



Μια παρουσίαση της VHDL!

## Σχεδιασμός Συστημάτων VLSI.

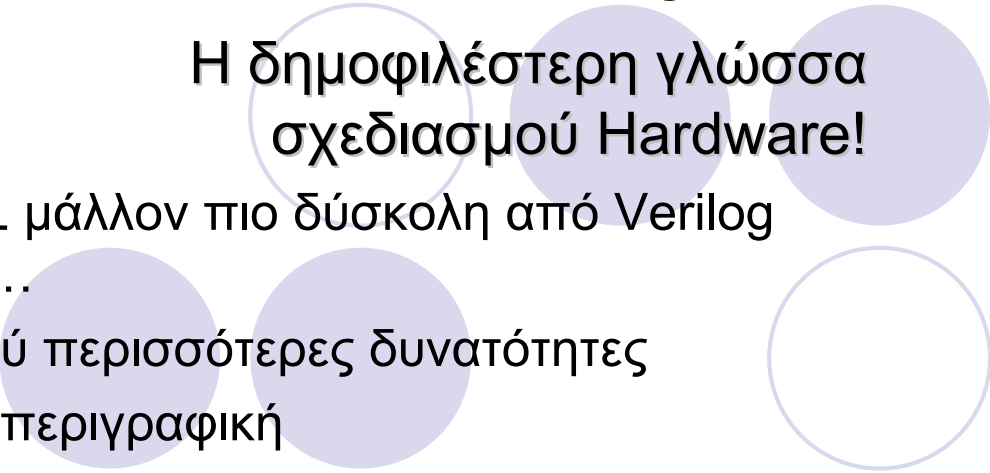
Καράγιωργας Νικόλαος Α.Μ. 90  
Τριανταφυλλόπουλος Σταύρος Α.Μ. 92  
Μεταπτυχιακό Πρόγραμμα **ΟΣΥΛ**

---

Μια παρουσίαση της VHDL!

Η δημοφιλέστερη γλώσσα  
σχεδιασμού Hardware!

VHDL μάλλον πιο δύσκολη από Verilog  
αλλά...

- πολύ περισσότερες δυνατότητες
  - πιο περιγραφική
  - πιο αυστηρή
  - Όχι για χαμηλά επίπεδα
  - Case insensitive
- 

# Εισαγωγή στη VHASIC Hardware Description Language (VHDL).

- Ανάγκη για μια standard γλώσσα περιγραφής ολοκληρωμένων συστημάτων.
- Εμφανίστηκε στα πλαίσια του προγράμματος Very High Speed Integrated Circuits (VHSIC) το 1980. Εξού και το αρχικό V.
- Αναπτύχθηκε και προσαρτήθηκε ως standard στο IEEE.
- Ομοιότητες με ADA software language.
- **Στόχοι :**
  - Περιγραφή της δομής του σχεδιασμού (διαχωρισμός σε sub-designs και διασύνδεση αυτών).
  - Χρήση προτύπων γλωσσών προγραμματισμού, για τις προδιαγραφές λειτουργίας του σχεδιασμού.
  - Εξομοίωση πριν την παραγωγή του. Γρήγορη σύγκριση εναλλακτικών και έλεγχος ορθότητας.

## Χαρακτηριστικά της VHDL.

- Υποστηρίζονται ιεραρχίες (block diagram), επαναχρησιμοποιούμενα components, διαχείριση λαθών και επιβεβαίωση λειτουργίας (verification).
- Τα components μπορούν να αποθηκευτούν σε library για επαναχρησιμοποίηση.
- Δυνατότητα verification λειτουργίας σε εξομοιωτή.
- Πιθανή η μετακίνηση κώδικα για simulation μεταξύ διαφορετικών development συστημάτων.
- Υποστηρίζονται διάφορες μέθοδοι ανάπτυξης όπως top-down, bottom-up κτλ.

# Entities και Architectures.

## Entity (οντότητα)

- Entity declaration :
- Περικλείεται από τον όνομα της οντότητας.
- Το port δηλώνει I/O pins.
- Τα I/O pins μπορεί να είναι in, out, inout, buffer.
- Std\_logic σημαίνει ότι είναι ένα digital pin.

## β) Architecture:

Ο κώδικας. Περιγράφει μια υλοποίηση της entity.

```
● entity adder is
● port (in0, in1 : in std_logic;
●      sum, cout : out
●      std_logic);
● end adder;

● architecture rtl of adder is
● begin
● sum <= in0 and in1;
● cout <= in0 xor in1;
● end rtl;

● architecture str of adder is
● begin
● ...
● end str;
```

# Architecture.

- Κάθε statement στο architecture body εκτελείται παράλληλα, εκτός των statements που περικλείονται από process.
- Η process συμπεριφέρεται σαν ένα παράλληλο statement.
- Process (sensitivity list): όταν ένα ή περισσότερα από τα signals της sensitivity list αλλάξουν τιμή, η process εκτελείται.
- Τα σχόλια δηλώνονται με "--".

# Περιεχόμενα μιας architecture.

- **architecture** beh of adder is
- **begin**
- sum\_proc: **process** (in0, in1)
- **begin**
- **if** (in0='1' and in1='1') **then**
- sum <= '1';
- **else**
- sum <= '0';
- **end if;**
- **end process** sum\_proc;
- cout <= in0 **xor** in1;
- **end beh;**

- Processes
  - Σειριακά statements.
  - Sensitivity list.
  - Variables.
- Απλά statements Concurrent,
- <= για τα signals in0,in1, sum, cout

# *Behavioral* architecture.

- Περιγράφει τον αλγόριθμο (λειτουργία) που εκτελείται *χωρίς* αναφορά στην εσωτερική δομή.
- Περιλαμβάνει *process statements*, που περιέχουν *sequential statements* οι οποίες περιλαμβάνουν *signal assignment statements* and *wait statements*.

