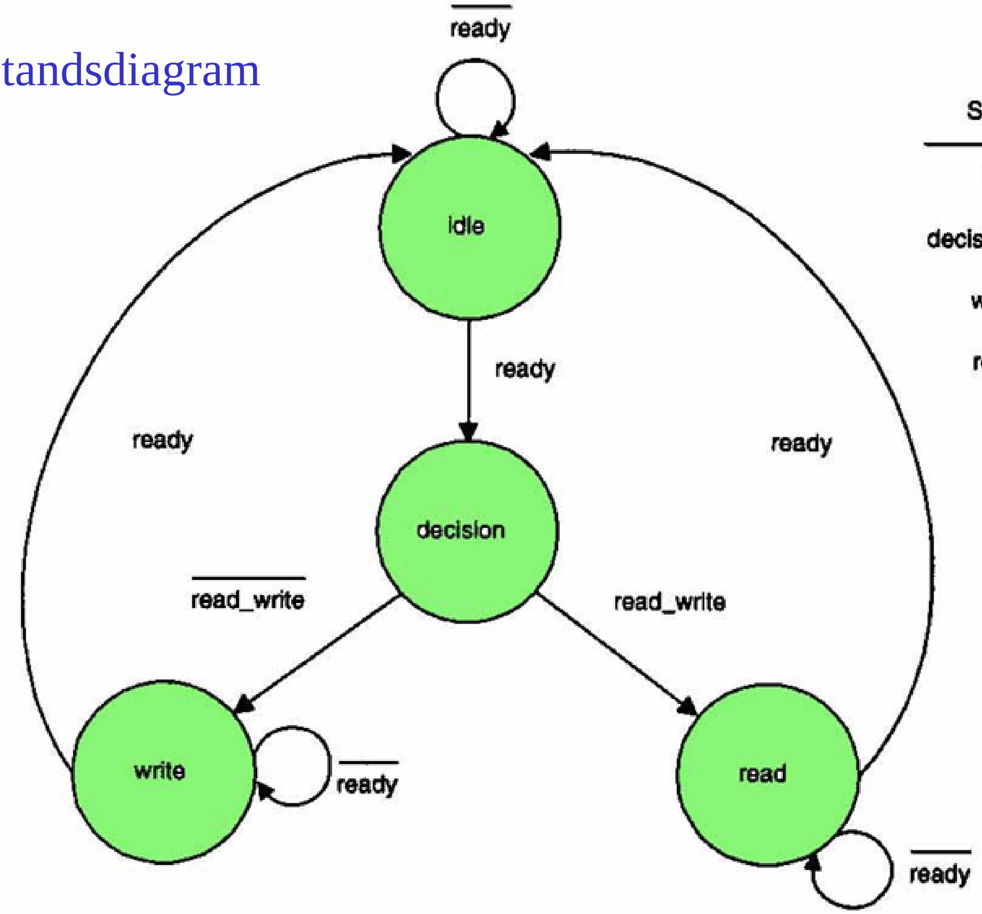


Figure 5-1 Simple state machine

Output tabell

	Outputs	
State	oe	we
idle	0	0
decision	0	0
write	0	1
read	1	0

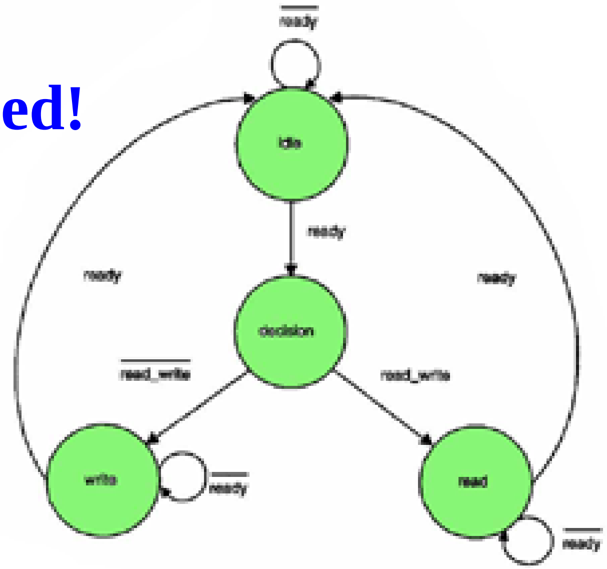
$ready$ = ‘1’

$\overline{ready}$ = ‘0’

State machines in VHDL

- An ASM chart (or a state diagram) can easily be translated to a VHDL description!

- In VHDL the state machine can be described in two different ways:
  - 1-process FSM ⟵ **Recommended!**
  - 2-process FSM
    - One process describes the <u>combinational logic</u>, and another describes <u>synchronization</u> of state transitions to the clock



