ΗΥ430 – Εργαστήριο Ψηφιακών Κυκλωμάτων

Διδάσκων: Χ. Σωτηρίου, Βοηθός: (θα ανακοινωθεί)

http://inf-server.inf.uth.gr/courses/CE430/

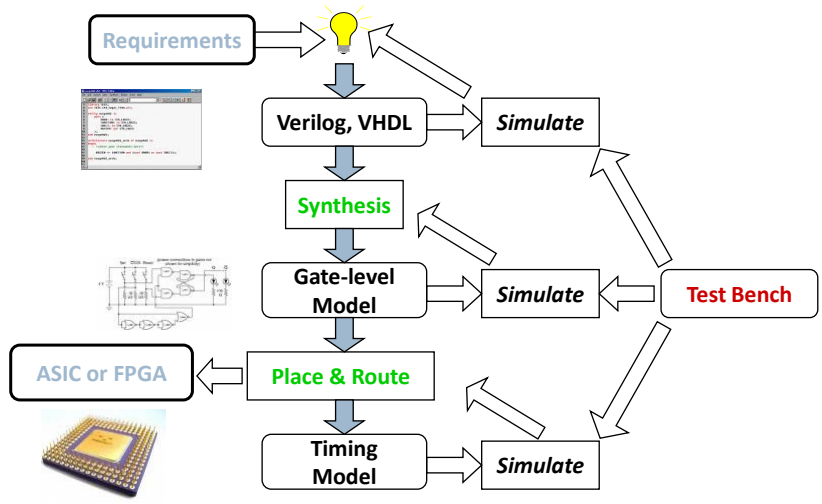
Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Τυπική Ροή Σχεδίασης (Design Flow)



Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Ιεραρχία στην Σχεδίαση

- **Top-Down** (**Επάνω προς Κάτω**) ή **Bottom-Up** (**Κάτω προς επάνω**)
  - Πρακτικά γίνεται μίξη των δυο
- Τελικό σύστημα αποτελείται από τα Τερματικά Τμήματα ή φύλλα (Leaf blocks) που λειτουργούν όλα παράλληλα.

Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

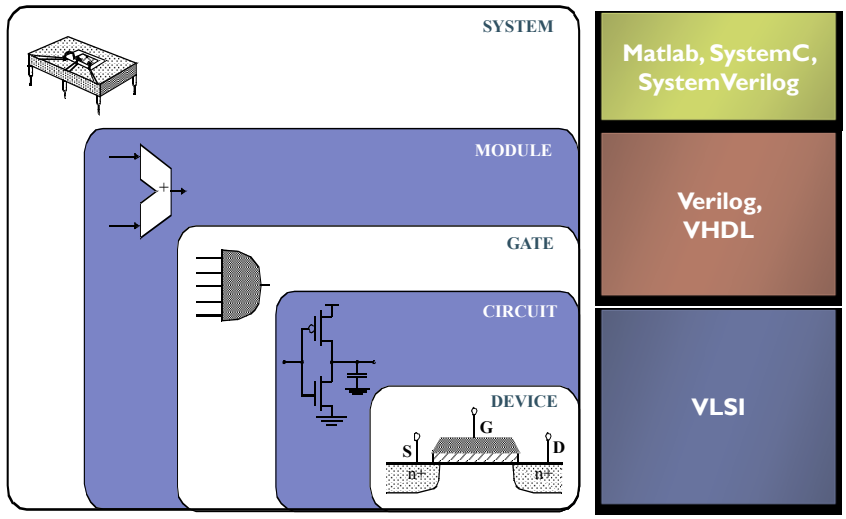
Η Γλώσσα Verilog

- Γλώσσα Περιγραφής Υλικού (HDL)
  - Γλώσσα προγραμματισμού με υποδομές για υλοποίηση υλικού
    - Έννοια του χρόνου, έννοια του σήματος
- Δυνατότητες
  1. να αναπαριστά (σε διάφορα επίπεδα) και
  2. να προσομοιώνει ψηφιακά κυκλώματα.
  3. ένα υποσύνολο της είναι συνθέσιμο (HDL→κυκλωματική δομή)
- Υποστηρίζει
  - Παράλληλη εκτέλεση τμημάτων υλικού και παράλληλες διαδικασίες
  - Σημασιολογία (semantics) για χρόνο και τιμές σημάτων
- Παραδείγματα σχεδίασης με Verilog HDL
  - Intel Pentium, AMD K5, K6, Athlon, ARM7, etc
  - Thousands of ASIC designs using Verilog HDL
- Άλλες HDL :VHDL, SystemC, SystemVerilog