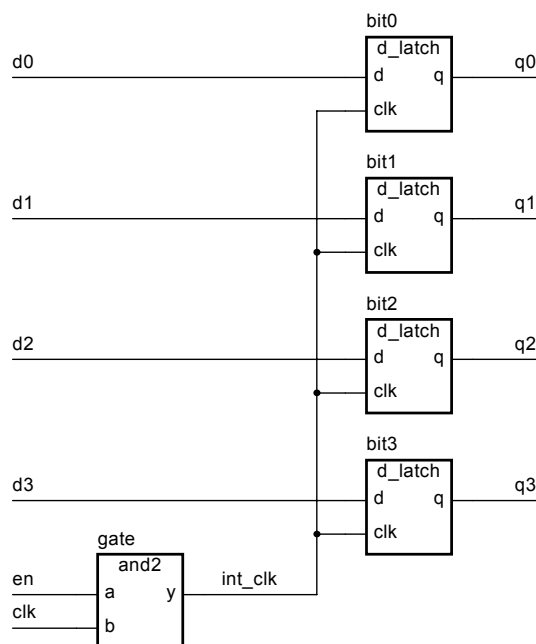
```
architecture behav of reg4 is
  begin
    storage : process is
      variable stored_d0, stored_d1,
        stored_d2, stored_d3 : bit;

      begin
        if en = '1' and clk = '1' then
          stored_d0 := d0;
          stored_d1 := d1;
          stored_d2 := d2;
          stored_d3 := d3;
        end if;
        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
        wait on d0, d1, d2, d3, en, clk;
      end process storage;
    end architecture behav;
```

# Structural architecture

- Το module υλοποιείται σαν συγκρότηση (composition) υποσυστημάτων (sub-modules).
- Περιλαμβάνει
  - *signal declaration*: για εσωτερικές διασυνδέσεις.
  - *component instances*: χρήση ήδη δηλωμένων ζευγαριών entity/architecture.
  - *port maps* στα component instances: σύνδεση των signals στα ports των components.
  - *wait statements*

## Structure Example





# Structure Example

Πρώτα δηλώνουμε τις entities και τις architectures για ένα d-latch και για μια and πύλη.

- **entity** d\_latch **is**  
**port** ( d, clk : **in** bit; q : **out** bit );  
**end entity** d\_latch;
- **architecture** basic **of** d\_latch **is**  
**begin**
- latch\_behavior : **process** **is**  
**begin**  
    **if** clk = '1' **then**  
        q <= d **after** 2 ns;  
    **end if**;  
    **wait on** clk, d;  
    **end process** latch\_behavior;
- **end architecture** basic;

- **entity** and2 **is**  
**port** ( a, b : **in** bit; y : **out** bit );  
**end entity** and2;
- **architecture** basic **of** and2 **is**  
**begin**
- and2\_behavior : **process** **is**  
**begin**  
    y <= a **and** b **after** 2 ns;  
    **wait on** a, b;  
    **end process** and2\_behavior;
- **end architecture** basic;



# Structure Example

- Τώρα χρησιμοποιούμε αυτά για να υλοποιήσουμε έναν register.

```
architecture struct of reg4 is  
    signal int_clk : bit;  
begin  
    bit0 : entity work.d_latch(basic)  
        port map ( d0, int_clk, q0 );  
    bit1 : entity work.d_latch(basic)  
        port map ( d1, int_clk, q1 );  
    bit2 : entity work.d_latch(basic)  
        port map ( d2, int_clk, q2 );  
    bit3 : entity work.d_latch(basic)  
        port map ( d3, int_clk, q3 );  
    gate : entity work.and2(basic)  
        port map ( en, clk, int_clk );  
end architecture struct;
```



## Identifiers.

- Ονόματα για αναγνώριση από το χρήστη των data object, π.χ. signals, variables κτλ
- Ο πρώτος χαρακτήρας, γράμμα.
- Ο τελευταίος χαρακτήρας δεν μπορεί να είναι underscore (\_).
- Επιτρέπονται μόνο γράμματα, ψηφία και underscore.
- Δεν επιτρέπονται 2 συνεχόμενα underscore.
- Παραδείγματα identifiers: a, b, c, a\_xy, clk ...



## Data objects.

- **Constant**: κρατά μια τιμή η οποία δεν μπορεί να αλλάξει κατά τη σχεδίαση.  
π.χ. **constant width: integer 8**
- **Signal**: για αναπαράσταση καλωδιακών συνδέσεων.  
Δηλώνονται: α) στο port της entity ή στην αρχή μιας αρχιτεκτονικής (global).  
πχ : **signal count: bit\_vector (3 downto 0);**  
-- count σημαίνει 4 καλώδια (wires) αυτά είναι count(3), count(2), count(1), count(0).
- **Variable**: εσωτερική αναπαράσταση που χρησιμοποιείται από τους προγραμματιστές. Ίσως να μην υπάρχουν. Δηλώνονται-χρησιμοποιούνται εσωτερικά σε μια process (local). π.χ. **variable v1 bool : BOOLEAN:= TRUE;**

## Τύποι Δεδομένων (Data types).

- Οι τύποι καθορίζουν τα χαρακτηριστικά των variables, των signals κτλ. Μόνο δεδομένα του ίδιου τύπου μπορούν να αλληλεπιδρούν μεταξύ τους.
- Μπορεί να χρησιμοποιηθεί η **TYPE** για δημιουργία νέων τύπων αλλά χρειάζεται προσοχή ποια library και standard VHDL παρέχονται.  
π.χ. **type** car\_state **is** (stop, slow, medium, fast);
- Είναι επιτρεπτό να παραχθούν **subtypes** από κάποιον προκαθορισμένο από το χρήστη τύπο.  
π.χ. **subtype** MY\_INTEGER **is** INTEGER **range** 48 **to** 80;
- Μπορούν να οριστούν πράξεις στους δημιουργημένους τύπους.

