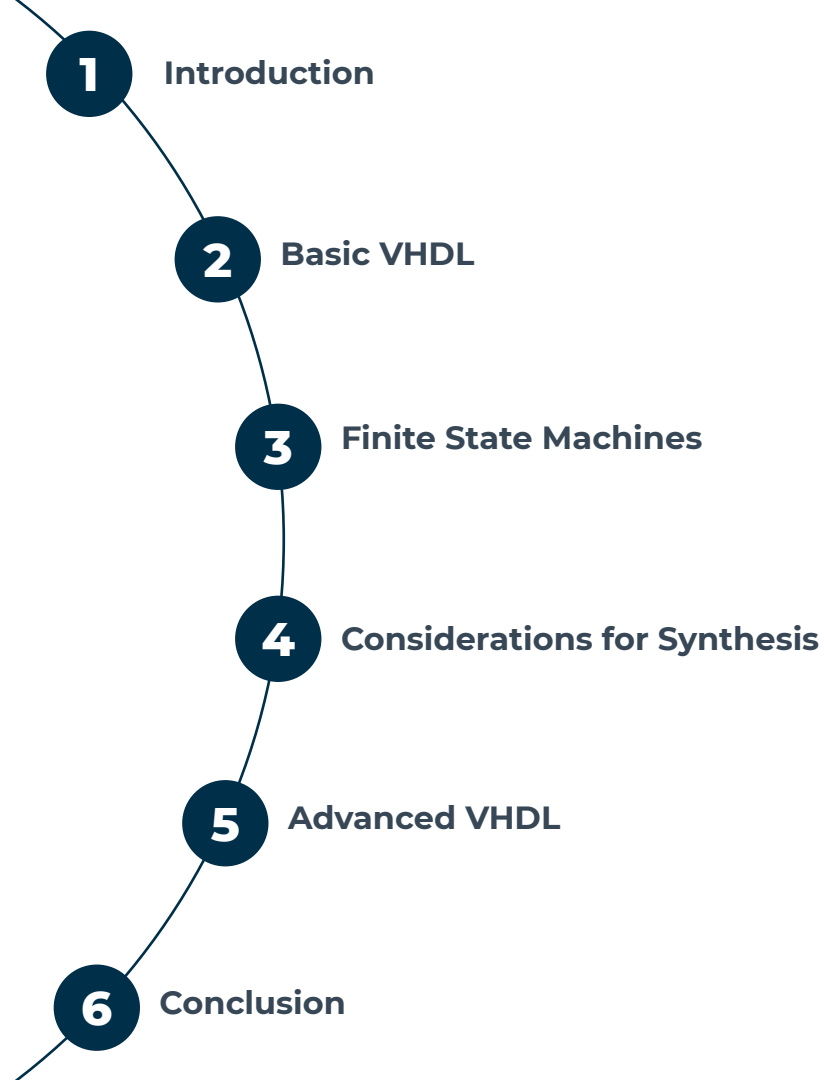
# VHDL for FPGA Synthesis

Rodrigo A. Melo

Oct | 2021

**Joint ICTP, SAIFR and UNESP School on Systems-on-Chip, Embedded Microcontrollers and their Applications in Research and Industry | (smr 3557)**

# Outline


1
Introduction
2
Basic VHDL
3
Finite State Machines
4
Considerations for Synthesis
5
Advanced VHDL
6
Conclusion

# Introduction

# Hardware Description Languages

**VHDL**

- **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit (**VHSIC**) + **HDL**
- U.S. Department of Defense (1983)
- Standard IEEE 1076 (87, **93**, 00, 02, 08, 19)

**Verilog**

- **VERI**fication + **LOG**ic
- Gateway Design Automation (1984), Cadence (1990)
- Standard IEEE 1364 (95, **01**, 05)
- Verilog is now part of System Verilog (IEEE 1800)

*Xilinx is now shipping Foundation Series design solutions capable of supporting both VHDL and Verilog.*

**The Foundation Series Software Now Delivers VHDL and Verilog**

Our Foundation Series software remains simple, yet continues to deliver complete front-to-back development tools supporting all Xilinx CPLD and FPGA devices. There are two schematic packages and two HDL packages. Both Foundation Series HDL packages include VHDL and Verilog, design entry, synthesis using the Synopsys FPGA Express, and implementation tools from Xilinx.

Xilinx offers comprehensive support services providing assistance on Xilinx technologies as well as advanced application assistance.

**Powerful Synthesis**

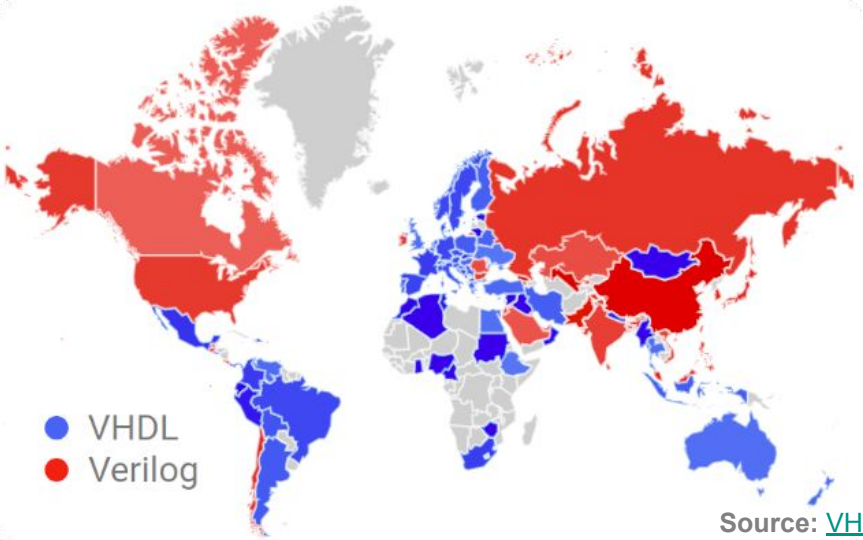
This past year, Xilinx added the Synopsys Express technology to the Foundation Series software. With Synopsys' history of providing state-of-the-art synthesis solutions for high-density designs, it's not surprising that the

Xcell Journal, issue 27, 1998

The most widely supported standards are **VHDL 93** and **Verilog 2001**.

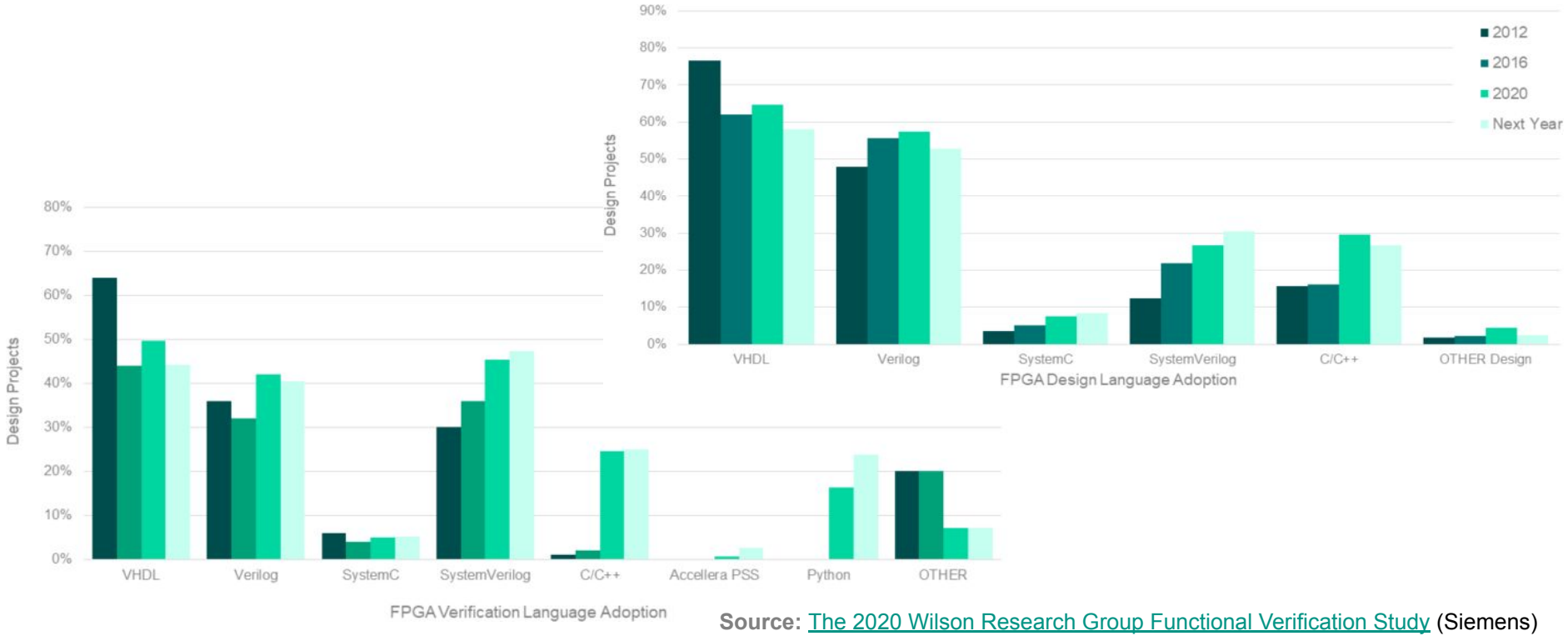
# VHDL vs Verilog

- VHDL is strongly typed, CasE InSEnSiTiVe and supports libraries. Based on ADA.
- Verilog is weakly typed, case sensitive and doesn't support libraries. C-like syntax.
- You can achieve the same with both of them.
- Verilog is more concise but allows you to write wrong code.



