Hardware Acceleration Through FPGAs

2. Basics of VHDL

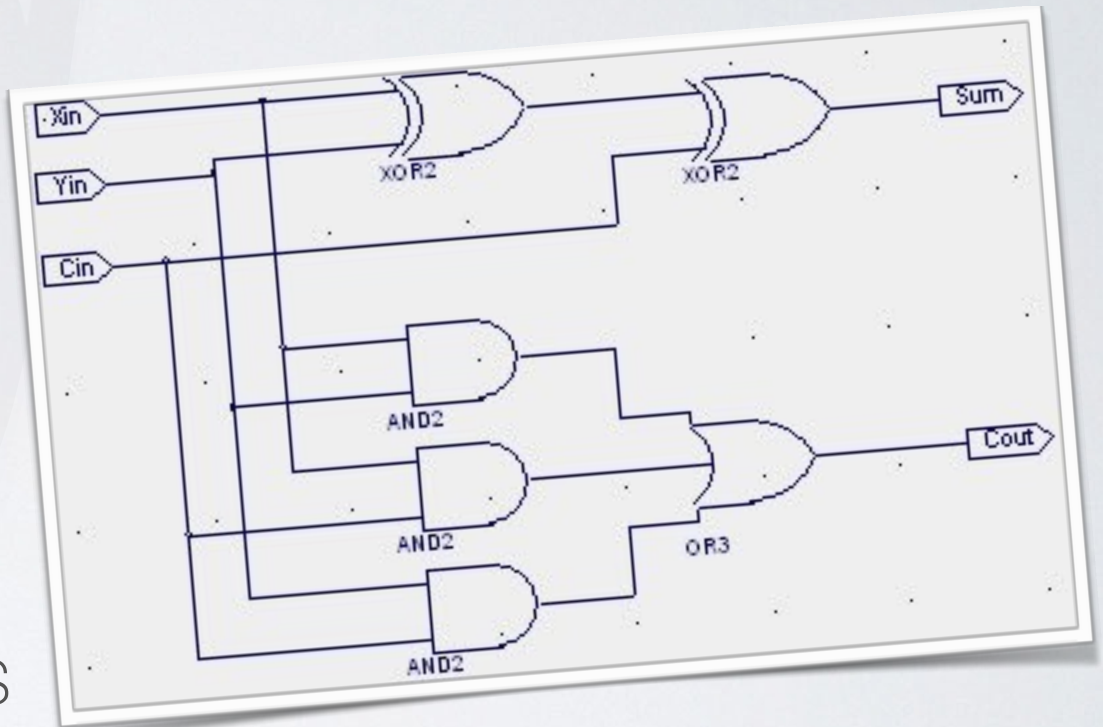
Giorgio Lopez
CERN TE/EPC/CCE

Summary

- FPGA-based Design Techniques
- VHDL and Design Styles
- Anatomy of an HDL Project / SoC Project
- Importance of Simulation and Testbenches
- Special Signals: Clocks and Resets
- VHDL Signal Types
- Basic Constructs of VHDL

FPGA-based Design Techniques: Schematic Entry

To describe user logic in an FPGA it is possible to manually draw the building blocks (multiplexers, logic gates, counters, etc.) and their connections by using a GUI.



Several disadvantages:

- lack of portability across platforms
- lack of maintainability
- hard to handle for large and complex projects

Not very used anymore, support is dropping.

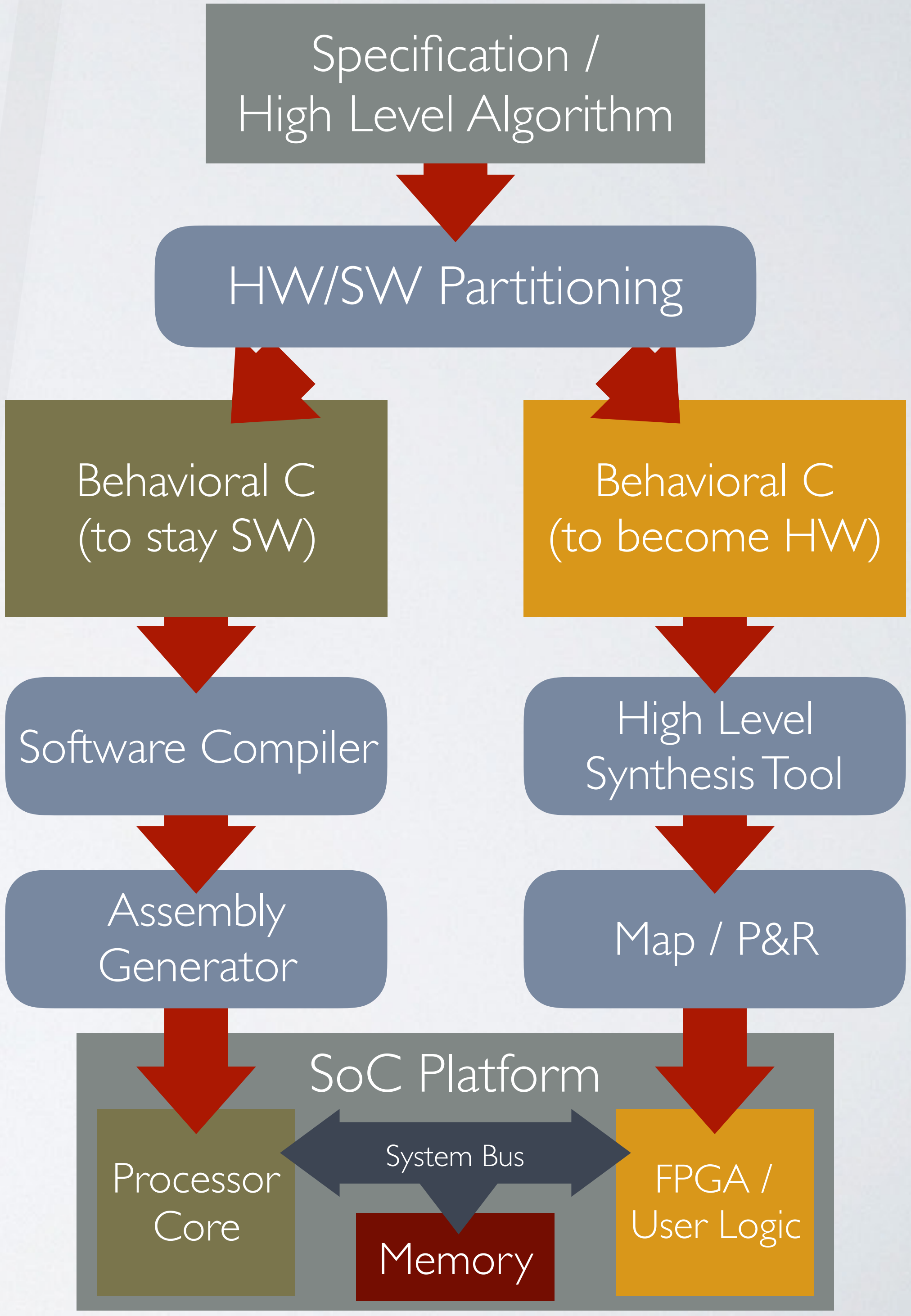
FPGA-based Design Techniques: HDL Languages

