ΥΠΟΥΡΓΕΙΟ ΕΘΝΙΚΗΣ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ ΕΠΕΑΕΚ

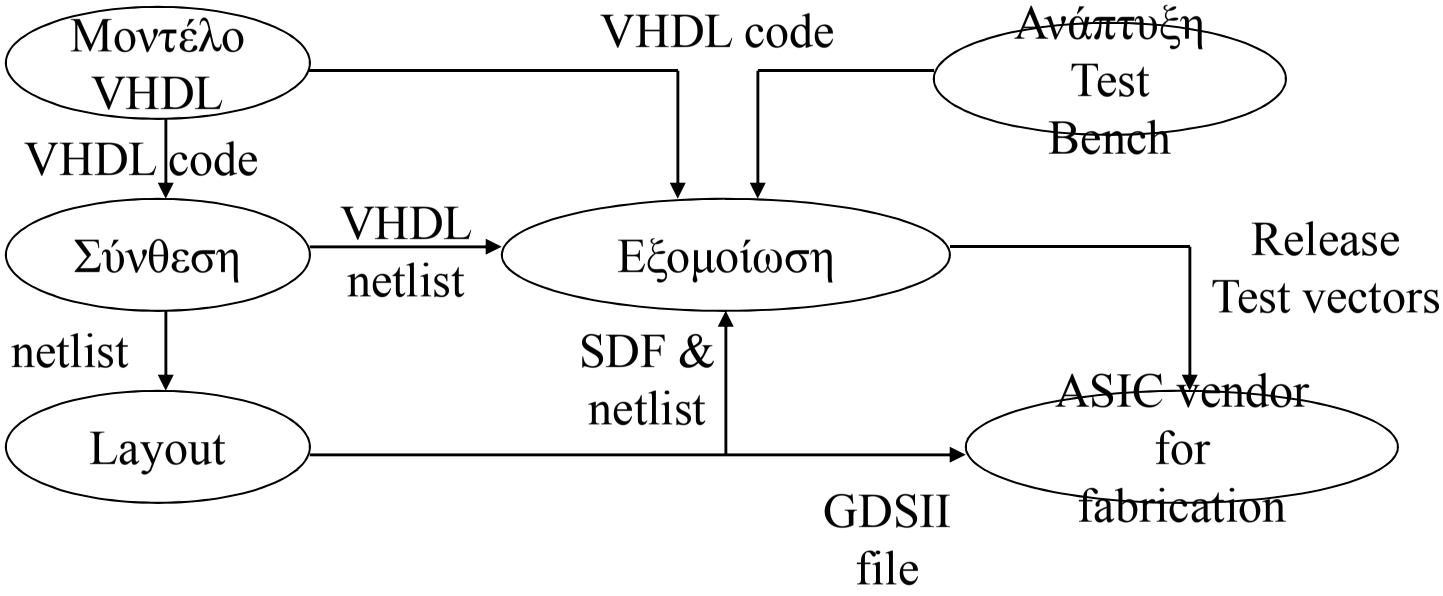
ΕΥΡΩΠΑΪΚΗ ΕΝΩΣΗ
ΣΥΓΧΡΗΜΑΤΟΔΟΤΗΣΗ
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Η ΠΑΙΔΕΙΑ ΣΤΗΝ ΚΟΡΥΦΗ
Επιχειρησιακό Πρόγραμμα
Εκπαίδευσης και Αρχικής
Επαγγελματικής Κατάρτισης

Σχεδιασμός Συστημάτων με VHDL

(Peter Ashenden, The Students Guide to VHDL)

Διαδικασία Σχεδιασμού

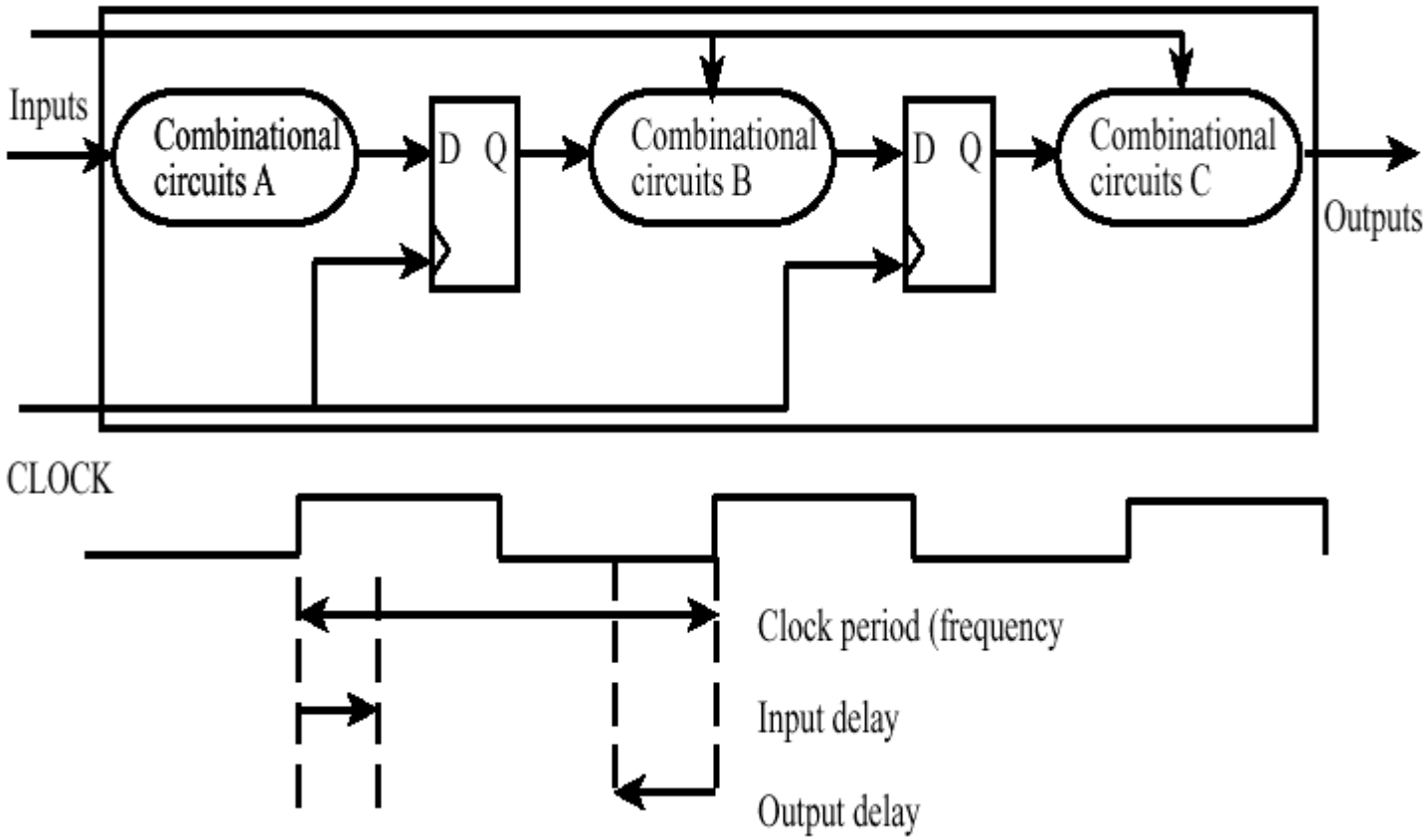


- Η συγγραφή προγραμμάτων για σύνθεση είναι διαφορετική από την συγγραφή προγραμμάτων για περιγραφή συστημάτων.
- Πολλές δομές της VHDL δεν είναι δυνατόν να συντεθούν.
- Διαφορετικοί τρόποι περιγραφής ίδιων λειτουργιών μπορεί να οδηγήσουν σε διαφορετικά κυκλώματα.

Διαδικασία Σχεδιασμού

- Η σύνθεση δημιουργεί ένα netlist βασισμένο σε cells μίας επιλεγμένης τεχνολογίας (library binding) το οποίο μπορεί να εξομοιωθεί με χρονική ακρίβεια.
- Μπορούν να εφαρμοστούν διαδικασίες βελτιστοποίησης οι οποίες οδηγούν είτε σε μικρότερα είτε σε γρηγορότερα κυκλώματα.
- Τα εργαλεία σύνθεσης παρέχουν κάθε χρονική πληροφορία (σε ns) και πληροφορία επιφάνειας (σε αριθμό πυλών) για το κύκλωμα που έχουν συνθέσει.
- Από τα αποτελέσματα της σύνθεσης καθορίζουμε την συχνότητα ρολογιού καθώς και χρόνους setup και hold για registers.

Αποτελέσματα Σύνθεσης



Η περίοδος ρολογιού δεν πρέπει να παραβιάζει τους απαιτούμενους χρόνους

D Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DFF is
  port(CLK, D : in  std_logic; Q: out std_logic);
end DFF;

architecture RTL of DFF is
begin
  seq0 : process
  begin
    wait until CLK'event and CLK = '1';
    Q <= D;
  end process;
end RTL;
```

D – Flip Flop

Το D-FF είναι το πιο κοινό ακολουθιακό στοιχείο στα ψηφιακά συστήματα.

Η αναμονή γεγονότος στο ρολόι σε συνδυασμό με την ανάθεση σήματος οδηγούν στην σύνθεση ff.

- Ο τύπος std_logic περιέχει τις τιμές (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’) οπότε η αρχική τιμή είναι η ‘U’.

D Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFFQN is
  port( CLK, D : in  std_logic;
    Q, Qn  : out std_logic);
end DFFQN;

architecture RTL of DFFQN is
  signal DFF : std_logic;
begin
  seq0 : process
  begin
    wait until CLK'event and CLK = '1';
    DFF <= D;
  end process;
  Q <= DFF; Qn <= not DFF;
end RTL;
```

D – Flip Flop με συμπληρωματικές εξόδους

Η αντιστροφή του Q μας υποχρεώνει να χρησιμοποιήσουμε ενδιάμεσο σήμα (το Q δεν μπορεί να διαβαστεί ώστε να αντιστραφεί).

D Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF_S is
  port( CLK, D, SR : in  std_logic;
    Q, Qn  : out std_logic);
end DFF_S;

architecture RTL of DFF_S is
  signal DFF : std_logic;
begin
  seq0 : process(CLK)
  begin
    if (CLK'event and CLK = '1') then
        if (SR='1') then DFF <='0';
        else DFF <= D; end if;
    end if;
  end process;
  Q <= DFF; Qn <= not DFF;
end RTL;
```

D – Flip Flop με σύγχρονο μηδενισμό

Ο έλεγχος του σήματος Μηδενισμού μετά τον έλεγχο γεγονότος στο ρολόϊ οδηγεί στον σύγχρονο μηδενισμό.

D Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF_AS is
  port( CLK, D, ASR : in  std_logic;
    Q, Qn  : out std_logic);
end DFF_AS;

architecture RTL of DFF_AS is
  signal DFF : std_logic;
begin
  seq0 : process(CLK,ASR)
  begin
    if (ASR='1') then DFF<='0';
    elsif (CLK'event and CLK = '1') then
        DFF <=D;
    end if;
   end process;
  Q <= DFF; Qn <= not DFF;
end RTL;
```

D – Flip Flop με
Ασύγχρονο μηδενισμό

Ο έλεγχος του σήματος Μηδενισμού πριν τον έλεγχο γεγονότος στο ρολόϊ οδηγεί στον ασύγχρονο μηδενισμό.

D Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF_AS is
  port( CLK, D, ASR, ASS: in  std_logic;
    Q, Qn  : out std_logic);
end DFF_AS;

architecture RTL of DFF_AS is
  signal DFF : std_logic;
begin
  seq0 : process(CLK,ASR,ASS )
  begin
    if (ASR='1') then DFF<='0';
    elsif (ASS='1') then DFF<='1';
    elsif (CLK'event and CLK = '1') then
        DFF <=D;
    end if;
   end process;
  Q <= DFF; Qn <= not DFF;
end RTL;
```

D – Flip Flop με Ασύγχρονη θέση / μηδένιση

Η μηδένιση έχει προτεραιότητα έναντι της θέσης.

JK Flip Flop

```
library IEEE;
use IEEE.std_logic_1164.all;

entity JKFF is
  port (
    CLK, RSTn, J, K : in  std_logic;
    Q, Qn           : out std_logic);
end JKFF;

architecture RTL of JKFF is
  signal FF : std_logic;
begin
  process (CLK, RSTn)
    variable JK: std_logic_vector(1 downto 0);
  begin
    if (RSTn = '0') then
      FF <= '0';
    elsif (CLK'event and CLK = '1') then
      JK := J & K;
      case JK is
        when "01"   => FF <= '0';
        when "10"   => FF <= '1';
        when "11"   => FF <= not FF;
        when others => FF <= FF;
      end case;
    end if;
  end process;
  Q <= FF after 2 ns; Qn <= not FF after 2 ns;
end RTL;
```

Latch

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Latch is
  port(LE, D : in  std_logic; Q: out std_logic);
end Latch;

architecture RTL of Latch is
begin
  seq0 : process (D, LE)
  begin
   if LE = '1‘ then
         Q <= D;
   end if;
  end process;
end RTL;
```

Η process είναι ευαίσθητη σε κάθε αλλαγή είτε της εισόδου δεδομένων είτε της εισόδου επίτρεψης.

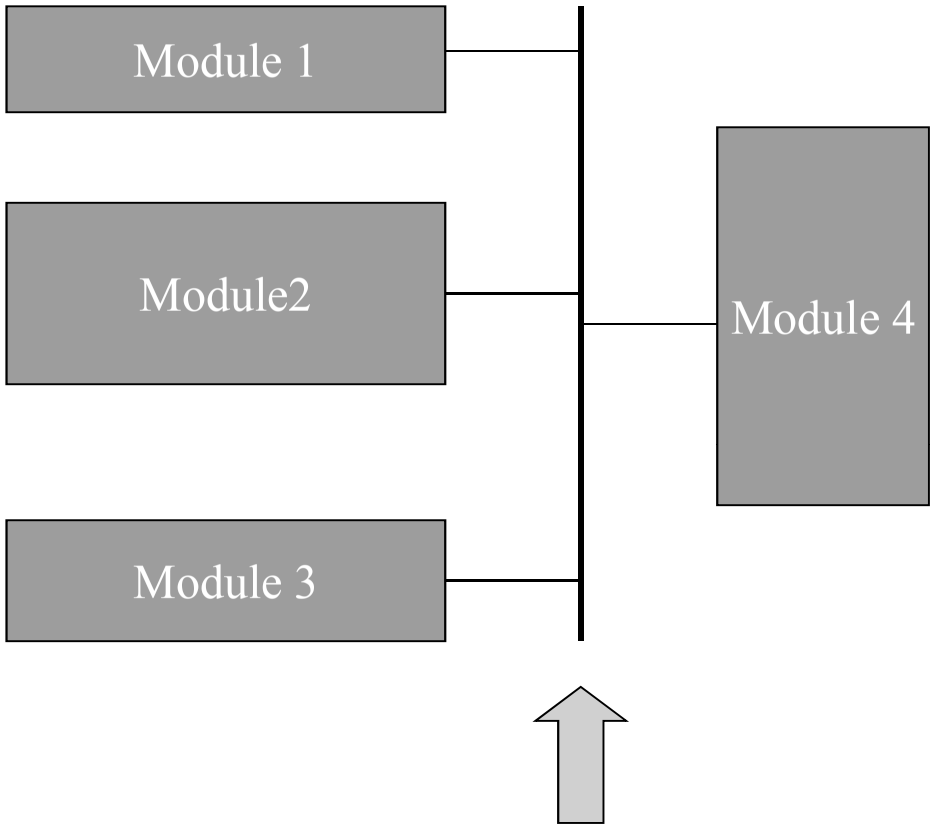
Με επίτρεψη ενεργή οποιαδήποτε μεταβολή της εισόδου περνάει στην έξοδο.

Latch

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Latch is
  port(Rst,LE, D : in  std_logic; Q: out std_logic);
end Latch;

architecture RTL of Latch is
begin
  seq0 : process (Rst,D, LE)
  begin
      if Rst = '1' then Q <= '0';
      elsif LE = '1' then Q <= D;
      end if;
  end process;
end RTL;
```

Η διαδικασία μηδένισης είναι όμοια με την ασύγχρονη μηδένιση των D-FF

Buffer 3-καταστάσεων



Χρήση υψηλής εμπέδησης

Buffer 3-καταστάσεων

- Η έξοδος του buffer είναι ίδια με την είσοδο όταν EN='1' αλλιώς είναι σε κατάσταση υψηλής εμπέδησης.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Buffer is
  port(EN, D : in  std_logic; Q: out std_logic);
end Buffer;

architecture RTL of Buffer is
begin
  seq0 : process (D, EN)
  begin
      if EN = '1' then Q <= D;
      else Q <= 'Z';
      end if;
  end process;
end RTL;
```

Συνδυαστικές Πύλες

- Υπάρχουν πολλοί τρόποι περιγραφής συνδυαστικών πυλών

```
library IEEE;
use IEEE.std_logic_1164.all;
entity NORGATE is
  port( D1,D2:in  std_logic;
  Y1, Y2, Y3, Y4: out std_logic);
end NORGATE;
architecture RTL of NORGATE is
begin
  Y1 <= D1 nor D2;
  Y2 <= '0' when (D1 or  D2) = '1' else '1';
  Y3 <= '1' when (D1 nor D2) = '1' else '0';
```

```
p0 : process (D1, D2)
  begin
    if (D1 = '1') or (D2 = '1') then
      Y4 <= '0';
    else
      Y4 <= '1';
    end if;
  end process;
end RTL;
```

Πύλες NOR

Πολυπλέκτες

- Υπάρχουν πολλοί τρόποι περιγραφής πολυπλεκτών

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX2 is
  port( D1,D2,SEL:in  std_logic;
  Y1, Y2, Y3, Y4: out std_logic);
end MUX2;

architecture RTL of MUX2 is
begin
  Y1 <= D1 when SEL='0' else D2;
  with SEL select
      Y2 <= D1 when '0', D2 when others;
```

```
p0 : process (D1, D2,SEL)
  begin
    if (SEL = '0') then Y3 <= D1;
    else Y3 <= D2;  end if;
  end process;

p1 : process (D1, D2,SEL)
  begin
    case SEL is
          when '0' => Y4 <= D1;
          when others=> Y4<=D2;
    end case;
  end process;
end RTL;
```

Σχεδίαση με πίνακες αλήθειας

Παράδειγμα: BCD ⇒ Excess-3

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bcd_to_excess3 is
generic (delay: time);
  port(
  BCD: in  std_logic_vector(3 downto 0);
  Exc: out std_logic_vector(3 downto 0));
end bcd_to_excess3;

architecture behave of bcd_to_excess3 is
begin
  convert: process (BCD)
  begin
  case input is
       when "0000" => Exc<="0011" after delay;
       when "0001" => Exc<="0100" after delay;
       when "0010" => Exc<="0101" after delay;
       when "0011" => Exc<="0110" after delay;
       when "0100" => Exc<="0111" after delay;
       when "0101" => Exc<="1000" after delay;
       when "0110" => Exc<="1001" after delay;
       when "0111" => Exc<="1010" after delay;
       when "1000" => Exc<="1011" after delay;
       when "1001" => Exc<="1100" after delay;
       when others  => Exc<="XXXX";
  end case;
  end process;
```

