Γλώσσες Περιγραφής Υλικού

Ανασκόπηση του μαθήματος

Νικόλαος Καββαδίας
nkavv@physics.auth.gr
nkavv@uop.gr

02 Ιουνίου 2009

Αντικείμενο και περίγραμμα του μαθήματος: Γλώσσες Περιγραφής Υλικού

- Αντικείμενο του μαθήματος
  - Εισαγωγή στην VHDL, βασικά δομικά στοιχεία της VHDL, σχεδιασμός κυκλωμάτων με την VHDL, VHDL για προχωρημένους
- Στόχοι του μαθήματος
  - Σχεδιασμός ψηφιακών κυκλωμάτων με τη γλώσσα περιγραφής υλικού VHDL
  - Παρουσίαση χαρακτηριστικών συνθέσιμων κυκλωμάτων
  - Εξάσκηση στην περιγραφή και προσομοίωση ψηφιακών κυκλωμάτων
- Τρόπος εξέτασης του μαθήματος
  - Γραπτές εξετάσεις: 60% του τελικού βαθμού
  - Εργασία: 40% του τελικού βαθμού
- Ενημέρωση για ανακοινώσεις, διαλέξεις, ύλη, εργασίες από τον ιστότοπο του μαθήματος: `http://eclass.uop.gr/courses/CST256/index.php`

Οργάνωση των παραδόσεων

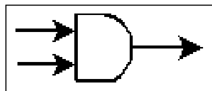
Διαλέξεις

1. Εισαγωγή στην VHDL
2. Δομές ακολουθιακού και συντρέχοντος κώδικα
3. Προχωρημένα στοιχεία της VHDL
4. Σύνταξη παραμετρικών περιγραφών
5. Σύνταξη κώδικα για λογική σύνθεση
6. Δομές ελέγχου/επαλήθευσης λειτουργίας των κυκλωμάτων
7. Μηχανές πεπερασμένων καταστάσεων
8. Υποδειγματική εργασία
9. Μη προγραμματιζόμενοι επεξεργαστές
10. Ανασκόπηση του μαθήματος

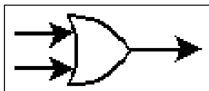
Εισαγωγικά

- Η VHDL αποτελεί μια γλώσσα για τη μοντελοποίηση της δομής και της συμπεριφοράς υλικού
- Επιτρέπει τη χρήση διαφορετικών μεθοδολογιών σχεδιασμού
- Προσφέρει ανεξαρτησία από την εκάστοτε τεχνολογία υλοποίησης (standard cell VLSI, FPGA)
- Διευκολύνει την επικοινωνία σχεδίων μεταξύ συνεργαζόμενων ομάδων σχεδιασμού
- Βοηθά στην καλύτερη διαχείριση του έργου του σχεδιασμού
- Στην VHDL μπορεί να περιγραφεί ένα μεγάλο εύρος ψηφιακών κυκλωμάτων
- Βασικά κριτήρια στην επιλογής HDL είναι: η διαθεσιμότητα εργαλείων ανάπτυξης, η δυνατότητα επαναχρησιμοποίησης κώδικα, και η οικειότητα με τις συντακτικές δομές της γλώσσας

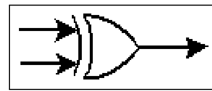
Προκαθορισμένες λογικές πύλες στη VHDL



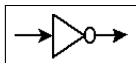
a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1



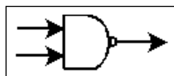
a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1



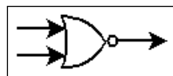
a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0



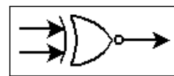
a	not a
0	1
1	0



a	b	a nand b
0	0	1
0	1	1
1	0	1
1	1	0

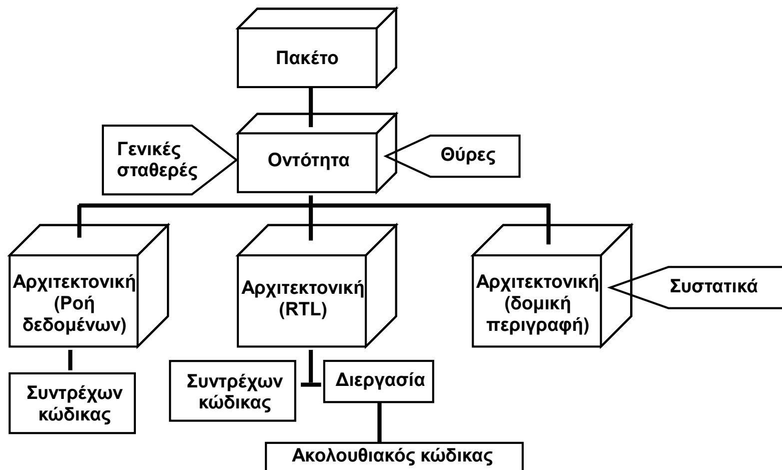


a	b	a nor b
0	0	1
0	1	0
1	0	0
1	1	0



a	b	a xnor b
0	0	1
0	1	0
1	0	0
1	1	1

Ιεραρχικός σχεδιασμός στην VHDL



ENTITY και ARCHITECTURE

- ENTITY: Η διεπαφή του κυκλώματος (θύρες εισόδου και εξόδου)
  - Για μια θύρα δηλώνονται: όνομα, κατευθυντικότητα, τύπος δεδομένων
  - Τύποι θυρών: *IN*, *OUT*, *INOUT*, *BUFFER*
- ARCHITECTURE: Καταγράφει τον τρόπο λειτουργίας του κυκλώματος

ENTITY

```
entity name-of-entity is
  generic (
    generic_list with initializations
  );
  port (
    port_list
  );
end [entity] name-of-entity;
```

ARCHITECTURE

```
architecture architecture-name is
  [architecture declarations]
begin
  [concurrent statements]
  [sequential statements]
  [structural code]
end [architecture] architecture-name;
```

Ο πλήρης αθροιστής δυαδικού ψηφίου σε VHDL

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

entity full_adder is
  port (
    a    : in  STD_LOGIC;
    b    : in  STD_LOGIC;
    cin  : in  STD_LOGIC;
    s    : out STD_LOGIC;
    cout : out STD_LOGIC
  );
end full_adder;

architecture structural of full_adder is
begin
  s    <= a xor b xor cin;
  cout <= (a and b) or (a and cin) or (b and cin);
end structural;
```

Συχνά χρησιμοποιούμενοι τύποι της VHDL

ΤΥΠΟΣ	ΤΙΜΗ	ΠΡΟΕΛΕΥΣΗ
std_ulogic	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	std_logic_1164
std_ulogic_vector	array of std_ulogic	std_logic_1164
std_logic	resolved std_ulogic	std_logic_1164
std_logic_vector	array of std_logic	std_logic_1164
unsigned	array of std_logic	numeric_std, std_logic_arith
signed	array of std_logic	numeric_std, std_logic_arith
boolean	true, false	standard
character	191 / 256 characters	standard
string	array of character	standard
integer	$-(2^{31}-1)$ to $2^{31}-1$	standard
real	$-1.0E38$ to $1.0E38$	standard
time	1 fs to 1 hr	standard

Τύποι δεδομένων στο package STANDARD

- BIT: `'0', '1'`
- BIT_VECTOR: `"001100", X"00FF"` στο δεκαεξαδικό
- BOOLEAN: `true, TRUE, TruE` για το αληθές και `False, false, FALsE` για το ψευδές
- CHARACTER: `'A', 'a', '@', '''`. Ένας πίνακας από `CHARACTER` αποτελεί συμβολοσειρά (string): `"hold time out of range", "i'll be back", "0$#1324"`
- REAL: `-1.0, +2.35, 36.6, -1.0E+38`
- INTEGER με εύρος τιμών {-2,147,483,647, +2,147,483,647}: +1, 862, -257, +15
- TIME: `10 ns, 100 us, 6.3 ns`

CONSTANT, VARIABLE και SIGNAL

- CONSTANT: αντικείμενο στο οποίο ανατίθεται μία αμετάβλητη τιμή