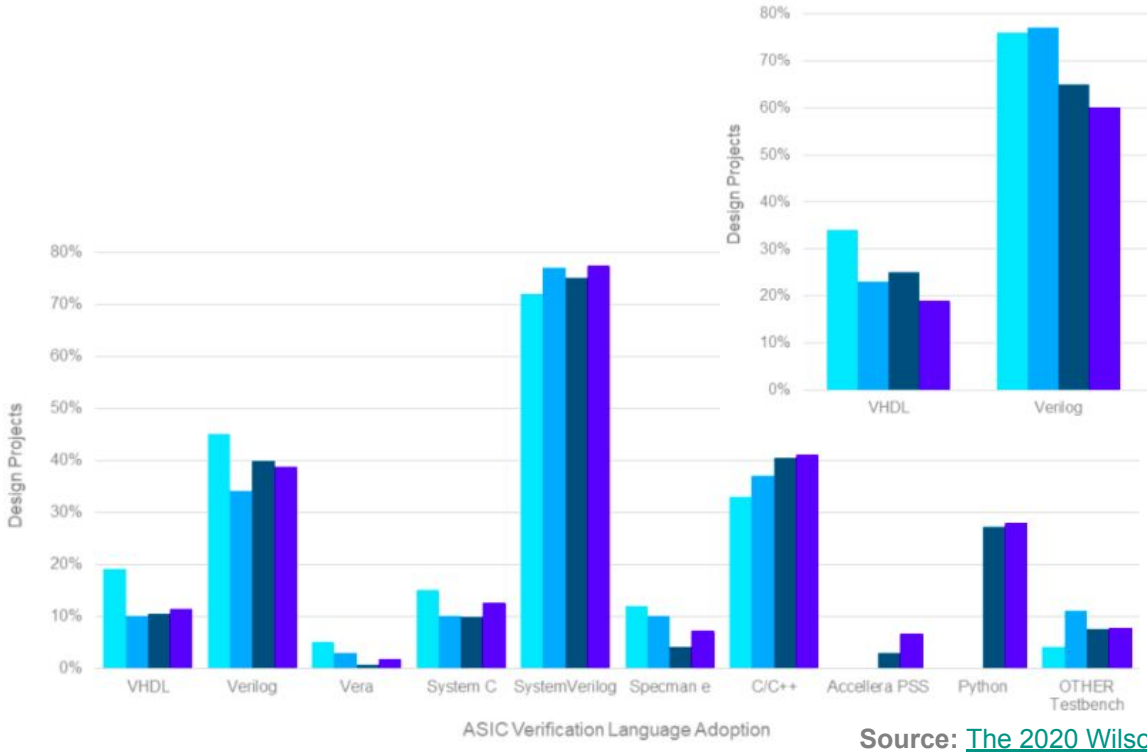

**Source:** VHDL vs Verilog (last 5 years) [Oct 2, 2021]

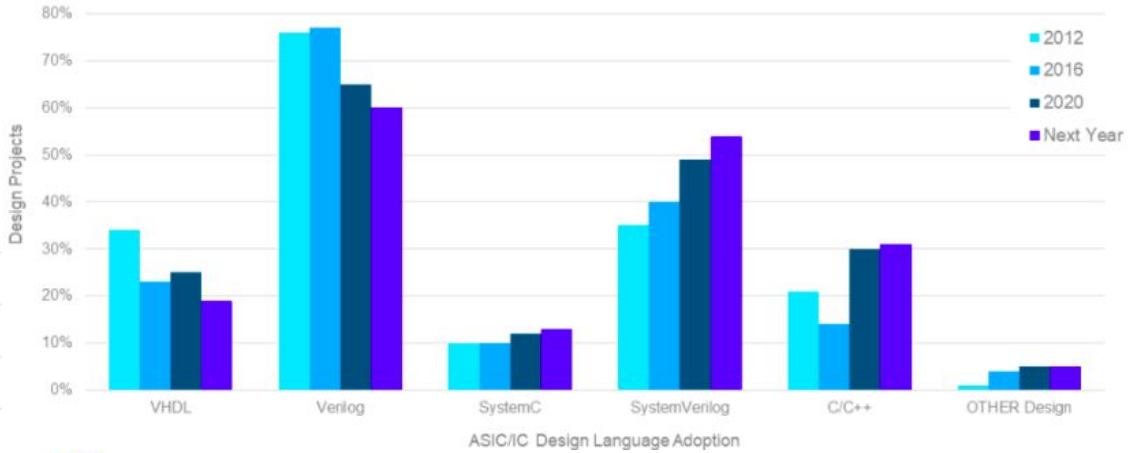
# FPGA languages trends




**Source:** [The 2020 Wilson Research Group Functional Verification Study](#) (Siemens)

# ASIC languages trends




**Source:** [The 2020 Wilson Research Group Functional Verification Study](#) (Siemens)

# Alternatives

**HLS**

- AKA algorithmic/behavioral synthesis.
- Subset of C (or variants) + directives (vendor-specific).
- The result is an extremely vendor-specific RTL.
- Useful for architecture exploration of algorithms.
- Can't be used to create processors or controllers, neither to deal with multiple clock domains.

**Others**

- Python based: (n)Migen, MyHDL
- Scala based: Chisel, SpinalHDL
- Haskell based: Clash, Bluespec
- Verilog based: TL-Verilog, Slice
- And more...


# HDL for Synthesis

- Only a small subset of the language is synthesizable.
- It is used to describe the behavior and/or the structure of a digital design.
- You are no writing a software program, you are describing hardware (concurrent code, executed in parallel).
- You can write small combinational circuits parts (asynchronous) but is recommendable to perform synchronous design (based on one or multiple clocks).

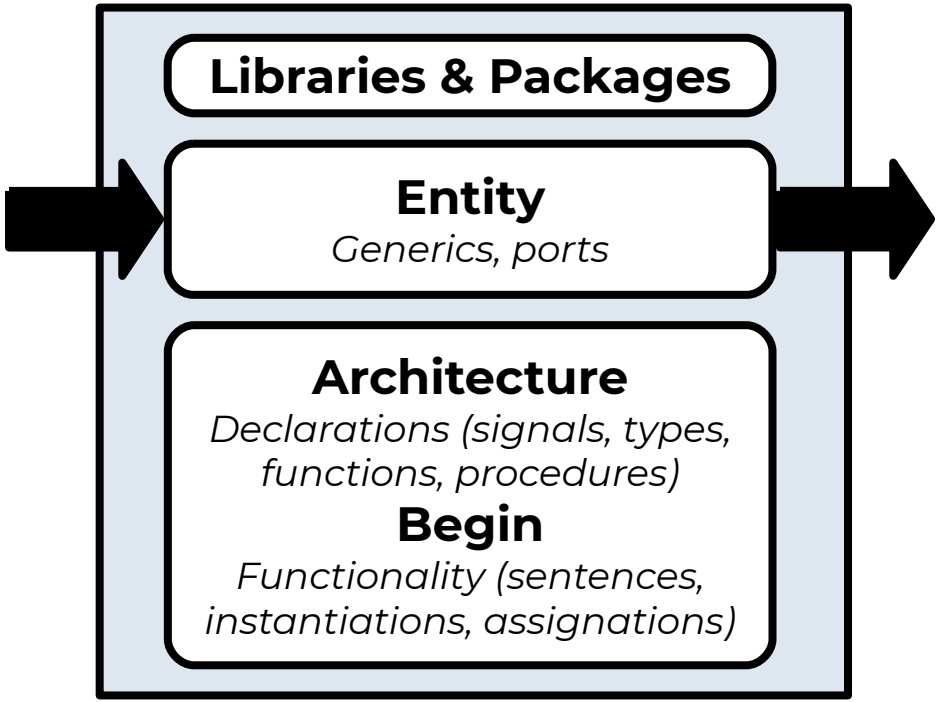
Now, we will take a course about the VHDL fundamentals for synthesis.