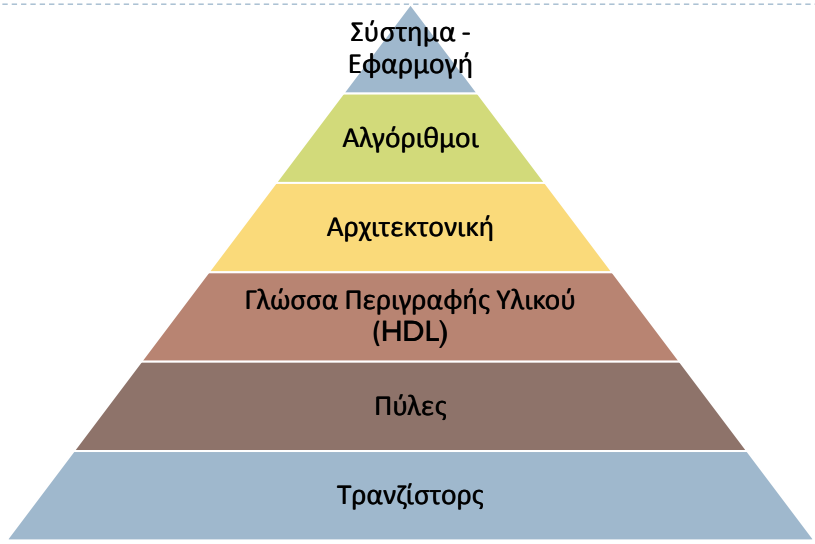
Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Επίπεδα Αφαίρεσης



Επίπεδα Αφαίρεσης



Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Αναπαράσταση και Υλοποίηση σε Verilog

- Η Verilog μπορεί να χρησιμοποιηθεί σε διάφορα στάδια για την υλοποίηση ενός συστήματος από ιδέα σε κύκλωμα
- **Δυνατότητες:**
  - Ορισμός Απαιτήσεων (Requirements Specification)
  - Έγγραφη Τεκμηρίωση (Documentation)
  - Έλεγχος μέσω Προσομοίωσης (Simulation)
  - Λειτουργικός Έλεγχος (Functional Test)
  - Συνθεσιμότητα σε Σχηματικό, δηλ. σύνολο από πύλες
- **Στόχοι**
  - Αξιόπιστη διεργασία σχεδίασης με χαμηλές απαιτήσεις κόστους και χρόνου
  - Αποφυγή και πρόληψη λαθών σχεδίασης

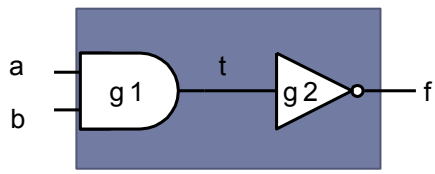
Περιεχόμενα

Μονάδα NAND – Ορισμός dataflow (ροής)


```
module nand(a, b, f);
input a, b;
output f;

endmodule
```

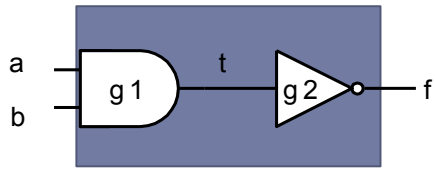

Μονάδα NAND – Ορισμός dataflow (ροής)




```
module nand(a, b, f);
input a, b;
output f;

endmodule
```

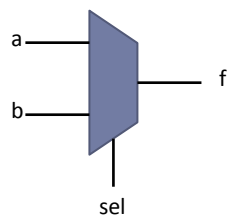
Μονάδα NAND – Ορισμός dataflow (ροής)




```
module nand(a, b, f);
input a, b;
output f;

wire t;
assign t = a & b;
assign f = ~t;
endmodule
```


Πολυπλέκτης – Ορισμός dataflow

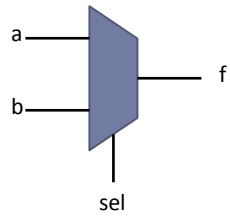



```
module mux(a, b, sel, f);
input  a, b, sel;
output f;

endmodule
```

a	b	sel	f
x	y	0	x
x	y	I	y

Πολυπλέκτης – Ορισμός dataflow



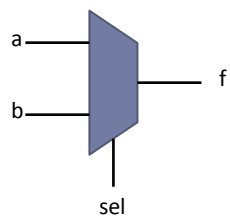

```
module sel(a, b, sel, f);
input  a, b, sel;
output f;

assign f = sel ? b : a;

endmodule
```

a	b	sel	f
x	y	0	x
x	y	1	y

Πολυπλέκτης – Ορισμός συμπεριφοράς

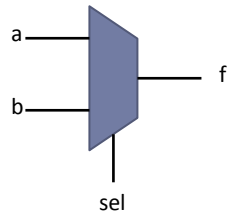



```
module sel(a, b, sel, f);
input  a, b, sel;
output f;
reg f;
always @(sel or a or b)
begin

end
endmodule
```

a	b	sel	f
x	y	0	x
x	y	1	y

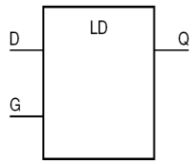
Πολυπλέκτης – Ορισμός συμπεριφοράς



a	b	sel	f
x	y	0	x
x	y	I	y

```
module sel(a, b, sel, f);
input  a, b, sel;
output f;
reg f;
always @(sel or a or b)
begin
  if (sel==0)
    f <= a;
  else
    f <= b;
end
endmodule
```

