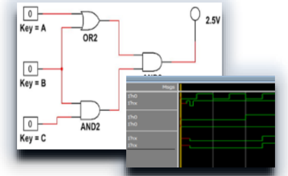
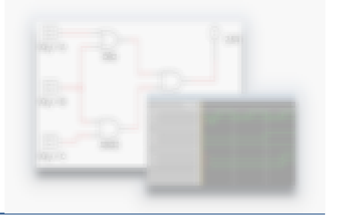


ECE119 – Ψηφιακή Σχεδίαση

Διδάσκοντες Εργαστηρίου: Δ. Καραμπερόπουλος
Δ. Γαρυφάλλου

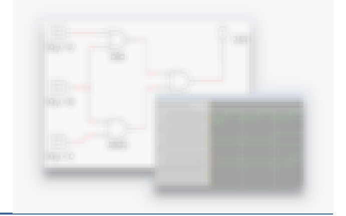
- Lab 4: Verilog (Μέρος 3)



Η Γλώσσα Verilog (Μέρος 3)

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
χρόνια δημιουργίας

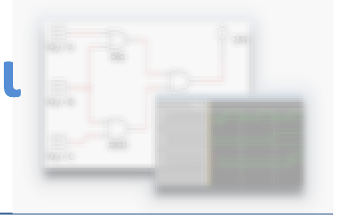
Μοντέλα HDL Συνδυαστικών κυκλωμάτων

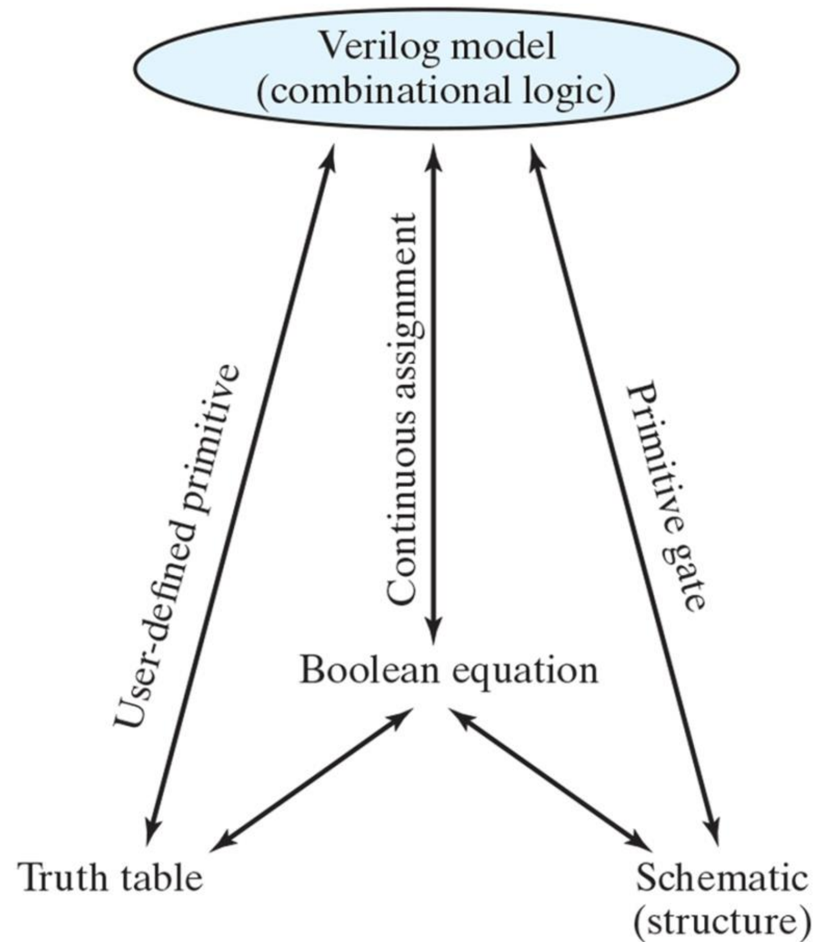


- Η βασική δομή για τη μοντελοποίηση κυκλωμάτων με χρήση της Verilog είναι η υπομονάδα (module)
- Η λογική λειτουργία μιας υπομονάδας μπορεί να περιγραφεί με ένα από τα ακόλουθα στιλ (τρόπους) μοντελοποίησης, ή με έναν συνδυασμό τους:
- **Μοντελοποίηση σε επίπεδο πυλών** (**gate-level modeling**), όπου χρησιμοποιούνται στιγμιότυπα είτε προκαθορισμένων, ή καθορισμένων από το χρήστη βασικών πυλών.
- **Μοντελοποίηση ροής δεδομένων** (**dataflow modeling**), όπου χρησιμοποιούνται εντολές διαρκούς ανάθεσης με την δεσμευμένη λέξη **assign**.
- **Μοντελοποίηση συμπεριφοράς** (**behavioral modeling**), όπου χρησιμοποιούνται διαδικαστικές εντολές ανάθεσης με την δεσμευμένη λέξη **always**.

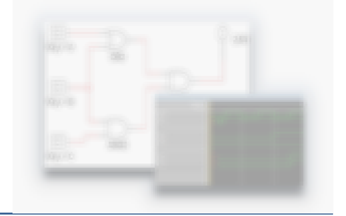
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ χρόνια δημιουργίας

Σχέση δομών της Verilog με πίνακες αληθείας, εξισώσεις Boole και σχηματικά διαγράμματα.



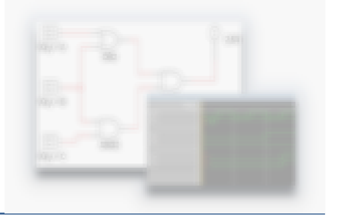


Μοντελοποίηση σε επίπεδο πυλών (gate-level)



- Το κύκλωμα προσδιορίζεται από τις λογικές πύλες του και τις διασυνδέσεις τους.
- Η Verilog περιλαμβάνει **12 βασικές πύλες** ως προκαθορισμένα, στοιχειώδη δομικά στοιχεία (primitives). Οι τέσσερις από αυτές είναι τρισταθείς.
- **and**, **nand**, **or**, **nor**, **xor** και **xnor**. Οι βασικές αυτές πύλες θεωρείται ότι έχουν **n εισόδους**.
- **buf** και **not** είναι πύλες **n εξόδων**. Επιτρέπεται μόνο **μια είσοδος** στις πύλες αυτές, η οποία όμως μπορεί να οδηγήσει πολλαπλές γραμμές εξόδου.

Μοντελοποίηση σε επίπεδο πυλών (gate-level)

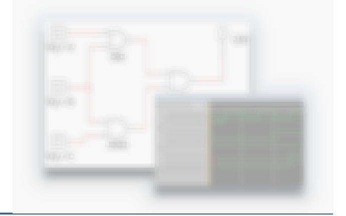


Τύποι Δεδομένων

Τιμές Σήματος σε Λογική 4-ρων τιμών (4-value logic)

Τιμή	Ερμηνεία	Χρήση
0	Λογικό 0, άρνηση	Λογικό 0
1	Λογικό 1, κατάφαση	Λογικό 1
X	Άγνωστο ή Μη αρχικοποιημένο	i. Τιμή εκκίνησης ακολουθιακών στοιχείων και σημάτων, ii. Έξοδος πύλης με εισόδους στο Z, iii. Τιμή σε περίπτωση ταυτόχρονης ανάθεσης (0 και 1)
Z	Υψηλής εμπέδησης - ασύνδετο ή τρικατάστατο	i. Τιμή μη οδηγούμενης εισόδου, ii. Έξοδος τρικατάστατου οδηγητή

Μοντελοποίηση σε επίπεδο πυλών (gate-level)



Πίνακες αληθείας των προκαθορισμένων πυλών

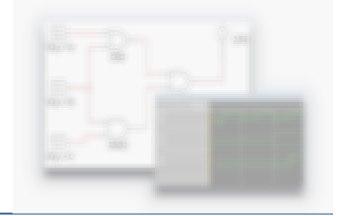
and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	input	output
	0	1
	1	0
	x	x
	z	x

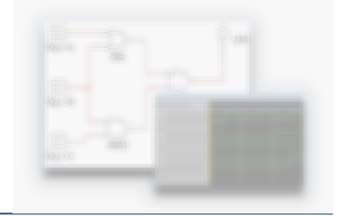
Μοντελοποίηση σε επίπεδο πυλών (gate-level)



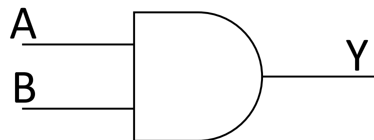
- Όταν αναφέρεται μια βασική πύλη σε μια υπομονάδα, λέμε ότι δημιουργείται ένα **στιγμιότυπο** της, ή **αντίγραφο** (a gate is instantiated) στην υπομονάδα αυτή.