**Our basic guidelines**

- Use UPPERCASE for constants
- Use indentation (4 spaces)
- Use coherent_and_descriptive names
- Use meaningful prefixes/suffixes (_i, _o, _r)

# Basic VHDL

# Structure of a component




The file extension is usually **vhd** or **vhdl**

- Grouped into **Libraries**, a **Package** provides data types, functions and components, to extend the language support.
- The **Entity** defines the name and the interface of our component.
- An **Architecture** implements the functionality of a given **Entity**.

Commonly, you will have one Entity and one or more related Architectures per file.

# Libraries and Packages inclusion

```
-- library LIBRARY;
-- use LIBRARY.PACKAGE.all;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Synopsys non-standard packages
-- use IEEE.std_logic_arith.all;
-- use IEEE.std_logic_signed.all;
-- use IEEE.std_logic_unsigned.all;
```

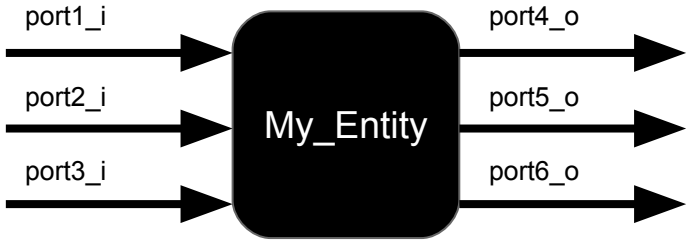
Mandatory for synthesizable designs. It provides the *std_logic* (1 bit) and *std_logic_vector* (bus) types (to support 'U', 'X', '**0**', '**1**', '**Z**', 'W', 'L', 'H' and '**-**' instead of only '0' and '1')

**These packages are commonly found in examples, but they are non-standard. Avoid them!!!**

The *numeric_std* package provides arithmetic functions for vectors. It derives from *std_logic_vector* two other types: *signed* and *unsigned*.

✓ I recommend to know the content (read the source code) of these two packages

# Entity




✓ use only **std_logic** and **std_logic_vector** types

✓ use only **in** and **out** modes

```
entity My_Entity is
    port (
        -- name: mode type [:=default];
        port1_i : in  std_logic;
        port2_i : in  std_logic;
        port3_i : in  std_logic_vector(7 downto 0);
        port4_o : out std_logic_vector(7 downto 0);
        port5_o : out std_logic;
        port6_o : out std_logic -- don't use ';' here
    );
end entity My_Entity;
```

Opt Opt

⚠ use **inout** mode only in the Top-level.

🚫 don't use the **buffer** mode.

# Architecture

```
architecture My_Arch of My_Entity is
    -- declarations
begin
    -- instantiations
    -- concurrent statements
    -- sequential statements
end architecture My_Arch;
```

Opt Opt

- Each Architecture belongs to an Entity (**of**).
- Generally, you will have more than one Architecture per Entity when looking for alternatives to the same functionality (high-speed vs area, different algorithms, etc).
- The Architecture is where you “design” your component.

# Our first example

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

entity nor3 is
    port (
        a_i : in  std_logic;
        b_i : in  std_logic;
        c_i : in  std_logic;
        q_o : out std_logic
    );
end entity nor3;

architecture rtl of nor3 is
begin
    q_o <= not(a_i or b_i or c_i);
end architecture rtl;
```

- Imports std_logic
- Entity definition
  - 3 x 1-bit inputs
  - 1-bit output
- The architecture implements a 3-input NOR logic function

# Instantiation

```
architecture alternative1 of top is
    component nor3 is
        port (
            a_i : in  std_logic;
            b_i : in  std_logic;
            c_i : in  std_logic;
            q_o : out std_logic
        );
    end component nor3;
begin
    -- label : name
    nor3_inst : nor3
    port map (
        a_i => port1_i, b_i => port2_i,
        c_i => port3_i, q_o => port4_o
    );
end architecture alternative1;
```

⚠ Labels are optional but recommended.

⚠ You can use **named** or **positional** association, but the first is strongly recommended (good practice).

Alternative 2: put your component declaration in a user defined package, in your own library.

```
architecture alternative3 of top is
begin
    -- label: entity library.name(arch)
    nor3_inst : entity work.nor3
    port map (
        a_i => port1_i, b_i => port2_i,
        c_i => port3_i, q_o => port4_o
    );
end architecture alternative3;
```

work?

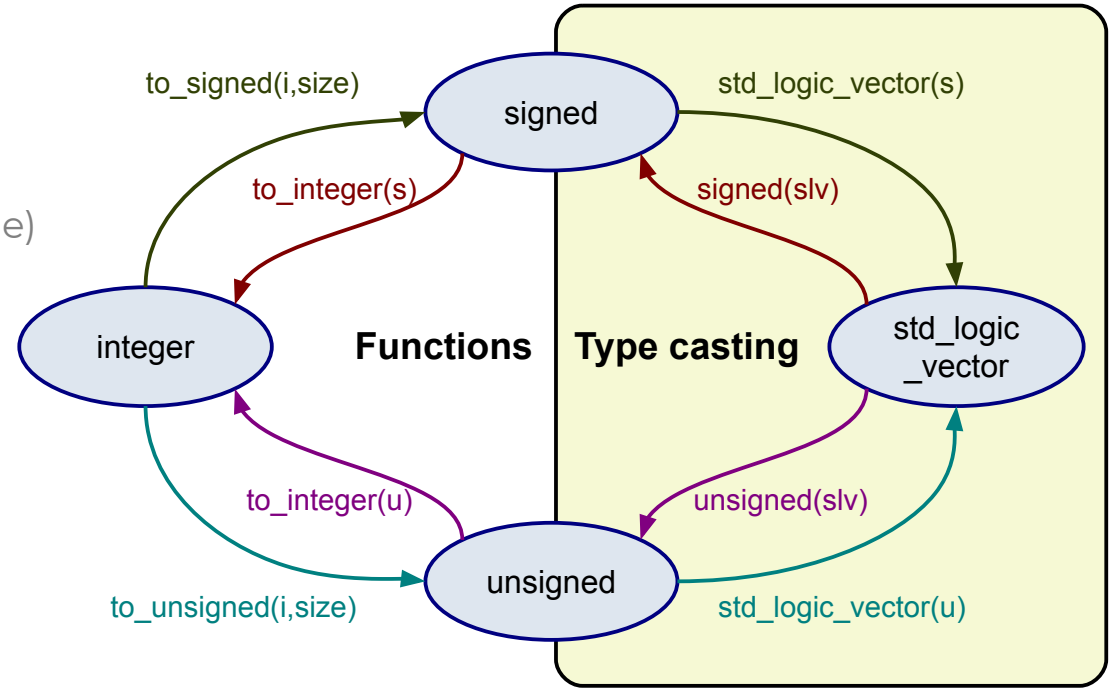
# Most used data types

- std_logic
- std_logic_vectors
- signed/unsigned
- boolean (false, true)

- integer (-, 0, +)
- natural (0, +)
- positive (+)

- real (ej: 3.14)

It doesn't implement floating point



# Operators

| Operator | Description |
|---|---|
| a ** b ⚠ | exponentiation |
| abs a | absolute value |
| not a | complement |
| a * b | multiplication |
| a / b ⚠ | division |
| a mod b ⚠ | modulo |
| a rem b ⚠ | remainder |
| +a | unary plus |
| -a | unary minus |
| a + b | addition |
| a - b | subtraction |
| a & b | concatenation |

| Operator | Description |
|---|---|
| a = b | test for equality |
| a /= b | test for inequality |
| a < b | test for less than |
| a <= b | test for less than or equal |
| a > b | test for greater than |
| a >= b | test for greater than or equal |
| a and b | logical and |
| a or b | logical or |
| a nand b | logical complement of and |
| a nor b | logical complement of or |
| a xor b | logical exclusive or |
| a xnor b | logical complement of exclusive or |

⚠

Most languages uses **==** and **!=** to test in/equality

# Shift/rotate functions

| Operator | Description | Function |
|---|---|---|
| a ssl N ⊘ | shift left logical | shift_left(a, N) |
| a srl N ⊘ | shift right logical | shift_right(a, N) |
| a sla N ⊘ | shift left arithmetic | shift_left(a, N) |
| a sra N ⊘ | shift right arithmetic | shift_right(a, N) |
| a rol N ⊘ | rotate left | rotate_left(a, N) |
| a ror N ⊘ | rotate right | rotate_right(a, N) |

⊘ Wrong defined, don't use them!!! (unexpected behaviour and/or extra hardware).

Defined into the *numeric_std* package. Another useful function there is *resize(a, N)*.

**Explanation:** Arithmetic and logical shifts and rotates are done with functions in VHDL, not operators

# Signals

