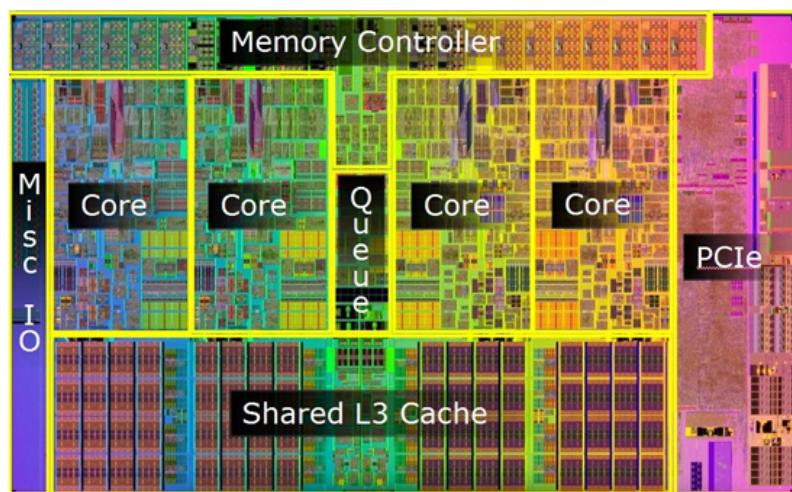


Hardware architectures for computer engineering

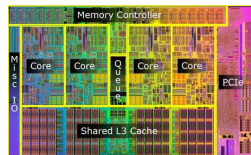
Reconfigurable Computing



Intel I5

Prof. Daniela Dragomirescu

Hardware architectures for computer engineering Reconfigurable Computing



Prof. Daniela Dragomirescu
INSA - DGEI

References

- ❖ VHDL - Du langage à la modélisation - R.Airiau et al. - Presses Polytechniques et Universitaires Romandes
- ❖ Digital Design and Modeling with VHDL and Synthesis - K.C. Chang - IEEE Press
- ❖ Advanced ASIC Chip Synthesis - H. Bhatnagar - Kluwer Academic Publisher
- ❖ Principles of CMOS VLSI Design: A system Perspective - N. Weste, K.Eshraghian - Addison Wesley

Student participation and discussion will be an essential part
of this course



Evolution of computing systems

- ❖ Embedded systems
- ❖ Mobile Network
- ❖ Internet of Things
- ❖ Factory of the future
- ❖ Smart cities

Hardware-Software
Systems



Overview

- ❖ 1. Introduction
 - * Hardware architecture - classical microprocessor modules
 - * High complexity
 - * Reconfigurability
 - * Evolution of design methodologies
 - * Design Flow for Digital Systems
- ❖ 2. Design of Computing Systems
 - * VHDL and digital synthesis
- ❖ 3. Study cases (lab classes):
 - * 5 pipe-line stages microprocessor design – RISC microprocessor
 - * C compiler for this microprocessor
 - * **Computer Science Project – Hardware and Software**

Overview

- ❖ 1. Introduction
- ❖ 2. VHDL language
 - * 2.1 Introduction
 - * 2.1.1 History
 - * 2.1.2 What is it a hardware description language?
 - * 2.1.3 Advantages and drawbacks of VHDL
 - * 2.1.4 IEEE Standard
 - * 2.2 Libraries
 - * 2.3 Design units
 - * 2.3.1 Entity
 - * 2.3.2 Architecture
 - * 2.3.3 Configuration
 - * 2.3.4 Package
 - * 2.4 Parallel and sequential field in VHDL
 - * 2.5 Test
 - * 2.6 Operators and words
 - * 2.7 Objects

Overview

- * 2.8 Types
 - * 2.8.1 Scalar types
 - ◆ Physicals Types, STD_LOGIC type
 - * 2.8.2 Complex types
 - ◆ Tables, Records, Agregats
 - * 2.8.3 Subtypes
- * 2.9 Signal and attribution of a signal
 - * 2.9.1 Unconditional attribution of a signal
 - * 2.9.2 Execution of a signal attribution
 - * 2.9.3 Conditional signal attribution
 - * 2.9.4 Selective signal attribution
 - * 2.9.5 Attributes
 - * 2.9.6 Distinction Rules between variable and signal
- * 2.10 Concurrent Instructions
 - * Process

Overview

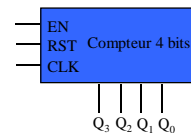
- * 2.11 Sequential instruction
 - * wait
 - * assert
 - * 2.12 Genericity
 - * 2.13 Compilation, elaboration, execution
 - * 2.14 Synthesys and Performances
- ❖ 3. FPGA – Field Programmable Gate Array

Introduction

- ❖ Hardware architecture - classical microprocessor modules
 - ❖ ALU
 - ❖ Data memory
 - ❖ Instruction memory
 - ❖ Pipe-line
 - ❖ DMA
 - ❖ MMU
- ❖ System on Chip – multi-processor – high complexity
- ❖ Network on Chip
- ❖ Reconfigurable computing

Introduction

- ❖ Evolution of design methodologies:
- ❖ Combinatory digital circuits
 - * their output is establish only by their courant entry
 - * logical gates, encoder, decoder, multiplexers
- ❖ Sequential digital circuits
 - * Their output is establish by the courant entry and by the previous state of the circuit. Internal **MEMORY** function.
 - * D Flip-flop, registres, RAM, counters
- ❖ Exercise : design a 4bits counter using D Flip-Flop and synchronic Sequential logic



Introduction Evolution of design methodologies

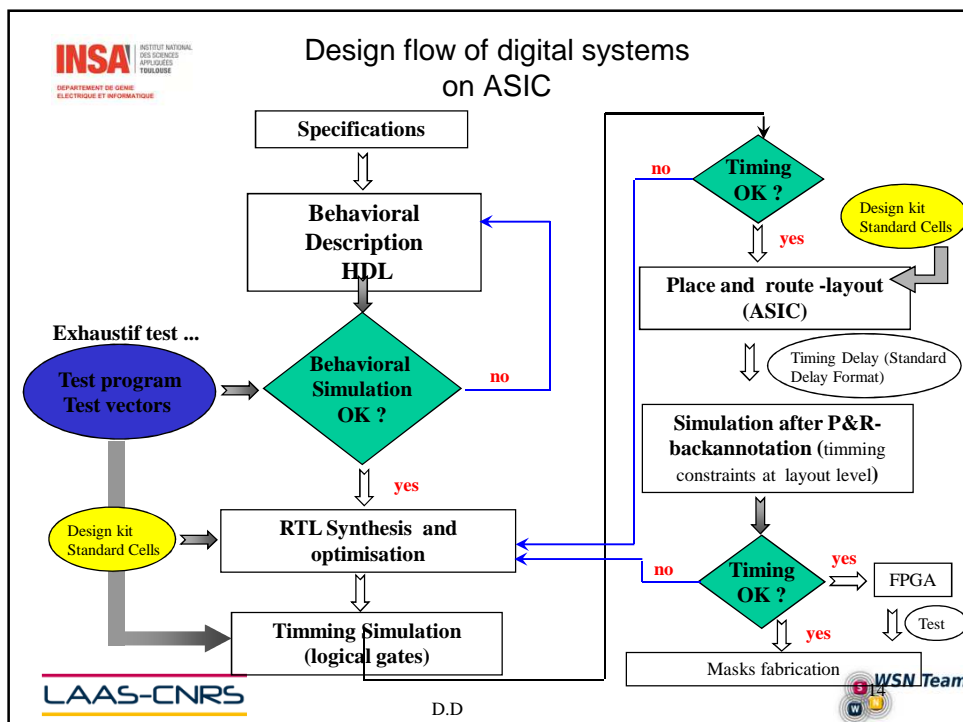
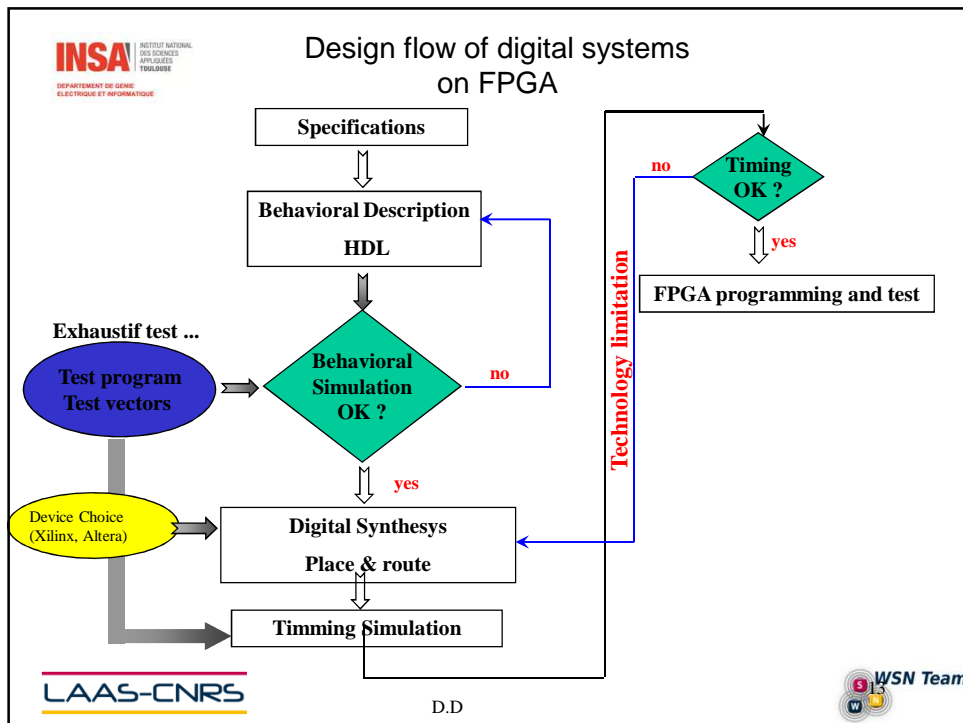
| Intel Processor | Product Date | Frequency | Nb. of transistors on chip |
|------------------|--------------|----------------|----------------------------|
| 8086 | 1978 | 8 MHz | 29 K |
| 80286 | 1982 | 12.5 MHz | 134 K |
| 80386 DX | 1985 | 20 MHz | 275 K |
| 80486 DX | 1989 | 25 MHz | 1.2 M |
| Pentium | 1993 | 60 MHz | 3.1 M |
| Pentium Pro | 1995 | 200 MHz | 5.5 M |
| Pentium II | 1997 | 266 MHz | 7 M |
| Pentium III | 1999 | 500 MHz | 8.2 M |
| Pentium III Xeon | 1999 | 700 MHz | 28 M |
| Pentium 4 | 2001 | 1.7 GHz | 42 M |

Introduction

- ❖ Evolution of design methodologies
 - ❖ HDL - Hardware Description Language
 - * Verilog
 - * VHDL
- } Same principle, distinct syntax
- ❖ Work with standard cells – basics cells (logical gates, registers) organized in libraries. Each foundry has its own library




Standard cells - design kit



2. Design of advanced digital systems VHDL language

2.1.1 History

- ❖ VHDL - VHSIC Hardware Description Language
VHSIC - Very High Speed Integrated Circuits
- ❖ 1980 - question of United States DoD to the necessity of a standard for software and for hardware systems
- ❖ Standards : ADA for the software and VHDL for the hardware
- ❖ VHDL –it is dedicated only to electronics? **NO !**

- ❖ VHDL describe hardware systems at a high level of abstraction
 - * Integrated circuits
 - * PCB – printed circuits board - ex: PC mother card
 - * Systems: software and hardware – ex: networks

2.1.2 What it is a hardware description language ?

- ❖ A hardware description language like VHDL can be used for :
 - * Specification
 - * Simulation
 - * Synthesis
 - * Formal Verification

2.1.3 Advantages and drawbacks of VHDL

- ❖ Advantages :
 - * VHDL is standardized language - IEEE standard
 - * The continuity of the language will be assured by the standard
 - * **Hierarchical design using entities**
 - * VHDL is a modern language:
 - * **High modularity**
 - * Safety to use
 - * reliable
 - * **The time is perfectly defined**
 - * different compilation unit
 - * strong types
 - * **genericity**
 - * **parallel language**
 - * using VHDL we minimize the risk of error on the silicon integrated circuits – the production cost will considerably decrease.
- ❖ Drawbacks:
 - * Simulation language ➡ every VHDL instruction can not be synthesized !