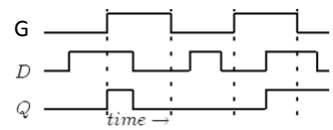

Μανταλωτής D (Latch) – ορισμός συμπεριφοράς



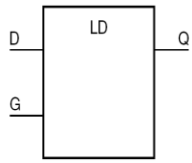
D	G	Q
x	0	Q
0	I	0
I	I	I

```
module latch(d, g, q);
input d, g;
output q;
reg q;


endmodule
```



Μανταλωτής D (Latch) – ορισμός συμπεριφοράς

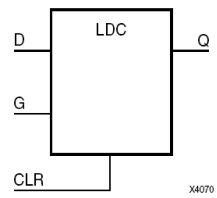


D	G	Q
x	0	Q
0	1	0
1	1	1

```
module latch(d, g, q);
input d, g;
output q;
reg q;

always @(g or d)
begin
  if (g)
    q = d;
end
endmodule
```


Μανταλωτής D (Latch) με CLR – ορισμός συμπεριφοράς

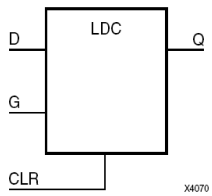


D	G	CLR	Q
x	x	1	0
0	1	0	0
1	1	0	1

```
module latch(d, g, clr, q);
input d, g, clr;
output q;
reg q;

endmodule
```

Μανταλωτής D (Latch) με CLR – ορισμός συμπεριφοράς

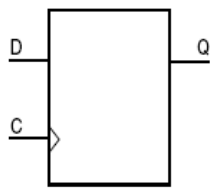


D	G	CLR	Q
x	x	1	0
0	1	0	0
1	1	0	1

```
module latch(d, g, clr, q);
input d, g, clr;
output q;
reg q;

always @(g or d or clr)
begin
  if (clr)
    q = 1'b0;
  else if (g)
    q = d;
end
endmodule
```

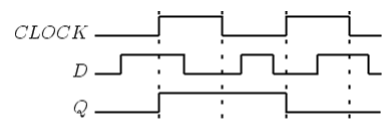

Καταχωρητής D (FF) – ορισμός συμπεριφοράς



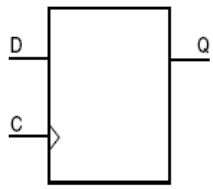
C	CLK	Q
0	↑	0
1	↑	1

```
module dff(d, clk, q);
input d, clk;
output q;
reg q;


endmodule
```



Καταχωρητής D (FF) – ορισμός συμπεριφοράς

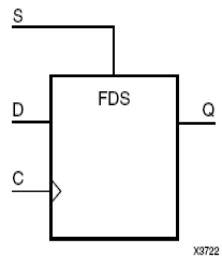


D	CLK	Q
0	↑	0
I	↑	I

```
module dff(d, clk, q);
input d, clk;
output q;
reg q;

always @(posedge clk)
begin
    q <= d;
end
endmodule
```


Καταχωρητής D (FF) με SET– ορισμός συμπεριφοράς

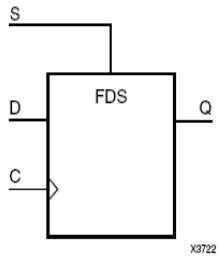


d	clk	S	q
x	↑	I	I
0	↑	0	0
I	↑	0	I

```
module dff(d, clk, s, q);
input d, clk, s;
output q;

endmodule
```

Καταχωρητής D (FF) με SET– ορισμός συμπεριφοράς

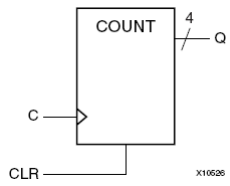


d	clk	S	q
x	↑	1	1
0	↑	0	0
1	↑	0	1

```
module dff(d, clk, s, q);
input d, clk, s;
output q;
reg q;

always @(posedge clk)
begin
  if (s)
    q <= 1'b1;
  else
    q <= d;
end
endmodule
```


Μετρητής – Ορισμός Συμπεριφοράς

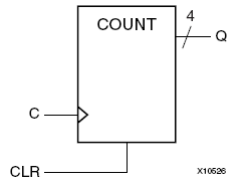



```
module counter (CLK, CLR, Q);
input CLK, CLR;
output [3:0] Q;
reg [3:0] tmp;


endmodule
```

CLR	CLK	Q
↑	X	0000
0	↑	Q+1

Μετρητής – Ορισμός Συμπεριφοράς



CLR	CLK	Q
↑	X	0000
0	↑	Q+1

```
module counter (CLK, CLR, Q);
input CLK, CLR;
output [3:0] Q;
reg [3:0] tmp;

always @(posedge CLK or posedge CLR)
begin
  if (CLR)
    tmp <= 4'b0000;
  else
    tmp <= tmp + 1'b1;
  end
end

assign Q = tmp;
endmodule
```

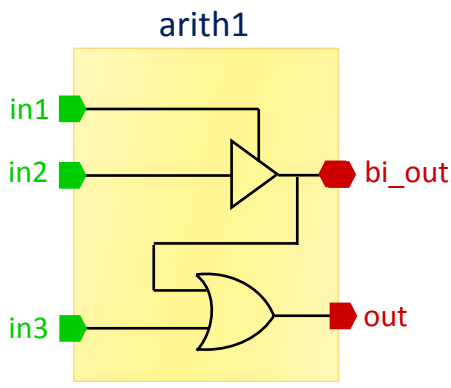

Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Βασική Οντότητα – Μονάδα - Module


```
module MOD1(a, b, f);
input a, b;
output f;
`include "mod2.v"
…
…
endmodule
```


