Micro-electronic

VHDL language

SCHMITZ Thomas

February 23, 2017

Université de Liège

Contact

SCHMITZ Thomas

mail:	T.Schmitz@ulg.ac.be
office:	1.81a
phone:	+32 4 366 2706

Outline I

Outline II

Section 1

Introduction

Goals

Goals

- Reminder of VHDL,
- Differences between simulation and synthesis
- Familiarization with Field Programmable Gate array (FPGA),
- Program FPGA in VHDL language.

2 seances:

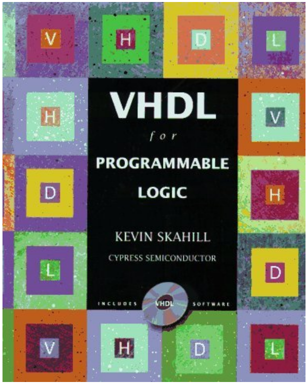
Seance 1: VHDL Bases (reminder),

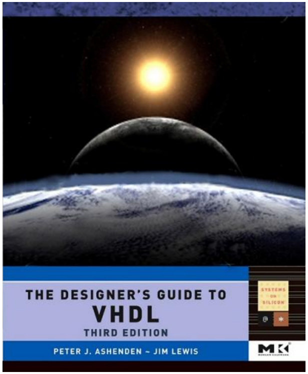
Seance 2: Advanced optimization of VHDL code for FPGA.

References

VHDL for Programmable Logic [3]

The Designer's Guide to VHDL [1]





http://www.textfiles.com/bitsavers/pdf/cypress/warp/VHDL_for_Programmable_Logic_Aug95.pdf

VHDL language

Definition

VHDL is a language of **material description** to represent the behavior and the architecture of a digital electronic system. The full name is VHSIC **Hardware Description Language** (VHSIC = Very High Speed Integrated Circuit).

VHDL code:

- can be emulated,
- can be transcribed in a circuit of logical gates .

Remarks:

- In simulation, the code will be executed sequentially, instruction by instruction.
- During the synthesis process, the circuit described by the code will be implemented by combinatory and/or sequential logic.

Conclusion

The description of a combinatorial circuit could take several instruction to be executed in simulation but will be executed in one time of propagation in real world.

Alternatives

Other philosophy: the schematic description (*schematic*)

- Simple,
- Low level,
- High performance .

Nevertheless, this description is not standardized so its is not perennial .

Remark

We can merge VHDL code and schematic description together.

Tools

Base tools:

- Text editor,
- A simulator,
- An analyzer (transformation of VHDL code to logic gates)
- A *Place & Route* (placement and connection of the logical elements in the component).

A lot of software have all theses functionality:

- Xillinx ISE
- **Altera Quartus**
- Lattice ISP Lever
- **Altium Designer**
- etc...

Section 2

VHDL bases

Subsection 1

Introduction example

VHDL Program content (code source example => ici)

All VHDL program have to be composed by a peer **entity**/**architecture**.

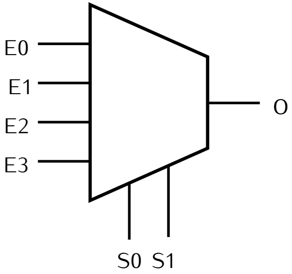
- Entity: it describes the input output of a component,
- Architecture: it describes the behavior of the component.

It has also need some libraries defining the types and operations of the used signals:

```
library ieee ;
use     ieee.std_logic_1164.all ;  -- definition of the type bit, bit vector, ...
use     ieee.numeric_std.all ;     -- (un)signed operations
use     ieee. std_logic_arith .all ; -- (un)signed operations on vector
```

Entity
Declaration

```
entity mux41 is port (
  E0, E1, E2, E3 : in  std_logic ;
  S0, S1         : in  std_logic ;
  O              : out std_logic
  );
end entity mux41;
```



- Declaration of the `mux41` entity
- Declaration of the input/output ports of this entity
- Declaration of the ports type (in, out, ...)

Entity

The entity correspond to the external view of the component.

Entity

Particularity

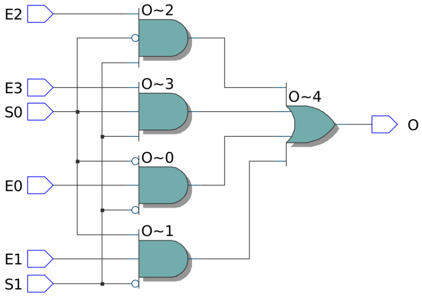
- Mode of the input/output (**ports**):

in:	input port,
out:	output port,
inout:	bidirectional port,
buffer:	output port, but the signal can be read for other internal computations.

Architecture data flow

Declaration

```
architecture  mux41_arch of mux41 is
begin
   O <= ((not S0) and (not S1) and E0) or
           (S0 and (not S1) and E1) or
          ((not  S0) and S1 and E2) or
           (S0 and S1 and E3) ;
end architecture  mux41_arch ;
```



Particularity

An architecture "data flow" describes the behavior of the entity by boolean equation or conditional forms.

Description of an architecture

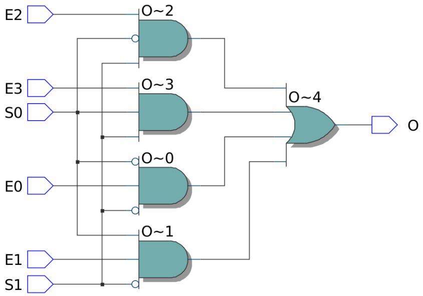
The description by data flow could be difficult for complex function, it exists other ways to describes an architecture:

- Structural : as a scheme, some components are instanced and linked by signals.
- Behavioral : The component is described by some sequential instruction in a *process*

Structural architecture

Declaration

```
architecture  mux41_arch of mux41 is
  signal   not_S0, not_S1 :  std_logic  ;
  signal X :  std_logic_vector (3  downto 0) ;
begin
  inv0  :  not1  port  map ( S0 , not_S0 ) ;
  inv1  :  not1  port  map ( S1 , not_S1 ) ;
  gate0 :  and3
     port  map (not_S0, not_S1, E0, X(0))  ;
  gate1 :  and3
     port  map (     S0, not_S1, E1, X(1))  ;
  gate2 :  and3
     port  map (not_S0,      S1, E2, X(2))  ;
  gate3 :  and3
     port  map (     S0,      S1, E3, X(3))  ;
  gate4 :  or4
     port  map (X(0), X(1), X(2), X(3), O) ;
end architecture  mux41_arch ;
```



- Use of existent elements (from libraries or files),
- Interconnection of theses elements by `port map`,

Structural architecture

Particularity

- This type of architecture use some existing components. An instance is formed like this:

A name:	to identify the instance,
A component:	which is instantiated,
Some connections:	made by **port map**, indicating how to connect the new element to the others with different signals.

Behavioral architecture

Declaration

```
architecture  mux41_arch of mux41 is
begin
  mux41: process(E0, E1, E2, E3, S0, S1)
  begin
     if  (S0 = '0'  and S1 = '0')  then
       O <= E0 ;
     elsif  (S0 = '1'  and S1 = '0')  then
       O <= E1 ;
     elsif  (S0 = '0'  and S1 = '1')  then
       O <= E2 ;
    else
       O <= E3 ;
    end  if  ;
  end process mux41 ;
end  architecture  mux41_arch ;
```

- declaration of a process `mux41`,
- with its list of sensitivity,
- Description of the process by a list of sequential instructions,
- The use of conditional instructions `if-then-elsif-else`,
- The use of *signal assignment* (**<=**),

Utility

A behavioral architecture describes the behavior of a system by a list of sequential instruction given in a *process*.

Behavioral architecture

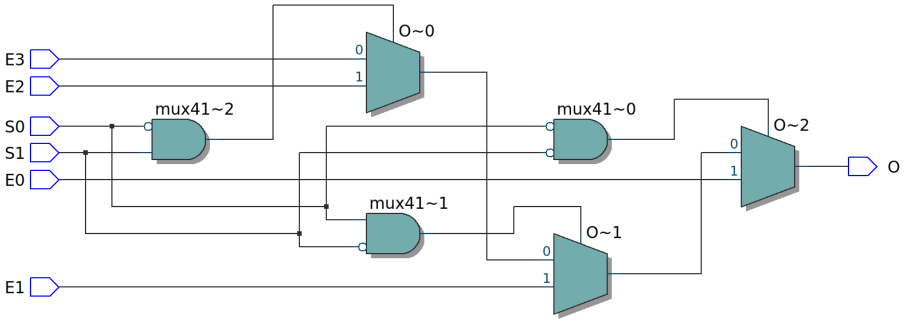
Particularity

- This type of architecture needs at least a **process**, which contains:

Name:	uniquely identifying the process,
A sensitivity list:	identifying the signals that make the process re-evaluated,
An instruction list:	used to synthesized the circuit (or sequentially executed during simulation)

Behavioral architecture

generated circuit



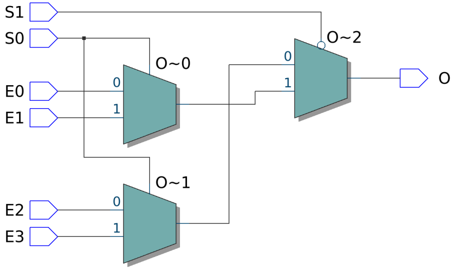
Observations

The synthesized circuit is different from those generated by the architecture *Data Flow* and *Structural*. Nevertheless the function is identical. In conclusion the synthesized circuit could be different in function of the method of description, the compiler, the used component, etc.