2.1.4 VHDL Standard

- ❖ IEEE Standard 1076 - 1987
- ❖ IEEE Standard 1076 - 1993
- ❖ IEEE Standard 1076 - 2003

2.2 Libraries

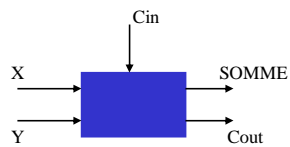
- ❖ Hierarchic language – small and hierarchic units
- ❖ Each design unit can be compiled alone
- ❖ Right VHDL description - Work library - **WORK**
 - * portability
- ❖ General models or utilities - resources libraries
 - * allow team work
- ❖ Libraries : IEEE , STD
 - * **package** IEEE.std_logic_1164 et **package** IEEE.std_logic_arith
 - * **package** STD.textio et **package** STD.STANDARD

Design units in VHDL

2.3 Design units 2.3.1 Entity

❖ Entity - VHDL model

* External view



```

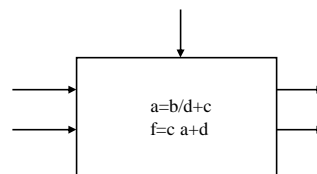
entity adder is
  port (
    X, Y, Cin: in bit;
    Somme, Cout: out bit);
end adder;
  
```

Ports

| FORMAL PORT CONNECTED OBJECT | MODE IN | MODE OUT | MODE INOUT | MODE BUFFER | MODE LINKAGE |
|---|------------|-------------|---------------|----------------|-----------------|
| Port of Mode in | oui | non | non | non | oui |
| Port of Mode out | non | oui | non | non | oui |
| Port of Mode inout | oui | oui | oui | non | oui |
| Port of Mode buffer | oui | non | non | oui | oui |
| Port of Mode linkage | non | non | non | non | oui |
| Local signal | oui | oui | oui | oui | oui |
| Key word open (not-connected) | non | oui | oui | oui | oui |

2.3 Design units 2.3.2 Architecture

- ❖ Inside view of an entity - **ARCHITECTURE**



- ❖ Architecture description styles :
 - * **Structural** description : components interconnection
 - * **Data flow** description: equation described behavior
 - * **Behavioral** Description : algorithmic behavior
- ❖ We can combine different description styles in the same architecture

2.3.2 Architecture

❖ Structural description of the adder

❖ architecture struct of adder is

component half_adder

port(I1,I2: in bit;
Carry: out bit;
Sum: out bit);

end component;

component gate_OR

port(I1, I2: in bit;
O: out bit);

end component;

signal a,b,c : bit;

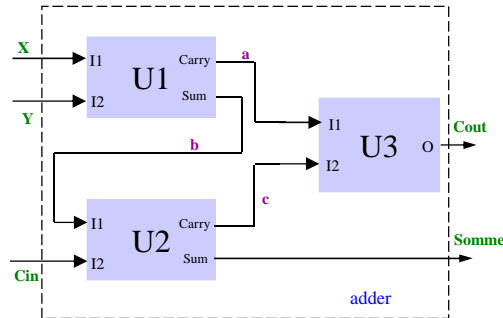
begin

U1: half_adder port map(X,Y,a,b);

U3: gate_OR port map(a,c,Cout);

U2: half_adder port map(b,Cin,c,Somme);

end struct;



2.3.2 Architecture

❖ Data flow architecture of the adder

$$S = X \oplus Y$$

$$\text{Somme} = S \oplus \text{Cin}$$

$$\text{Cout} = XY + SCin$$

Booleans equations of the
adder

architecture data_flow of adder is

signal S: bit;

begin

Somme <= S xor Cin ; -- after 1 ns

S <= X xor Y ; -- after 1 ns

Cout <= (X and Y) or (S and Cin) ; -- after 2 ns

end data_flow;

2.3.2 Architecture

❖ Behavioral architecture of the adder

* Represented by a verity table IN THIS PARTICULAR CASE

architecture beh of adder is

begin

process -- concurrent instruction

variable N: integer;

constant sum_vector: bit_vector(0 to 3):= "0101";

constant carry_vector: bit_vector(0 to 3):= "0011";

begin

N:=0;

if X= '1' then N:=N+1; end if;

if Y= '1' then N:=N+1; end if;

if Cin='1' then N:=N+1; end if;

Somme <= sum_vector(N);

Cout <= carry_vector(N);

wait on X, Y, Cin;

end process;

end beh;

| X | Y | C _{in} | Sum | C _{out} |
|---|---|-----------------|-----|------------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

architecture beh of adder is

Signal N : integer;

begin

process -- concurrent instruction

constant sum_vector: bit_vector(0 to 3):= "0101";

constant carry_vector: bit_vector(0 to 3):= "0011";

begin

N<=0; --N=3

if X= '1' then N<=N+1; end if; -- driver du signal N passera à 1

if Y= '1' then N<=N+2; end if; -- driver du signal N passera à 0+2

if Cin='1' then N<=N+3; end if; -- driver du signal N passera à 0+3

Somme <= sum_vector(N);-- N=0

Cout <= carry_vector(N);

wait on X, Y, Cin;

end process; -- N=3 N=5

end beh;

2.3.3 Configuration

- ❖ Give the correspondance between the component and its VHDL model from the libraries
- ❖ It allows to link the components used in an architecture and their effective design

Ex: Entity « three_registres » with an architecture « beh » which has 3 registre R1, R2 et R3

```
1. use work.three_registres;
   configuration alpha of trois_registres is
     for beh
       for R1,R2 : registre_8bit use entity work.registre_8bit(arch);
       for R3 : registre_8bit use entity work.registre_8bit(shift);
     end for;
   end alpha;
```

```
2. FOR ALL : registre_8bit use entity work.registre_8bit(arch);
```

Exemple : adder

```
entity adder is
  port (X, Y, Cin : in bit;
        Sum, Cout : out bit);
end adder;
architecture struct of adder is
  component half_adder
    port ( I1,I2:      in bit;
          Carry:      out bit;
          Sum:        out bit);
  end component;
  component gate_OR
    port ( I1, I2:      in bit;
          O:          out bit);
  end component;
  for all : half_adder use entity syst_ing.half_adder(beh);
  for all: gate_OR use entity work.porte_ou(struct);
  signal a,b,c : bit;
  begin
    U1: half_adder port map(X,Y,a,b);
    U2: half_adder port map(b,Cin,c,Sum);
    U3: gate_OR   port map(a,c,Cout);
  end struct;
```

```
library syst_ing;
```

```
for all : half_adder use entity syst_ing.half_adder(beh);
```

2.3.4 The package

- ❖ It contains algorithms, functions, new types, subtypes, objects: signals and constants (no variables).
- ❖ One or more packages in the same library
- ❖ 2 design units :
 - * Package specification – external view
 - * Presents all the things exported by the package (algorithms, objects, types)
 - * **Package body** – internal view - *optional*
 - * contain the description of algorithms
- ❖ To access a package you have to use the key word **use**
 - * **library** IEEE;
 - use** IEEE.std_logic_1164.all ;
 - * **library** my_lib;
 - use** my_lib.4GSI.all;

2.3.4 The package

- ❖ Specification of a package :

```

package SIMPLE is

  constant TAILLE_MAX: integer :=1024;

  subtype my_integer is INTEGER range 0 to TAILLE_MAX;

  signal addition_10bits : my_integer ;

  function MIN(A,B:INTEGER) return INTEGER;
  function MAX(A,B:INTEGER) return INTEGER;

end SIMPLE;
  
```

- * The objects declared in the package specification will be known and can be used in the package body.

2.3.4 The package

❖ Package body

```
package body SIMPLE is
  function MIN(A,B:INTEGER) return INTEGER is
  begin
    if A<B then return A;
    else return B;
    end if;
  end MIN;
  function MAX(A,B:INTEGER) return INTEGER is
  begin
    if B<A then return A;
    else return B;
    end if;
  end MAX;
end SIMPLE;
```

2.4 Parallel and sequential field in VHDL

- ❖ The description of a hardware system is naturally parallel
- ❖ The VHDL description will be concurrent
- ❖ The concurrent and sequential field co-existent in VHDL
- ❖ Concurrent field
 - * The entities
 - * The body of the architectures
- ❖ Sequential field
 - * The body of process
 - * The body of packages

2.5 TEST

- ❖ To **verify** (simulate) the behavior of circuits described in VHDL we have to **test** them.
- ❖ The test program is a VHDL program, with the same structure like any usual VHDL program: entity, architecture
 - * We apply the test vectors on the inputs and the simulation give us the behavior of the outputs of the device under test (DUT)
- ❖ If possible - exhaustive test
 - * Difficult to set up for the complex systems
 - * Formal verification

2.5 TEST

❖ Exemple: adder test

```
entity test_adder is
end test_adder;
architecture bench of test_adder is
  COMPONENT adder is
    port (X, Y, Cin : in bit;
          Somme, Cout : out bit);
  END COMPONENT;

  For all : adder use entity work.adder(data_flow);

  SIGNAL data1, data2, data3 :bit;
  SIGNAL dataout, carry_out : bit;

  BEGIN
    additionneur: adder PORT MAP (data1,data2, data3,dataout, carry_out);
    data1 <= '0', '1' after 30 ns;
    data2 <= '1', '0' after 50 ns, '1' after 60 ns;
    data3 <= '0', '1' after 12 ns;

  END bench;
```

Test non-exhaustif

2.6 Operators and words

❖ Operator classes by increasing priority

1. Logics : and or nand nor xor;
- defined on Boolean and BIT type
2. Relational: = /= < <= > >=
- defined on all types with the exception of FILE type
3. Math and concatenation: + - &
4. Signe: + -
- the signe operators are less important than the multiplication operators !!!!!
5. Multiplication: * / mod rem
6. Power, module, complément: ** abs not
- ** the power has to be integer

❖ It is possible to overcharge the operators. This will not change their priorities.

2.6 Operators and words

❖ Classifications:

- * number - Decimal - integer (1345) ou real (13.0)
- Binary '0', '1', "1010"
- Hexadecimal : X"10AB"
- * character and words : 'a', "Bonjour"
- * Bits vectors "1010"
- * null

❖ OBS: 1 - integer '1' - bit

❖ Expressions – has a type and get a value at the execution time

❖ Identifiers – starts with a character; no difference between capital and minor characters

2.7 The objects

- ❖ The objects contains values.
- ❖ There are 4 classes of objects in VHDL : **constants**, **variables**, **files** and **signals**.
- ❖ The constants has a fixed value
- ❖ The variables has a changeable value
- ❖ The files contains sequences of values which can be read or write.
- ❖ The **signals** keep the history of the passed values and the actual value and the values calculated for the future : only the future values can be modified by signal attribution.
- ❖ All the objects have a type

2.8 Types

- ❖ VHDL is a strong type language
- ❖ A type is a collection of regulated values
- ❖ The type is **static** : it can't be modified
- ❖ It is possible to define new types using the pre – defined types and the type constructors
- ❖ Classification :
 - * Scalar type
 - * Complex type
 - * **Access** type (pointer) - only the **variables** can be of access type
 - * File type – key word **file**

2.8.1 Scalar types

❖ Integers

* subtype my_integer is INTEGER range -65_536 to 65_535 ;

❖ Real

* subtype my_flottant is REAL range 5.36 downto 2.15;

❖ Enumerated types

* type Color is (YELLOW, GREEN, RED);

* type BOOLEAN is (FALSE, TRUE);

* type LOGIC4 is ('X', '0', '1', 'Z');

❖ Physical types

* TIME – defined in the package STANDARD

* TIME –the time known by the simulator

* There are characterized by the base unit, the authorized values, the collection of their units and their correspondence.

2.8.1 Physical type

❖ TIME

type TIME is range -9_223_372_036_854_775_808 to
9_223_372_036_854_775_807

-- coded on 64 bits ;

units fs ;

ps = 1000 fs;

ms = 1000 us;

ns = 1000 ps;

sec = 1000 ms;

us = 1000 ns;

min = 60 sec ;

hr = 60 min;

end units;

❖ DISTANCE

type DISTANCE is range 0 to 1E16

units A ;

nm = 10 A;

um = 1000 nm;

mm = 1000 um;

cm = 10 mm;

m = 1000 mm;

km = 1000 m;

end units;

2.8.1 Type STD_LOGIC

- ❖ It is defined in the package `IEEE.std_logic_1164`
- ❖ Package `IEEE.std_logic_unsigned`
- ❖ Package `IEEE.std_logic_arith`
- ❖ Replace the type BIT ; it is called resolved BIT
- ❖ It has 5 values:
 - * '1' - 1 logic
 - * '0' - 0 logic
 - * 'Z' – high impedance
 - * 'U' - not-initialized
 - * 'X' - indeterminate
- ❖ STD_LOGIC_VECTOR
 - * **signal** A : std_logic_vector(0 to 23);

2.8.2 Complex types

- ❖ Classification :
 - * Tables : collection of objects having same type - **array**
 - * Articles : collection of objects having different types - **record**
- ❖ Tables
 - * The elements of a table are called by their position number
 - * Constraints Tables
 - * **type** MOT **is array** (0 to 31) **of** STD_LOGIC;
 - * Non-constraints Tables
 - * **type** STD_LOGIC_VECTOR **is array** (NATURAL **range <>**) **of** STD_LOGIC;
 - * **Index definition at the execution time**
 - ◆ **signal** bus : STD_LOGIC_VECTOR (63 **downto** 0);
 - * Tables attributes :
 - LEFT, RIGHT, HIGH, LOW, RANGE, REVERSE_RANGE, LENGTH

2.8.2. Tables - Attributes

❖ Example :

subtype INDEX 1 is INTEGER range 1 to 20;
 subtype INDEX2 is INTEGER range 19 downto 2;
 type VECTEUR1 is array (INDEX1) of STD_LOGIC;
 type VECTEUR2 is array (INDEX2) of STD_LOGIC;

- * VECTEUR1'LEFT will give 1 et VECTEUR2'RIGHT will give 2
- * VECTEUR1'HIGH will give 20 et VECTEUR2'HIGH will give 19
- * VECTEUR1'LOW will give 1 et VECTEUR2'LOW will give 2
- * VECTEUR1'RANGE will give 1 to 20 – used for the loops
- * VECTEUR1'REVERSE_RANGE will give 20 downto 1
- * VECTEUR1'LENGTH will give the numbers of tables elements – in this case: 20

