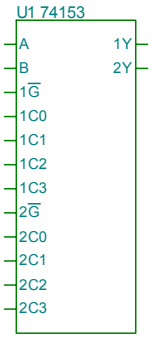
9. ΜΕΛΕΤΗ ΠΟΛΥΠΛΕΚΤΩΝ

Στο κεφάλαιο αυτό μας ενδιαφέρει η η γνώση και κατανόηση της χρήσης ενός πολυπλέκτη στην επίλυση προβλημάτων συνδυαστικής λογικής και η υλοποίηση λογικών συνάρτησεων με πολυπλέκτη.

1. IC SN74153. Διπλός πολυπλέκτης 4x1



Εικόνα 1. Σύμβολο στο TINA.

Το σύμβολο του SN74153 (και των επόμενων πολυπλεκτών) στο TINA, βρίσκεται μέσα στην παλέτα Logic ICs-MCUs στην κατηγορία MUX.

SELECT INPUTS		DATA INPUTS				STROBE	OUTPUT
B	A	C0	C1	C2	C3	G’	Y
X	X	X	X	X	X	H	L
L	L	L	X	X	X	L	L
L	L	H	X	X	X	L	H
L	H	X	L	X	X	L	L
L	H	X	H	X	X	L	H
H	L	X	X	L	X	L	L
H	L	X	X	H	X	L	H
H	H	X	X	X	L	L	L
H	H	X	X	X	H	L	H

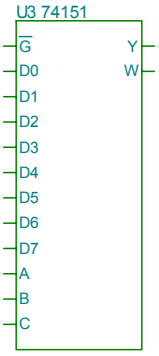
Πίνακας 1. Πίνακας αληθείας SN74153.

Συνοπτικότερα:

SELECT INPUTS		STROBE	OUTPUT
B	A	G’	Y
X	X	H	L
L	L	L	C0
L	H	L	C1
H	L	L	C2
H	H	L	C3

Πίνακας 2. Πίνακας αληθείας SN74153.

2. IC SN74151. Πολυπλέκτη 8x1

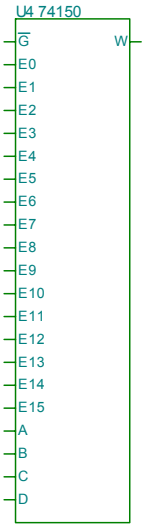


Εικόνα 2. Σύμβολο στο TINA.

SELECT INPUTS			STROBE	OUTPUTS	
C	B	A	G’	Y	W
X	X	X	H	L	H
L	L	L	L	D0	D0’
L	L	H	L	D1	D1’
L	H	L	L	D2	D2’
L	H	H	L	D3	D3’
H	L	L	L	D4	D4’
H	L	H	L	D5	D5’
H	H	L	L	D6	D6’
H	H	H	L	D7	D7’

Πίνακας 3. Πίνακας αληθείας SN74151.

3. Το IC SN74158. Πολυπλέκτης 16x1



Εικόνα 3. Σύμβολο στο TINA.

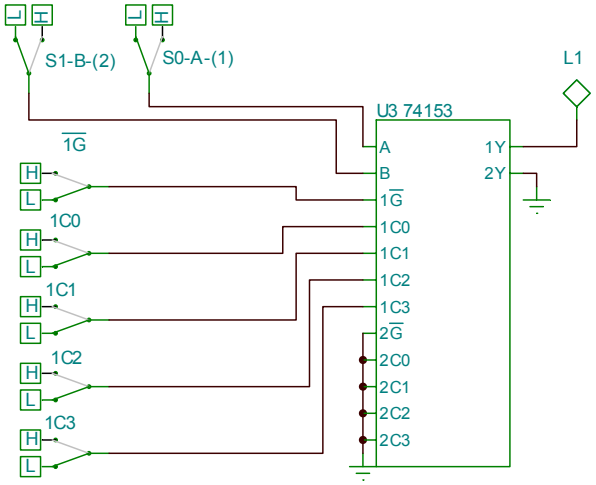
SELECT INPUTS				STROBE	OUTPUTS
D	C	B	A	G’	W
X	X	X	X	H	H
L	L	L	L	L	E0’
L	L	L	H	L	E1’
L	L	H	L	L	E2’
L	L	H	H	L	E3’
L	H	L	L	L	E4’

L	H	L	H	L	E5’
L	H	H	L	L	E6’
L	H	H	H	L	E7’
H	L	L	L	L	E8’
H	L	L	H	L	E9’
H	L	H	L	L	E10’
H	L	H	H	L	E11’
H	H	L	L	L	E12’
H	H	L	H	L	E13’
H	H	H	L	L	E14’
H	H	H	H	L	E15’

Πίνακας 4. Πίνακας αληθείας SN74158.

4. Λειτουργία του 74153. (MUX 4x1 (x2))

Σχεδιάστε το επόμενο κύκλωμα (**Εικόνα 4**), συμπληρώστε στον πίνακα τη στήλη του 1Y (**Πίνακας 5**) και περιγράψτε τη λειτουργία του.



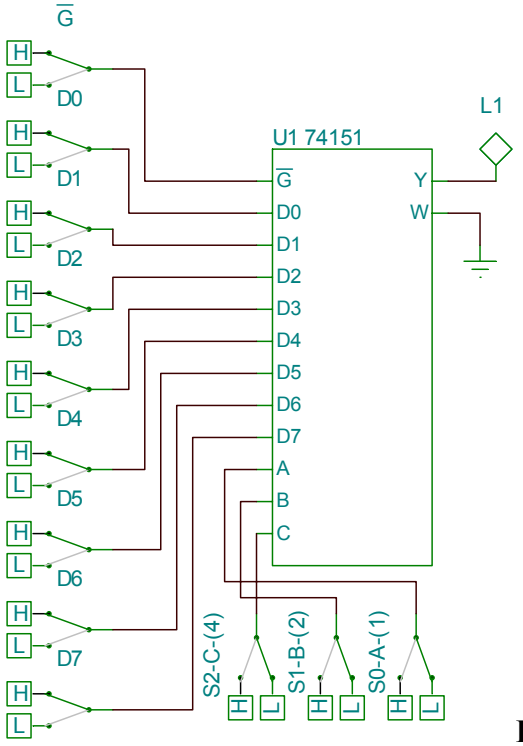
Εικόνα 4.