- Θα μπορούσε να θεωρήσει κανείς ότι η δημιουργία ενός στιγμιότυπου στην HDL είναι η αντίστοιχη ενέργεια της τοποθέτησης ενός συγκεκριμένου εξαρτήματος σε ένα πραγματικό ηλεκτρονικό κύκλωμα και της διασύνδεσης του με τα υπόλοιπα εξαρτήματα του κυκλώματος.

Μεταβλητές τύπου NET



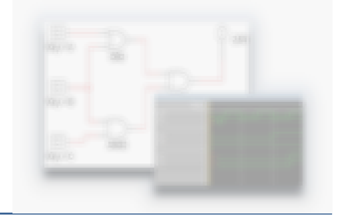
- Μπορούν να θεωρηθούν ως **καλώδια** υλικού οδηγούμενα από τη λογική
- Έχουν τιμή *z* όταν είναι ασύνδετα.
- Τύποι NET:
  - **wire**
  - **wand** (wired-AND)
  - **wor** (wired-OR)
  - **tri** (tri-state)
- Στο επόμενο παράδειγμα το Y ενημερώνεται διαρκώς και υπολογίζεται αυτόματα κάθε φορά που το A ή το B αλλάζει




```
wire Y;   // declaration
assign Y = A & B;
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ χρόνια δημιουργίας

Μεταβλητές τύπου NET



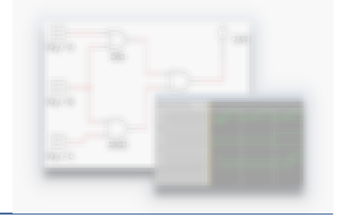
- Περιγράφουν μόνο συνδυαστική λογική
  - δεν έχουν μνήμη
  - δεν υλοποιούν στοιχεία μνήμης
- Η αξιολόγηση τους και η σημασιολογία τους αντιστοιχούν σε παράλληλες οντότητες

```
wire sum  = a ^ b;
wire c = sum | b;
wire a = ~d;
```

```
wire sum;
...
assign sum = a ^ b;
```

```
wire muxout = (sel == 1) ? a : b;
wire op = ~(a & ((b) ? ~c : d) ^ (~e));
```

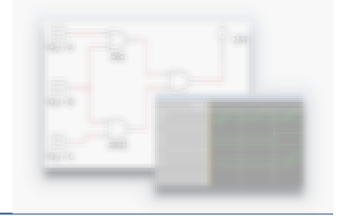
Μεταβλητές τύπου REGISTER



- Μεταβλητές με μνήμη
  - διατηρούν την κατάσταση τους μέχρι την επόμενη ανάθεση
- Μεταβλητές διαδικασιών
  - always, initial
- Δεν συνεπάγονται καταχωρητή σε επίπεδο υλικού
- Μόνο ένας τύπος:
  - **reg**

```
reg a;

initial begin
 a = 0;
 #5;
 a = 1;
end
```

```
reg q;

always @(posedge clk)
begin
  q = #2 (load) ? d : q;
end
```

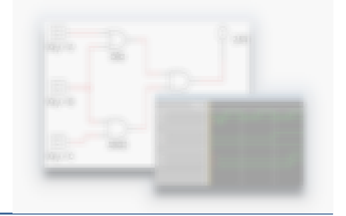


Μεταβλητές - Συνοπτικά

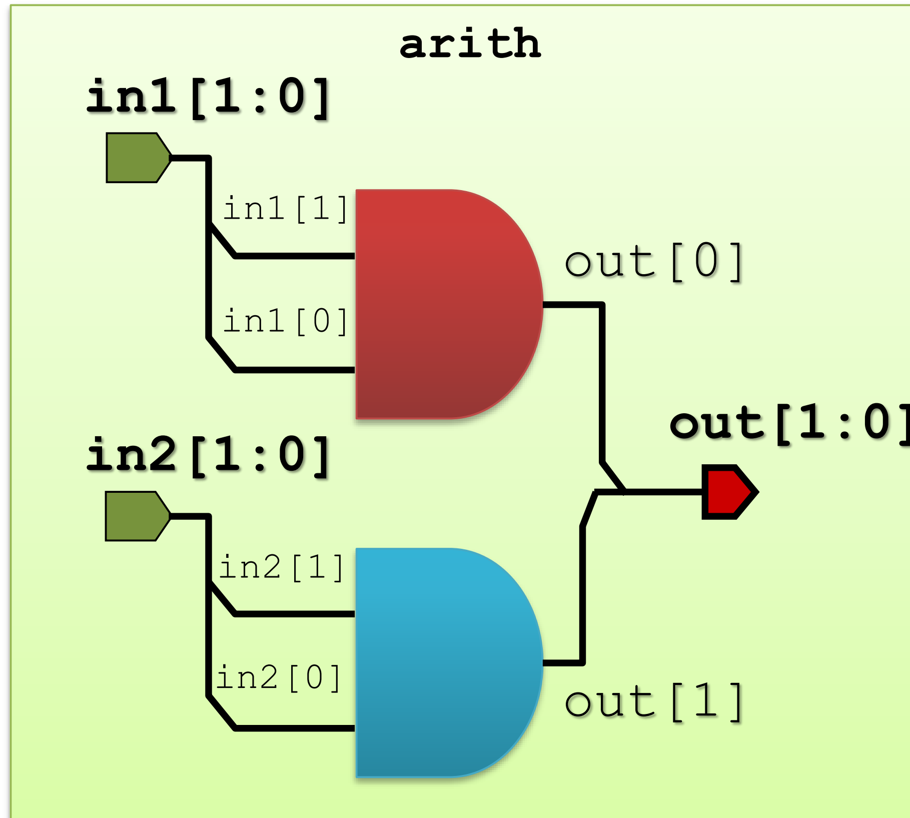
Τύπος	Ιδιότητες	Παραδείγματα

wire	Μοντελοποιεί μια σύνδεση, «καλώδιο», η οποία δομικά διασύνδεει δυο σήματα	`wire Net1;`
		`wire [2:0] fout;`
		`assign Net1 = 1’b1;`

reg	Αποθηκεύει τιμή ανάθεσης από διαδικασία, κρατώντας την για κύκλο «δέλτα» ή μέχρι την επόμενη ανάθεση. Δεν συνεπάγεται απαραιτήτως σύνθεση σε καταχωρητή.	`reg [3:0] Y1, Y2;`
parameter	Σταθερά. Πρέπει να είναι ακέραια τιμή για σύνθεση.	`parameter A=4’b1011, B=4’b1000;` `parameter Stop=0, Slow=1, Medium=2, Fast=3;`
integer	Ακέραια μεταβλητή για χρήση σε βρόχους. Δεν έχουν απεικόνιση στο υλικό και κρατάνε απλά αριθμητικές τιμές.	`integer N;`

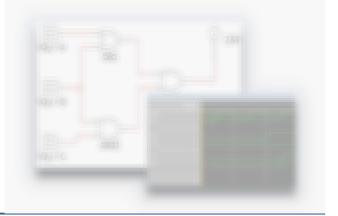


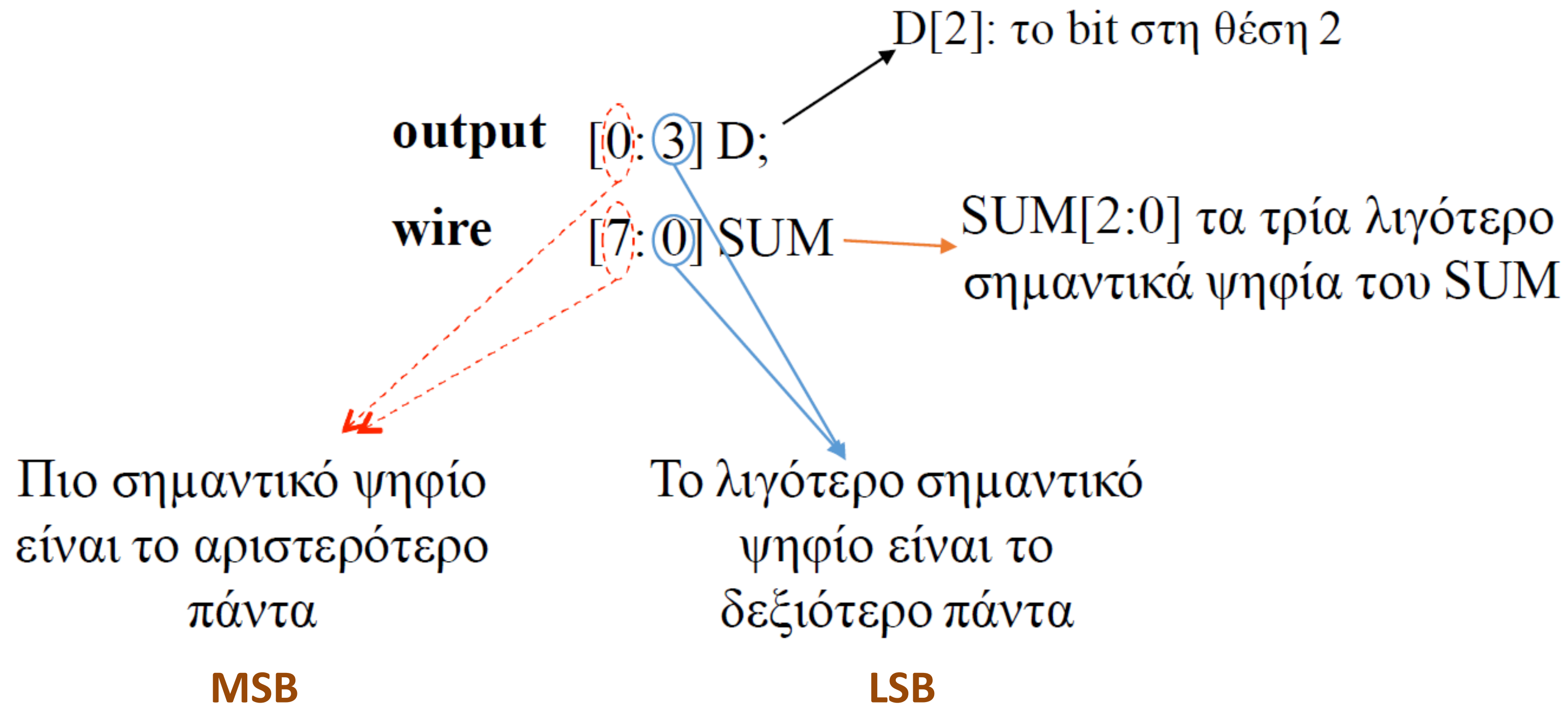
Δίαυλοι (Busses) ή Διανύσματα (Vectors)



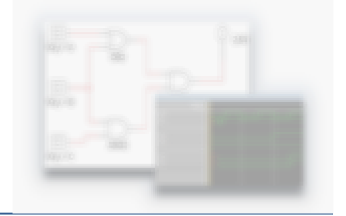