Σχεδίαση με πίνακες αλήθειας

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bcd_to_excess3 is
generic (delay: time);
port(
  BCD: in  std_logic_vector(3 downto 0);
  Exc: out std_logic_vector(3 downto 0));
end bcd_to_excess3;

architecture behave of bcd_to_excess3 is
type truth is array (0 to 9) of std_logic_vector (3 downto 0);
constant truth_table: truth:= ("0011", "0100", "0101","0110", "0111", "1000", "1001",
                               "1010", "1011", "1100")
begin
     Exc <= truth_table(conv_integer(BCD));
end behave;
```

Κανόνες σύνθεσης VHDL

Οι βασικοί κανόνες σύνθεσης με VHDL είναι οι ακόλουθοι:

- Όταν υπάρχει έκφραση που ανιχνεύει την ακμή ενός σήματος (πχ. CLK'event and CLK='1') τότε τα σήματα που αναθέτονται (target) συνθέτονται ως flip flops.
- Οι ακμές σημάτων μπορεί να είναι θετικές ή αρνητικές αλλά καλό είναι να χρησιμοποιείται μόνο μία κατηγορία από αυτές.
- Όταν σε ένα σήμα αναθέτουμε την τιμή 'Z' τότε ένας buffer 3-καταστάσεων χρησιμοποιείται.
- Όταν δεν υπάρχει ανίχνευση ακμής σήματος και ένα σήμα δεν παίρνει τιμή για κάθε πιθανή περίπτωση τότε χρησιμοποιείται latch.
- Όταν δεν ισχύει καμία από τις παραπάνω περιπτώσεις έχουμε σύνθεση συνδυαστικών κυκλωμάτων.

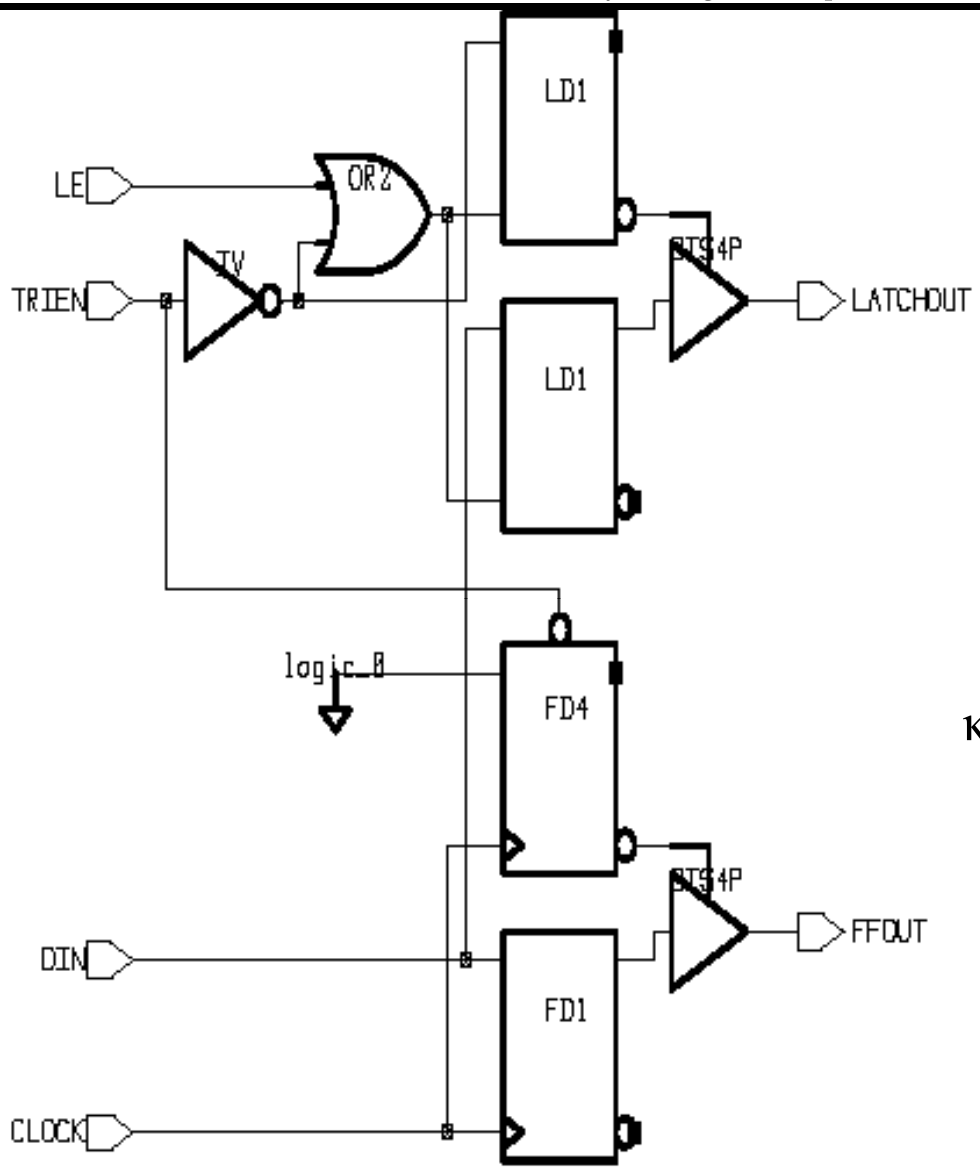
Συνδυασμός Περιπτώσεων

```
entity TRIREG is
  port ( CLOCK, LE, TRIEN, DIN : in  std_logic;
    FFOUT, LATCHOUT       : out std_Logic);
end TRIREG;

architecture RTL of TRIREG is
begin
  ff0 : process (TRIEN, CLOCK)
  begin
    if (TRIEN = '0') then  FFOUT <= 'Z';
    elsif (CLOCK'event and CLOCK = '1') then
      FFOUT <= DIN;  end if;
  end process;

  latch0 : process (LE, TRIEN, DIN)
  begin
    if (TRIEN = '0') then LATCHOUT <= 'Z';
    elsif (LE = '1') then  LATCHOUT <= DIN;
    end if;
  end process;
end RTL;
```

Σύνθεση Flip flop και buffer 3 καταστάσεων

Σύνθεση latch (λείπει η κατάσταση LE=‘0’) και buffer 3 καταστάσεων

Συνδυασμός περιπτώσεων



Ο συνδυασμός των περιπτώσεων χωρίς να είναι σαφής ο διαχωρισμός περιπτώσεων οδηγεί σε κύκλωμα μεγαλύτερο και μη αναμενόμενο.

(2 ff και 2 latches σε κάθε κύκλωμα)

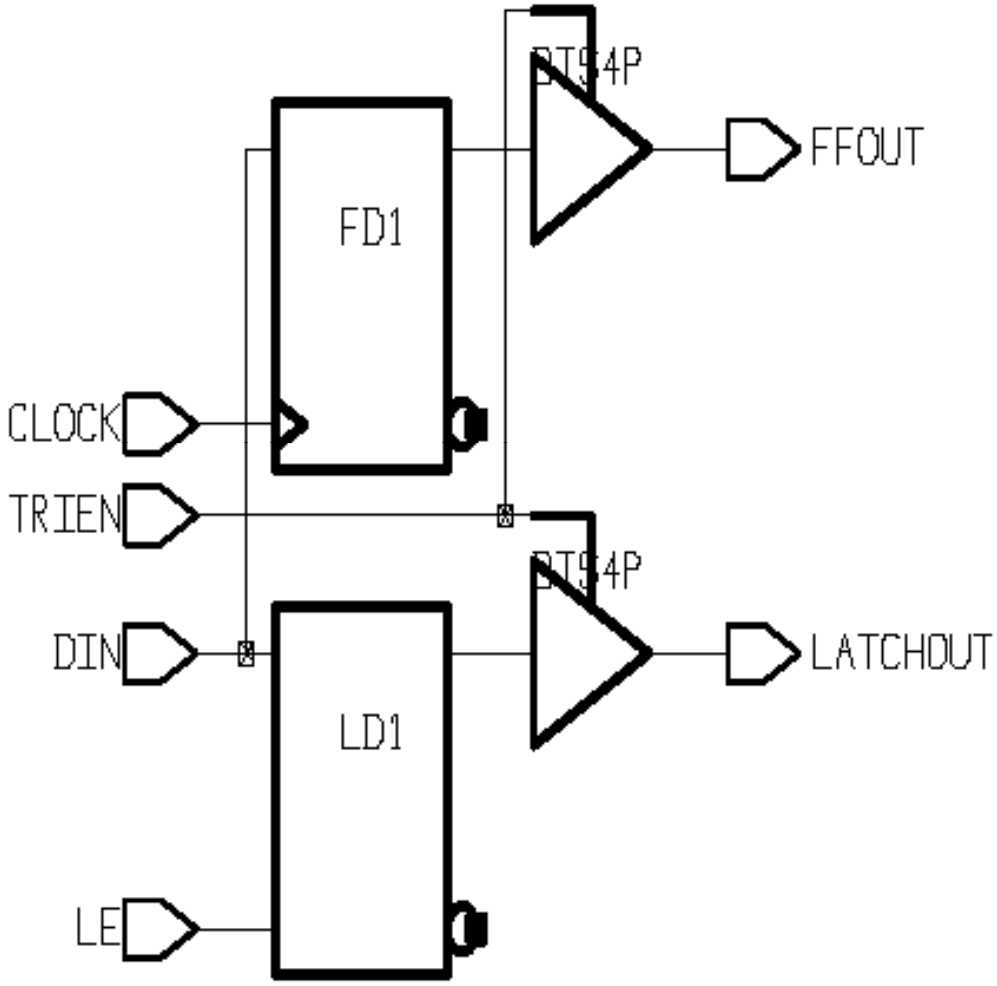
Σωστός Συνδυασμός Περιπτώσεων