S1-B-(2)	S0-A-(1)	~1G~	1C0	1C1	1C2	1C3	1Y
0	0	0	0	X	X	X	
0	0	0	1	X	X	X	
0	1	0	X	0	X	X	
0	1	0	X	1	X	X	
1	0	0	X	X	0	X	
1	0	0	X	X	1	X	
1	1	0	X	X	X	0	
1	1	0	X	X	X	1	
0	0	1	0	X	X	X	
0	0	1	1	X	X	X	
0	1	1	X	0	X	X	
0	1	1	X	1	X	X	
1	0	1	X	X	0	X	
1	0	1	X	X	1	X	
1	1	1	X	X	X	0	
1	1	1	X	X	X	1	

Πίνακας 5.

5. Λειτουργία του 74151. (MUX 8x1)

Σχεδιάστε, προσομοιώστε και εξάγετε τον πίνακα αληθείας του επόμενου κυκλώματος (**Εικόνα 5**).



Εικόνα 5.

6. Υλοποίηση συναρτήσεων Boole με τη χρήση πολυπλέκτη

Οι πολυπλέκτες μπορούν να χρησιμοποιηθούν για την υλοποίηση συναρτήσεων Boole. Η πιο απλή μέθοδος για την υλοποίηση είναι η επόμενη.

- **Βήμα 1**. Γράφουμε τον πίνακα αληθείας της λογικής συνάρτησης
- **Βήμα 2**. Επιλέγουμε όλες τις μεταβλητές εκτός αυτής με τη μικρότερη αξία για γραμμές επιλογής. Π.χ. αν η συνάρτηση εξαρτάται από τις μεταβλητές A B C D (με το A το ψηφίο με τη μεγαλύτερη αξία), τότε επιλέγουμε τις μεταβλητές A, B, C για γραμμές επιλογής ενός 2^3x1 MUX.
- **Βήμα 3**. Εκφράζουμε τη συνάρτηση εξόδου συναρτήσεις μόνο της μεταβλητής που αφήσαμε (δηλαδή τη D στο παράδειγμά μας) και για κάθε συνδυασμό A, B, C βάζουμε στην κατάλληλη είσοδο την απαραίτητη σχέση που αντιστοιχεί στον τύπο της F.

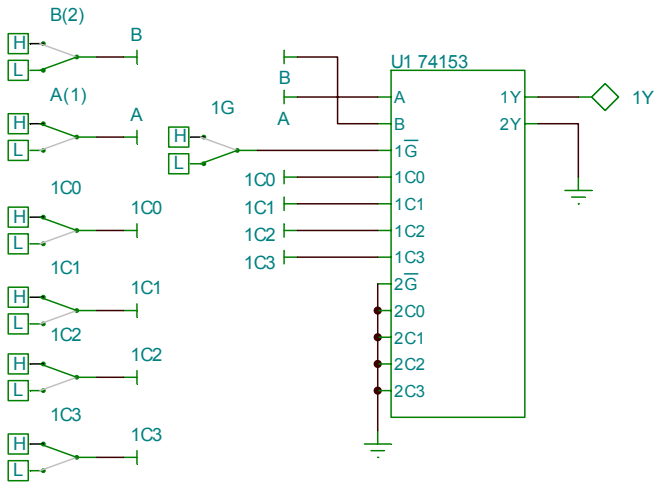
Η διαδικασία θα γίνει καλύτερα κατανοητή με τα επόμενα παραδείγματα-ασκήσεις.

7. Ασκήσεις

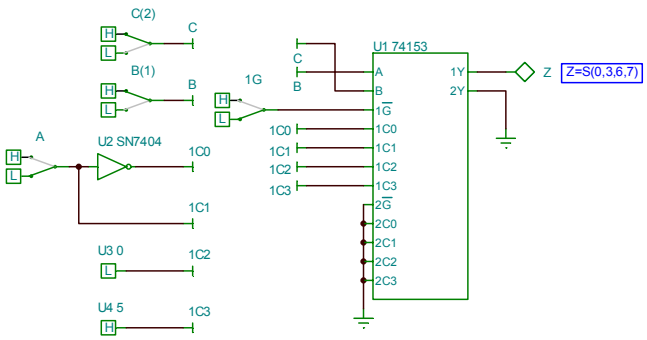
1. Υλοποίηση λογικής συνάρτησης με MUX. Υλοποιήστε με MUX 4x1 (74153) τη λογική συνάρτηση: Z = Σ(0,3,6,7).

Λύση.

Στο επόμενα σχήμα (**Εικόνα 6, 7**) βλέπουμε την υλοποίηση που ενεργοποιεί τον 1^{ο} από τους δύο πολυπλέκτες. Προσοχή χρειάζεται στα select A, B των οποίων η αξία είναι 2 για το B και 1 για το A. Δηλαδή η σειρά των select για το 74153 είναι B,A. Επίσης για να λειτουργήσει ο MUX πρέπει το 1G να είναι στο 0 ώστε το 1G' = 1. Δημιουργώ τον πίνακα αληθείας της Z και σημειώνω τους 1 στις αντίστοιχες θέσεις ελαχιστόρων. Από τον πίνακα αληθείας (**Πίνακας 6**) επιλέγω τις δύο μεγαλύτερες αξίες σε ρόλο select. Δηλαδή εδώ τα C, B και τα συνδέω με τα B, A του MUX αντίστοιχα. Στη συνέχεια εκφράζω τη Z συναρτήσει μόνο του A για κάθε συνδυασμό των CB.



Εικόνα 6.



Εικόνα 7.