Behavioral architecture

Particularity

```
architecture  mux41_arch of mux41 is
begin
  mux41: process(E0, E1, E2, E3, S0, S1)
  begin
    if  (S1 = '0')  then
      if  (S0 = '0')  then
        O <= E0 ;
      else
        O <= E1 ;
      end if  ;
    else
      if  (S0 = '0')  then
        O <= E2 ;
      else
        O <= E3 ;
      end if  ;
    end if  ;
  end process mux41 ;
end architecture  mux41_arch ;
```



- The way to write the code influence directly the circuit scheme

Section 2

VHDL bases

Subsection 2

A fundamental difference

Difference between simulation and synthesis

Synthesis:

Transformation of the code in base logic:

- multiplexers,
- registers,
- logic gates,
- etc.

The the code can be transformed in a circuit.

Difference between simulation and synthesis

Simulation:

the code is compiled, then the instructions are sequentially executed. But the simulation of a digital circuit have some subtlety:

- How to execute several instruction or process in parallel?
- How to manage the affectations?
- How to deal with specific value such as '-' (*Don't care*), 'Z' (*High impedance*) or 'U' (*unknown*)?

Conclusion

In order to write a code that have the same behavior in simulation and in synthesis, it is important to understand the working of the simulator

The simulation cycle

Theory

The simulator operation can be seen as following:

- For each each signal we hold two values:
  - The actual value,
  - The next value.
- The simulation is executed step by step: all the instructions to be executed in time t are executed and the those for the time $t+1$, etc.
- The process whose have changed signals in their sensitivity list are executed.
- At each step, the simulator take into account the actual value of the signals (present time) and compute the next value for those signals (time $t+1$).
- The value of a signal is **never** modified at present time (t)
- The (eventual) change of the signal's value will be programmed after an infinitesimal delay δ after the present time.
- in the case of many modification of a signal's value in the same process, they will be a succession of change for this value spaced by *delta* after the present time. Only the last modification will be taken into account.

Examples

Illustration of a possible mistake

```
entity  my_and8 is port(
  a  :  in  std_logic_vector (  7 downto 0 )  ;
  x  :  buffer  std_logic
  );
end my_and8 ;

architecture  wont_work of my_and8 is
begin
anding:  process(a)
  begin
    x <= '1' ;
    for  i  in  7  downto 0 loop
      x  <= a(i) and x ;
    end loop  ;
  end process  ;
end architecture  wont_work ;
```

This code doesn't work and always give a null output, why ?

- initialization at 0.
- a becomes $=$ '11111111'.
- $\rightarrow$ evaluation of the process *anding* since a is in its sensitivity list.
- `x <= '1'` is evaluated and the **transaction** (x='1') is recorded for the next step
- unfortunately the loop is executed with the present value of x $= 0$
- the result of the anding loop will be zero and a transaction x<=0 will be register for the next step
- the output will stay at '0'.

Examples

Correction

The previous code can be corrected with the introduction of a **variable** `tmp`:

```
entity  my_and8 is port(
  a  :  in  std_logic_vector (  7 downto 0 )  ;
  x  :  buffer  std_logic
  );
end my_and8 ;

architecture  will_work  of  my_and8 is
begin
anding: process(a)
  variable tmp :  std_logic  ;
  begin
    tmp := '1'  ;
    for  i  in  7 downto 0 loop
      tmp := a(i)  and tmp ;
    end loop ;
    x  <= tmp ;
  end process ;
end architecture  will_work  ;
```

Conclusion

It is important to understand how the simulator works to write code that have the right behavior during the synthesis.

Simulation vs Synthesis

Note that a code which can be synthesized can be simulated but a code of simulation may not be wright for synthesis.

Reference

See [3], chapter 3.3.5.

Simulation summary

The simulation process can be seen as:

- Sequential execution of the code.
- At present time t,each signal eventually record a *transaction* for the time $t + \delta$.
- Application of the *transactions* at time $t + \delta$.

Section 3

Sequential logic

Subsection 1

Base

Introduction examples

Flip flop senitive to the edge (Flip flop D)

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port(
  d, clk : in std_logic ;
  q      : out std_logic
  );
end entity dff_logic ;

architecture dff_logic_arch of dff_logic is
begin
  process( clk )
  begin
    if( clk'event and clk = '1' )
    then
      q <= d ;
    end if ;
  end process ;
end architecture dff_logic_arch ;
```

- Sensitivity list on `clk`,
- rising edge detecting
  `if( clk'event and clk = '1' )`
- No `else`: the flip flop keep its value.

Remark

Most of the simulator does not allow an `else` statement after a `if( clk'event and clk = '1' )`. Indeed this construction will be ambiguous since the event is true just for a short moment.

Examples of introduction

Flip flop sensitive to the value (transparency Latch)

```
library ieee;
use ieee.std_logic_1164.all;
entity ff_logic is port(
  d, clk : in std_logic ;
  q      : out std_logic
  );
end entity ff_logic ;

architecture ff_logic_arch of ff_logic is
begin
  process( clk , d )
  begin
    if ( clk = '1' ) then
      q <= d ;
    end if ;
  end process ;
end architecture ff_logic_arch ;
```

- Sensitivity list `clk` and `d`,
- Condition on the `clk` value,(not on the edge anymore),
- no `else`.

Examples

Flip flop with asynchronous reset

```
use ieee.std_logic_1164.all;
entity rff_logic is port(
  d, clk, rst : in std_logic ;
  q           : out std_logic
  );
end rff_logic ;

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

- Function `rising_edge()`,
- Reset in the sensitivity list,
- asynchronous reset (priority to reset operation) ,
- for synchronous reset , put it after the `rising_edge()` operation

Exemples introductifs

wait

The use of **wait**:

```
architecture dff_logic_arch2 of dff_logic is
begin
  process begin
    wait until ( clk = '1' ) ;
      q <= d ;
    end process ;
end architecture dff_logic_arch2 ;
```

it works if `wait until()` is the first instruction of the process, so it is impossible to make a asynchronous reset with the wait clause.

Subsection 2

State machine

Introduction

The conception of a state machine needs the following step

- State diagram,
- Transformation of the state diagram into binary code,
- Optimization with Karnaugh table,
- => Boolean equations.

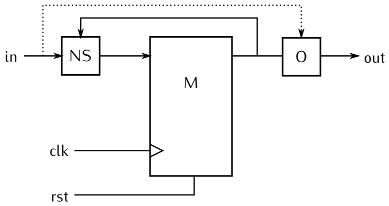
With the VHDL we can make it more easily:

- State machine,
- => Code.

Several ways to encode a state machine

A state machine can be represented as follow:

- A function that compute the next state (`NS`),
- A function that compute the outputs (`O`),
- A memory (`M`), that keeps the current state.



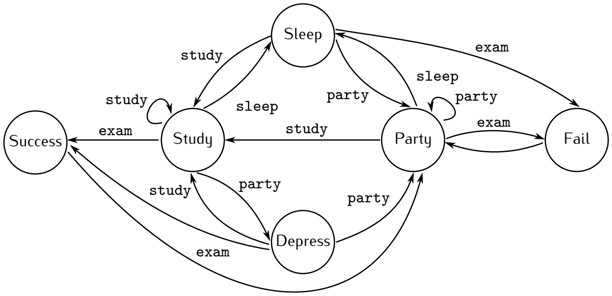
1 process: (NS + M + O),

2 process: M + (NS + O); O + (NS + M),

3 process: NS + M + O.

Example

Model of a student:



- States of the students : Sleep, Study, Success, Depress, Party, Fail
- Input signals :
  - Teacher : gives a exam.
  - students needs : need to study, need to sleep, need to go to a party.
  - Clock of the system : one activity per day.