```
architecture RTLA of TRIREG is
  signal DFF, LATCH : std_logic;
begin
  ff0 : process
  begin
     wait until (CLOCK'event and CLOCK = '1') ;
     DFF <= DIN;
  end process;
  FFOUT <= DFF when TRIEN = '1' else 'Z';

  latch0 : process (LE, DIN)
  begin
    if (LE = '1') then
      LATCH <= DIN;
    end if;
  end process;
  LATCHOUT <= LATCH when TRIEN = '1' else 'Z';
end RTLA;
```

Σαφής διαχωρισμός περιπτώσεων.

(1 ff και 1 latches σε κάθε κύκλωμα)

Σωστός Συνδυασμός Περιπτώσεων



Συνδυαστικό Κύκλωμα Επιλογέα

- Ένα συνδυαστικό κύκλωμα επιλογέα δέχεται N εισόδους και με $\lceil \mathrm{Log}_2 N \rceil$ γραμμές επιλογής επιλέγει την είσοδο που περνά στην έξοδο.
- Η δήλωση της οντότητας για N=16 φαίνεται ακόλουθα:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
  port (
    A   : in  std_logic_vector (15 downto 0);
    SEL : in  std_logic_vector ( 3 downto 0);
    Y   : out std_logic);
end SELECTOR;
```

Αρχιτεκτονική I

```
architecture RTL1 of SELECTOR is
begin
 p0 : process (A, SEL)
 begin
  if    (SEL = "0000") then   Y <= A(0);
  elsif (SEL = "0001") then   Y <= A(1);
  elsif (SEL = "0010") then   Y <= A(2);
  elsif (SEL = "0011") then   Y <= A(3);
  elsif (SEL = "0100") then   Y <= A(4);
  elsif (SEL = "0101") then   Y <= A(5);
  elsif (SEL = "0110") then   Y <= A(6);
  elsif (SEL = "0111") then   Y <= A(7);
  elsif (SEL = "1000") then   Y <= A(8);
  elsif (SEL = "1001") then   Y <= A(9);
  elsif (SEL = "1010") then   Y <= A(10);
  elsif (SEL = "1011") then   Y <= A(11);
  elsif (SEL = "1100") then   Y <= A(12);
  elsif (SEL = "1101") then   Y <= A(13);
  elsif (SEL = "1110") then   Y <= A(14);
  else   Y <= A(15);
  end if;
 end process;
end RTL1;
```

Αρχιτεκτονική II

```
architecture RTL2 of SELECTOR is
begin
 p1 : process (A, SEL)
 begin
  case SEL is
    when "0000" => Y <= A(0);
    when "0001" => Y <= A(1);
    when "0010" => Y <= A(2);
    when "0011" => Y <= A(3);
    when "0100" => Y <= A(4);
    when "0101" => Y <= A(5);
    when "0110" => Y <= A(6);
    when "0111" => Y <= A(7);
    when "1000" => Y <= A(8);
    when "1001" => Y <= A(9);
    when "1010" => Y <= A(10);
    when "1011" => Y <= A(11);
    when "1100" => Y <= A(12);
    when "1101" => Y <= A(13);
    when "1110" => Y <= A(14);
    when others => Y <= A(15);
  end case;
 end process;
end RTL2;
```

Αρχιτεκτονική III-IV

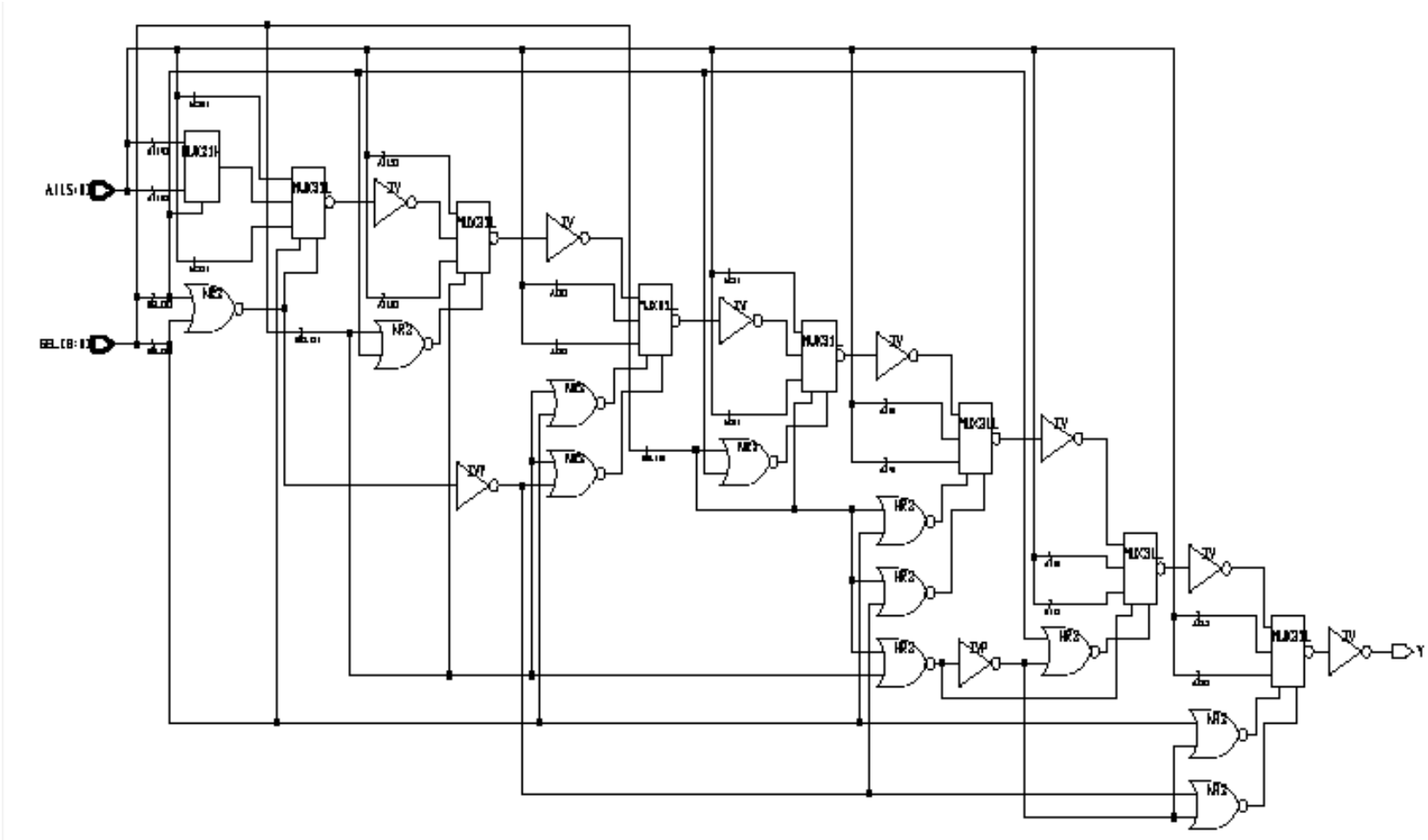
```
architecture RTL3 of SELECTOR is
begin
 with SEL select
  Y <=   A(0)  when "0000",
         A(1)  when "0001",
         A(2)  when "0010",
         A(3)  when "0011",
         A(4)  when "0100",
         A(5)  when "0101",
         A(6)  when "0110",
         A(7)  when "0111",
         A(8)  when "1000",
         A(9)  when "1001",
         A(10) when "1010",
         A(11) when "1011",
         A(12) when "1100",
         A(13) when "1101",
         A(14) when "1110",
         A(15) when others;
end RTL3;
```

Αρχιτεκτονική IV

```
architecture RTL4 of SELECTOR is
begin
  Y <= A(conv_integer(SEL));
end RTL4;
```