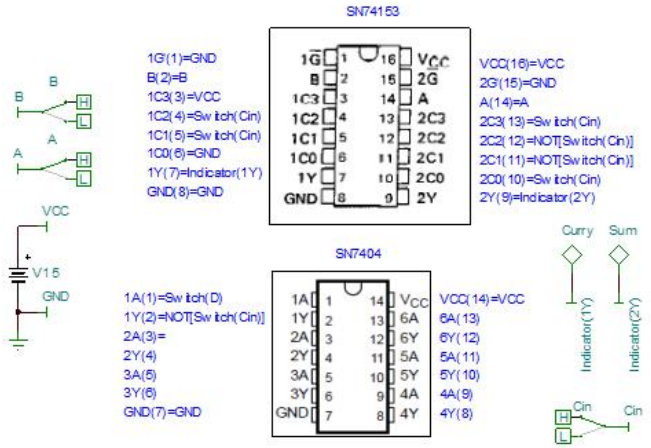
Number	C	B	A	Z	Z=
0	0	0	0	1	
1	0	0	1	0	1C0=A'
2	0	1	0	0	
3	0	1	1	1	1C1=A
4	1	0	0	0	
5	1	0	1	0	1C2=0
6	1	1	0	1	
7	1	1	1	1	1C3=1

Πίνακας 6.

2. Υλοποιήσετε με το 74153 τη συνάρτηση του πλήρη αθροιστή.

Λύση.

Βλέπε **Πίνακα 7** και **Εικόνα 8**.



Εικόνα 8.

					1ος MUX.4x1 (1C0,1C1,1C2,1C3,1Y) 1G'(1)=0		2ος MUX.4x1 (2C0,2C1,2C2,2C3,2Y) 2G'(15)=0
i	**B(2)**	**A(14)**	**Cin**	**Cout**	**1Y(7) = Cout**	**Sum**	**2Y(9)=Sum**
0	0	0	0	0		0	
1	0	0	1	0	**1C0(6)=0**	1	**2C0(10)=Cin**
2	0	1	0	0		1	
3	0	1	1	1	**1C1(5)=Cin**	0	**2C1(11)=Cin'**
4	1	0	0	0		1	
5	1	0	1	1	**1C2(4)=Cin**	0	**2C2(12)=Cin'**
6	1	1	0	1		0	
7	1	1	1	1	**1C3(3)=1**	1	**2C3(13)=Cin**

Πίνακας 7.

3. Υλοποίηση λογικής συνάρτησης με MUX. Υλοποιήστε με MUX 4x1(74153) τη λογική συνάρτηση: Z = Σ(0,1,3,8,9,10,15).

Λύση.

Επιλέγω τις δύο μεγαλύτερης τάξης μεταβλητές (D, C) σε ρόλο select. Με τον τρόπο αυτό ο πίνακας αληθείας χωρίζεται σε 4 περιοχές. Στη συνέχεια εκφράζω τη Z=Z(B,A) και κάνω την κατάλληλη σύνδεση στις εισόδους του πολύπλέκτη. Στη συγκεκριμένη περίπτωση θα χρειαστούν αρκετές επιπλέον πύλες. (Βλέπε **Πίνακα 8**).

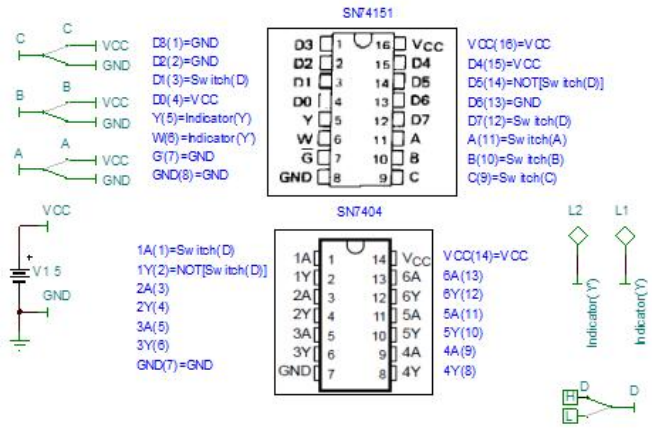
Number	D	C	B	A	Z	Z=
0	0	0	0	0	1	
1	0	0	0	1	1	
2	0	0	1	0	0	
3	0	0	1	1	1	Z =I0= B'A'+B'A+BA
4	0	1	0	0	0	
5	0	1	0	1	0	
6	0	1	1	0	0	
7	0	1	1	1	0	Z = I1 = 0
8	1	0	0	0	1	
9	1	0	0	1	1	
10	1	0	1	0	1	
11	1	0	1	1	0	Z = I2 = B'A' + B'A +BA'
12	1	1	0	0	0	
13	1	1	0	1	0	
14	1	1	1	0	0	
15	1	1	1	1	1	Z = I3 = BA

Πίνακας 8.

4. Υλοποίηση λογικής συνάρτησης με MUX. Υλοποιήστε με MUX 8x1(74151) τη λογική συνάρτηση: Z = Σ(0,1,3,8,9,10,15).

Λύση.

Βλέπε **Πίνακα 9** και **Εικόνα 9**.



Εικόνα 9.

i	C(9)	B(10)	A(11)	D	Z	Y(5) G'(7)=0
0	0	0	0	0	1	
1	0	0	0	1	1	D0(4)=1
2	0	0	1	0	0	
3	0	0	1	1	1	D1(3)=D
4	0	1	0	0	0	
5	0	1	0	1	0	D2(2)=0
6	0	1	1	0	0	
7	0	1	1	1	0	D3(1)=0
8	1	0	0	0	1	
9	1	0	0	1	1	D4(15)=1
10	1	0	1	0	1	
11	1	0	1	1	0	D5(14)=D'
12	1	1	0	0	0	
13	1	1	0	1	0	D6(13)=0
14	1	1	1	0	0	
15	1	1	1	1	1	D7(12)=D

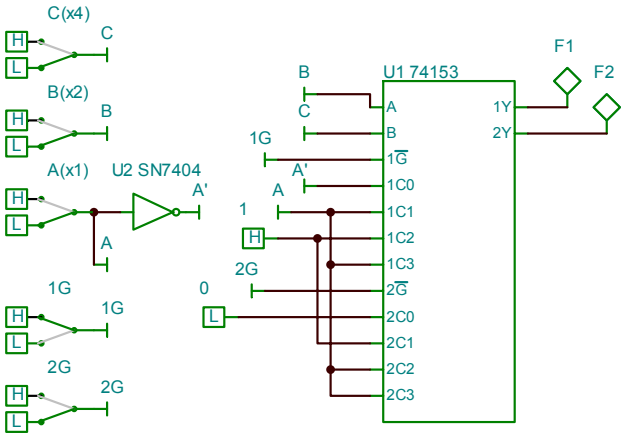
Πίνακας 9.

