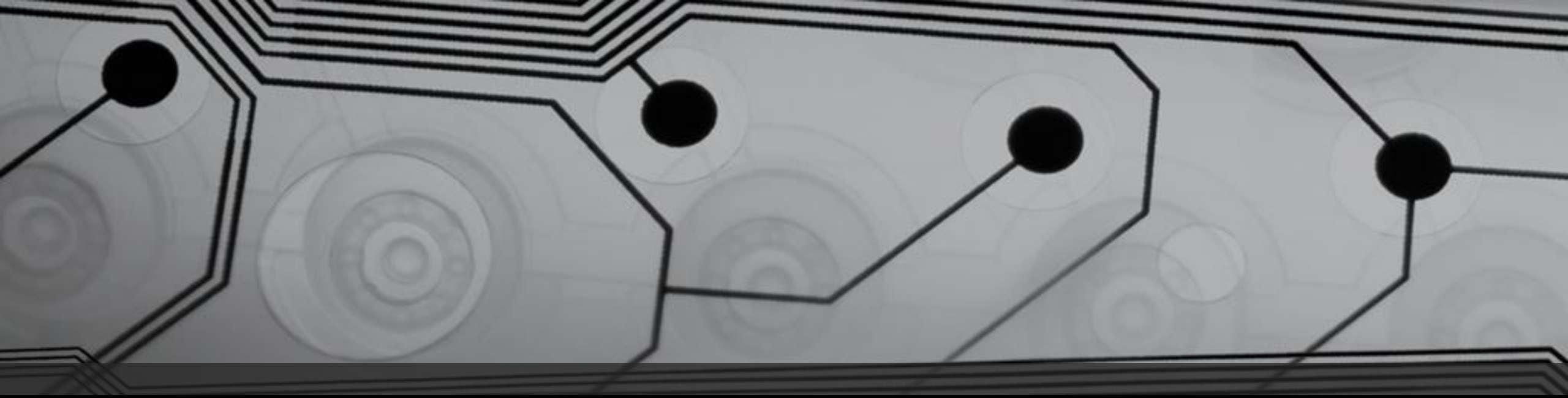


VHDL Introduction

Subtitle

Getting Started …

VHDL

means …

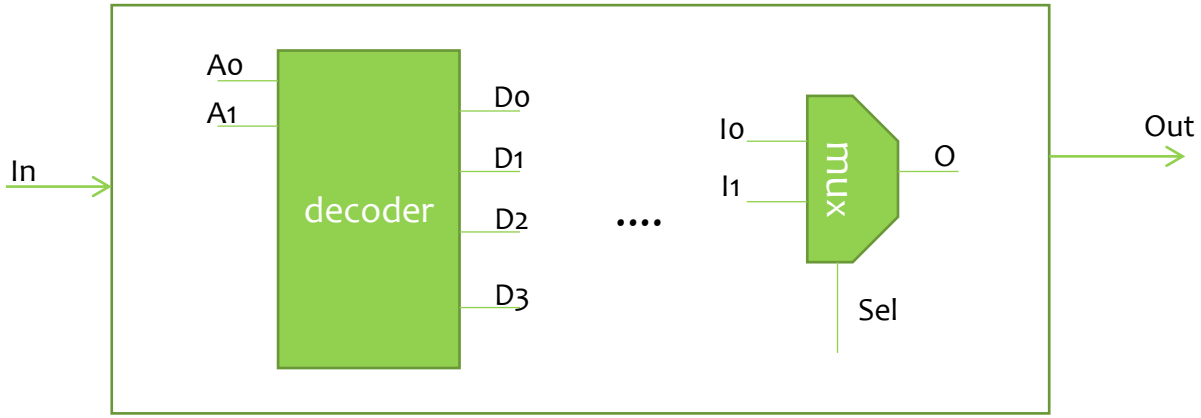
Very Hard Difficult Language

That's a lie!!!

- τα αρχικά VHDL είναι συντομογραφία του VHSIC Hardware Description Language, ενώ το VHSIC αντιπροσωπεύει με τη σειρά του «Very High Speed Integrated Circuits» … λίγο μπερδεμένο ;

VHDL Introduction

- Εμείς απλά θα την λέμε «γλώσσα περιγραφής υλικού» γιατί αυτό ακριβώς κάνει.
- Είναι μία γλώσσα προγραμματισμού που μας επιτρέπει να σχεδιάζουμε πολύπλοκα ψηφιακά συστήματα σε ένα δυναμικό περιβάλλον.
- Κάθε μεγάλο ηλεκτρονικό σχέδιο για είναι πιο εύκολα κατανοητό το χωρίσουμε σε τμήματα (blocks) πχ ένας decoder,multiplexer, κτλ… και τα blocks αυτά συνδέονται μεταξύ τους και σχηματίζουν το τελικό σχέδιο.



VHDL Introduction

- Ένα VHDL design μπορεί να έχει ένα block η περισσότερα.
- Από δω και στο εξής αντί για block θα λέμε entity.
- Στην VHDL ένα entity περιγράφει το «interface» ενός block και ένα δεύτερο τμήμα της VHDL, που σχετίζεται με το entity περιγράφει πως λειτουργεί το συγκεκριμένο block.

```
entity NAME_OF_ENTITY is
        port (signal_names: mode type;
              signal_names: mode type;
                    :
              signal_names: mode type);
end NAME_OF_ENTITY ;
```

- Ένα entity πάντα ξεκινάει με τη λέξη **entity,** που ακολουθείται από το όνομά του και τη λέξη **is.**

- Μετά ακολουθούν οι ορισμοί των σημάτων εισόδου/εξόδου μέσα στο κομμάτι που ξεκινάει με τη λέξη **port.**

- Ένα entity τελειώνει πάντα με τη λέξη **end** και ακολουθείται από το όνομα που του δώσαμε**.**

VHDL Introduction

signal_names πρόκειται για μία λίστα από έναν ή περισσότερα σήματα ορισμένα από το χρήστη , τα οποία χωρίζουμε με κόμμα.

Mode είναι μία από τις δεσμευμένες λέξεις της VHDL η οποία χρησιμοποιείται για να περιγράψουμε την κατεύθυνση του σήματος. Δηλ.

in ορίζουμε ένα σήμα εισόδου
out ορίζουμε ένα σήμα εξόδου
inout σημαίνει ότι το σήμα είναι είσοδος και έξοδος

type είναι ο τύπος του σήματος, πχ bit, bit_vector, std_logic, integer, character κτλ.

- *bit* – μπορεί να πάρει τιμή 0 και 1
- *bit_vector* – είναι ένα πίνακας με δυαδικές τιμές (πχ. bit_vector (0 to 7)
- *std_logic* παίρνει τιμές 0, 1, X(αδιάφορο), Z(high imbedance)
- *boolean* – παίρνει τιμές TRUE και FALSE
- *integer* –a range of integer values
- *real* –a range of real values
- *character* – οποιοσδήποτε χαρακτήρας ('a', 'B' για ένα χαρακτήρα ενώ μέσα σε διπλά quotes για strings πχ "This is a string")

```
entity NAME_OF_ENTITY is
        port (signal_names: mode type;
              signal_names: mode type;
                    :
              signal_names: mode type);
end NAME_OF_ENTITY ;
```

Παράδειγμα

```
entity nor_gate is
        port (a , b : in bit;
                    y : out bit);
end nor_gate;
```

--- έχουμε ένα νέο entity , το όνομα του είναι nor_gate και μέσα στο port περιγράφουμε το interface του δηλαδή τα σήματα που είναι είσοδοι ή έξοδοι .

--- με τη λέξη **bit** λέμε ότι το σήμα αυτό μπορεί να πάρει τιμές 0 ή 1.

--- Δηλαδή το σήμα a λέμε ότι είναι *type* bit.
Το σήμα a θα μπορούσε να είναι type STD_LOGIC και τότε θα έπαιρνε τιμές 0, 1, X(αδιάφορο), Z(high imbedance) .

Architecture

- Αφού ορίσουμε το entity στη συνέχεια πρέπει να καθορίσουμε και τη λειτουργία του.
- Αυτό γίνεται στο κομμάτι Architecture του VHDL κώδικά μας.
- Στο κομμάτι αυτό μπορούμε να περιγράψουμε τι κάνει το κύκλωμά μας , με διάφορους τρόπους.
- ‘Ενας από αυτούς είναι αυτό που ονομάζεται Behavoural Design, όπου απλά περιγράφουμε τη σχέση μεταξύ της εισόδου και της εξόδου (πχ με μια boolean έκφραση) όπως θα δούμε σε παράδειγμα στη συνέχεια.
- Ο άλλος τρόπος (Structural Design) είναι να περιγράψουμε το ψηφιακό μας κύκλωμα σαν ένα σύνολο από άλλα entities ή πύλες τα οποία συνδέονται όλα μαζί για να δώσουν την επιθυμητή λειτουργία.
- Μπορούμε όμως να συνδυάσουμε και τους δύο αυτούς τρόπους στη σχεδίαση μας.

Architecture

- **architecture** architecture_name **of** NAME_OF_ENTITY **is**

  -- Declarations

  -- components declarations

  -- signal declarations

  -- constant declarations

  -- process declarations

  -- type declarations

  begin

  -- Statements

- **end** architecture_name;

- Μέχρι τώρα είδαμε σήματα τα οποία μπορεί να είναι είσοδοι ή έξοδοι. Τα σήματα είναι τα wires του σχηματικού τα οποία έχουν τρέχουσες τιμές και θα αλλάξουν τιμή στο μέλλον ανάλογα με την λειτουργία του κυκλώματος. Από την άλλη στη VHDL έχουμε μεταβλητές (Variables) και σταθερές (Constants) οι οποίες χρησιμοποιούνται μέσα σε processes ή συναρτήσεις, με τρόπο παρόμοιο με αυτόν σε άλλες γλώσσες προγραμματισμού. Θα τα περιγράψουμε σύντομα στη συνέχεια.

Constants

- Μία σταθερά μπορεί να έχει μία τιμή για ένα τύπο δεδομένων και δεν μπορεί να αλλάξει κατά τη διάρκεια του simulation. Μία σταθερά δηλώνεται ως εξής

constant *list_of_name_of_constant*: type [:= initial value] ;

(το initial value είναι προαιρετικό)

- Οι σταθερές μπορούν να δηλωθούν στην αρχή του architecture και μπορούν να χρησιμοποιηθούν σε όλο το architecture.
- Οι σταθερές που ορίζονται μέσα σε ένα process μπορούν να χρησιμοποιηθούν μόνο μέσα στο συγκεκριμένο process.
- Παραδείγματα

constant RISE_FALL_TME: time := 2 ns;

constant RISE_TIME, FALL_TIME: time:= 1 ns;

constant DATA_BUS: integer:= 16;

Variables

- Μία μεταβλητή μπορεί να έχει μία τιμή όπως και μία σταθερά αλλά η μεταβλητή
  α. Δηλώνεται μόνο μέσα σε ένα process και χρησιμοποιείται μόνο από αυτό.
  β. Η μεταβλητή μπορεί να αλλάξει τιμή μέσα στο porcess και μάλιστα η τιμή της ανανεώνεται χωρίς καμία καθυστέρηση, αμέσως δηλαδή μόλις εκτελεστεί η εντολή εκχώρησης.
- Μία μεταβλητή δηλώνεται ως εξής

variable *list_of_variable_names*: type [:= initial value] ;

(το initial value είναι προαιρετικό)

- Ακολουθούν μερικά παραδείγματα

variable CNTR_BIT: bit :=0;

variable VAR1: boolean :=FALSE;

variable SUM: integer **range** 0 **to** 256 :=16;

variable STS_BIT: bit_vector (7 **downto** 0);

Η μεταβλητή SUM, είναι ένας integer που έχει εύρος από 0 έως 256 με αρχική τιμή 16 .