Θύρες – Είσοδοι, Έξοδοι μιας Μονάδας

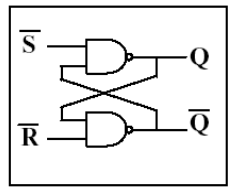



```
module arith1 (bi_out,
out, in1, in2, in3);
inout bi_out;
output out;
input in1, in2;
input in3;
…
…
endmodule
```

Μονάδες και Εμφανίσεις (Instances)

- Η διαδικασία που επικαλούμαστε μια μονάδα και την τοποθετούμε στο κύκλωμα ονομάζεται εμφάνιση (instantiation)

Storage Cell



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	0	undef	
0	1	1	0
1	0	0	1
1	1	Q	$\overline{Q}$

```
module nand(out, a, b,);
input a, b;
output out;

wire out = ~ (a & b);

endmodule
```

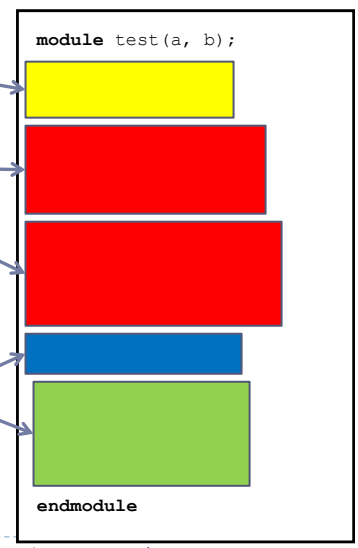
```
module SRLATCH(Q, Qbar, Sbar, Rbar);
input  Sbar, Rbar;
output Q, Qbar;

// Instantiate lower-level modules
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

endmodule
```


Συντακτική Δομή μιας Μονάδας

- δηλώσεις
- τμήματα always
  - 1 ή περισσότερα
- τμήμα initial
  - 1 ή κανένα
- modules/primitives instantiations



Χρόνος στην Verilog - 1

- **Μονάδες χρόνου στην προσομοίωση**
  - `timescale <time_unit base>/<precision base>
    - time_unit base
      - μονάδα μέτρησης χρόνου: αριθμός 1, 10 ή 100 με φυσικές μονάδες
    - time_precision
      - βήματα δεκαδικής ακρίβειας της μονάδας κατά την προσομοίωση
    - φυσικές μονάδες χρόνου: s, ms, us, ns, ps, fs
  - Παράδειγμα:
    - `timescale 1 ns / 10 ps
    - οι μονάδες θα είναι σε 1 ns με ακρίβεια 10 ps, 2 δεκαδικά
- **Αδρανειακή Καθυστέρηση**
  - #(delay) : αναμονή για χρόνο (delay) μονάδες
  - Παράδειγμα:
    - #5 a=8'h1a;

Χρόνος στην Verilog - 2

- **Χρονική Αναμονή Σημάτων**
  - @(<edge> σήμα or <edge> σήμα or …)
  - edge
    - **posedge** (θετική ακμή) ή **negedge** (αρνητική ακμή)
  - or
    - Αναμονή σε πολλαπλά εναλλακτικά σήματα
  - Παραδείγματα:

always @(posedge clk)	always @(negedge clk)
always @(w)	always @(a or b or c)
always @(posedge clk or posedge reset)	

Θεμελιώδη Στοιχεία

- Η Verilog ορίζει θεμελιώδεις μονάδες-στοιχεία που μπορούν να χρησιμοποιηθούν ως εμφανίσεις
- `gate_type #(delay) instance_name [instance_array_range] (terminal, terminal, …)`

Τύπος Πύλης		Σειρά Συνδέσεων
and or xor	nand nor xnor	1 έξοδος, 1 ή περισσότερες είσοδοι
buf	Not	1 έξοδος, 1 είσοδος

- Παραδείγματα:

```
and i1 (out,in1,in2);
and #2 U100 (out, a, b);
and #5 (o,i1,i2,i3,i4);
not ib [31:0] (y, a);
```


Τύποι Μοντελοποίησης

Δομική

```
module_name instance_name [instance_array_range] (signal, signal, ... );

module_name instance_name [instance_array_range]
(.port_name(signal), (.port_name(signal), ...);
```

(Structural)

```
counter counter_1( clk, enable, count_out);
dff u2 (.clk(clock),.q(q[1]),.data(d[1]));
```

Ροή Δεδομένων

```
gate_type #(delay) instance_name [instance_array_range] (terminal, terminal, …);
```

(Dataflow)

```
wire = (a & b) | (c & d);
```

Συνθέσιμη;;

Συμπεριφορά ή Διαδικαστική

```
initial | always @ (sensitivity list) begin--end
```

RTL

```
ays @(a or b or ci)
gin
  sum = a + b + ci;
d
```

```
initial
begin
bus = 16'h0000; #10 bus = 16'hC5A5;
#20 bus = 16'hFFAA;
end
```