- Hardware Description Languages (HDL) enable a formal description of the behavior and/or structure of a digital circuit.
- More scalable, can be managed with versioning systems
- Most common examples: VHDL or Verilog

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

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

FPGA-based Design Techniques: High Level Synthesis

- HLS tools enable automatic translation of blocks from a high level language (such as C/C++) into an HDL
- Typical use is in HW/SW partitioned architectures
- For the implementation to be effective, though, a deep knowledge of the tools and a proper constraining of the translation is needed



VHDL Design Styles

VHDL can be written according to several basic styles, depending on the used constructs and the way logic is described. Main styles are:

- Structural VHDL
- Dataflow VHDL
- Behavioral VHDL

VHDL Design Styles: Structural VHDL

- Structural VHDL describes the structure (as in, the components that are visible in a structure). The visible components are instantiated in the declarative part of the architecture body.

```
architecture structural of mux4to1 is
    component and3
    port( in1,in2,in3 :in std_logic;
          out :in std_logic);
    end component and3;
    component or4
    port( in1,in2,in3,in4 :in std_logic;
          out :in std_logic);
    end component or4;
begin
   A0 : and3 port map( in1 => NOT s0,
                       in2 => NOT s1,
                       in3 => in0
                       out => out0);
   A1 : and3 port map( in1 => s0,
                       in2 => NOT s1,
                       in3 => in1
                       out => out1);
   A2 : and3 port map( in1 => NOT s0,
                       in2 => s1,
                       in3 => in2
                       out => out2);
   A3 : and3 port map( in1 => s0,
                       in2 => s1,
                       in3 => in3
                       out => out3);
   OUT : or4 port map( in1 => out0,
                       in2 => out1,
                       in3 => out2,
                       in4 => out3,
                       out => muxout);
```

VHDL Design Styles: Dataflow VHDL

- In Dataflow VHDL the boolean or arithmetic transformation applied to data are explicitly described with signal assignments

- Keep in mind: all statements are concurrent!

```
...
architecture dataflow of mux4to1 is

begin

   muxout <= out0 OR out1 OR out2 OR out3;

   out0 <= in0 AND NOT s0 AND NOT s1;
   out1 <= in1 AND     s0 AND NOT s1;
   out2 <= in2 AND NOT s0 AND     s1;
   out3 <= in3 AND     s0 AND     s1;
```

VHDL Design Styles: Behavioral VHDL

- Behavioral VHDL describes the operation of the digital circuit with processes where concurrent statements are elaborated in a sequential way with the control flow constructs of traditional programming languages (if..else.., case.., etc)

```
...
architecture behavioral of mux4to1 is

begin

    process(S, A0, A1, A2, A3)
    begin
        case S is
            when "00"   => muxout <= A0;
            when "01"   => muxout <= A1;
            when "10"   => muxout <= A2;
            when others => muxout <= A3;
        end case;
    end process;
```