```
architecture MyArch of MyEntity is
   -- signal name: type [:=default];
   signal slv8 : std_logic_vector(7 downto 0); -- default="UUU"
   signal slv3 : std_logic_vector(2 downto 0):="101";
   signal slv5 : std_logic_vector(4 downto 0):=(others => '0');
   signal slv4 : std_logic_vector(3 downto 0);

   signal slva, slvb, slvc : std_logic_vector(3 downto 0);
   signal to1, to2 : std_logic_vector(1 to 4);
   signal nat : natural range 0 to 15:=1; -- default=0
begin
   -- signal_name <= expression;
   slv8 <= slv3 & slv5;      -- concatenation ("10100000")
   slv4 <= slv5(3 downto 0); -- slice
   slva <= slvb xor slvc;    -- propagation time involved
   to2  <= "0101";           -- hardwired value
   to1  <= to2;              -- connection
end architecture MyArch;
```

**<=** is employed to assign the value of a **signal** (can be time involved).

⚠ For synthesis, always use **range** with integers and its subtypes (natural, positive).

⚠ Default/initial values are ignored by ASIC synthesis tools.

⚠ By convention, we generally use **downto**.

# Concurrent vs Sequential statements

```
architecture MyArch of MyEntity is
   -- declarations
begin
   Concurrent statement;
   Concurrent statement;
   process ()
   begin
      Sequential statement;
      Sequential statement;
      Sequential statement;
   end process;
   Concurrent statement;
   begin
      Sequential statement;
      Sequential statement;
   end process;
end architecture MyArch;
```

**Concurrent Statements**

- Instantiation
- Signal assignment
- **when/else**
- **with/select**
- **process**

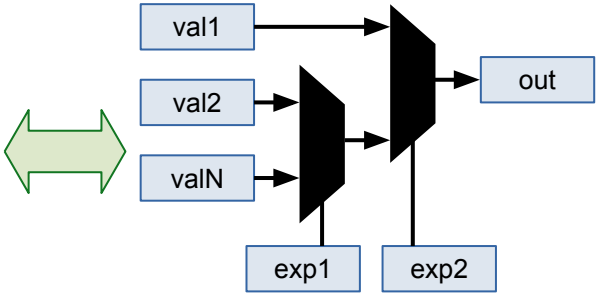
**Sequential statements**

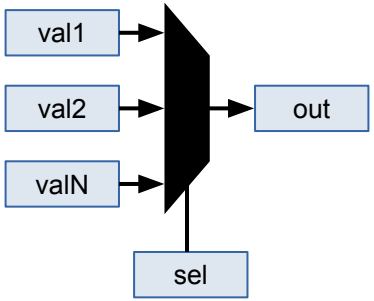
- **if/else**
- **case/when**
- **for/loop**
- **while/loop**
- **loop**

Advanced

# Concurrent statements

```
out <=
    val1 when exp1 else
    val2 when exp2 else
    ...
    valN; -- default
```




```
with sel select out <=
    val1 when op1
    val2 when op2
    ...
    valN when others;
```

⚠ Priorities and differents propagation times involved.

- **valX** can be a value, signal or expression.
- **expX** must be a boolean expression.
- **selX** can ve a signal or expression.
- **opX** are different values of sel.

# Processes

```
architecture MyArch of MyEntity is
   -- declarations
begin
   ...
   label : process (sensitivity list)
   begin
      Sequential statement;
      Sequential statement;
      Sequential statement;
   end process label;
   ...
end architecture MyArch;
```

Opt

Opt

Opt

- Is a circuit part which can be active or inactive.
- A process activates when a signal in the sensitivity list changes its value.
- All the process blocks are executed in parallel (concurrent statements).
- Sequential statements allow us to describe the abstract behaviour of a circuit rather than using low-level components (easier for humans).

Sequential statements are sequentially evaluated (not executed as in a processor).

Inside a process, a signal can be assigned multiple times, but only the last assignment takes effect.

# Variables

```
label : process (a, b)
    -- variable name: type [:=default];
    variable tmp0, tmp1, tmp2 : std_logic;
begin
    -- variable_name := expression;
    tmp0 := '0';
    tmp1 := tmp0 or a;
    tmp2 := tmp1 or b;
    y_o  <= tmp2;
end process label;
```

- Are similar but different than a signal.
- Are declared and visible inside a process.
- Its value changes without delay involved.
- Are assigned with **:=** instead of **<=**.

The VHDL variables are similar to a programming language variable because you can assign them in a line and read its updated value in the following one. It doesn't happen with a signal.

You can use a variable to produce the same hardware than a signal, but I recommend you to use them only to store intermediate values.

# Sequential statements

```
label : process (...)
   -- declarations;
begin
   if exp1 then
      -- sentences
   elsif exp2 then          -- Opt
      -- sentences
   else                     -- Opt
      -- default
      -- sentences
   end if;
end process label;
```

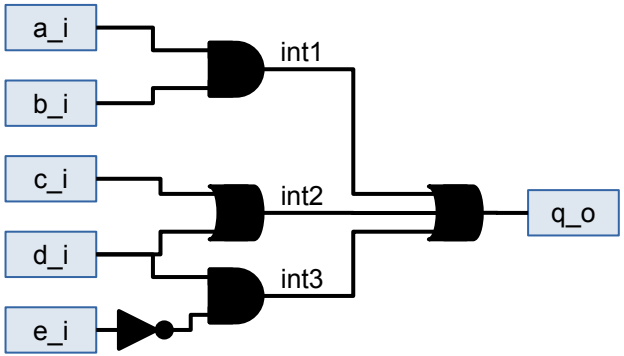
⚠ Be careful with incomplete assignments (memory inference).

```
label : process (...)
   -- declarations;
begin
   case sel is
      when op1 =>
         -- sentences
      when op2 to op5 =>
         -- sentences
      when op6 | op8 | op11 =>
         -- sentences
      when others =>
         -- default
         -- sentences
   end case;
end process label;
```

Are similar and shares features with its concurrent counterpart (**if** and **when/else**, **case** and **select/with**), but allows grouping of statements and can be nested.

# Concurrent circuits (combinational / asynchronous)

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

entity comb is
    port (
        a_i, b_i, c_i, d_i, e_i : in  std_logic;
        q_o : out std_logic
    );
end entity comb;

architecture alt1 of comb is
    signal int1, int2, int3 : std_logic;
begin
    int1 <= a_i and b_i;
    int2 <= c_i or d_i;
    int3 <= d_i and (not e_i);
    q_o <= int1 or int2 or int3;
end architecture alt1;
```




- No internal state (no storage, so only LUTs are inferred)
- The outputs only depends on the inputs

Avoid combinational loops
(a <= a + b;)

# Concurrent circuits (using a process)

```
architecture alt2 of comb is
    signal int1, int2, int3 : std_logic;
begin
    process (
        a_i, b_i, c_i, d_i, e_i,
        int1, int2, int3
    )
    begin
        int1 <= a_i and b_i;
        int2 <= c_i or d_i;
        int3 <= d_i and (not e_i);
        q_o <= int1 or int2 or int3;
    end process;
end architecture alt2;
```

```
-- using variables
architecture alt3 of comb is
begin
    process (a_i, b_i, c_i, d_i, e_i)
        variable int1, int2, int3 :
            std_logic;
    begin
        int1 := a_i and b_i;
        int2 := c_i or d_i;
        int3 := d_i and (not e_i);
        q_o <= int1 or int2 or int3;
    end process;
end architecture alt3;
```

Synthesizers don't check the Sensitivity list. All the inputs must be included to avoid a simulation mismatch!

# Sequential circuits (synchronous)

- They have an internal state (flip-flops, aka registers, are inferred)
- The output depends on the inputs and the internal state
- Depends on a clock (synchronous)

| | Asynchronous | Synchronous |
|---|---|---|
| Speed | **Faster (max)** | **Depends on clock and the arch** |
| Power | **Probably lower** | **Depends on the arch** |
| Area | **Probably higher** | **Depends on the arch** |
| Development time | **Longer** | **Shorter** |
| Debug | **Very difficult** | **Easiest** |
| Reliability | **Need a lot of testing** | **Strong** |

# Clock

```
label : process (clk_i)
begin
    if rising_edge(clk_i) then
        -- do something
    end if;
end process label;
```

**OR**

```
label : process (clk_i)
begin
    if falling_edge(clk_i) then
        -- do something
    end if;
end process label;
```

```
label : process (clk_i)
begin
    if rising_edge(clk_i) or falling_edge(clk_i) then
        -- don't do that!!! The FPGA has only one clock input per FF.
        -- You will be using more are to be slower
    end if;
end process label;
```

# Reset

```
label : process (clk_i)
begin
   if rising_edge(clk_i) then
      if rst_i = '1' then
         -- assign default
         -- values
      else
         -- do something
      end if;
   end if;
end process label;
```

**OR**

```
label : process (clk_i, rst_i)
begin
   if rst_i = '1'  then
      -- assign default
      -- values
   elsif rising_edge(clk_i) then
      -- do something
   end if;
end process label;
```

- With FPGA, you will normally use synchronous reset.
- Asynchronous reset is common in ASIC designs.