2.8.2 Complex types - ARTICLES

❖ Articles – key word **record** - **end record**

- * It is formed by elements of different types
- * Its elements are called by their **name**
- * Example :

```

* type BIT_COMPLET is
  record
    value : std_logic;
    fan_out_min, fan_out_typ, fan_out_max: integer;
  end record;
signal A : BIT_COMPLET;
variable B : BIT_COMPLET;
```

➡ Then **A.value** is of **std_logic** type, **A.fan_out_typ** is of integer type

* Attribution

```

A <= B ;
or
A.value <= B.value;
```

2.8.2 Aggregates

- ❖ An aggregate represents the mode to indicate the value of a complex type
- ❖ An aggregate is marked between brackets, the elements are separated by commas
- ❖ The compiler verifies the type
- ❖ Example :

* type TAB is array (1 to 3) of INTEGER ;

* type ART is record

Field1 : NATURAL;

Field2 : STD_LOGIC;

Field3 : NATURAL;

end record;

* signal A : TAB; signal B : ART;

- ❖ 3 different notations of the aggregate :

* Position notation

A <= (12, 13, 14);

B <= (5, '0', 15);

* Name notation

A <= (1=>12, 2=>13, 3=>14); B <= (Field3 =>15, Field1 => 5, Field2 =>'0');

* Mixte (position and name) notation

A <= (5, others =>0);

B <= (Field2 =>'0', others =>0);

A <= (others =>15);

2.8.3 Subtypes

- ❖ Sub type declaration when we want/need to decrease the range of a type.
- ❖ A subtype is always compatible with its base type.
- ❖ Examples:

* subtype integer_8bits is INTEGER range 0 to 255 ;

* subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

- ❖ Dynamic Subtypes:

* Contains expressions which will be calculated at the simulation time

Subtype MOT is STD_LOGIC_VECTOR (1 to MAX);

where MAX is a parameter of a sub-program or a generic parameter of an entity.

2.9 Signal and signal attribution

- ❖ A signal corresponds to the hardware representation of the information support in VHDL.
- ❖ A signal has a time evolution (a variable - non)
- ❖ The signal declaration has to be done in the concurrent field – declaration area of architectures and packages
- ❖ All the ports of an entity are signals
- ❖ The signal driver – each signal can remember its past values, if these will be used in other place, each signal know its present value and the values calculated for the future.
- ❖ Signal attribution :
 - * connection to an output port of a device
 - * signal attribution in the concurrent field
 - * signal attribution in the sequential field

2.9.1 Unconditional attribution of a signal

$S \leq '0', '1'$ after 20 ns;

- ❖ The value has to have a compatible type with the signal declared type.
- ❖ Each signal attribution has a value element and an delay element of type physique TIME (delay null by default without after clause)

$S \leq A$ after 10 ns ;

- ❖ All the signals used in the expression are locals signals or inputs ports

2.9.2 Execution of a signal attribution

- ❖ The signal attribution doesn't change the current value, but it modifies the futures values which the signal can take.

$A \leq B$; or $A \leq B$ after 0 ns;

- ❖ The A signal take the current value of B signal after a delta delay
- ❖ Delta delay – it is a delay which is null for the simulation but it represent the relations cause to effect.

2.9.2 Execution of a signal attribution

$S \leq A+B$ after 20 ns;

- ❖ The expression attributed to a signal is evaluated at each change of the inputs signals
- ❖ Each value calculated by the expression is stocked in the destination signal with its apparition delay.

2.9.3 Conditional attribution of signal

❖ Simple Condition:

```
* S <= A when condition else
    B;                                Concurrent conditional Instruction
* S <= A after 20 ns when condition else
    B after 10 ns;
```

❖ Multiple Condition with precedence order

```
* S <= 5 when A='1' and B='1' else
    4 when A='1' else           Concurrent conditional Instruction
    0;
```

❖ When one of the input signal change its value, all the conditions are evaluated in order (from left to right).

❖ The first value given by the first expression which has the true condition will be attributed to the signal.

2.9.4 Selective attribution of the signal

❖ One unique condition will establish which expression will be attributed to the signal :

```
❖ type op_bit is (et, ou, non) ;
    signal opcode: op_bit;
    signal result, A, B, : std_logic;
    .....
    with opcode select          -- Concurrent selective instruction
        result <= A and B when et,
                A or B when ou,
                not A when non,
                '0' when others;
```

❖ Selective attribution models components like multiplexers or decoders.

2.9.5 Attributes

- ❖ The attribute is a property associated with a type or an object
- ❖ the user can define new attributes
- ❖ Predefined :
 - ❖ S'quiet(T) - TRUE if the signal S don't change during T
 - ❖ S'stable(T) -TRUE if no event appear on the signal S from the time T
 - ❖ S'delayed(T) -signal S delayed by T after NOW
- ❖ S'event - TRUE if one event just happened on S.
- ❖ S'last_value – last value of signal S before the last event
- ❖ S'last_event – the time value after the last event which happened on signal S

2.9.6 Distinction rules between variables and signals

- ❖ All the variables will be declared in the sequential field: in the declarative part of process or in the sub-programs
- ❖ The variable are used and attributed only in the sequential field
a:=1'
- ❖ The signals will be declared in the concurrent filed : in the declarative part of architectures, the ports of the entities, the package specification
- ❖ The signals are used in both sequential and concurrent field
a <='1'
- ❖ A variable attribution is done immediately, or a signal attribution is delayed after the evaluation of all signal attribution.

2.10 Concurrentes Instructions

- ❖ We don't simulated in real time – we simulate in virtual time – simulation time
- ❖ Theses functioning will be described be concurrent instructions executed in asynchronous manner
 - * Signal attribution (inconditional, conditional , selective)
 - * Component instantiation
 - * **Process instruction**
 - * Sub-programs call
 - * Assertion instruction
 - * Instruction **generate**


2.10 Concurrentes instructions

- ❖ **Process**
 - * Is the main object use by the simulator
 - * Each concurrent instruction can be translated in its equivalent process
 - * A process is sensible to signals
 - * A process contains only sequential instructions
 - * The lifetime of the process is the lifetime of the simulation : a process is cyclic

2.10 Concurrent instructions - Process

```
{label: } process { list_of_supervised_signals}
  Declarations
  begin
    Sequential Instructions
  end process {label};
```

Sensitivity list



- * The declaration will be elaborated only once, at the initialization.
- * A process is executed at least once at the initialization till the instruction **WAIT** (if it exists)
- ❖ At each event on a supervised signal the process will be executed
- ❖ Restrictions:
 - * Instruction **wait** stop the process – see more information on the instruction **wait** au chapter 2.11 Sequential Instructions
 - * The instruction **wait** can't be used in a process with sensitivity list

2.10 Concurrent Instructions - Process

Process with sensitivity list :

```
{label: } process {list_of_supervised_signals}
  Declarations
  begin
    Sequential Instructions
  end process {label};
```

- * This process will be executed once at initialization

Process without sensitivity list :

```
{label: } process
  Declarations
  begin
    wait on {supervised_signals}
    Sequential Instructions
  end process {label};
```

- * This process will not be executed at initialization

2.11 Sequential instructions

- ❖ The sequential instructions are used in the process body or in sub-programs

- * Instruction **WAIT**
- * Instruction **ASSERT**
- * Variables and signal attribution
- * Sub-programs call
- * Conditional instructions
- * Control instructions
- * Null Instruction - **null**

2.11 Sequential instructions

- ❖ Instruction **WAIT**

- * The execution of process can be suspended depending on a condition and for a time indicated by the instruction **wait**

- * **wait** { **on** signals_list } { **until** boolean_condition } { **for** time};

- * Each event on a signal from the supervised signals list will conduct to the evaluation of boolean condition.

- * Exemple:

```

process
begin
    wait on CLK;
    if clk'event and CLK='1' then
        q <=d;
    end if;
end process;

process
begin
    wait until CLK'event and CLK ='1';
    q <=d;
end process;

```

2.11 Sequential instructions

❖ Instruction **ASSERT**

- * Supervised a condition and will print a message if the condition is false.
- * The execution of the program starts immediately after the assert instruction.
- * **assert** condition {**report** msg} {**severity** level}
- * **assert** NOW<1 min report "Fin simu" severity ERROR;
- * Type Severity_Level is (note, warning, error, faillure);

2.11 Sequential instructions

❖ Attribution Instructions of variables and signals

- * Variable1 **:=** 2;
- * Signal1 **<=** 2 **after** 20 ns;

❖ Sub-program call

- * **procedure** alfa (nr:**integer**, msg:**string**) –sub-program definition
- * Alfa(4, "GO"); --positional call
- * Alfa (nr =>4, msg =>"GO");

❖ Instruction **return**

- * Reserved to sub-programs

❖ Instruction null

- * **null**
- * The execution goes to next code line
- * The instruction null is not necessary to the compilation of empty process or empty sub-programs

2.11 Sequential instructions

❖ Conditionals Instructions

```
* if condition1 then instructions1
  elsif condition2 then instructions2
  else instructions3
  end if;
```

```
* Choice instructions:
  case expression is
  when val1 => instructions1
  when val2 => instructions2
  when others => instr3
  end case;
```

2.11 Sequential instructions

❖ Loop instructions

```
* loop
  * Sequential Instructions
  end loop;
```

```
* for indices in range loop
  * Sequential instructions
  end loop;
```

```
* while condition loop
  * Sequential Instructions
  end loop;
```

```
* next loop_label when condition --stops the current iteration
```

```
* exit loop_label when condition
```

Bascule D

```
entity basculeD is
port(
  D, CLK: in std_logic;
  Q: inout std_logic;
  Qb: out std_logic);
end basculeD;
architecture beh1 of basculeD is
begin
  process
  begin
    wait until clk'event and clk='1';
    Q <= d;
    Qb <= not Q;
  end process;
end beh1 ;
```



LAAS-CNRS

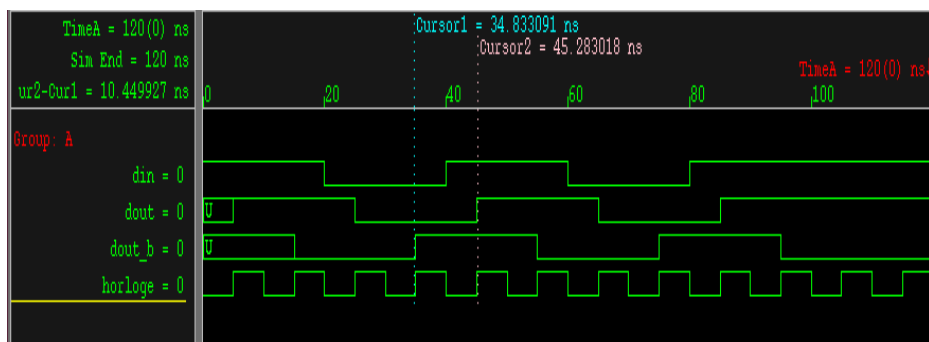
D.D

```
entity basculeD is
port(
  D, CLK: in std_logic;
  Q: inout std_logic;
  Qb: out std_logic);
end basculeD;
architecture beh2 of basculeD is
Begin
  Qb <= not Q;
  process
  begin
    wait until clk'event and clk='1';
    Q <= d;
  end process;
end beh2 ;
```



WSN Team

Bascule D - architecture beh1

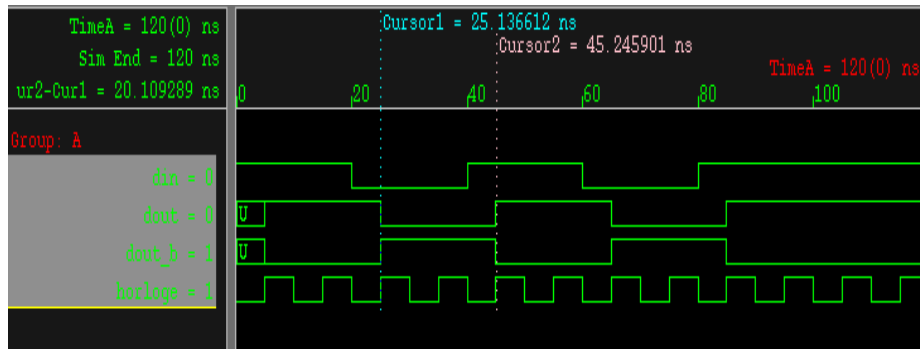


LAAS-CNRS

D.D

WSN Team

Bascule D - architecture beh2

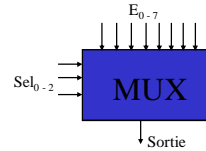


Bascule D

```
entity basculeD is
  port(
    D, CLK: in std_logic;
    Q: out std_logic;
    Qb: out std_logic);
end basculeD;

architecture beh3 of basculeD is
  Begin
    process
    begin
      wait until clk'event and clk='1';
      Q<= D ;
      Qb <= not D;
    end process;
  end beh3 ;
```