```
module arith (out, in1,
in2);
output [1:0] out;
input [1:0] in1, in2;
…
…
endmodule
```

Δίαυλοι (Busses) ή Διανύσματα (Vectors)





Για εύκολο και σίγουρο κώδικα/design: wire XYZ [MSB : LSB]



Δίαυλοι (Busses) ή Διανύσματα (Vectors)

- Represent buses

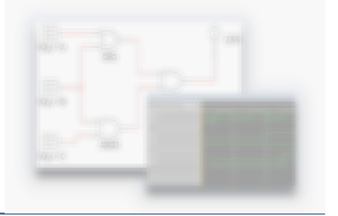
```
wire  [3:0] busA;
reg   [1:4] busB;
reg   [1:0] busC;
```

- Left number is MS bit
- Slice management

```
busC = busA[2:1];   ⇔   { busC[0] = busA[1];
                          busC[1] = busA[2];
```

- Vector assignment (by position!!)

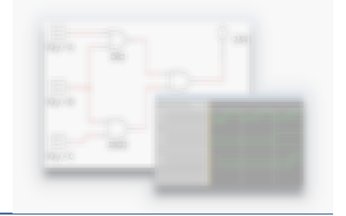
```
                  { busB[4] = busA[0];
                    busB[3] = busA[1];
busB = busA;  ⇔     busB[2] = busA[2];
                    busB[1] = busA[3];
```



Δίαυλοι (Busses) ή Διανύσματα (Vectors)

```
input   [3:0] a;       // 4 bits
output        b;       // 1 bit

a = 4'b1101;


b = a[0];             -> b=1
b = a[1];             -> b=0
b = a[2];             -> b=1
b = a[3];             -> b=1
```

Δίαυλοι (Busses) ή Διανύσματα (Vectors)

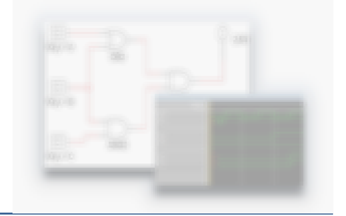


```
module example (a, b, out1, out2, out3, out4);
input   [3:0]   a, b;        // 4 bits
output          out1;        // 1 bit
output  [1:0]   out2;        // 2 bits
output  [2:0]   out3;        // 3 bits
output  [3:0]   out4;        // 4 bits

assign out1 = a[1];          // 1 bit
assign out2 = a[2:1];        // 2 bits
assign out3 = b[3:1];        // 3 bits
assign out4 = b;             // 4 bits

endmodule
```

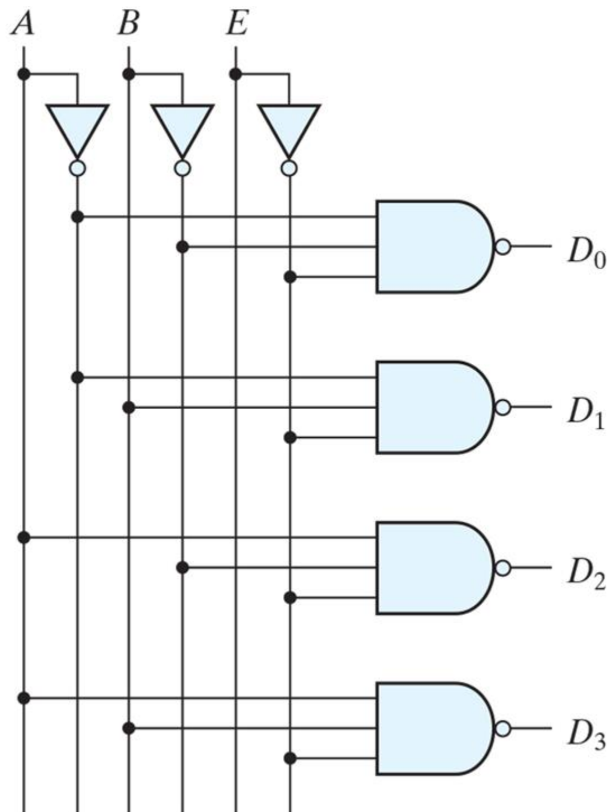
```
module t_example;
reg   [3:0]   a, b;
wire          out1;
wire  [1:0]   out2;
wire  [2:0]   out3;
wire  [3:0]   out4;

example    dut(a, b, out1, out2, out3, out4);

initial begin
    a=4'b1101;
    b=4'b1100;
end
initial $monitor("a = %b  b = %b  out1 = %b out2 = %b
out3 = %b out4 = %b", a, b, out1, out2, out3, out4);
endmodule
```