5. Υλοποιήστε με το 74153 και τις δύο συναρτήσεις F1 = Σ(0,3,4,5,7) και F2 = Σ(2,3,5,7).

Λύση.

i	C(x4) MUX =B	B(x2) MUX =A	A(x 1)	F1 MUX=(1Y)		F2 MUX=(2Y)	
0	0	0	0	1	**1C0= A'**	0	**2C0 =0**
1	0	0	1	0		0	
2	0	1	0	0	**1C1= A**	1	**2C1 =1**
3	0	1	1	1		1	
4	1	0	0	1	**1C2= 1**	0	**2C2 =A**
5	1	0	1	1		1	
6	1	1	0	0	**1C3= A**	0	**2C3 =A**
7	1	1	1	1		1	

Πίνακας 10.



Εικόνα 10.

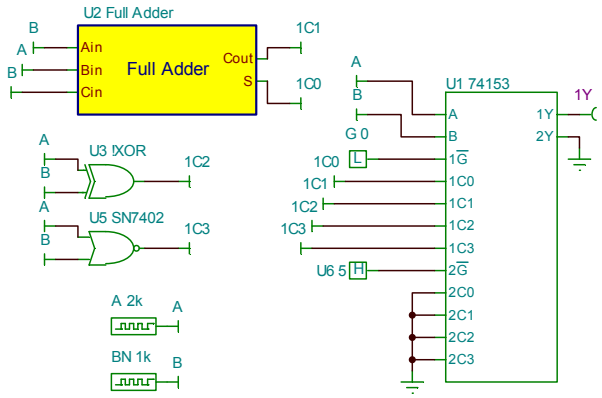
6. Συμπληρώστε τον πίνακα αληθείας της F.

C	B	A	F		U1 74153		
0	0	0			**1G**	0	
0	0	1			**A**	B	
0	1	0			**B**	C	

0	1	1			**1C0**	0	
1	0	0			**1C1**	A	
1	0	1			**1C2**	1	
1	1	0			**1C3**	A'	
1	1	1			**1Y**	F	

Πίνακας 10.

7. **Υλοποιήστε με χρήση πολυπλέκτη τις βασικές πύλες ψηφιακής λογικής, NOT, AND, OR, XOR XNOR, NAND, NOR με δύο και τρεις εισόδους.**
8. **Υλοποιήστε με MUX τη συνάρτηση F = A'D + C + D' + BC'.**
9. **Σχεδιάστε διάγραμμα χρονισμού για το επόμενο κύκλωμα.**



Εικόνα 11.

Υλοποιήσεις σε VHDL

MUX2x1 αρχιτεκτονικής τύπου ροής δεδομένων.

Λύση.

```
--Dataflow level VHDL description for the 2-input multiplexer

ENTITY multiplexer IS PORT(d0, d1, s: IN BIT; y: OUT BIT);
END multiplexer;
ARCHITECTURE Dataflow OF multiplexer IS
BEGIN
        y <= d0 WHEN s = '0' ELSE d1;
END Dataflow;
```

Λίστα 1.

MUX2x1 αρχιτεκτονικής τύπου συμπεριφοράς.

Λύση.

```
--Behavioral level VHDL description for the 2-input multiplexer

ENTITY multiplexer IS PORT (d0, d1, s: IN BIT; y: OUT BIT);
END multiplexer;
ARCHITECTURE Behavioral OF multiplexer IS
BEGIN
        PROCESS(s, d0, d1)
        BEGIN
        y <= d0 WHEN s = '0' ELSE d1;
        END PROCESS;
END Behavioral;
```

Λίστα 2.

MUX2x1 αρχιτεκτονικής τύπου διασύνδεσης διατάξεων.

Λύση.

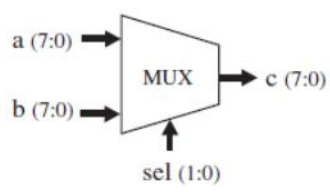
```
--Structural level VHDL description for the 2-input multiplexer

ENTITY myand2 IS PORT (i1, i2: IN BIT; o: OUT BIT);
END myand2;
ARCHITECTURE Dataflow OF myand2 IS
BEGIN
        o <= i1 AND i2;
END Dataflow;
ENTITY myor2 IS PORT (i1, i2: IN BIT; o: OUT BIT);
END myor2;
ARCHITECTURE Dataflow OF myor2 IS
BEGIN
        o <= i1 OR i2;
END Dataflow;
ENTITY myinv IS PORT (i: IN BIT; o: OUT BIT);
END myinv;
ARCHITECTURE Dataflow OF myinv IS
BEGIN
        o <= not i;
END Dataflow;
ENTITY multiplexer IS PORT (d0, d1, s: IN BIT;y: OUT BIT);
END multiplexer;
```

```
ARCHITECTURE Structural OF multiplexer IS
COMPONENT myand2 PORT (i1, i2: IN BIT;o: OUT BIT);
END COMPONENT;
COMPONENT myor2 PORT (i1, i2: IN BIT; o: OUT BIT);
END COMPONENT;
COMPONENT myinv PORT (i: IN BIT; o: OUT BIT);
END COMPONENT;
SIGNAL sn, asn, sb: BIT;
BEGIN
U1: myinv PORT MAP(s, sn);
U2: myand2 PORT MAP(d0, sn, asn);
U3: myand2 PORT MAP(s, d1, sb);
U4: myor2 PORT MAP(asn, sb, y);

END Structural;
```

Λίστα 3.

Άσκηση.