Multiplexer 8 bits (3 selection inputs)



```

entity mux is
    port (
        sel : in integer range 0 to 7 ;
        entree : in std_logic_vector(0 to 7);
        Sortie : out std_logic);
    end mux;

Architecture beh of mux is
begin
    with sel select
        sortie <= entree(0) when 0,
                  entree(1) when 1,
                  entree(2) when 2,
                  entree(3) when 3,
                  entree(4) when 4,
                  entree(5) when 5,
                  entree(6) when 6,
                  entree(7) when 7;
    end beh;
end mux;

```

2.12 Genericity

- ❖ The genericity allow to define a family of components reported to a property of the entity (numbers of entries, numbers of outputs)
- ❖ The property value is fixed at the instantiation of the entity

```

* entity Porte_et is
generic (nombre_entrees: Natural:=2);
port (A: in STD_LOGIC_VECTOR (1 to nombre_entrees);
      B: out STD_LOGIC);
end Porte_et;

```

- ❁ Component instantiation :

* Porte_et_4entrees : Porte_et **generic map** (nombre_entrees => 4)
port map (A=> alfa, B=>beta);

was already defined :

- ★ Signal alfa : std_logic_vector (1 to 4);

```
★ Signal beta : std_logic;
```

2.12 Genericity

Concurrent Instruction **generate**

- ❖ Allow the iterative or conditional elaboration of codes lines
- ❖ 2 forms : conditional and iterative
- ❖ Conditional form
 - * label : **if** boolean_condition **generate**
 concurrent instructions
 end generate {label};
- ❖ Iterative Forme
 - * **for** name_of_generation_parameter **in** range **generate**
 Concurrent instructions
 end generate {label};

2.12 Genericity

- ❖ Example: **Generic (N)** shift register
 - * inputs Data et Clock - std_logic;
 - * output S - std_logic_vector (N-1 downto 0)
 - * structural architecture using a D Flip Flop component
- ```

entity reg_decalage is
 GENERIC(Nb_bits: natural :=8);
 port (
 Data, clock : in std_logic;
 S : out std_logic_vector(Nb_bits-1 downto 0));
end reg_decalage;
```

## 2.12 Généricité

**ARCHITECTURE** structurelle **OF** reg\_decalage **IS**

```
COMPONENT basculeD PORT (
 D, clk : in STD_logic;
 Q : OUT STD_LOGIC ;
 Qb : out std_logic);
```

**END COMPONENT**;

**for all** : basculeD **use entity** work.basculeD(beh3);

**begin**

gauche : basculeD **port map** (data, clock, S(Nb\_bits-1), **open**);

```
boucle : for i IN 1 to Nb_bits-1 generate
 circ : basculeD PORT MAP (S(Nb_bits-i), clock, S(Nb_bits-i-1), open);
END GENERATE boucle;
```

**END** structurelle ;

## 2.12 Généricité

**ARCHITECTURE** structurelle **OF** reg\_decalage **IS**

```
COMPONENT basculeD PORT (
 D, clk : in STD_logic;
 Q : OUT STD_LOGIC ;
 Qb : out std_logic);
```

**END COMPONENT**;

signal aux : std\_logic\_vector (Nb\_bits-1 downto 0);

**for all** : basculeD **use entity** work.basculeD(beh3);

**begin**

gauche : basculeD **port map** (data, clock, aux(Nb\_bits-1), **open**);

```
boucle : for i IN 1 to Nb_bits-1 generate
 circ : basculeD PORT MAP (aux(Nb_bits-i), clock, aux(Nb_bits-i-1), open);
END GENERATE boucle;
```

S <= aux ;

**END** structurelle ;



## 2.12 Généricité

**ARCHITECTURE** beh **OF** reg\_decalage **IS**

**begin**

**comport** : **process**

**begin**

**wait until** clock'event and clock ='1';

s(Nb\_bits-1) <= Data ;

**copie** : **For** i **IN** Nb\_bits-1 **downto** 1 **LOOP**

s(Nb\_bits-i-1) <= s(Nb\_bits-i);

**end loop** copie;

**end process** comport ;

**END** beh;

## 2.12 Généricité

**ARCHITECTURE** beh **OF** reg\_decalage **IS**

**signal** aux : std\_logic\_vector (Nb\_bits-1 downto 0);

**begin**

s <= aux ;

**comport** : **process**

**begin**

**wait until** clock'event and clock ='1';

aux(nb\_bits-1) <= Data ;

**copie** : **For** i **IN** Nb\_bits-1 **to** 1 **LOOP**

aux(Nb\_bits-i-1) <= aux(nb\_bits-i);

**end loop** copie;

**end process** comport ;

**END** beh;

## 2.13. Compilation, elaboration, execution, exploitation

### ❖ VHDL utilization :

- \* compilation - static aspects of the description;
- \* elaboration - dynamic aspects of the description;
  - do the instantiation of VHDL description;
  - if the structure is hierarchical you have to start with the higher level
- \* execution - for a simulator  $\Leftrightarrow$  simulation
  - for a synthesizer  $\Leftrightarrow$  synthesis
- \* exploitation of results from simulation and from synthesis

## 2.14. Synthesis

- ❖ It is a process of translation :
  - \* input : behavioral description
  - \* output : structural description using basic elements predefined - **standard cells**
- ❖ Respect of the original functionality
- ❖ Respect of time, area, power constraints
- ❖ Physical representation :
  - \* ASIC
  - \* FPGA

## 2.14. VHDL and the synthesis

### ❖ Strong Points :

- \* genericity. The field is in rapid evolution. The generality of the VHDL language and its abstraction capacity show that it is able to support this evolution.
- \* strong link with the simulation

### ❖ Weak Points :

- \* there is not a subset of VHDL standardized for the synthesis
- \* the semantics of VHDL for the simulation is standardized, but not the VHDL semantics for synthesis !

## Synthesis using VHDL

### ❖ Two ways :

- \* Logic synthesis - industry
  - \* behavioral specification at **RTL ( Register Transfer Level) level**
  - \* library of hardware elements – design kit – standard cells
- \* Architectural synthesis – still on research field
  - \* abstract behavioral specifications ( loops, pointers)
  - \* compilation, allocation and resources partitioning

## 2.14. Logic Synthesis

- ❖ Sub-set of VHDL for the synthesis:
  - \* **synchronous** descriptions (evident clocks)
  - \* delay expressions are ignored (clauses **after**)
  - \* restrictions of a process writing
  - \* only some types are allowed
- ❖ The detail of restrictions varies from one synthesis software to another
- ❖ We will present the minimal restrictions
- ❖ The configuration is not supported for the synthesis. The name of the "component" = name of VHDL model in the library

## 2.14. Types for logic synthesis

- ❖ enumerated
- ❖ type BIT and associated operations
- ❖ types from STD\_LOGIC package
- ❖ integers :
  - \* the variation range will determine the number of necessary bits
  - \* the bits can not be access individually
- ❖ Types non synthesizable
  - \* REAL, ACCES, FILE
  - \* Types physiques: TIME or others physical types defined by the user

## 2.14. Synthesis - Remarque

- ❖ The use of enumerated types are recommended : the VHDL code is small and we leave to the synthesis tool the choice of the good code strategy.
- ❖ There is no consensus on the code of the enumerated types
- ❖ The tables of more than one dimension can not be synthesized because of the difficulty of addresses calculation.
- ❖ The types defined in package IEEE\_1164 are strongly recommended

## 2.14. Objects and logical synthesis

- ❖ Constants
  - \* Accepted for all the synthesizable types. Their declaration doesn't produce any hardware. Their use will produce hardware in the following cases:
    - \* right side in the signal attribution
    - \* In a instruction **if** or **case**
    - \* In a concurrent conditional instruction
- ❖ Signals
  - \* wires or bus
- ❖ Variables (variable attribution)
  - \* there is NOT general rule to know the hardware obtained for a variable.
  - \* the use context of the variable will be determinant

## 2.14. Initial values

- ❖ In VHDL 3 types of initial values:
  - \* Default Values inherited from the type definition
  - \* Explicit initialization at the declaration of the object
  - \* Value attributed by instruction at the begin of the process
  
- ❖ Synthesis – the 2 first cases are ignored by the synthesis tools.

## Generic Parameters

- ❖ The genericity allow to define a family of components reported to a property of the entity ( numbers of entries, numbers of outputs)
  
- ❖ The value is fixed at the instantiation of the entity.
  
- ❖ Synthesis :
  - \* The type of generic parameter which can be synthesize is depending upon the synthesis tool.

## 2.14. Sequential Instructions and logical synthesis

- ❖ Synchronization Instruction :
  - \* a signal from the sensitivity list (process ( A,B,C) or wait on A,B,C )
    - ★ combinatory hardware
  - \* on signal recognized like clock (clk '**event**' or clk='1').
    - ★ Only the transition from '0' to '1' or the inverse are recognized
    - ★ Sequential hardware
- ❖ Iteration (loops) Instructions
  - \* **for** : is synthesizable if the variation range is static
  - \* **while** : non-synthesizable at RTL level
- ❖ Sub-programs call
  - \* Some function are known by the synthesis tool ( adder, incrementer) – no additional hardware
  - \* functions with parameters pass by values - combinatory hardware
  - \* functions with parameters pass by pointers - non-synthesizable at RTL level

## 2.14. Process for logic synthesis

- ❖ Main question : combinatory or sequential hardware ?
- ❖ Combinatory process if :
  - \* Sensitivity list or only one synchronization point (wait)
  - \* All the signal read are in the sensitivity list
  - \* No variable declaration or locales variables systematically attributed before to be read
  - \* **All the signals written are attributed on all the branches.**
- ❖ If memory – sequential process
  - \* synchronic style - DFlipFlop

## 2.14. Concurrent Instructions and logic synthesis

- ❖ Unconditional signal attribution – combinatory hardware
- ❖ Conditional or selective signal attribution
  - \*  $S \leq A$  when  $X = '1'$  else  $B$  when  $Y = '1'$  else  $C$  - **combinatory HW**
  - \*  $S \leq A$  when  $X = '1'$  else  $B$  when  $Y = '1'$  else  $S$  - **sequential HW**
- ❖ Instantiation (of components, generate)
  - \* The concept of entity and of component allows to have a hierarchy in the system description.
  - \* The configuration – is not supported by the synthesis.
    - ★ The name of the component and its port has to be the same like in the entity

## 2.14. Architectural synthesis

- ❖ The architectural synthesis work on the VHDL specifications to change them to a level where the logical synthesis can be applied.
- ❖ Steps :
  - \* behavioral transformation on the specifications:
    - ★ eliminate the redundant code
    - ★ linearise the loops
    - ★ propagation of constants
  - \* partitioning
  - \* establish the task order
    - ★ construction of the **control part**
  - \* resources allocation
    - ★ construction of the **data path**
  - \* product the RTL description



## 2.14. Final remarque on the synthesis

- ❖ The semantic of VHDL for the synthesis is defined only on a sub-ensemble of the language because :
  - \* The industrial tools do only logic synthesis at RTL level
  - \* Some VHDL construction are not synthesizable.

## 2.14. Performances

- ❖ Timing
  - \* Estimation of the delay produced by the combinatory elements; detection of the **critical path**
  - \* Estimation of the delay produced by memory elements ( set-up and hold times)
  - \* Estimation of the clock period
- ❖ Surface
  - \* Estimation depending on the HW components used:
    - ★ Standard cells
    - ★ FPGA
- ❖ Power Consumption
- ❖ Testability, reliability, manufacturability

## 2.14. Timing constraints

- ❖ 4 timing constraints has to be study to optimize un circuit --> minimums or maximums time between different points of the circuits:
  - \* inputs - registers
  - \* registers - outputs
  - \* inputs - outputs
  - \* registers-registers
- ❖ Definition of the global clock period  $T_{\text{period}}$  + the additional time on the inputs  $T_{\text{in}}$  and the additional time on the outputs  $T_{\text{out}}$
- ❖ To establish  $T_{\text{out}}$  – we have to know information on the equivalent charge on each output = fan-out
- ❖ To establish  $T_{\text{in}}$  - we have to know information on the output power of the component connected in the entry
- ❖  $T_{\text{period}}$ 
  - \* skew information = the maximal phase between the arrival of the clock on the Flip-Flops
  - \* Latency information = the maximal time between the commutation of the clock and the last flip-flop which will receive the clock. The latency appear on the skew between the external components.

## 2.14. Timing optimization

- ❖ Allow to optimize the clock frequency
- ❖ The CAO tools will not modify the number of Flip-Flops in the circuits during the optimization.
- ❖ The CAO tools optimize the internal combinatory logic
- ❖ We have to separate the combinatory longer path = critical path →  
**OPTIMIZATION OF CRITICAL PATH :**
  - \* All the combinatory logic commun between the critical path and the other logical combinatory path will be duplicated to allow to separate the critical path for a strong optimization of this last one
  - \* Increase of the surface of the circuits after a timing optimization

# FPGAs

Xilinx

Spartan

et

Virtex



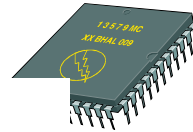
## FPGA's Structure

- ❖ CI Standards (gates, Flip-Flop, decoders, MPX...)
- ❖ CI ASIC 'on demand'
- ❖ CI Configurable (FPGA ...)

Different choices :

Cost  
Design time  
Frequency  
Consumption/weight  
Reliability ...

## FPGA's Structure



### ❖ Configurable Circuits Semi-Custom

Circuits programmable once or many times  
Technologies : Fusible or SRAM

### ❖ PLD : PLA / PLS Programmable Logic Devices

### ❖ EPLD/CPLD Electrically PLD / Complex PLD

### ❖ FPGA Field Programmable Gate Array

ALTERA  
AMD  
LATTICE  
XILINX  
...