Στο τέταρτο παράδειγμα ορίζετε ένας bit vector με 8 στοιχεία: STS_BIT(7), STS_BIT(6),… STS_BIT(0).

Process

- Η γενική μορφή ενός process στη VHDL είναι :

  process_name: **process** (*sensitivity_list*)

  declarations

  begin

  sequential_statements

  end process;

- Το Sensitivity list (optional) είναι μία λίστα από σήματα στα οποία το process λέμε ότι είναι «sensitive».αυτό σημαίνει ότι ένα process εκτελείται μόνο όταν υπάρχει ένα event σε ένα τουλάχιστον από τα σήματα στη λίστα αυτή.

- **Σημείωση.** Ένα process πρέπει να έχει είτε ένα Sensitivity list είτε ένα WAIT statement διαφορετικά δεν εκτελείται.

Παράδειγμα variable μέσα σε process

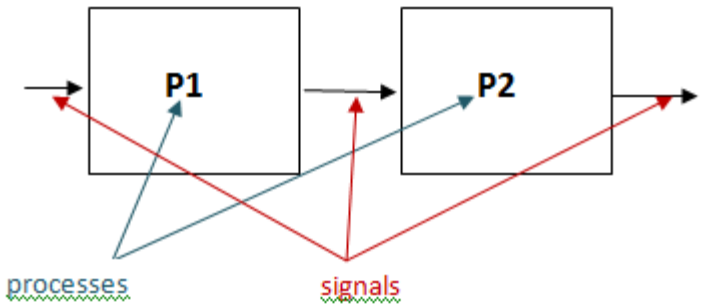
```
count: process (x)
        variable cnt : integer := -1;
begin   cnt:=cnt+1;
end process;
```

Variable_name := expression;

Εντολή εκχώρησης τιμής σε μεταβλητή (χωρίς delay)

Process

- Η επικοινωνία ανάμεσα στα processes επιτυγχάνεται με τα σήματα (signals).



Signals

- Τα σήματα δηλώνονται έξω από το process με τον παρακάτω τρόπο :

  signal *list_of_signal_names*: type [:= initial value] ;

- Παραδείγματα

  signal SUM, CARRY: std_logic;

  signal CLOCK: bit;

  signal TRIGGER: integer :=0;

  signal DATA_BUS: bit_vector (0 to 7);

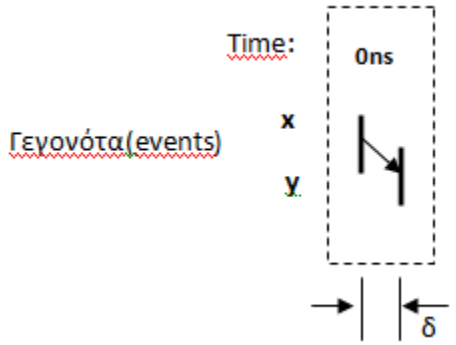
  signal VALUE: integer **range** 0 **to** 100;

Signals

- Τα σήματα γίνονται update όταν εκτελείται μία εντολή εκχώρησης τιμής τους και μετά από μία καθυστέρηση όπως δείχνουμε στη συνέχεια.

SUM <= (A **xor** B) **after** 2 ns;

- Αν δεν έχει οριστεί η καθυστέρηση τότε το σήμα θα πάρει τη νέα του τιμή μετά από ένα χρόνο δ . Πχ y<=x after 0ns



Διαφορά μεταξύ signals και variables

- Θυμηθείτε ότι μία ανάθεση τιμής σε ένα σήμα (signal assignment) δεν εκτελείται αμέσως. Είναι σαν να υπάρχει ένα γεγονός το οποίο έχει προγραμματιστεί για κάποιο σήμα και το γεγονός αυτό δεν έχει άμεση επίδραση στο σήμα .
- Όταν εκτελείται ένα process, τρέχουν οι εντολές του σειριακά από πάνω προς τα κάτω, αλλά όλες οι αλλαγές θα εκτελεστούν αμέσως μόλις ολοκληρωθεί το process.
- Στο παρακάτω process δύο γεγονότα έχουν προγραμματιστεί για τα σήματα x και y

```
signal x,y,z : bit;

process (y)

begin

        x<=y;

        z<=not x;

end process;
```

Διαφορά μεταξύ signals και variables

- Αν αλλάξει το σήμα y τότε υπάρχει ένα προγραμματισμένο γεγονός (event) το οποίο θα αλλάξει το x , ώστε να γίνει ίδιο με το y. Επιπλέον υπάρχει και ένα ακόμα γεγονός για να γίνει το σήμα z το αντίστροφο του x.

```
process (y)
begin
        x<=y;
        z<=not x;
end process;
```

Αν αλλάξει το σήμα y τότε υπάρχει ένα προγραμματισμένο γεγονός (event) το οποίο θα αλλάξει το x , ώστε να γίνει ίδιο με το y. Επιπλέον υπάρχει και ένα ακόμα γεγονός για να γίνει το σήμα z το αντίστροφο του x.

Το ερώτημα εδώ είναι ¨η τιμή του z θα είναι τελικά το αντίστροφο του x?

Η απάντηση είναι όχι , και αυτό γιατί όταν εκτελεστεί η δεύτερη εντολή , το x δεν έχει ακόμα αλλάξει τιμή, οπότε η τιμή του z θα είναι το αντίστροφο του x πριν ξεκινήσει το process.

Διαφορά μεταξύ signals και variables

- Οι μεταβλητές από την άλλη δρουν διαφορετικά. Για παράδειγμα

```
process (y)
        variable x,z : bit;
        begin
                x:=y;
                z:=not x;
end process;
```

- Η τιμή της μεταβλητής z θα είναι το αντίθετο του y και αυτό γιατί η τιμή της μεταβλητής x αλλάζει αμέσως.

ΑΣΚΗΣΗ

- Δίνονται δύο processes, το ένα σας δείχνει τη χρήση μεταβλητών και το άλλο τη χρήση σημάτων . Ποιο θα είναι το αποτέλεσμα για το σήμα RESULT σε κάθε μία από τις δύο περιπτώσεις. Εξηγήστε γιατί .

Process με variables	Process με signals

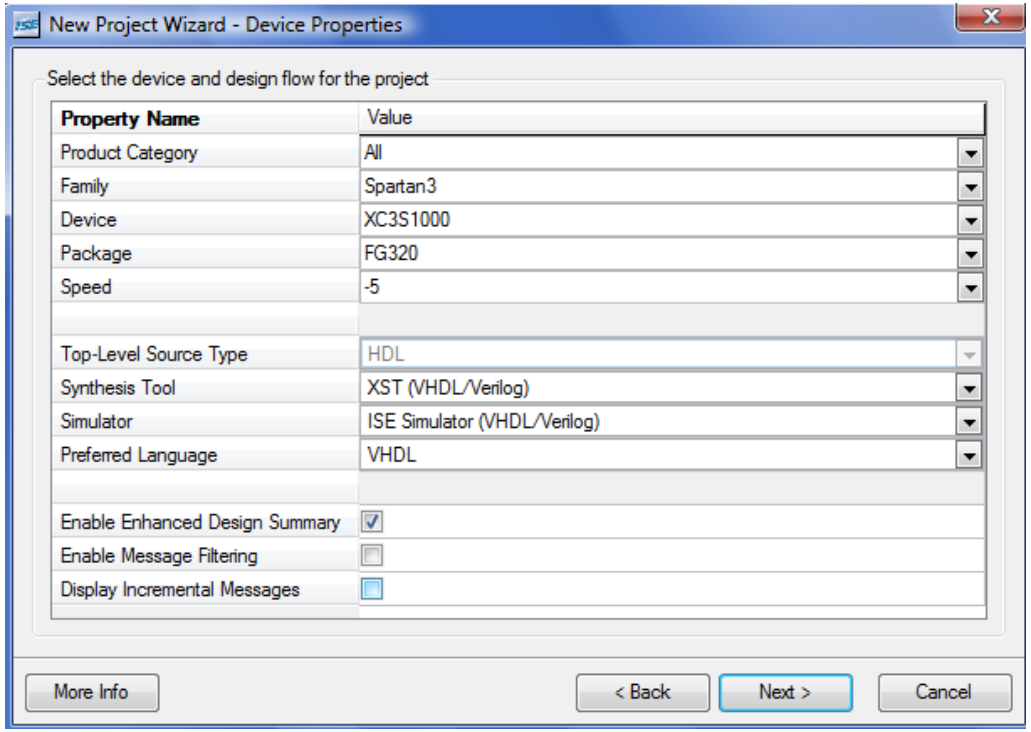
architecture VAR of EXAMPLE is
signal TRIGGER, RESULT: integer := 0;
begin
process
variable variable1: integer :=1;
variable variable2: integer :=2;
variable variable3: integer :=3;
begin
wait on TRIGGER;
variable1 := variable2;
variable2 := variable1 + variable3;
variable3 := variable2;
RESULT <= variable1 + variable2 + variable3;
end process;
end VAR
	architecture SIGN of EXAMPLE is
	signal TRIGGER, RESULT: integer := 0;
	signal signal1: integer :=1;
	signal signal2: integer :=2;
	signal signal3: integer :=3;
	begin
	process
	begin
	wait on TRIGGER;
	signal1 <= signal2;
	signal2 <= signal1 + signal3;
	signal3 <= signal2;
	RESULT <= signal1 + signal2 + signal3;
	end process;
	end SIGN;

Παραδείγματα

- Στη συνέχεια θα χρησιμοποιήσουμε το εργαλείο Xillinx για να φτιάξουμε πύλες και να τις συνδυάσουμε ώστε να φτιάξουμε ένα πιο πολύπλοκο ψηφιακό κύκλωμα, χρησιμοποιώντας VHDL κώδικα .