## Data Types- Operators.

- 4 κύριες κατηγορίες data types:
  - Scalar types (φυσικές ποσότητες κτλ)
  - Composite types (arrays, records)
  - Access types (pointers).
  - File types (δεν περιγράφονται εδώ).
- 6 κατηγορίες operators
  1. Logical operators.
  2. Relational operators.
  3. Shift operators.
  4. Adding operators.
  5. Multiplying operators.
  6. Miscellaneous operators.

# Data Types

## 1. Scalar Types

Υπάρχουν 4 διαφορετικά είδη scalar types:

a) Enumeration: Ένα σύνολο τιμών ορισμένες από το χρήστη

Η σειρά με την οποία εμφανίζονται οι τιμές στην δήλωση του τύπου απαρίθμησης καθορίζουν τη σειρά τους.

π.χ. **type** MICRO\_OP **is**  
(LOAD,STORE,ADD,SUB,MUL,DIV);

Οι δυαδικοί αριθμοί που εκχωρούνται είναι:

Load = 000, Store = 001, Add = 010, Sub = 011,  
Mul = 100, Div = 101

Μπορούν να χρησιμοποιηθούν για σύγκριση

**if** (load < store) **then...**

# Data Types - Scalar Types

- Οι προκαθορισμένοι τύποι απαρίθμησης της γλώσσας είναι:

-*Character*

-*BIT*: {'0', '1'}

-*BOOLEAN*: {FALSE, TRUE}

-*SEVERITY\_LEVEL*: {NOTE, WARNING,  
ERROR, FAILURE}

π.χ. - STD\_ULOGIC είναι ένας τύπος απαρίθμησης (ορίζεται στο LOGIC\_1164)

**type** STD\_ULOGIC **is** (            -- is used for single bit data type

'U',            --Uninitialized

'Z',    -- high impedance

'X',            --Forcing unknown

'L',    -- weak 0

'0',    -- forcing 0

'H',            -- weak 1

'1',            -- forcing 1

'-'    -- Don't care

'W',    -- Weak unknown );

# Data Types

β) Integer: ορίζει μια περιοχή ακεραίων αριθμών

-π.χ. **type** CountValue **is range** 0 **to** 15;

**variable** a: **integer range** -255 **to** 255

είναι 98\_71\_28=987128

-Range :-( $2^{31}-1$ ) εως ( $2^{31}-1$ ) (ascending, descending)

γ) Floating point -Δεν υποστηρίζεται από τα synthesis tools λόγω μεγάλων απαιτήσεων σε resources. Χρησιμοποιούνται για αναπαράσταση πραγματικών αριθμών.

-range:-1.0E38 εως 1.0E38

π.χ. **type** probability **is range** 0.0 **to** 1.0

δ) Physical(δεν υποστηρίζεται από τα synthesis tools).

Χρησιμοποιείται για αναπαράσταση φυσικών ποσοτήτων

όπως χρόνος, τάση κτλ. Π.χ.

**type** resistance **is range** 0 **to** 1E9

**units**

ohm;

**end units** resistance

# Data Types- Composite Types

- Αναπαριστούν μια συλλογή από τιμές οι οποίες όλες μαζί απαρτίζουν ένα array ή ένα record.

Μόνο τα στοιχεία ενός array πρέπει να είναι του ίδιου τύπου.

- Composite array data type:χρήσιμο για μοντελοποίηση γραμμικών δομών (RAM,ROM).

π.χ. **type** data\_word **is array** (7 **downto** 0) **of** STD\_ULOGIC;

**type** ROM **is array** (0 **to** 125) **of** data\_word;

**type** string **is array** (positive **range**<>) **of** character;

--μη περιορισμένο array

- Είναι επιτρεπτός ο καθορισμός οποιασδήποτε διάστασης στο array.(τα synthesis tools υποστηρίζουν μια ή δυο).



# Data Types- Composite Types

- **Composite record data type:** συλλογή από ονομασμένα στοιχεία, πιθανώς διαφορετικού τύπου.

Χρήσιμος για μοντελοποίηση πακέτων δεδομένων.

```
type PIN_TYPE is range 0 to 10;
```

```
type MODULE is
```

```
  record
```

```
    SIZE                :INTEGER range 20 to 200;
```

```
    CRITICAL_DLY       :TIME;
```

```
    NO_INPUTS          :PIN_TYPE;
```

```
  end record;
```



# Operators

- **Logical Operators:**

- and, or, nand, nor, xor, xnor, not

- Αυτοί οι τελεστές ορίζονται για

- τους προκαθορισμένους τύπους BIT and BOOLEAN

- για array μιας διάστασης των BIT and BOOLEAN

- **Relational operators:**

=    /=  
<    <=  
>    >=

- Ο τύπος του αποτελέσματος είναι boolean

- Οι τελεστές <, <=, >, >= ενεργούν σε οποιοδήποτε scalar type.

π.χ. BIT\_VECTOR('0', '1', '1') < BIT\_VECTOR('1', '0', '1')

είναι true εφόσον η σύγκριση εκτελείται κατά ένα στοιχείο ανά μια χρονική στιγμή από αριστερά προς τα δεξιά.