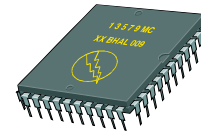
ACTEL  
ALTERA  
XILINX  
...

## Structure des FPGAs

### ❖ Programming Techniques

|        |                                                               |
|--------|---------------------------------------------------------------|
| OTP    | One Time Programmable<br>Fuse ou Antifuse                     |
| EPROM  | Electrically programmable, erasable using UV                  |
| EEPROM | EEPROM erasable electrically                                  |
| FLASH  | EEPROM common erasable                                        |
| SRAM   | Static RAM : volatile memory CMOS<br>+ ROM or external EEPROM |

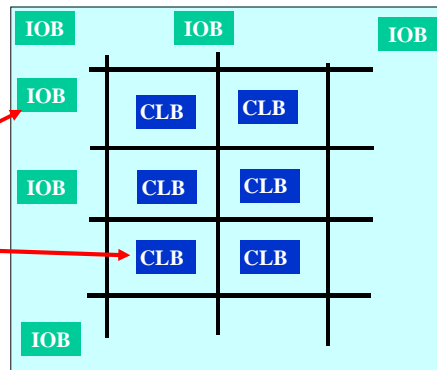
## FPGA's Structure



### ❖ Structure

IOB : Input/Output  
Block

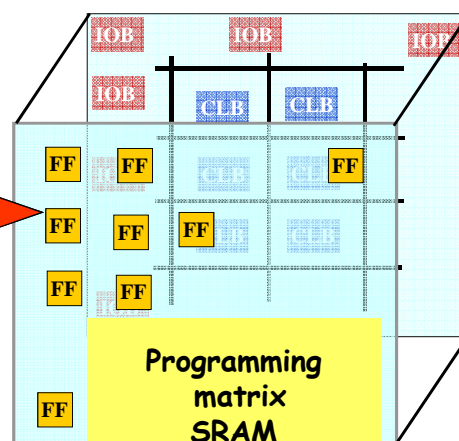
CLB : Configurable  
Logic Block



## Structure des FPGAs

### ❖ Programming

- Anti-Fusible
- Mémoire SRAM  
(FPGA Spartan)



# SPARTAN II

## Structure des FPGAs (SPARTAN)

### ❖ Structure IO

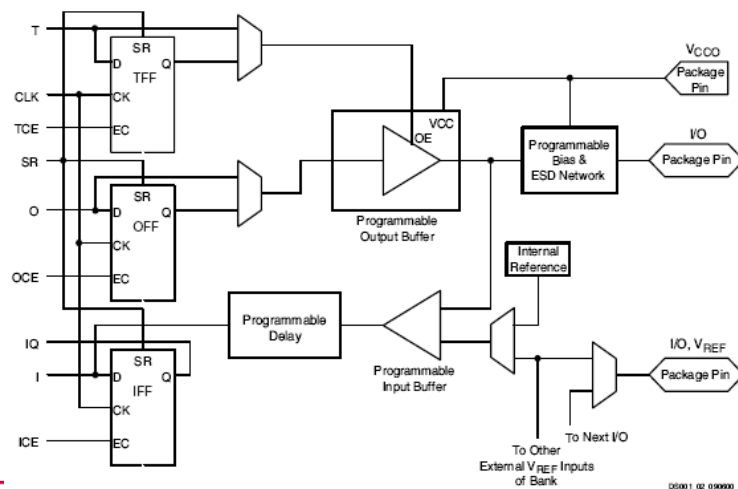
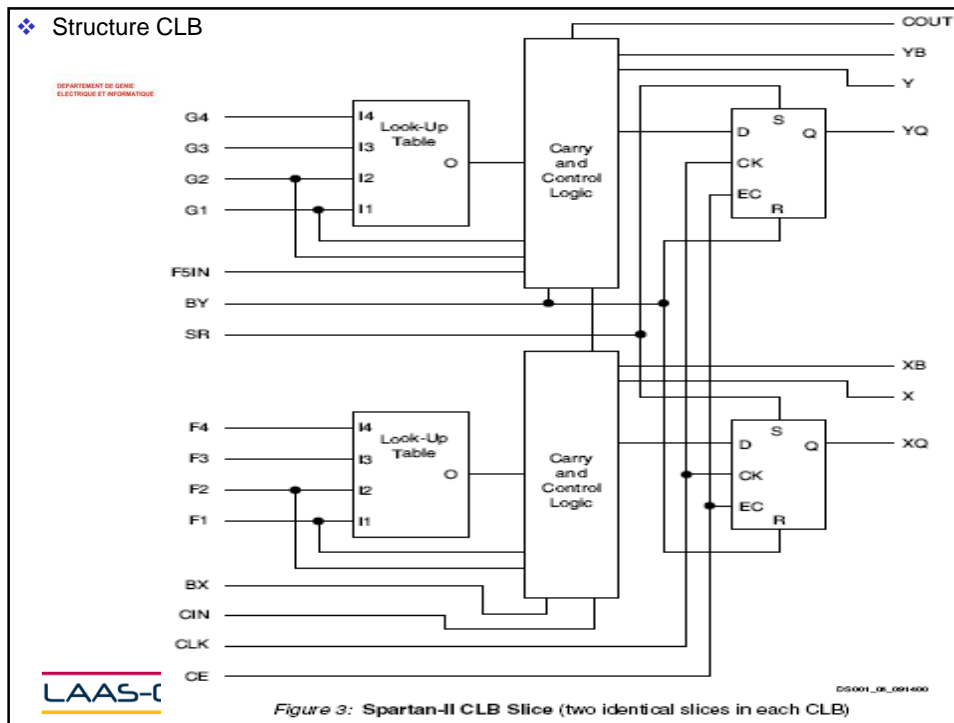


Figure 1: Spartan-II Input/Output Block (IOB)



INSA INSTITUT NATIONAL DES SCIENCES APPLIQUEES TOULOUSE  
DEPARTEMENT DE GENIE ELECTRIQUE ET INFORMATIQUE

## FPGA Spartan II

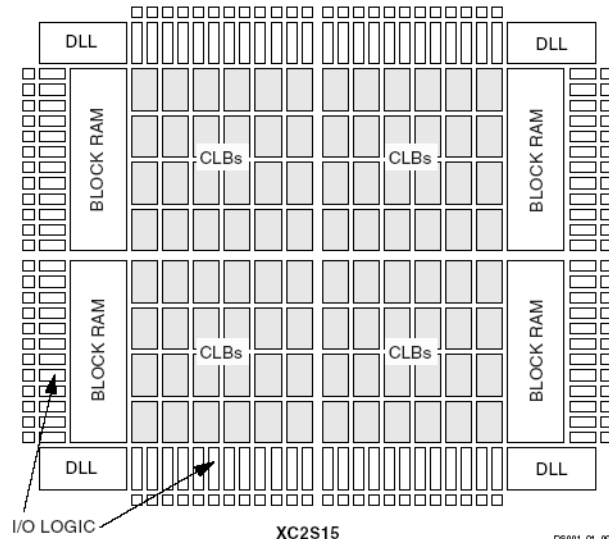
| Device  | Logic Cells | System Gates (Logic and RAM) | CLB Array (R x C) | Total CLBs | Maximum Available User I/O <sup>(1)</sup> | Total Distributed RAM Bits | Total Block RAM Bits |
|---------|-------------|------------------------------|-------------------|------------|-------------------------------------------|----------------------------|----------------------|
| XC2S15  | 432         | 15,000                       | 8 x 12            | 96         | 86                                        | 6,144                      | 16K                  |
| XC2S30  | 972         | 30,000                       | 12 x 18           | 216        | 132                                       | 13,824                     | 24K                  |
| XC2S50  | 1,728       | 50,000                       | 16 x 24           | 384        | 176                                       | 24,576                     | 32K                  |
| XC2S100 | 2,700       | 100,000                      | 20 x 30           | 600        | 196                                       | 38,400                     | 40K                  |
| XC2S150 | 3,888       | 150,000                      | 24 x 36           | 864        | 260                                       | 55,296                     | 48K                  |
| XC2S200 | 5,292       | 200,000                      | 28 x 42           | 1,176      | 284                                       | 75,264                     | 56K                  |

LAAS-CNRS

D.D

WSN Team

## FPGA Spartan II

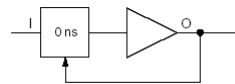


## DLL (Delay Lock Loop)

- ❖ Delay-Locked Loop (DLL) circuits which provide zero propagation delay and low clock skew between output clock signals distributed throughout the device.
- ❖ Each DLL can drive up to two global clock routing networks within the device. The global clock distribution network minimizes clock skews due to loading differences. By monitoring a sample of the DLL output clock, the DLL can compensate for the delay on the routing network, effectively eliminating the delay from the external input port to the individual clock loads within the device.
- ❖ The DLL can provide multiple phases of the source clock.
- ❖ The DLL can also act as a clock doubler or it can divide the user source clock by up to 16.

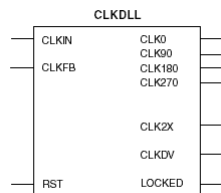


## Library DLL Symbols



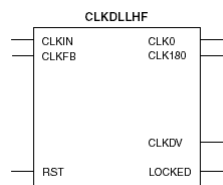
### Simplified DLL Macro Symbol BUFGDLL

This macro delivers a quick and efficient way to provide a system clock with zero propagation delay throughout the device.



### Standard DLL Symbol CLKDLL

access to the complete set of DLL features



### High-Frequency DLL Symbol CLKDLLHF

access to the complete set of DLL features

## Block RAM Features

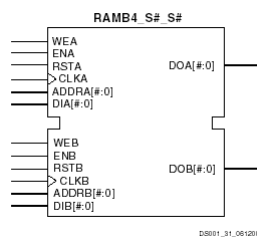
- ❖ The Spartan-II FPGA family provides dedicated blocks of on-chip, true dual-read/write port synchronous RAM, with 4096 memory cells.
- ❖ Each port of the block RAM memory can be independently configured as a read/write port, a read port, a write port, and can be configured to a specific data width.
- ❖ **Operating Modes**
  - \* Block RAM memory supports two operating modes.
    - \* Read Through
    - \* Write Back

## Block RAM Features

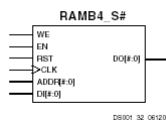
### Block RAM Characteristics:

- ❖ 1. All inputs are registered with the port clock and have a setup to clock timing specification.
- ❖ 2. All outputs have a read through or write back function depending on the state of the port WE pin. The outputs relative to the port clock are available after the clock-to-out timing specification.
- ❖ 3. The block RAM are true SRAM memories and do not have a combinatorial path from the address to the output. The LUT cells in the CLBs are still available with this function.
- ❖ 4. The ports are completely independent from each other (*i.e.*, clocking, control, address, read/write function, and data width) without arbitration.
- ❖ 5. A write operation requires only one clock edge.
- ❖ 6. A read operation requires only one clock edge.
- ❖ 7. The output ports are latched with a self timed circuit to guarantee a glitch free read. The state of the output port will not change until the port executes another read or write operation.

## RAM Library Primitives



Dual-Port Block RAM Memory



Single-Port Block RAM Memory

## RAM Library Primitives

Table 10: Available Library Primitives

| Primitive     | Port A Width | Port B Width |
|---------------|--------------|--------------|
| RAMB4_S1      | 1            | N/A          |
| RAMB4_S1_S1   |              | 1            |
| RAMB4_S1_S2   |              | 2            |
| RAMB4_S1_S4   |              | 4            |
| RAMB4_S1_S8   |              | 8            |
| RAMB4_S1_S16  |              | 16           |
| RAMB4_S2      | 2            | N/A          |
| RAMB4_S2_S2   |              | 2            |
| RAMB4_S2_S4   |              | 4            |
| RAMB4_S2_S8   |              | 8            |
| RAMB4_S2_S16  |              | 16           |
| RAMB4_S4      | 4            | N/A          |
| RAMB4_S4_S4   |              | 4            |
| RAMB4_S4_S8   |              | 8            |
| RAMB4_S4_S16  |              | 16           |
| RAMB4_S8      | 8            | N/A          |
| RAMB4_S8_S8   |              | 8            |
| RAMB4_S8_S16  |              | 16           |
| RAMB4_S16     | 16           | N/A          |
| RAMB4_S16_S16 |              | 16           |

Table 11: Block RAM Port Aspect Ratios

| Width | Depth | ADDR Bus   | Data Bus   |
|-------|-------|------------|------------|
| 1     | 4096  | ADDR<11:0> | DATA<0>    |
| 2     | 2048  | ADDR<10:0> | DATA<1:0>  |
| 4     | 1024  | ADDR<9:0>  | DATA<3:0>  |
| 8     | 512   | ADDR<8:0>  | DATA<7:0>  |
| 16    | 256   | ADDR<7:0>  | DATA<15:0> |

## RAM Library Primitives

- ❖ Block RAM instances can have LOC properties attached to them to constrain the placement. The block RAM placement locations are separate from the CLB location naming convention, allowing the LOC properties to transfer easily from array to array.
- ❖ The LOC properties use the following form:

$$\text{LOC} = \text{RAMB4\_R\#C\#}$$

RAMB4\_R0C0 is the upper left RAMB4 location on the device.