Example: 2 process FSM

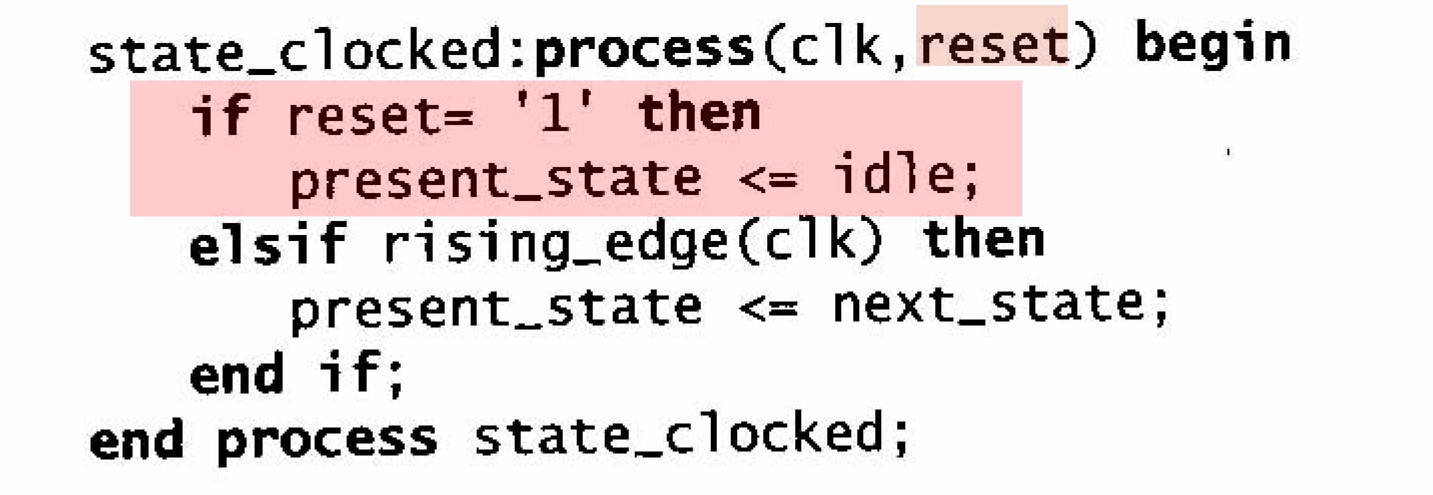
```
library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
    reset, read_write, ready, burst, clk     : in std_logic;
    bus_id                                  : in std_logic_vector(7 downto 0);
    oe, we                                  : out std_logic;
    addr                                    : out std_logic_vector(1 downto 0));
end memory_controller;

architecture state_machine of memory_controller is
                type StateType is (idle, decision, read1, read2, read3, read4, write);
                signal present_state, next_state : StateType;
begin
state_comb:process(reset, bus_id, present_state, burst, read_write, ready) begin
                if (reset = '1') then
                                oe <= '-'; we <= '-'; addr <= "--";
                                next_state <= idle;
                else
                                case present_state is
                                                …
                                end case;
                end if;
end process state_comb;
```

Combinational logic

```
state_clocked:process(clk) begin
                if rising_edge(clk) then
                                present_state <= next_state;
                end if;
end process state_clocked;
end;
```

Synchronous logic

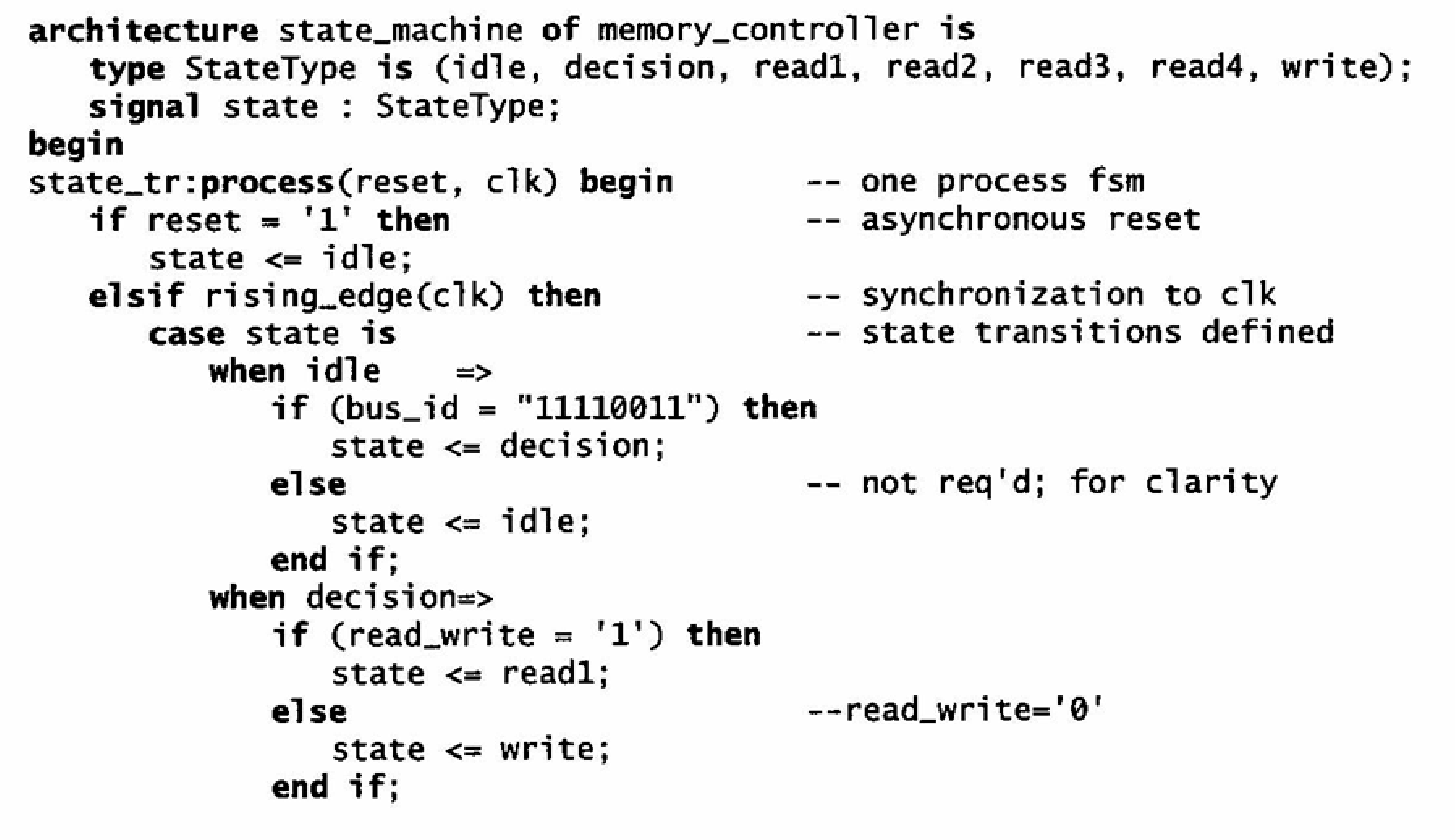
Asynchronous reset in 2-process FSM



```
state_clocked:process(clk,reset) begin
   if reset= '1' then
      present_state <= idle;
   elsif rising_edge(clk) then
      present_state <= next_state;
   end if;
end process state_clocked;
```