# Operators

## – Shift Operators:

sll srl sla sra rol ror

π.χ. “1001010” sll 2 is “0101000”

## – Adding operators

+ - &(concatenation)

π.χ. ‘0’ & ‘1’ results of “01”

## – Multiplying operators

\* / mod rem

- \* για πολλαπλασιασμό

- / για διαίρεση

- mod για modulus

- rem : remainder

## – Miscellaneous operators

abs (absolute) , \*\* (exponentiation)



# Test Benches

- Έλεγχος σχεδιασμού με χρήση εξομοίωσης.
- Χρήση ενός test bench μοντέλου
  - εφαρμόζει μια ακολουθία από τιμές ελέγχου στις εισόδους.
  - περιλαμβάνει ένα architecture body το οποίο περιέχει ένα στιγμιότυπο της υπό έλεγχο σχεδίασης.
  - Παρακολούθηση των output signals με χρήση εξομοιωτή (χρήση της assert).

```
process(clk)
```

```
begin
```

```
assert now < 900 ns
```

```
report “stopping simulator (max simulation time 900 ns)
```

```
severity Failure;
```

```
end process;
```

- Πιθανή η χρήση του ίδιου testbench κατά τη διάρκεια διαφορετικών φάσεων της σχεδίασης.

# Sequential Statements

Εκτελούνται σύμφωνα με τη σειρά με την οποία εμφανίζονται στο πρόγραμμα. Επιτρέπονται μόνο μέσα σ' μια διεργασία.

- IF STATEMENT
- CASE STATEMENT
- LOOP STATEMENT
  - FOR LOOPS
  - WHILE LOOPS
- ASSERTION STATEMENT
- WAIT STATEMENT

## IF Statement

- **if** *CONDITION* **then**  
    sequential statements  
**end if;**
- **if** *CONDITION* **then**  
    sequential statements  
**else**  
    sequential statements  
**end if;**
- **if** *CONDITION* **then**  
    sequential statements  
**elsif** *CONDITION* **then**  
    sequential statements  
    ...  
**else**  
    sequential statements  
**end if;**

```
entity IF_STATEMENT is
  port (A, B, C, X : in bit_vector (3 downto 0);
        Z          : out bit_vector (3 downto 0));
end IF_STATEMENT;
architecture EXAMPLE of IF_STATEMENT is
begin
  process (A, B, C, X)
  begin

    if (X = "1111") then
      Z <= B;
    elsif (X > "1000") then
      Z <= C;
    else
      Z <= a;
    end if;
  end process;
end EXAMPLE;
```



# CASE Statement

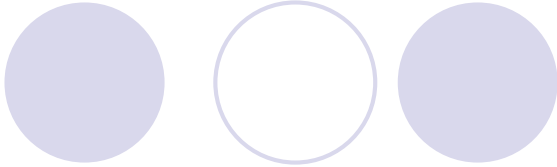
```
case EXPRESSION is
    when VALUE_1 =>
        sequential statements

    when VALUE_2 | VALUE_3 =>
        sequential statements

    when VALUE_4 to VALUE_N
        sequential statements

    when others
        sequential statements

end case ;
```



```
entity CASE_STATEMENT is
    port (A, B, C, X : in integer range 0 to
        15;
         Z : out integer range 0 to
        15;
    end CASE_STATEMENT;

    architecture EXAMPLE of CASE_STATEMENT is
    begin
        process (A, B, C, X)
        begin
            case X is
                when 0 =>
                    Z <= A;
                when 7 | 9 =>
                    Z <= B;
                when 1 to 5 =>
                    Z <= C;
                when others =>
                    Z <= 0;
            end case;
        end process;
    end EXAMPLE;
```



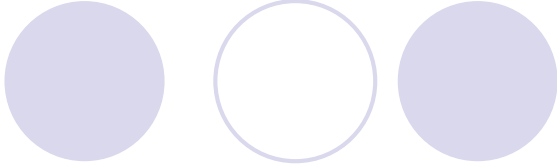
# LOOP Statement

- FOR LOOP :

```
for IDENTIFIER in DISCRETE_RANGE
loop
    sequential statements
end loop ;
```

- WHILE LOOP :

```
while CONDITION loop
    sequential statements
end loop ;
```



```
Entity ex is
    port(a,b,c: in std_logic_vector(4
        downto 0);
         q: out std_logic_vector(4
        downto 0));
end;
Architecture rtl of ex is
begin
    process(a,b,c)
    begin
        for i in 0 to 4 loop
            if a(i)='1' then
                q(i) <= b(i);
            else
                q(i) <= c(i)
            end if
        end loop;
    end process;
end;
```



# ASSERTION Statement

```
assert BOOLEAN EXPRESSION  
report EXPRESSION ;  
severity EXPRESSION ;
```

Αυτού του είδους η πρόταση χρησιμοποιείται για να μπορούμε να

τεστάρουμε ότι το μοντέλο που έχουμε κατασκευάσει λειτουργεί κανονικά. Πιο συγκεκριμένα αν η συνθήκη που ελέγχεται με την εντολή *assert* δεν τηρείται όταν το μοντέλο εξομοιώνεται, ένα μήνυμα συγκεκριμένης σοβαρότητας παρουσιάζεται στο χρήστη. Υπάρχουν 4 διαφορετικά επίπεδα σοβαρότητας που μπορούν να εμφανίζονται στο μήνυμα. Αυτά είναι τα:

- **Note**
- **Warning**
- **Error**
- **Failure**



# WAIT Statement

Μια *Wait Statement* σταματάει την εκτέλεση μιας διεργασίας, η οποία συνεχίζεται μόλις ικανοποιηθεί η συνθήκη της εντολής.

Αν δεν υπάρχει συνθήκη τότε η διεργασία δεν ενεργοποιείται ξανά.

Μπορεί να χρησιμοποιηθεί μόνο σε διεργασίες που δεν

διαθέτουν *sensitivity list*.