# VIRTEX II



## Virtex II

- ❖ Virtex2 follow Virtex-E
- ❖ Made in December 2000
- ❖ 0.12µm technology
- ❖ Architecture completely re designed to obtain very fast link between the components
- ❖ HW Multipliers (18 bits x 18 bits)
  - \* Faster than the logical gates
  - \* Associated to a memory bloc
- ❖ DCI –digitally controlled impedances

## Virtex II

- ❖ Direct interconnections between the CLB
- ❖ Double port memory blocs
  - \* Inputs, outputs and clock separately
  - \* Configurable from 16K \* 1bit to 512 \* 36 bits
  - \* 3 Write Modes
    - ★ Write First
    - ★ Read First
    - ★ No Change
  - \* Advise : Avoid to write and read on the same memory case at the same moment.

## Virtex II

- ❖ Global clock redirected in 4 directions
  - \* FPGA - better clock propagation
- ❖ DCM – Digital Clock Manager
  - \* Phase Advance
  - \* Multiplication by 2, or multiplication by a quotient
- ❖ Internal clock up to ~ 500 MHz
- ❖ If the clock is not used, the clock will not be powered
  - \* Energy saving

## Clock repartition

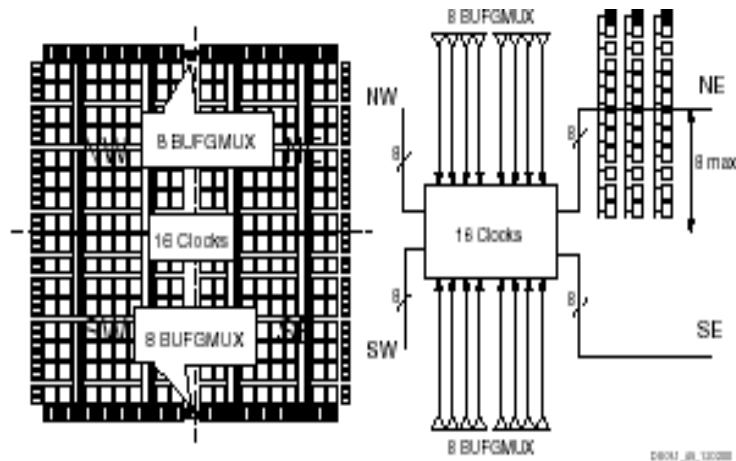


Figure 40: Virtex-II Clock Distribution

## Virtex II family

Table 1: Virtex-II Field-Programmable Gate Array Family Members

| Device   | System Gates | CLB<br>(1 CLB = 4 slices = Max 128 bits) |        |                               | Multiplier Blocks | SelectRAM Blocks |                 | DCMs | Max I/O Pads <sup>(1)</sup> |
|----------|--------------|------------------------------------------|--------|-------------------------------|-------------------|------------------|-----------------|------|-----------------------------|
|          |              | Array Row x Col.                         | Slices | Maximum Distributed RAM Kbits |                   | 18 Kbit Blocks   | Max RAM (Kbits) |      |                             |
| XC2V40   | 40K          | 8 x 8                                    | 256    | 8                             | 4                 | 4                | 72              | 4    | 88                          |
| XC2V80   | 80K          | 16 x 8                                   | 512    | 16                            | 8                 | 8                | 144             | 4    | 120                         |
| XC2V250  | 250K         | 24 x 16                                  | 1,536  | 48                            | 24                | 24               | 432             | 8    | 200                         |
| XC2V500  | 500K         | 32 x 24                                  | 3,072  | 96                            | 32                | 32               | 576             | 8    | 264                         |
| XC2V1000 | 1M           | 40 x 32                                  | 5,120  | 160                           | 40                | 40               | 720             | 8    | 432                         |
| XC2V1500 | 1.5M         | 48 x 40                                  | 7,680  | 240                           | 48                | 48               | 864             | 8    | 528                         |
| XC2V2000 | 2M           | 56 x 48                                  | 10,752 | 336                           | 56                | 56               | 1,008           | 8    | 624                         |
| XC2V3000 | 3M           | 64 x 56                                  | 14,336 | 448                           | 96                | 96               | 1,728           | 12   | 720                         |
| XC2V4000 | 4M           | 80 x 72                                  | 23,040 | 720                           | 120               | 120              | 2,160           | 12   | 912                         |
| XC2V6000 | 6M           | 96 x 88                                  | 33,792 | 1,056                         | 144               | 144              | 2,592           | 12   | 1,104                       |
| XC2V8000 | 8M           | 112 x 104                                | 46,592 | 1,456                         | 168               | 168              | 3,024           | 12   | 1,108                       |

## Virtex II Architecture

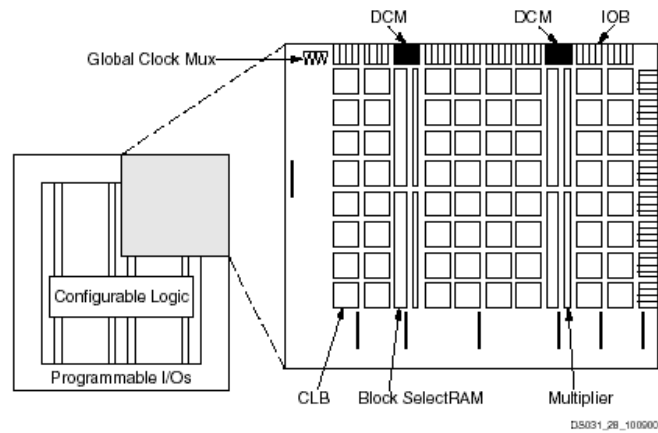


Figure 1: Virtex-II Architecture Overview

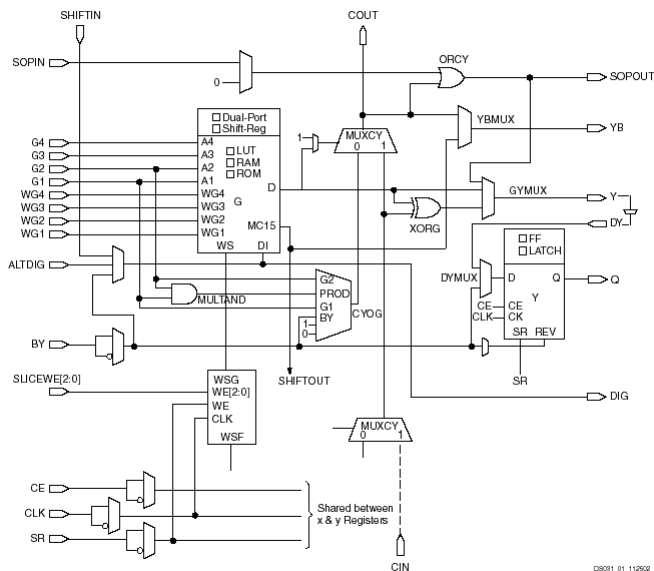


Figure 16: Virtex-II Slice (Top Half)

## Blocks RAM

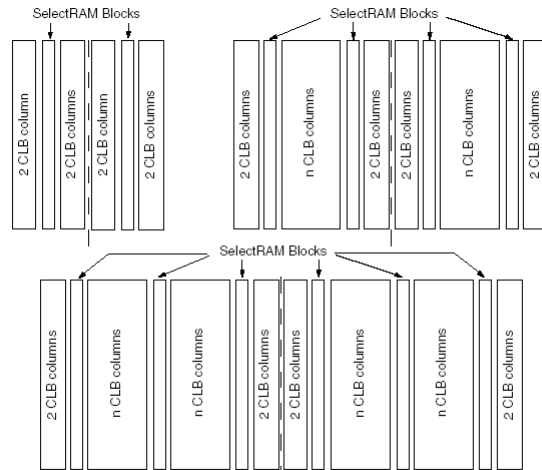
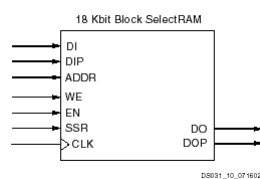


Figure 34: Block SelectRAM (2-column, 4-column, and 6-column)

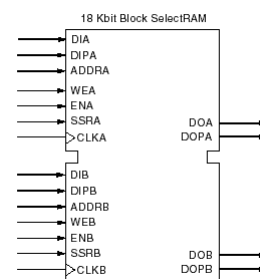
DS031\_36\_101600

## Blocks RAM



DS031\_10\_071602

18 Kbit Block SelectRAM Memory in Single-Port Mode



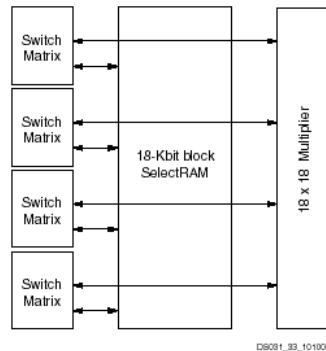
DS031\_11\_071602

18 Kbit Block SelectRAM in Dual-Port Mode



## 18-Bit x 18-Bit Multipliers

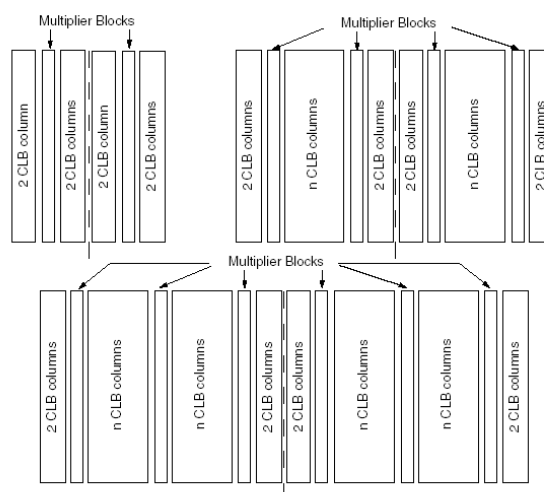
Virtex-II devices incorporate many embedded multiplier blocks. These multipliers can be associated with an 18 Kbit block SelectRAM resource or can be used independently. They are optimized for high-speed operations and have a lower power consumption compared to an 18-bit x 18-bit multiplier in slices.



D8021\_33\_101000

Figure 35: SelectRAM and Multiplier Blocks

## 18-Bit x 18-Bit Multipliers



D8021\_36\_101000

Figure 37: Multipliers (2-column, 4-column, and 6-column)

# SPARTAN III



## Spartan III

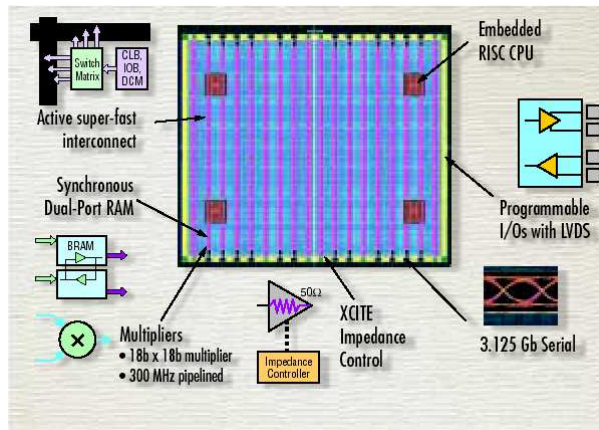
- ❖ Fabricated since 2003
- ❖ Good ideas from Spartan2 and Virtex2
- ❖ Virtex2 architecture
- ❖ 90 nm technology
- ❖ DCM, HW multipliers
- ❖ High density of CLBs
- ❖ Max. frequency ~ 350Mhz
- ❖ Low cost

## Spartan III

| Famille     | Virtex2     | Spartan3  | Virtex2     | Spartan3  |
|-------------|-------------|-----------|-------------|-----------|
| Nom         | XC2V250     | XC3S200   | XC2V1500    | XC3S1500  |
| Logic Cells | 3456        | 4320      | 17280       | 29952     |
| Multipliers | 24          | 12        | 48          | 32        |
| Max ram     | 432 Kbits   | 216 Kbits | 864 Kbits   | 576 Kbits |
| DCM         | 8           | 4         | 8           | 4         |
| CLB         | 384         | 480       | 1920        | 3328      |
| Prix        | 158 dollars | ?         | 600 dollars | ?         |

## Virtex II Pro

## Xilinx Virtex II Pro



## Xilinx Virtex II Pro

- ❖ Matrix of HW multipliers to support high speed parallel signal processing
- ❖ Multi-giga bit serial link for inter-chip communications or inter-systems
- ❖ RISC microprocessor for the decision tasks and to run real-time OS.
- ❖ Dynamically configurable impedances for the inputs /outputs → easier design of the PCB of the system including the FPGA

## Platform for Software Defined Radio

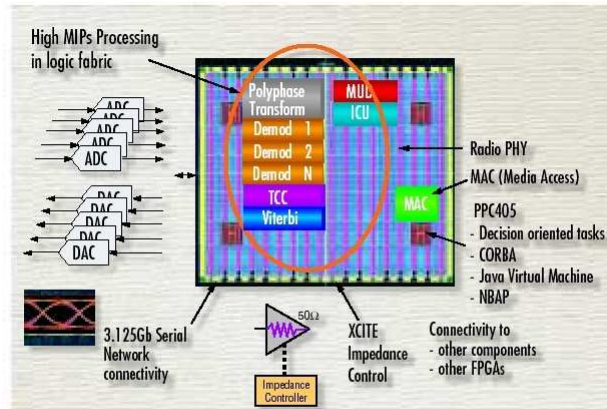
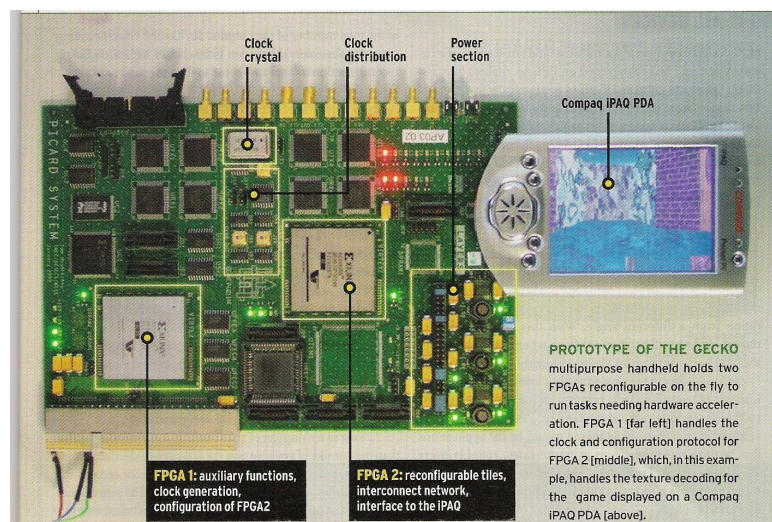


Figure 3. Platform FPGA approach to software-defined radio realization. The high MIPs processing is implemented in the logic fabric, while decision-oriented and non real-time tasks are provided as embedded software running on the Power PC. The multi-Gigabit transceivers could be used for providing connectivity to the broader network.