```
constant identifier : type-indication [:=expression];
constant PI : REAL := 3.147592;
constant FIVE : BIT_VECTOR := "0101";
```

- VARIABLE: αντικείμενο στο οποίο κάποια στιγμή ανατίθεται μία τιμή η οποία μπορεί να μεταβληθεί εντός μιας PROCESS

```
variable identifier : type-indication [constraint] [:=expression];
variable index: INTEGER range 1 to 50 := 50;
variable x, y : INTEGER;
variable memory : STD_LOGIC_VECTOR(0 to 7);
```

- SIGNAL: υλοποίηση διασυνδέσεων εντός ενός κυκλώματος αλλά και για την εξωτερική διασύνδεση διαφορετικών μονάδων σχεδιασμού

```
signal identifier : type-indication [constraint] [:=expression];
signal control: BIT := '0';
signal count: INTEGER range 0 to 100;
signal y: STD_LOGIC_VECTOR(7 downto 1);
```

Τελεστές της VHDL (VHDL operators)

Συνοπτικός πίνακας των τελεστών

λογικοί	and	or	nand	nor	xor	xnor	not
αριθμητικοί	+	-	*	/	mod	rem	
σύγκρισης	=	/=	<	<=	>	>=	
ολίσθησης	sll	srl	sla	sra	rol	ror	
μοναδιαίοι	+	-					
άλλοι	**	abs	&				

- Οι τελεστές αναγωγής σε δύναμη "**", απόλυτης τιμής "abs", και υπολογισμού ακέραιου υπολοίπου ("mod", "rem") δεν είναι συνθέσιμοι
- Οι τελεστές πολλαπλασιασμού και διαίρεσης υποστηρίζονται από ορισμένα εργαλεία λογικής σύνθεσης υπό προϋποθέσεις
- Η διαφορά των mod και rem είναι ότι: το `A rem B` παίρνει το πρόσημο του A ενώ το `A mod B` το πρόσημο του B

Αναθέσεις σε VARIABLE και SIGNAL

- Η ανάθεση σε `VARIABLE` αντικαθιστά την τρέχουσα τιμή της με μια νέα τιμή
- Τα `SIGNAL` προσφέρουν επικοινωνία μεταξύ διαφορετικών PROCESS και `COMPONENT` instances

```
ix := 'a';
y := "0000";
```

```
ix <= 'a';
y <= "0000";
```

Απόδοση τιμής	SIGNAL <=	VARIABLE :=
Χρησιμότητα	Αναπαριστά κυκλωματικές διασυνδέσεις	Αναπαριστά τοπική πληροφορία
Εμβέλεια	Μπορεί να είναι καθολική	Τοπική (διεργασία, συνάρτηση ή διαδικασία)
Συμπεριφορά	Η ενημέρωση δεν είναι άμεση σε ακολουθιακό κώδικα	Άμεση ενημέρωση
Χρήση	PACKAGE, ENTITY, ARCHITECTURE	PROCESS, FUNCTION, PROCEDURE

PROCESS

- Η PROCESS προσφέρει τη δυνατότητα σχεδιασμού ακολουθιακού κώδικα

```
[process_label:] process [(sensitivity list)]
  [declarations (subprogram, type, subtype, constant, variable, file,
  alias, attribute), attribute specification, use clause]
begin
  sequential_statements
end process [process_label];
```

- Η λίστα ευαισθησίας αποτελεί κατάλογο εισόδων και SIGNAL για μεταβολές των οποίων μία PROCESS υποχρεούται να αναμένει
- Παράδειγμα:

```
process (a, b)
begin
  if (a /= b) then
    cond <= '1';
  else
    cond <= '0';
  end if;
end process;
```

Ευαισθησία επιπέδου (level-sensitivity) και ακμοπυροδότηση (edge triggering)

- Οι μεταβολές των σημάτων που δηλώνονται σε μία λίστα ευαισθησίας και οι οποίες ενεργοποιούν τον υπολογισμό μεταβλητών και σημάτων σε μία PROCESS είναι δύο τύπων:
  - Μεταβολή επιπέδου (για σήματα επίτρεψης ή ενεργοποίησης και δεδομένα)
  - Ανερχόμενη ή κατερχόμενη ακμή (για σήματα ρολογιού)

- Μανδαλωτής (μεταβολή επιπέδου)

```
process (en, a)
begin
  if (en = '1') then
    temp <= a;
  end if;
end process;
```

- Συγχρονισμός ως προς ανερχόμενη ακμή

```
process (clk, a)
begin
  if (clk = '1' and clk'EVENT) then
    temp <= a;
  end if;
end process;
```

☞ Η έκφραση `clk'EVENT` είναι TRUE όταν έχει συμβεί μεταβολή ($0 \rightarrow 1$ ή $1 \rightarrow 0$) στο σήμα `clk`

ℹ Για ορισμένα εργαλεία σύνθεσης, τα σήματα εκτός του `clk` μπορούν να παραλειφθούν από μια λίστα ευαισθησίας

Δομές ελέγχου σε ακολουθιακό κώδικα

- Η εντολή IF αποτελεί τη θεμελιώδη δομή για την εκτέλεση κώδικα υπό συνθήκη

```
if ... then
  s1;
[elsif ... then
  s2;]
[else
  s3;]
end if;
```

- Η εντολή CASE χρησιμοποιείται για την περιγραφή δομών αποκωδικοποίησης

```
case expression is
  when value => s1; s2; ... sn;
  when value1 | value2 | ... | valuen
    => s1; s2; ... sn;
  when value1 to value2 => ...
  when others => ...
end case;
```

Δομές επανάληψης

- Η εντολή LOOP προσφέρει έναν βολικό τρόπο για την περιγραφή επαναληπτικών κυκλωματικών δομών

```
[loop_label:] [iteration_scheme] \textbf{loop}
  sequence_of_statements
end loop [loop_label];
```

- Το σχήμα επανάληψης (iteration scheme) μπορεί να είναι τύπου WHILE ή τύπου FOR:

```
while condition
for identifier in discrete_range
```

Παράδειγμα υπολογισμού τετραγώνων ακεραίων με FOR

```
FOR i IN 1 to 10 LOOP
  i_squared := i * i;
END LOOP;
```

Παράδειγμα υπολογισμού τετραγώνων ακεραίων με WHILE

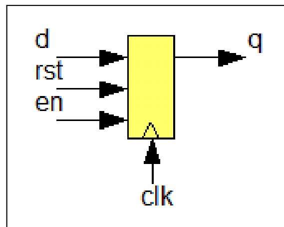
```
i := 1;
WHILE (i<11) LOOP
  i_squared := i * i;
  i := i + 1;
END LOOP;
```

Καταχωρητής με επίτρεψη φόρτωσης (load enable)

Με επίτρεψη φόρτωσης

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

entity dreg is
  port (
    clk, d, rst : in std_logic;
    en          : in std_logic;
    q           : out std_logic
  );
end dreg;

architecture rtl of dreg is
signal temp: std_logic;
begin
process (clk, rst, d)
  if (clk'event and clk = '1') then
    if (rst = '1') then
      temp <= '0';
    else
      if (en = '1') then
        temp <= d;
      end if;
    end if;
  end if;
end process;
q <= temp;
end rtl;
```

Σχηματικό διάγραμμα



Δομές συντρέχοντος κώδικα

- Η εντολή WHEN/ELSE αποτελεί μία συντρέχουσα εντολή η οποία έχει ένα στόχο (target) επιλέγοντας από περισσότερες από μία εκφράσεις

```
target <= {expression when condition else} expression;
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN (ctl='1') ELSE
        "010";
```

- Η εντολή WITH/SELECT προσφέρει τη δυνατότητα επιλεκτικής ανάθεσης σε ένα στόχο (target) επιλέγοντας από περισσότερες από μία εκφράσεις

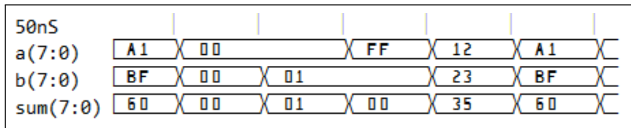
```
WITH expression SELECT
  target <= {expression WHEN choices,} expression;
WITH control SELECT
  output <= reset WHEN "000",
            set WHEN "111",
            UNAFFECTED WHEN others;
```