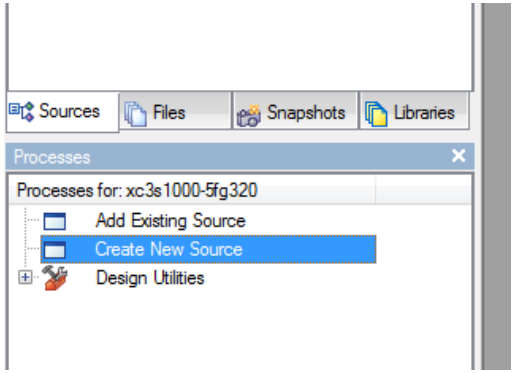
Πύλη NOR

- Στον Project Navigator επιλέγουμε File -> New Project , θα δώσουμε το όνομα basic_gates και θα πατήσουμε Next.
- Συνεχίζουμε με τις παρακάτω επιλογές

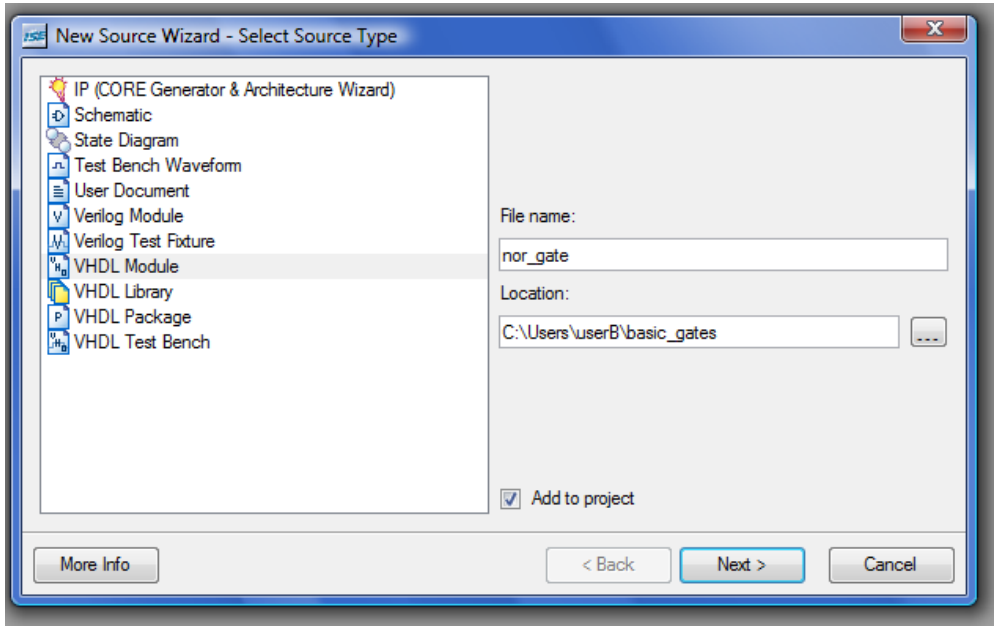


Πύλη NOR

- Στο παράθυρο Processes κάνουμε διπλό κλικ στο Create New Source

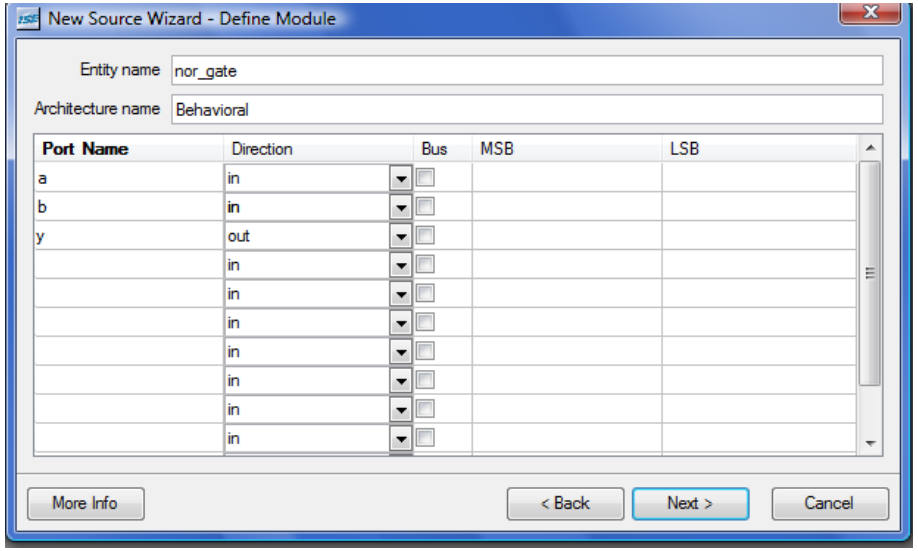


- Και επιλέγουμε VHDL Module , δίνουμε το όνομα nor_gate, προσέχουμε η επιλογή Add to Project να είναι πατημένη, όπως φαίνεται παρακάτω.



Πύλη NOR

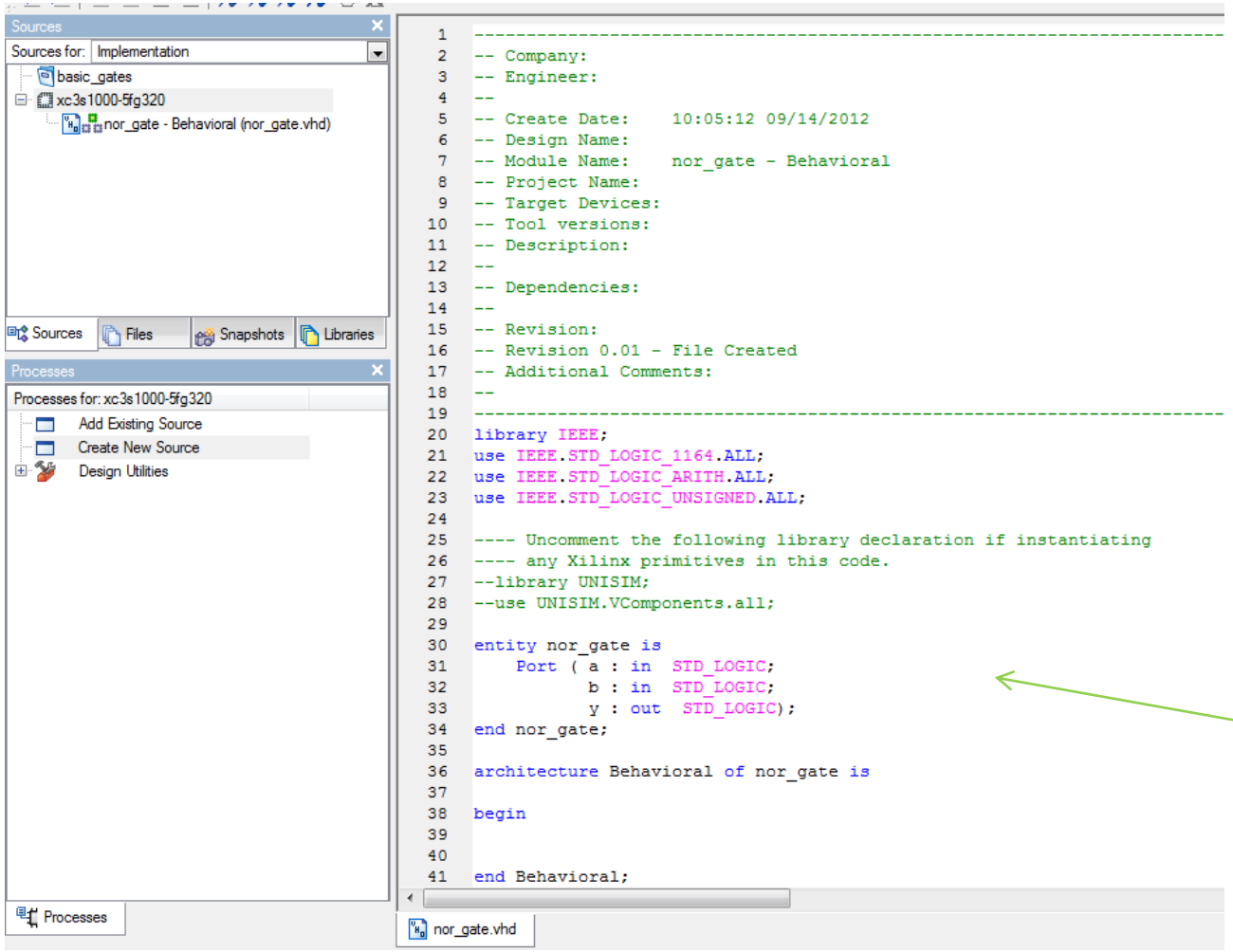
- Στη συνέχεια θα ορίσουμε στο παράθυρο που εμφανίζεται τις εισόδους και τις εξόδους του entity. Για το παράδειγμά μας θα ορίσουμε τις εισόδους (in) a,b και μία έξοδο y (out) όπως φαίνεται στο παρακάτω σχήμα.



- Και πατάμε Next, Finish .

- Οπότε τώρα ανοίγει ένας editor όπου βλέπουμε σε vhdl κώδικα το entity που μόλις ορίσαμε

Πύλη NOR




```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    10:05:12 09/14/2012
6  -- Design Name:
7  -- Module Name:    nor_gate - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_ARITH.ALL;
23 use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25 ---- Uncomment the following library declaration if instantiating
26 ---- any Xilinx primitives in this code.
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity nor_gate is
31     Port ( a : in  STD_LOGIC;
32            b : in  STD_LOGIC;
33            y : out  STD_LOGIC);
34 end nor_gate;
35
36 architecture Behavioral of nor_gate is
37
38 begin
39
40
41 end Behavioral;
```

Προσοχή στη χρήση των ερωτηματικών στον ορισμό του entity

Πύλη NOR

- **entity** nor_gate is

  Port (a : in STD_LOGIC; -- κάθε σήμα τελειώνει με ένα ερωτηματικό

  b : in STD_LOGIC;

  y : out STD_LOGIC); --εκτός από το τελευταίο σήμα !

  end nor_gate;

- Παρατηρούμε ότι στον τύπο των σημάτων που ορίσαμε ως εισόδους/εξόδους το εργαλείο της Xillinx έδωσε το STD_LOGIC . (Αυτό θα μπορούσαμε το αλλάξουμε και θα το κάνουμε **bit**, αλλά για το παράδειγμα μας δεν θα το πειράξουμε) .

Πύλη NOR

- Στη συνέχεια θα ορίσουμε πως λειτουργεί η πύλη μας. Αυτό γίνεται στο κομμάτι του κώδικα που αρχίσει με τη λέξη architecture μέσα στο begin/end κομμάτι.
- Η λέξη «Behavioral» είναι μία λέξη που από default δίνει το εργαλείο της Xilinx και μπορούμε να την αλλάξουμε με ότι άλλο θέλουμε (προσοχή υπάρχει στην αρχή αυτού του τμήματος κώδικα αλλά και στο τέλος)
- **architecture** Behavioral of nor_gate is

  begin

  y<= a nor b;

  end Behavioral;

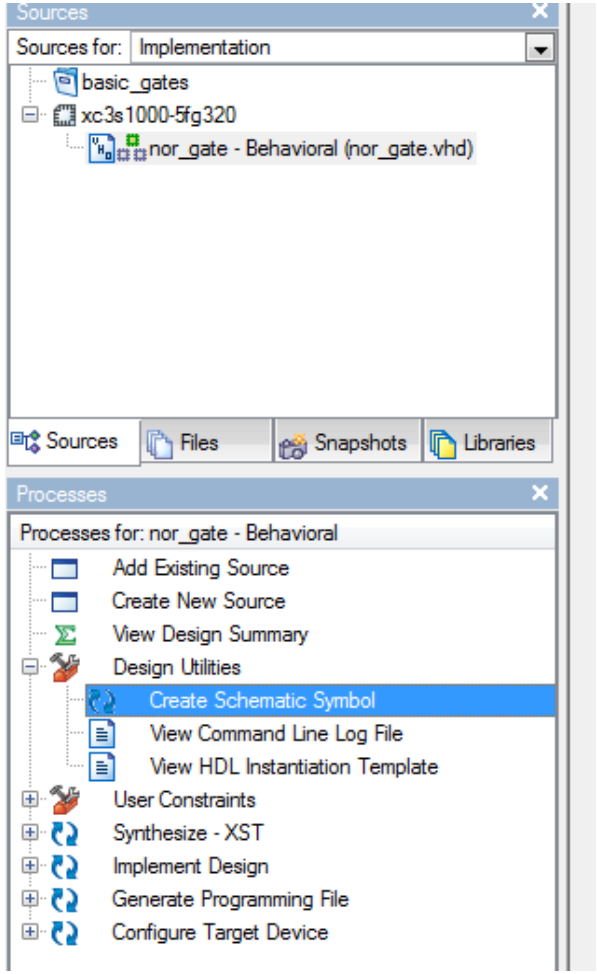
Πύλη NOR

- Προσέξτε τη χρήση του συμβόλου <= , δεν πρόκειται για σύμβολο μικρότερο ίσο, όπως το ξέρουμε από τις άλλες γλώσσες προγραμματισμού , αλλά είναι ένα σύμβολο που δείχνει ότι τα δεδομένα μεταφέρονται (data flow) από το σήμα που βρίσκεται δεξιά από το σύμβολο στο σήμα που βρίσκεται αριστερά του.