Example

3 process

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

entity student is port(
  professor : in std_logic ;
  need      : in std_logic_vector ( 1 downto 0 ) ;
  day       : in std_logic ;
  stud_out  : out std_logic_vector ( 2 downto 0 )
  );
end entity student ;

architecture student_arch of student is
  type student_state is (Sleep, Party, Study, Depress,
                         Success, Fail) ;

  constant Exam : std_logic := '1';
  constant Sleep : std_logic_vector ( 1 downto 0 ) := "00" ;
  constant Study : std_logic_vector ( 1 downto 0 ) := "01" ;
  constant Party : std_logic_vector ( 1 downto 0 ) := "10" ;

  signal present_state : student_state := Sleep ;
  signal next_state    : student_state := Sleep ;

  begin

  -- First process (compute next state, combinatorial)
  next_state_calc : process( professor, need, present_state )
  begin
    case present_state is
```

```
      when Repos =>
        next_state <= Sleep ;
        if professor = Exam then
          next_state <= Fail ;
        else
          if need = Study then
            next_state <= Study ;
          elsif need = Party then
            next_state <= Party ;
          end if ;
        end if ;

      when Party =>
        next_state <= Sleep ;
        if professor = Exam then
          next_state <= Fail ;
        else
          if need = Study then
            next_state <= Study ;
          elsif need = Party then
            next_state <= Party ;
          end if ;
        end if ;
```

Example

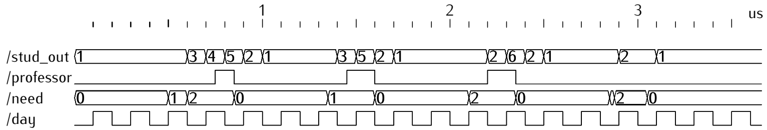
3 process

```
    when Study =>
      next_state <= Sleep ;
      if professor = Exam then
        next_state <= Success ;
      else
        if need = Study then
          next_state <= Study ;
        elsif need = Party then
          next_state <= depress ;
        end if ;
      end if ;

    when depress =>
      next_state <= depress ;
      if professor = Exam then
        next_state <= Success ;
      else
        if need = Study then
          next_state <= Study ;
        elsif need = Party then
          next_state <= Party ;
        end if ;
      end if ;

    when Fail =>
      next_state <= Party ;

    when Success =>
      next_state <= Party ;
```

```
      end case ;
  end process next_state_calc ;

  -- Second process
  -- (update current state, sequential)
  update: process( day )
  begin
    if (day'event) then
      present_state <= next_state ;
    end if;
  end process update ;

  -- Third process (implicit)
  -- (compute output, combinatorial)
  with present_state select
  stud_out <= "001" when Sleep,
              "010" when Party,
              "011" when Study,
              "100" when depress,
              "101" when Success,
              "110" when Fail,
              "111" when others ;

end architecture student_arch ;
```

Example

Simulation



Legend:

`stud_out`:

- 1 – Sleep
- 2 – Party
- 3 – Study
- 4 – Depress
- 5 – Success
- 6 – Fail

`need`:

- 0 – Sleep
- 1 – Study
- 2 – Party

Example

Simulation

Remarks:

- The modification of the current state is triggered on a transition of the signal `day`.
- 2 process (the third one is implicit but could be put in a process sensitive to the `present_state`).
- On the last simulation example, the student change its need too fast, only the last need is taken into account.

Conclusions

Other ways to do the same:

- Update of the outputs in the `case` statement,
- Process `update` integrated in the process of next step computation (with a change in the list of sensitivity)
- You will find a code example for a FIR in VHDL here

Section 4

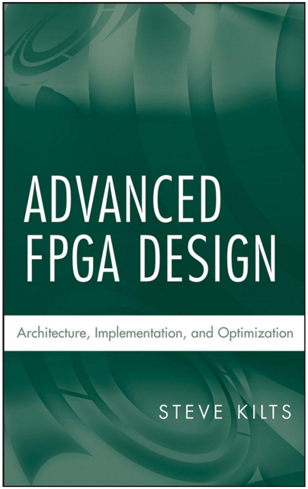
Code optimisation

Subsection 1

Speed optimization

Reference

Advanced FPGA Design: Architecture, Implementation, and Optimization [?]



Introduction

Several possibilities of optimization

Throughput:	The number of treated bits per second.
Latency:	duration between input time and output time a given data
Timing:	Duration of the propagation delay in the combinatory circuit. Constrain the speed of the clock.

The*timings* are the combinatorial delays between two registers. The critical path is the path with the longest delay, it will fix the maximum frequency clock.

Example

We take the following example:

```
XPower = 1 ;
for ( i = 0 ; i < 3 ; i++ )
  XPower = X * XPower ;
```

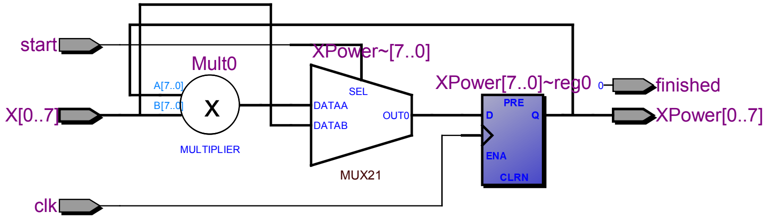
Iterative algorithm:

- The same variable is used for all the iterations,
- New data cannot come-in until the end of the computation of the current data.

Example

```vhdl
architecture iterativ of power3 is
  signal ncount : integer := 0;
  begin

   calculus : process(clk, start)
  variable intermediate : std_logic_vector (0 to 15);
  begin
     if (rising_edge(clk)) then
       if (start ='1') then
         XPower <= X;
         ncount <= 2;
       elsif (finished = '0') then
         ncount <= ncount - 1;
         intermediate := XPower * X;
         XPower <= intermediate(0 to 7);
      end if;
    end if;
     finished <= '1' when (ncount = 0) else '0';
  end process calculus;
end architecture;
```

Example



Speed:

Throughput:	8/3bits / clock
Latency:	3 clocks
Timing :	A multiplier on the critical path

Increase of the throughput

Goal

This type of optimization aims to treat the bigger number of data per second independently of the input/output/delay

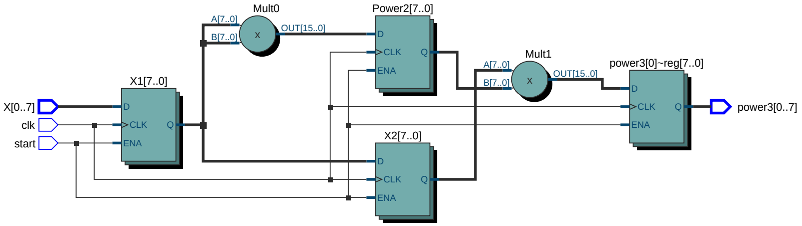
key notions:

Pipeline:	data are treated in several steps allowing new input data at each rising edge of the clock,
unrolling loop:	algorithm to obtain pipelined code from classic code.

Throughput increase

```
architecture pipelined_arch of power3 is

  variable X1 : std_logic_vector (0 to 7);
  variable X2 : std_logic_vector (0 to 7);
  variable power2 : std_logic_vector (0 to 15);

begin
  calculus : process(clk, start)
  variable I : std_logic_vector (0 to 15);
  begin
     if (rising_edge(clk)) then
       if (start = '1') then
         -- Pipeline Stage 1
         X1 <= X;
         -- Pipeline Stage 2
         X2 <= X1;
         Power2 <= X1 * X1;
         -- Pipeline Stage 3
         I := X2 * Power2(0 to 7);
         power3 <= I(0 to 7);
       end if;
     end if;
  end process calculus;
end architecture pipelined_arch;
```

Throughput increase



Speed:

Throughput:	8bits / clock
Latency:	3 clocks
Timing:	One multiplier on the critical path

clk	X	X1	X2	Power2	Power3
0	d1	U	U	U	U
1	d2	d1	U	U	U
2	d3	d2	d1	d1*d1	U
3	d4	d3	d2	d2*d2	d1*d1*d1
4	d5	d4	d3	d3*d3	d2*d2*d2

Throughput increase

The loop has been *unrolled*,each step can be performed without waiting the end of an other task.

- Better Throughput,
- More resources are used (space),

Rule

Unroll a loop increase the throughput with a factor n to the detriment of an increase of the surface with a factor n.

Latency optimization

Goal

Minimize the input output duration

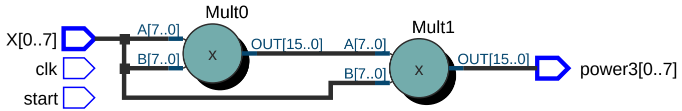
Key notions:

Parallelism:	Shorten path by making several paths
Deleting registers:	deleting pipeline.

Latency optimization

```
architecture latency of power3 is

  begin

   calculus : process(X)
  variable I1, X1, X2 : std_logic_vector (0 to 7);
  variable I2, I3 : std_logic_vector (0 to 15);

  begin
    I1 := X;
    X1 := X;