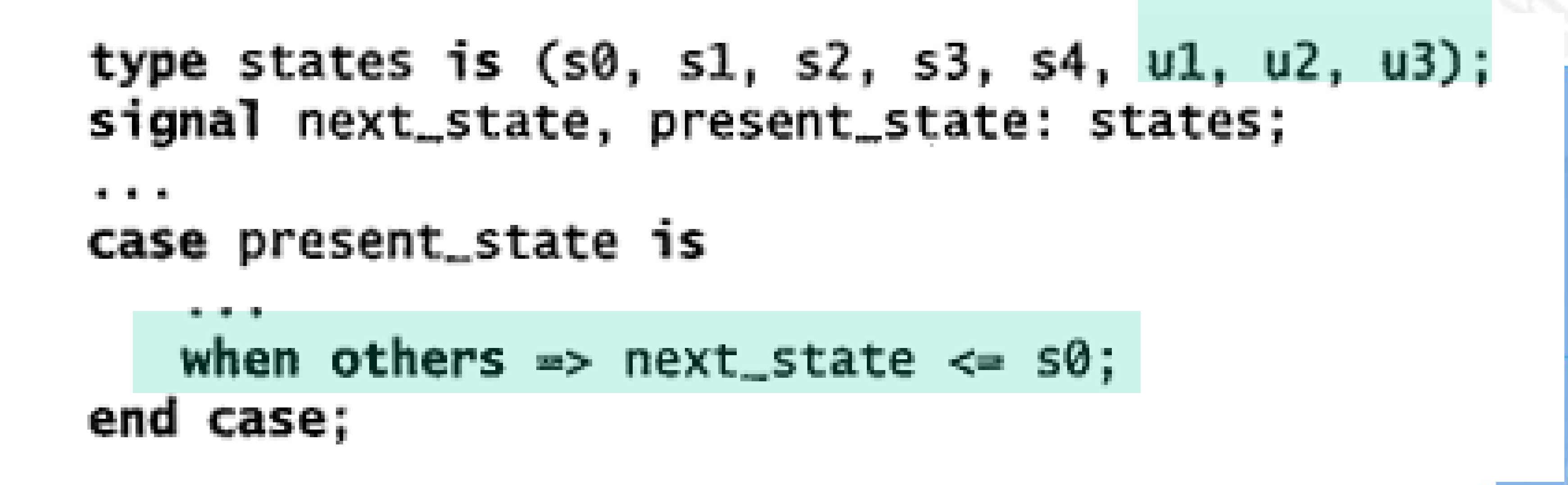
1-process FSM

Functionally identical to the 2-process FSM, and the same logic is produced



```
architecture state_machine of memory_controller is
   type StateType is (idle, decision, read1, read2, read3, read4, write);
   signal state : StateType;
begin
state_tr:process(reset, clk) begin              -- one process fsm
   if reset = '1' then                          -- asynchronous reset
      state <= idle;
   elsif rising_edge(clk) then                  -- synchronization to clk
      case state is                             -- state transitions defined
         when idle    =>
            if (bus_id = "11110011") then
               state <= decision;
            else                                -- not req'd; for clarity
               state <= idle;
            end if;
         when decision=>
            if (read_write = '1') then
               state <= read1;
            else                                --read_write='0'
               state <= write;
            end if;
```

FSM with 3 undefined states



```
type states is (s0, s1, s2, s3, s4, u1, u2, u3);
signal next_state, present_state: states;
...
case present_state is
   ...
   when others => next_state <= s0;
end case;
```

Output from a state machine

1)

```
FSM: process (reset, clk)
begin
  if (reset = ’1’) then
  state <= idle;

 elsif rising_edge (clk) then
   case state is
      when idle =>
        oe <= ’1’;
        if (input1 = ’1’) then
            state <= s1;
        else
            state <= idle;
        end if;
      when s1 =>
         oe <= ’0’;

      .......
```

Recommended!

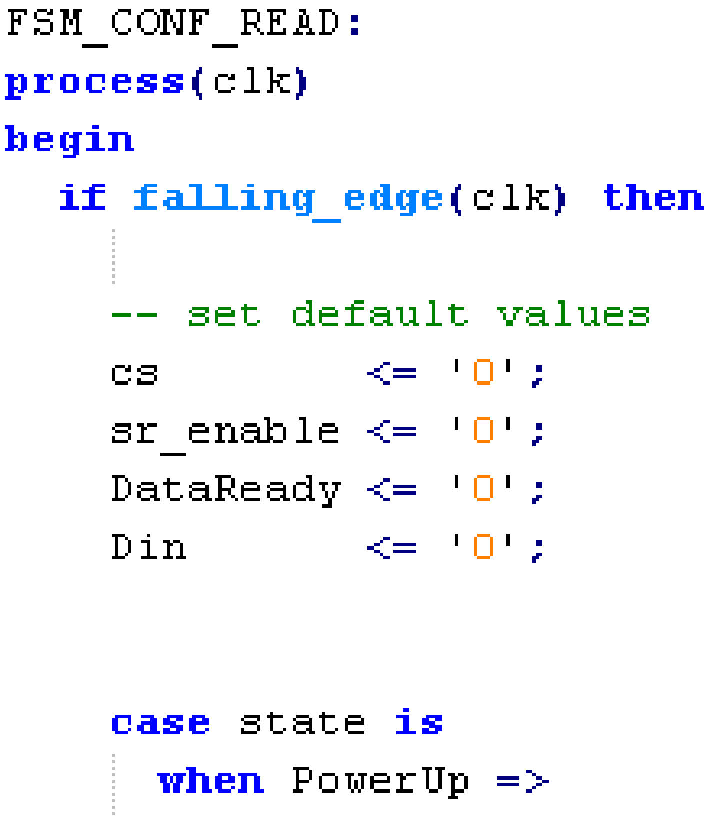
2)

```
FSM: process (reset, clk)
begin
  if (reset = ’1’) then
  state <= idle;

 elsif rising_edge (clk) then
   oe <= ’1’; -- Default value
   case state is
      when idle =>
         if (input1 = ’1’) then
            state <= s1;
        else
            state <= idle;
        end if;
      when s1 =>
         oe <= ’0’;

      .......
```

3)

```
FSM: process (reset, clk)
begin
  if (reset = ’1’) then
  state <= idle;

 elsif rising_edge (clk) then
   case state is
      when idle =>
          if (input1 = ’1’) then
             state <= s1;
         else
             state <= idle;
         end if;
      when s1 =>
      .......
   end process;
with state select
  oe <= ’1’ when idle,
          ’0’ when S1;
end state_machine;
```