- Το *nor* είναι ένα built-in component που ονομάζεται *operator*, γιατί λειτουργεί (operates) πάνω σε κάποια δεδομένα και παράγει νέα δεδομένα. Δηλαδή θα μπορούσαμε να πούμε ότι το σήμα y παράγεται από τα δεδομένα a,b τα οποία έχουν πρώτα επεξεργαστεί από τον *nor operator.* Άλλοι operators είναι: **and, or, nand, xor, xnor** και **not**.

- Ο τρόπος αυτός ορισμού και περιγραφής των entities ονομάζεται **DataFlow** Design

Πύλη NOR



Κάνουμε Save το αρχείο vhd που δημιουργήσαμε και στη συνέχεια θα επιλέξουμε το nor_gate από το παράθυρο Sources και θα πάμε στα Processes για να δούμε όλες τις διαθέσιμες επιλογές για αυτό το entity που μόλις φτιάξαμε .

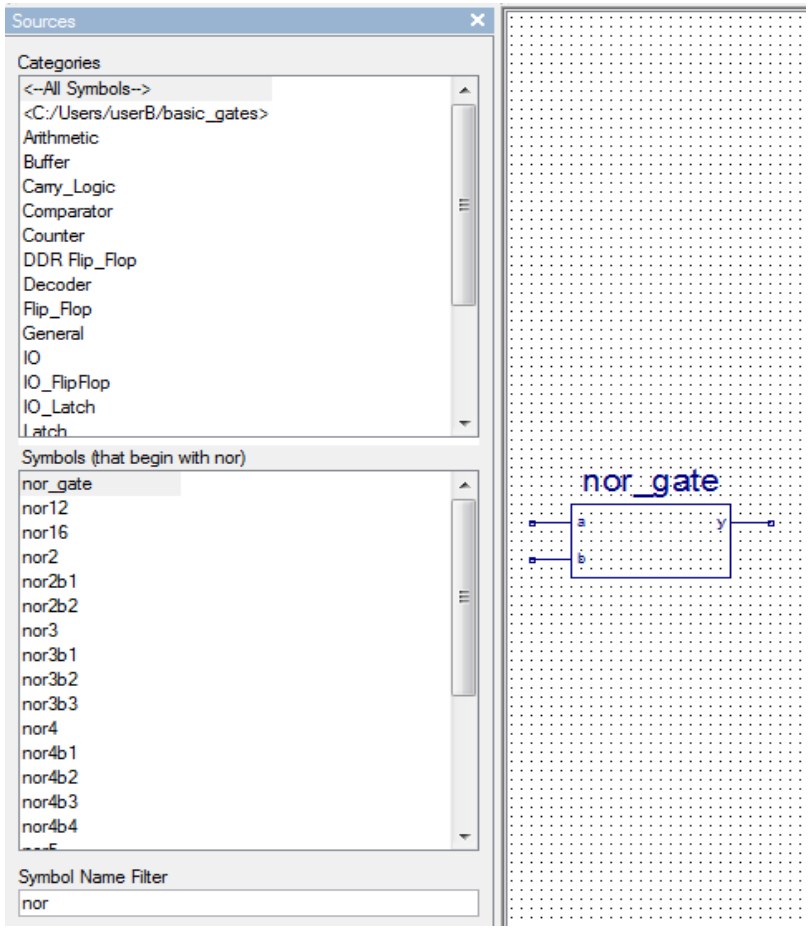
Συγκεκριμένα θα πατήσουμε την επιλογή Design Utilities , και με διπλό κλικ στο Create Schematic Symbol το συγκεκριμένο entity θα γίνει ένα component το οποί θα μπορεί να χρησιμοποιηθεί μέσα σε άλλα σχέδια.

Αφού τρέξει το συγκεκριμένο Process πρέπει να δούμε το μήνυμα *"Create Schematic Symbol" completed successfully* που δείχνει την ορθότητα του entity.

Πύλη NOR

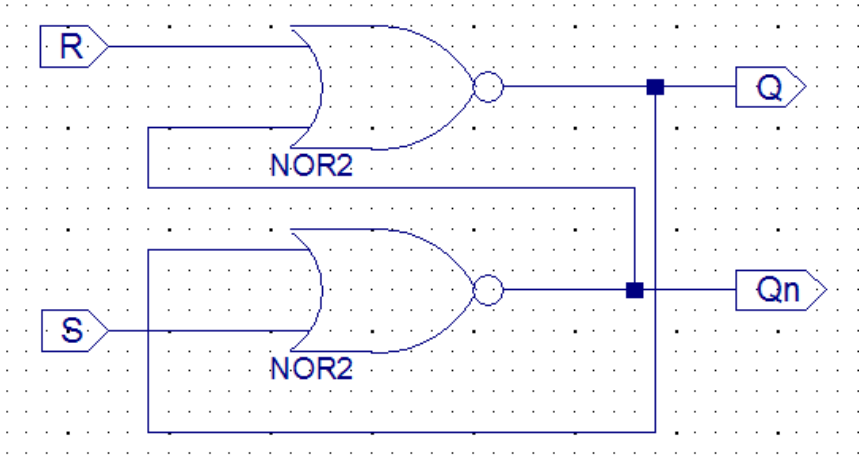
Η πύλη nor που σχεδιάσαμε σε VHDL κώδικα είναι τώρα διαθέσιμη σαν component στη λίστα των συμβόλων της Xillinx, και μπορούμε να τη χρησιμοποιήσουμε για να σχεδιάσουμε ένα νέο ψηφιακό κύκλωμα σε σχηματική μορφή .

Δηλαδή, αν πατήσουμε Create New source, επιλέξουμε το Scematic από το παράθυρο που εμφανίζεται, ας δώσουμε και ένα όνομα , πχ test_nor και μπορούμε στη αναζήτη συμβόλων να βρούμε και το nor_gate όπως δείχνουμε στο σχήμα.



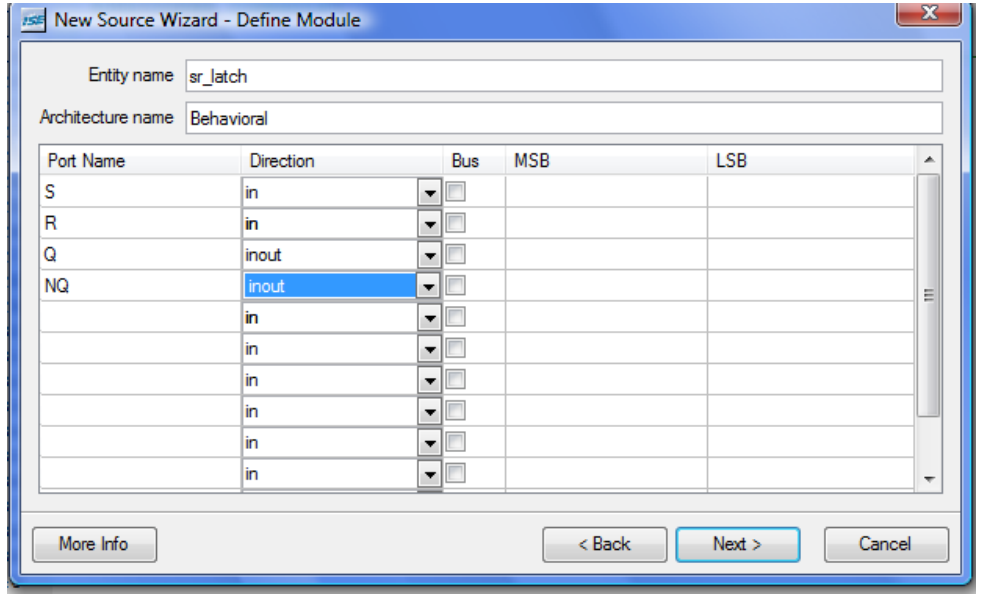
Το SR-Latch

- Εμείς δεν θα συνεχίσουμε με τη σχηματική σχεδίαση, αλλά θα χρησιμοποιήσουμε την πύλη nor σαν component σε vhdl για να φτιάξουμε νέα ψηφιακά κυκλώματα.
- Το SR-Latch που θέλουμε να φτιάξουμε φαίνεται στο παρακάτω σχήμα. Θα φτιάξουμε το latch στη VHDL με δύο διαφορετικούς τρόπους (Dataflow & Structural).



To SR-Latch - Dataflow Design

- Ξεκινάμε με τον Project Navigator, όπου επιλέγουμε Create new source, Vhdl Module, δίνουμε το όνομα sr_latch , και δίνουμε τα παρακάτω στις εισόδους-εξόδους . Προσοχή τα σήματα Q,NQ ορίζονται σαν **inout ,** που σημαίνει ότι το σήμα μπορεί να λειτουργεί σαν είσοδος αλλά και σαν έξοδος



- Πατάμε Next και μετά Finish και βλέπουμε το νεό entity που φτιάξαμε.

To SR-Latch - Dataflow Design

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity sr_latch is
31      Port ( S : in  STD_LOGIC;
32             R : in  STD_LOGIC;
33             Q : inout  STD_LOGIC;
34             NQ : inout  STD_LOGIC);
35  end sr_latch;
36
37  architecture Behavioral of sr_latch is
38
39  begin
```

To SR-Latch - Dataflow Design

- Στη συνέχεια θα γράψουμε τον κώδικα για την λειτουργία του latch, όπως φαίνεται παρακάτω

```
29
30  entity sr_latch is
31      Port ( S : in  STD_LOGIC;
32             R : in  STD_LOGIC;
33             Q : inout  STD_LOGIC;
34             NQ : inout  STD_LOGIC);
35  end sr_latch;
36
37  architecture Behavioral of sr_latch is
38
39  begin
40       Q <= R nor NQ;
41       NQ <= S nor Q;
42
43  end Behavioral;
44
```

- Τώρα μπορούμε να κάνουμε Create Schematic Symbol με τον τρόπο που ξέρουμε και στη συνέχεια μπορούμε να προχωρήσουμε στη διαδικασία του Simulation για να ελέγξουμε την ορθότητα του σχεδίου μας.

Το SR-Latch - Simulation

- Γίνεται για να τροφοδοτηθούν οι είσοδοι και να παραχθούν οι έξοδοι ώστε να επαληθεύσουμε τη λειτουργία του κυκλώματος.
- Χρειαζόμαστε ένα VHDL test bench όπως έχουμε κάνει και σε προηγούμενα εργαστήρια. Οπότε ξεκινάμε με Project-New source -> VHDL Test Bench δίνουμε ένα όνομα πχ *sr_latch_tb* (όπως έχουμε δει στα προηγούμενα εργαστήρια) .
- Στη συνέχεια συσχετίζουμε το πηγαίο αρχείο **sr_latch** με το το test bench που θα φτιάξουμε, πατάμε Next και finish.
- Τώρα εμφανίζεται ο ISE TEXT editor με ένα έτοιμο VHDL template για το αρχείο test bench που θέλουμε να δημιουργήσουμε.

Το SR-Latch – test bench

Για να περιγράψουμε λίγο το αρχείο αυτό θα το χωρίσουμε σε δύο τμήματα.

Στο πρώτο μέρος βλέπουμε το entity και το γενικό πλαίσιο της αρχιτεκτονικής.