    X2 := X1;
    I2 := I1 * X1;

    I3 := I2(0 to 7) * X2;
    power3 <= I3(0 to 7);
  end process calculus;
end architecture;
```

Latency optimization



Speed:

Throughput:	8bits / clock
Latency:	2 multiplier propagation delay, 0 clocks
Timing :	2 multiplier on the critical path

Latency optimization

All the register have been deleted

- Lower latency.
- Increase timing.

Rule

The latency can be reduced by suppressing pipelines and registers but the combinatorial delay increases.

Timing reduction

Goal

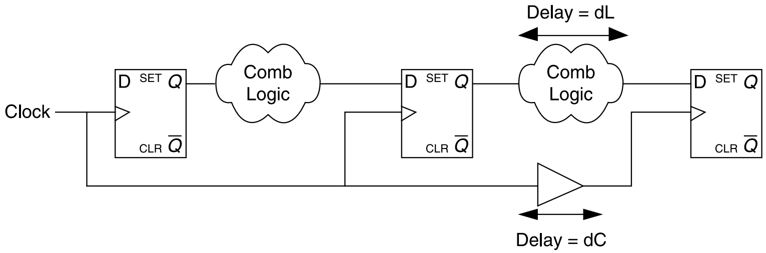
This type of optimization aims to increase the maximal frequency clock by reducing the length of combinatorial logic between to registers.

Key notions:

$$F_{max} = \frac{1}{T_{clk \to q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}}$$

$T_{clk \to q}$:	delay between the clock and the output of a register Q,
T_{logic}:	Combinatorial delay,
$T_{routing}$:	routing propagation delay between the different logical elements,
T_{setup}:	minimum setup time (stability of the input D),
T_{skew}:	Propagation delay between the clock and the registers.

Tiling reduction



different methods are available to reduce these delays

Timing reduction

Adding registers: A maximum of pipeline.

Example: In a Fir filter, execution of the code in one rising edge of the clock:

```
if (rising_edge(clk)) then
  X1 <= X;
  X2 <= X1;
  Y <= A*X + B*X1 + C*X2;
end if;
```

In this FIR a register is added between the product terms and the sum:

```
if (rising_edge(clk)) then
  X1 <= X;
  X2 <= X1;
  prod1 <= A*X;
  prod2 <= B*X1;
  prod3 <= C*X2;
end if;
Y <= prod1 + prod2 + prod3;
```

Rule

Adding register enhance the combinatorial delay by splitting the critical path into several shorter critical paths.

Timing reduction

Parallelism: Cutting one critical path into several path with a shorter critical path.
Example: With the power example, we can repesent our X data as a set of two subset:

$$X = \{A, B\}$$

with A the MSB and B the LSB. We have:

$$X * X = \{A, B\} * \{A, B\} = \{(A * A), (2 * A * B), (B * B)\}$$

$$X * X = \{(A << 4) + B\} * \{(A << 4) + B\} = \{(A * A) << 8 + (2 * A * B) << 4 + (B * B)\}$$

Where each element can be calculated in parallel.
We have more multiplier but smaller (4bits and not 8).

Rule

Splitting function in smaller function allows to increase parallelism and thus to minimize the timing of each paths

Timing reduction

Priority encoder: If the signals are mutually exclusives take it into account!

Example: Mux example:

```
   if  ( ctrl (0)  = '1')  then  rout (0)  <= in;
elsif  ( ctrl (1)  = '1')  then  rout (1)  <= in;
elsif  ( ctrl (2)  = '1')  then  rout (2)  <= in;
elsif  ( ctrl (3)  = '1')  then  rout (3)  <= in;
end if;
```

Logic gates are added to modelize the priority implied in the structure `if...elsif`. But we can write :

```
if  ( ctrl (0)  = '1')  then  rout (0)  <= in; end if;
if  ( ctrl (1)  = '1')  then  rout (1)  <= in; end if;
if  ( ctrl (2)  = '1')  then  rout (2)  <= in; end if;
if  ( ctrl (3)  = '1')  then  rout (3)  <= in; end if;
```

Rule

Deleting the priority where is it possible allow to reduce the complexity of the combinatorial logic.

Timing reduction

Reference

Learn more in [2]!

Subsection 2

Space optimization

Introduction

Plusieurs optimisations possibles:

- Réutiliser les ressources au maximum,
- Diminuer la surface utilisée par une opération,
- Utiliser les *ressources spéciales* du FPGA (RAM, multiplicateur, etc).

Enrouler les pipelines

Objectif

Faire l'opération inverse utilisée précédemment pour utiliser moins d'éléments, au prix d'un débit plus faible.

Notions clés:

- Connaître les ressources dédiées d'un composant.
- Essayer de reformuler le problème.

Enrouler les pipelines

Soit un multiplicateur, qui ne serait pas implémenté via un module DSP:

```
architecture simple of mult is
  begin

   calculus : process (clk)
  begin
     if (rising_edge(clk)) then
       product <= A * B;
     end if;
  end process;
end architecture;
```

Ressources utilisées:

Fonctions combinatoires:	103
Registres:	16
Eléments logiques:	103

Enrouler les pipelines

Version plus lente, mais utilisant des accumulateurs:

```
architecture small of mult is
 begin
 calculus : process(clk, start)
   variable multcounter : unsigned(2 downto 0);
   variable adden : std_logic;
   variable shiftA : unsigned(15 downto 0) := 0;
   variable shiftB : unsigned(7 downto 0);
 begin
   adden := shiftB(0) AND (NOT done);
   done <= multcounter(0) AND multcounter(1) AND multcounter(2);
   if (rising_edge(clk)) then
     if (start = '1')   then multcounter := "000";
     elsif (done = '0') then multcounter := multcounter + 1;
     end if;

     if (start = '1') then shiftB := B;  -- Shift for B
     else
       shiftB(6 downto 0) := shiftB(7 downto 1);
       shiftB(7)          := '0';
     end if;

     if (start = '1') then shiftA(7 downto 0) := A; -- Shift for A
     else
       shiftA(15 downto 1) := shiftA(14 downto 0);
       shiftA(0)           := '0';
     end if;