Αθροιστές απρόσημων και προσημασμένων (2's complement) ακεραίων (1)

- Περιγραφή για απρόσημους αριθμούς

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

entity adder is
  port (
    a,b : in std_logic_vector(7 downto 0);
    sum : out std_logic_vector(7 downto 0)
  );
end adder;

architecture rtl of adder is
  signal temp : std_logic_vector(8 downto 0);
begin
  temp <= ('0' & a)  + ('0' & b);
  sum <= temp(7 downto 0);
end rtl;
```

- Διάγραμμα χρονισμού

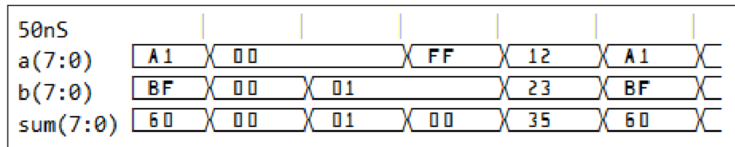


Αθροιστές απρόσημων και προσημασμένων (2's complement) ακεραίων (2)

- Για την άθροιση προσημασμένων (συμπλήρωμα-ως-προς-2) απαιτείται η επέκταση προσήμου (sign extension) του ενδιάμεσου αποτελέσματος
- Περιγραφή για προσημασμένους αριθμούς

```
...
  signal temp : std_logic_vector(8 downto 0);
begin
  temp <= (a(7) & a) + (b(7) & b);
  sum <= temp(7 downto 0);
end rtl;
```

- Διάγραμμα χρονισμού



Απαριθμητοί τύποι δεδομένων (enumerated data types)

- Ο χρήστης ορίζει τη λίστα των επιτρεπόμενων τιμών που μπορούν να ανατεθούν σε ένα αντικείμενο που δηλώνεται με τον συγκεκριμένο τύπο
- Απαριθμητός τύπος δεδομένων χαρακτηριστικός για μηχανές πεπερασμένων καταστάσεων (FSM)

```
TYPE fsm_state is (idle, forward, backward, stop);
...
signal current_state := IDLE;
```

- Απαριθμητός τύπος δεδομένων που καταγράφει τα επιτρεπόμενα χρώματα στο πρότυπο TELETEXT

```
TYPE rgb3 is (black, blue, green, cyan, red, magenta, yellow, white);
```

- Τύπος δεδομένων για την υλοποίηση λογικής 4 επιπέδων κατά Verilog

```
TYPE verilog_mvl4 is ('0', '1', 'X', 'Z');
```

Σύνθετοι τύποι: Πίνακες

- Πίνακας: συλλογή από αντικείμενα του ίδιου τύπου
- Δήλωση για τον ορισμό ενός νέου τύπου πίνακα και δήλωση SIGNAL αυτού του τύπου:

```
TYPE <array name> IS ARRAY specification OF <data type>;
SIGNAL <signal name>: <array type> [:= <initial value];
```

- Παραδείγματα

```
TYPE image is ARRAY (0 to 31) of byte; -- 1Dx1D
TYPE matrix2D is ARRAY (0 to 3, 1 downto 0) OF STD_LOGIC;
...
SIGNAL x: image; -- an 1Dx1D signal
SIGNAL y: matrix2D;
-- Initialization
... := "0001"; -- 1D
... := ('0','0','0','1'); -- 1D
... := (('0','1','1','1'),('1','1','1','0')); -- 1Dx1D or 2D
-- Assignments
x(0) <= y(1)(2); -- 1Dx1D
x(0) <= v(1,2); -- 2D
x <= y(0); -- entire row
x(3 downto 1) <= y(1)(4 downto 2);
x(3 downto 0) <= (3 => '1', 2 => '0', others => '0');
```

Αναπαράσταση ακεραίων με διανύσματα

Δυαδικό και δεκαεξαδικό

Ακέραιος	Δυαδικό	hex
0	"00000"	X"0"
1	"00001"	X"1"
2	"00010"	X"2"
3	"00011"	X"3"
4	"00100"	X"4"
5	"00101"	X"5"
6	"00110"	X"6"
7	"00111"	X"7"
8	"01000"	X"8"
9	"01001"	X"9"
10	"01010"	X"A"
11	"01011"	X"B"
12	"01100"	X"C"
13	"01101"	X"D"
14	"01110"	X"E"
15	"01111"	X"F"
16	"10000"	X"10"
17	"10001"	X"11"

Συμπλήρωμα ως προς 2

Ακέραιος	Δυαδικό	hex
-8	"1000"	X"8"
-7	"1001"	X"9"
-6	"1010"	X"A"
-5	"1011"	X"B"
-4	"1100"	X"C"
-3	"1101"	X"D"
-2	"1110"	X"E"
-1	"1111"	X"F"
0	"0000"	X"0"
1	"0001"	X"1"
2	"0010"	X"2"
3	"0011"	X"3"
4	"0100"	X"4"
5	"0101"	X"5"
6	"0110"	X"6"
7	"0111"	X"7"

Υπερφόρτωση τελεστή (operator overloading)

- Στη VHDL μπορούν να δηλωθούν (π.χ. σε πακέτα) εκ νέου τελεστές από τον χρήστη οι οποίοι να έχουν τα ίδια ονόματα με προκαθορισμένους τελεστές, αλλά να δρουν σε διαφορετικούς τύπους δεδομένων
- Η τεχνική αυτή ονομάζεται *υπερφόρτωση τελεστή*
- Υπερφόρτωση του τελεστή '+' για την πρόσθεση ενός INTEGER με ένα BIT

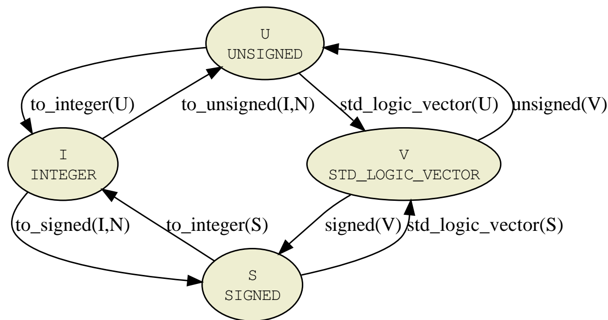
```
FUNCTION "+" (a:INTEGER, b:BIT) RETURN INTEGER IS
BEGIN
  IF (b='1') THEN
    RETURN a+1;
  ELSE
    RETURN a;
  END IF;
END "+";
...
SIGNAL inp1,outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
...
outp <= inp1 + inp2;
```

Σύνοψη των συναρτήσεων μετατροπής τύπου του πακέτου numeric_std

- Δήλωση για τη χρήση του πακέτου

```
LIBRARY ieee;
USE ieee.numeric_std.all;
```

- Γραφική αναπαράσταση των επιτρεπόμενων μετατροπών



Συναρτήσεις μετατροπής του πακέτου std_logic_arith

- Στο std_logic_arith μπορούμε να βρούμε τις συναρτήσεις μετατροπής

  conv_integer, conv_unsigned, conv_signed, conv_std_logic_vector

Κλήση συνάρτησης	Περιγραφή
conv_integer(param)	Μετατρέπει μια παράμετρο *param* τύπου INTEGER, UNSIGNED, SIGNED ή STD_ULOGIC σε μια τιμή τύπου INTEGER
conv_unsigned(param,b)	Μετατρέπει μια παράμετρο *param* τύπου INTEGER, UNSIGNED, SIGNED ή STD_ULOGIC σε μια τιμή τύπου UNSIGNED με μέγεθος *b* bit
conv_signed(param,b)	Μετατρέπει μια παράμετρο *param* τύπου INTEGER, UNSIGNED, SIGNED ή STD_ULOGIC σε μια τιμή τύπου SIGNED με μέγεθος *b* bit
conv_std_logic_vector(param,b)	Μετατρέπει μια παράμετρο *param* τύπου INTEGER, UNSIGNED, SIGNED ή STD_ULOGIC σε μια τιμή τύπου STD_LOGIC_VECTOR με μέγεθος *b* bit

Το πακέτο std_logic_unsigned ορίζει υπερφορτωμένες εκδοχές των πρώτων τριών συναρτήσεων μετατροπής για δεδομένα τύπου STD_LOGIC_VECTOR

Παραδείγματα μετατροπών με βάση το πακέτο std_logic_arith)

- Μετατροπή UNSIGNED, SIGNED ⇒ INTEGER