Do you really need a reset?

**Additional read:** How do I reset my FPGA?

# Example - Counter (part I)

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

entity cnt12 is
    port (
        clk_i : in  std_logic;
        rst_i : in  std_logic;
        cnt_o : out std_logic_vector(3 downto 0)
    );
end entity cnt12;

architecture RTL of cnt12 is
    constant MODULE : positive := 12;
    signal cnt : unsigned(3 downto 0); -- 16
begin
```

- We will implement a counter module 12 (from 0 to 11).
- A **constant** is employed to avoid a ***magic number*** (good practice).
- There are two reasons to use the signal **cnt** instead of the port **cnt_o** (next slide).

Remember (good practice):
use only **std_logic** and
**std_logic_vector** types for ports.

# Example – Counter (part II)

```
begin
    counter : process (clk_i)
    begin
        if rising_edge(clk_i) then
            if rst_i = '1' then
                cnt <= (others => '0');
            else
                if cnt < MODULE then
                    cnt <= cnt + 1;
                else
                    cnt <= (others => '0');
                end if;
            end if;
        end if;
    end process counter;

    cnt_o <= std_logic_vector(cnt);
end architecture RTL;
```

- Addition (cnt + 1) is not defined for std_logic_vector.
- An output can't be read (cnt <= cnt + 1;). You need an intermediate signal connected to the output.

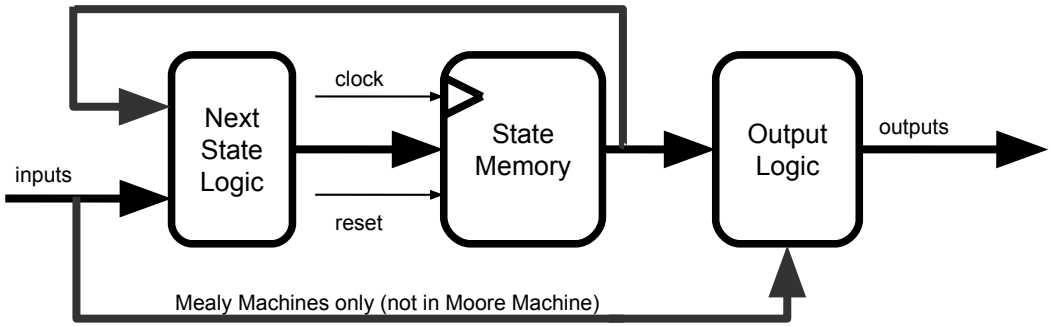
Remember (good practice):
use synchronous reset
(if needed).

# Finite State Machines

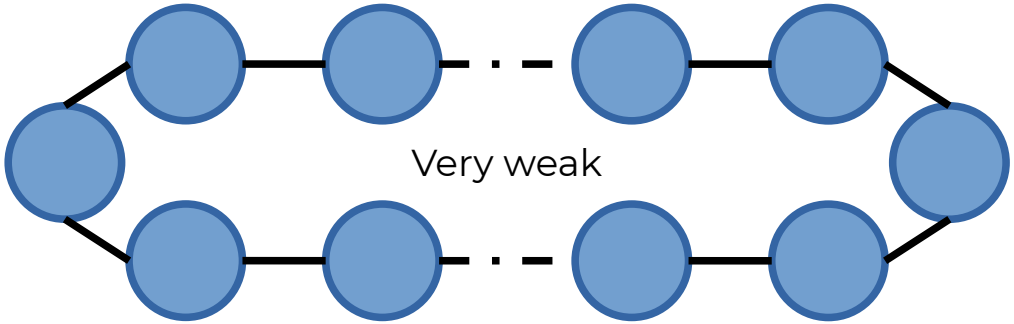
# What is an FSM?

| Asynchronous/Combinational | Synchronous/Sequential |
|---|---|
| No internal state (no memory) | They have an internal state (FFs, registers) |
| Outputs only depends on the inputs | Output depends on inputs/internal state |
| No depends on a clock | Depends on a clock |

- A systematic design technique for sequential circuits, which leads to near/optimal implementations
- Clock-by-clock the machine will be in one of the finite possible states
- The state segmentation helps to detect where there are problems

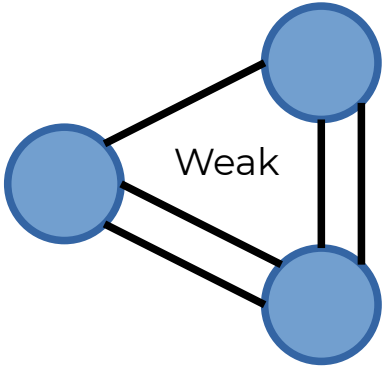


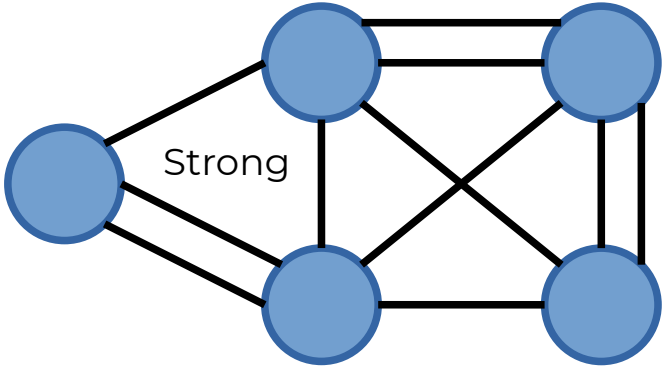
# When to use an FSM?




⚠ You can use a counter instead

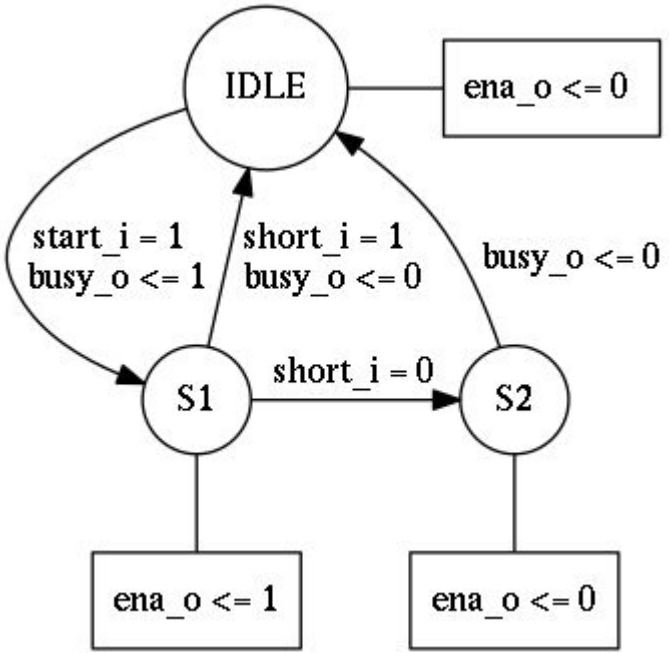
⚠ In case of 2 states you can use an **if** instead



# States diagram

- Graphical representation of the functional specification
- It must include all the possible states.
- All transition conditions that are not unconditional must be specified
- The list of output signals must be the same in all the states



# States

```
architecture FSM of My_Entity is
    type state_type is (IDLE_S, S1_S, S2_S);
    signal state: state_type;
begin
    -- the FSM process here
end architecture FSM;
```

- States in VHDL are defined using enumerations.
- Enumerations in VHDL are defined creating a new **type**.
- FSM are synchronous and modelated with a **case**/**when** statements.
- Each state specify actions and transaction conditions.
- All the states must be specified (use **when others** when needed).