     if    (start = '1') then product <= "0000000000000000";
     elsif (adden = '1') then product <= product + shiftA;
     end if;
   end if;
 end process;
end architecture;
```

Ressources utilisées:

Fonctions combinatoires: 45

Registres: 43

Eléments logiques: 45

Réutilisation des ressources

Objectif

Utilisation de logique supplémentaire pour permettre une réutilisation des ressources.

Notions clés:

- Utilisation d'une machine d'état
- Utilisation de signaux de contrôle

Réutilisation des ressources

Prenons l'exemple d'un filtrage de type FIR:

$$y = A * X[0] + B * X[1] + C * X[2]$$

En l'état, utilisation de 3 multiplicateurs et d'une somme.

Possibilité d'utiliser un seul multiplicateur:

- Utilisation d'une machine d'état
- Les multiplications sont réalisées séquentiellement

Partage de ressources

Objectif

Utilisation de ressources identiques pour des fonctions complètement différentes

Notions clés:

- Détection de ressources partageables,
- Création de ressources génériques

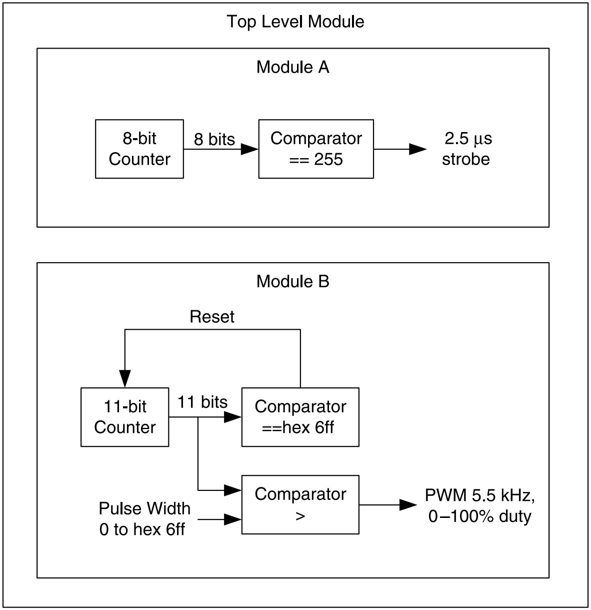
Partage de ressources

Les compteurs sont utilisés dans de multiples programmes. Soient les deux compteurs suivants:

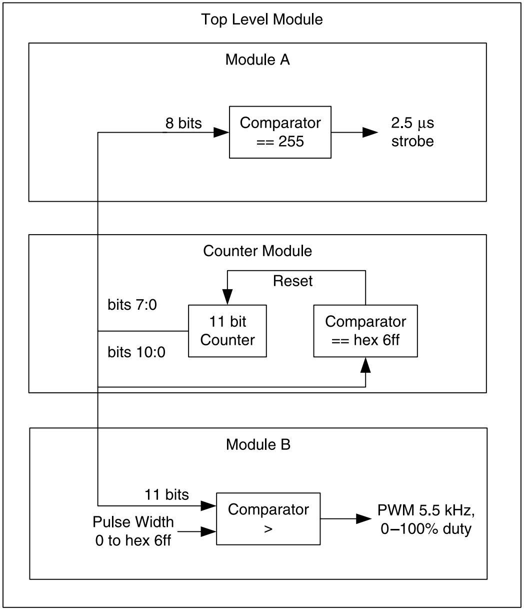
- Comptage sur 8 bits pour diviser la fréquence d'une horloge,
- Comptage sur 11 bits pour la génération d'un signal PWM

Il est possible de coder un compteur commun qui sera utilisé simultanément par les deux entités.

Partage des ressources



Partage des ressources



Utilisation des ressources dédiées

Objectif

Utiliser au maximum les ressources dédiées d'un FPGA, telles que les multiplicateurs, RAM, etc

Notions clés:

- Connaître les limites du matériel (présence de *set* ou de *reset*, asynchrone ou non?)
- Particulièrement vrai lors de l'utilisation de mémoire vive!
- Utilisation des plugins offerts (Altera: MegaWizard)

Subsection 3

Power optimization

Introduction

La puissance dissipée dans un circuit CMOS peut être calculée via la formule exprimant le courant de charge des circuits capacitifs:

$$I = V \times C \times f$$

$$P = V \times I$$

V:	Tension d'alimentation, elle est fixée par le fabriquant,
C:	La capacitance,
f:	La fréquence.

On a donc deux possibilités d'action (en supposant V fixé):

- Diminuer la fréquence d'horloge,
- Diminuer la capacitance en diminuant le nombre de portes ou parties de circuit pilotées par le signal d'horloge.

Réduction de la puissance

Autres remarques:

- Réduire le temps de montée/descente des signaux aux entrées du FPGA,
- Ne JAMAIS laisser d'entrées flottantes.
- Ne pas diminuer V en dessous de la valeur préconisée.
- Utilisation de "Dual-edge triggered flip-flops" (seulement si présents de base).

Contrôle d’horloge

La manière la plus simple est de désactiver des parties de circuit: on parle de "*Clock Control*". Deux possibilités:

- Utilisation du "*Clock Enable*" des flips-flops ou d’un *Multiplexeur* global,
- Utilisation de "*Clock Gating*" (Utiliser de la logique pour contrôler l’horloge).

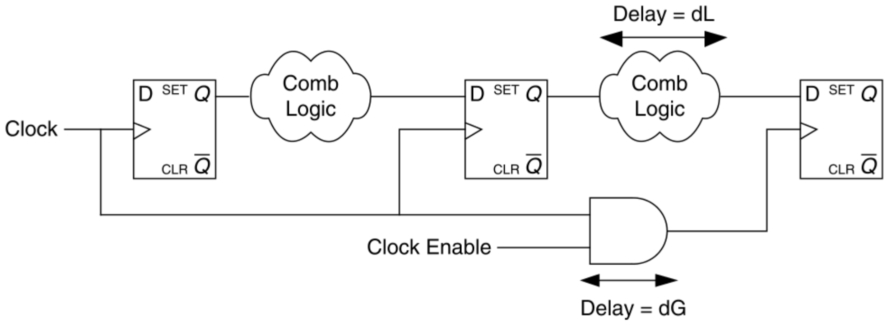
Règle

La meilleur manière de réduire la puissance consommée est de désactiver l’horloge dans les modules non-utilisés, via les ressources dédidées du FPGA.

Contrôle d’horloge

Utiliser le "*Clock Gating*" est une **très** mauvaise solution:

- Introduction de délai d’horloge (*skew*).
- Et donc possibilité de violation de timing (voir section suivante).
- Possibilité de "fly-through" si $dL < dG$.



Section 5

Advanced notions

Subsection 1

Clock domain

Introduction

FPGA are generic component, they need to be interfaced with other components such as:

- Processors,
- Memories,
- ...

Often theses components have they own frequency clock, we have to be sure that the data are well transmitted between them.

Definition

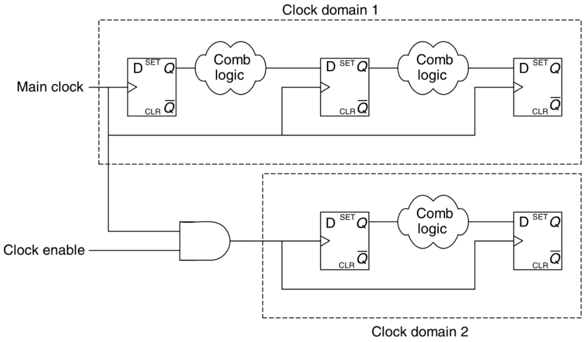
A "*Clock Domain*" is a part of the FPGA working at the same frequency clock.

Introduction

Several types of problems can occur when we transit data/signals between two clock domains:

No repeatability: errors depend of the relatives delay between the two clocks,

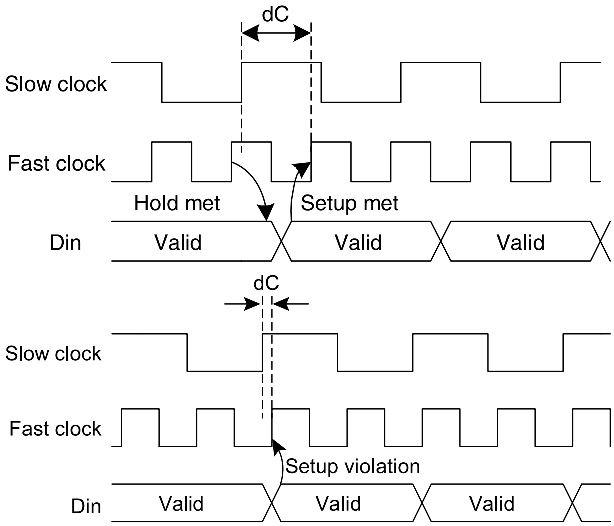
No simulation: the simulation of timing is difficult since we have only max min delay in the datasheet,



Origin of the problem

2 words

Timing Violation



Origin of problem

When a timing problem occurs, we can have meta-stability:

- no valid tension in a Flip Flop,
- Unknown output during a period of the clock (theoretically an infinity).

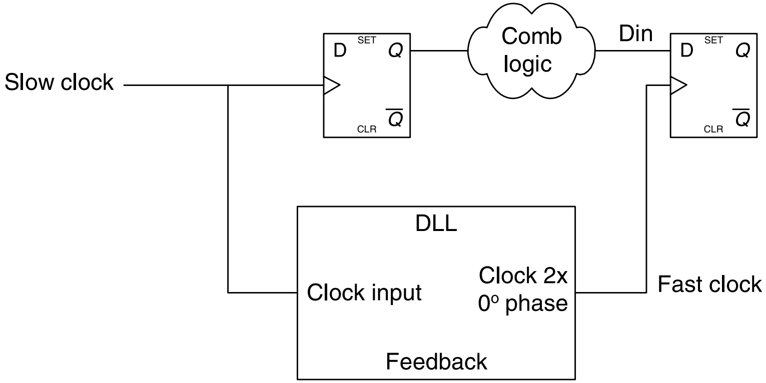
Several solutions:

- Phase control,
- Double Flip-Flop,
- FIFO

Phase control

Principle

A special circuit capable of synchronized two clocks that are a multiple of each other.



Double Flip-flop

Principle

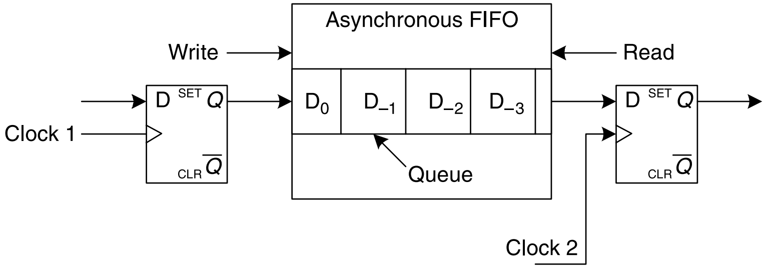
Use a cascade of two FF in order to reduce the probability of meta-stability.