# WAIT Statement

- **wait for a specific time:**  
wait for  
*SPECIFIC\_TIME*;
- **wait for a signal event**  
wait on  
*SIGNAL\_LIST*;
- **wait for a true condition (requires an event)**  
wait until  
*CONDITION*;
- **indefinite (process is never reactivated)**  
wait;

```
entity FF is
    port (D, CLK : in bit;
          Q : out bit);
end FF;

architecture Beh of FF is
begin
    process
    begin
        wait on CLK;
        if (CLK = '1') then
            Q <= D;
        end if;
    end process;
end Beh;
```

Οι εκφράσεις: *wait on CLK, if (CLK = '1')*  
*then*  
μπορούν να αντικατασταθούν από την  
**wait until CLK='1'**;

# Concurrent Statements

Πρόκειται για προτάσεις που εκτελούνται την ίδια χρονική στιγμή, ανεξάρτητα της σειράς με την οποία εμφανίζονται στο πρόγραμμα.

Διακρίνονται δύο κατηγορίες τέτοιων προτάσεων :

- **Conditional Signal Assignment (when-else) :**  
Η απόφαση βασίζεται σε παραπάνω από ένα σήματα.  
Ισοδύναμη των ακολουθιακών δηλώσεων **if..elsif..else**.
- **Selected Signal Assignment (with-select-when) :**  
Η απόφαση βασίζεται στις τιμές ενός σήματος.  
Ισοδύναμη των ακολουθιακών δηλώσεων **case..when..**

# Conditional Signal Assignment

```
TARGET <= VALUE_1 when
    CONDITION_1
else
    VALUE_2 when
    CONDITION_2
else
    .....
    VALUE_n;
```

```
entity C_A is
    port (A, B, C, X : in bit_vector (3 downto 0);
          Z_CONC : out bit_vector (3 downto 0);
          Z_SEQ : out bit_vector (3 downto 0))
end C_A;

architecture EXAMPLE of C_A is
begin
Concurrent version of conditional signal assignment
    Z_CONC <= B when X = "1111" else
              C when X > "1000" else
              A;

Equivalent sequential statements
    process (A, B, C, X)
    begin
        if (X = "1111") then
            Z_SEQ <= B
        elsif (X > "1000") then
            Z_SEQ <= C;
        else
            Z_SEQ <= A;
        end if;
    end process;
end EXAMPLE;
```

# Selected Signal Assignment

**with** *EXPRESSION* **select**

```
TARGET <= VALUE_1 when CHOICE_1,
    VALUE_2 when CHOICE_2 |
    CHOICE_3,
    VALUE_3 when CHOICE_4 to
    CHOICE_5,
    .....
    VALUE_n when others;
```

```
entity S_A is
    port (A, B, C, X : in integer range 0 to 15;
          Z_CONC : out integer range 0 to 15;
          Z_SEQ : out integer range 0 to 15)
end S_A;

architecture EXAMPLE of S_A is
begin
Concurrent version of selected signal assignment
    with X select
        Z_CONC <= A when 0,
                B when 7 | 9,
                C when 1 to 5,
                0 when others;

Equivalent sequential statements
    process (A, B, C, X)
    begin
        case X is
            when 0 => Z_SEQ <= A;
            when 7 | 9 => Z_SEQ <= B;
            when 1 to 5 => Z_SEQ <= C;
            when others => Z_SEQ <= 0;
        end case;
    end process;
end EXAMPLE;
```



# Subprograms

Παρόμοια με τις τυπικές γλώσσες προγραμματισμού, η VHDL παρέχει τη δυνατότητα χρήσης υποπρογραμμάτων στη μορφή *συναρτήσεων (Functions)* και *διαδικασιών (Procedures)*.

- Ο ορισμός ενός υποπρογράμματος αποτελείται από τη *δήλωση* υποπρογράμματος, όπου καθορίζεται η λίστα των παραμέτρων του υποπρογράμματος, και από το *σώμα* του υποπρογράμματος το οποίο και καθορίζει τη συμπεριφορά του.
- Ο κώδικας που περιέχεται στο *σώμα* του υποπρογράμματος εκτελείται πάντοτε ακολουθιακά.
- Τα υποπρογράμματα μπορούν να καθοριστούν στα εξής σημεία μέσα στον

κώδικα VHDL:

- **Package**
- **Entity**
- **Architecture**
- **Process**
- **Subprogram**



# Subprograms

## Παράμετροι και καταστάσεις λειτουργίας

- **Formal parameters (parameter declaration):**
  - default mode: IN
  - default parameter class for mode IN: constant, for mode OUT/INOUT: variable
- **Actual parameters (subprogram call)**
  - must match classes of formal parameter
  - class constant matches actual constants, signals or variables

# Functions

- Αυθαίρετος αριθμός παραμέτρων εισόδου
- Οι παράμετροι έχουν πάντα κατάσταση λειτουργίας *IN* ,και δεν μπορούν να δηλωθούν ως *variables*.
- Ακριβώς μία επιστρεφόμενη τιμή
- Δεν επιτρέπονται Wait Statements
- Χρησιμοποιούνται ως *expressions* σε προτάσεις της VHDL(παράλληλες και ακολουθιακές).  
*function call*  $\Leftrightarrow$  *VHDL expression*

# Procedures

- Αυθαίρετος αριθμός παραμέτρων εισόδου που μπορούν να έχουν κάθε πιθανή κατάσταση λειτουργίας (IN,OUT,INOUT).
- Δεν επιστρέφεται κάποια τιμή, παρότι η δεσμευμένη λέξη *return* μπορεί να χρησιμοποιηθεί για να υποδείξει τον τερματισμό του υποπρογράμματος.
- Ανάλογα με τη θέση της μέσα στον VHDL κώδικα, είτε μέσα σε μια αρχιτεκτονική είτε σε μια διεργασία,μια διαδικασία στο σύνολό της εκτελείται παράλληλα ή ακολουθιακά, αντίστοιχα.
- *procedure call*  $\Leftrightarrow$  *VHDL statement*