Θεωρούμε το μπλοκ διάγραμμα ενός πολυπλέκτη και τον αντίστοιχο πίνακα αληθείας του:



sel	c
00	0
01	a
10	b
11	Z

Συμπληρώστε τα κενά στον επόμενο κώδικα:

```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE _____________________________ ;
4 ---------------------------------------
5 ENTITY mux IS
6 PORT ( __ , __ : ___ STD_LOGIC_VECTOR
(7 DOWNTO 0);
7 sel : IN _____________________________ ;
8 ___ : OUT STD_LOGIC_VECTOR (7
DOWNTO 0));
9 END _____ ;
10 ---------------------------------------
11 ARCHITECTURE example OF _____ IS
12 BEGIN
13 PROCESS (a, b, ____ )
14 BEGIN
15 IF (sel = "00") THEN
16 c <= "00000000";
17 ELSIF (___________) THEN
18 c <= a;
19 _____ (sel = "10") THEN
20 c <= __;
21 ELSE
22 c <= (OTHERS => '__');
23 END ___ ;
24 END _________ ;
25 END _________ ;
26 ---------------------------------------
```

Σύγχρονος (Concurrent) VHDL Κώδικας

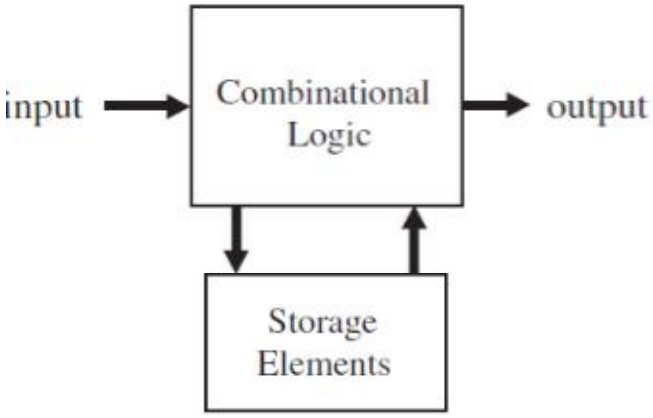
Ο κώδικας της VHDL μπορεί να είναι παράλληλος (concurrent) ή σειριακός.

Οι παράλληλες δηλώσεις στη VHDL χρησιμοποιούν τις WHEN και GENERATE. Εκτός από αυτές, δηλώσεις που χρησιμοποιούν μόνον τελεστές (AND, NOT, +, *, SLL κτλ) μπορούν να χρησιμοποιηθούν για την κατασκευή παράλληλου κώδικα. Τέλος η εντολή BLOCK χρησιμοποιείται για αυτό το είδος κωδικοποίησης.

Εξορισμού, η συνδυαστική λογική είναι αυτή στην οποία η έξοδος τους κυκλώματος εξαρτάται αποκλειστικά από τις τρέχουσες εισόδους του. Δηλαδή σε μια τέτοια περίπτωση το σύστημα δε χρειάζεται μνήμη και μπορεί να υλοποιηθεί μόνον με τη χρήση συμβατικών πυλών.

Αντίθετα, στην ακολουθιακή λογική η έξοδος του κυκλώματος εξαρτάται από προηγούμενες τιμές των εισόδων. Οπότε χρειάζονται στοιχεία μνήμης, τα οποία συνδέονται με τα συνδυαστικά στοιχεία του κυκλώματος έτσι που να σχηματίζεται βρόγχος ανάδρασης.

Ένα συνηθισμένος λάθος είναι να θεωρούμε ότι κάθε κύκλωμα που έχει στοιχεία μνήμης είναι οπωσδήποτε ακολουθιακό. Π.χ. η RAM είναι ένα τέτοιο παράδειγμα και το μπλοκ διάγραμμά της φαίνεται στη συνέχεια:



Σε αυτό το μοντέλο, τα στοιχεία μνήμης δεν εμφανίζονται σε κύκλο ανάδρασης. Η διαδικασία ανάγνωσης εξαρτάται μόνο από την τρέχουσα τιμή του διανύσματος διεύθυνσης που εφαρμόζουμε στην είσοδο της RAM και η τιμή που προκύπτει δεν εξαρτάται από τις προηγούμενες προσπελάσεις την μνήμης.

Ο κώδικας VHDL είναι ενδογενώς παράλληλος. Σειριακά εκτελούνται μόνον δηλώσεις που βρίσκονται μέσα σε σώμα PROCESS, FUNCTION ή PROCEDURE. Παρόλο που εντός αυτών των τμημάτων κώδικα η εκτέλεση των εντολών είναι σειριακή, το μπλοκ στο σύνολό του εκτελείται παράλληλα με οποιεσδήποτε άλλες εξωτερικές δηλώσεις. Ο παράλληλος κώδικας ονομάζεται και κώδικας ροής δεδομένων (dataflow code).

Συνεπώς, στον παράλληλο κώδικα δεν έχει σημασία η σειρά με την οποία γράφονται οι δηλώσεις εντολών. Οπότε, πλήρως παράλληλος κώδικας δε μπορεί να χρησιμοποιηθεί για την υλοποίηση σύγχρονων κυκλωμάτων (η μόνη εξαίρεση είναι με τη χρήση του GUARDED BLOCK). Δηλαδή με παράλληλο κώδικα μπορούμε να υλοποιήσουμε μόνο συνδυαστικά κυκλώματα. Για σειριακά κυκλώματα (ακολουθιακά), θα πρέπει να χρησιμοποιήσουμε σειριακό κώδικα. Με αυτόν μπορούμε να υλοποιήσουμε τόσο ακολουθιακά όσο και συνδυαστικά κυκλώματα.

Στη συνέχεια θα μελετήσουμε τα χαρακτηριστικά του παράλληλου κώδικα, δηλαδή δηλώσεων που γράφονται έξω από PROCESSES,

FUNCTION ή PROCEDURES. Δηλαδή τις δηλώσεις WHEN και GENERATE.

Χρήση τελεστών για παράλληλη κωδικοποίηση

Πρόκειται για το βασικό είδος παράλληλης κωδικοποίησης στη VHDL. Ο επόμενος πίνακας συνοψίζει τους τελεστές και τους τύπους δεδομένων πάνω στους οποίους ενεργούν:

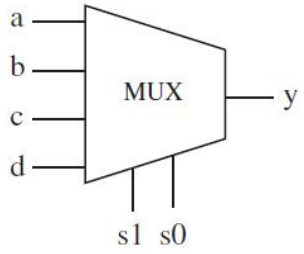
Τύπος τελεστή	**Τελεστής**	**Τύπος δεδομένων**
Λογικής	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Αριθμητικής	þ, _, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Σύγκρισης	=,/=,<,>,<=,>=	Όλοι οι παραπάνω
Ολίσθησης	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Συρραφής	&, (…)	Όπως και για τους τελεστές λογικής και για τους SIGNED και UNSIGNED

Οι τελεστές μπορούν να χρησιμοποιηθούν για να υλοποιήσουν οποιοδήποτε συνδυαστικό κύκλωμα. Ωστόσο, τα πολύπλοκα κυκλώματα είναι ευκολότερο να υλοποιηθούν με ακολουθιακό κώδικα, ακόμα και αν το κύκλωμα δεν περιέχει ακολουθιακά στοιχεία.