```
my_fsm : process (clk_i) begin
    if rising_edge(clk_i) then
        if rst_i = '1' then
            state <= IDLE_S;
        else
            case state is
                when IDLE_S =>
                    -- ...
                when S1_S =>
                    -- ...
                when S2_S =>
                    -- ...
            end case;
        end if;
    end if;
end process my_fsm;
```

# State encoding

- **Sequential:** conventional binary code ($2^N$ states)
- **One-Hot:** one bit per state (N states)
- **Johnson:** uses a Johnson ring counter (2xN states)
- **Gray:** uses Gray encoding ($2^N$ states)
- **Modified One-Hot:** the bit 0 is inverted to start in reset (N states)
- **User-defined** and **auto** (**default**)

It is normally selected by the Synthesis tool, but
you can specify another one with tool-specific options.

# Transitions

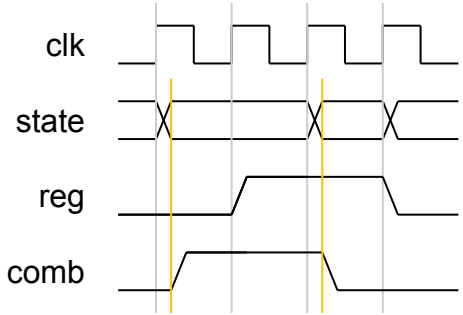
```
when STATE1_S =>
    state <= STATE2_S; -- unconditional
when STATE2_S =>
    if cond1 then
        state <= STATE3_S;
    end if;
when STATE3_S =>
    if cond1 then
        state <= STATE1_S;
    elsif cond2 then
        state <= STATE2_S;
    else
        state <= STATE4_S;
    end if;
```

- The transition to a new state is achieved by assigning the signal that models the state.
- Conditional transitions are modeled with **if**, **elsif**, **else** (be careful with the priorities).
- **condX** could be input ports, signals (internal or external to the FSM, such a counter value), etc.

# Outputs

```
-- Output registered in the process
-- which implements the FSM
when STATE1_S =>
   port1_o <= '1';
when STATE2_S =>
   if cond1 then
      port1_o <= '0';
   end if;
```

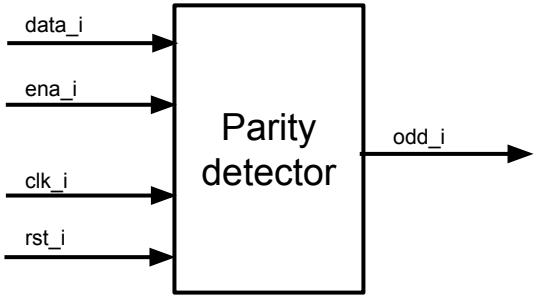
```
-- Output registered in another
-- process
do_assign : process (clk_i)
begin
   if state=STATE2_S then
      port2_o <= '0';
      if cond2 then
         port2_o <= '1';
      end if;
   end if;
end process do_assign;
```

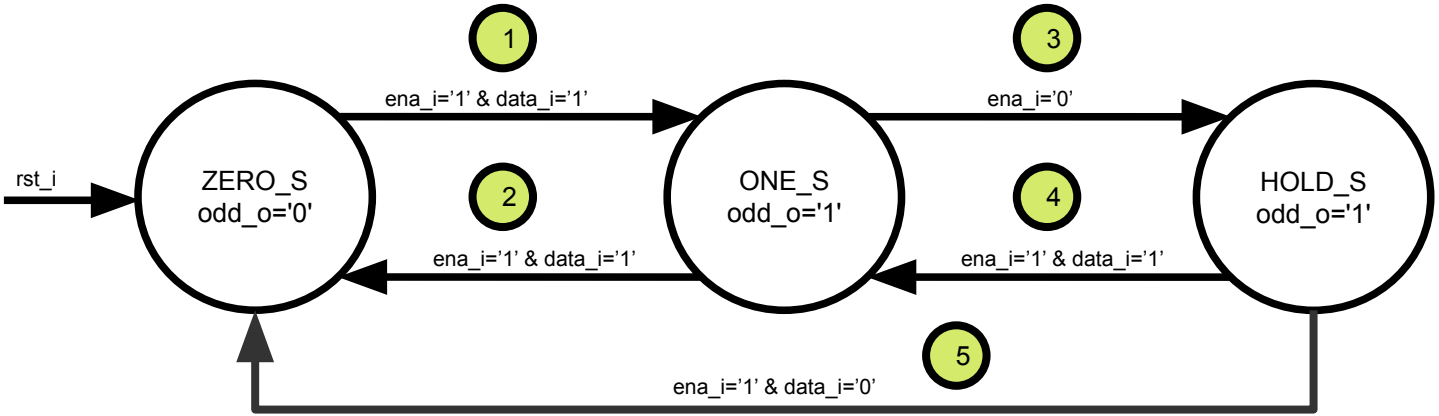



```
-- Combinational assign
port3_o <= '1' when state = STATE3_S else '0';
```

# Example – Parity Detector – Definition

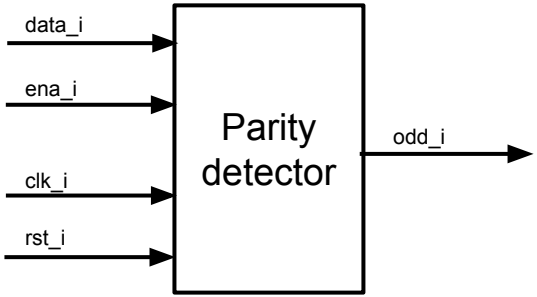
- Serial input data.
- While enabled, parity is observed.
- When enable goes down, the output indicates an even or odd quantity.



# Example - Parity Detector - Entity

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

entity ParityDetector is
    port (
        clk_i  : in  std_logic;
        rst_i  : in  std_logic;
        ena_i  : in  std_logic;
        data_i : in  std_logic;
        odd_o  : out std_logic
    );
end entity ParityDetector;

architecture FSM of ParityDetector is
    type state_type is (ZERO_S, ONE_S, HOLD_S);
    signal state : state_type;
begin
```



# Example – Parity Detector – Architecture

```
do_fsm : process (clk_i)
begin
   if rising_edge(clk_i) then
      if rst_i = '1' then
         state <= ZERO_S;
      else
         -- the case here
      end if;
   end if;
end process do_fsm;

odd_o <= '1' when state/=ZERO_S;

end architecture FSM;
```

```
case state is
   when ZERO_S =>
      if ena_i='1' and data_i='1' then   (1)
         state <= ONE_S;
      end if;
   when ONE_S =>
      if ena_i='1' then                  (2)
         if data_i='1' then
            state <= ZERO_S;
         end if;
      else
         state <= HOLD_S;                (3)
      end if;
   when HOLD_S =>
      if ena_i='1' then
         if data_i='1' then              (4)
            state <= ONE_S;
         else
            state <= ZERO_S;             (5)
         end if;
      end if;
end case;
```

# Considerations for Synthesis

# Signals

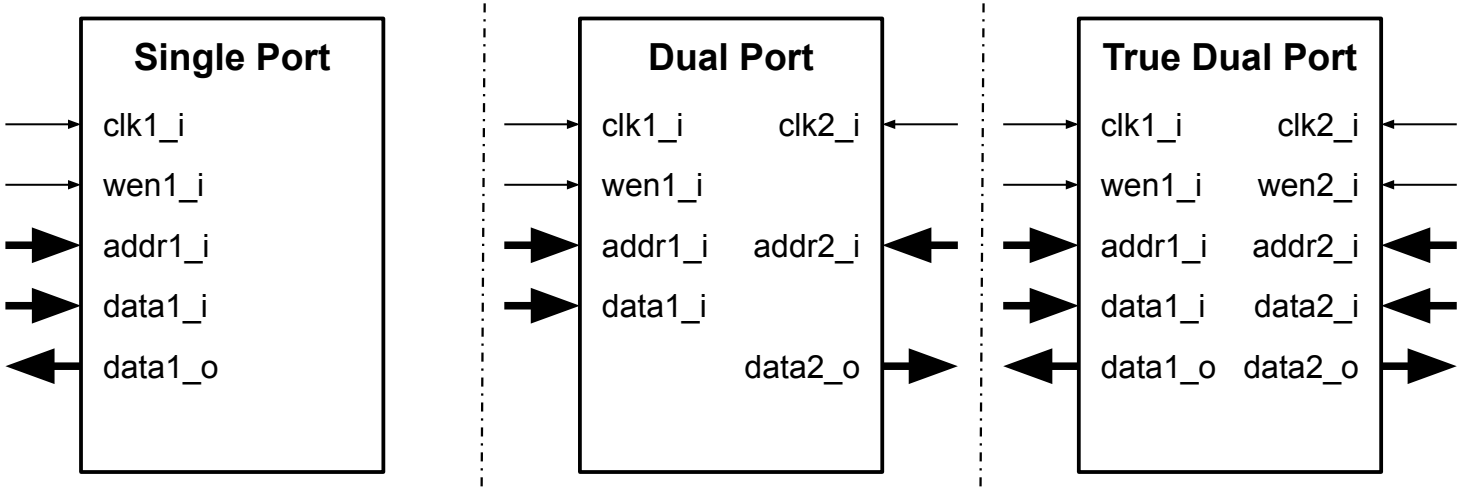
```
architecture my_arch of my_ent is
   signal data : std_logic;
begin
   proc1 : process (clk_i)
   begin
      if rising_edge(clk_i) then
         data <= '0';
      end if;
   end process proc1;

   proc2 : process (clk_i)
   begin
      if rising_edge(clk_i) then
         data <= '1';
      end if;
   end process proc1;
end architecture my_arch;
```

Multiple drivers, can't be synthesized.

```
entity bidir is
   port (
      data_io : inout std_logic;
      data_i  :   in std_logic;
      data_o  :  out std_logic;
      wr_i    :   in std_logic
   );
end entity bidirr;

architecture RTL of bidir is
begin
   data_io <= data_i when wr_i='1' else 'Z';
   data_o  <= data_io;
end architecture RTL;
```

You can use **inout** in the IO blocks of an FPGA, but normally not internally.