```
 dff  : process (clk)
begin
   if (risingedge(clk) then
      ff1  <= input;
      ff2  <= ff1;
  end if;
end process;
```

FIFO

Principle

Use of an asynchronous FIFO receiving data from one clock domain and transmitting them to an other.



Subsection 2

Trap to avoid

Introduction

- Priority encoder,
- Loop,
- Variable VS Signal,
- *Inferred Latch*.

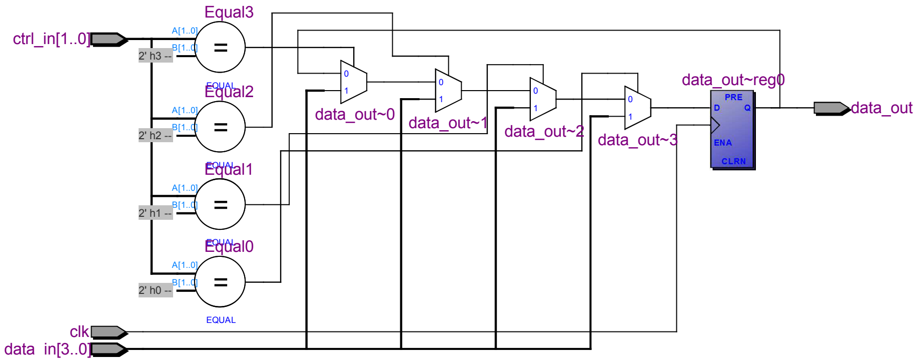
Priority encoding

Rules

To describe a priority encoding, chose a structure of type `if-elsif-else`

```
architecture priority of tree is
 begin

 proc : process (clk)
 begin
   if (rising_edge(clk)) then
         if ( ctrl_in = "00") then data_out <= data_in(0);
      elsif ( ctrl_in = "01") then data_out <= data_in(1);
      elsif ( ctrl_in = "10") then data_out <= data_in(2);
      elsif ( ctrl_in = "11") then data_out <= data_in(3);
     end if;
   end if;
 end process;
end architecture;
```

Priority encoding



Priority encoding

Attention

Chose a structure of type `if-elsif-else` and not a succession of `if`, if not, the compiler will chose an arbitrary order.

```
architecture bad of tree is
  begin

  proc : process (clk)
  begin
     if (rising_edge(clk)) then
        if ( ctrl_in = "00) then data_out <= data_in(0); end if;
        if ( ctrl_in = "01") then data_out <= data_in(1); end if;
        if ( ctrl_in = "10") then data_out <= data_in(2); end if;
        if ( ctrl_in = "11") then data_out <= data_in(3); end if;
     end if;
  end process;
end architecture;
```

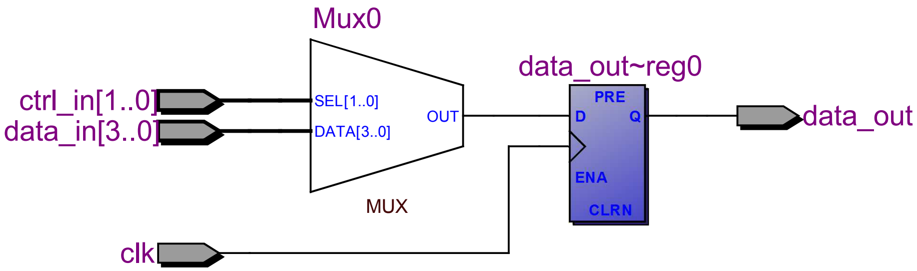
Non priority encoding

Rules

To describe a non priority encoding, use a succession of `if`, or a structure of type `case-select`

```
architecture parallel of tree is
  begin

  proc : process(clk)
  begin
     if(rising_edge(clk)) then
       case ctrl_in is
         when "00" => data_out <= data_in(0);
         when "01" => data_out <= data_in(1);
         when "10" => data_out <= data_in(2);
         when "11" => data_out <= data_in(3);
       end case;
     end if;
  end process;
end architecture;
```

Non priority encoding



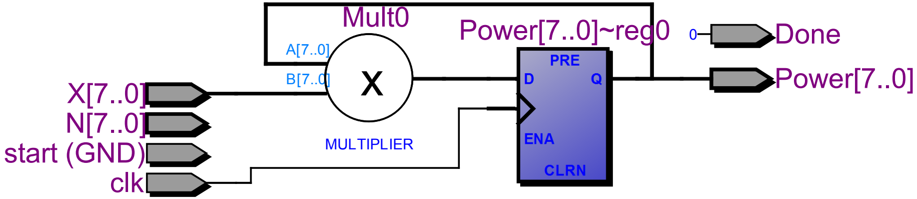
For loop

Rules

Don't use loop `for` to make algorithm iterations unless the right bound of the loop is a constant.

```
architecture bad of loop_prob is
  begin

   calculus : process(clk)
     variable intermediate : unsigned(15 downto 0);
     variable i : integer;

     begin
        if (rising_edge(clk)) then
          for i in 0 to to_integer(N) loop
            intermediate := Power * X;
            Power <= intermediate(7 downto 0);
          end loop;
        end if;
     end process;
end architecture;
```

Boucles for



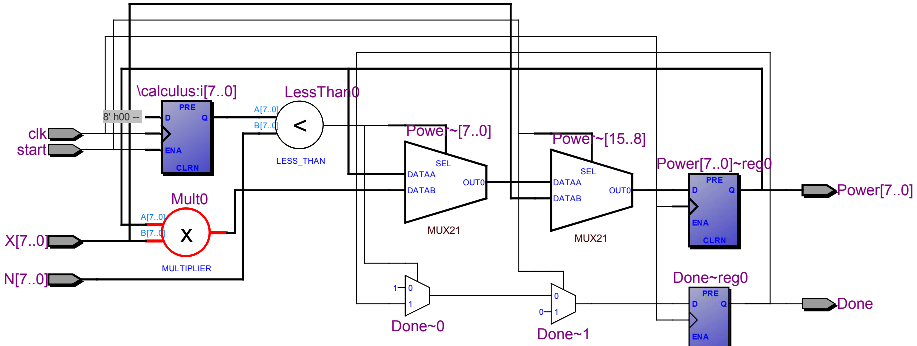
```
Warning (11792):  VHDL warning at loop_prob.vhd(26):  right bound of range must
be a constant
```

For loop

The problem could be avoid in this way :

```
architecture good of loop_prob is
  begin

   calculus : process (clk)
  variable i : unsigned(7 downto 0);
  variable intermediate : unsigned(15 downto 0);

  begin
   if (rising_edge(clk)) then
     if (start = '1') then
       Power <= X;
       i := (others => '0');
       Done <= '0';
     elsif (i < N) then
       Intermediate := Power * X;
       Power <= Intermediate(7 downto 0);
     else
       Done <= '1';
     end if;
  end if;

  end process;
end architecture;
```

Boucles for



\calculus:i[7..0]
PRE
D
Q
ENA
CLRN
8' h00 --
LessThan0
A[7..0]
B[7..0]
<
LESS_THAN
clk
start
Mult0
A[7..0]
B[7..0]
X
MULTIPLIER
X[7..0]
N[7..0]
Power~[7..0]
SEL
DATAA
DATAB
OUT0
MUX21
Power~[15..8]
SEL
DATAA
DATAB
OUT0
MUX21
Power[7..0]~reg0
PRE
D
Q
ENA
CLRN
Power[7..0]
Done~0
1
0
1
Done~1
0
0
1
Done~reg0
PRE
D
Q
ENA
CLRN
Done

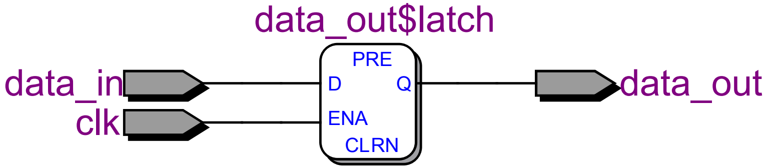
Inferred Latch

Rule

Always complete all the possibilities in a structure `if-else`, otherwise a register will be used to maintain the previous value of the signal.