Στη συνέχεια θα δούμε παραδείγματα κυκλωμάτων υλοποιημένα μόνον με τελεστές λογικής.

Παράδειγμα. Πολυπλέκτης

Θα υλοποιήσουμε έναν πολυπλέκτη 4x1 του οποίου το μπλοκ διάγραμμα φαίνεται στη συνέχεια:



Καθεμιά από τις εισόδους τους είναι 1bit. Η έξοδος θα πρέπει να είναι ίση με την αντίστοιχη είσοδο, που επιλέγεται μέσω των s1 s0. Η υλοποίησή του μόνο με τελεστές λογικής φαίνεται στη συνέχεια:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
PORT ( a, b, c, d, s0, s1: IN
STD_LOGIC;
y: OUT STD_LOGIC);
END mux;

ARCHITECTURE pure_logic OF mux IS
BEGIN
y <= (a AND NOT s1 AND NOT s0) OR
 (b AND NOT s1 AND s0) OR
 (c AND s1 AND NOT s0) OR
 (d AND s1 AND s0);
END pure_logic;
```

Η χρήση του WHEN

Η WHEN είναι από τις βασικές εντολές παράλληλου κώδικα (μαζί με τους τελεστές και την GENERATE). Εμφανίζεται σε δύο μορφές: ως WHEN/ELSE (απλή WHEN) και ως WITH / SELECT/WHEN (selected WHEN). Η σύνταξή της δίνεται στη συνέχεια:

```
WHEN / ELSE:
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
WITH / SELECT / WHEN:
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

Όταν χρησιμοποιούμε την WITH/SELECT/WHEN, πρέπει να ελέγξουμε όλους τους συνδυασμούς, οπότε η OTHERS είναι χρήσιμη σε αυτή την περίπτωση. Μια ακόμα σημαντική εντολή είναι η UNAFFECTED, η οποία χρησιμοποιείται όταν δεν πρέπει να γίνει καμία ενέργεια.

Παράδειγμα:

```
------ With WHEN/ELSE
outp <=
"000"  WHEN  (inp='0'  OR  reset='1')
ELSE
"001" WHEN ctl='1'
ELSE
"010";

---- With WITH/SELECT/WHEN
WITH control SELECT
output <=
```

```
"000" WHEN reset,
"111" WHEN set,
UNAFFECTED WHEN OTHERS;
```

Μια άλλη πτυχή της εντολής WHEN είναι αυτή της "WHEN value" που είδαμε στο προηγούμενο παράδειγμα, η οποία παίρνει τρεις μορφές:

```
WHEN value
-- single value

WHEN value1 to value2
-- range, for enumerated data types
-- only

WHEN value1 | value2 |...
-- value1 or value2 or ...
```

Παράδειγμα. Υλοποίηση πολυπλέκτη με τη χρήση της WHEN

Με τη χρήση της WHEN/ELSE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
PORT ( a, b, c, d: IN
STD_LOGIC;
sel: IN
STD_LOGIC_VECTOR (1 DOWNTO 0);
y: OUT STD_LOGIC);
END mux;

ARCHITECTURE mux1 OF mux IS
BEGIN
y <= a WHEN sel="00" ELSE
     b WHEN sel="01" ELSE
     c WHEN sel="10" ELSE
     d;
END mux1;
```

Με τη χρήση της WHEN/SELECT/WHEN:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
PORT ( a, b, c, d: IN
STD_LOGIC;
sel: IN
STD_LOGIC_VECTOR (1 DOWNTO 0);
y: OUT STD_LOGIC);
END mux;

ARCHITECTURE mux2 OF mux IS
BEGIN
WITH sel SELECT
y <= a WHEN "00",
-- notice "," instead of ";"
     b WHEN "01",
     c WHEN "10",
     d WHEN OTHERS;
-- cannot be "d WHEN "11"
END mux2;
```

Στην παραπάνω λύση, το sel θα μπορούσε να δηλωθεί ως INTEGER, οπότε ο κώδικας γίνεται:

Με τη χρήση της WHEN/ELSE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
sel: IN INTEGER RANGE 0 TO 3;
y: OUT STD_LOGIC);
END mux;

---- Solution 1: with WHEN/ELSE ----------
ARCHITECTURE mux1 OF mux IS
BEGIN
y <= a WHEN sel=0 ELSE
     b WHEN sel=1 ELSE
     c WHEN sel=2 ELSE
     d;
END mux1;
```