## DO-IT-All Devices

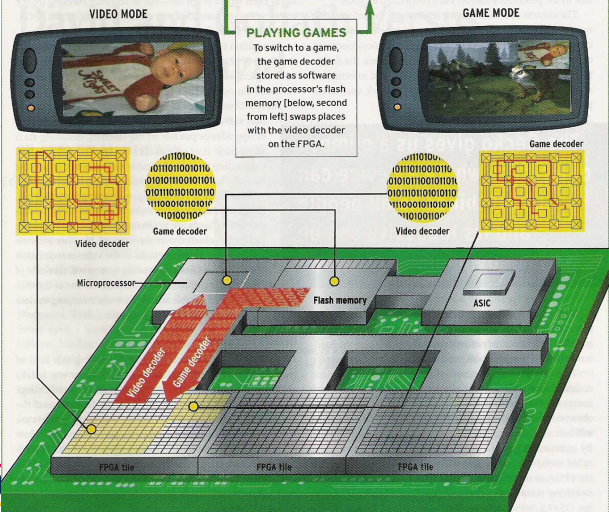


PROTOTYPE OF THE GECKO multipurpose handheld holds two FPGAs reconfigurable on the fly to run tasks needing hardware acceleration. FPGA 1 (far left) handles the clock and configuration protocol for FPGA 2 (middle), which, in this example, handles the texture decoding for the game displayed on a Compaq IPAQ PDA (above).

## DO-IT-All Devices

**AS A BABY MONITOR** In this artist's rendering of a production version of the Gecko, the video decoder task (far left and bottom on circuit board) that is displaying a full-screen image of a baby (on handheld, below) runs on one of three "tiles" on an FPGA (on circuit board) to provide the best resolution at the highest frame rate.

**PLACES TRADED** The game runs at maximum quality on the FPGA (on circuit board, far right) and takes up most of the display area. But the baby monitor still functions. The video decoder restarts as software on the processor (below, second from right) and the baby is displayed smaller than before, at lower resolution, and at a slower frame rate, in a corner of the screen (below).



LAAS-CN

WSN Team

Compléments de cours


Exercices corrigés

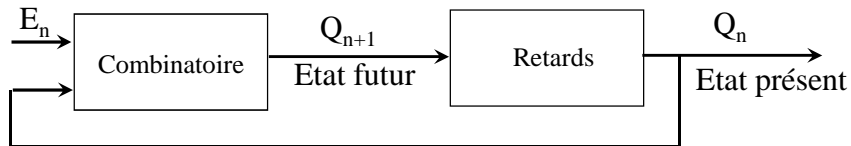
LAAS-CNRS

D.D

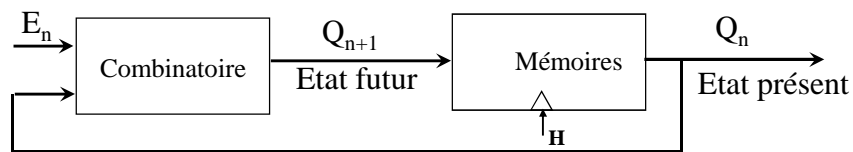
WSN Team

## Systèmes séquentiels Synchrone et Asynchrone

- ❖ Système asynchrone  Plus rapide mais de conception complexe

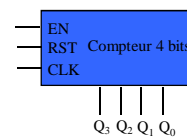


- ❖ Système synchrone  Conception moins complexe



## Exercice : Synthèse d'un compteur 4 bits modulo 10

- ❖ Logique séquentielle **synchrone**
  - EN = '1' - comptage autorisé
  - EN = '0' - comptage bloqué
- ❖ Utilisez de bascules D
  - RST = '1' - RAZ
  - RST = '0' - comptage



| Q <sub>3</sub> | Q <sub>2</sub> | Q <sub>1</sub> | Q <sub>0</sub> | En | D <sub>3</sub> | D <sub>2</sub> | D <sub>1</sub> | D <sub>0</sub> | En | D <sub>3</sub> | D <sub>2</sub> | D <sub>1</sub> | D <sub>0</sub> |
|----------------|----------------|----------------|----------------|----|----------------|----------------|----------------|----------------|----|----------------|----------------|----------------|----------------|
| 0              | 0              | 0              | 0              | 1  | 0              | 0              | 0              | 1              | 0  | 0              | 0              | 0              | 0              |
| 0              | 0              | 0              | 1              | 1  | 0              | 0              | 1              | 0              | 0  | 0              | 0              | 0              | 1              |
| 0              | 0              | 1              | 0              | 1  | 0              | 0              | 1              | 1              | 0  | 0              | 0              | 1              | 0              |
| 0              | 0              | 1              | 1              | 1  | 0              | 1              | 0              | 0              | 0  | 0              | 0              | 1              | 1              |
| 0              | 1              | 0              | 0              | 1  | 0              | 1              | 0              | 1              | 0  | 0              | 1              | 0              | 0              |
| 0              | 1              | 0              | 1              | 1  | 0              | 1              | 1              | 0              | 0  | 0              | 1              | 0              | 1              |
| 0              | 1              | 1              | 0              | 1  | 0              | 1              | 1              | 1              | 0  | 0              | 1              | 1              | 0              |
| 0              | 1              | 1              | 1              | 1  | 1              | 0              | 0              | 0              | 0  | 0              | 1              | 1              | 1              |
| 1              | 0              | 0              | 0              | 1  | 1              | 0              | 0              | 1              | 0  | 1              | 0              | 0              | 0              |
| 1              | 0              | 0              | 1              | 1  | 0              | 0              | 0              | 0              | 0  | 1              | 0              | 0              | 1              |

Sorties des bascules D

état futur des entrées des  
bascules D pour EN='1'

état futur des entrées des  
bascules D pour EN='0'

Exercice : Synthèse d'un compteur 4 bits modulo 10

|                               |    | En=0 |    |    |    |
|-------------------------------|----|------|----|----|----|
|                               |    | 00   | 01 | 11 | 10 |
| Q <sub>3</sub> Q <sub>2</sub> | 00 | 0    | 1  | 1  | 0  |
|                               | 01 | 4    | 5  | 7  | 6  |
|                               | 11 | 12   | 13 | 15 | 14 |
|                               | 10 | 8    | 9  | 11 | 10 |

|                               |    | En=1 |    |    |    |
|-------------------------------|----|------|----|----|----|
|                               |    | 00   | 01 | 11 | 10 |
| Q <sub>3</sub> Q <sub>2</sub> | 00 | 16   | 17 | 19 | 18 |
|                               | 01 | 20   | 21 | 23 | 22 |
|                               | 11 | 28   | 29 | 31 | 30 |
|                               | 10 | 24   | 25 | 27 | 26 |

$$D_0 = Q_0 \oplus E_n$$

Exercice : Synthèse d'un compteur 4 bits modulo 10

|                               |    | En=0 |    |    |    |
|-------------------------------|----|------|----|----|----|
|                               |    | 00   | 01 | 11 | 10 |
| Q <sub>3</sub> Q <sub>2</sub> | 00 | 0    | 1  | 3  | 2  |
|                               | 01 | 4    | 5  | 7  | 6  |
|                               | 11 | 12   | 13 | 15 | 14 |
|                               | 10 | 8    | 9  | 11 | 10 |

|                               |    | En=1 |    |    |    |
|-------------------------------|----|------|----|----|----|
|                               |    | 00   | 01 | 11 | 10 |
| Q <sub>3</sub> Q <sub>2</sub> | 00 | 16   | 17 | 19 | 18 |
|                               | 01 | 20   | 21 | 23 | 22 |
|                               | 11 | 28   | 29 | 31 | 30 |
|                               | 10 | 24   | 25 | 27 | 26 |

$$D_1 = \bar{E}_n Q_1 + Q_1 \bar{Q}_0 + E_n \bar{Q}_3 \bar{Q}_1 Q_0$$



Exercice : Synthèse d'un compteur 4 bits modulo 10

| $D_2$ |       | $Q_1 Q_0$ |    | En=0 |    |
|-------|-------|-----------|----|------|----|
| $Q_3$ | $Q_2$ | 00        | 01 | 11   | 10 |
| 00    | 0     | 0         | 1  | 3    | 2  |
| 01    | 0     | 0         | 0  | 0    | 0  |
| 11    | 1     | 1         | 1  | 1    | 1  |
| 10    | 0     | 0         | 0  | 0    | 0  |

|       |       | $Q_1 Q_0$ |    | En=1 |    |
|-------|-------|-----------|----|------|----|
| $Q_3$ | $Q_2$ | 00        | 01 | 11   | 10 |
| 00    | 0     | 0         | 0  | 1    | 0  |
| 01    | 1     | 1         | 1  | 0    | 1  |
| 11    | 0     | 0         | 0  | 0    | 0  |
| 10    | 0     | 0         | 0  | 0    | 0  |

$$D_2 = \overline{E_n} Q_2 + Q_2 \overline{Q_1} + Q_2 \overline{Q_0} + E_n \overline{Q_2} Q_1 Q_0$$

Exercice : Synthèse d'un compteur 4 bits modulo 10

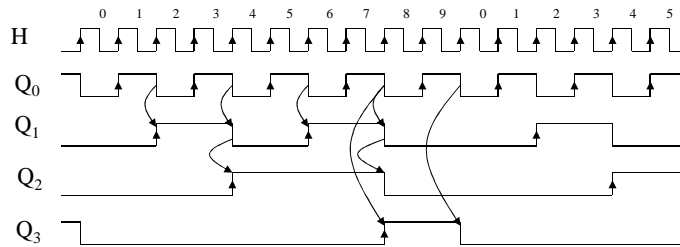
| $D_3$ |       | $Q_1 Q_0$ |    | Enable=0 |    |
|-------|-------|-----------|----|----------|----|
| $Q_3$ | $Q_2$ | 00        | 01 | 11       | 10 |
| 00    | 0     | 0         | 0  | 3        | 2  |
| 01    | 0     | 0         | 0  | 0        | 0  |
| 11    | 0     | 0         | 0  | 0        | 0  |
| 10    | 1     | 1         | 1  | 1        | 1  |

|       |       | $Q_1 Q_0$ |    | Enable=1 |    |
|-------|-------|-----------|----|----------|----|
| $Q_3$ | $Q_2$ | 00        | 01 | 11       | 10 |
| 00    | 0     | 0         | 0  | 0        | 0  |
| 01    | 0     | 0         | 0  | 1        | 0  |
| 11    | 0     | 0         | 0  | 0        | 0  |
| 10    | 1     | 0         | 0  | 0        | 0  |

$$D_3 = \overline{E_n} Q_3 + Q_3 \overline{Q_0} + E_n Q_2 Q_1 Q_0$$

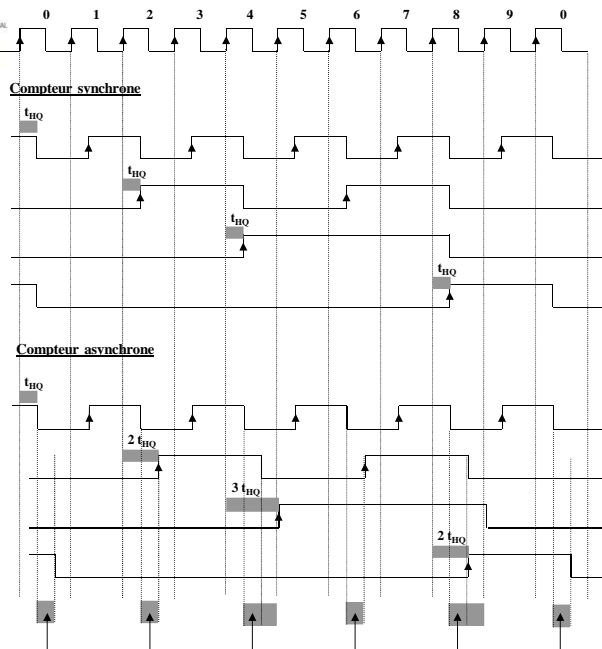
## Exercice : Synthèse d'un compteur 4 bits modulo 10 asynchrone