```
27  ----------------------------------------------------------------
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.ALL;
30  USE ieee.std_logic_unsigned.all;
31  USE ieee.numeric_std.ALL;
32
33  ENTITY sr_latch_b_tb IS
34  END sr_latch_b_tb;
35
36  ARCHITECTURE behavior OF sr_latch_b_tb IS
37
38      -- Component Declaration for the Unit Under Test (UUT)
39
40      COMPONENT sr_latch_b
41      PORT(
42           S : IN  std_logic;
43           R : IN  std_logic;
44           Q : INOUT  std_logic;
45           NQ : INOUT  std_logic
46          );
47      END COMPONENT;
48
49
50     --Inputs
51     signal S : std_logic := '0';
52     signal R : std_logic := '0';
53
54     --BiDirs
55     signal Q : std_logic;
56     signal NQ : std_logic;
57
```

Το κύκλωμα ως component

Το σήματα που θα χρησιμοποιήσουμε ορισμένα σαν internal signals

Το SR-Latch – test bench

- Στο δεύτερο μέρος είναι το σώμα της αρχιτεκτονικής (BEGIN/END)

```
57
58  BEGIN
59
60     -- Instantiate the Unit Under Test (UUT)
61     uut: sr_latch_b PORT MAP (
62            S => S,
63            R => R,
64            Q => Q,
65            NQ => NQ
66          );
67
68     -- No clocks detected in port list. Replace <clock> below with
69     -- appropriate port name
70
71     constant <clock>_period := 1ns;
72
73     <clock>_process :process
74     begin
75        <clock> <= '0';
76        wait for <clock>_period/2;
77        <clock> <= '1';
78        wait for <clock>_period/2;
79     end process;
80
81
82     -- Stimulus process
83     stim_proc: process
84     begin
85        -- hold reset state for 100ms.
86        wait for 100ms;
87
88        wait for <clock>_period*10;
89
90        -- insert stimulus here
91
92        wait;
93     end process;
94
95  END;
96
```

1-1 αντιστοιχία σημάτων με το component

Αυτό το κομμάτι δε το χρειαζόμαστε γιατί δεν έχουμε ρολόι στη σχεδίαση μας

Ένα προκαταρτικό test bench process που περιέχει και clk σήμα το οποίο θα τροποποιήσουμε γιατί δεν έχουμε ρολόι στη σχεδίαση μας

Το SR-Latch – test bench

- Θα συμπληρώσουμε το αρχεία αυτό ανάλογα με τις απαιτήσεις της σχεδίασης μας. Συγκεκριμένα ο πίνακας αλήθειας του SR latch είναι:

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	Μετά από S=1 ,R=0
0	1	0	1	
0	0	0	1	Μετά από S=0 ,R=1
1	1	0	0	!!!

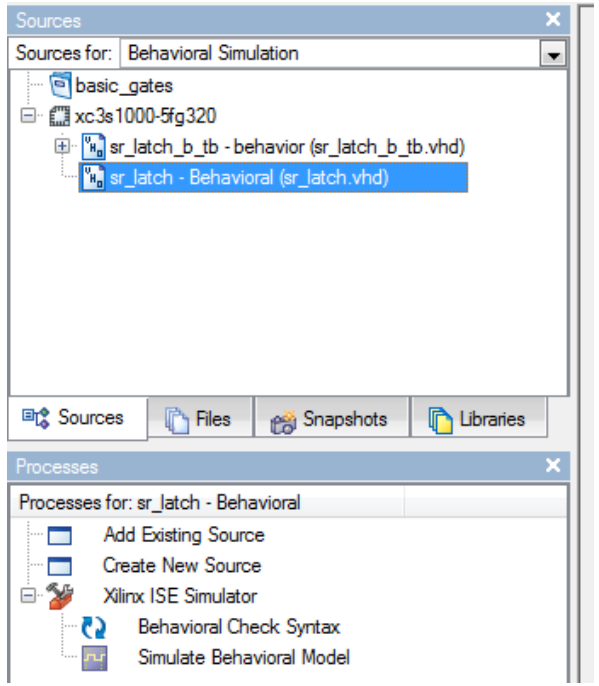
- Με βάση αυτόν το πίνακα θα δώσουμε τιμές στα σήματα S,R στο test bench αρχείο το οποίο θα γίνει τώρα ως εξής (φυσικά ο καθένα μπορεί να το τροποποιήσει αναλόγως)

To SR-Latch – test bench

```
67
68      -- Stimulus process
69    stim_proc: process
70    begin
71      S<='0' ; R<='0';
72       wait for 100ns;
73
74       S<='1' ;
75       wait for 50ns;
76
77       S<='0' ;
78       wait for 50ns;
79
80       R<='1';
81       wait for 50ns;
82
83       R<='0';
84       wait for 50ns;
85
86       S<='1' ;
87       wait for 50ns;
88
89       S<='0' ;
90       wait for 50ns;
91
92       R<='1';
93       wait for 50ns;
94
95       R<='0';
96       wait for 50ns;
97
98       S<='1' ; R<='1';
99       wait for 100ns;
100
101
102   end process;
```

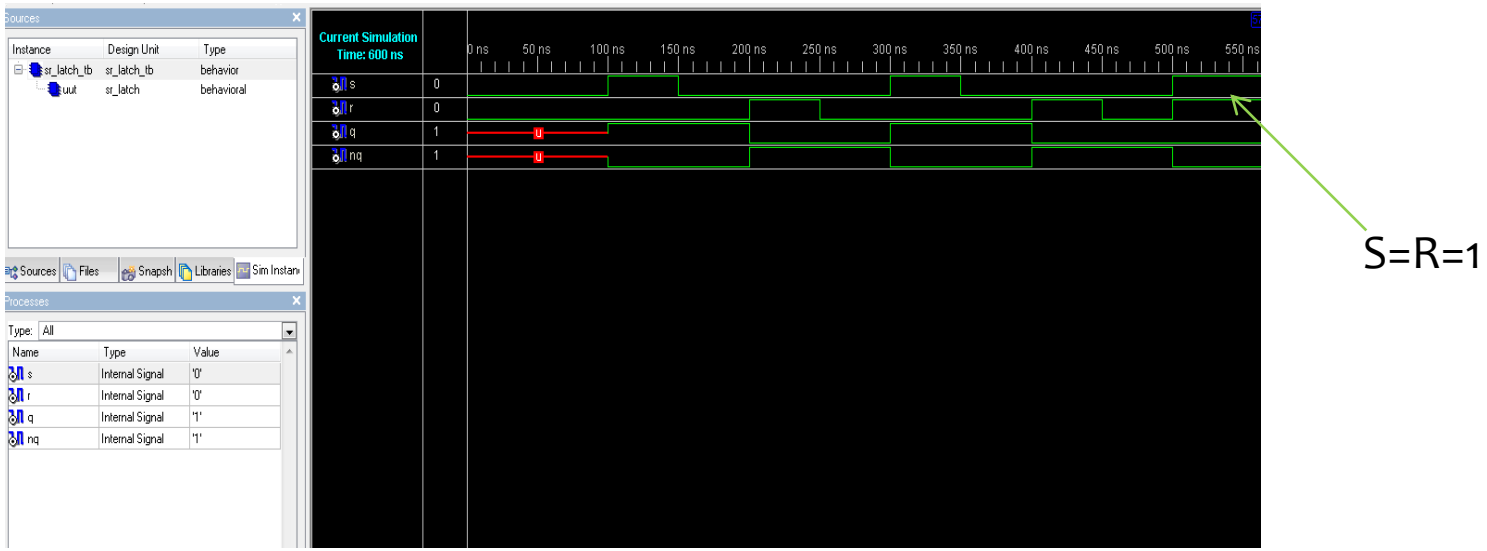
Το SR-Latch – test bench

- Αφού σώσουμε το παραπάνω επιλέγουμε το tb αρχείο που φτιάξαμε στο παράθυρο Sources και επιπλέον επιλέγουμε Behavoural Simulation όπως φαίνεται στη συνέχεια. Πηγαίνουμε στο παράθυρο με τα Processes και ξεκινάμε το Simulate Behavoural Model (βλ. screenshot που ακολουθεί)



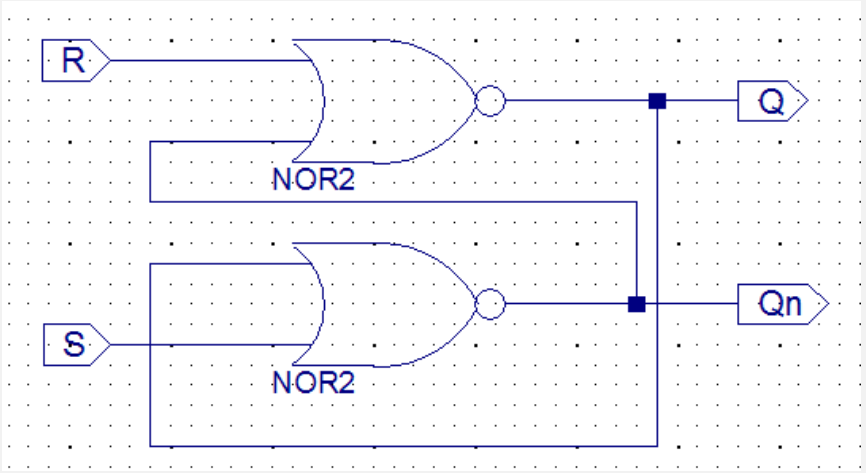
SR-latch Behavoural Simulation

- Το ενδιαφέρον με το SR latch είναι να παρατηρήσουμε τι γίνεται όταν και οι δύο R,S γίνουν 1 ταυτόχρονα. Τότε το Q και το Q' δεν είναι η μία το συμπλήρωμα της άλλης όπως φαίνεται στο παρακάτω screenshot .



Β τρόπος – Structural Design

- Θα χρησιμοποιήσουμε την πύλη nor που φτιάξαμε στο πρώτο παραδείγμα , για να φτιάξουμε στη VHDL το SR-latch.
- Αυτός ο τρόπος σχεδίασης ονομάζεται **Structural** Design και στην ουσία πρόκειται για vhdl κώδικα όπου περιγράφουμε το σχηματικό μας.



SR latch - Structural

- Οπότε πάμε πάλι στον Project Navigator->Create New Source και θα το ονομάσουμε sr_latch_b, θα ορίσουμε τις εισόδους/εξόδους όπως κάναμε πριν (προσοχή στο inout) και θα πάρουμε το παρακάτω , που φυσικά μέχρι στιγμής δεν διαφέρει και πολύ από το προηγούμενο !!