```
a = 1101  b = 1100  out1 = 0 out2 = 10 out3 = 110 out4 = 1100
```

Δίαυλοι (Busses) ή Διανύσματα (Vectors)



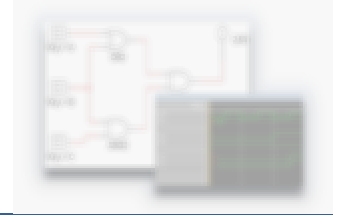
- **Παράδειγμα: Αποκωδικοποιητής 2-σε-4**



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

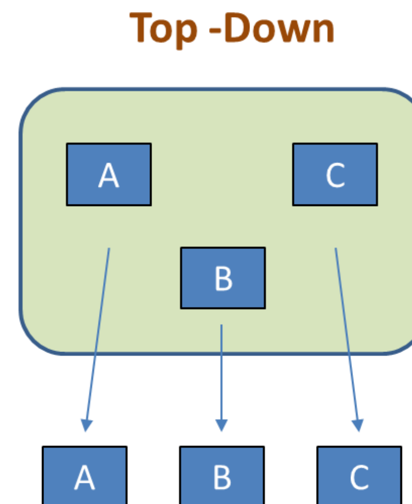
```
module decoder_2x4_gates (D, A, B, enable);
 output [0: 3]      D;
 input              A, B;
 input              enable;
 wire               A_not, B_not, enable_not;
not
   G1  (A_not, A),  // Comma-separated list of primitives
   G2  (B_not, B),
   G3  (enable_not, enable);
nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Ιεραρχική περιγραφή

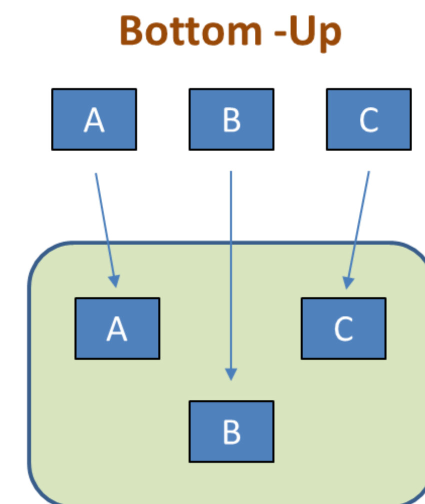


- Μπορούμε να συνδυάσουμε δύο ή περισσότερες υπομονάδες για να συνθέσουμε μια **ιεραρχική περιγραφή** ενός ευρύτερου υπό σχεδίαση κυκλώματος.
- **Top-down** - **Bottom-up**
- Τα επιμέρους κυκλώματα και οι περιγραφές τους αποκαλούνται επίσης **δομικά** ή **κυκλωματικά μπλοκ** [**building blocks**].

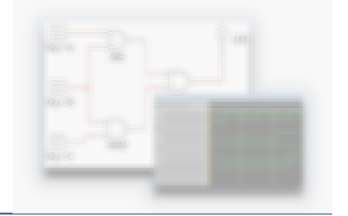
Top -Down



Bottom -Up



Ιεραρχική περιγραφή



- Προκειμένου να δημιουργήσουμε μια ιεραρχική δομή στη σχεδίασή μας απαιτείται η υλοποίηση υποσχεδιάσεων "**χαμηλότερου**" επιπέδου (υπομονάδες) και η περίληψή τους εντός της περιγραφής του "**υψηλότερου**" (top level) επιπέδου.

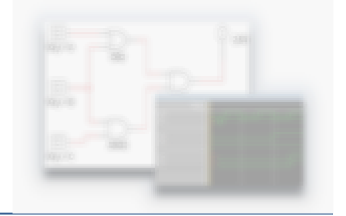
- Η ιεραρχική σχεδίαση θεωρείται άκρως σημαντική και αποτελεσματική, καθώς μας δίνει την δυνατότητα να διαχωρίσουμε την σχεδίαση σε επιμέρους τμήματα, με ό,τι αυτό συνεπάγεται.

- Μια πλήρως ιεραρχική περιγραφή περιλαμβάνει μόνο τις αναφορές στις υπομονάδες και την περιγραφή των μεταξύ τους συνδέσεων.

- Μια υπομονάδα στην Verilog είναι κατ'ουσίαν ένα απλό Verilog module.

- Όλες οι υπομονάδες (sub-modules), όπως και στην περίπτωση των "απλών" εντολών καλούνται και "εκτελούνται" ταυτόχρονα.

Ιεραρχική περιγραφή



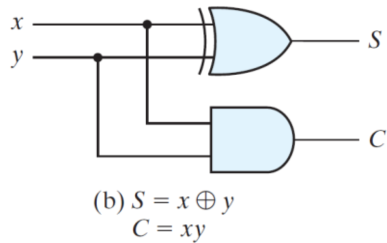
- **Lower-level module instantiation:**

module_name <instance_identifier> (port mapping);

  - **module_name**: το όνομα του module που καλείται, πρέπει να είναι όμοιο με το όνομα που έχει δηλωθεί το lower-level module.
  - **instance_identifier**: προαιρετικό, χρησιμοποιείται κυρίως στην περίπτωση πολλαπλών κλήσεων του ίδιου lower-level module.
  - **port_mapping**: καθορισμός του τρόπου διασύνδεσης μεταξύ των modules, υπάρχουν δύο προσεγγίσεις:
    - **Positional**: τα σήματα τα οποία πρόκειται να χρησιμοποιηθούν για την σύνδεση με τα lower level modules αναγράφονται <u>με την ίδια σειρά</u> που καθορίζονται τα ports στο sub-module.
    - **Explicit**: τα ονόματα των ports από τα lower-level modules χρησιμοποιούνται σε συνδυασμό με τα σήματα στα οποία συνδέονται. <u>Μπορούμε να ακολουθήσουμε οποιαδήποτε σειρά αναφοράς τους</u>.

Ιεραρχική περιγραφή

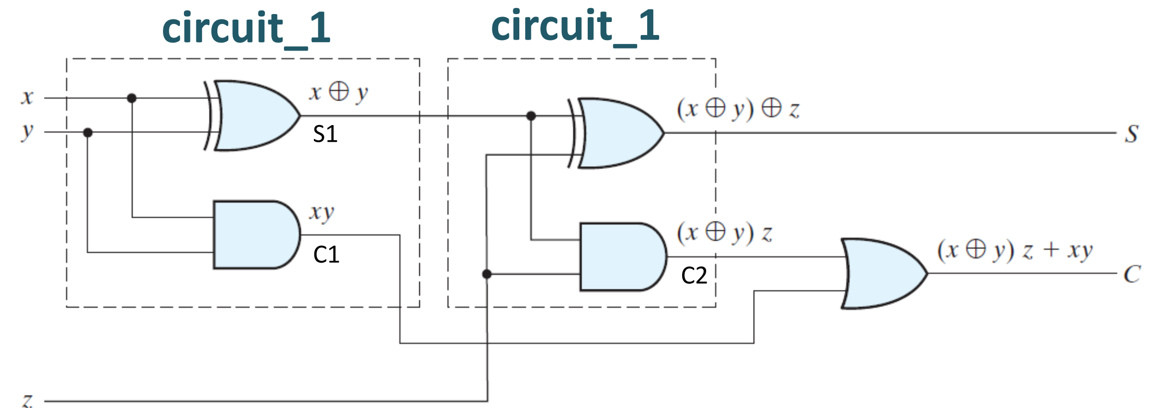
- Παράδειγμα: **Positional** port mapping:



(b) $S = x \oplus y$
$C = xy$

sub-module

```
module circuit_1 (S, C, x, y);
output   S, C;
input    x, y;

  xor    (S, x, y);
  and    (C, x, y);
endmodule
```

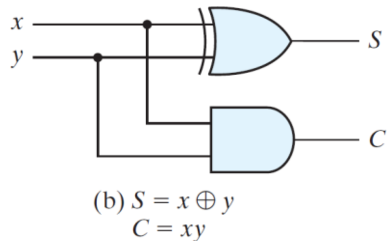


top-level

```
module top_circuit (S, C, x, y, z);
output    S, C;
input     x, y, z;
wire      S1, C1, C2;
// Instantiate circuit_1
circuit_1    Cir1(S1, C1, x, y);
circuit_1    Cir2(S, C2, S1, z);

or        G1 (C, C2, C1);
endmodule
```

Ιεραρχική περιγραφή

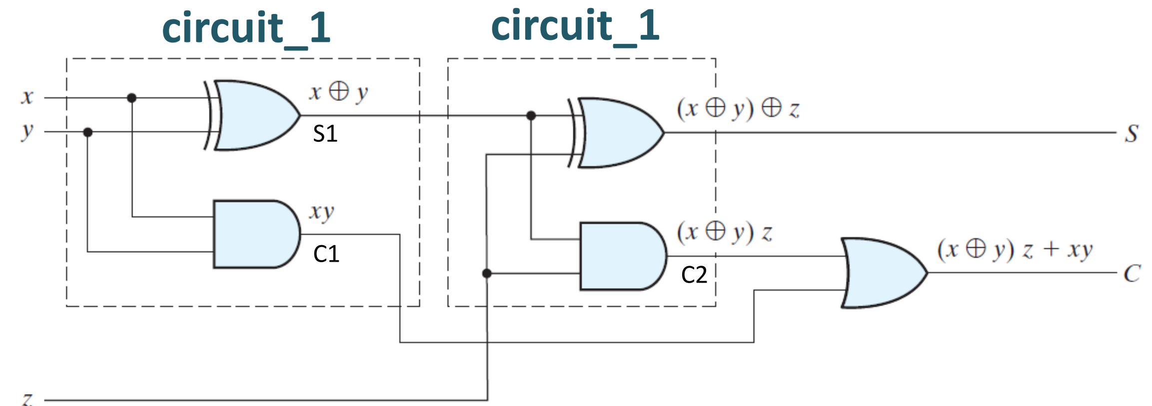
- Παράδειγμα: **Explicit** port mapping:



sub-module

```
module circuit_1 (S, C, x, y);
output    S, C;
input     x, y;

  xor     (S, x, y);
  and     (C, x, y);
endmodule
```

circuit_1 circuit_1

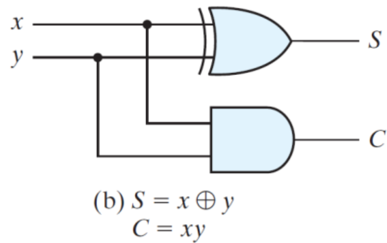


top-level

```
module top_circuit (S, C, x, y, z);
output    S, C;
input     x, y, z;
wire      S1, C1, C2;
// Instantiate circuit_1
circuit_1    Cir1(.S(S1), .C(C1), .x(x), .y(y));
circuit_1    Cir2(.S(S), .C(C2), .x(S1), .y(z));

or        G1 (C, C2, C1);
endmodule
```

Ιεραρχική περιγραφή

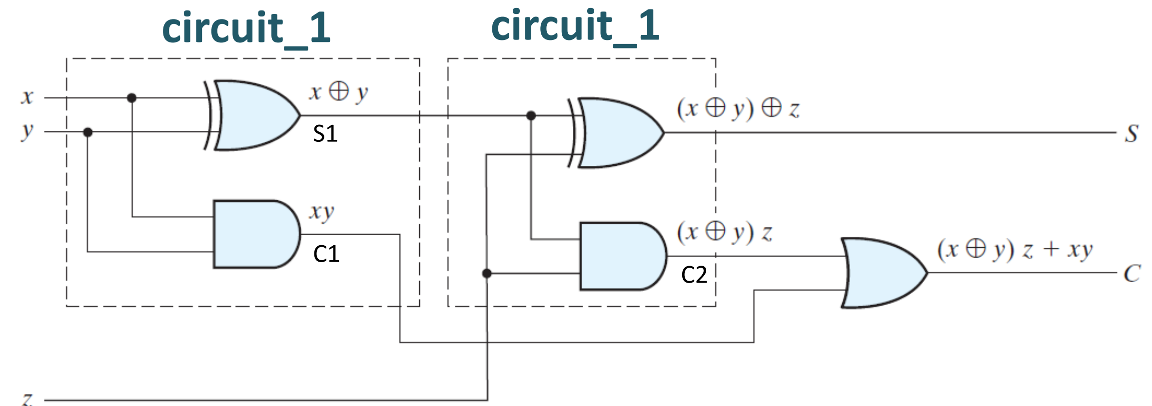
- Παράδειγμα: **Explicit** port mapping:



(b) $S = x \oplus y$
$C = xy$

sub-module

```
module circuit_1 (S, C, x, y);
output    S, C;
input     x, y;

  xor     (S, x, y);
  and     (C, x, y);
endmodule
```



top-level

```
module top_circuit (S, C, x, y, z);
output    S, C;
input     x, y, z;
wire      S1, C1, C2;
// Instantiate circuit_1
circuit_1    Cir1(.x(x), .y(y), .C(C1), .S(S1));
circuit_1    Cir2(.C(C2), .S(S), .x(S1), .y(z));

or        G1 (C, C2, C1);
endmodule
```