# Subprogram Examples

```
architecture EXAMPLE of FUNCTIONS is
  function COUNT_ZEROS (A : bit_vector)
    return integer is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS + 1;
      end if;
    end loop;
    return ZEROS;
  end COUNT_ZEROS;

  signal WORD: bit_vector(15 downto 0);
  signal WORD_0: integer;
  signal IS_0: boolean;
begin
  WORD_0 <= COUNT_ZEROS(WORD);
  process
  begin
    IS_0 <= true;
    if COUNT_ZEROS("01101001") > 0 then
      IS_0 <= false;
    end if;
    wait;
  end process;
end EXAMPLE;
```

```
architecture EXAMPLE of PROCEDURES is
  procedure COUNT_ZEROS
    (A: in bit_vector;
     signal Q: out integer) is
    variable ZEROS : integer;
  begin
    ZEROS := 0;
    for I in A'range loop
      if A(I) = '0' then
        ZEROS := ZEROS + 1;
      end if;
    end loop;
    Q <= ZEROS;
  end COUNT_ZEROS;

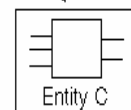
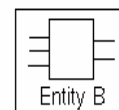
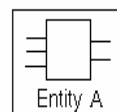
  signal COUNT: integer;
  signal IS_0: boolean;
begin
  process
  begin
    IS_0 <= true;
    COUNT_ZEROS("01101001", COUNT);
    wait for 0 ns;
    if COUNT > 0 then
      IS_0 <= false;
    end if;
    wait;
  end process;
end EXAMPLE;
```

# Packages

- Ένα πακέτο είναι μια συλλογή από ορισμούς στους οποίους μπορεί να γίνει αναφορά από πολλές VHDL σχεδιάσεις την ίδια χρονική στιγμή.
- Ένα πακέτο μπορεί να περιλαμβάνει : *functions, procedures, constants, data objects, subtypes, components.*
- Τις περισσότερες φορές ένα πακέτο αποτελείται από δύο τμήματα: *τη δήλωση και το σώμα του πακέτου.*
- Για να μπορεί να γίνει αναφορά σ' ένα πακέτο πρέπει αυτό να δηλωθεί με χρήση της πρότασης *use*.

```
package PROJECT_PACK is
  -- constants
  -- data types
  -- components
  -- sub routines
end PROJECT_PACK;
```

```
use work.PROJECT_PACK.all;
```



# Package Syntax

## Package declaration:

```
package IDENTIFIER is
  Declaration of
  -- types and subtypes
  -- subprograms
  -- constants, signals and
  shared
    variables
  -- files
  -- aliases
  -- components
end [ package ] [ IDENTIFIER ] ;
```

## Package body declaration:

```
package body IDENTIFIER is
  Definition of previously
  declared
  -- constants
  -- subprograms
  Declaration/definition of
  additional
  -- types and subtypes
  -- subprograms
  -- constants, signals and shared
    variables
  -- files
  -- aliases
  -- components
end [ package body ] [ IDENTIFIER ]
;
```

# Package Example

```
package PKG is
  type T1 is ...
  type T2 is ...
  constant C : integer;
  procedure P1 (...);
end PKG;
package body PKG is
  type T3 is ...

  C := 17;

  procedure P1 (...) is
    ...
  end P1;

  procedure P2 (...) is
    ...
  end P2;
end PKG;
```

```
library STD; -- VHDL default
library WORK; -- VHDL default
use STD.standard.all; -- VHDL default
use work.PKG.all;

entity EXAMPLE is
end EXAMPLE;

architecture BEH of EXAMPLE is
  signal S1 : T1;
  signal S2 : T2;
  signal S3 : T3; -- error: T3 not declared

begin

  P1 (...);
  P2 (...); -- error: P2 not declared

end BEH;
end PKG;
```



# Libraries

- Πρόκειται για συλλογές από μεταγλωττισμένες μονάδες σχεδίασης της VHDL.
- Στη VHDL μια «βιβλιοθήκη» είναι ένα λογικό όνομα μέσω του οποίου δίνεται η δυνατότητα να γκρουπάρονται τα μεταγλωττισμένα αντικείμενα.
- Το λογικό αυτό όνομα μπορεί να αντιστοιχίζεται σ' ένα άλλο λογικό όνομα, αλλά απαραίτητα πρέπει να αντιστοιχίζεται σε ένα φυσικό μονοπάτι σ' ένα μέσο αποθήκευσης.
- Αν δεν καθορίζεται μια βιβλιοθήκη τότε οι μονάδες σχεδίασης μεταγλωττίζονται εξ ορισμού στη βιβλιοθήκη *work*.



# The LIBRARY Statement

Για να μπορεί να είναι ορατή η βιβλιοθήκη που έχει δημιουργήσει ένας χρήστης πρέπει αυτή να δηλώνεται ρητά στον VHDL κώδικα. Αυτό γίνεται με χρήση της statement *library*.

- **Μια τέτοια δήλωση μπορεί να τοποθετηθεί μόνο μπροστά από κάθε μονάδα σχεδίασης και είναι έγκυρη μόνο για τη μονάδα που ακολουθεί.**
- **Οι δευτερεύουσες μονάδες σχεδίασης «κληρονομούν» δηλώσεις βιβλιοθήκης που εφαρμόζονται στην αντιστοιχή τους πρωτεύουσα μονάδα.**
- **Οι βιβλιοθήκες WORK και STD είναι εξ ορισμού ορατές και έτσι δεν χρειάζεται να δηλώνονται.**

# The USE Statement

Μια δήλωση βιβλιοθήκης από μόνη της δεν είναι αρκετή ώστε τα αντικείμενα που περιέχονται σ' αυτήν να είναι προσβάσιμα. Για να γίνει αυτό πρέπει να γίνει χρήση της πρότασης *use*.