```
Unsigned_int <= CONV_INTEGER( A_uv );
Signed_int   <= CONV_INTEGER( B_sv );
```

- Μετατροπή INTEGER ⇒ UNSIGNED, SIGNED

```
A_uv  <= CONV_UNSIGNED(Unsigned_int,8);
B_sv  <= CONV_SIGNED (Signed_int,8);
```

- Μετατροπή UNSIGNED,SIGNED ⇒ STD_LOGIC_VECTOR

```
C_slv  <= CONV_STD_LOGIC_VECTOR(CONV_INTEGER(A_uv),8);
D_slv  <= CONV_STD_LOGIC_VECTOR(CONV_INTEGER(B_sv),8);
```

- Παράδειγμα χρήσης: Διευθυνσιοδότηση σε μνήμη ROM

```
Data_slv  <= ROM( CONV_INTEGER(Addr_uv) ); -- or
Data_slv  <= ROM( CONV_INTEGER(Addr_slv) );
```

Σήματα για τα παραδείγματα:

```
signal A_uv : unsigned (7 downto 0) ;
signal Unsigned_int : integer range 0 to 255 ;
signal C_slv, D_slv : std_logic_vector( 7 downto 0) ;
signal B_sv : signed (7 downto 0) ;
signal Signed_int : integer range -128 to 127;
```

Αναλυτικός πίνακας προκαθορισμένων ιδιοτήτων

- Οι προκαθορισμένες ιδιότητες προσφέρουν τη δυνατότητα περιγραφής γενικών τμημάτων κώδικα για οποιοδήποτε μέγεθος διανύσματος ή πίνακα

Ιδιότητα	Επιστρεφόμενη τιμή
Σε συνηθισμένα δεδομένα (όχι απαριθμητά)	
`d'LOW`	Ο χαμηλότερος δείκτης του πίνακα
`d'HIGH`	Ο υψηλότερος δείκτης του πίνακα
`d'LEFT`	Ο αριστερός δείκτης του πίνακα
`d'RIGHT`	Ο δεξιός δείκτης του πίνακα
`d'LENGTH`	Το μέγεθος του διανύσματος
`d'RANGE`	Το εύρος του διανύσματος
`d'REVERSE_RANGE`	Το αντίστροφο εύρος του διανύσματος
Σε απαριθμητά δεδομένα	
`d'VAL(pos)`	Η τιμή στην καθοριζόμενη θέση
`d'POS(value)`	Η θέση (διεύθυνση) της καθοριζόμενης τιμής
`d'LEFTOF(value)`	Η τιμή στη θέση στα αριστερά της καθοριζόμενης τιμής
`d'VAL(row,column)`	Η τιμή στην καθοριζόμενη θέση σε ένα διδιάστατο πίνακα
Σε αντικείμενα τύπου SIGNAL	
`s'EVENT`	Αληθές όταν προκύπτει συμβάν στο `s`
`s'STABLE`	Αληθές όταν δεν προκύπτει συμβάν στο `s`
`s'ACTIVE`	Αληθές αν το `s` είναι σε υψηλή στάθμη

Συστατικό στοιχείο (COMPONENT) - Στιγμιότυπο ενός COMPONENT

- Το COMPONENT αποτελεί δήλωση για τη χρησιμοποίηση ενός κυκλώματος ως υπομονάδα

```
COMPONENT <component name> IS
  [GENERIC (
    <generic name> : <type> [:= <initialization>];
    ...
  )]
  PORT (
    <port name> : [direction] <signal type>;
    ...
  );
END COMPONENT;
```

- Το στιγμιότυπο (instance) ενός COMPONENT αποτελεί ένα αντίτυπό του που χρησιμοποιείται στα πλαίσια της δομικής περιγραφής ενός κυκλώματος

```
<label>: <component name>
  [GENERIC MAP (
    [<generic name> =>] <expression>,
    ... );
  PORT MAP (
    [<port name> =>] <expression>,
    ... );
```

PACKAGE

- Το πακέτο (PACKAGE) αποτελεί μέσο για την οργάνωση τμημάτων κώδικα

```
PACKAGE <package name> IS
  statements
END <package name>;
[PACKAGE BODY <package name> IS
  implementation of functions and procedures
END <package name>;]
```

- Δήλωση χρήσης ενός πακέτου

```
USE <library name>.<package name>.<package parts>;
```

- Παράδειγμα

```
PACKAGE my_package IS
  TYPE state IS (st1, st2, st3, st4);
  FUNCTION positive_edge(SIGNAL s: BIT) RETURN BOOLEAN;
END my_package;
PACKAGE BODY my_package IS
  FUNCTION positive_edge(SIGNAL s: BIT) RETURN BOOLEAN IS
  BEGIN
    RETURN (s'EVENT and s='1');
  END positive_edge;
END my_package;
```

FUNCTION

- Οι συναρτήσεις αποτελούν είδος υποπρογράμματος το οποίο επιστρέφει μία μοναδική τιμή
- Δεν μπορούν να τροποποιήσουν τις παραμέτρους εισόδου τους

```
FUNCTION <function name> [<parameter list>] RETURN <type> IS
  [statements]
BEGIN
  sequential statements
END <function name>;
```

- Παράδειγμα: Συνάρτηση $\lceil log_2 \rceil$

```
function LOG2C(input: INTEGER) return INTEGER is
  variable temp,log: INTEGER;
  begin
    log := 0; temp := 1;
    for i in 0 to input loop
      if temp < input then
        log := log + 1; temp := temp * 2;
      end if;
    end loop;
    return (log);
end function LOG2C;
```

PROCEDURE

- Μπορούν να τροποποιούν τις τιμές των παραμέτρων τους
- Δεν περιλαμβάνουν εντολή RETURN
- Παράμετροι (CONSTANT, SIGNAL ή VARIABLE) με κατευθυντικότητα (IN, OUT, INOUT)

```
PROCEDURE <procedure name> [<parameter list>] IS
  [statements]
BEGIN
  sequential statements
END <procedure name>;
```

- Παράδειγμα: Διαδικασία sort

```
PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 to limit;
  SIGNAL min, max: OUT INTEGER RANGE 0 to limit) IS
BEGIN
  IF (in1 >= in2) THEN
    max <= in1;  min <= in2;
  ELSE
    max <= in2; min <= in1;
  END IF;
END sort;
...
sort (inp1, inp2, outp1, outp2); -- usage
```

GENERATE

- Η εντολή GENERATE παρέχει τη δυνατότητα περιγραφής επαναλαμβανόμενων κυκλωματικών δομών με συμπαγή τρόπο σε συντρέχοντα κώδικα
- Διευκολύνει την περιγραφή δομών που παρουσιάζουν κανονικότητα
- Ειδικότερα, μια εντολή GENERATE μπορεί να περικλείει στιγμιότυπα συστατικών ή άλλες GENERATE
- Σχήματα FOR και IF

```
<label> : (FOR|IF) parameter_specification GENERATE
  [declaration_statements]
BEGIN
  {concurrent_statements}
END GENERATE <label>;
```

Παραμετρικός αθροιστής ριπής κρατουμένου (parameterized ripple-carry adder)

- Η γενική σταθερή N καθορίζει το εύρος bit του αθροιστή
- Για κάθε m στην εντολή GENERATE δημιουργείται ένας πλήρης αθροιστής με διασύνδεση του κρατουμένου εξόδου του στο κρατούμενο εισόδου της επόμενης βαθμίδας

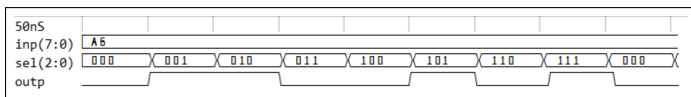
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rca is
  generic (N: integer := 8);
  port (
    a, b : in unsigned(N-1 downto 0);
    cin : in std_logic;
    sum : out unsigned(N-1 downto 0);
    cout : out std_logic
  );
end rca;
```

```
architecture gatelevel of rca is
  signal c : unsigned(N downto 0);
begin
  c(0) <= cin;
  G1: for m in 0 to N-1 generate
    sum(m) <= a(m) xor b(m) xor c(m);
    c(m+1) <= (a(m) and b(m)) or
              (b(m) and c(m)) or
              (a(m) and c(m));
  end generate G1;
  cout <= c(N);
end gatelevel;
```