```
-- Solution 2: with WITH/SELECT/WHEN --------
ARCHITECTURE mux2 OF mux IS
BEGIN
WITH sel SELECT
y <= a WHEN 0,
     b WHEN 1,
     c WHEN 2,
     d WHEN 3;
-- here, 3 or OTHERS are equivalent,
-- for all options are tested anyway
END mux2;
```

Σημειώνουμε σε αυτό το σημείο, ότι κάθε φορά μια μόνο ARCHITECTURE γίνεται να συντεθεί κάθε φορά. Άρα όταν μέσα στον κώδικα παρουσιάζονται περισσότερες αρχιτεκτονικές, είναι σημαντικό όλες εκτός μίας να γίνονται σχόλια (με χρήση --), αλλιώς θα πρέπει να χρησιμοποιήσουμε κείμενο σύνθεσης (synthesis script), για να γίνει σύνθεση της μίας. Στις προσομοιώσεις, η χρήση της δήλωσης CONFIGURATION εξασφαλίζει την επιλογή μιας συγκεκριμένης αρχιτεκτονικής.

Παράδειγμα. Τρισταθής συγκρατητής (tristate buffer)

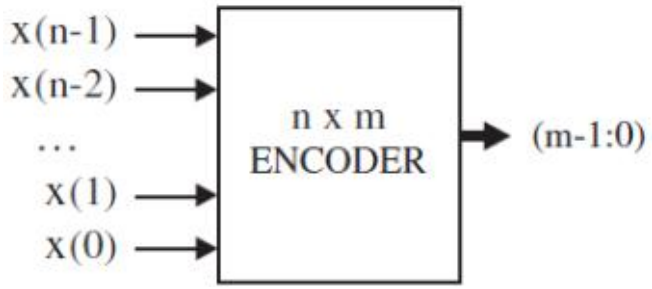
Θα δούμε ακόμα ένα παράδειγμα με τη χρήση της WHEN. Ο τρισταθής συγκρατητής παρέχει output = input όταν ena=low ή output = "ZZZZZZZZ» (υψηλή εμπέδηση) διαφορετικά. Αυτό γίνεται ως εξής:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tri_state IS
```

```
PORT ( ena: IN
STD_LOGIC;
input: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
output: OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
END tri_state;

ARCHITECTURE tri_state OF tri_state
IS
BEGIN
output <=
  input WHEN (ena='0') ELSE
 (OTHERS => 'Z');
END tri_state;
```

Παράδειγμα. Κωδικοποιητής

Με τη χρήση της WHEN θα υλοποιήσουμε τον κωδικοποιητή του οποίου το μπλοκ διάγραμμα βλέπουμε στη συνέχεια:



Υποθέτουμε ότι το n είναι δύναμη του 2, οπότε $m=\log_2 n$. Κάθε φορά μόνο ένα bit εισόδου θα είναι high και η διεύθυνση αυτή κωδικοποιείται στην έξοδο. Δύο λύσεις παρουσιάζονται.

Με χρήση WHEN/ELSE:

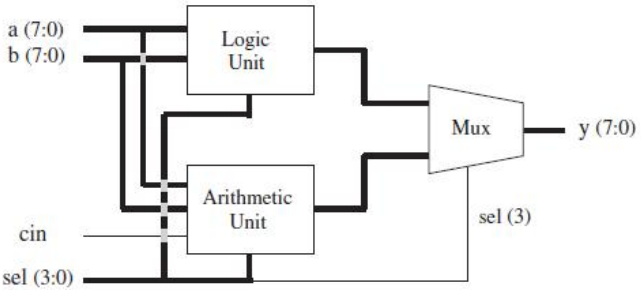
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY encoder IS
PORT ( x: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
y: OUT
STD_LOGIC_VECTOR (2 DOWNTO 0));
END encoder;

ARCHITECTURE encoder1 OF encoder IS
BEGIN
y <= "000" WHEN x="00000001" ELSE
     "001" WHEN x="00000010" ELSE
     "010" WHEN x="00000100" ELSE
     "011" WHEN x="00001000" ELSE
     "100" WHEN x="00010000" ELSE
     "101" WHEN x="00100000" ELSE
     "110" WHEN x="01000000" ELSE
     "111" WHEN x="10000000" ELSE
     "ZZZ";
END encoder1;
```

Με χρήση WITH/SELECT/WHEN:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY encoder IS
PORT ( x: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
y: OUT
STD_LOGIC_VECTOR (2 DOWNTO 0));
END encoder;

ARCHITECTURE encoder2 OF encoder IS
BEGIN
WITH x SELECT
y <= "000" WHEN "00000001",
"001" WHEN "00000010",
"010" WHEN "00000100",
"011" WHEN "00001000",
"100" WHEN "00010000",
"101" WHEN "00100000",
"110" WHEN "01000000",
"111" WHEN "10000000",
"ZZZ" WHEN OTHERS;
END encoder2;
```

Παράδειγμα. ALU (Arithmetic Logic Unit)

Το μπλοκ διάγραμμα μιας μονάδας αριθμητικής – λογικής εικονίζεται στη συνέχεια:



Η γενικές λειτουργίες της περιγράφονται συνοπτικά στον επόμενο πίνακα:

sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

Η έξοδος (αριθμητική ή λογική) επιλέγεται από το MSB του sel, ενώ η συγκεκριμένη λειτουργία από τα υπόλοιπα bits του sel.

Η υλοποίηση της ALU που φαίνεται στη συνέχεια, χρησιμοποιεί μόνον παράλληλο κώδικα. Επίσης χρησιμοποιεί τον ίδιο τύπο δεδομένων τόσο για αριθμητικές πράξεις, όσο και για πράξεις λογικής. Αυτό επιτυγχάνεται με τη χρήση του πακέτου std_logic_unsigned της βιβλιοθήκης ieee. Δύο σήματα με τα ονόματα arith και logic χρησιμοποιούνται για να κρατάνε τα αποτελέσματα των μονάδων αριθμητικής και λογικής αντίστοιχα και οι τιμές τους περνάνε στην έξοδο μέσου του πολυπλέκτη.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY ALU IS
PORT (a, b: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
sel: IN
STD_LOGIC_VECTOR (3 DOWNTO 0);
cin: IN
STD_LOGIC;
y: OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
END ALU;

ARCHITECTURE dataflow OF ALU IS
SIGNAL arith, logic:
STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
----- Arithmetic unit: ------
WITH sel(2 DOWNTO 0) SELECT
arith <=
a WHEN "000",
a+1 WHEN "001",
a-1 WHEN "010",
b WHEN "011",
b+1 WHEN "100",
b-1 WHEN "101",
a+b WHEN "110",
a+b+cin WHEN OTHERS;
----- Logic unit: -----------
WITH sel(2 DOWNTO 0) SELECT
logic <=
NOT a WHEN "000",
NOT b WHEN "001",
a AND b WHEN "010",
a OR b WHEN "011",
a NAND b WHEN "100",
a NOR b WHEN "101",
a XOR b WHEN "110",
NOT (a XOR b) WHEN OTHERS;
-------- Mux: --------------
WITH sel(3) SELECT
y <= arith WHEN '0',
logic WHEN OTHERS;
END dataflow;
```

Η GENERATE

Η GENERATE είναι μια ακόμα δήλωση παράλληλου κώδικα (μαζί με τους τελεστές και τη WHEN). Είναι ισοδύναμη με την ακολουθιακή δήλωση LOOP με την έννοια ότι επιτρέπει σε ένα κομμάτι κώδικα να επαναληφθεί για ένα πλήθος φορών, οπότε δημιουργεί αρκετές εκδόσεις των ίδιων εντολών.

Η κανονική της μορφή είναι ως FOR / GENERATE με τη δομή που φαίνεται στη συνέχεια. Σημειώνουμε ότι πρέπει να έχει τιτλοδότηση (label):