- Μια τέτοια δήλωση μπορεί να τοποθετηθεί μόνο μπροστά από κάθε μονάδα σχεδίασης και είναι έγκυρη μόνο για τη μονάδα που ακολουθεί.
- Οι δευτερεύουσες μονάδες σχεδίασης «κληρονομούν» τις δηλώσεις που εφαρμόζονται στην αντιστοιχή τους πρωτεύουσα μονάδα.
- Μπορεί να προσφέρει πρόσβαση σε μεμονωμένες μονάδες:
  - Μονάδες σχεδίασης (π.χ. *entities*)
  - Αντικείμενα πακέτων (π.χ. *constants*)
  - Η χρήση της λέξης 'all' δίνει πρόσβαση σε όλα τα αντικείμενα
- Σε περίπτωση παράλειψης της εντολής απαιτείται το πλήρες λογικό όνομα του κάθε αντικειμένου.

# Configuration

Στη VHDL μια *entity* μπορεί να έχει παραπάνω από μία αρχιτεκτονικές. Η ιδιότητα αυτή αποδεικνύεται ιδιαίτερα σημαντική για τη περίπτωση που μια σχεδίαση περιγράφεται σε διάφορα επίπεδα αφαίρεσης π.χ. *RTL, behavioral, gate level*. Με χρήση της μονάδας σχεδίασης **configuration** μπορεί να καθοριστεί ποια αρχιτεκτονική θα χρησιμοποιείται κάθε φορά για την συγκεκριμένη οντότητα.

```
Entity HALFADD is
  Port (A, B : in bit;
        SUM, CARRY: out bit );
end HALFADD ;

Architecture X of HALFADD is
begin
  .....
end X;

Architecture Y of HALFADD is
begin
  .....
end Y;

Configuration cfg1 of HALFADD is
  for x
  end for;
End cfg1;
Configuration cfg2 of HALFADD is
  for Y
  end for;
End cfg2;
```

# Generics

- Ένα *generic* μπορεί να χρησιμοποιηθεί όταν επιθυμούμε να παραμετροποιήσουμε μια σχεδίαση. Με τον τρόπο αυτό μπορούμε να εισάγουμε μια πληροφορία μέσα στο μοντέλο, π.χ. χρονική πληροφορία, η οποία μπορεί να αλλάζει κάθε φορά που ένα στιγμιότυπο του *component* που την περιέχει, χρησιμοποιείται. Αυτό δηλώνεται μέσω της εντολής *generic map*.
- Ένα *generic* δηλώνεται στο σώμα μιας οντότητας πριν τη δήλωση *port*.
- Μεταχειρίζεται ως *constant* μέσα σε μια αρχιτεκτονική.

```
Entity generic_ex is
  generic (delay:time:=
10ns);
  port(a,b: in std_logic;
        c: out std_logic );
end generic_ex;
Architecture g_beh of
generic_ex is
begin
  c<=a and b after delay;
end generic_beh;
```

# Generate Statement

- Στη περίπτωση που το ίδιο *component* πρέπει να χρησιμοποιηθεί πολλές φορές στην ίδια αρχιτεκτονική, είναι πιο αποδοτικό να δημιουργήσουμε ένα *loop* από *port map* εντολές. Αυτό γίνεται εφικτό με χρήση της εντολής *generate*.
- [label:] **For**  
<loop\_index><iteration scheme>  
**generate**  
    <label>: <entity> **port map** (...);  
**end generate** [label];

```
Entity top is
  port(a,b: in std_logic_vector(4 downto 0);
        q: out std_logic_vector(4 downto 0));
end ;
Architecture rtl of top is
  component ci
    port (a,b: in std_logic;
          q: out std_logic);
  end component;
  For U1: c1 Use entity work.c1(rtl);
begin
  c_gen: For i in 0 to 4 generate
    U: c1 port map (a(i),b(i),q(i));
  end generate c_gen;
end;
```

# Development steps

- Typical programming language

- **Compilation**
- **Link and Load**
- **Execute**

- VHDL

- **Analysis-Sequence of Compilation**
- **Simulation Flow**
  - **Elaboration**
  - **Initialisation**
  - **Execution**
- **Synthesis**

# Simulation

- **Sequence of Compilation**
- **Simulation Flow**
- **Testbenches**

# Design Units



Όλα τα μέρη μιας VHDL σχεδίασης πρέπει να αναλυθούν/μεταγλωττιστούν για να μπορούν να χρησιμοποιηθούν στη συνέχεια για τις διαδικασίες της εξομοίωσης και της σύνθεσης

- **entity/architecture**
- **package/package body**
- **Configuration**

Από αυτές οι *entity*, *package* και *configuration* θεωρούνται πρωτεύουσες μονάδες ενώ οι *architecture* και *package body* δευτερεύουσες.

Ένα αρχείο πρέπει να περιλαμβάνει τουλάχιστον μία μονάδα σχεδίασης

για να μπορεί να γίνει αποδεκτό από τον *compiler*.