Example: Use of default values in FSMs

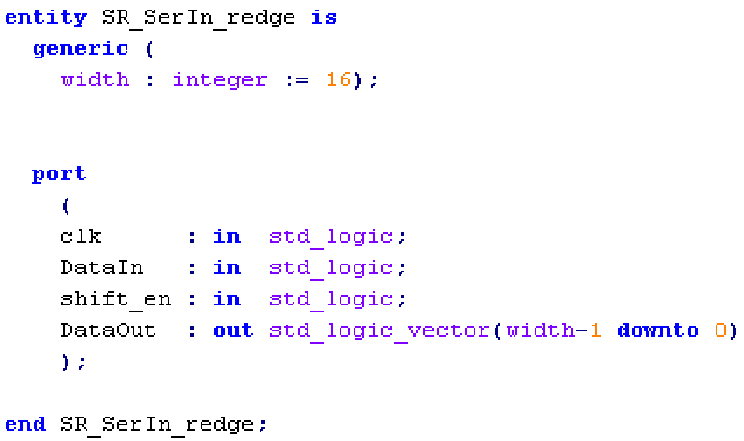


```
FSM_CONF_READ:
process(clk)
begin
  if falling_edge(clk) then

    -- set default values
    cs        <= '0';
    sr_enable <= '0';
    DataReady <= '0';
    Din       <= '0';


    case state is
      when PowerUp =>
```

Generics

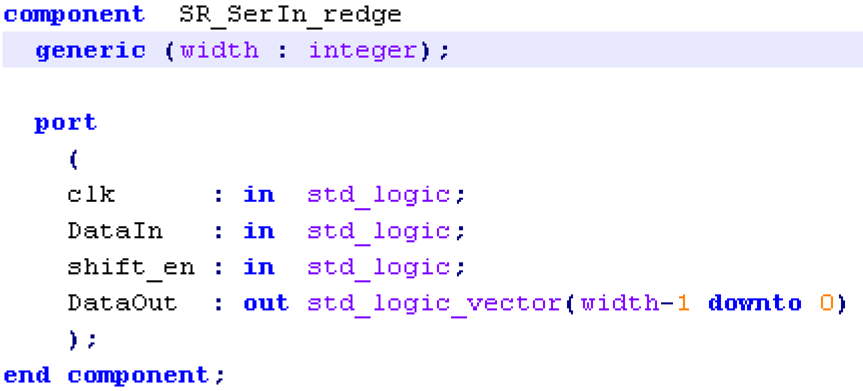
- The width of a signal can be specified using a parameter
- Useful for **registers** and **counters** with different number of bits; only necessary to make one component

Entity using a generic

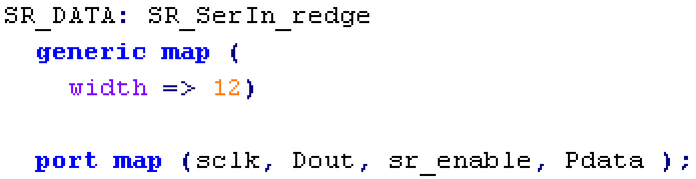


```
entity SR_SerIn_redge is
  generic (
    width : integer := 16);


  port
    (
    clk      : in  std_logic;
    DataIn   : in  std_logic;
    shift_en : in  std_logic;
    DataOut  : out std_logic_vector(width-1 downto 0)
    );

end SR_SerIn_redge;
```

How to use this component with a generic

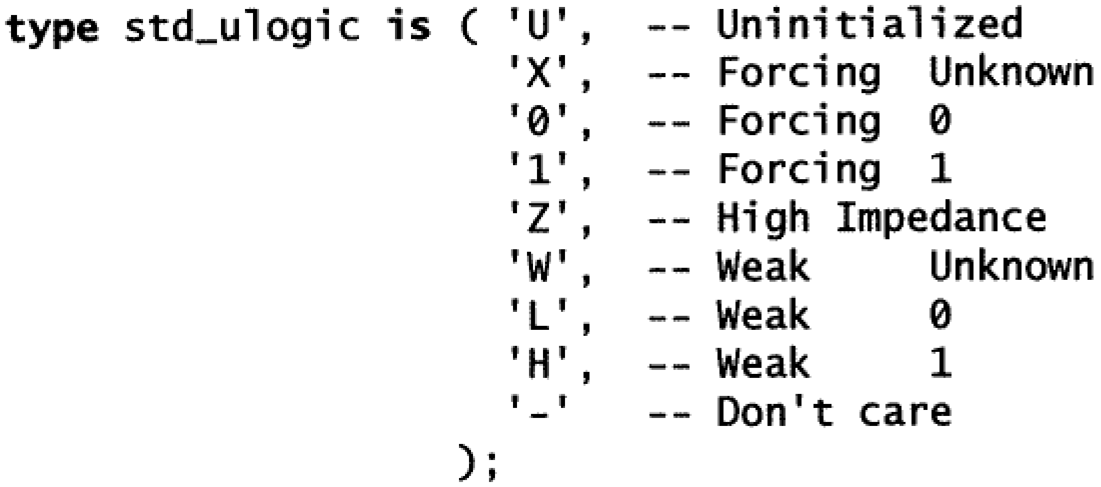


```
component  SR_SerIn_redge
  generic (width : integer);

  port
    (
    clk      : in  std_logic;
    DataIn   : in  std_logic;
    shift_en : in  std_logic;
    DataOut  : out std_logic_vector(width-1 downto 0)
    );
end component;
```



```
SR_DATA: SR_SerIn_redge
  generic map (
    width => 12)

  port map (sclk, Dout, sr_enable, Pdata );
```

The data type std_logic 1164

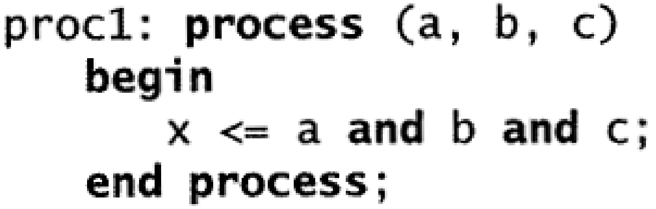


```
type std_ulogic is ( 'U',   -- Uninitialized
                     'X',   -- Forcing  Unknown
                     '0',   -- Forcing  0
                     '1',   -- Forcing  1
                     'Z',   -- High Impedance
                     'W',   -- Weak     Unknown
                     'L',   -- Weak     0
                     'H',   -- Weak     1
                     '-'    -- Don't care
                   );
```

9 different values!