Περιγραφή γενικευμένου πολυπλέκτη

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.various_pkg.all; -- the package where "log2c(<integer>) lives

entity muxnto1 is
  generic (N : integer := 8);
  port (
    inp   : in std_logic_vector(N-1 downto 0);
    sel   : in std_logic_vector(log2c(N)-1 downto 0);
    outp  : out std_logic
  );
end muxnto1;

architecture rtl of muxnto1 is
begin
  outp <= inp(to_integer(unsigned(sel)));
end rtl;
```

Διάγραμμα χρονισμού του κυκλώματος



Κανόνες για τη σύνταξη συνθέσιμων περιγραφών (1)

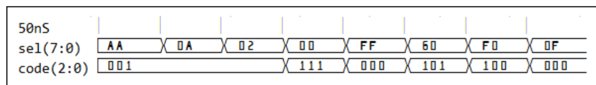
1. Χρήση ενός σήματος ρολογιού
2. Αποθήκευση τιμών στο κύκλωμα σε καταχωρητές ή μνήμες
3. Ανάθεση μιας τιμής ανά σήμα σε κάθε κύκλο ρολογιού
4. Χρησιμοποίηση μόνο σύγχρονης επανατοποθέτησης (synchronous reset)
5. Χρήση μόνο ακμοπυροδότησης στα flip-flop
6. Να μην παράγονται νέα σήματα χρονισμού με βάση το εξωτερικό ρολόι, αλλά αντί αυτού να χρησιμοποιούνται σήματα επίτρεψης/φόρτωσης για την επιλεκτική ενεργοποίηση κάποιας υπομονάδας

Κανόνες για τη σύνταξη συνθέσιμων περιγραφών (2)

7. Σε μια λίστα ευαισθησίας αναφέρονται όλα τα σήματα ή είσοδοι οι οποίες 'διαβάζονται' μέσα στη διεργασία
8. Σε μια λίστα ευαισθησίας μιας αποκλειστικά σύγχρονης διεργασίας επιτρέπεται να περιληφθεί μόνο το σήμα ρολογιού (`clk`)
9. Για την τρέχουσα και επόμενη κατάσταση ενός FSM, να χρησιμοποιείται απαριθμητός τύπος δεδομένων
10. Σε ένα κύκλωμα θα πρέπει να γίνεται ανάθεση σε όλα τα σήματα εξόδου για όλες τις περιπτώσεις λειτουργίας για την αποφυγή δημιουργίας ανεπιθύμητων μανδαλωτών
11. Επιτρέπεται η αρχική ανάθεση σε σήμα για την κάλυψη όλων των πιθανών περιπτώσεων
12. Να μην χρησιμοποιούνται οι τιμές `'X'` και `'Z'` ενός σήματος για τον έλεγχο περιπτώσεων (δήλωση WHEN σε μια CASE)

Κωδικοποιητής προτεραιότητας (priority encoder)

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

entity priority is
  port (
    sel  : in  std_logic_vector(7 downto 0);
    code : out unsigned(2 downto 0)
  );
end priority;

architecture imp of priority is
begin
  code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          "110" when sel(6) = '1' else "111";
end imp;
```



Βασικά στοιχεία στο σχεδιασμό κυκλωμάτων μνήμης

- Τρόποι ανάγνωσης
  - Ασύγχρονη ανάγνωση: αποτελέσματα διαθέσιμα στον ίδιο κύκλο στον οποίο διευθυνσιοδοτήθηκαν με κάποια συνδυαστική χρονική καθυστέρηση
  - Σύγχρονη ανάγνωση: αποτελέσματα διαθέσιμα στον επόμενο κύκλο ρολογιού
- Σήματα επίτρεψης
  - RAM: Επίτρεψη ανάγνωσης (read enable ή `re`)
  - RAM: Επίτρεψη εγγραφής (write enable ή `we`)
  - RAM,ROM: Επίτρεψη εξόδου (output enable ή `oe`)
- Παράμετροι
  - Αριθμός θέσεων (καταχωρήσεων): `N` ή `NR`
  - Εύρος λέξης διεύθυνσης (address width): `AW`
  - Εύρος λέξης δεδομένων (data width): `DW`
  - ℹ Η παράμετρος AW ορισμένες φορές υπολογίζεται από την `NR` μέσω της έκφρασης: $AW = \lceil log_2(NR) \rceil$
  - Αριθμός θυρών εισόδου (`NWP`) και εξόδου (`NRP`)

Σύγχρονη μνήμη ROM των 8-bit με 16 θέσεις και χρήση CONSTANT

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

entity rom_16_8 is
  port (
    clk, re : in std_logic;
    addr    : in std_logic_vector(3 downto 0);
    data    : out std_logic_vector(7 downto 0)
  );
end rom_16_8;

architecture impl of rom_16_8 is
  type rom_type is array (0 to 15) of std_logic_vector(7 downto 0);
  constant ROM : rom_type :=
  (X"01", X"02", X"04", X"08", X"10", X"20", X"40", X"80",
   X"01", X"03", X"07", X"0F", X"1F", X"3F", X"7F", X"FF");
begin
  process (clk)
  begin
    if (clk='1' and clk'EVENT) then
      if (re = '1') then
        data <= ROM(conv_integer(addr));
      end if;
    end if;
  end process;
end impl;
```

Μνήμη τυχαίας προσπέλασης (RAM)

- Μία RAM διαθέτει τουλάχιστον μία είσοδο για τη διευθυνσιοδότηση (`address`) και τουλάχιστον μία θύρα για την ανάγνωση ή/και εγγραφή δεδομένων από και προς συγκεκριμένη θέση στη μνήμη θέση στη μνήμη
- Υποχρεωτικά διαθέτει είσοδο ρολογιού (`clk`) και επίτρεψη εγγραφής (`we`) για κάθε θύρα εγγραφής
- Τα περιεχόμενα της RAM υλοποιούνται ως SIGNAL
- Μπορεί να οριστεί και επίτρεψη ανάγνωσης θύρας εξόδου
- Οι πολλαπλές αιτήσεις για εγγραφή στην ίδια θέση δημιουργούν πρόβλημα διαμάχης και επιλύονται με κατάλληλη λογική ελέγχου (προτεραιότητα)
- Τρόπος εγγραφής READ FIRST: Τα περιεχόμενα της διευθυνσιοδοτούμενης θέσης μνήμης εμφανίζονται στην έξοδο. Τα δεδομένα εισόδου γράφονται στην ίδια θέση (ανάγνωση πριν την εγγραφή)

RAM με ασύγχρονη ανάγνωση

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

entity ram_async is
  port (
    clk, we : in std_logic;
    rwaddr : in std_logic_vector(5 downto 0);
    di : in std_logic_vector(15 downto 0);
    do : out std_logic_vector(15 downto 0)
  );
end ram_async;

architecture synth of ram_async is
  type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
  signal RAM: ram_type;
begin
  process (clk)
  begin
    if (clk='1' and clk'EVENT) then
      if (we = '1') then
        RAM(conv_integer(rwaddr)) <= di;
      end if;
    end if;
  end process;
  do <= RAM(conv_integer(rwaddr));
end synth;
```