```
architecture inf of inferred is
  begin

  proc : process(clk, data_in)
  begin
     if (clk = '1') then
       data_out <= data_in;
     end if;
  end process;

end architecture;
```

Inferred Latches



```
Warning (10631):  VHDL Process Statement warning at inferred.vhd(18):  inferring
latch(es) for signal or variable "data_out", which holds its previous value in
one or more paths through the process
```

Section 6

Project

Goals and instructions

Goals

To be familiarized with:

- The creation of a simple electronic circuit,
- VHDL language.

Le projet consistera en la réalisation d'un code tournant sur les cartes **DE0-nano**

Timing

Three steps:

25 March:	Short Report:

- Description in 5 rows of the project,
- Description of the structure of the code,
- Needed materials.

1 Mai : Final report

- Code VHDL,
- Schematic circuit,
- Explanations code and structure.

Session : Demonstration in the lab R100

- Date fixed at the beginning of the session,
- Oral presentation around 10-15 minutes.

Attention

Titre du fichier envoyé : ELEN0037Projet(ou PreProjet)_JDaniels_WLawson_BLabel

Advices

- Don't forget you write a circuit not a program
- Don't forget how simulator works (sequential)

Stuck ?

- Example code Mux41 and Fir filter
- Learn VHDL on youtube
- VHDL code examples
- VHDL tutorials

Good work!

Section 7

Syntax

Sequential and parallel execution

Fondamental

- A l'extérieur d'un processus, les instructions/processus s'exécutent en parallèle
- A l'intérieur d'un processus, les instructions s'exécutent séquentiellement lors de la simulation, ou afin de décrire le circuit sous-jacent

- Ainsi, certaines instructions sont plus spécifiques à une description parallèle, telles que les instructions de choix:
  Si une variable vaut x, alors une autre variable vaudra y
  Ce genre d'instruction peut être exécutée en parallèle puisque on peut vérifier cette phrase à tout moment.
- En revanche, certaines instructions sont plus spécifiques à une description séquentielle:
  Une variable vaut x, puis si une autre variable change, alors elle vaudra y
  Ne peut pas s'exécuter en permanence puisqu'elle attend des changements.
- Les instructions plus spécifiques parallèle peuvent se trouver aussi dans les processus.

Subsection 1

Syntaxe de base

Syntaxe de base

Fonction	Exemple ou règle
Commentaires	*−− Commentaire sur une ligne* **entity** reg4 **is** *−− Déclaration de l'entité*
Identificateurs	• Caractères alphabétiques, chiffres ou underscores • Doit démarrer avec un caractère alphabétique • Ne peut se terminer avec un underscore • Ne peut contenir deux underscores successifs • Insensible à la casse (sauf pour les énumérations de type littéral);
Mots réservés	• **and**, **or**, **nor**, etc... • **if**, **else**, **elsif**, **for**, etc... • **entity**, **architecture**, etc...

Fonction	Exemple ou règle
Nombres	Real literal: 0, 23, 2E+4, etc... Integer literal: 23.1, 42.7, 2.9E-3, etc... Base literal: 2#1101#, 8#15#, 10#13#, 16#D#, etc... Underscore: 2_867_918, 2#0010_1100_1101#
Caractères et chaînes	'A' −− *caractère* "A string" −− *Chaîne de caractères* "A string in a ""A string"". " B"0100011" −− *Chaîne binaire* X"9F6D3" −− *Chaîne hexadécimale* 5B#11# = B"00011" −− *Chaîne à taille fixe*

Notation EBNF

Définition

Métalanguage permettant de décrire avec précision la syntaxe d'un langage comme le VHDL.

Fonction	Exemple ou règle
Définition: ⇐	Affectation <= nom_variable := expression;
Optionnel: [ ]	Fonction <= nom_fonction [(arguments)];
0 ou plus: { }	Déclaration_de_process <= **process is** { définition_process } **begin** { déclarations } **end process**;
Répétition: ...	Déclaration_case <= **case** expression **is** déclaration {...} **end case**;
liste: {"delimiteur"...}	Liste_identificateurs <= identificateur {, ...}
Ou: |	mode <= **in** | **out** | **inout**

Constantes

Syntaxe

constant identificateur {, ...} : sous-type [:= expression]

Exemples:

```
constant number_of_bytes : integer := 4;
constant number_of_bits : integer := 8*number_of_bytes;
constant size_limit , count_limit : integer := 255;
constant prop_delay : time := 3ns;
```

Usage:

- Similaire aux déclarations préprocesseur en C.
- Pratique pour rendre le code lisible.

Variables

Syntaxe

```
variable  identificateur  {,  ...}  : sous−type [ := expression ]

variable_à_assigner <= nom := expression;
```

Exemples:

```
variable number_of_bytes : integer := 4;
variable  size_limit ,  count_limit  :  integer := 255;
variable prop_delay : time := 3ns;
 ...
number_of_bytes := 2;
number_of_bits := number_of_bytes ∗ 8;
```

Usage:

- Similaire aux variables C, à l'intérieur d'un processus.
- L'affectation d'une valeur à une variable prend effet immédiatement.
- Pas de signification physique.
- Ne peut être déclarée et accessible que dans un process.

Variables

Attention!

Différent d'une affectation de signal! La variable est modifiée tout de suite, le signal entraîne une modification future!

Attention!

Une variable n'a pas de signification physique, et n'est en général pas synthétisable. Les variables sont souvent utilisées comme indices de boucles.

Type

Syntaxe

type identificateur **is** type_definition ;

subtype identificateur **is** subtype_definition ;

type_definition <= **range** expression (**to** | **downto**) expression;

subtype_definition <= **type** [**range** expression (**to** | **downto**) expression]

Exemples:

```
type players is range 0 to 4;
type days is 0 to 7;
subtype bit_index is integer range 31 downto 0;
```

Usage:

- Contraindre des constantes, variables, etc, à certaines valeurs.
- Très utile pour la lisibilité du code.

Remarque

- On peut déclarer des types avec des entiers ou des flottants.
- La valeur par défaut est la première déclarée (plus petite si `to`, plus grande si `downto`).

Type

Enumérations

Syntaxe

type identificateur **is** type_definition ;

type_definition <= ((identificateur | caractère) {, ...}) ;

Exemples:

```
type player is (Player1, Player2, Player3, CPU)
type days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun)
 ...
variable today : days;
today := Mon;
```

Usage:

- Facilité de lecture du code.
- Optimisation.

Remarque

- Il s'agit d'un type mais avec des noms, donc ce qui s'applique aux types s'applique aux énumérations également.

Types

Prédéfinis

Booléens: **type** boolean **is** (false, true);

Bits: **type** bits **is** ('0', '1');

std_ulogic: ou std_logic

```
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
```

Caractères: **type** character **is** (nul, soh, ' ', '*', '1', 'A', '...');

Remarque

Pour le type `std_logic`, il convient d'inclure une librairie:

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

Types

Tableaux

Syntaxe

array_def <= **array** (discrete_range {, ...}) **of** elements

discrete_range <= discrete | expression (**to** | **downto**) expression

Exemples:

```
type word is array (0 to 31) of std_logic;
 ...
type controller_state is ( initial , idle , active , error );
type state_counts is array ( controller_state range idle to error ) of natural;
 ...
variable buffer : word;
 ...
buffer (0) := '0';
```

Types

Tableaux – multidimensionnels

Exemple:

```
type table16x8 is array(0 to 15, 0 to 7) of std_logic;
constant 7seg: table16x8 := (
   "00111111",  -- 0
   "00000110",  -- 1
   "01011011",  -- 2
   "01001111",  -- 3
   "01100110",  -- 4
   "01101101",  -- 5
   "01111101",  -- 6
   "00000111",  -- 7
   "01111111",  -- 8
   "01101111",  -- 9
   "01110111",  -- A
   "01111100",  -- B
   "00111001",  -- C
   "01011110",  -- D
   "01111001",  -- E
   "01110001");  --F
```

Types

Tableaux - agrégats

Syntaxe

aggregate <= (([choix =>] expression) {, ...})

Exemples:

```
type point is array (1 to 3) of real;
constant origine : point := (0.0, 0.0, 0.0);
variable view_point : point := (10.0, 20.0, 0.0.) ;
variable view_point : point := (1 => 10.0, 2 => 20.0, 3 => 0.0);

variable coeff : coeff_array := (0 => 1.6, 1=> 2.3, 2 => 1.6, 3 to 63 => 0.0);
variable coeff : coeff_array := (0 => 1.6, 1=> 2.3, 2 => 1.6, others => 0.0);
variable coeff : coeff_array := (others=> 0.0);
```

Usage:

- Stockage de données prédéfinies.

Remarque:

- Autre utilisation possible: assignation de plusieurs signaux en une instruction.
  (z_flag , n_flag , v_flag , c_flag) <= flag_reg;

Types

Tableaux – non contraints

Syntaxe

```
array((type range <>) {, ...})  of elements;
```

Exemples:

```
type sample is array (natural range <>) of integer;
variable short_sample_buf : sample(0 to 63);
```

Usage:

- Déclaration de nombreux tableaux identiques mais de tailles différentes.
- Facilite la lecture du code.

Remarque:

- Il faut **toujours** préciser la taille dans l'affectation, puisqu'elle est inconnue à priori.

Types

Tableaux - ports I/O

Exemple:

```
architecture behavioral of and_multiple is
begin
  and_reducer: process (i) is
    variable result : bit ;
  begin
    result := '1' ;
    for index in i'range loop
      result := result and i( index ) ;
    end loop ;
    y <= result ;
  end process and_reducer ;
end architecture behavioral ;
```

Usage:

- Création de blocs génériques, pouvant supporter différentes tailles d'entrées/sorties.

Records

Syntaxe