- ❖ Les bascules ne sont plus reliées à la même horloge
- ❖ Choisir le signal de commande le plus adapté



H commande  $Q_0$ .  
 $\overline{Q_0}$  commande les bascule 1 et 3.  
 $Q_1$  commande la bascule 2

$$\begin{aligned} D_0 &= \overline{Q_0} \\ D_1 &= \overline{Q_3} \overline{Q_1} \\ D_2 &= \overline{Q_2} \\ D_3 &= Q_2 Q_1 \end{aligned}$$



## Régistre à decalage

```
❖ Library IEEE;
❖ Use IEEE.std_logic_1164.all;

❖ entity reg_decalage is
❖ GENERIC(n: natural :=2);
❖ port (
❖ Data, clock : in std_logic;
❖ S : inout std_logic_vector(n-1
❖ downto 0));
❖ end reg_decalage;

❖ Architecture structurelle OF reg_decalage IS
❖ COMPONENT mem PORT (D, clk : in STD_logic;
❖ Q : INOUT STD_LOGIC ;
❖ Qb : out std_logic);
❖ end COMPONENT;

❖ begin
❖ gauche : mem Port map (data, clock, S(n-1));
❖ boucle : For i IN 1 to n-1 generate
❖ circ: mem PORT MAP (S(n-i), clock, S(n-
❖ i-1));
❖ END GENERATE boucle;
❖ END structurelle ;

❖ configuration reg_config of reg_decalage is
❖ for structurelle
❖ for all: mem use entity work.basculeD(beh);
❖ end for;
❖ end for;
❖ end reg_config;
```

## Régistre à decalage

```
❖ Architecture beh OF reg_decalage IS
❖ begin
❖
❖ comport : process
❖ begin
❖ wait until clock ='1';
❖ s(n-1) <= Data after 5 ns;
❖ copie : For i IN n-1 to 1 LOOP
❖ s(n-i-1) <= s(n-i) after 5 ns;
❖ end loop copie;
❖ end process comport ;
❖ end beh;
```

## Conception d'un compteur synchrone 8 bits

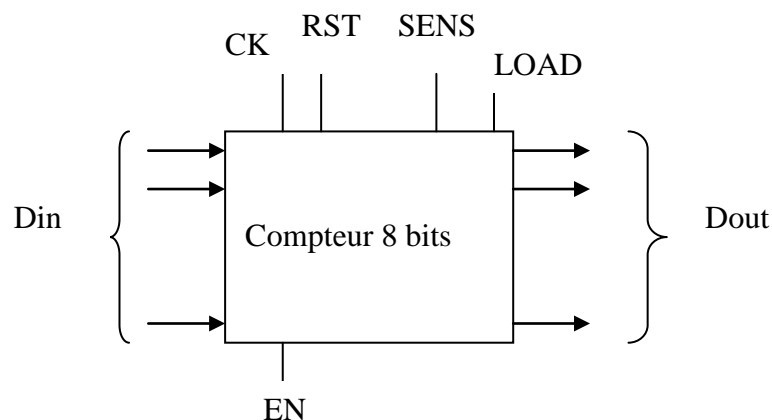
Objectifs du TD :

- Apprendre à écrire son premier code VHDL
- Apprendre et maîtriser le flot de conception FPGA (synthèse, placement routage et implémentation sur FPGA)
- Découvrir le logiciel XILINX ISE

### Spécifications :

Il faut écrire le code VHDL décrivant le comportement d'un compteur 8 bits synchrone sur le front montant. Ce compteur a les signaux suivants :

|      |                                                                                                                           |
|------|---------------------------------------------------------------------------------------------------------------------------|
| CK   | - horloge.                                                                                                                |
| RST  | - reset – signal de remise à zéro du compteur <u>synchrone</u> avec l'horloge (actif bas).                                |
| LOAD | - signal de commande <u>synchrone</u> de chargement du compteur (actif haut).                                             |
| SENS | - à l'état bas, le circuit décrément à chaque transition montante de l'horloge,<br>- à l'état haut, le circuit incrément. |
| EN   | - enable – permet au compteur de compter s'il est à la masse (actif bas)                                                  |
| Din  | - donnée à charger dans le compteur (sur 8 bits) quand la commande LOAD est active                                        |
| Dout | - sortie sur 8 bits                                                                                                       |



Les signaux CK, SENS, RST et LOAD sont du type **std\_logic**.

Les signaux Din et Dout sont des **std\_logic\_vector**.

A chaque front montant de l'horloge, si le signal de RESET est activé (égal à 0), la remise à zéro du compteur sera faite. Le signal RESET a la plus grande priorité. Si le signal de LOAD du compteur est activé (égal à 1) sur le front montant d'horloge, la donnée présente sur les entrées Din sera chargé dans le compteur et affiché sur la sortie Dout. Tant que le signal de

LOAD est actif, la sortie du compteur est maintenue à la valeur de Din. Si le signal LOAD n'est pas actif et le signal ENABLE est actif (égal à 0), le compteur va compter ou décompter en fonction de la valeur du signal SENS.

### **Travail à faire :**

1. Ecrire le code VHDL correspondant.
2. Ecrire le programme de test de ce compteur.
3. Simuler.
4. Faire la synthèse de ce code VHDL.
  - Regarder le circuit logique obtenu
  - Regarder le rapport de synthèse et vérifier le nombre des bascules obtenus et la fréquence maximale de fonctionnement
5. Effectuez le placement routage.
6. Effectuer la simulation du compteur 8 bits placé et routé. Comparez avec la simulation fonctionnelle du compteur (code VHDL).
  - Trouver la fréquence maximale de fonctionnement de votre compteur (une indication est donnée dans le rapport après synthèse, mais n'oubliez pas que lors de la synthèse les temps de propagation sur les interconnexions ne sont pas pris en compte)
  - Faites fonctionner le compteur à une fréquence très grande, pour bien le voir décrocher (il ne compte plus correctement)
  - Il y a-t-il des états aléatoires dans votre circuit ? Sont-ils gênants ?
7. Implémenter le compteur 8 bits sur le FPGA. Placez :
  - Din sur les switch
  - RST, LOAD, SENS, EN sur les boutons poussoirs
  - Dout sur les LEDs
  - CLK à ralentir à 1-3Hz (clk de la maquette à 50MHz). Concevez votre circuit diviseur d'horloge en VHDL.

## Réalisation d'un contrôleur DMA

**Concevez l'architecture et écrivez le code VHDL correspondant du contrôleur DMA décrit par la suite.**

Le transfert des données entre les périphériques (disque dur par exemple) et la mémoire RAM est géré par le contrôleur DMA. Pendant le temps de transfert des données entre le disque dur et la mémoire, le processeur est déconnecté du bus des données et le contrôleur DMA devient le maître du système des bus.

Le contrôleur DMA active les bus des données et d'adresses pour gérer le transfert entre le périphérique et la mémoire. La méthode la plus utilisée consiste à employer un signal de contrôle spécial qui s'appelle « bus request » BR à '1', signal envoyé par le DMA vers le microprocesseur pour lui demander l'utilisation du bus. Le  $\mu P$  fini l'exécution de l'instruction en cours, relâche les bus en mettant les lignes en haute impédance et il envoie le signal « bus grant » BG à '1' vers le contrôleur DMA. A la fin de transfert DMA, le contrôleur met le signal BR à '0' et le  $\mu P$  reprend le contrôle.

Une fois que BG = '1', le contrôleur DMA prend le contrôle des bus pour communiquer directement avec la mémoire.

Le bus de données et la taille des mots dans la mémoire est de 32 bits.

Pendant un transfert DMA un seul mot de 32 bits peut être transféré ou un bloc entier contenant plusieurs mots. Le contrôleur DMA à réaliser est présenté dans la Fig.1

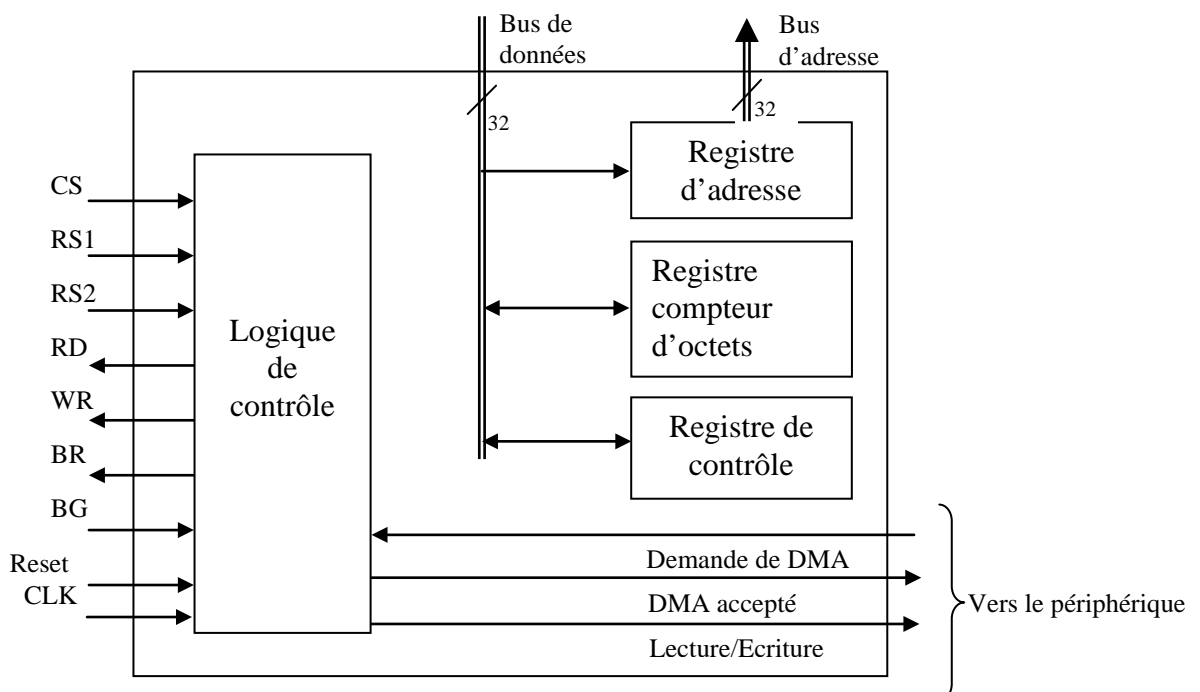


Figure 1. Contrôleur DMA

Le contrôleur dispose de trois registres :

Registre d'adresse : C'est un registre de 16 bits qui spécifie l'adresse du mot de la mémoire principale (RAM) que l'on veut accéder. La valeur du registre doit sortir sur le bus d'adresse qui a une taille de 32 bits. Le registre d'adresse est incrémenté à chaque transfert DMA d'un octet. Comme le bus a 4 octets, le registre s'incrémente de quatre en quatre.

Registre compteur d'octets : C'est un registre de 16 bits qui est initialisé avec le nombre d'octets à transférer lors d'un accès DMA. Il est décrémenté après chaque transfert de quatre en quatre.

Registre de contrôle : C'est un registre de 3 bits. Il sert à indiquer le type de transfert désiré, soit en lecture (de la mémoire vers le périphérique= bit 1 du registre à '1') soit en écriture (du périphérique vers la mémoire = bit 2 du registre à '1') et pour programmer le transfert DMA (le bit 0 du registre à '1').

La programmation de ces trois registres est faite par le  $\mu P$ , en utilisant le bus de données et les signaux de contrôle CS, RS1, RS2. Les registres du DMA sont sélectionnés par le  $\mu P$  en activant registre d'adresse ; RS1=1 et RS2=0 au registre compteur et RS1=0 et RS2=1 au registre de contrôle.

Quand le signal BG='1' le DMA peut communiquer directement avec la mémoire en spécifiant une adresse sur le bus d'adresse et en activant un de signaux de contrôle RD ou WR.

Le contrôleur DMA communique avec le périphérique à travers les signaux de « demande DMA », de « DMA accepté » et « lecture/écriture » ( voir fig.1). « Lecture » – transfert de la mémoire vers le périphérique et « écriture »- transfert du périphérique vers la mémoire. Le contrôleur DMA met le signal « DMA accepté » à '1' quand le  $\mu P$  a donnée le contrôle au contrôleur DMA. Quand le périphérique fait une demande de DMA (signal « demande DMA » ='1'), le contrôleur demande le bus au  $\mu P$ (a vous de voir les signaux qu'il faut affecter). Le périphérique est connecté directement au bus de données.

Le transfert DMA fini quand le registre compteur arrive à zéro ou quand le périphérique met le signal « demande DMA » à zéro.

Tout le fonctionnement du DMA est synchrone sur le front montant de l'horloge (signal CLK).

Le reset est lui aussi synchrone avec l'horloge. Le signal de reset est active bas '0' et provoque la mise à zéro des 3 registres.