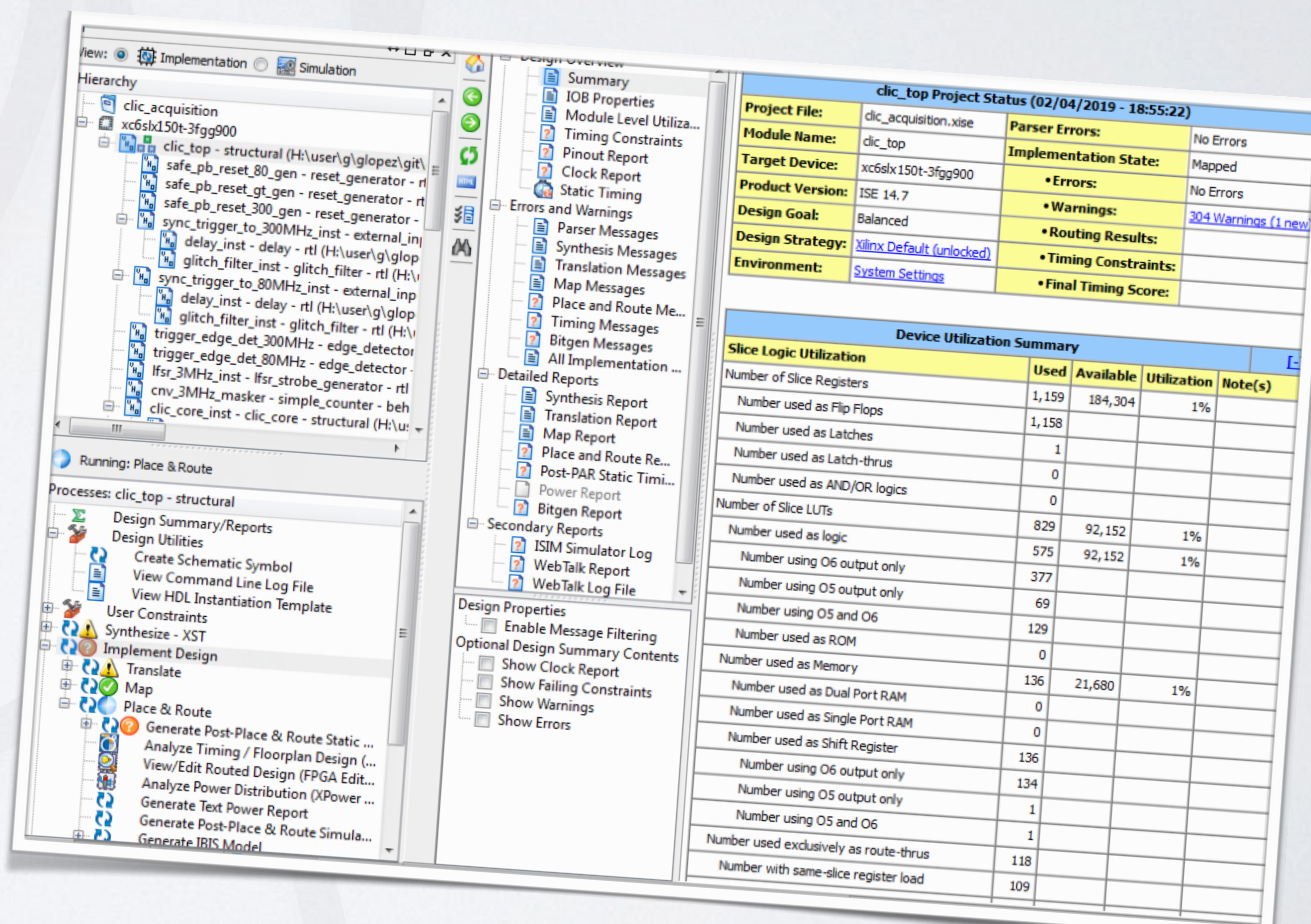
VHDL Design Styles Recap

- Dataflow can be used for simple units and/or where a higher visibility of the logical connections between signals is desirable
- Structural is mainly useful when only an interconnection of other building blocks is to be put in place (e.g: high hierarchical level modules)
- Behavioral better describes more complex control flows (e.g: Finite State Machines)
- Nonetheless, mixed approaches can be used!

Anatomy of an HDL Project

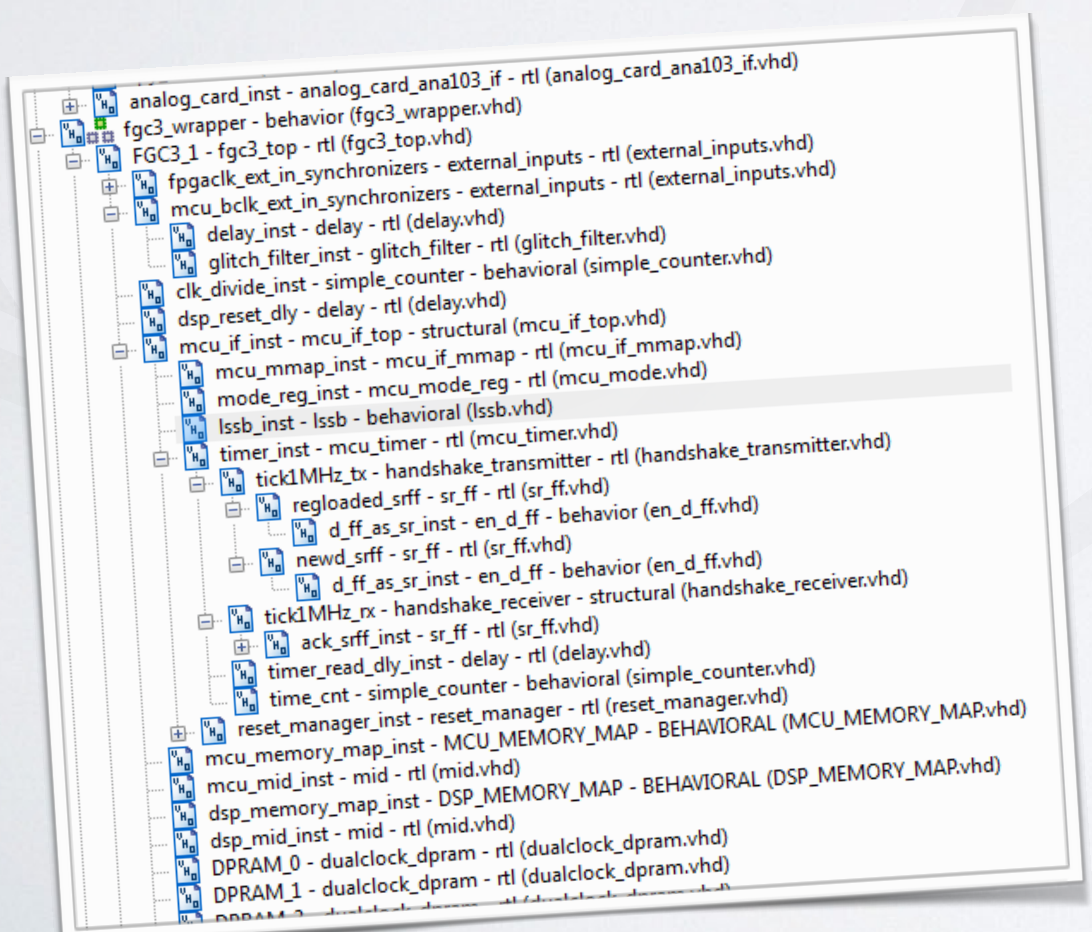
Elements that compose an HDL Project are:

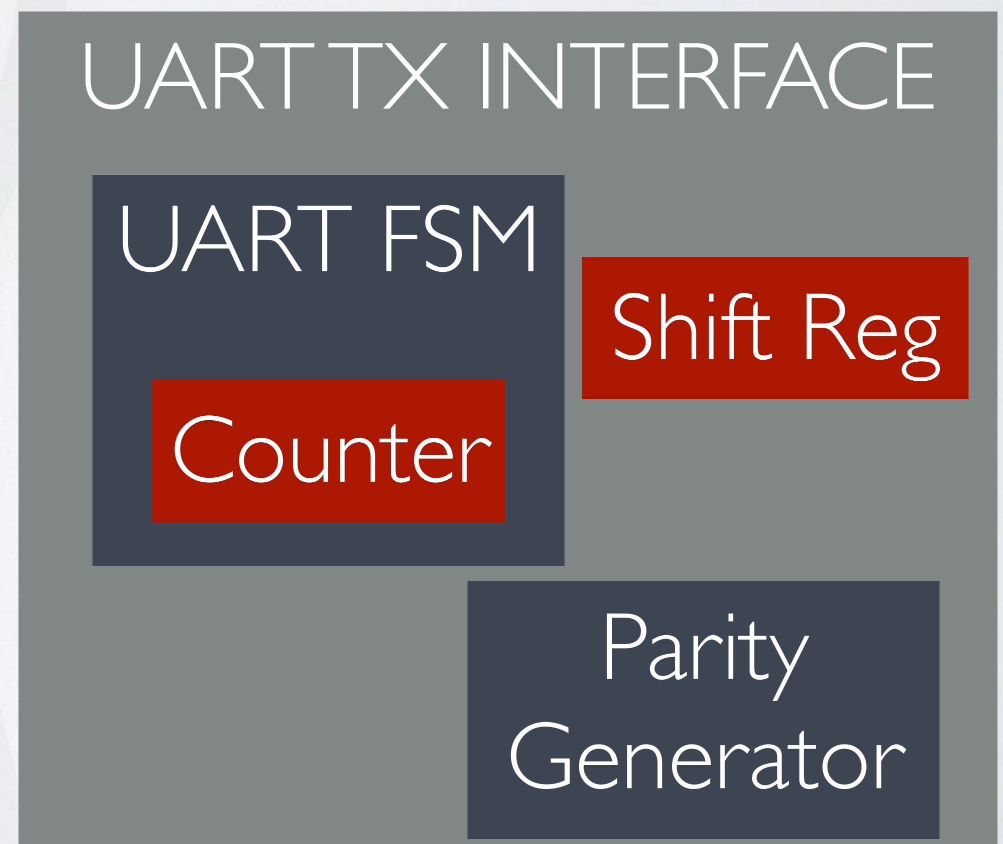
- Modules Hierarchy
- Top Level Entity
- Design Constraints



Modules Hierarchy

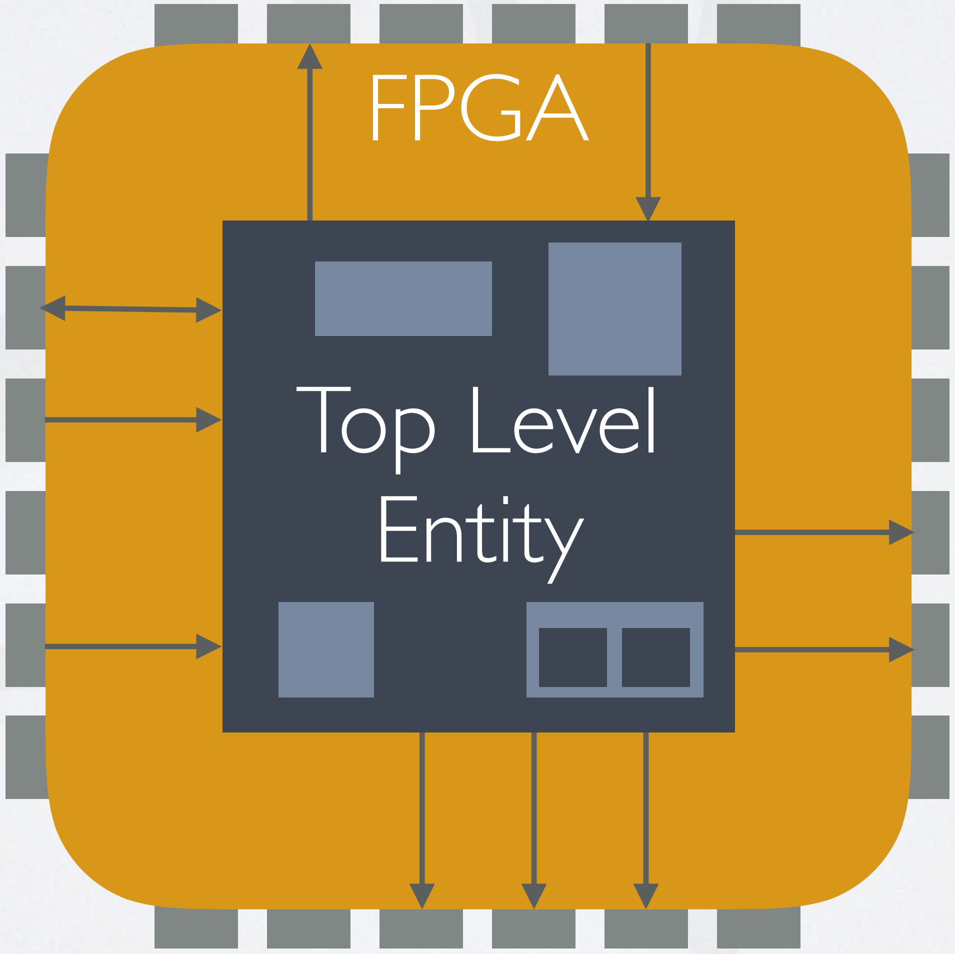
- The VHDL modules in a project compose a "tree" of nested entities which, all together, implement the user logic.





Top Level Entity

- This is the module that holds all connection with the "external world". It wraps all the logic and its inputs and outputs correspond to the physical IO blocks of the FPGA.