```
record
  ( identifier {, ...} : subtype_indication ; )
   {...}
end record [ identifier ]
```

Exemple:

```
type time_stamp is record
  seconds : integer range 0 to 59;
  minutes : integer range 0 to 59;
  hours   : integer range 0 to 23;
end record time_stamp;

variable sample_time : time_stamp := current_time;

sample_hour := sample_time.hours;
current_time.seconds := clock mod 60;
```

Usage:

- Définition de types plus complexes, similaire au `typedef` en C.

Assertions

Syntaxe

```
assert condition [report expression][severity expression]

type severity_level is (note, warning, error, failure);
```

Exemples:

```
assert buffer_size /= SIZEMAX
  report " buffer full "
  severity warning;
 ...
assert buffer_size <= SIZEMAX
  report " buffer error "
  severity failure ;    -- should not happen!
```

Usage:

- Affiche un message lors de la simulation, lorsque la condition évaluée est fausse.
- Debug, vérification.

Remarque:

- Par défaut, la sévérité vaut *error*.

Attributs

Un attribut fournit une information sur un élément:

- Entité,
- Architecture,
- Type,
- Tableau,
- ...

Les attributs disponibles dépendent de l'élément sur lequel ils portent.

Attributs

de types

Syntaxe:

`T'left`	valeur de gauche
`T'right`	valeur de droite
`T'low`	valeur minimale
`T'high`	valeur maximale
`T'ascending`	vrai si les valeurs de T sont croissantes
`T'image(x)`	chaine représentant la valeur de x
`T'value(s)`	la valeur de T représentée par s

Attributs

de tableaux

Syntaxe:

`A'left(N)`	borne gauche
`A'right(N)`	borne droite
`A'low(N)`	valeur minimale
`A'high(N)`	valeur maximale
`A'range(N)`	variation maximale d'index
`A'downrange(N)`	variation maximale d'index
`A'length(N)`	taille du tableau
`A'ascending`	vrai si les index de A sont croissantes
`A'element`	sous type des éléments

Remarque:

- *N* représente la dimension étudiée.

Attributs

de signaux

Syntaxe:

`S'delayed(T)`	signal S décalé de T
`S'stable(T)`	vrai si S a été stable pendant T
`S'quiet(T)`	vrai si S a été stable depuis T
`S'transaction`	vrai si il y a eu une transaction sur S
`S'event`	vrai si il y a un évènement sur S au pas de simulation présent
`S'active`	vrai si il y a une transaction sur S au pas de simulation présent
`S'last_event`	temps depuis le dernier évènement
`S'last_active`	temps depuis la dernière transaction
`S'last_value`	dernière valeur avant le dernier évènement

Remarque:

- Beaucoup de ces attributs sont utilisés pour des vérifications de *timing*.

Opérateurs

Syntaxe

```
and, or, nand, nor, xor, xnor, not      -- logiques
+, -, &                                 -- additifs
*, /, mod, rem                          -- multiplicatifs
abs, **                                 -- divers
<=, :=                                  -- assignation
=>                                      -- association
sll, srl, sla, sra, rol, ror            -- décalages
=, /=, <, <=, >, >=                     -- relations
```

Exemples:

```
X(1) <=   (a(3) and not(s(1)) and not (s(0))
        or (b(3) and not(s(1)) and s(0))
        or (c(3) and s(1) and not(s(0)))
        or (d(3) and s(1) and s(0));
 bit_vector <= bit_vector sll 2;
```

Remarques:

- Il n'y a pas de précédence des opérateurs en VHDL (pas de priorité par défaut).
- L'usage des parenthèses est donc obligatoire pour éviter des erreurs de compilation.

Subsection 2

Déclarations parallèles

Condition with-select-when

Syntaxe

```
with select_expression select
  nom <= {expression when value,}
           expression when value;
```

Exemple:

```
with S select
  O <= E0  when "00",
  O <= E1  when "01",
  O <= E2  when "10",
  O <= E3  when others;
```

Usage:

- Affectation d'un signal, soumise aux valeurs d'un signal de sélection.

Remarque:

- Modélise typiquement un multiplexeur.
- Les différents choix doivent être mutuellement exclusifs, et tous représentés!

Condition when-else

Syntaxe

nom <= {expression **when** condition **else**}
            expression ;

Exemple:

```
result  <= (a - b) when (mode = substract) else (a + b);
```

Usage:

- Assignation d'un signal, soumise à différentes conditions.

Attention

Les différentes conditions ne sont pas forcément mutuellement exclusives. Ainsi, si plusieurs conditions sont validées, la première rencontrée aura priorité. Dès lors, faites attention à l'écriture de votre code, qui pourrait être implémenté différemment que prévu dans ces cas particuliers (voir par exemple encodeur prioritaire)!
Voir [3] 167 - 172.

Subsection 3

Déclarations séquentielles

Condition if-then-else

Syntaxe

```
if condition then
  sequence_of_statements
{ elsif condition then
  sequence_of_statements}
[else
  sequence_of_statements]
end if;
```

Exemple:

```
    if  (S0 = '0' and S1 = '0') then O <= E0;
elsif  (S0 = '1' and S1 = '0') then O <= E1;
elsif  (S0 = '0' and S1 = '1') then O <= E2;
elsif  (S0 = '1' and S1 = '1') then O <= E3;
else O <= '-';
end if;
```

Usage:

- Choix simple ou série de choix simples

Condition case-when

Syntaxe

```
case expression is
  {when condition => sequence_of_statements}
end case;

condition <= identifier | expression | discrete_range | others
discrete_range <= expression (to|downto) expression
```

Exemples:

```
case S is
  when "00" => O <= E0;
  when "01" => O <= E1;
  when "10" => O <= E2;
  when "11" => O <= E3;
  when others => O <= '-';
end case;
```

```
case day is
  when Mon to Wed =>
    O <= '1';
  when Sun downto Fri =>
    O <= '0';
  when others =>
    O <= 'Z';
end case;
```

Usage:

- Choix multiples sur un signal.

Boucles

Syntaxe

```
[loop_label :] loop
  {sequence_of_statements}
end loop [loop_label];
```

```
[loop_label :] while condition
      loop
    {sequence_of_statements}
end loop [loop_label];
```

```
[loop_label :] for
      identificateur in
      discrete_range loop
   {sequence_of_statements}
end loop [loop_label];
```

discrete_range <= expression (**to**|**downto**) expression

exit [**when** condition];
next [**when** condition];

Usage

- Les différentes boucles peuvent toute effectuer la même chose,
- mais un choix judicieux du type de boucle permet un code plus concis et lisible.
- `exit` permet de quitter la boucle.
- `next` permet de passer à l'itération suivante, sans exécuter les instructions qui suivent.

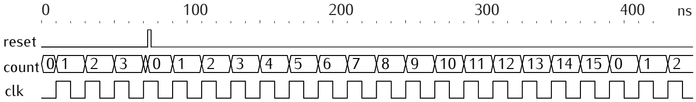
Boucles

Exemples

Compteur modulo 16 avec reset

```
Library IEEE;
Use     IEEE.std_logic_1164.all;
entity counter is
  port (clk, reset:   in std_logic;
        count:        out integer);
end entity counter;
```

```
architecture behavior of counter is
begin
  incrementer: process is
    variable count_value : integer := 0;
    begin
    count <= count_value;
      loop
        loop
          wait until clk = '1' or reset = '1';
           exit when reset = '1';
          count_value := (count_value + 1) mod 16;
          count <= count_value;
        end loop;
        count_value := 0;
        count <= count_value;
        wait until reset = '0';
      end loop;
    end process incrementer;
end architecture behavior;
```



Boucles

Exemples

Décodeur 3-8

```
Library IEEE;
Use     IEEE.std_logic_1164.all ;
entity dec38 is
  port( E0, E1, E2:   in  std_logic ;
        S:            out std_logic_vector (7 downto 0));
end entity dec38;

architecture behavior of dec38 is
begin
  process (E0, E1, E2)
  variable N : integer ;
  begin
    N := 0;
    if E0 = '1' then N := N+1; end if;
    if E1 = '1' then N := N+2; end if;
    if E2 = '1' then N := N+4; end if;

    S <= "00000000";
    for I in 0 to 7 loop
      if I = N then
        S(I) <= '1';
      end if ;
    end loop;
  end process;
end behavior;
```

Bibliographie I

P.J. Ashenden.
The Designer's Guide to VHDL.
Systems on Silicon Series. Morgan Kaufmann, 2008.

Steve Kilts.
Advanced FPGA Design: Architecture, Implementation, and Optimization.
Wiley-IEEE Press, 2007.

K. Skahill and J. Legenhausen.
VHDL for programmable logic.
Electrical engineering/digital desing. Addison-Wesley, 1996.