```
19  ----------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity sr_latch_b is
31      Port ( S : in  STD_LOGIC;
32             R : in  STD_LOGIC;
33             Q : inout  STD_LOGIC;
34             NQ : inout  STD_LOGIC);
35  end sr_latch_b;
36
37  architecture Behavioral of sr_latch_b is
38
39  begin
40
```

SR latch - Structural

Τώρα όμως θα περιγράψουμε τη συνδεσμολογία του σχηματικού SR-Latch χρησιμοποιώντας την πύλη nor σαν component στη σχεδίαση με τον διπλανό τρόπο (διακρίνεται τις διαφορές μεταξύ του entity-component-component instance)

```
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity sr_latch_b is
31      Port ( S : in  STD_LOGIC;
32             R : in  STD_LOGIC;
33             Q : inout  STD_LOGIC;
34             NQ : inout  STD_LOGIC);
35  end sr_latch_b;
36
37  architecture Behavioral of sr_latch_b is
38
39     component nor_gate
40     port(a,b: in STD_LOGIC;
41          y:out STD_LOGIC);
42     end component;
43
44  begin
45
46     n1: nor_gate port map(R,NQ,Q);
47     n2: nor_gate port map(S,Q,NQ);
48
49  end Behavioral;
50
```

entity

Η πύλη nor_gate χρησιμοποιείται σαν component του σχεδίου.

Component instance

Το port map καθορίζει ποια σήματα της σχεδίασης θα συνδεθούν στο component. Προσοχή πρέπει τα σήματα να ακολουθούν τη σειρά με την οποία είναι ορισμένα στο component declaration.

SR latch - Structural

- Αφού τελειώσαμε με το κομμάτι της περιγραφής, συνεχίζουμε με τα βήματα που κάναμε και στα προηγούμενα παραδείγματα (create schematic symbol… κτλ) μέχρι να φτάσουμε στο Simulation, όπου μπορούμε και εδώ να ετοιμάσουμε ένα test bench αρχείο και να ελέγξουμε την ορθότητα της σχεδίασης. Τα βήματα αυτά τα αφήνουμε για την ώρα σαν άσκηση , για να συνεχίσουμε με άλλα παραδείγματα σε VHDL.
- **ΑΣΚΗΣΗ για το σπίτι:** Υλοποίηση του full adder που φτιάξαμε σε σχηματικό , με χρήση VHDL .

D flip-flop

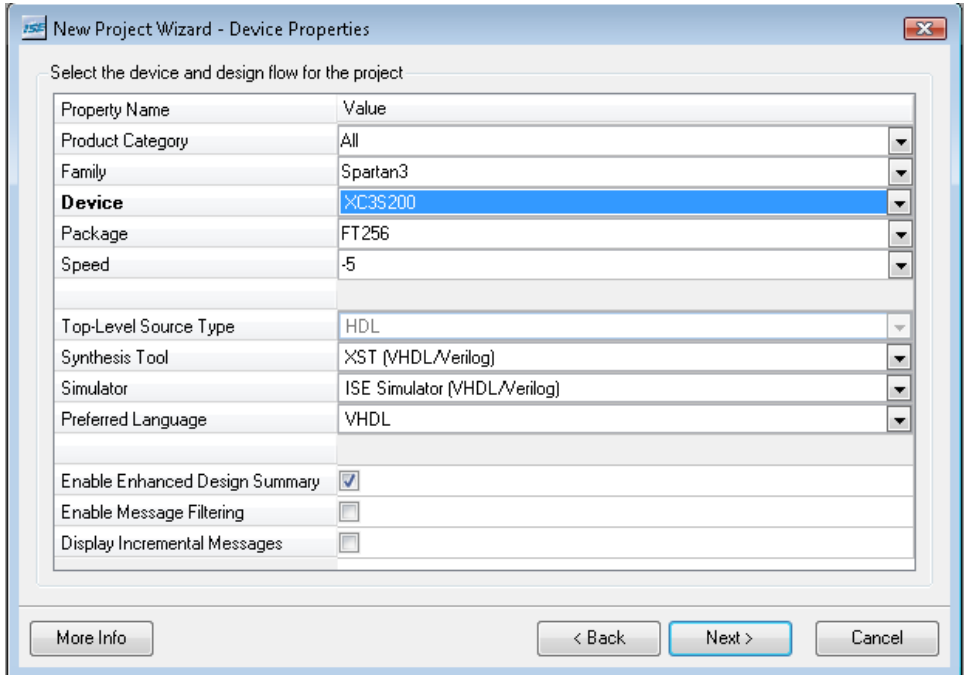
- Το D flip-flop έχει 2 εισόδους τις D και CP . Οι δυαδικές πληροφορίες που βρίσκονται στην είσοδο δεδομένων του D flip-flop μεταφέρονται στην έξοδο Q όταν η είσοδος CP ενεργοποιηθεί.
- Η έξοδος ακολουθεί τα δεδομένα της εισόδου όσο ο παλμός CP παραμένει στην κατάσταση 1. Όταν ο παλμός πάει στο 0 οι πληροφορίες που βρίσκονταν στην είσοδο δεδομένων την ώρα που συνέβη η μετάβαση του παλμού, παραμένουν στην έξοδο Q μέχρι να ενεργοποιηθεί ξανά η είσοδος CP.
- Ο χαρακτηριστικός πίνακας του D flip-flop είναι:

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

- Και η χαρακτηριστική εξίσωση Q(t+1)=D .

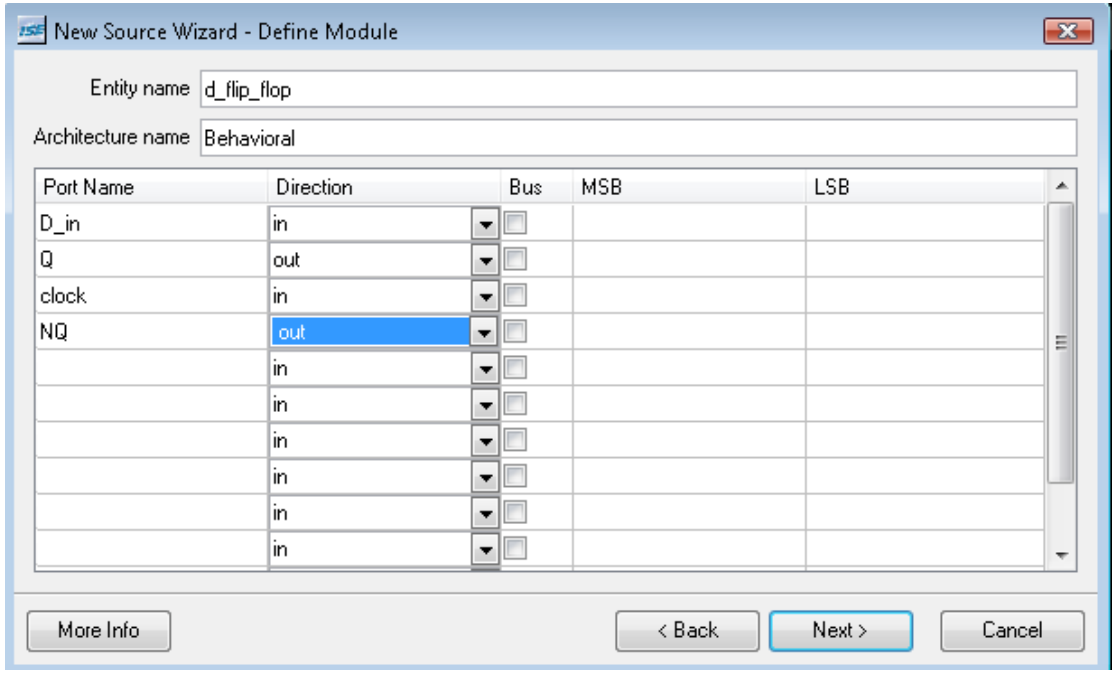
D flip-flop

- Ξεκινάμε με τον Project Navigator -> New Project -> όνομα DFF



D flip-flop

- Μετά Create New source επιλέγουμε VHDL Module και δίνουμε ένα όνομα πχ d_flip_flop. Δίνουμε ονόματα στις εισόδους και στις εξόδους όπως φαίνεται στο παρακάτω σχήμα



D flip-flop

- Εδώ χρησιμοποιούμε ένα διαφορετικό τρόπο περιγραφής του ψηφιακού μας κυκλώματος από ότι έχουμε δει μέχρι τώρα στα προηγούμενα παραδείγματα. Συγκεκριμένα ορίζουμε ένα process statement μέσα στο οποίο περιγράφεται η λειτουργία του flipflop.

- Το flipflop είναι το καλύτερο παράδειγμα για να καταλάβουμε την έννοια του process στη VHDL και αυτό γιατί παραμένει ανενεργό, δεν αλλάζει κατάσταση, μέχρι ένα γεγονός (πχ άνοδος παλμού ή ασύχρονο reset) να του αλλάξει την κατάσταση.

- Στην περίπτωσή μας το process περιμένει το σήμα clock να γίνει 1 (WAIT UNTIL) και μετά εκτελεί τις επόμενες εντολές του.

D flip-flop

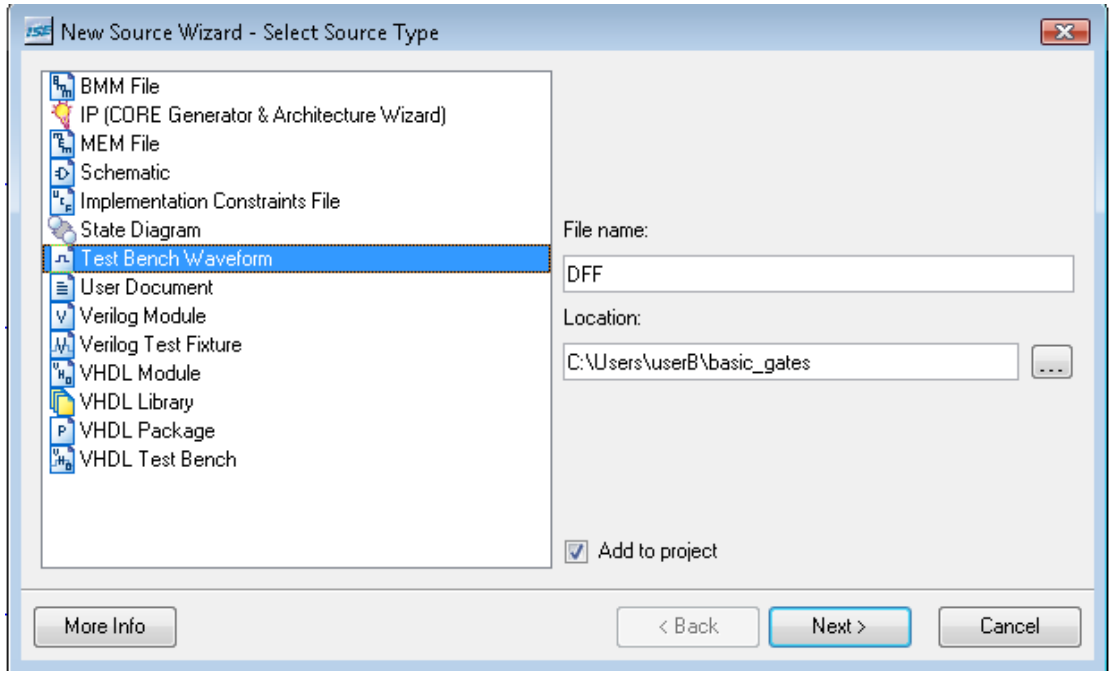
- Πηγαίνουμε στο vhd αρχείο που δημιουργήθηκε και προσθέτουμε τις γραμμές κώδικα που δείχνουμε στη συνέχεια.