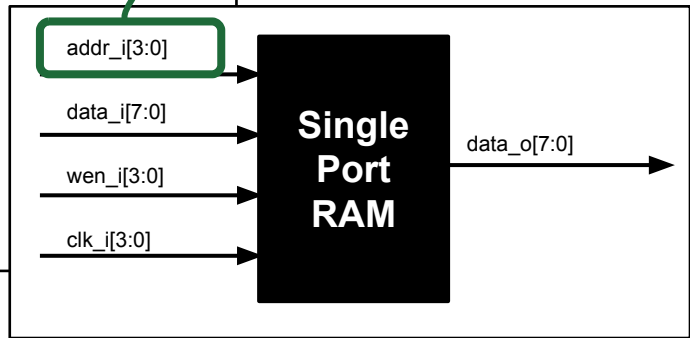
# Memory Inference (I)



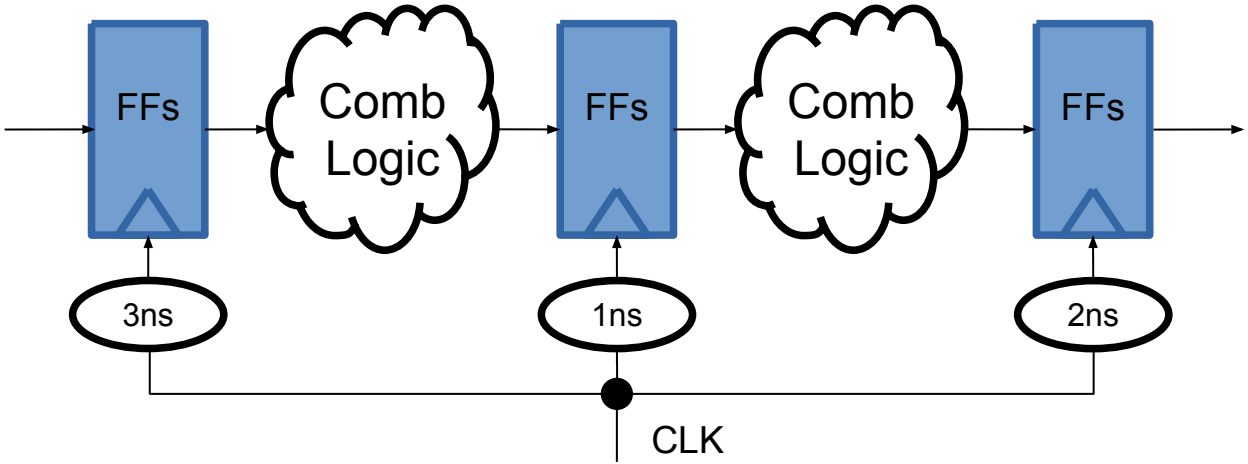

- Modern FPGAs support Single, Dual and True Dual Port RAMs.
- Can be instantiated or inferred.
- The description of an unsupported characteristic produces distributed memory.

# Memory Inference (II)

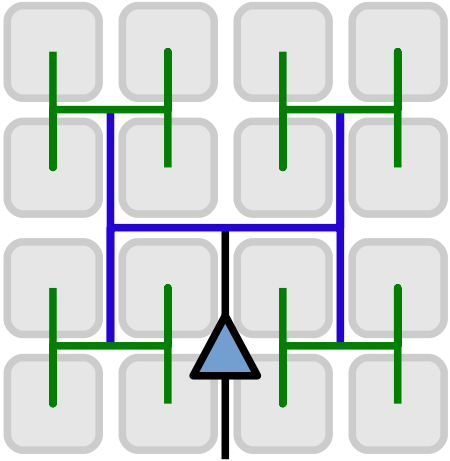
```
architecture Memory of SinglePortRAM is
    type ram_type is array (0 to 15) of std_logic_vector(7 downto 0);
    signal ram : ram_type;
begin

    ram_p: process (clk_i)
    begin
        if rising_edge(clk_i) then
            data_o <= ram(to_integer(unsigned(addr_i)));
            if wen_i='1' then
                ram(to_integer(unsigned(addr_i))) <= data_i;
            end if;
        end if;
    end process ram_p;

    -- data_o <= ram(to_integer(unsigned(addr_i)));

end architecture Memory;
```



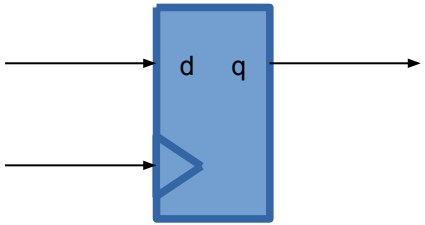
# Clock




- **Clock skew:** the same clock signal arrives different components at different times.
- To reduce this effect, you must use **global buffers**, to employ the **clock tree**.

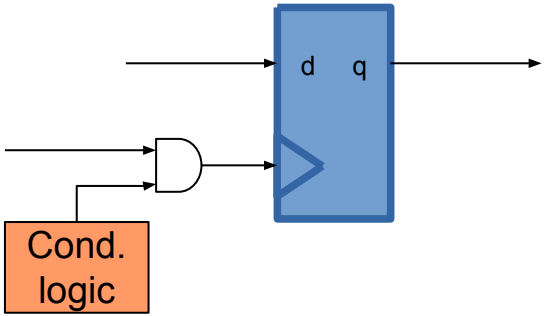


# Clock strategies



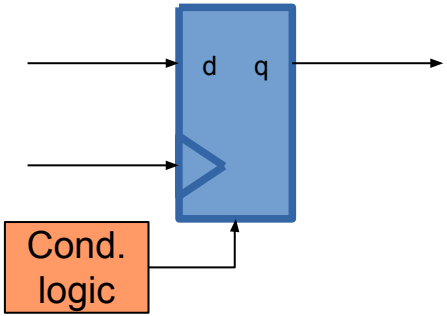

Free running clock

⚠ FPGAs have a predefined clock tree.




Free running clock
+ clock gating

🚫 Don't do that in an FPGA.

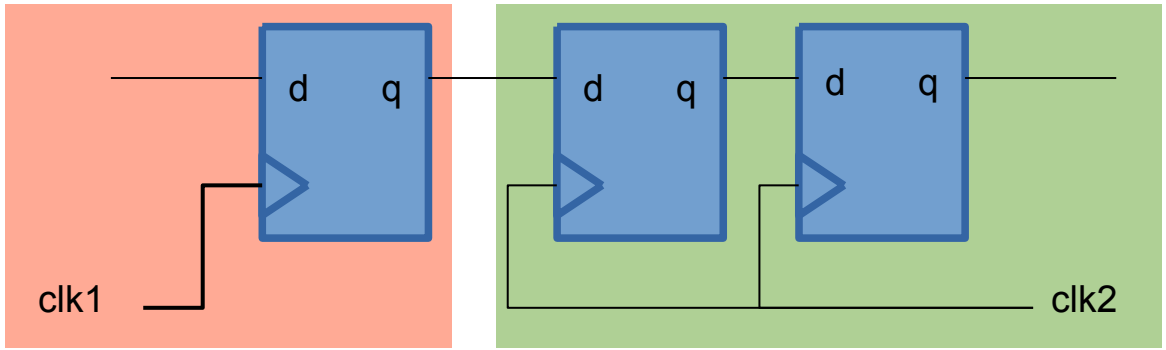



Free running clock
+ clock enable

✓ The FFs of an FPGA have a chip-enable port.

# Clock Domain Crossing

- Metastability can cause system failures in digital devices when a signal is transferred between unrelated or asynchronous clock domains.
- CDC techniques:
  - Chain of FFs
  - Handshake
  - Dual-port RAM
  - Asynchronous FIFO
  - Others (debouncer)




**Additional read:** Understanding Metastability in FPGAs (Altera white paper)

# Latches

```
architecture Latches of my_ent is
begin

    process (ena_i, data_i)
    begin
        if ena_i = '1' then
            latch1_o <= data_i;
        end if;
    end process ram_p;

    latch2_o <=
        "0000" when sel_i = "00" else
        "0011" when sel_i = "01" else
        "1111" when sel_i = "10";

end architecture Latches;
```

- FFs are active by a clock edge, while latches are active by level.
- Latches are created when you have an **incomplete assignment** using a combinational process or a conditional assignment.
- **Latches should never be used in your FPGA design**:
  - They are usually unintentional.
  - They are usually a problem for the FPGA tools (which normally complains about them).