”Process”

- The process is executed when one of the signals in the sensitivity list has a change (an event)
- Then, the sequential signal assignments are executed
- The process continue to the last signal assignment, and terminates
- **The signals are updated just before the process terminates!**
- The process is not executed again before one of the signals in the sensitivity list has a new event (change)



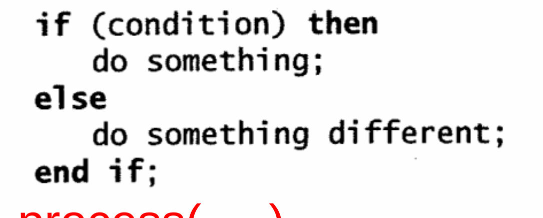
```
proc1: process (a, b, c)
    begin
        x <= a and b and c;
    end process;
```

```
process (<sens list>)
< declaration>
begin
    <signal assignment1>
    .
    .
    <signal assignment n>
end process;
```

Sequential vs concurrent statements

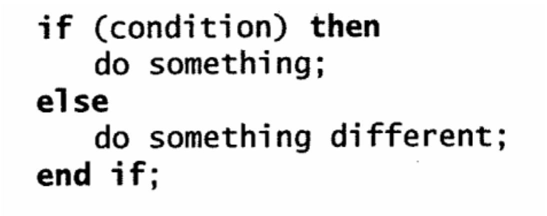
Sequential statements

process(.....)



```
if (condition) then
    do something;
else
    do something different;
end if;
```

process(.....)



```
if (condition) then
    do something;
else
    do something different;
end if;
```

- The order of the sequential statements (in the process) is important!
- If there are multiple processes they are all executed in parallel and concurrent with other ”concurrent statements” in the architecture!

Concurrent statements

aeqb <= '1' **when** (a = b) **else** '0';

ceqd <= '1' **when** (c = d) **else** '0';

- ”*concurrent statements*” are used outside ”*process*”
- Executed concurrently (samtidig)
- The order of ”concurrent statements” is arbitrary

Signals and variables

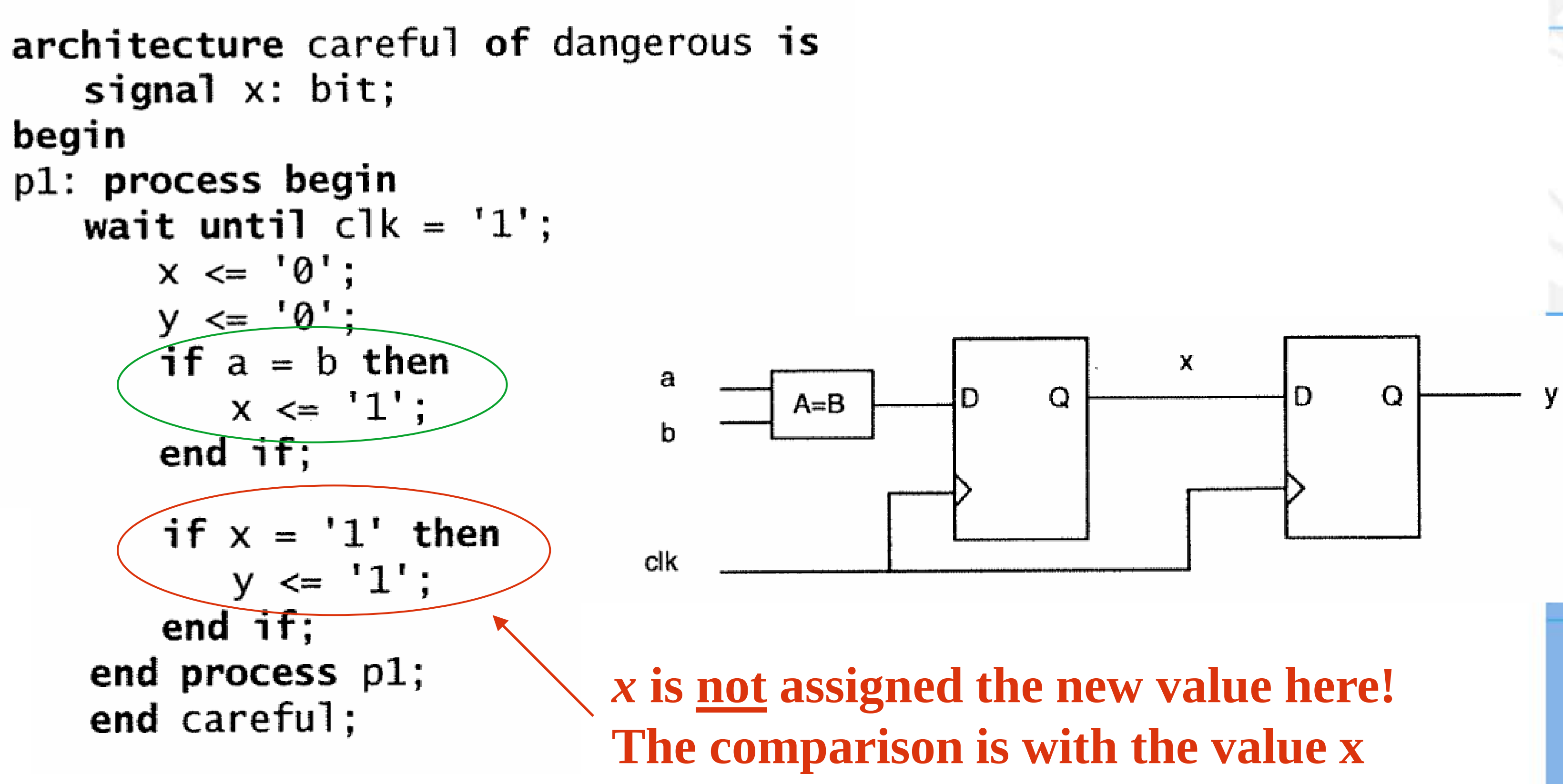
Signals:

- Signal assignment <=
- Defined in architecture (before ***begin)***
- Signals are updated just before the process terminates!
- Use signals instead of variables when possible!