# Sequence of Compilation

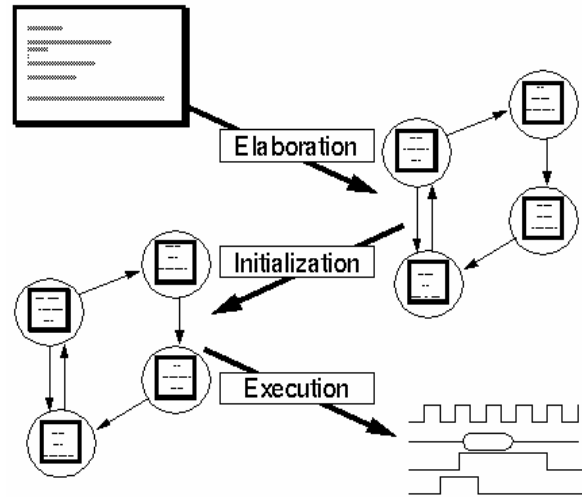


- Οι πρωτεύουσες μονάδες σχεδίασης πρέπει να αναλύονται πριν από τις δευτερεύουσες, καθώς οι τελευταίες χρειάζονται πληροφορίες που υπάρχουν στις πρώτες.
  - entity before architecture(s)**
  - package before package body**
- Τα στοιχεία στα οποία γίνονται αναφορές από άλλα πρέπει επίσης να αναλύονται πρώτα.
  - package before entity/architecture**
  - configuration after entity/architecture**

# Simulation flow

Η εξομοίωση ενός VHDL μοντέλου πραγματοποιείται σε τρεις φάσεις:

- Design elaboration
- Signal initialisation
- Execution



# Simulation flow

## Elaboration:

- Στη φάση αυτή δημιουργούνται τα στοιχεία του μοντέλου που θα εξομοιωθεί, όπως *ports, signals, processes*.
- Οι διεργασίες και οι παράλληλες δηλώσεις της όλης σχεδίασης συνδυάζονται σ'ένα επικοινωνιακό μοντέλο, το οποίο καθορίζει ποια διεργασία μπορεί να ενεργοποιείται και από ποια.

## Initialisation:

- Ανάθεση αρχικών τιμών στα σήματα.
- Η αρχική τιμή είναι αυτή που καθορίζεται στη δήλωση του σήματος ή, αν δεν καθορίζεται εκεί, η πρώτη τιμή στον ορισμό του τύπου του σήματος.
- Στο τέλος αυτής της φάσης κάθε διεργασία εκτελείται μια φορά μέχρι να ανασταλλεί, χωρίς όμως να ενημερώνονται οι τιμές των σημάτων.

## Execution:

- Στη φάση αυτή η εξομοίωση του μοντέλου γίνεται πραγματικότητα.
- Κάθε σήμα μπορεί να αναπαρασταθεί και να ελεγχθεί.

# Delta Cycles

- Πρόκειται για τον κύκλο εξομοίωσης που περιλαμβάνει τη φάση της ενημέρωσης σήματος και τη φάση της εκτέλεσης διεργασίας.
- Στην αρχή όλα τα σήματα ενημερώνονται, και δημιουργείται μια λίστα με διεργασίες που θα εκτελεστούν με τις αλλαγές σήματος. Όλες οι διεργασίες αυτής της λίστας εκτελούνται η μία μετά την άλλη στο πρώτο *delta cycle*. Όταν η εκτέλεση ολοκληρωθεί, θα έχει ως αποτέλεσμα νέα ενημέρωση σήματος. Όπως και πριν μια νέα λίστα διεργασιών που επηρεάζονται από την αλλαγή σήματος, θα δημιουργηθεί. Αυτό συνεχίζεται μέχρι η λίστα διεργασιών να είναι κενή.
- Τότε οι δηλώσεις που προκαλούν πραγματική χρονική μεταβολή ('*wait for ...*', '*... after ...*') εκτελούνται και ο χρόνος εξομοίωσης προχωρά για το συγκεκριμένο χρονικό διάστημα. Στη συνέχεια τα *delta cycles* για το νέο χρονικό σημείο εξομοίωσης ξεκινούν.

# Delta Cycles-Example

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity DELTA is
port (A, B : in std_ulogic;
      Y, Z : out std_ulogic);
end DELTA;

architecture EXAMPLE of DELTA is
signal X : std_ulogic;
begin
process (A, B, X)
begin
Y <= A;
X <= B;
Z <= X;
end process;
end EXAMPLE;
```

## Event on B (first delta cycle)

### future value of

Y receives the **current value of A** (no change)  
X receives the **current value of B (new value)**  
Z receives the **current value of X** (no change)

### signal update

## Event on X (second delta cycle)

### future value of

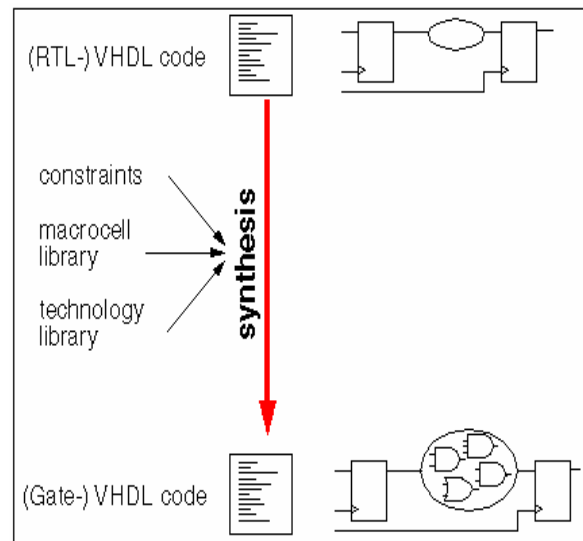
Y receives the **current value of A** (no change)  
X receives the **current value of B** (no change)  
Z receives the **current value of X (new value)**

### signal update

No further events on A, B, X

# Synthesis

- Μετατροπή μιας αφηρημένης περιγραφής σε μια πιο λεπτομερή περιγραφή.
- Η μετατροπή εξαρτάται από δύο κυρίως παράγοντες:
  - *target technology*
  - *tool capabilities*



# Βιβλιογραφία.

1. Peter J. Ashenden “*The VHDL Cookbook.*”
2. Peter J. Ashenden “*The Designer’s Guide to VHDL.*”
3. Stefan Sjöholm, Lennart Lindh “*VHDL for Designers.*”