# Advanced VHDL

(parametric and reutilizable code)

# Generics - Declaration

```
entity RAM is
   generic (
      -- NAME: type [:=default];
      AWIDTH : positive := 4;
      DWIDTH : positive := 8;
      DEPTH  : positive := 16
   );
   port (
      clk_i  :  in std_logic;
      addr_i :  in std_logic_vector(AWIDTH-1 downto 0);
      data_i :  in std_logic_vector(DWIDTH-1 downto 0);
      data_o : out std_logic_vector(DWIDTH-1 downto 0);
      wr_i   :  in std_logic
   );
end entity RAM;
architecture Memory of RAM is
   type ram_type is array (0 to DEPTH-1) of std_logic_vector(DWIDTH-1 downto 0);
   signal ram : ram_type;
```

Generics are constant values defined at instantiation time. It allows writing of parametric designs (reusability).

They are commonly **natural**, **positive** or **boolean**. Sometimes (Xilinx), you can found **string** or **real**.

# Generics – Instantiation

```
architecture my_arch of my_ent is
   signal in1, in2, out1, out2 : std_logic_vector(7 downto 0);
   signal addr1, addr2         : std_logic_vector(2 downto 0);
begin
   ram1 : ram
   generic map (AWIDTH => 2, DWIDTH => 8, DEPTH => 4)
   port map (
      clk_i  => clk_i, wen_i => '1',
      addr_i => addr1(1 downto 0), data_i => in1, data_o => out1
   );
   ram2 : ram
   generic map (AWIDTH => 3, DEPTH => 8) -- DWIDTH = 8 (default)
   port map (
      clk_i  => clk_i, wen_i => '1',
      addr_i => addr2, data_i => in2, data_o => out2
   );
end architecture my_arch;
```

# Generates

```
entity my_ent is
    generic (
        ENABLE   : boolean  := true;
        QUANTITY : positive := 2
    );
    port (
        ...
    );
end entity my_ent;
```

```
architecture my_arch of my_ent is
begin
    label_for : for i in 0 to QUANTITY-1 generate
        -- concurrent statements
        -- or instantiation
    end generate label_for;
end architecture my_arch;
```

You must deal with indexes!

```
architecture my_arch of my_ent is
begin
    label_if : if ENABLE generate
        -- concurrent statements
        -- or instantiation
    end generate label_if;

    label_not :
    if not ENABLE generate
        -- there is not an
        -- else generate
    end generate label_not;
end architecture my_arch;
```

# Loops

```
entity vector_inv is
    generic (WIDTH : positive := 4);
    port (
        data_i :  in std_logic_vector(WIDTH-1 downto 0);
        data_o : out std_logic_vector(WIDTH-1 downto 0)
    );
end entity my_ent;

architecture my_arch of my_ent is
begin
    my_for : process (data_i)
    begin
        for i in 0 to WIDTH-1 loop
            data_o[WIDTH-1-i] <= data_i[i];
        end loop;
    end process my_for;
end architecture my_arch;
```

The range must be a CONSTANT value!

You must deal with indexes!

- Useful for iterative HW replication. Be careful! Think on **loop unrolling**.
- The range attribute can be useful (data_i'RANGE).
- You can use **while** and **loop** but uncommon for synthesis.

# Subprograms – functions

```
architecture my_arch of my_ent is
   function bin2gray(arg: unsigned) return unsigned is
      -- declarations (no signals)
   begin
     return shift_right(arg, 1) xor arg;
   end function bin2gray;

   function bin2gray(arg: std_logic_vector) return std_logic_vector is
   begin
     return std_logic_vector(bin2gray(unsigned(arg)));
   end function bin2gray;

   signal aux1 : unsigned(7 downto 0);
   signal aux2 : std_logic_vector(7 downto 0);
begin
   aux1 <= bin2gray(“10101010”);
   aux2 <= bin2gray(“10101010”);
end architecture my_arch;
```

- Only inputs
- Sequentially evaluated
- No time (no signals)
- One output (return)
- Supports overloading

# Subprograms - procedures

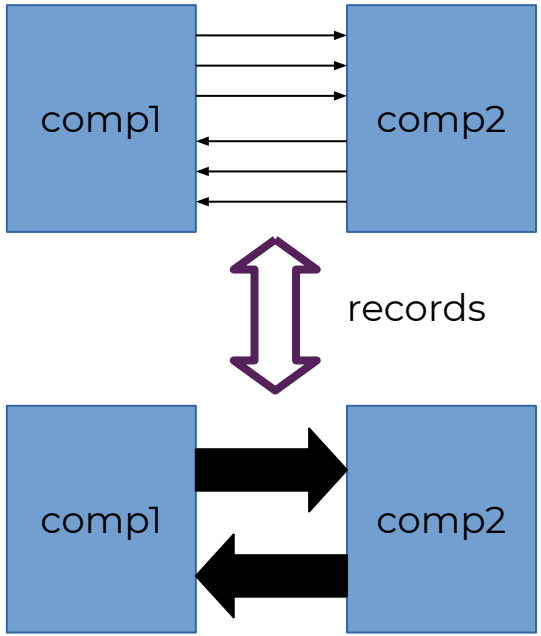
```
architecture my_arch of my_ent is
   procedure example(in1, in2 : in unsigned; q : out unsigned) is
      -- declarations
   begin
     q <= in1 + in2;
   end procedure example;

   signal a, b, c : unsigned(7 downto 0);
begin
   example(in1 => a, in2 => b, q => o);
end architecture my_arch;
```

- Inputs and outputs as argument
- Sequentially evaluated
- Time involved
- Supports overloading

- Functions and Procedures are like C macros (replaced in place).
- For synthesis, you can found simple functions (types conversion or small computations), being the procedures rarely employed (similar to components).

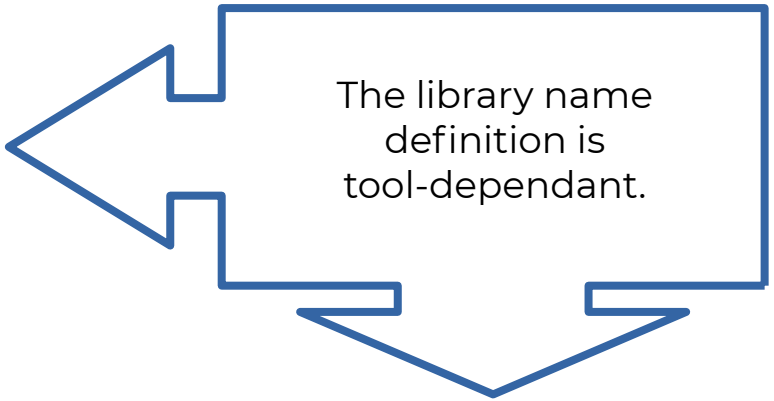
# Records

```
architecture my_arch of my_ent is
    type instruction_t is record
        opcode : std_logic_vector(3 downto 0);
        addr   : std_logic_vector(11 downto 0);
        data   : std_logic_vector(15 downto 0);
    end record instruction_t;
    signal ir : instruction_t;
begin
    ...
    ir.opcode <= “1010”;
    ir.addr   <= X“123”;
    ir.data   <= X“CAFE”;
    ...
    data_o    <= ir.data;
    ir_o      <= ir;
end architecture my_arch;
```



# User defined Libraries and Packages

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

package My_Package is
    -- constants
    -- components declarations
    -- functions declarations
    -- procedures declarations
    -- types, subtypes, records
end entity My_Package;

package body My_Package is
    -- functions implementations
    -- procedures implementations
end package body My_Package;
```




```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
library My_Library;
use My_Library.My_Package.all;
```

# Others

- Attributes:
  - Provides additional information about a **signal** (S'EVENT)) or a **type** (T'RIGHT).
  - There are predefined attributes in the VHDL specification, predefined attributes per tool and can be also user-defined.
  - Allows parametric and more clear code (normally employed in libraries).
- Types:
  - You can define new **types** (such as **std_logic_vector**) and **subtypes** (such as **signed** and **unsigned**).
  - Essential for FSM (enumerations) and memory inference (arrays).
- Configurations: I have never seen FPGA projects using a **configuration** (ASIC?).

# Conclusion

# Final remarks

- What we saw today is enough to develop a small IP core from scratch.
- There are more things to know when you want to understand any VHDL description.
- There are even more to understand about FPGAs and the EDA tools for a complete system integration.
- **Be synchronous and apply good practices!** All will be easier and better.
- If you want to design for ASICs/FPGAs, you should know [System]Verilog and/or VHDL (recommendation)

rodrigomelo9@gmail.com

rodrigoalejandromelo

@rodrigomelo9ok

rodrigomelo9

rodrigomelo9