Variable:

- Variable assignment : =
- Variable assignment is instantaneous
- In synthesis they are used as index variables and temporal storage of data
- Can be used to simplify algorithms
- Can be used inside a process
- Must be defined inside a process

Important - signal assignment



```
architecture careful of dangerous is
   signal x: bit;
begin
p1: process begin
   wait until clk = '1';
      x <= '0';
      y <= '0';
      if a = b then
         x <= '1';
      end if;

      if x = '1' then
         y <= '1';
      end if;
   end process p1;
   end careful;
```

x **is <u>not</u> assigned the new value here! The comparison is with the value x got the last time the process was executed!**

Listing 4-37 A signal in an assignment and as an operand within the same process

Building blocks

- Library
  - library IEEE
  - use IEEE.std_logic_1164.all;

To make the library visible

- Packages
  - use work.my_package.all; -- your own package
  - **work** is the directory where the design files are located
- Components
  - An entity used in another entity
  - Needs a component declaration to make the component visible
  - Needs a component instantiation to connect the component to the top entity – using ***port map()***

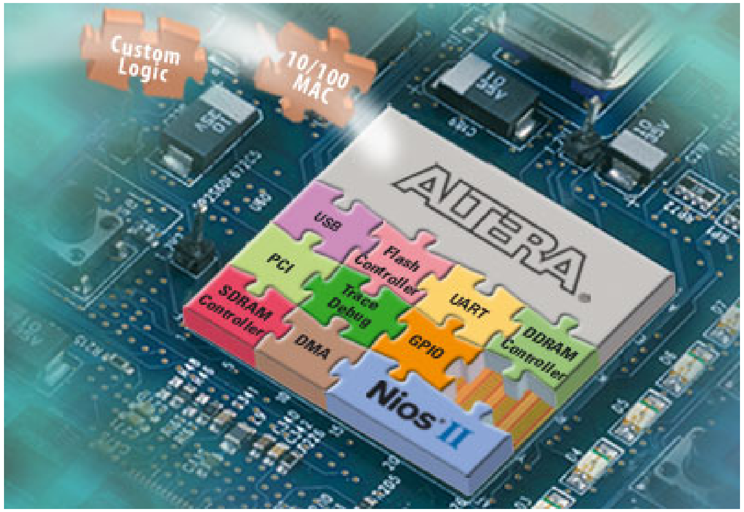
Operator overloading & important functions

- Understand the need and use of the following packages:

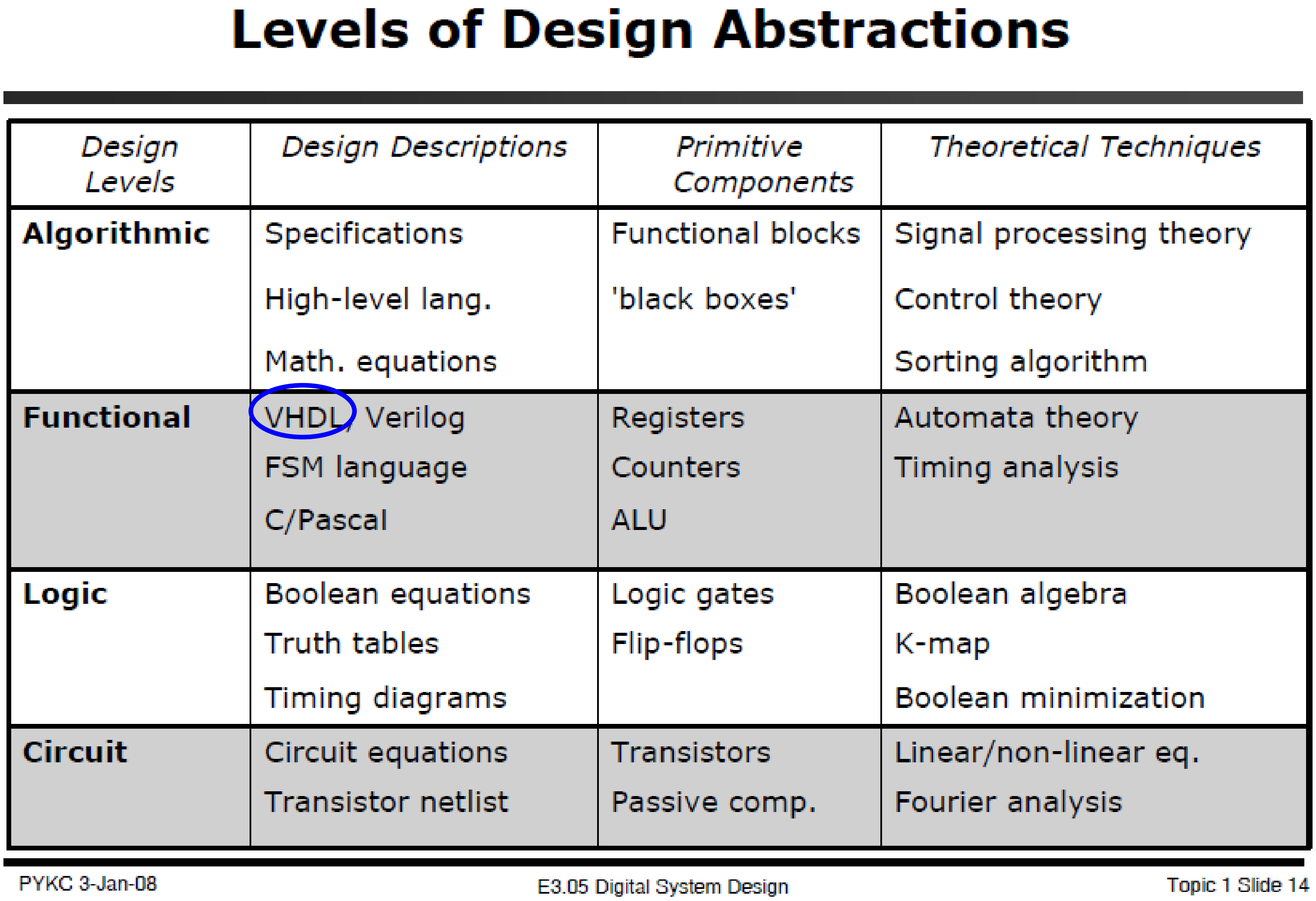
  use IEEE.**std_logic_1164**.all;
  use IEEE.**numeric_std**.all;
  use IEEE.**std_logic_arith**.all;