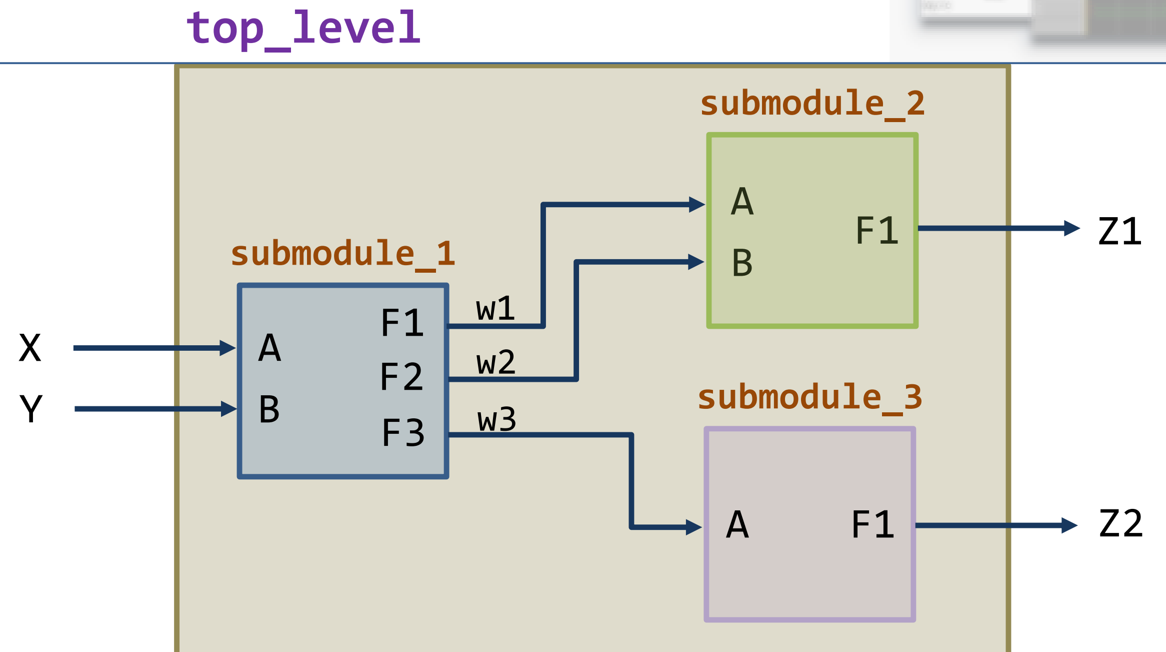
Ιεραρχική περιγραφή

- Παράδειγμα: **Positional** port mapping:

```
module submodule_1 (F1, F2, F3, A, B);
output   F1, F2, F3;
input    A, B;
// code…
endmodule
```

```
module submodule_2 (F1, A, B);
output   F1;
input    A, B;
// code…
endmodule
```

```
module submodule_3 (F1, A);
output   F1;
input    A;
// code…
endmodule
```




```
module top_level (Z1, Z2, X, Y);
output   Z1, Z2;
input    X, Y;
wire     w1, w2, w3;
submodule_1 m1(w1, w2, w3, X, Y);
submodule_2 m2(Z1, w1, w2);
submodule_3 m3(Z2, w3);
endmodule
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
χρόνια δημιουργίας

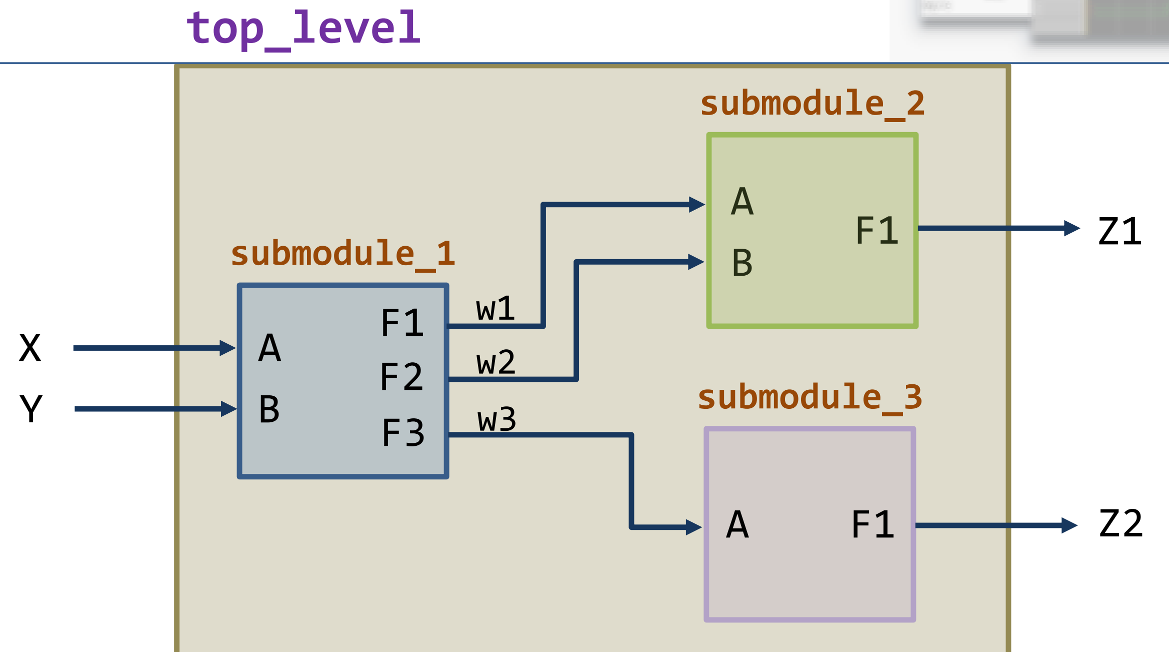
Ιεραρχική περιγραφή

- Παράδειγμα: **Explicit** port mapping:

```
module submodule_1 (F1, F2, F3, A, B);
output   F1, F2, F3;
input    A, B;
// code…
endmodule
```

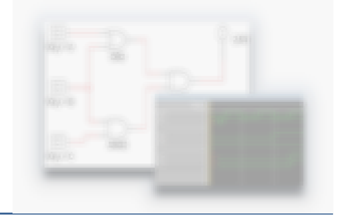
```
module submodule_2 (F1, A, B);
output   F1;
input    A, B;
// code…
endmodule
```

```
module submodule_3 (F1, A);
output   F1;
input    A;
// code…
endmodule
```




```
module top_level (Z1, Z2, X, Y);
output   Z1, Z2;
input    X, Y;
wire     w1, w2, w3;
submodule_1 m1(.F1(w1), .F2(w2), .F3(w3), .A(X), .B(Y));
submodule_2 m2(.A(w1), .B(w2), .F1(Z1));
submodule_3 m3(.A(w3), .F1(Z2));
endmodule
```

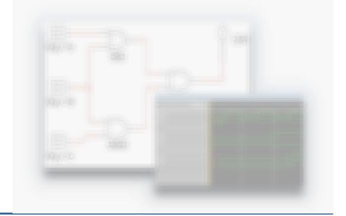
Τελεστές - Αριθμητικοί



		Αριθμητικοί
Τελεστής	**Χρήση**	**Περιγραφή**
+	m + n	Πρόσθεσε m και n
-	m – n	Αφαίρεσε n από m
-	-m	Συμπλήρωμα/Άρνηση του m (2’s complement)
*****	m * n	Πολλαπλασίασε m και n
/	m / n	Διαίρεση m με n
%	m % n	Υπόλοιπο Διαίρεσης m με n

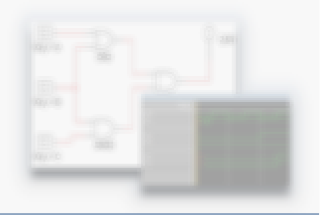
- Οι ποσότητες (τελεστέοι) πάνω στις οποίες εφαρμόζονται αριθμητικοί τελεστές είναι αριθμοί.
- Ο τελεστής ακέραιου υπόλοιπου (modulus ή modulo) δίνει το υπόλοιπο της ακέραιας διαίρεσης δύο αριθμών. Π.χ. 14 % 3 = 2

Τελεστές - Επιπέδου bit (bitwise operators)



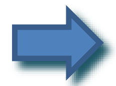
		Επιπέδου bit
~	~m	Αντέστρεψε κάθε ψηφίο του m
&	m & n	AND κάθε ψηφίου των m και n
|	m | n	OR κάθε ψηφίου των m και n
^	m ^ n	XOR κάθε ψηφίου των m και n
~^ **^~**	m ~^ n m ^~ n	XNOR κάθε ψηφίου των m και n

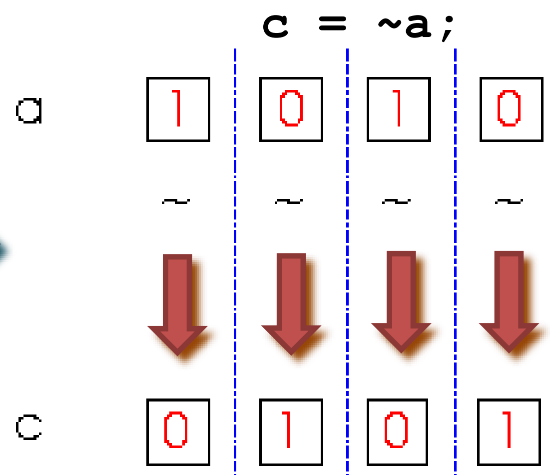
- Εκτελούν την ίδια λογική πράξη σε κάθε ζεύγος bit αντίστοιχης τάξης των δύο λέξεων (διανύσματα από bit)
- Το σχετικό αποτέλεσμα είναι επίσης ένα διάνυσμα από bit.
- Η άρνηση (~) είναι τελεστής με ένα όρισμα μόνο, δρα σε μία μόνο λέξη (διάνυσμα), συμπληρώνοντας τα bit του διανύσματος.

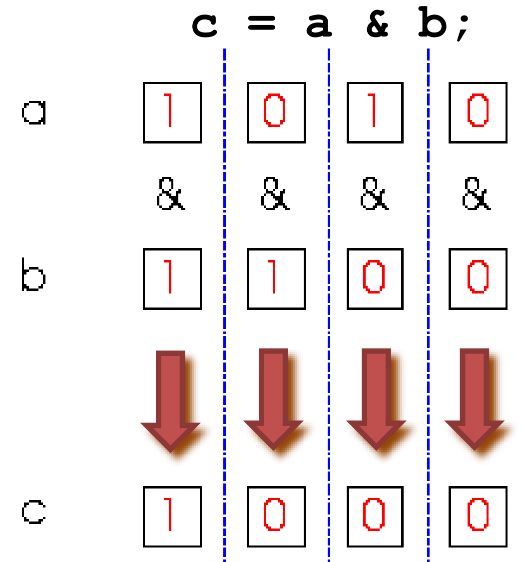


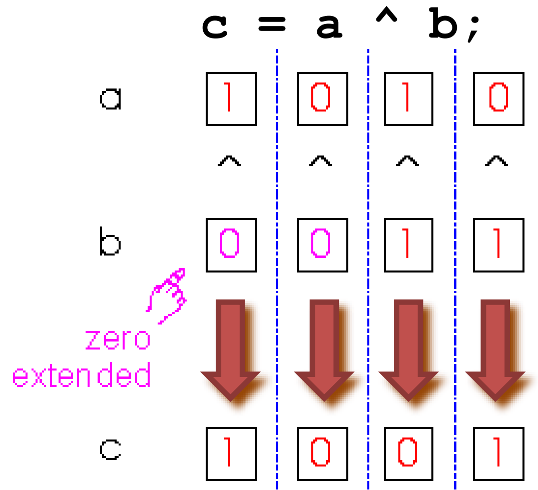
Τελεστές - Επιπέδου bit (bitwise operators)