Συμβάσεις της γλώσσας Verilog

- Η Verilog είναι case sensitive.
  - Οι λέξεις κλειδιά πρέπει να είναι με μικρά γράμματα
- Σχόλια
  - Για μία γραμμή: `//`
  - Για πολλές: `/*  */`
- Τύποι Δεδομένων

Τιμή	Ερμηνεία
0	Λογικό 0, άρνηση
1	Λογικό 1, κατάφαση
X	Άγνωστο ή Μη αρχικοποιημένο
Z	Υψηλής εμπέδησης –ασύνδετο ή τρικατάστατο

Αναπαράσταση Ακέραιων Αριθμών

- **<size>'<base><number>**
  - <size> δείχνει τον αριθμό από bits
  - '<base> είναι η βάση

Βάση	Σύμβολο	Επιτρεπτές Τιμές
Δυο	b ή B	0, 1, x, X, z, Z, ?, _
Οκτώ	o ή O	0-7, x, X, z, Z, ?, _
Δέκα	d ή D	0-9, x, X, z, Z, ?, _
Δεκαέξι	h ή H	0-9, a-f, A-F, x, X, z, Z, ?, _

  - Αν παραλειφθεί ο αριθμός bits το μέγεθος εξαρτάται (32-bits)
  - Το _ χρησιμοποιείται για ευκολότερη ανάγνωση

100	**4'b1111**	**6'h3a**
6'b111010	12'h13x	8'b10_10_1110

Τελεστές -1

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

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

Τελεστές - 2

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)

Λογικοί

!	!m	Είναι το m ψευδές; (1-bit αποτέλεσμα)
&&	m && n	Είναι το m και το n αληθή; (1-bit αποτέλεσμα)
||	m || n	Είναι το m ή το n αληθές; (1-bit αποτέλεσμα)

Τελεστές - 3

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	Είναι το m μεγαλύτερο του n;
>=	m >= n	Είναι το m μεγαλύτερο ή ίσο του n; (1-bit αποτέλεσμα)
<	m < n	Είναι το m μικρότερο του n;
<=	m <= n	Είναι το m μικρότερο ή ίσο του n; (1-bit αποτέλεσμα)

Μετατόπισης

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

Τελεστές - 4

Διάφοροι

Τελεστής	Χρήση	Περιγραφή
? :	sel ? m : n	Αν το sel είναι αληθές επέστρεψε m αλλιώς n
{}	{m, n}	Ένωσε τα διανύσματα m και n επιστρέφοντας την συνένωση τους
{{ }}	{n{m}}	Επανάλαβε το διάνυσμα m ν φορές

Τύποι μεταβλητών στην Verilog

Τύπος	Ιδιότητες	Παράδειγματα

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;`

Μεταβλητές wire

- 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 wire

- Παράδειγμα μονάδας πρόσθεσης 1-bit

```
module adder(a, b, sum, cout);
input  a, b;
output sum, cout;

wire sum  = a ^ b;
wire cout = a & b;

endmodule
```


Μεταβλητές reg, διεργασίες και ακολουθιακή λογική

- μεταβλητές με μνήμη
  - διατηρούν την κατάσταση τους μέχρι την επόμενη ανάθεση
- μεταβλητές διαδικασιών
  - `always, initial`
- δεν συνεπάγονται καταχωρητή σε επίπεδο υλικού

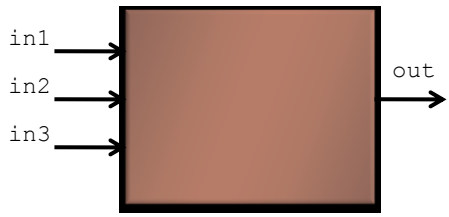
```
reg a;

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

```
reg q;

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

Μεταβλητές reg, διεργασίες και ακολουθιακή λογική




```
reg out;

always @(in1 or in2 or in3)
       out = in1 | (in2 & in3);
```

Ισοδύναμοι ορισμοί

```
wire out = in1 | (in2 & in3);
```


Αναθέσεις σε Διαδικασίες

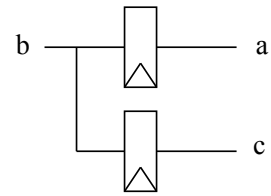
Ανάθεση	Σημασιολογία
`Register_data_type = expression;`	**Κλειδωμένη (blocking) ανάθεση** η έκφραση αξιολογείται και γίνεται η ανάθεση πριν την εκτέλεση επόμενης γραμμής
`Register_data_type <= expression;`	**Μη κλειδωμένη (non-blocking) ανάθεση** η έκφραση αξιολογείται, αλλά η ανάθεση θα γίνει στο τέλος του τρέχοντος βήματος χρόνου και η εκτέλεση συνεχίζει

Παράδειγμα	`begin`
	`  n = m;`
	`  m = n;`
	`end`
		`begin`
		`  n <= m;`
		`  m <= n;`
		`end`