Timing Constraints

All complementary information which is needed for the design to be implemented on the device according to the timing requirements is put in the timing constraints. These may include, for example:

- External and derived clock period
- Timing relationships between externally fed signals
- Paths to be ignored in the timing analysis
- Multi - cycle signals

And so on…

```
# PROCESSOR CLOCK
NET MCU_BCLK_INT KEEP;
NET MCU_BCLK_INT TNM = TN_MCUBCLK;
TIMESPEC TS_MCUBCLK = PERIOD TN_MCUBCLK 40ns;
NET DSP_EM_CLK_INT KEEP;
NET DSP_EM_CLK_INT TNM = TN_DSPEMCLK;
TIMESPEC TS_DSPEMCLK = PERIOD TN_DSPEMCLK 10ns;

# ADC SERIAL CLK
NET FGC3_1/analog_card_inst/ADC_SCK_FB KEEP;
NET FGC3_1/analog_card_inst/ADC_SCK_FB TNM =
TN_ADC_SCK_RB;
TIMESPEC TS_ADC_SCK_RB = PERIOD TN_ADC_SCK_RB 20ns;

NET FGC3_1/CLK_32MHZ KEEP;
NET FGC3_1/CLK_32MHZ TNM = TN_CLOCK_32MHZ;
TIMESPEC TS_CLOCK_32MHZ = PERIOD TN_CLOCK_32MHZ
31.25ns;
[...]

#IGNORE CROSS DOMAIN PATHS BETWEEN CLOCKS
NET "MCU_BCLK_INT" TNM_NET = mcu_grp;
NET "DSP_EM_CLK_INT" TNM_NET = dsp_grp;
NET "FGC3_1/fpgaclock2" TNM_NET = fpga_grp; #GIO
#NET "fpgaclock2" TNM_NET = fpga_grp;  #PHIL
NET "FGC3_1/CLK_2MHZ" TNM_NET = 2m_grp;
NET "FGC3_1/CLK_5MHZ" TNM_NET = 5m_grp;
[...]

TIMESPEC TS_01 = FROM "mcu_grp" TO "fpga_grp" TIG;
TIMESPEC TS_02 = FROM "mcu_grp" TO "2m_grp" TIG;
TIMESPEC TS_03 = FROM "mcu_grp" TO "5m_grp" TIG;
TIMESPEC TS_04 = FROM "mcu_grp" TO "16m_grp" TIG;
[...]
```

I/O Constraints

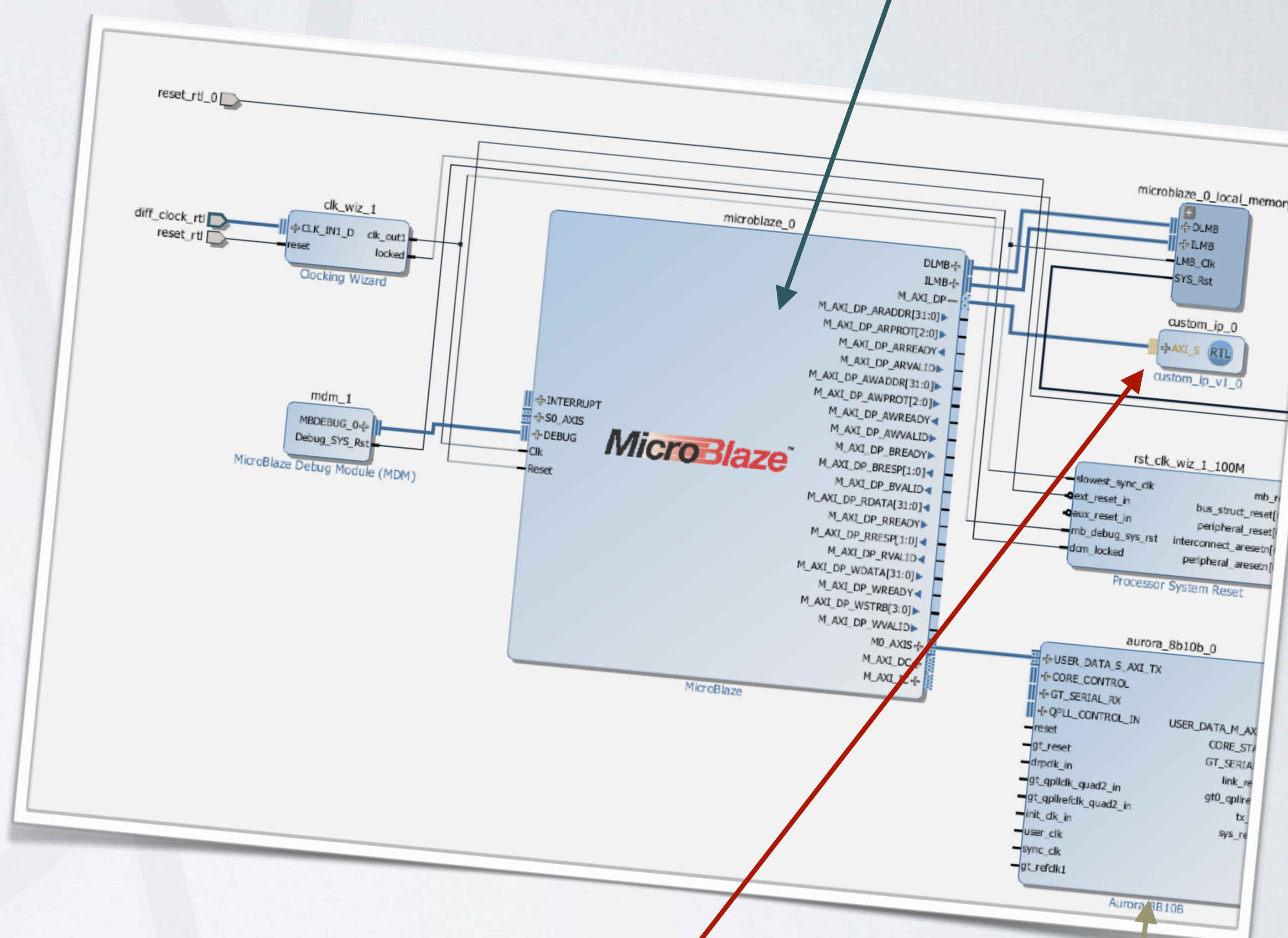
- These constraints are mainly needed to bind signals to physical locations on the FPGA
- A typical example are I/O Location Constraints, which assign top level entity ports to physical IO blocks on the FPGA
- They can also specify other characteristics of the IO signals, e.g. drive strength, type of electric termination, digital voltage standard

```
NET "NotLED_VS_BLUE"        LOC = P2   | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "NotLED_VS_GREEN"       LOC = K5   | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "NotLED_VS_RED"         LOC = P1   | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "C62_DIN<0>"            LOC = F19  | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "C62_DIN<1>"            LOC = F20  | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "C62_DIN<2>"            LOC = F18  | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "C62_DIN<3>"            LOC = E20  | IOSTANDARD = LVCMOS33  | SLEW = SLOW | DRIVE =  8 ;
NET "DSP_EM_CKE"            LOC = B11  | IOSTANDARD = LVCMOS33  | PULLUP ;
NET "DSP_EM_CLK"            LOC = A11  | IOSTANDARD = LVCMOS33  | PULLUP ;
NET "DSP_EM_CS0"            LOC = G7   | IOSTANDARD = LVCMOS33  | PULLUP ;
```