```
a = 4'b1010;
b = 4'b1100;
```

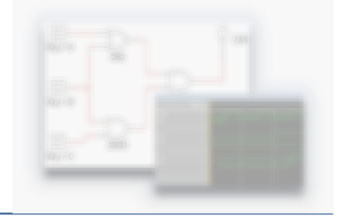








```
a = 4'b1010;
b = 2'b11;
```



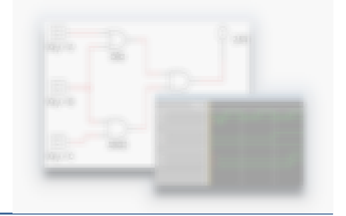
Τελεστές - Ελάττωσης

Ελάττωσης		
Τελεστής	**Χρήση**	**Περιγραφή**
&	&m	AND όλων των ψηφίων του m (1-bit αποτέλεσμα)
~&	~&m	NAND όλων των ψηφίων του m (1-bit αποτέλεσμα)
|	|m	OR όλων των ψηφίων του m (1-bit αποτέλεσμα)
~|	~|m	NOR όλων των ψηφίων του m (1-bit αποτέλεσμα)
^	^m	XOR όλων των ψηφίων του m (1-bit αποτέλεσμα)
~^ **^~**	~^m ^~m	XNOR όλων των ψηφίων του m (1-bit αποτέλεσμα)

- Πράξεις με έναν τελεστέο. Μπορεί να είναι 1-bit ή διάνυσμα (multi-bit).
- Το αποτέλεσμα της πράξης είναι 1-bit.

```
a = 4’b1001;
..
c = |a;        // c = 1|0|0|1 = 1
```

Τελεστές - Λογικοί

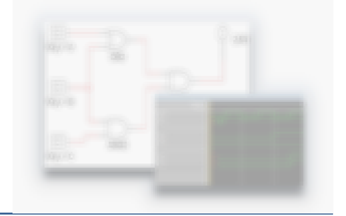


		Λογικοί
!	!m	NOT: Είναι το m ψευδές; (1-bit αποτέλεσμα)
&&	m && n	AND: Είναι το m και το n αληθή; (1-bit αποτέλεσμα)
||	m || n	OR: Είναι το m ή το n αληθές; (1-bit αποτέλεσμα)

- Οι τελεστέοι αξιολογούνται σαν 1-bit: 0, 1, x
- Το αποτέλεσμα της πράξης είναι 1-bit: 0, 1, x

```
A = 5;
B = 0;
C = x;
```

```
A && B   → 1 && 0   → 0
A || !B  → 1 || 1   → 1
C || B   → x || 0   → x
```



Τελεστές - Ισότητας, Ανίσωσης

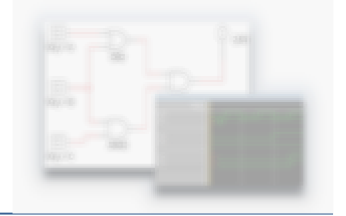
Ισότητας, Ανίσωσης		
Τελεστής	**Χρήση**	**Περιγραφή**
==	m == n	Είναι το m ίσο με το n; (1-bit αποτέλεσμα)
!=	m != n	Είναι το m διάφορο του n; (1-bit αποτέλεσμα)
> **>=**	m > n m >= n	Είναι το m μεγαλύτερο του n; Είναι το m μεγαλύτερο ή ίσο του n; (1-bit αποτέλεσμα)
< **<=**	m < n m <= n	Είναι το m μικρότερο του n; Είναι το m μικρότερο ή ίσο του n; (1-bit αποτέλεσμα)

- Το αποτέλεσμα της πράξης είναι 1-bit: 0, 1, x

```
1 > 0          → 1
3'b1x1 <= 0    → x
10 < z         → x
```

```
4'b0011 == 4'b0011 → 1
4'b0011 != 4'b0011 → 0
4'b1z0x == 4'b1z0x  → x
4'b1z0x != 4'b1z0x  → x
```

Τελεστές - Μετατόπισης ή ολίσθησης

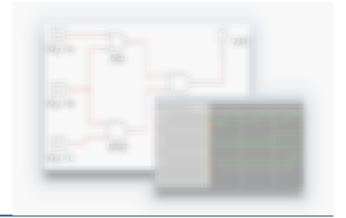


	Μετατόπισης	
<<	m << n	Μετατόπισε το m αριστερά n φορές
>>	m >> n	Μετατόπισε το m δεξιά n φορές

- Το αποτέλεσμα της πράξης έχει το ίδιο μέγεθος (σε bit)
- Οι προκύπτουσες κενές θέσεις συμπληρώνονται με μηδενικά

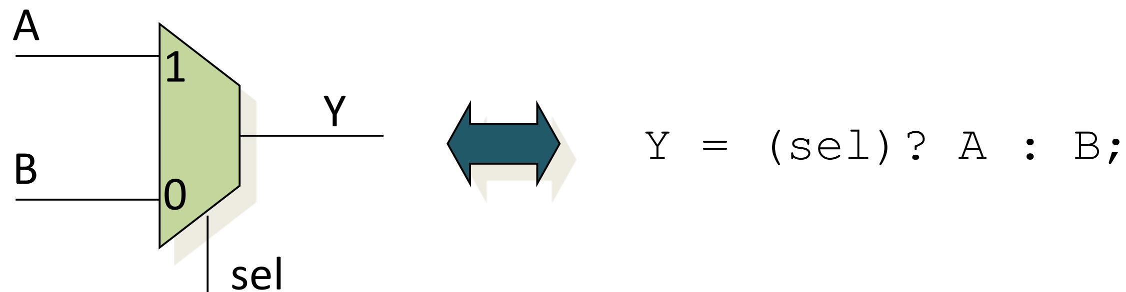
```
a = 4'b1010;
...
d = a >> 2;   // d = 0010
c = a << 1;   // c = 0100
```

Τελεστές - Υπό συνθήκη

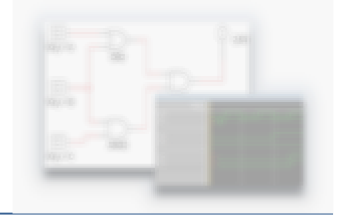


Conditional operator		
Τελεστής	Χρήση	Περιγραφή
? :	sel ? m : n	(Υπό συνθήκη, Conditional operator) Αν το sel είναι αληθές επέστρεψε m αλλιώς n

- **cond_expr ? true_expr : false_expr**
- Like a 2-to-1 mux ..

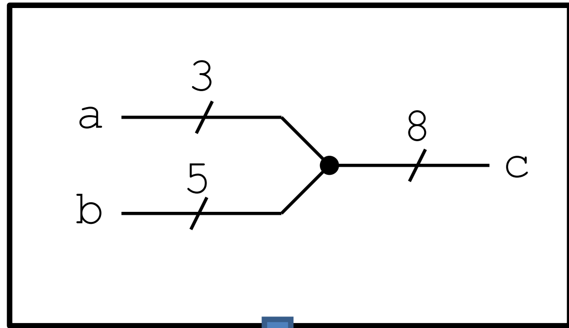