RAM με τρόπο εγγραφής READ FIRST

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

entity ram_rf is
  port (
    clk, we : in std_logic;
    rwaddr : in std_logic_vector(5 downto 0);
    di : in std_logic_vector(15 downto 0);
    do : out std_logic_vector(15 downto 0)
  );
end ram_rf;

architecture synth of ram_rf is
  type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
  signal RAM: ram_type;
begin
  process (clk)
  begin
    if (clk='1' and clk'EVENT) then
      if (we = '1') then
        RAM(conv_integer(rwaddr)) <= di;
      end if;
      do <= RAM(conv_integer(rwaddr));
    end if;
  end process;
end synth;
```

Αρχείο καταχωρητών (register file) με NWP θύρες εισόδου και NRP θύρα εξόδου (1)

ℹ Το αρχείο καταχωρητών ελέγχθηκε για: $NRP = 2$, $NWP = 1$, $DW = 8$, και $AW = 4$

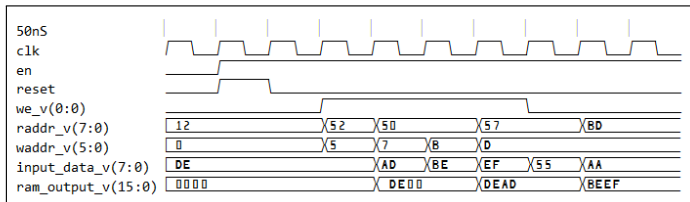
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity regfile is
  generic (
    NWP : integer :=  1;
    NRP : integer :=  2;
    AW  : integer :=  8;
    DW  : integer := 32
  );
  port (
    clock         : in  std_logic;
    reset         : in  std_logic;
    en            : in  std_logic;
    we_v          : in  std_logic_vector(NWP-1 downto 0);
    waddr_v       : in  std_logic_vector(NWP*AW-1 downto 0);
    raddr_v       : in  std_logic_vector(NRP*AW-1 downto 0);
    input_data_v  : in  std_logic_vector(NWP*DW-1 downto 0);
    ram_output_v  : out std_logic_vector(NRP*DW-1 downto 0)
  );
end regfile;
```

Αρχείο καταχωρητών (register file) με *NWP* θύρες εισόδου και *NRP* θύρα εξόδου (2)

```
architecture synth of regfile is
  type mem_type is array ((2**AW-1) downto 0) of
    std_logic_vector(DW-1 downto 0);
  signal ram_name : mem_type := (others => (others => '0'));
begin
  process (clock)
  begin
    if (clock'EVENT and clock = '1') then
      if (en = '1') then
        for i in 0 to NWP-1 loop
          if ((we_v(i) = '1')) then
            ram_name(conv_integer(waddr_v(AW*(i+1)-1 downto AW*i))) <=
            input_data_v(DW*(i+1)-1 downto DW*i);
          end if;
        end loop;
      end if;
    end if;
  end process;
  G_DO_NRP: for i in 0 to NRP-1 generate
   ram_output_v(DW*(i+1)-1 downto DW*i) <=
   ram_name(conv_integer(raddr_v(AW*(i+1)-1 downto AW*i)));
  end generate G_DO_NRP;
end synth;
```

Αρχείο καταχωρητών (register file) με *NWP* θύρες εισόδου και *NRP* θύρα εξόδου (3)

- Σε μοντέρνα FPGA με άφθονους πόρους ενσωματωμένης μνήμης (block RAM) η υλοποίηση του γενικευμένου αρχείου καταχωρητών με *NRP* θύρες ανάγνωσης και *NWP* θύρες εγγραφής απαιτεί τη χρήση $NWP \times NRP$ block RAM
- Διάγραμμα χρονισμού του κυκλώματος



ASSERT

- Η ASSERT είναι μία μη συνθέσιμη εντολή που χρησιμοποιείται για την επιστροφή μηνυμάτων στο τερματικό κατά την προσομοίωση
  1. Τμήμα συνθήκης (condition)
  2. Τμήμα αναφοράς μηνύματος που προσδιορίζεται από τη λέξη κλειδί REPORT
  3. Τμήμα σοβαρότητας στο οποίο γίνεται δήλωση της επίδρασης που έχει η μη ικανοποίηση της συνθήκης στη συνέχεια της προσομοίωσης. Σημειώνεται με τη λέξη κλειδί SEVERITY

```
ASSERT <condition>
  [REPORT "<message>"]
  [SEVERITY <severity level>];
END <package name>;
```

- Τα επίπεδα σοβαρότητας είναι: σημείωση (NOTE), προειδοποίηση (WARNING), σφάλμα (ERROR), ή αποτυχία (FAILURE)

Αρχεία στη VHDL

- Ο τύπος ΑΡΧΕΙΟΥ (FILE) προσφέρει ένα βολικό τρόπο για την επικοινωνία μιας περιγραφής VHDL με το περιβάλλον του μηχανήματος-ξενιστή (ο υπολογιστής στον οποίο γίνεται η ανάπτυξη και ο έλεγχος λειτουργίας της περιγραφής)

```
type LINE is access STRING; -- A LINE is a pointer to a STRING value.
type TEXT is file of STRING;
```

- Διαδικασίες για το χειρισμό αρχείων κειμένου (TEXTIO)

```
procedure FILE_OPEN (file F: TEXT; External_Name; in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);
procedure FILE_OPEN (Status: out FILE_OPEN_STATUS; file F: TEXT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);
procedure FILE_CLOSE (file F: TEXT);
function ENDFILE (file F: TEXT) return BOOLEAN;
procedure READLINE (file F: TEXT; L: inout LINE);
procedure WRITELINE (file F: TEXT; L: inout LINE);
procedure READ (file F: TEXT; VALUE: out STRING);
procedure WRITE (file F: TEXT; VALUE: in STRING);
procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
procedure HWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                 JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
```

Testbench

- Το testbench αποτελεί ένα εικονικό κύκλωμα το οποίο εφαρμόζει εισόδους προς (διέγερση) και λαμβάνει εξόδους (απόκριση) από το πραγματικό κύκλωμα
- Η entity ενός testbench δεν περιλαμβάνει καμία δήλωση θύρας, μπορεί όμως να περιλαμβάνει generic
- Στο testbench, δηλώνεται το COMPONENT του συνολικού κυκλώματος

```
ENTITY testbench IS
-- no PORT statement necessary
END testbench;

ARCHITECTURE example IS testbench
  COMPONENT entity_under_test
    PORT(...)
  END COMPONENT;
BEGIN
  Generate_waveforms_for_test;
  Instantiate_component;
  Monitoring_statements;
END example;
```

Διέγερση σημάτων εισόδου από process

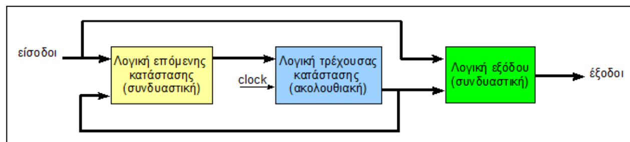
```
...
  CLK_GEN_PROC: process(clk)
  begin
    if (clk = 'U') then
      clk <= '1';
    else
      clk <= not clk after CLK_PERIOD/2;
    end if;
  end process CLK_GEN_PROC;

  DATA_INPUT: process
    variable ix : integer range 0 to 7;
  begin
    in1 <= X"DE"; in2 <= X"AD"; in3 <= X"BE"; in4 <= X"EF";
    sel <= "000"; reset <= '1';
    wait for CLK_PERIOD;
    --
    reset <= '0';
    for i in 0 to 7 loop
      sel <= std_logic_vector(to_unsigned(i,3));
      wait for CLK_PERIOD;
    end loop;
  end process DATA_INPUT;
...
```

Παράδειγμα: Εγγραφή αποτελεσμάτων σε αρχείο εξόδου

```
  component add ...
  signal a, b, sum : std_logic_vector(Dw-1 downto 0);
  file output_log : text open write_mode is "add.log";
begin
  UUT : add
    generic map (Dw => Dw)
    port map (a => a, b => b, sum => sum);

  process
  begin
    a <= X"FF"; b <= X"10"; wait for 10 ns;
    a <= X"10"; b <= X"89"; wait for 10 ns;
  end process;

  output_log_proc: process
    variable out_line : line;
  begin
    write(out_line, NOW, left, 8);
    write(out_line, string'(" a:"), right, 4);
    hwrite(out_line, a, right, 4);
    write(out_line, string'(" b:"), right, 4);
    hwrite(out_line, b, right, 4);
    write(out_line, string'(" sum:"), right, 4);
    hwrite(out_line, sum, right, 4);
    writeline(output_log, out_line);
    wait for 10 ns;
  end process output_log_proc;
...
```