	Το n και το m παίρνουν την τιμή του m	**Το n και m ανταλλάσουν τιμές**

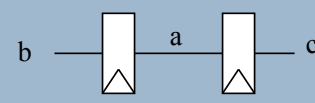
Αναθέσεις σε Διαδικασίες

```
always @(posedge clk)
begin
    a = b;
    c = a;
// c παίρνει τιμή του
b //
end
```



η σειρά έχει σημασία

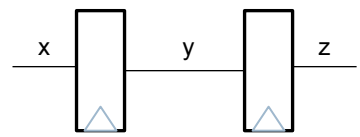
```
always @(posedge clk)
begin
    a <= b;
    c <= a;
// c παίρνει παλιά
τιμή του a //
end
```



η σειρά δεν έχει σημασία

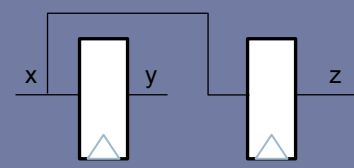
Αναθέσεις σε Διαδικασίες

```
always @(posedge clk)
begin
  z = y;
  y = x;
end
```



shift register

```
always @(posedge clk)
begin
  y = x;
  z = y;
end
```



παράλληλα FF

Αναθέσεις σε Διαδικασίες

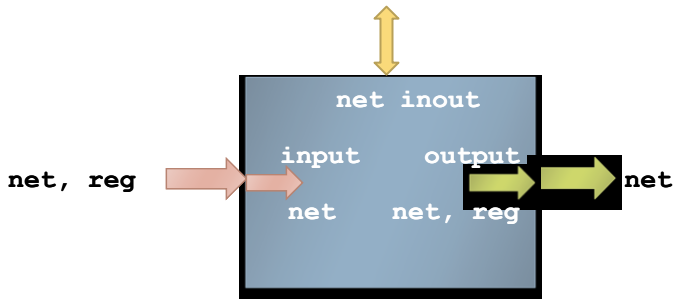
- Με καθυστέρηση

`a = #10 b;`
`c = a;`
	@t = 0 …
	@t =10 : a = b; // b @t=0 //
	@t = 10 : c = a; // a @t= 10 //

`#10 a = b;`
`c = a`
	@t = 0 …
	@t =10 : a = b; // b @t=10 //
	@t = 10 : c = a; // a @t= 10 //

`a <= #10 b;`
`c <= a;`
	@t = 0 …
	@t =10 : a = b; // b @t= 0 //
	@t = 0 : c = a; // a @t= 0 //

Θύρες Μονάδων



- Οι είσοδοι και οι είσοδοι-έξοδοι έχουν τύπο `wire` μέσα στην μονάδα
- Οι έξοδοι μπορεί να έχουν είτε τύπο `wire` (συνδυαστική εξίσωση) είτε `reg` από `always` τμήμα

Συνδέσεις Μονάδων και Εμφανίσεων

Κατά θέση Σήματος

```
module adder(sum, in1, in2);
...
endmodule;

adder adder_inst (a, b, c);
// sum = a,
// in1 = b,
// in2 = c.
```

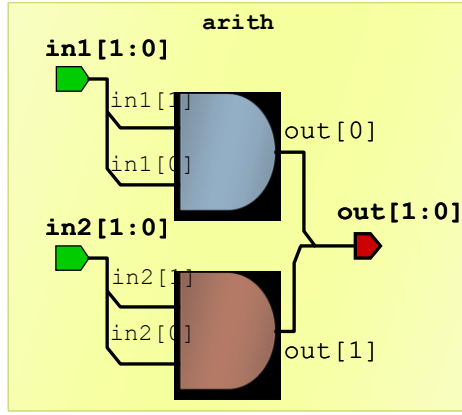
Κατά όνομα Σήματος

```
module adder(sum, in1, in2);
...
endmodule;

adder adder_inst(.sum(c),
.in1(a), .in2(b));

// sum = c,
// in1 = a,
// in2 = b.
```


Δίαυλοι - Busses




```
module arith (out, in1,
in2);

output [1:0] out;

input [1:0] in1, in2;

…

…

endmodule
```

Δίαυλοι - Busses

- Συμβάσεις
  - [MSB:LSB]
- Προσοχή στο πλάτος

```
module adder (a, b, sum, cout);
input [7:0] a, b;
output [7:0] sum;
output cout;

wire [8:0] temp = a + b;

wire [7:0] sum = temp[7:0];
wire cout = temp[8];

endmodule;
```


Συνθήκη `if/else`

- το `if/else` επιτρέπεται μόνο σε `always`
- αν η συνθήκη περικλείει πολλαπλές εντολές χρησιμοποιείται `begin/end`
- επιτρέπονται
  - πολλαπλά `else if`
  - `if` μέσα σε `if`

```
module mux (a, b, sel, out);
input [4:0] a, b;
input sel;
output [4:0] out;

reg [4:0] out;

always @(a or b or sel)
begin
  if (!sel)
       out = a;
  else
       out = b;
end
endmodule
```