```
Y = (sel)? A : B;
```



Τελεστές - Συνένωσης

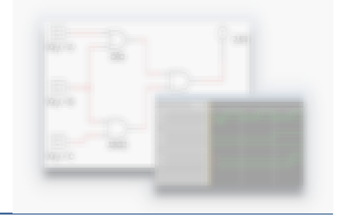
Τελεστής συνένωσης		
Τελεστής	**Χρήση**	**Περιγραφή**
{}	{m, n}	(Τελεστής συνένωσης, Concatenation operator) Ένωσε τα διανύσματα m και n επιστρέφοντας την συνένωση τους



```
wire [2:0] a;
wire [4:0] b;

wire [7:0] c = {a , b};
```

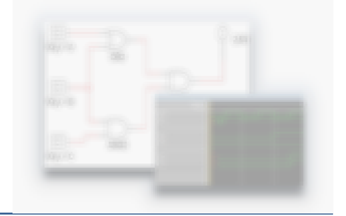
```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};          // catx = 1_010_101  (1010101)
caty = {b, 2'b11, a};      // caty = 010_11_1   (010111)
```

Τελεστές - Replication



Διάφοροι		
Τελεστής	**Χρήση**	**Περιγραφή**
{{ }}	{n{m}}	(Τελεστής αντιγραφής, Replication) Επανάλαβε το διάνυσμα m ν φορές

```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catr = {{4{a}}, b, {2{c}}};        // catr = 1111_010_101101
```

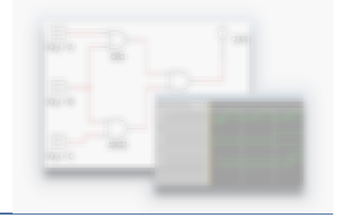


Μοντελοποίηση ροής δεδομένων (dataflow)

- Χρησιμοποιεί τελεστές που επενεργούν σε μαθηματικές ποσότητες (τελεστέους) και παράγουν δυαδικά αποτελέσματα.
- Με την εντολή **assign**.
- Λέγεται και **Διαρκής Ανάθεση**
  - π.χ. Assign Y = (A && B) || C
- Η Verilog παρέχει περίπου 30 διαφορετικούς τελεστές.

Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		



Μοντελοποίηση ροής δεδομένων (dataflow)

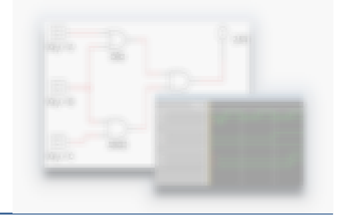
- Σύνταξη:

```
assign  #delay  <id>  =  <expr>;
```

- Που μπορούμε να τοποθετήσουμε την εντολή assign:
  - Μέσα σε ένα module
  - Έξω από Διαδικασίες

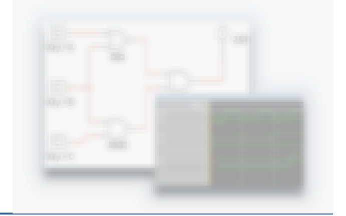
- Χαρακτηριστικά:
  - Εκτελούνται όλες παράλληλα
  - Δεν έχει σημασία η σειρά γραφής τους
  - Είναι συνεχώς ενεργές

Μοντελοποίηση ροής δεδομένων (dataflow)



- Οι εξισώσεις Boole που περιγράφουν **συνδυαστική λογική** γράφονται με μία εντολή διαρκούς ανάθεσης, την **assign**.
  - π.χ. **assign D** = (A && B) || (!C);
  - Ο προσομοιωτής εντοπίζει τις χρονικές στιγμές αλλαγής τιμής μίας ή περισσοτέρων εισόδων (A, B, C) και κάθε φορά που συμβαίνει αυτό, ενημερώνει την τιμή της D.
- Προσδιορίζει μία **μόνιμη σχέση** μεταξύ των μεταβλητών (A, B, C) και της ανατιθέμενης τιμής (D).
- Θεωρούμε ότι έχει ένα ισοδύναμο λογικό κύκλωμα σε επίπεδο πυλών. Αυτό ονομάζεται **υποκρυπτόμενη συνδυαστική λογική** (implicit combinatorial logic).

Μοντελοποίηση ροής δεδομένων (dataflow)



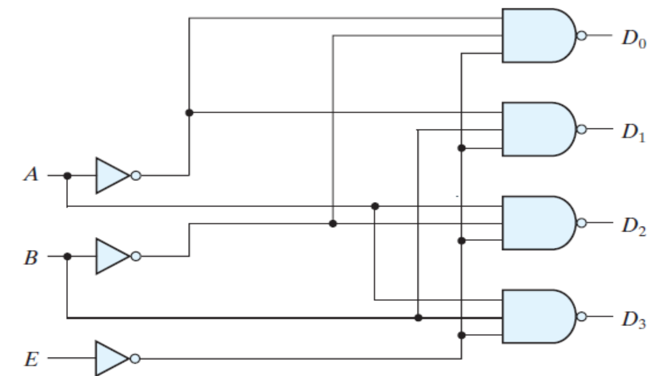
- Τα HDL μοντέλα **ροής δεδομένων** περιγράφουν συνδυαστικά κυκλώματα **με βάση τη λειτουργία τους** <u>και όχι </u>με βάση τη δομή των πυλών τους.

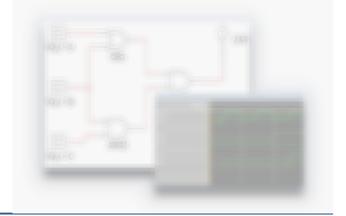
```
module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]         D;
 input                      A, B;
 input                      enable;
 wire                       A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Gate-Level

```
module decoder_2x4_df (                          // Verilog 2001, 2005 syntax
   output    [0: 3]        D,
   input                   A, B,
                           enable
);
   assign    D[0] = !((!A) && (!B) && (!enable)),
             D[1] = !(*!A) && B && (!enable)),
             D[2] = !(A && B && (!enable),
             D[3] = !(A && B && (!enable));
endmodule
```

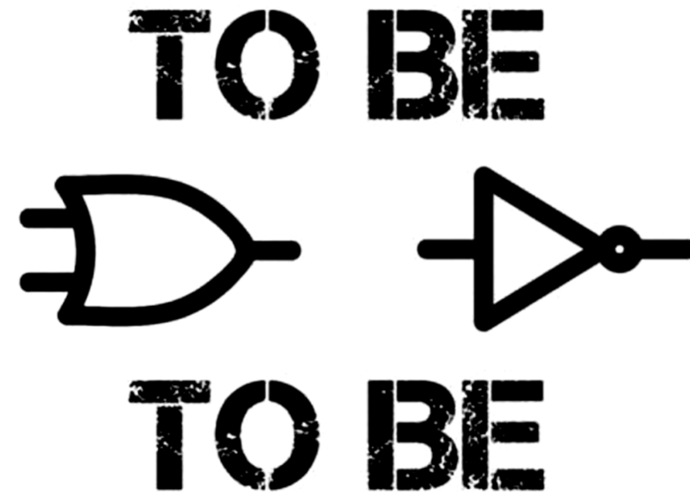
Dataflow





Ευχαριστώ για την προσοχή σας!

- **Ερωτήσεις / Απορίες ;**



Επικοινωνία: ece119.uth@gmail.com