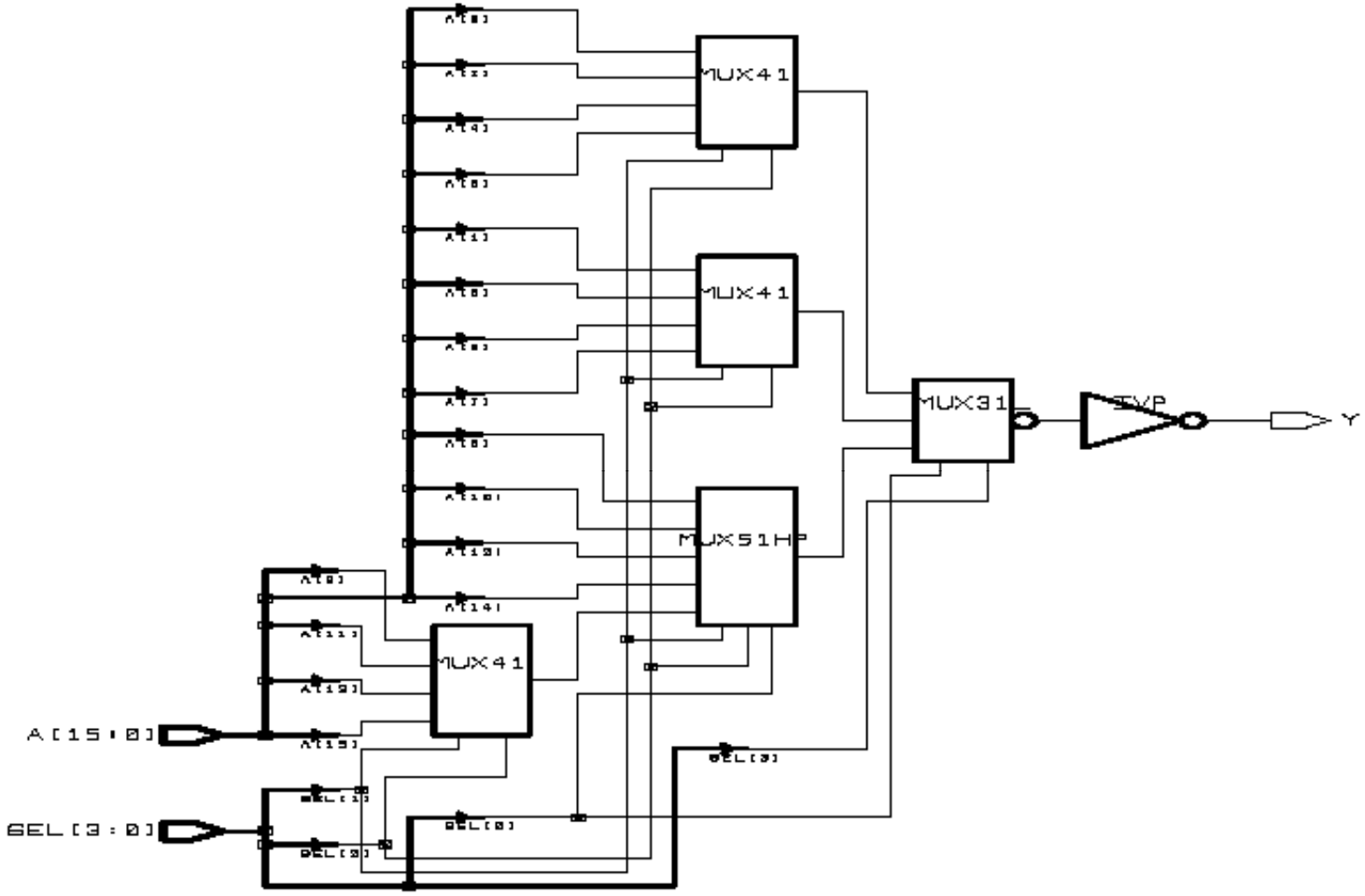
std_logic_unsigned

Αποτελέσματα Σύνθεσης



Αρχιτεκτονική I

Αποτελέσματα Σύνθεσης



Αρχιτεκτονική II, III, IV

Αξιολόγηση Αποτελεσμάτων Σύνθεσης

- Από τις αναφορές επιφάνειας – ταχύτητας φαίνεται ότι η Αρχιτεκτονική I απαιτεί μεγαλύτερο και πιο αργό κύκλωμα όταν συντεθεί.
- Οι αρχιτεκτονικές II, III, IV οδηγούν στο ίδιο σχηματικό.
- Το πρόβλημα δημιουργεί η δομή **if then else** η οποία οδηγεί σε μία αλυσίδα από συγκρίσεις με προτεραιότητα: κάθε προηγούμενο if ελέγχεται πριν το επόμενο.
- Οι υπόλοιπες αρχιτεκτονικές δίνουν την δυνατότητα παραλληλισμού οπότε οδηγούν σε μικρότερα και γρηγορότερα κυκλώματα.

Συμπέρασμα: όπου είναι δυνατόν, πρέπει να αποφεύγεται η χρήση της δομής **if then else**.

Κωδικοποιητής

Σχεδιασμός κυκλώματος που δέχεται σαν είσοδο 4-bit επιλογή και ενεργοποιεί την αντίστοιχη από 16 εξόδους.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ENCODER is
  port (
    SEL : in  std_logic_vector(3 downto 0);
    Y   : out std_logic_vector(15 downto 0));
end ENCODER;
```

Αρχιτεκτονική I

```
architecture RTL1 of ENCODER is
begin
  p0 : process (SEL)
  begin
    Y <= (Y'range => '1');
    if    (SEL = "0000") then Y(0) <= '0';
    elsif (SEL = "0001") then Y(1) <= '0';
    elsif (SEL = "0010") then Y(2) <= '0';
    elsif (SEL = "0011") then Y(3) <= '0';
    elsif (SEL = "0100") then Y(4) <= '0';
    elsif (SEL = "0101") then Y(5) <= '0';
    elsif (SEL = "0110") then Y(6) <= '0';
    elsif (SEL = "0111") then Y(7) <= '0';
    elsif (SEL = "1000") then Y(8) <= '0';
    elsif (SEL = "1001") then Y(9) <= '0';
    elsif (SEL = "1010") then Y(10) <= '0';
    elsif (SEL = "1011") then Y(11) <= '0';
    elsif (SEL = "1100") then Y(12) <= '0';
    elsif (SEL = "1101") then Y(13) <= '0';
    elsif (SEL = "1110") then Y(14) <= '0';
    else Y(15) <= '0';
    end if;
  end process;
end RTL1;
```

Αρχιτεκτονική II

```
architecture RTL2 of ENCODER is
begin
  p1 : process (SEL)
  begin
    Y <= (Y'range => '1');
    case SEL is
      when "0000" => Y(0)  <= '0';
      when "0001" => Y(1)  <= '0';
      when "0010" => Y(2)  <= '0';
      when "0011" => Y(3)  <= '0';
      when "0100" => Y(4)  <= '0';
      when "0101" => Y(5)  <= '0';
      when "0110" => Y(6)  <= '0';
      when "0111" => Y(7)  <= '0';
      when "1000" => Y(8)  <= '0';
      when "1001" => Y(9)  <= '0';
      when "1010" => Y(10) <= '0';
      when "1011" => Y(11) <= '0';
      when "1100" => Y(12) <= '0';
      when "1101" => Y(13) <= '0';
      when "1110" => Y(14) <= '0';
      when others => Y(15) <= '0';
    end case;
  end process;
end RTL2;
```

Αρχιτεκτονική III - IV

```
architecture RTL3 of ENCODER is
begin
  with SEL select
    Y <= "1111111111111110" when "0000",
         "1111111111111101" when "0001",
         "1111111111111011" when "0010",
         "1111111111110111" when "0011",
         "1111111111101111" when "0100",
         "1111111111011111" when "0101",
         "1111111110111111" when "0110",
         "1111111101111111" when "0111",
         "1111111011111111" when "1000",
         "1111110111111111" when "1001",
         "1111101111111111" when "1010",
         "1111011111111111" when "1011",
         "1110111111111111" when "1100",
         "1101111111111111" when "1101",
         "1011111111111111" when "1110",
         "0111111111111111" when others;
end RTL3;
```

```
architecture RTL4 of ENCODER is
begin
  p3 : process (SEL)
  begin
    Y <= (Y'range => '1');
    Y(conv_integer(SEL)) <= '0';
  end process;
end RTL4;
```

Ελεγκτής Ισότητας

Σχεδιασμός κυκλώματος που δέχεται σαν είσοδο 2 N-bit διανύσματα και δίνει μία έξοδο ίση με '1' όταν τα δύο διανύσματα είναι ίδια.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity EQUAL is
  generic (N : in integer := 8);
  port (
    A, B  : in  std_logic_vector(N-1 downto 0);
    SAME  : out std_logic);
end EQUAL;
```

Αρχιτεκτονική I

```
architecture RTL1 of EQUAL is
begin
  p0 : process (A, B)
    variable SAME_SO_FAR : std_logic;
  begin
    SAME_SO_FAR := '1';
    for i in A'range loop
      if (A(i) /= B(i)) then
        SAME_SO_FAR := '0';
        exit;
      end if;
    end loop;
    SAME <= SAME_SO_FAR;
  end process;
end RTL1;
```

Αλγοριθμική προσέγγιση

Αρχιτεκτονική II-III

```
architecture RTL2 of EQUAL is
  signal EACHBIT : std_logic_vector(A'length-1 downto 0);
begin
  xnor_gen : for i in A'range generate
    EACHBIT(i) <= not (A(i) xor B(i));
  end generate;
  p0 : process (EACHBIT)
    variable BITSAME : std_logic;
  begin
    BITSAME := EACHBIT(0);
    for i in 1 to A'length-1 loop
      BITSAME := BITSAME and EACHBIT(i);
    end loop;
    SAME <= BITSAME;
  end process;
end RTL2;
```

Χρήση πυλών XNOR για σύγκριση ανά δύο των bits και μίας πύλης AND για συγκέντρωση αποτελεσμάτων

Αρχιτεκτονική III

```
architecture RTL3 of EQUAL is
begin
  SAME <= '1' when A = B else '0';
end RTL3;
```

Αθροιστές

Χρήση Concurrent statements

Ημιαθροιστής

```
library IEEE;
use IEEE.std_logic_1164.all;
entity HA is
  port (
    A, B  : in  std_logic;
    S, C  : out std_logic);
end HA;

architecture RTL of HA is
begin
  S <= A xor B;
  C <= A and B;
end RTL;
```

Πλήρης αθροιστής

```
library IEEE;
use IEEE.std_logic_1164.all;
entity FA is
  port (
    A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
end FA;

architecture RTL of FA is
begin
  S    <= A xor B xor CI;
  COUT <= (A and B) or (A and CI) or (B and CI);
end RTL;
```

Αθροιστές

Πακέτο Πυλών

```
library IEEE;
use IEEE.std_logic_1164.all;
package GATECOMP is
  component AND2
    port ( A,B: in  std_logic;
           Y : out std_logic );
  end component;
  component OR3
    port ( A, B, C : in  std_logic;
           Y : out std_logic );
  end component;
  component XOR2
    port (A,B : in  std_logic;
           Y : out std_logic );
  end component;
end GATECOMP;
```

Όταν επιθυμούμε συγκεκριμένη υλοποίηση χρησιμοποιούμε δομική περιγραφή

Πλήρης αθροιστής

```
use work.GATECOMP.all;
architecture COMP of FA is
  signal S1, A1, A2, A3 : std_logic;
begin
  xor_0 : XOR2 port map (A,  B,  S1);
  xor_1 : XOR2 port map (CI, S1, S);
  and_0 : AND2 port map (A,  B,  A1);
  and_1 : AND2 port map (A,  CI, A2);
  and_2 : AND2 port map (B,  CI, A3);
  or_0  :  OR3 port map (A1, A2, A3, COUT);
end COMP;
```