```
29
30  entity d_flip_flop is
31      Port ( D_in : in  STD_LOGIC;
32             Q : out  STD_LOGIC;
33             clock : in  STD_LOGIC;
34             NQ : out  STD_LOGIC);
35  end d_flip_flop;
36
37  architecture Behavioral of d_flip_flop is
38
39  begin
40  flipflop: process
41  begin
42     wait until clock='1';
43     Q<=D_in;
44     NQ<=not D_in;
45  end process;
46
47
48  end Behavioral;
49
```

1. Το όνομα *flipflop* για το process είναι προαιρετικό
2. Δεν έχουμε sensitivity list αλλά wait statement
3. Το process περιμένει το σήμα clock να γίνει 1

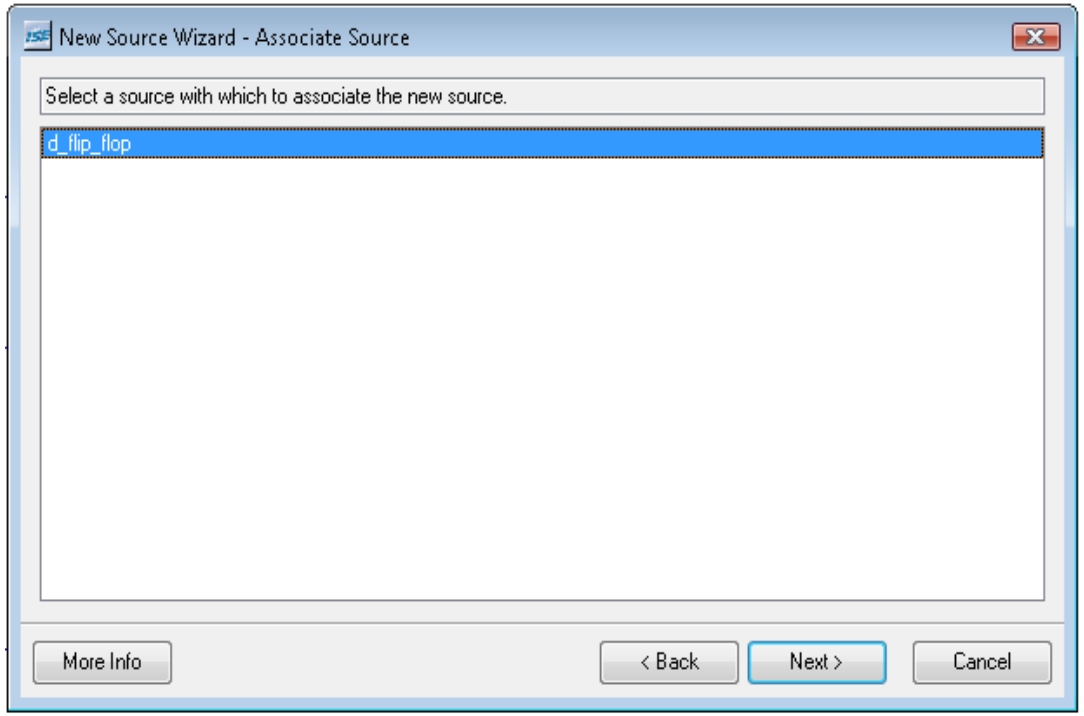
D flip-flop - simulation

- Για να επαληθεύσουμε τη λειτουργία του flipflop θα δούμε τώρα ένα άλλον τρόπο που μας παρέχει το εργαλείο της Xilinx για να βλέπουμε κυματομορφές. Κάνουμε Create new source και επιλέγουμε Test Bench Waveform , δίνουμε ένα όνομα



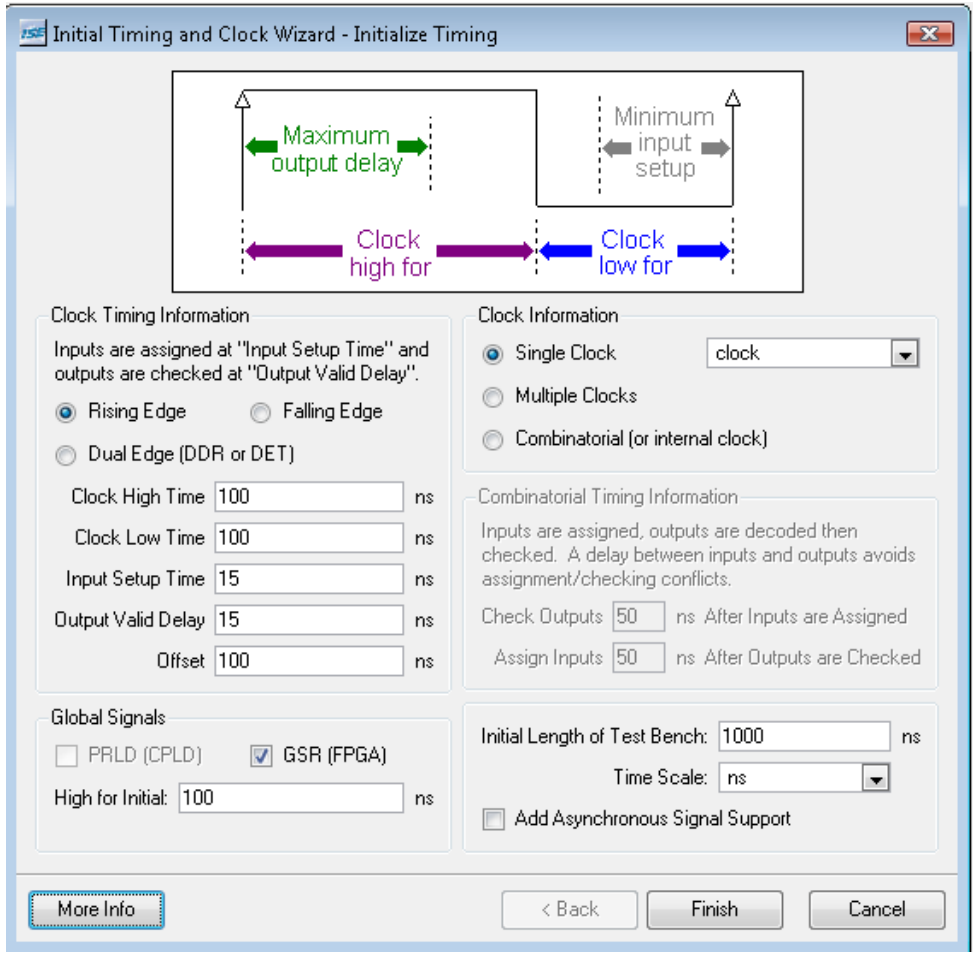
D flip-flop - simulation

- Πατάμε Next και συσχετίζουμε το αρχείο d_flip_flop



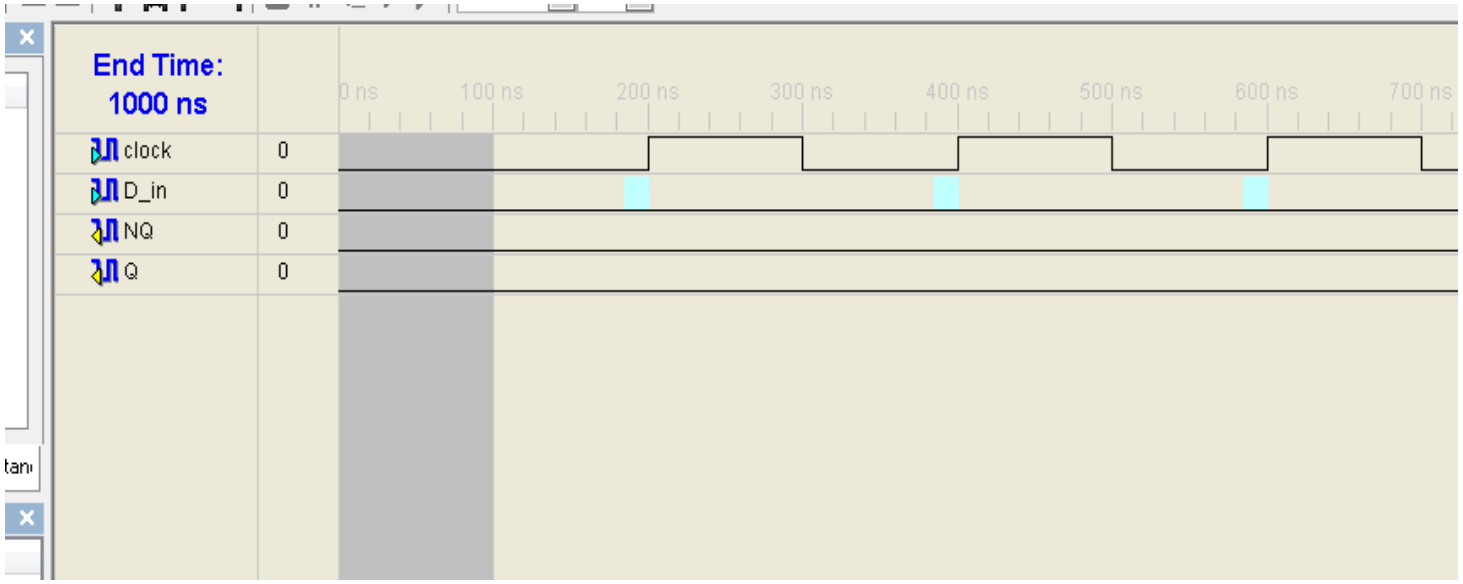
D flip-flop - simulation

- Στη συνέχεια εμφανίζεται το παρακάτω παράθυρο, στο οποίο καθορίζουμε λεπτομέρειες σχετικά με τους παλμούς ρολογιού που χρησιμοποιεί το ψηφιακό μας κύκλωμα.



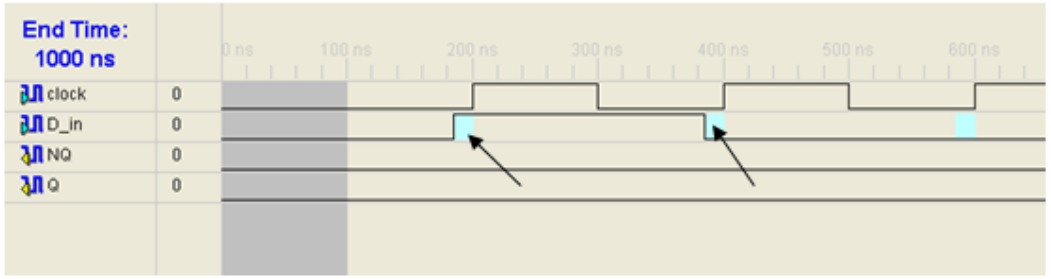
D flip-flop - simulation

- Αφήνουμε τις τιμές αυτές όπως είναι για την ώρα , πατάμε Finish και εμφανίζεται η παρακάτω οθόνη



D flip-flop - simulation

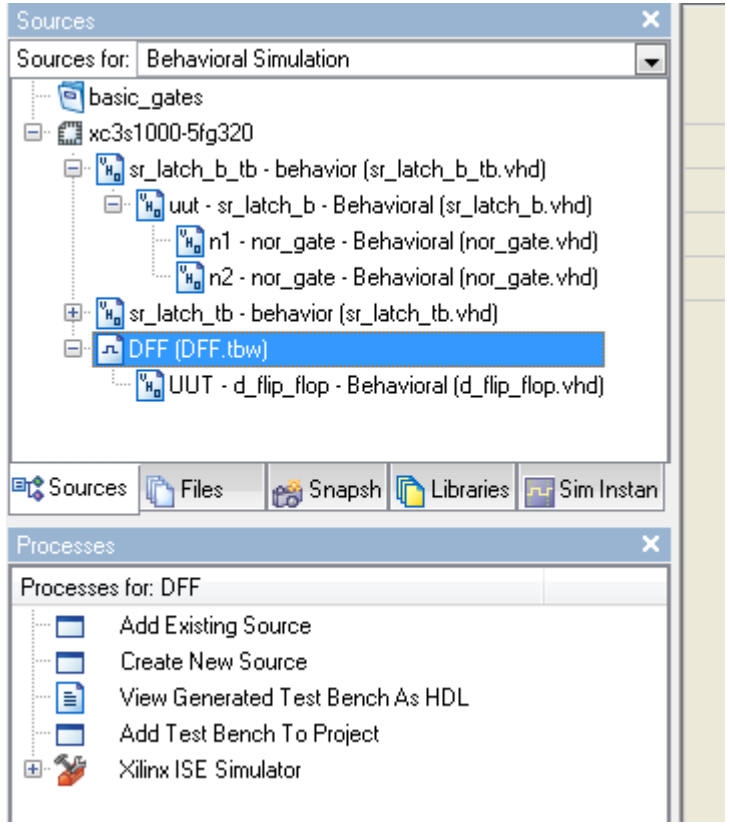
- Κάνοντας κλικ με το ποντίκι μας στη γραμμή που αφορά το D_in στα σημεία που φαίνονται με γαλάζιο χρώμα , αλλάζει η τιμή του παλμού από 0 σε 1 (η αν ηταν 1 γίνεται 0). Πηγαίνουμε στα σημεία αυτά και δίνουμε τιμές στην είσοδο D_in .



D flip-flop – simulation

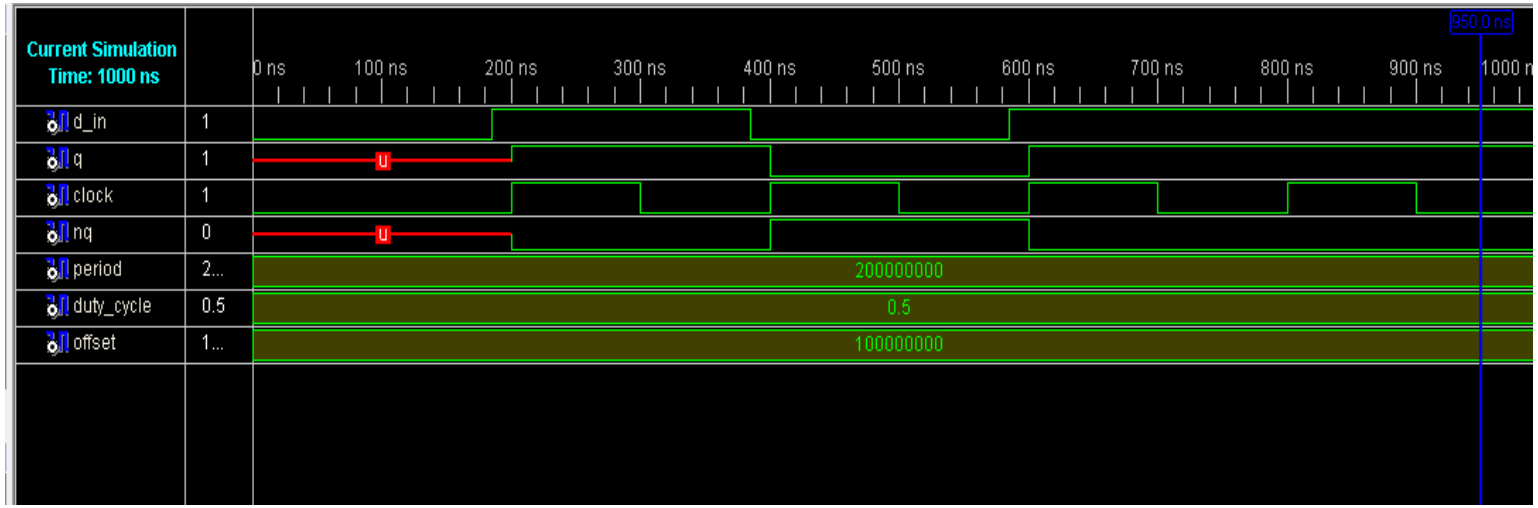
- Αφού φτιάξουμε την κυματομορφή για την είσοδο D_in, κάνουμε Save αρχείο *tbw που έχει δημιουργηθεί. Στο παράθυρο με τα Sources αριστερά πατάμε Behavoural simulation και επιλέγουμε το αρχείο DFF.twb για να δούμε τα διαθέσιμα Processes.

Εδώ θα πάμε στο Xillinx ISE Simulator και θα τρέξουμε Simulate Behavoural Model , όπως ακριβώς και όταν είχαμε το test bench αρχείο.



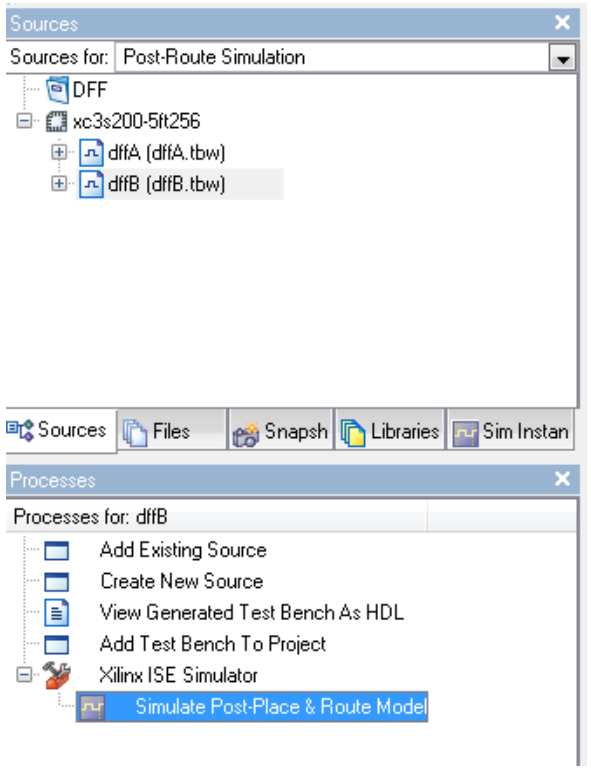
D flip-flop - simulation

- Και τα αποτελέσματα είναι



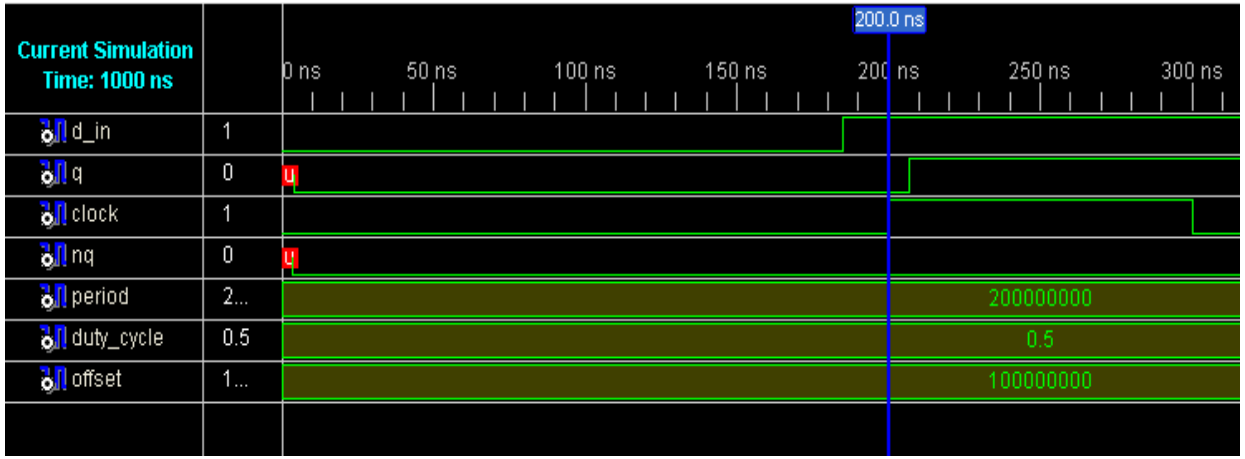
D flip-flop - simulation

- Μπορούμε να κάνουμε την ίδια διαδικασία δημιουργώντας ένα νέο Test Bench Waveform για να κάνουμε και το Post route Simulation. Κάνουμε Create New Source, επιλέγουμε Test Bench Waveform, δίνουμε ένα διαφορετικό όνομα πχ dffB και συνεχίζουμε την διαδικασία όπως κάναμε πριν. Τώρα θα επιλέξουμε Post route simulation



D flip-flop - simulation

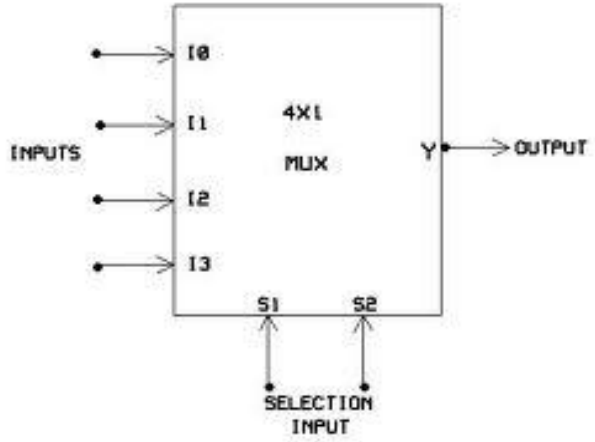
- Αφού ολοκληρωθεί με επιτυχία το simulation θα πάρουμε τις παρακάτω κυματομορφές



- Αυτό το οποίο μπορούμε να παρατηρήσουμε εδώ (έχουμε προσθέσει ένα time mark με δεξί κλικ πάνω στις κυματομορφές στα 200ns) είναι ότι υπάρχει μία μικρή καθυστέρηση στην αλλαγή της εξόδου q μετά την άνοδο του παλμού clock (δηλαδή εδώ βλέπουμε το δ delay time το οποίο περιγράψαμε όταν μιλήσαμε για τα processes).

Multiplexer

- Ένας πολυπλέκτης είναι ένα συνδυαστικό κύκλωμα που επιλέγει δυαδικές πληροφορίες ανάμεσα σε πολλές γραμμές εισόδου και τις κατευθύνει σε μια μοναδική γραμμή εξόδου.
- Η επιλογή μιας συγκεκριμένης γραμμής εξόδου γίνεται μέσω των γραμμών επιλογής. Υπάρχουν γραμμές εισόδου και *n* γραμμές επιλογής που οι συνδυασμοί των bits τους καθορίζουν ποια είσοδος επιλέγεται.
- Στη συνέχεια θα δούμε ένα παράδειγμα σε VHDL που δείχνει πως ένα απλό συνδυαστικό κύκλωμα ενός 4-σε-1 πολυπλέκτη μπορεί να περιγραφεί χρησιμοποιώντας ένα process.



Multiplexer entity declaration