Δομή ενός FSM

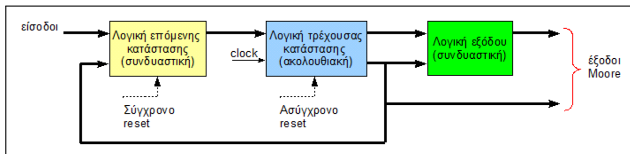
Τυπική οργάνωση ενός FSM



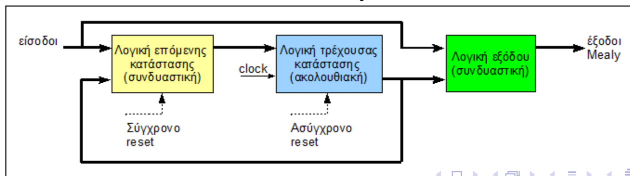
1. Λογική τρέχουσας κατάστασης: Υλοποιείται από καταχωρητή για την αποθήκευση της τρέχουσας κατάστασης του FSM. Η τιμή του αντιπροσωπεύει το συγκεκριμένο στάδιο στο οποίο βρίσκεται η λειτουργία του FSM
2. Λογική επόμενης κατάστασης: Συνδυαστική λογική η οποία παράγει την επόμενη κατάσταση της ακολουθίας. Η επόμενη κατάσταση αποτελεί συνάρτηση των εισόδων του FSM και της τρέχουσας κατάστασης
3. Λογική εξόδου: Συνδυαστική λογική που χρησιμοποιείται για την παραγωγή των σημάτων εξόδου του κυκλώματος. Οι έξοδοι αποτελούν συνάρτηση της εξόδου του καταχωρητή (τρέχουσας) κατάστασης και ΠΙΘΑΝΩΣ των εισόδων του FSM

Κατηγορίες FSM: τύπου Moore και τύπου Mealy

- Στα FSM τύπου Moore οι έξοδοι είναι συνάρτηση μόνο της τρέχουσας κατάστασης
- Οργάνωση ενός FSM τύπου Moore



- Στα FSM τύπου Mealy οι έξοδοι είναι συνάρτηση των εισόδων και της τρέχουσας κατάστασης
- Οργάνωση ενός FSM τύπου Mealy



Κωδικοποίηση της κατάστασης στα FSM

- sequential: σε κάθε κατάσταση ανατίθενται δυαδικοί αριθμοί κατά αύξουσα σειρά
- one-hot: σε κάθε κατάσταση αντιστοιχίζεται ξεχωριστό flip-flop. Σε κάθε κατάσταση ένα μόνο flip-flop έχει την τιμή '1'
- Κωδικοποίηση καθοριζόμενη από το χρήστη

```
constant S1: std_logic_vector(3 downto 0) := "0110";
constant S2: std_logic_vector(3 downto 0) := "0111";
constant S3: std_logic_vector(3 downto 0) := "0000";
```

- Κωδικοποίηση καθοριζόμενη από το εργαλείο υλοποίησης (λογικής σύνθεσης)

```
type STATES is (S1, S2, S3, S4);
signal STATE : STATES;
```

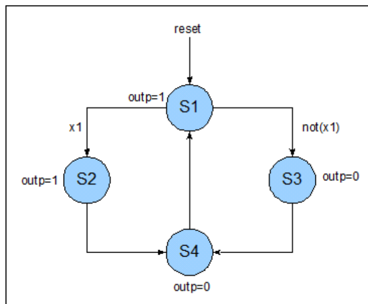
- Άλλες κωδικοποιήσεις: Gray, Johnson, one-cold

Παρατηρήσεις

- Για την αρχικοποίηση του FSM σε μία γνωστή αρχική κατάσταση επιβάλλεται η χρήση ασύγχρονης επανατοποθέτησης (asynchronous reset)
- Γενικά υφίστανται αρκετές τεχνικές για την σύνταξη της περιγραφής ενός FSM
  1. FSM με μία διεργασία (process): Η λογική επόμενης κατάστασης, τρέχουσας κατάστασης και εξόδουι σε μία PROCESS
  2. FSM με δύο διεργασίες: Η λογική επόμενης κατάστασης και τρέχουσας κατάστασης σε μία PROCESS και η λογική εξόδου σε μία δεύτερη
  3. FSM με τρεις διεργασίες: Η λογική επόμενης κατάστασης, τρέχουσας κατάστασης και εξόδου σε ξεχωριστές PROCESS
  4. FSM με δύο διεργασίες με τη λογική τρέχουσας κατάστασης σε μία PROCESS και τη λογική επόμενης κατάστασης και εξόδου σε μία δεύτερη PROCESS
  5. FSM με αποθηκευμένα σήματα εξόδου

Διάγραμμα μεταγωγής καταστάσεων για ένα απλό FSM 4 καταστάσεων

- Το FSM του παραδείγματος καθορίζεται από:
  - Τέσσερις καταστάσεις: S1, S2, S3, S4
  - Μία είσοδο: x1
  - Μία έξοδο: outp
  - Πέντε περιπτώσεις μετάβασης από κατάσταση σε κατάσταση σύμφωνα με το παρακάτω διάγραμμα μεταγωγής καταστάσεων



Παράδειγμα FSM με μία διεργασία

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

entity fsm_1 is
port (
  clk, reset, x1 : IN std_logic;
  outp : OUT std_logic
);
end fsm_1;

architecture beh1 of fsm_1 is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type;
begin
  process (clk, reset)
  begin
  if (reset ='1') then
    state <= s1;
    outp <= '1';
  elsif (clk='1' and clk'event) then
    case state is
```

```vhdl
      when s1 =>
        if (x1 = '1') then
          state <= s2;
          outp <= '1';
        else
          state <= s3;
          outp <= '0';
        end if;
      when s2 =>
        state <= s4;
        outp <= '0';
      when s3 =>
        state <= s4;
        outp <= '0';
      when s4 =>
        state <= s1;
        outp <= '1';
    end case;
  end if;
  end process;
end beh1;
```

Παράδειγμα FSM με δύο διεργασίες

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

entity fsm_2 is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic
  );
end fsm_2;

architecture beh1 of fsm_2 is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type;
begin
  process1: process (clk, reset)
  begin
  if (reset ='1') then
    state <= s1;
  elsif (clk='1' and clk'EVENT) then
    case state is
      when s1 =>
        if (x1 = '1') then
          state <= s2;
        else
          state <= s3;
        end if;
```

```
      when s2 =>
        state <= s4;
      when s3 =>
        state <= s4;
      when s4 =>
        state <= s1;
    end case;
  end if;
  end process process1;

  process2 : process (state)
  begin
  case state is
    when s1 => outp <= '1';
    when s2 => outp <= '1';
    when s3 => outp <= '0';
    when s4 => outp <= '0';
  end case;
  end process process2;
end beh1;
```

Παράδειγμα FSM με τρεις διεργασίες

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

entity fsm_3 is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic
  );
end fsm_3;

architecture beh1 of fsm_3 is
  type state_type is (s1,s2,s3,s4);
  signal current_state, next_state:
    state_type;
begin
  process1: process (clk, reset)
  begin
  if (reset ='1') then
    state <= s1;
  elsif (clk='1' and clk'EVENT) then
    current_state <= next_state;
  end if;
  end process process1;
```