Αθροιστές Ριπής Ι

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RPADDER is
  generic (N : in integer := 8);
  port ( A, B  : in  std_logic_vector(N-1 downto 0);
    CI    : in  std_logic;
    S     : out std_logic_vector(N-1 downto 0);
    COUT  : out std_logic);
end RPADDER;

architecture RTL1 of RPADDER is
  component FA
    port ( A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
  end component;
  signal C : std_logic_vector(A'length-1 downto 1);
```

Ο αθροιστής ριπής μπορεί να σχεδιαστεί με μία αλυσίδα Πλήρων Αθροιστών

Χρήση του Πλήρους Αθροιστή. Βρίσκεται στην βιβλιοθήκη work από την προηγούμενη ενότητα

Αθροιστές Ριπής Ι

```
begin
  gen : for j in A'range generate
    genlsb : if j = 0 generate
      fa0 : FA port map (A => A(0), B => B(0),
        CI => CI, S => S(0), COUT => C(1));
    end generate;
    genmid : if (j > 0) and (j < A'length-1) generate
      fa0 : FA port map (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => C(j+1));
    end generate;
    genmsb : if j = A'length-1 generate
      fa0 : FA port map (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => COUT);
    end generate;
  end generate;
end RTL1;
```

Τα generate statements αυτοματοποιούν την διαδικασία δημιουργίας δομικών περιγραφών όταν αφορούν επαναληπτικές δομές.

Αθροιστές Ριπής I

```
architecture RTL1 of RPADDER is
  component FA
    port ( A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
  end component;
  signal C : std_logic_vector(A'length downto 0);

begin
C(0)<=CI;
COUT<=C(A´length)
gen : for j in A'range generate
   fa0 : FA port map (A => A(j), B => B(j), CI => C(j), S => S(j), COUT =>C(j+1));
end generate;
end RTL1;
```

Αθροιστές Ριπής ΙΙ

```
architecture RTL2 of RPADDER is
  procedure FADDER (
    signal A, B, CI : in  std_logic;
    signal S, COUT  : out std_logic) is
  begin
    S    <= A xor B xor CI;
    COUT <= (A and B) or (A and CI)
              or (B and CI);
  end FADDER;
  signal C :
     std_logic_vector(A'length-1 downto 1);

begin
gen : for j in A'range generate
    genlab : if j = 0 generate
      FADDER (A => A(0), B => B(0),
        CI => CI, S => S(0), COUT => C(1));
    end generate;
genmid : if (j> 0) and (j < A'length-1) generate
      FADDER (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => C(j+1));
 end generate;

 genmsb : if j = A'length-1 generate
      FADDER (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => COUT);
  end generate;
end generate;
end RTL2;
```

Ορισμός του Πλήρους Αθροιστή ως procedure

Πολλαπλασιαστές

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mult is
  generic (product_delay : time);
  port ( a, b : in  std_logic_vector(3 downto 0);
           product : out std_logic_vector(7 downto 0));
end mult;

architecture behave of mult is
begin
   product <= std_logic_vector (unsigned(a) * unsigned(b)) after product_delay;
end;
```

Μνήμη ROM 8x8

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity rom8x8 is
  generic (delay : time);
  port ( read_enable : in  std_logic; address: in std_logic_vector(2 downto 0);
         data_out : out std_logic_vector(7 downto 0));
end rom8x8;

architecture behave of rom8x8 is
begin
    process (read_enable, address)
    begin
```

Μνήμη ROM 8x8

```
    if (read_enable = '1') then
        case (address) is
            when O"0" => data_out <= X"12" after delay;
            when O"1" => data_out <= X"18" after delay;
            when O"2" => data_out <= X"f2" after delay;
            when O"3" => data_out <= X"e2" after delay;
            when O"4" => data_out <= X"13" after delay;
            when O"5" => data_out <= X"02" after delay;
            when O"6" => data_out <= X"10" after delay;
            when O"7" => data_out <= X"23" after delay;
        end case;
    end if;
  end process;
end behave;
```

Για μεγάλες μνήμες η περιγραφή δεν είναι αποδοτική

Μνήμη ROM 8 x 8 (version 2)

```
type matrix is array (integer range <>) of std_logic_vector(7 downto 0); -- in a package

entity rom8x8 is
  generic (delay : time);
  port ( read_enable : in  std_logic; address: in std_logic_vector(2 downto 0);
           data_out : out std_logic_vector(7 downto 0));
end rom8x8;

architecture behave of rom8x8 is
signal mem: matrix (7 downto 0) :={x"12", x"18", x"f2", x"e2", x"13", x"02", x"10", x"23"};
begin
    process (read_enable, address)
    begin
      if (read_enable = '1') then
          data_out <= mem(conv_integer(address)) after delay;
     end if;
  end process;
end behave;
```

Μνήμη ROM n x n

```
    if (read_enable = '1') then
        case (address) is
            when O"0" => data_out <= X"12" after delay;
            when O"1" => data_out <= X"18" after delay;
            when O"2" => data_out <= X"f2" after delay;
            when O"3" => data_out <= X"e2" after delay;
            when O"4" => data_out <= X"13" after delay;
            when O"5" => data_out <= X"02" after delay;
            when O"6" => data_out <= X"10" after delay;
            when O"7" => data_out <= X"23" after delay;
        end case;
    end if;
  end process;
end behave;
```

ALU 8 πράξεων

Selector	Function
000	x+y
001	x-y
010	x+1
011	x-1
100	x or y
101	not x
110	x and y
111	x xor y

ALU 8 πράξεων (package)

Στο πακέτο δηλώνουμε τις βασικές σταθερές και πράξεις.

```
package arith_operations is
  constant size: integer :=8;
  constant select_bits: integer :=3;
  procedure add (  x, y: in bit_vector(size-1 downto 0);
                   sum: out bit_vector(size-1 downto 0), c_out: out bit);
  procedure sub (  x, y: in bit_vector(size-1 downto 0);
                   d: out bit_vector(size-1 downto 0), c_out: out bit);
  function inc (x: in bit_vector) return bit_vector;
  function dec (x: in bit_vector) return bit_vector;
end arith_operations;

package body arith_operations is
```

Στο σώμα θα ενσωματώσουμε τις συναρτήσεις – διαδικασίες

ALU 8 πράξεων (package body/1)

```
procedure add ( x, y: in bit_vector(size-1 downto 0); sum: out bit_vector(size-1 downto 0),
                c_out: out bit);
variable s: bit_vector (size downto 0);
begin
        s:= '0' & x + '0' & y;
        sum:= s(size-1 downto 0);
        c_out:= s(size);
end add;
procedure sub ( x, y: in bit_vector(size-1 downto 0); d: out bit_vector(size-1 downto 0),
                c_out: out bit);
variable s: bit_vector (size downto 0);
begin
        s:= '0' & x + not ('0' & y) + ((size-1 downto 0)=>'0') & '1';
        d:= s(size-1 downto 0);
        c_out:= s(size);
end add;
```

ALU 8 πράξεων (package body/2)

```
function inc ( x: in bit_vector(size-1 downto 0)) return bit_vector is
variable x1: bit_vector (size-1 downto 0);
begin
        x1:= x + ((size-2 downto 0)=>'0') & '1';
        return x1;
end inc;
function dec ( x: in bit_vector(size-1 downto 0)) return bit_vector is
variable x1: bit_vector (size-1 downto 0);
begin
        x1:= x + ((size-1 downto 0)=>'1'); -- x + 2s complement of 00…01
        return x1;
end inc;
end arith_operations;
```

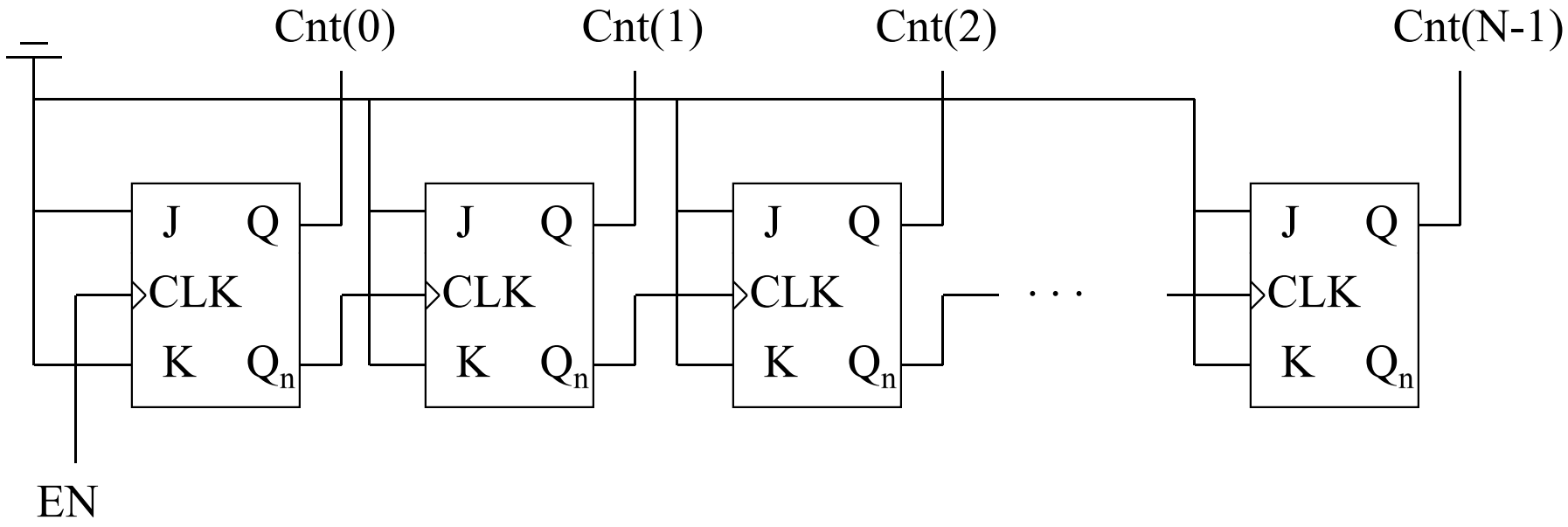
ALU 8 πράξεων