FPGA advantages

- High reliability
- High determinism
- High performance
- True parallelism
- Reconfigurability



- Scalability
- System integration (System On a Programmable Chip)



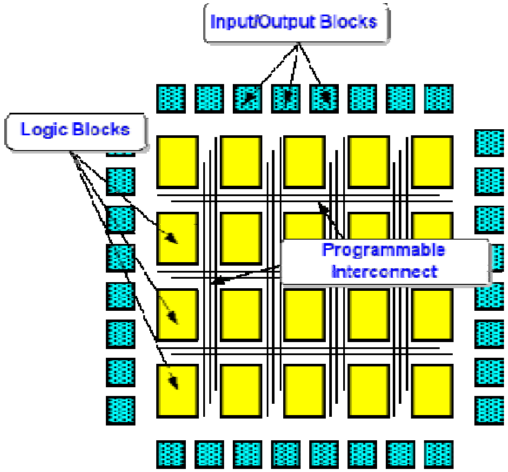
Levels of Design Abstractions

Design Levels	*Design Descriptions*	*Primitive Components*	*Theoretical Techniques*
Algorithmic	Specifications High-level lang. Math. equations	Functional blocks 'black boxes'	Signal processing theory Control theory Sorting algorithm
Functional	VHDL, Verilog FSM language C/Pascal	Registers Counters ALU	Automata theory Timing analysis
Logic	Boolean equations Truth tables Timing diagrams	Logic gates Flip-flops	Boolean algebra K-map Boolean minimization
Circuit	Circuit equations Transistor netlist	Transistors Passive comp.	Linear/non-linear eq. Fourier analysis

PYKC 3-Jan-08 E3.05 Digital System Design Topic 1 Slide 14

The difference between a processor and programmable logic

- A processor is programmed with ***instructions***
- A programmable logic circuit is programmed with ***a circuit description***
- A programmable logic circuit contains configurable blocks with logics and configurable connection lines between these blocks

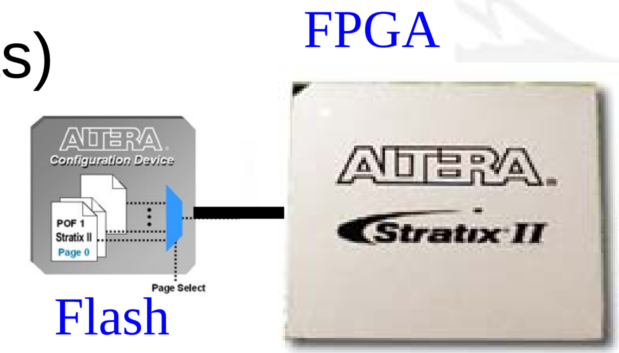


CPLD - Complex Programmable Logic Device

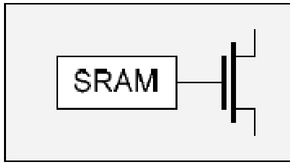
- Programming technology: non-volatile memory, such as EEPROM or FLASH.
  - Configuration stored in the circuit (even without power)
  - High voltage (EEPROM) or logic voltage (FLASH)
- Used in ”small and medium size” designs

FPGA - Field Programmable Gate Array

- Typically contains more logic then a CPLD
- Have many flip-flops (memory elements)
- Usually have on-chip memory (RAM)
- Supports processor cores, IPs etc



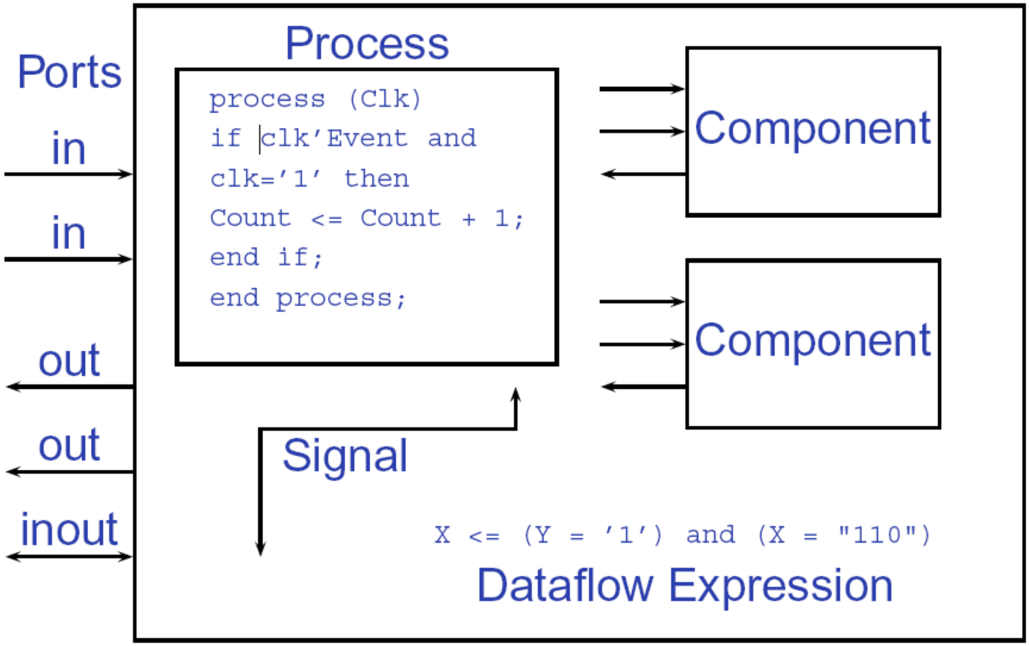
- Programming technology: usually static memory (SRAM)
  - Needs an external configuration circuit with a non-volatile memory (based on EEPROM/FLASH) which loads the configuration into the FPGA at power on.
  - SRAM memory inside the FPGA stores the circuit configuration (when the power is on).