```
  process2 : process (current_state, x1)
  begin
  case current_state is
    when s1 =>
      if (x1 = '1') then
        next_state <= s2;
      else
        next_state <= s3;
      end if;
    when s2 =>
      next_state <= s4;
    when s3 =>
      next_state <= s4;
    when s4 =>
      next_state <= s1;
  end case;
  end process process2;

  process3 : process (current_state)
  begin
  case current_state is
    when s1 => outp <= '1';
    when s2 => outp <= '1';
    when s3 => outp <= '0';
    when s4 => outp <= '0';
  end case;
  end process process3;
end beh1;
```

Το ίδιο παράδειγμα με δύο διεργασίες και απομόνωση της λογικής τρέχουσας κατάστασης

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

entity fsm_2b is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic
  );
end fsm_2b;

architecture beh1 of fsm_2b is
  type state_type is (s1,s2,s3,s4);
  signal current_state, next_state:
    state_type;
begin
  process1: process (clk, reset)
  begin
  if (reset ='1') then
    state <= s1;
  elsif (clk='1' and clk'EVENT) then
    current_state <= next_state;
  end if;
  end process process1;

  process2: process (current_state, x1)
  begin
  case current_state is
```

```
    when s1 =>
      outp <= '1';
      if (x1 = '1') then
        state <= s2;
      else
        state <= s3;
      end if;
    when s2 =>
      outp <= '1';
      state <= s4;
    when s3 =>
      outp <= '0';
      state <= s4;
    when s4 =>
      outp <= '0';
      state <= s1;
  end case;
  end process process2;
end beh1;
```

Το ίδιο παράδ. με δύο διεργασίες, απομόνωση λογικής τρέχουσας κατάστασης και αποθηκευμένη έξοδο

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

entity fsm_2c is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic
  );
end fsm_2c;

architecture beh1 of fsm_2c is
  type state_type is (s1,s2,s3,s4);
  signal current_state, next_state:
    state_type;
  signal temp: std_logic;
begin
  process1: process (clk, reset)
  begin
  if (reset ='1') then
    state <= s1;
  elsif (clk='1' and clk'EVENT) then
    outp <= temp;
    current_state <= next_state;
  end if;
  end process process1;

  process2: process (current_state, x1)
  begin
  case current_state is
     when s1 =>
       temp <= '1';
       if (x1 = '1') then
         state <= s2;
       else
         state <= s3;
       end if;
     when s2 =>
       temp <= '1';
       state <= s4;
     when s3 =>
       temp <= '0';
       state <= s4;
     when s4 =>
       temp <= '0';
       state <= s1;
   end case;
   end process process2;
end beh1;
```

Μη προγραμματιζόμενοι επεξεργαστές

- Μη προγραμματιζόμενοι επεξεργαστές είναι εκείνα τα κυκλώματα τα οποία έχουν σχεδιαστεί έτσι ώστε να μπορούν να επιλύσουν ένα μόνο πρόβλημα
- Ένας μη προγραμματιζόμενος επεξεργαστής αποτελείται από το χειριστή ελέγχου (controller ή control unit) και το χειριστή δεδομένων (datapath)
- Ο χειριστής ελέγχου παράγει σήματα ελέγχου για την δρομολόγηση των μηχανισμών που λαμβάνουν χώρα στον χειριστή δεδομένων
- Ο χειριστής δεδομένων επιστρέφει στο χειριστή ελέγχου σήματα κατάστασης (status signals) τα οποία κατευθύνουν τη μετάβαση ανάμεσα στις εσωτερικές καταστάσεις του χειριστή ελέγχου
- Ο χειριστής ελέγχου υλοποιείται συχνά ως FSM

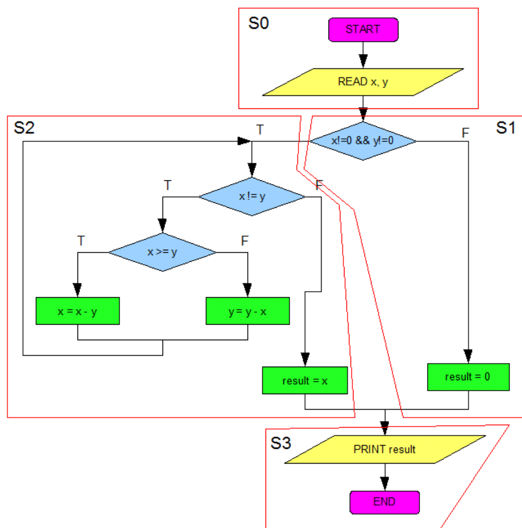
Ο αλγόριθμος του μέγιστου κοινού διαιρέτη δύο αριθμών (GCD)

- Δεχόμαστε ότι: $gcd(n, 0) = gcd(0, n) = gcd(0, 0) = 0$
- Το ζητούμενο είναι η εύρεση αριθμού m ο οποίος να είναι ο μεγαλύτερος θετικός ακέραιος ο οποίος διαιρεί και τους δύο αριθμούς

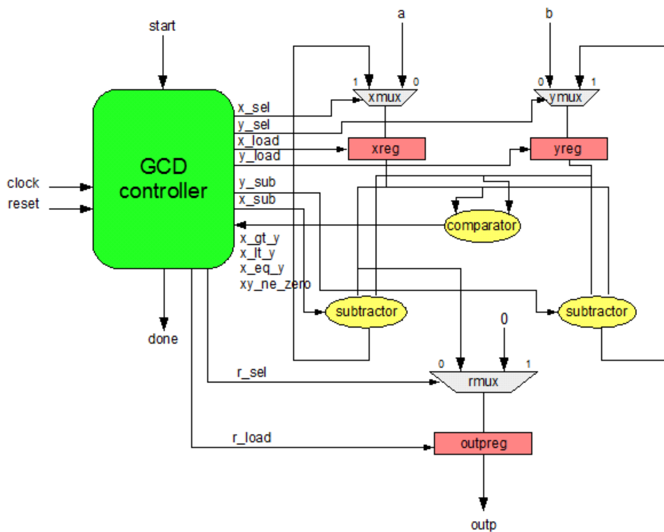
```
int gcd(int a, int b)
{
  int result;
  int x, y;

  x = a;
  y = b;

  if (x!=0 && y!=0)
  {
    while (x != y)
    {
      if (x >= y)
        x = x - y;
      else
        y = y - x;
    }
```

```
      result = x;
  }
  else
  {
    result = 0;
  }
  return (result);
}

int main()
{
  int result = gcd(196, 42);
  return (result);
}
```

Αλγοριθμικό διάγραμμα ροής για τον αλγόριθμο GCD



Το συνολικό κύκλωμα του επεξεργαστή GCD



Περιγραφή της υλοποίησης FSMD του επεξεργαστή GCD (1)

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

entity gcd is
  generic (
    WIDTH : integer
  );
  port (
    clock : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    a     : in std_logic_vector(WIDTH-1 downto 0);
    b     : in std_logic_vector(WIDTH-1 downto 0);
    outp  : out std_logic_vector(WIDTH-1 downto 0);
    done  : out std_logic
  );
end gcd;

architecture fsmd of gcd is
  type state_type is (s0,s1,s2,s3);
  signal state: state_type;
  signal x, y, res : std_logic_vector(WIDTH-1 downto 0);
begin
```

Περιγραφή της υλοποίησης FSMD του επεξεργαστή GCD (2)

```
  process (clock, reset)
  begin
done <= '0';
--
    if (reset = '1') then
      state <= s0;
      x <= (others => '0');
      y <= (others => '0');
      res <= (others => '0');
    elsif (clock='1' and clock'EVENT) then
      case state is
        when s0 =>
          if (start = '1') then
            x <= a;
            y <= b;
            state <= s1;
          else
            state <= s0;
          end if;
        when s1 =>
          if (x /= 0 and y /= 0) then
            state <= s2;
          else
            res <= (others => '0');
            state <= s3;
          end if;
```

Περιγραφή της υλοποίησης FSMD του επεξεργαστή GCD (3)

```
        when s2 =>
          if (x > y) then
            x <= x - y;
            state <= s2;
          elsif (x < y) then
            y <= y - x;
            state <= s2;
          else
            res <= x;
            state <= s3;
          end if;
        when s3 =>
          done <= '1';
          state <= s0;
      end case;
    end if;
  end process;

  outp <= res;

end fsmd;
```

Προσομοίωση του επεξεργαστή GCD

- Δηλώσεις FILE για λήψη εισόδων από αρχείο και εκτύπωση διαγνωστικής εξόδου σε αρχείο

```
file TestDataFile: text open read_mode is "gcd_test_data.txt";
file ResultsFile: text open write_mode is "gcd_alg_test_results.txt";
```

- Περιεχόμενα του "gcd_test_data.txt" (*A*, *B*, *result*)

```
21 49 7
25 30 5
19 27 1
40 40 40
250 190 10
5 250 5
1 1 1
0 0 0
```