```
use work.arith_operations.all;
entity alu is
  generic (delay :time);
  port (     x, y: in bit_vector (size-1 downto 0);
             function_select: in bit_vector (select_bits-1 downto 0);
             f: out bit_vector (size-1 downto 0));
end alu;

architecture behave of alu is
begin
    p0: process (x,y,function_select)
    variable f_out: bit_vector (size-1 downto 0);
    begin
      case function_select is
        when "000" => add (x, y, f_out);
                      f<=f_out after delay;
        when "001" => sub (x, y, f_out);
                      f<=f_out after delay;
        when "010" => f<=inc (x) after delay;
        when "011" => f<=dec (x) after delay;
        when "100" => x or y after delay;
        when "101" => not x after delay;
        when "110" => x and y after delay;
        when "111" => x xor y after delay;
      end case;
    end process;
end bahave;
```

Μετρητές

- Υπάρχουν δύο είδη μετρητών: σύγχρονοι και ασύγχρονοι.
- Οι σύγχρονοι μετρητές έχουν διασυνδεδεμένα όλα τα flip flops σε ένα ρολόι και αλλάζουν κατάσταση στην ακμή του ρολογιού.
- Οι ασύγχρονοι μετρητές δεν έχουν κοινό ρολόι αλλά ένα flip flop μπορεί να ενεργοποιεί ένα άλλο κ.ο.κ.
- Τα flip flop που χρησιμοποιούνται για την κατασκευή μετρητών είναι αυτά που έχουν την δυνατότητα να αντιστρέφουν την κατάστασή τους (όπως τα JK, T).
- Μπορούμε να μοντελοποιήσουμε την συμπεριφορά ενός JK Flip flop δίνοντας και χρόνους καθυστέρησης.
- Οι καθυστερήσεις που περιγράφονται στην VHDL δεν συντίθενται αλλά χρησιμοποιούνται μόνο για λόγους εξομοίωσης.

Ασύγχρονος Μετρητής



Ασύγχρονος Μετρητής

```
library IEEE;
use IEEE.std_logic_1164.all;
entity ACNT is
  port (
    RSTn, EN : in  std_logic;
    CNTR: out
      std_logic_vector(7 downto 0));
end ACNT;

architecture RTL of ACNT is
  component JKFF
  port (
    CLK, RSTn, J, K : in  std_logic;
    Q, Qn           : out std_logic);
  end component;
  signal FFQ: std_logic_vector(8 downto 0);
  signal FFQn: std_logic_vector(8 downto 0);
  signal VDD: std_logic;
begin
  VDD <= '1'; FFQn(0) <= EN;
  jk0 : for j in 1 to 8 generate
    b17 : JKFF port map (CLK => FFQn(j-1),
            RSTn => RSTn, J => VDD, K => VDD,
            Q => FFQ(j), Qn => FFQn(j));
  end generate;
  CNTR <= FFQ(8 downto 1);
end RTL;
```

Παράδειγμα απλής ιεραρχικής σχεδίασης

Σύγχρονος Μετρητής

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity SCNT is
  port (
    CLK, RSTn, EN : in  std_logic;
    CNTR: out std_logic_vector(7 downto 0));
end SCNT;

architecture RTL of SCNT is
  signal FF : std_logic_vector(7 downto 0);
begin
  process (CLK, RSTn)
   begin
    if (RSTn = '0') then
      FF <= (FF'range => '0');
    elsif (CLK'event and CLK = '1') then
      if (EN = '1') then
        FF <= FF + 1;
      end if;
    end if;
  end process;
  CNTR <= FF;
end RTL;
```

Καταχωρητής ολίσθησης

```
library IEEE;
use IEEE.std_logic_1164.all;

entity SHIFTREN is
  port (
    CLK, RSTn, SI, EN : in  std_logic;
    SO: out std_logic);
end SHIFTREN;

architecture RTL of SHIFTREN is
  signal FF8 : std_logic_vector(7 downto 0);
begin
```

```
posedge : process (RSTn, CLK)
begin
    if (RSTn = '0') then
      FF8  <= (FF8'range => '0');
    elsif (CLK'event and CLK = '1') then
      if (EN = '1') then
            FF8 <= SI & FF8(FF8'length-1
                                  downto 1);
      end if;
    end if;
  end process;
  SO <= FF8(0);
end RTL;
```

Μνήμη RAM

```
package RAM_DECL is
  constant address_size: integer:=8;
  constant word_size: integer:=8;
  type matrix is array (0 to 2**address_size-1) of bit_vector(word_size-1 downto 0);
end RAM_DECL;

use work.RAM_DECL.all;
entity ram is
generic (delay: time);
port (      mode, enable, clk: in bit;
            address: in bit_vector (address_size-1 downto 0);
            data_in: in bit;
            data_out: out bit);
end ram;
```

Μνήμη RAM

```
architecture behave of ram is
  signal mem: matrix;
begin
read_proc: process (clk, mode)
begin
    if clk='1' and clk'event then
        if enable='1' then
            if mode='0' then data_out<=mem(conv_integer(address)) after delay;
            elsif mode='1' then mem(conv_integer(address))<=data_out after delay;
            end if;
       end if;
    end if;
end process;
end ram;
```

Μνήμη RAM Dual-Port

Μία θύρα ανάγνωσης/εγγραφής – Δεύτερη θύρα ανάγνωσης μόνο

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

entity dual_port_SSRAM is
  port ( clk          : in  std_logic;
         en1, wr1     : in  std_logic;
         a1           : in  unsigned(11 downto 0);
         d_in1        : in  std_logic_vector(15 downto 0);
         d_out1       : out std_logic_vector(15 downto 0);
         en2          : in  std_logic;
          a2          : in  unsigned(11 downto 0);
         d_out2       : out std_logic_vector(15 downto 0) );
end entity dual_port_SSRAM;
```

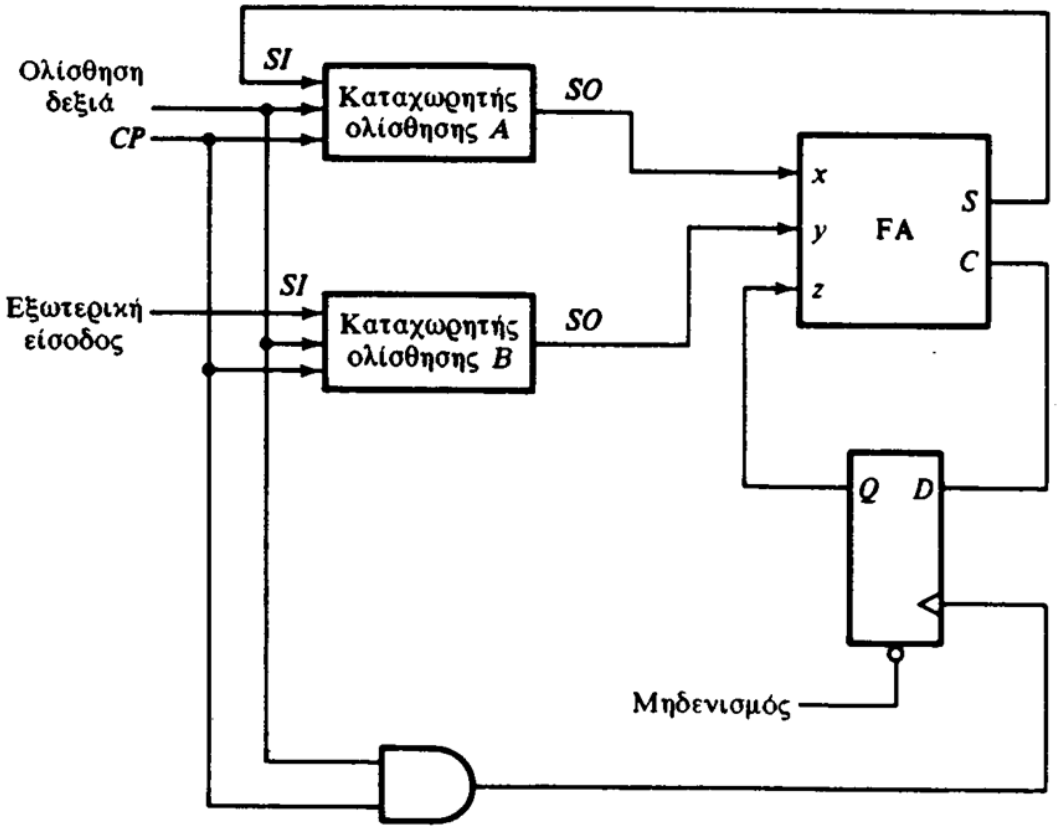
Μνήμη RAM Dual-Port