Combinational vs. Sequential logic, and synchronous logic

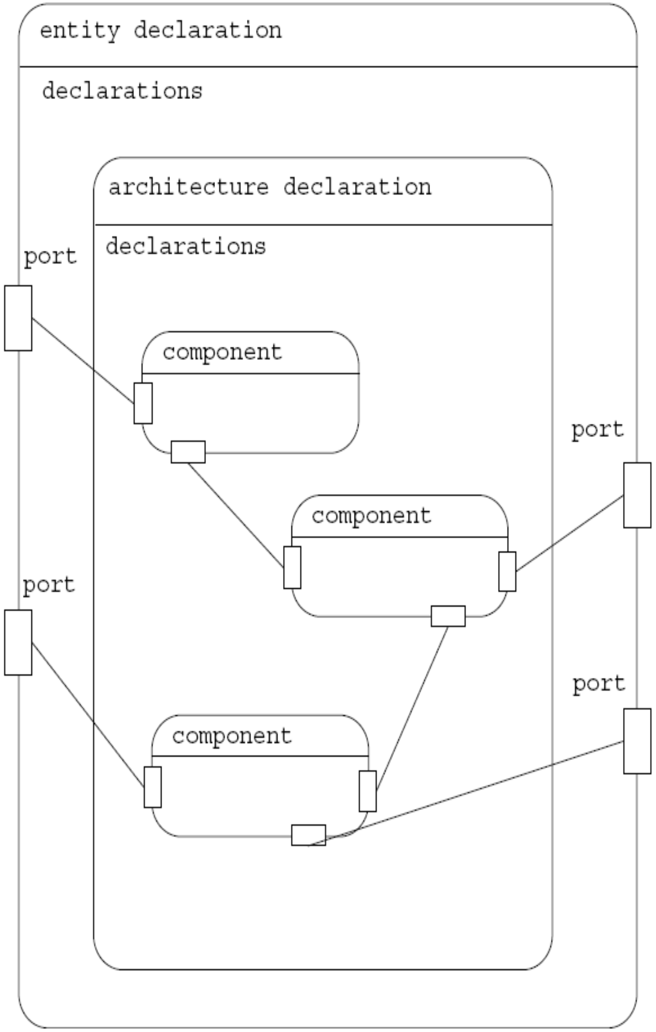
- In **combinational logic** the output is only dependent on the present input.
- In **sequential logic** the output is dependent on both the present input and the state (memory, based on earlier inputs).
- Synchronous logic use a clock such that the memory elements are updated only at specific times (at the rising/falling clock edge)

Block diagrams



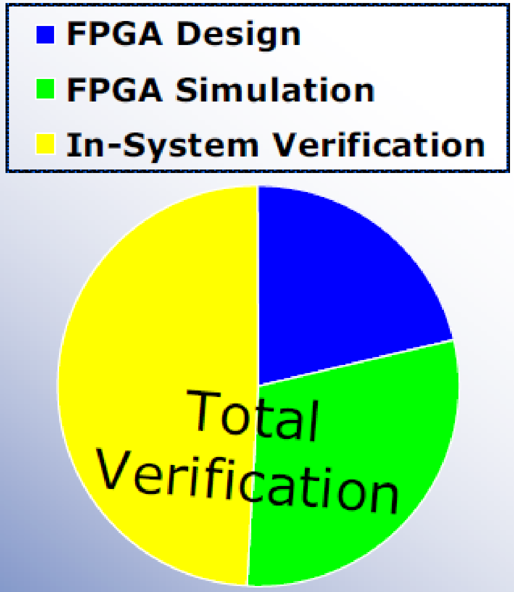
```
process (Clk)
if clk'Event and
clk='1' then
Count <= Count + 1;
end if;
end process;
```

```
X <= (Y = '1') and (X = "110")
```



Why focus on Verification

Consider a) FPGA development and
b) Further work related to FPGA quality



Average Design & Functional Verification tasks

- as seen in some reasonably structured projects.

Test benches

- Add a stimuli (input) to the circuit under test, using VHDL, and observe the outputs to verify correct behavior/functionality
- Can have a table with test vectors integrated into the test bench or in a separate file
- Test benches are not to be synthesized, and can therefore use the entire VHDL language (e.g. **after**)
- File I/O Package defined in IEEE 1076: **textio**
  - Read test patterns from file
  - Write results to file and compare manually with an answer file
  - The test bench can also read the answer file such that the test bench can compare the results and the correct answers

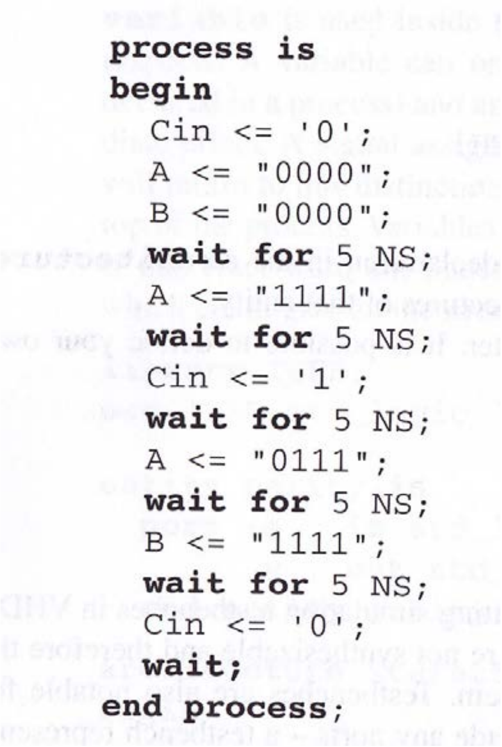
- Can build in models for external circuits on the PCB
  - demands correct modeling of the external circuits

Testbench example

signal clk : **std_logic** :=’0’;

begin

clk <= **not**(clk) **after** 50 ns; **--** gives a clock period of 100 ns



```
process is
begin
  Cin <= '0';
  A <= "0000";
  B <= "0000";
  wait for 5 NS;
  A <= "1111";
  wait for 5 NS;
  Cin <= '1';
  wait for 5 NS;
  A <= "0111";
  wait for 5 NS;
  B <= "1111";
  wait for 5 NS;
  Cin <= '0';
  wait;
end process;
```