Creating a SoC Project

- EDA tools typically offer graphical interfaces to create SoC projects based on the use of IP blocks provided either by the tool vendor or by third parties

- Custom created IP blocks (generated from RTL code) can then be added and integrated by using standard interfaces



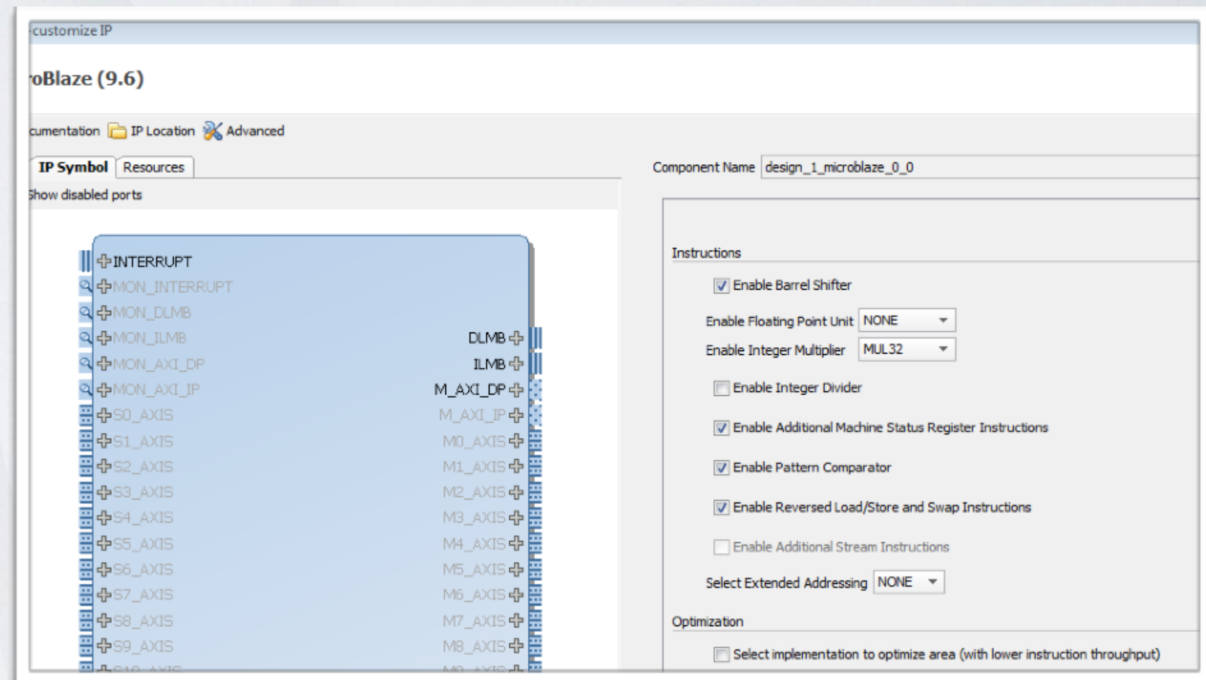
"Soft" CPU core

Custom block from RTL VHDL code

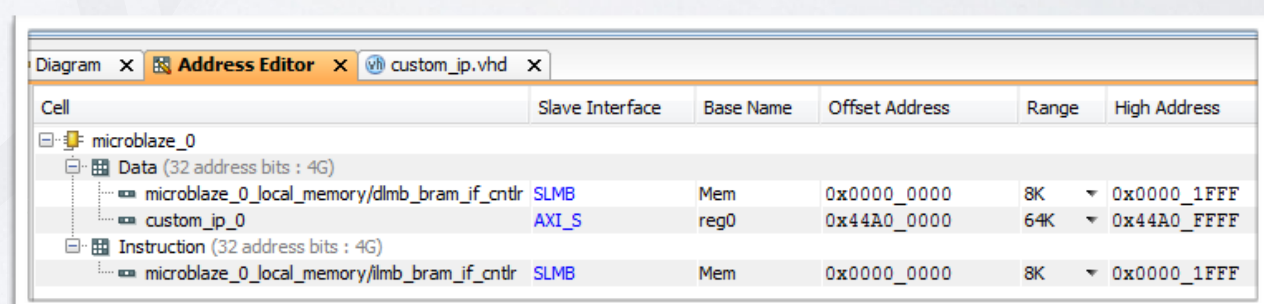
IP block by tool vendor

Creating a SoC Project

- IP blocks can be typically customized in several aspects to match the needs of the designer (i.e: CPU cores can be made more performing and feature rich or lower in resource footprint)



- The tools allow easy generation of the memory map helping the development of drivers to access the peripherals from the CPU at the high abstraction level of the C/C++ code or the OS



Structure of a VHDL Module

A VHDL module is characterized by:

- Used libraries declarations
- Entity declaration
- Ports list (input output inout etc.)
- Architecture head and body (entity implementation)
- Other features (generics, multiple architectures, etc.)

```
library ieee;
use ieee.std_logic_1164.all;

entity toplevel is
    port (clk : in  std_logic;
          rst : in  std_logic;
          d   : in  std_logic;
          q   : out std_logic);
end toplevel;

architecture rtl of toplevel is
    signal intern : unsigned(2 downto 0);
begin

    process(clk, rst)
        begin
...
```

HDL is not programming!