```
architecture synth of dual_port_SSRAM is
  type RAM_4Kx16 is array (0 to 4095) of std_logic_vector(15 downto 0);
  signal data_RAM : RAM_4Kx16;
begin

  read_write_port : process (clk) is
  begin
    if rising_edge(clk) then
      if en1 = '1' then
        if wr1 = '1' then
          data_RAM(to_integer(a1)) <= d_in1;  d_out1 <= d_in1;
        else d_out1 <= data_RAM(to_integer(a1));
        end if;
      end if;
    end if;
  end process read_write_port;
```

Μνήμη RAM Dual-Port

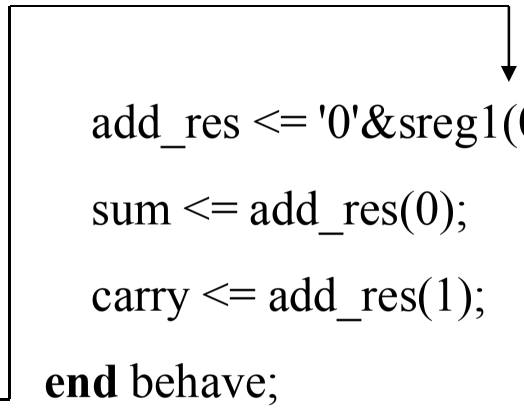
```
 read_only_port : process (clk) is
 begin
   if rising_edge(clk) then
     if en2 = '1' then
       d_out2 <= data_RAM(to_integer(a2));
     end if;
   end if;
 end process read_only_port;

end architecture synth;
```

Σειριακός αθροιστής



Σειριακός αθροιστής

```
entity serial_adder is port
    ( clk, s_in, sh_ctrl: in bit; sreg1: inout bit_vector (3 downto 0) );
end serial_adder;

architecture behave of serial_adder is
signal sum, carry, Q: bit;
signal add_res: bit_vector(1 downto 0);
signal sreg2: bit_vector (3 downto 0);
begin
SR_proc: process (sh_ctrl, clk)
begin
 if (sh_ctrl='1')
  if (clk'event and clk='1') then
   sreg1 <= sum & sreg1(3 downto 1);
   sreg2 <= s_in & sreg2(3 downto 1);
   Q<=carry;
  end if;
 end if;
end process;

    add_res <= '0'&sreg1(0) + '0'&sreg2 +'0'&Q;
    sum <= add_res(0);
    carry <= add_res(1);
end behave;
```



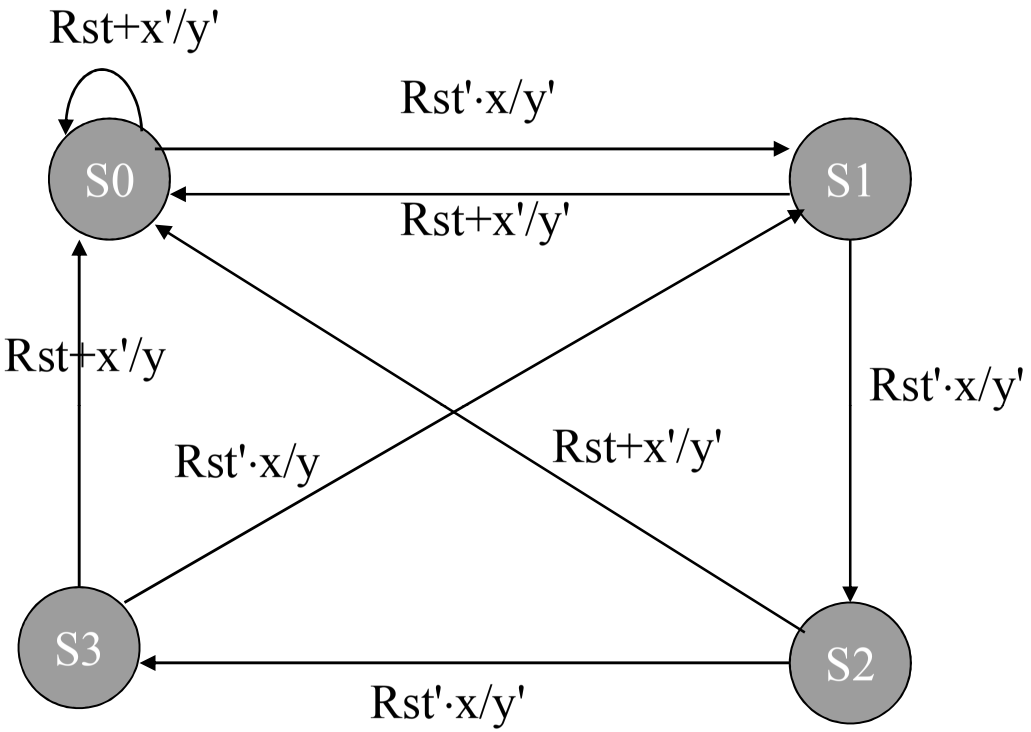
Μηχανή Καταστάσεων

Βήματα:

1. Κατασκευή διαγράμματος καταστάσεων.
2. Κατασκευή πίνακα καταστάσεων.
3. Κατασκευή λίστας μεταβάσεων (πιο απλή από το διάγραμμα για μεγάλα διαγράμματα). Οι μεταβάσεις αντί να σχεδιάζονται ορίζονται σε πίνακα.

Παράδειγμα: κύκλωμα ανιχνευτή ακολουθίας τριών άσσων

Μηχανή Καταστάσεων



Διάγραμμα Καταστάσεων

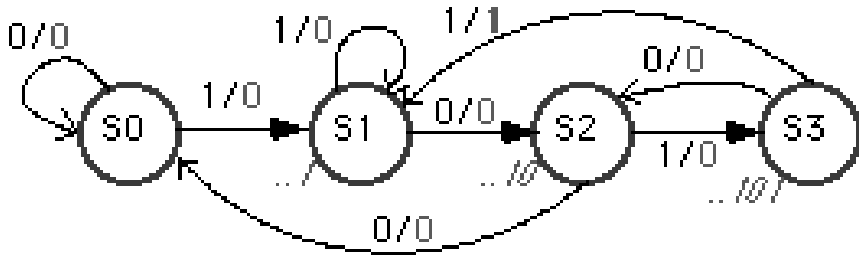
Παρούσα	**Επόμενη**		**y**
	X=0	X=1	
S0	S0	S1	0
S1	S0	S2	0
S2	S0	S3	0
S3	S0	S1	1

Πίνακας Καταστάσεων

Μηχανή Καταστάσεων

```
entity seq_detector is
generic (delay: time);
port ( clk, Rst, x: in std_logic; y: out std_logic );
end seq_detector;

architecture FSM of seq_detector is
type states is (S0, S1, S2, S3);
signal state: states:=S0;
begin
set_state: process (clk, Rst)
begin
 if (Rst='1') then state<=S0;
 elsif (clk'event and clk='1') then
   case state is
       S0 => if (x='1') then state<=S1 else state<=S0; end if;
       S1 => if (x='1') then state<=S2 else state<=S0; end if;
       S2 => if (x='1') then state<=S3 else state<=S0; end if;
       S3 => if (x='1') then state<=S1 else state<=S0; end if;
   end case;
end if;
      y <=  '1' when state=S3,
                else '0' when others;
end behave;
```

Μηχανή Καταστάσεων



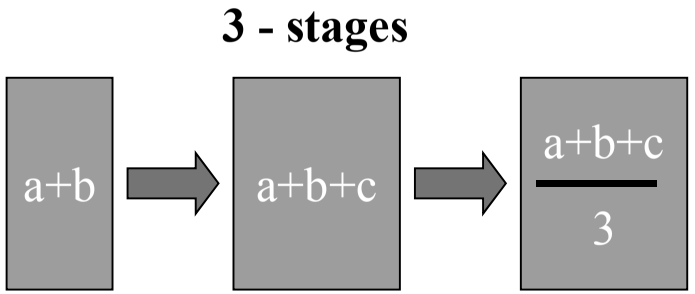
VHDL file for a sequence detector (1011) implemented as a Mealy Machine

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

entity myvhdl is
   port (CLK, RST, X: in STD_LOGIC;
         Z: out STD_LOGIC);
end;

architecture myvhdl_arch of myvhdl is
-- SYMBOLIC ENCODED state machine: Sreg0
type Sreg0_type is (S1, S2, S3, S4);
signal Sreg0: Sreg0_type;
begin
```

```
begin
--concurrent signal assignments
Sreg0_machine: process (CLK)
begin
if CLK'event and CLK = '1' then
   if RST='1' then
      Sreg0 <= S1;
   else
   case Sreg0 is
      when S1 =>
         if X='0' then
            Sreg0 <= S1;
         elsif X='1' then
            Sreg0 <= S2;
         end if;
      when S2 =>
         if X='1' then
            Sreg0 <= S2;
         elsif X='0' then
            Sreg0 <= S3;
         end if;
      when S3 =>
         if X='1' then
            Sreg0 <= S4;
         elsif X='0' then
            Sreg0 <= S1;
         end if;
      when S4 =>
         if X='0' then
            Sreg0 <= S3;
         elsif X='1' then
            Sreg0 <= S2;
         end if;
      when others =>
         null;
   end case;
   end if;
end if;
end process;
-- signal assignment statements for combinatorial outputs
Z_assignment:
Z <= '0' when (Sreg0 = S1 and X='0') else
      '0' when (Sreg0 = S1 and X='1') else
      '0' when (Sreg0 = S2 and X='1') else
      '0' when (Sreg0 = S2 and X='0') else
      '0' when (Sreg0 = S3 and X='1') else
      '0' when (Sreg0 = S3 and X='0') else
      '0' when (Sreg0 = S4 and X='0') else
      '1' when (Sreg0 = S4 and X='1') else
      '1';
end myvhdl_arch;
```

Παράδειγμα Δομή Διοχέτευσης

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


entity average_pipeline is
  port ( clk  : in  std_logic;
         a, b, c : in  sfixed(5 downto -8);
         avg     : out sfixed(5 downto -8) );
end entity average_pipeline;


architecture rtl of average_pipeline is
  signal a_plus_b, sum, sum_div_3 : sfixed(5 downto -8);
  signal saved_a_plus_b, saved_c, saved_sum : sfixed(5 downto -8);
begin
```



Αριθμοί σταθερής υποδιαστολής

Παράδειγμα Δομή Διοχέτευσης

```
a_plus_b <= a + b;
reg1 : process (clk) is
  begin
    if rising_edge(clk) then
      saved_a_plus_b <= a_plus_b;
      saved_c        <= c;
    end if;
  end process reg1;

sum <= saved_a_plus_b + saved_c;
reg2 : process (clk) is
  begin
    if rising_edge(clk) then
      saved_sum <= sum;
    end if;
  end process reg2;
```

```
sum_div_3 <= saved_sum * to_sfixed(1.0/3.0,
                 sum_div_3'left, sum_div_3'right);

reg3 : process (clk) is
begin
    if rising_edge(clk) then
      avg <= sum_div_3;
    end if;
end process reg3;

end architecture rtl;
```