Συνθήκη `case`

- το `case` επίσης επιτρέπεται μόνο σε `always`
- μόνο για σταθερές εκφράσεις
- δεν υπάρχει `break`

```
module mux (a, b, c, d, sel, out);
input [4:0] a, b, c, d;
input [1:0] sel;
output [4:0] out;

reg [4:0] out;
always @(a or b or c or d or sel)
begin
  case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    2'b11: out = d;
    default: out = 5'bx;
  endcase
endmodule
```


Συνθεσιμότητα – Συνθέσιμες και μη εκφράσεις

- Εκφράσεις Verilog που είναι συνθέσιμες παράγουν σχηματικό (δίκτυο πυλών) για ASIC ή FPGA
- λ.χ.:

```
wire [7:0] sum = temp[7:0]
& {8{a}};

wire cout = temp[8]
```

- Μη συνθέσιμος κώδικας χρησιμοποιείται για την διαδικασία προσομοίωσης
- Δεν αποτελεί μέρος του σχεδίου, αλλά της δοκιμής/ελέγχου του
- λ.χ.:

```
initial begin

a = 0; b = 0;
#5 a = 1;
b = 1;

end
```

Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Χρόνος – Καθυστέρηση στην Verilog

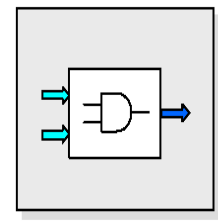
- **Λειτουργικός Έλεγχος** (Functional Test)
  - Προσεγγιστική, υψηλού επιπέδου καθυστέρηση
    - `always @(posedge clk)`
      `q <= #2 d; // FF με 2 μονάδες καθυστέρηση //`
  - Στον λειτουργικό έλεγχο η συνδυαστική καθυστέρηση θεωρείται αμελητέα
    - Έλεγχος ανά κύκλο (cycle-based, cycle-accurate)
    - `wire a = (b & c) | d;`
      `// μόνο λειτουργία, όχι καθυστέρηση //`
  - Στην δοκιμή οι καθυστερήσεις χρησιμοποιούνται για να φτιαχτεί το εξωτερικό ερέθισμα του κυκλώματος
    - Διανύσματα εισόδου και ο χρονισμός τους
- **Στατική Χρονική Ανάλυση** (Static Timing Analysis)
  - Μετά την σύνθεση γίνεται έλεγχος του κύκλου

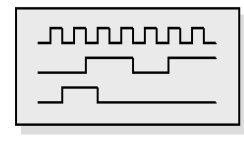
Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση

Λειτουργικός Έλεγχος

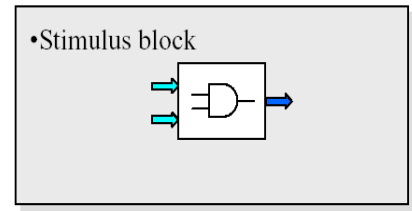
- Ελέγχουμε την κάθε μονάδα χωριστά
  - Προδιαγραφές
  - Χρονισμός (ανά κύκλο)
  - Τιμές σημάτων
  - Λειτουργίας
- Συνδέουμε τις επιμέρους μονάδες στο συνολικό σχέδιο
  - Ελέγχουμε την συνολική λειτουργία

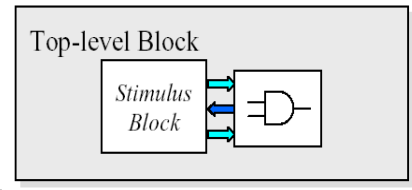




Λειτουργικός Έλεγχος

- Για κάθε μονάδα υλοποιούμε ένα **πλαίσιο ελέγχου – testbench**
  - **Εμφανίζει την υπό έλεγχο μονάδα**
  - **Εφαρμόζει διανύσματα εισόδων στον χρόνο**
  - **Ελέγχει τα διανύσματα εξόδων στον χρόνο**
- Μεθοδολογίες ελέγχου
  1. **Μη-αυτόματα**
     - Ο μηχανικός ελέγχει ότι τα σήματα είναι σωστά
  2. **Αυτόματα**
     - Οι τιμές ελέγχονται αυτόματα έναντι των αναμενόμενων





% κάλυψης ;;;

Παράδειγμα testbench

Testbench

```
module half_adder_testbench;
reg a, b;
wire sum, cout;

half_adder half_adder_instance (a, b, sum, cout);

initial begin
a = 0; b = 0;
#5 $display("a: %x, b: %x, sum: %x, cout: %x", a, b,
sum, cout);
a = 1;
#5 $display("a: %x, b: %x, sum: %x, cout: %x", a, b,
sum, cout);
B = 1;
#5 $display("a: %x, b: %x, sum: %x, cout: %x", a, b,
sum, cout);
a = 0;
#5 $display("a: %x, b: %x, sum: %x, cout: %x", a, b,
sum, cout);

end
endmodule;
```