A very important thing to remember (if not the most).

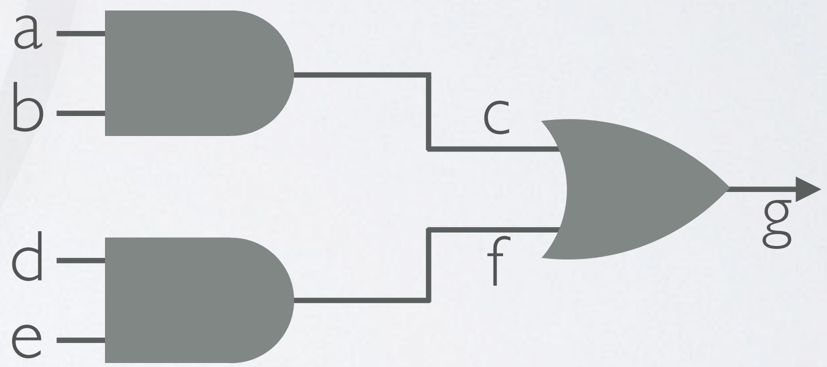
Typical gotcha:

```
c <= a and b;
f <= d and e;
g <= c or f;
```

and

```
g <= c or f;
c <= a and b;
f <= d and e;
```

yield the same result!



This is because HDL is describing logic circuits, not operations!

HDL STATEMENTS ARE CONCURRENT! THINK HARDWARE!

Non-Synthesizable VHDL

- Some constructs in VHDL make code non synthesizable.
- A remarkable case: delays

```
r_Enable <= '0';
wait for 100 ns;
r_Enable <= '1';
```

- Another one: loops. Loops in synthesizable VHDL are only use to replicate logic. For other purposes, use sequential processes
- Nevertheless, non-synthesizable VHDL is ok for testbenches…
- It's important to know what is synthesizable and what is not

The importance of simulation

- One of the biggest problems in dealing with FPGAs is design validation.
- Once the design is deployed on the device access to signals is very limited
    - Only the top level ports
    - Test points? Maybe, but how many?
    - Internal Logic Analyzers (they occupy resources, modify the routed design and can break timing)
- Finding bugs can be **VERY** frustrating and time consuming this way
- Only solution is to use simulation as an integrated part of the design flow

Testbench design

- Testbench design becomes a very important phase of the design flow (comparable to the development of the actual logic)
- A sophisticated testbench can (and should) include features such as:
    - Input randomization (or constrained randomization)
    - Assertions for automatic output validation
    - File I/O (e.g: to read input vector)
    - Code Coverage verification
    - Bus transaction modeling
- There are languages and methodologies which are specifically developed with verification in mind (see SystemVerilog, UVM)…and they are more complex than VHDL itself!

Special Signals Handling: Clock

Special care must be taken with clock for multiple reasons:

- Clock is fed to an enormous number of Flip Flops: high fanout / need for careful buffering
- Skew must be kept low (balanced clock trees)

All this is taken care by the tools almost transparently. But still:

- Remember to constraint accurately
- Avoid “playing” with clock (e.g: don’t use gating, there are other ways to obtain the same effect)
- Typically, there can be more than a single clock in a design: watch out for domain boundaries!

Special Signals Handling: Reset - 1

Reset can be either synchronous or asynchronous:

- Synchronous reset takes effect on next clock edge and is treated by synthesizer as any other synchronous signal (timing closure takes care of correctness of reset propagation)
- Asynchronous reset is instantaneous and takes effect regardless of the presence of clock edges

Special Signals Handling: Reset - 2

Since asynchronous resets are not handled by timing closure, special care must be taken with their use for multiple reasons:

- Asynchronous resets should be kept active for a sufficient time to make sure they propagate correctly to all circuits
- De-assertion of an asynchronous reset should be simultaneous across device to avoid state to progress in some areas while some other are still being kept reset
- Typical choice is to use some additional logic to de-assert the asynchronous reset **synchronously** with clock edge

Signal Types in VHDL - 1

- std_logic : represents a single bit of information. It can be “0” or “1” but also hold other states: the most common are “X” for unknown, “U” for unresolved and “Z” for high impedance (they are typically useful when simulating).

```
signal flag : std_logic;

flag <= ‘0’;
```

Signal Types in VHDL - 2

- std_logic_vector : represents an array of std_logic and can be used for buses.

```
signal data_bus : std_logic_vector(7 downto 0);

data_bus <= “01001001”;

data_bus <= x”FA”;

data_bus(1) <= ‘1’;

data_bus(3 downto 0) <= “0101”;
```

Signal Types in VHDL - 3

When dealing with arithmetic operations, it’s most convenient to use the Unsigned and Signed types (according to the type of data/operation)

- Part of ieee.numeric_std package
- Sign extension is taken care of automatically
- Anyway VHDL will whine if you don’t

Signal Types in VHDL - 4

- Arrays are custom types that can typically be used to represent blocks of memory

```
type memory16x32 is array(0 to 15) of \
                     std_logic_vector(31 downto 0);

signal memblock : memory16x32;

memblock(12) <= x“5A5A”;
```

Signal Casting in VHDL

- VHDL is strongly typed: if assignments have different signal types on the two sides the synthesizer will issue an error.