```
entity simple_mux is
        port (Sel: in bit_vector (0 to 1);
                A, B, C, D: in bit;
                Y: out bit);
end simple_mux;
```

Multiplexer dataflow design

```
architecture behavior of simple_mux is
begin
  process(Sel, A, B, C, D)
  begin
    if Sel = "00" then
      Y <= A;
    elsif Sel = "01" then
      Y <= B;
    elsif Sel = "10" then
      Y <= C;
    elsif Sel = "11" then
      Y <= D;
    end if;
  end process;
end simple_mux;
```

Η εντολή if στην VHDL ακολουθεί την ίδια λογική με ότι ξέρουμε από τις άλλες γλώσσες προγραμματισμού. Στη γενική της μορφή είναι

```
if condition then
        sequential statements
  [elsif condition then
        sequential statements ]
  [else
        sequential statements ]
end if;
```

Multiplexer simulation

- Η δημιουργία του αντίστοιχου vhd αρχείου στο Xillinx και η επαλήθευση της ορθής λειτουργίας τους (είτε με test bench αρχείο είτε με test bench waveform) αφήνετε σαν άσκηση.

- ΑΣΚΗΣΗ : Να γίνει ο decoder του διπλανού

σχήματος σε VHDL (decoder 5σε32 με χρήση 3σε8

και 2σε4 decoders)