Μονάδα

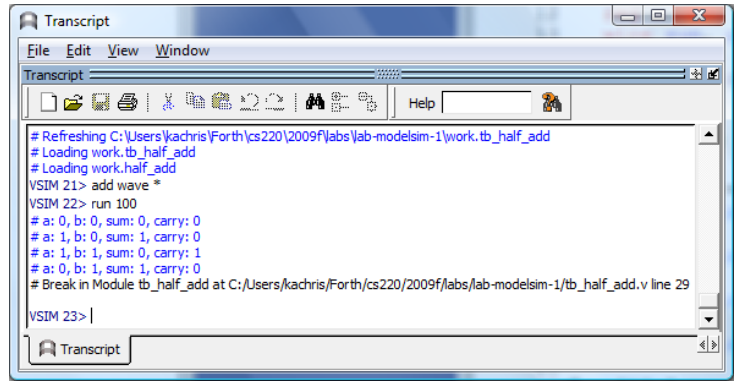
```
module half_adder(a,
b, sum, cout);

wire sum = a ^ b;
wire cout = a & b;

endmodule;
```

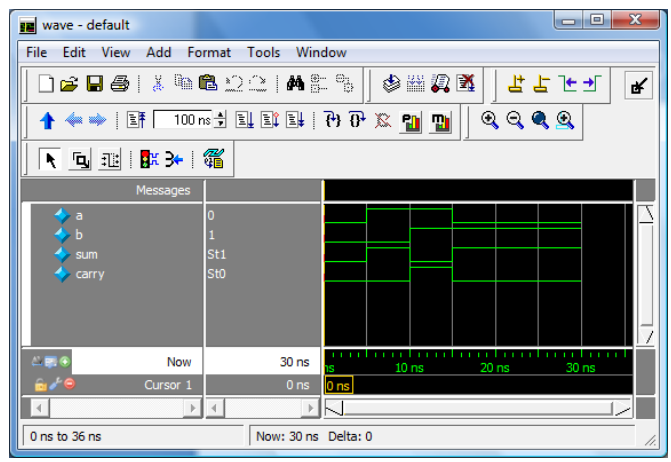
Προσομοιωτής

- Παράθυρο κειμένου



Προσομοιωτής

- Παράθυρο κυμματομορφών



Παράδειγμα – Μετρητής 8-bit

Σχέδιο Μετρητή

```
module counter(clk, reset,
count);
input clk, reset;
output [7:0] count;

Reg [7:0] count;

always (@posedge clk)
begin
  if (reset)
     count = #2 8'b0;
  else
     count = #2 count + 1;
end
endmodule
```

Μονάδα Ελέγχου

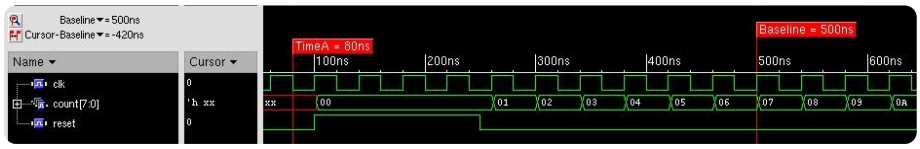
```
module counter_testbench; reg clk,
reset;
wire [7:0] count;

counter counter_instance (clk,
reset, count);

initial begin
  clk = 0;
  $monitor("time %d ns: count=%d",
$time, count);
  #100 reset = 1;
  #150 reset = 0;
end

always begin
   #20 clk = ~clk;
end
endmodule
```


Παράδειγμα – Μετρητής 8-bit

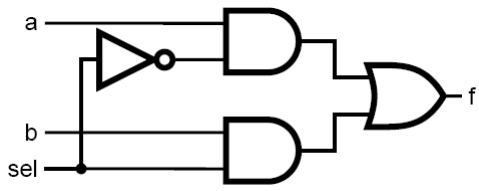



```
time                   0 ns: count=  x
time                 102 ns: count=  0
time                 262 ns: count=  1
time                 302 ns: count=  2
time                 342 ns: count=  3
time                 382 ns: count=  4
time                 422 ns: count=  5
time                 462 ns: count=  6
```

Περιεχόμενα

- Τυπική Ροή Σχεδίασης
- Ιεραρχία στην Σχεδίαση
- Η Γλώσσα Verilog
- Επίπεδα Αφαίρεσης στην Σχεδίαση
- Αναπαράσταση και Υλοποίηση σε Verilog
- Εισαγωγή στην Verilog μέσω παραδειγμάτων
- Αναλυτική Επισκόπηση της Verilog
- Χρόνος-Καθυστέρηση στην Verilog
- Λειτουργικός Έλεγχος και Προσομείωση
- … μια άσκηση

Πολυπλέκτης – Ορισμός πυλών

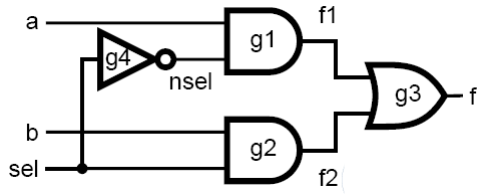



```
module sel(a, b, sel, f);
input  a, b, sel;
output f;


endmodule
```

a	b	sel	f
x	y	0	x
x	y	I	y

Πολυπλέκτης – Ορισμός πυλών



a	b	sel	f
x	y	0	x
x	y	1	y

```
module sel(a, b, sel, f);
input  a, b, sel;
output f;
wire nsel, f1, f2;

assign nsel = ~sel;
assign f1 = a & nsel;
assign f2 = b & sel;
assign f = f1 | f2;
endmodule
```