```
label: FOR identifier IN range
GENERATE
(concurrent assignments)
END GENERATE;
```

Μια μη κανονική της μορφή χρησιμοποιεί τη δομή IF / GENERATE (το IF είναι ακολουθιακή δήλωση). Δεν επιτρέπεται στην περίπτωση αυτή το ELSE. Η IF / GENERATE μπορεί να είναι εμφωλευμένη εντός FOR / GENERATE και το αντίστροφο:

```
label1: FOR identifier IN range
GENERATE
...
label2: IF condition GENERATE
(concurrent assignments)
END GENERATE;
...
END GENERATE;
```

Παράδειγμα:

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
z(i) <= x(i) AND y(i+8);
END GENERATE;
```

Μια σημαντική παρατήρηση για τη GENERATE (και το ίδιο ισχύει για τη LOOP) είναι ότι και τα δύο όριά της πρέπει να είναι στατικά. Για παράδειγμα, στον επόμενο κώδικα, η choice είναι είσοδος μη-στατική, οπότε ο κώδικα αυτός γενικά δεν είναι συνθέσιμος:

```
NotOK: FOR i IN 0 TO choice GENERATE
(concurrent statements)
END GENERATE;
```

Θα πρέπει επίσης να προσέχουμε για σήματα που είναι πολλαπλώς καθοδηγούμενα, π.χ. ο επόμενος κώδικας είναι εντάξει:

```
OK: FOR i IN 0 TO 7 GENERATE
output(i)<='1' WHEN (a(i) AND
b(i))='1' ELSE '0';
END GENERATE;
```

Αλλά ο μεταγλωττιστής θα σταματήσει τη μεταγλώττιση σε καθεμιά από τις δύο επόμενες περιπτώσεις:

```
NotOK: FOR i IN 0 TO 7 GENERATE
accum <="11111111" WHEN (a(i) AND
b(i))='1' ELSE "00000000";
END GENERATE;
```

```
NotOK: For i IN 0 to 7 GENERATE
accum <= accum + 1 WHEN x(i)='1';
END GENERATE;
```

Παράδειγμα. Ολισθητής διανύσματος

Στο παράδειγμα μας αυτό, θα δούμε τη χρήση της GENERATE. Το διάνυσμα εξόδου θα είναι μια ολισθημένη έκδοση του διανύσματος εισόδου, με το διπλάσιο του μήκους του και με μήκος ολίσθησης που προσδιορίζεται από μια άλλη είσοδο:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shifter IS
PORT ( inp: IN
STD_LOGIC_VECTOR (3 DOWNTO 0);
sel: IN
INTEGER RANGE 0 TO 4;
outp: OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
END shifter;

ARCHITECTURE shifter OF shifter IS
SUBTYPE vector IS
STD_LOGIC_VECTOR (7 DOWNTO 0);
TYPE matrix IS ARRAY (4 DOWNTO 0) OF
vector;
SIGNAL row: matrix;
BEGIN
row(0) <= "0000" & inp;
G1: FOR i IN 1 TO 4 GENERATE
row(i) <= row(i-1)(6 DOWNTO 0) &
'0';
END GENERATE;
outp <= row(sel);
END shifter;
```

Η BLOCK

Υπάρχουν δύο εκδόσεις της δήλωσης BLOCK: η Simple και η Guarded.

Στην απλή της μορφή, πρόκειται για έναν τρόπο τοπικής τμηματοποίησης κώδικα. Επιτρέπει σε ένα σύνολο παράλληλων δηλώσεων να μαζευτούν ως ένα BLOCK ώστε ο κώδικας να γίνει πιο ευανάγνωστος και ευκολότερος στη διαχείριση. Η σύνταξή της φαίνεται στη συνέχεια:

```
label: BLOCK
[declarative part]
BEGIN
(concurrent statements)
END BLOCK label;
```

Η γενική ιδέα του κατά-μπλοκ τμηματοποιημένου κώδικα φαίνεται στη συνέχεια:

```
ARCHITECTURE example ...
BEGIN
...
block1: BLOCK
BEGIN
...
END BLOCK block1
...
block2: BLOCK
BEGIN
...
END BLOCK block2;
...
END example;
```

Παράδειγμα:

```
b1: BLOCK
SIGNAL a: STD_LOGIC;
BEGIN
a <= input_sig WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

Ένα BLOCK, (simple ή guarded) μπορεί να είναι εμφωλευμένη μέσα σε ένα άλλο BLOCK. Η αντίστοιχη σύνταξη έχει ως εξής:

```
label1: BLOCK
[declarative part of top block]
BEGIN
[concurrent statements of top block]
label2: BLOCK
[declarative part nested block]
BEGIN
(concurrent statements of nested
block)
END BLOCK label2;
[more concurrent statements of top
block]
END BLOCK label1;
```

Η guarded BLOCK είναι μια ειδική έκδοση της BLOCK, που περιέχει μια επιπρόσθετη έκφραση, την guard expression. Η προφυλαγμένη έκφραση σε μια φυλαγμένη BLOCK εκτελείται μόνο όταν η δήλωση φύλακας είναι TRUE:

```
label: BLOCK (guard expression)
[declarative part]
BEGIN
(concurrent guarded and unguarded
statements)
END BLOCK label;
```

Παρόλο που μόνο παράλληλες εκφράσεις γράφονται μέσα σε ένα BLOCK, με μια φυλαγμένη BLOCK μπορούμε να υλοποιήσουμε και ακολουθιακά κυκλώματα, αλλά αυτή δεν είναι συνήθως η μέθοδος που ακολουθούμε για τα

ακολουθιακά.