- Casting from one type to another is necessary

```
signal data_bus : std_logic_vector(3 downto 0);
signal operand : unsigned(3 downto 0);

operand <= unsigned(data_bus);
```

- Recommendation: use std_logic/std_logic_vector for entity ports

Basic Constructs of VHDL

- Signals and Assignations
- Processes and Variables
- When .. else statement
- Case statement

Signals and Assignations

- Assignations describe the physical connections between signals

- They can be simple wires or describe more complex structures like logic gates or multiplexers/LUTs

```
x_test <= test_in;

z <= a OR (b AND c) OR d;


muxout <= in0 when s = ‘0’ else
          in1;

lut_q <= “1000” when s = “100” else
         “0100” when s = “011” else
         “0010” when s = “010” else
         “0001” when s = “001” else
         “0000”;
```

Processes and Variables

- Processes are used to describe in a higher level of abstraction combinatorial or sequential logic. They are activated when a state change happens on any of the signals in their “sensitivity list”:

- *Sequential processes: only clock signal (and asynchronous sets/resets, if present)*

```
process(clk, rst)
begin
    if rst = '1' then
        q <= '0';
    elsif rising_edge(clk) then
        q <= d;
    end if;
end process;
```

- *Combinatorial processes: all signals which appear on the right hand side of an assignment*

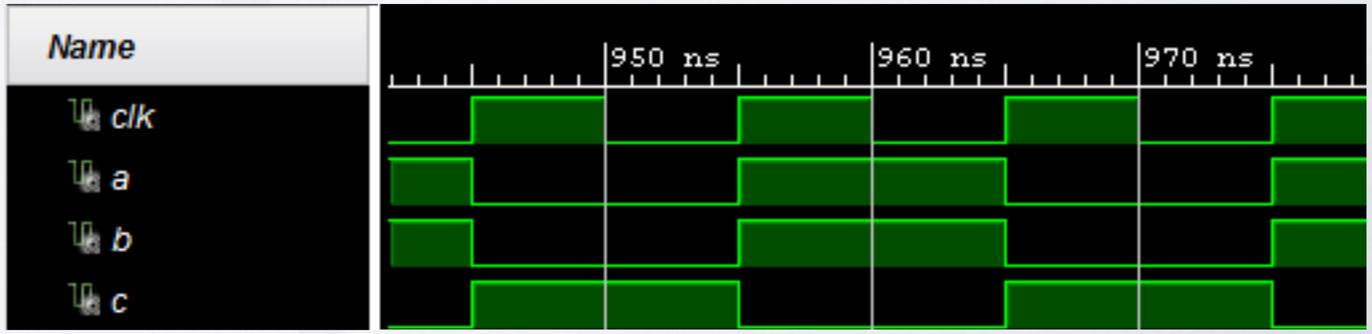
```
process(S, A0, A1, A2, A3)
begin
    case S is
        when "00"   => muxout <= A0;
        when "01"   => muxout <= A1;
        when "10"   => muxout <= A2;
        when others => muxout <= A3;
    end case;
end process;
```

Again, the classic HDL trap!

- Classic programming statements can be used in processes, but this is NOT programming: the statements, in order, are all scheduled to happen at the end of the process

- In this process, the first assignment never happens, and in the third and fourth assignments “c” and “a” get assigned the “old” (before the clock edge) values of “a” and “c” respectively

```
process(clk) is
begin
    if rising_edge(clk) then
        a <= b;
        b <= c;
        c <= a;
        a <= c;
    end if;
end process;
```



Case statement

- Used in processes
- Similar in structure and syntax as in programming languages
- Typically used for Finite State Machines
- It's very important to specify values of outputs for all cases (doing the opposite may result in latches).

```
-- Default output assignments
adc_cnv_o               <= '0';
adc_clk_tick_timer_en   <= '0';
adc_clk_tick_timer_pre  <= '1';
acq_msb_timer_en        <= '0';
cnv_h_timer_en          <= '0';
data_en_o               <= '0';
case cur_state is
    when IDLE =>
        if en_i = '1' then
            next_state <= CNV_H;
        end if;
    when CNV_H =>
        if cnv_h_elapsed = ‘1' then
            next_state <= CNV_L;
        end if;
        adc_cnv_o        <= '1';
        acq_msb_timer_en <= '1';
        cnv_h_timer_en   <= '1';
    when CNV_L =>
        if acq_msb_elapsed = '1' then
            next_state <= CLK_TOGGLE;
        end if;
        acq_msb_timer_en <= '1';
    when CLK_TOGGLE =>
        if clk_pulse_train_over = '1' then
            next_state <= READY;
        end if;
        adc_clk_tick_timer_en  <= '1';
        adc_clk_tick_timer_pre <= '0';
    when READY =>
        if en_i = '1' then
            next_state <= CNV_H;
        end if;
        data_en_o <= '1';
    when OTHERS =>
        next_state <= IDLE;
end case;
```

A trap to avoid: incomplete case (or if) statements

- If we remove the default assignments to outputs, the synthesizer won't know what to do with outputs that are unspecified for given cases
- They will keep their value, generating a register in a sequential process
- In a combinatorial process, though, this creates unwanted "latches" (which are bad practice for FPGA design, in general).

```
case cur_state is
    when IDLE =>
        if en_i = '1' then
            next_state <= CNV_H;
        end if;
    when CNV_H =>
        if cnv_h_elapsed = '1' then
            next_state <= CNV_L;
        end if;
        adc_cnv_o        <= '1';
        acq_msb_timer_en <= '1';
        cnv_h_timer_en   <= '1';
    when CNV_L =>
        if acq_msb_elapsed = '1' then
            next_state <= CLK_TOGGLE;
        end if;
        acq_msb_timer_en <= '1';
    when CLK_TOGGLE =>
        if clk_pulse_train_over = '1' then
            next_state <= READY;
        end if;
        adc_clk_tick_timer_en  <= '1';
        adc_clk_tick_timer_pre <= '0';
    when READY =>
        if en_i = '1' then
            next_state <= CNV_H;
        end if;
        data_en_o <= '1';
    when OTHERS =>
        next_state <= IDLE;
end case;
```

Take Home Messages

- HDLs are not programming languages. They are a tool to describe digital logic circuits.
- Watch Out for Non-Synthesizable Code and for the traps of a “programmer mindset”
- Simulation is **essential!** Writing good testbenches is as important as writing good logic.
- Handle adequately resets and clock signals