

HY220

Εργαστήριο Ψηφιακών Κυκλωμάτων

**Χειμερινό Εξάμηνο
2020**

Verilog: Μια πιο κοντινή ματιά

Δομή της γλώσσας

- Μοιάζει αρκετά με τη C
 - Preprocessor
 - Keywords
 - Τελεστές

```
=  
==, !=  
<, >, <=, >=  
&& ||  
? :
```

```
& and  
| or  
~ not  
^ xor
```

- Γλώσσα «event driven»

```
`timescale 1ns / 1ns  
  
`define dh 2  
(e.g q <= #`dh d)  
  
`undef dh  
  
`ifdef dh / `ifndef dh  
    ...  
`else  
    ...  
`endif  
  
`include "def.h"
```

Events in Verilog (1/3)

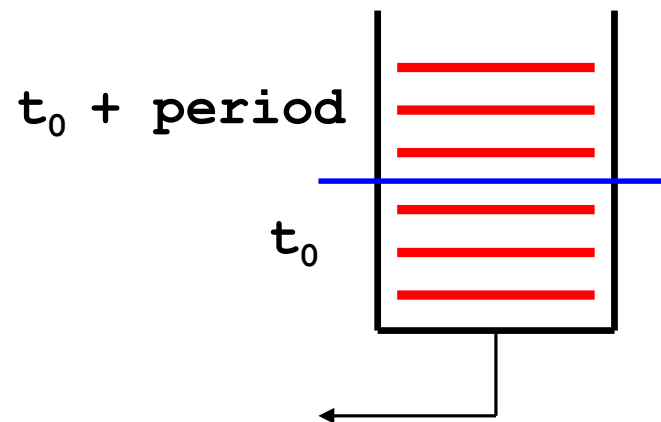
- Δουλεύει μόνο όταν κάτι αλλάξει
- Όλο το simulation δουλεύει γύρω από μια ουρά από γεγονότα (event queue)
 - Περιέχει events και ετικέτες με το χρόνο στον οποίο θα εκτελεστούν
 - **Καμιά εγγύηση για τη σειρά εκτέλεσης γεγονότων που πρέπει να γίνουν στον ίδιο χρόνο!!!**

```
always clk = #(`period / 2) ~clk;  
always @(posedge clk) a = b + 1;  
always @(posedge clk) b = c + 1;
```



Events in Verilog (2/3)

- Βασική ροή προσομοίωσης
 - Εκτέλεση των events για τον τρέχοντα χρόνο
 - Οι εκτέλεση events αλλάζει την κατάσταση του συστήματος και μπορεί να προκαλέσει προγραμματισμό events για το μέλλον
 - Όταν τελειώσουν τα events του τρέχοντος χρόνου προχωράμε στα αμέσως επόμενα χρονικά!



Events in Verilog (3/3)

- 2 τύποι events
 - **Evaluation:** υπολογίζουν τις συναρτήσεις των εισόδων της έκφρασης (RHS)
 - **Update:** αλλάζουν τις εξόδους (LHS)
 - Λαμβάνουν υπόψιν delays – non-blocking assignments

Update: Γράφει το νέο a και προγραμματίζει evaluation events για κώδικα που εξαρτάται από το a .

Evaluation: διαβάζει τις τιμές b και c , υπολογίζει, αποθηκεύει εσωτερικά και προγραμματίζει ένα update event

$$\boxed{a} \leq \boxed{b + c}$$

Blocking vs Non-blocking assignments and Events

- **Blocking =**

- Evaluation/read (RHS) και assignment/write (LHS) (update event) στον ίδιο χρόνο
- Εκτέλεση σειριακή

- **Non-blocking <=**

- Evaluation και assignment σε 2 βήματα
 - Evaluation στο δεξί μέλος (RHS) άμεσα
 - Assignment (update) στο αριστερό μέλος (LHS) όταν τελειώσουν όλα τα evaluations του τρέχοντος χρόνου

```
always @(posedge clk)
  a = b;
always @(posedge clk)
  b = a;
```



Swap ?

```
always @(posedge clk)
  a <= b;
always @(posedge clk)
  b <= a;
```



Delays and Events

- Regular / Inter-Assignment delays

```
#5 a = b + c; // a=b+c at time 5
#4 d = a;      // d=anew at time 9
```

- Intra-Assignment delays

- Evaluation του RHS πριν την καθυστέρηση

- With blocking assignments:

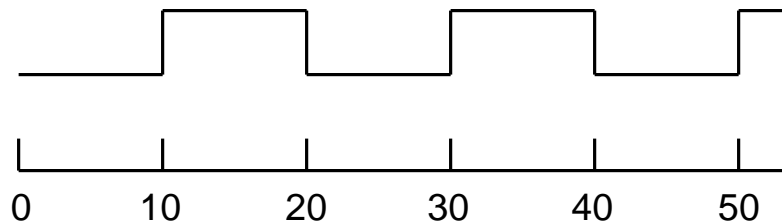
```
a = #5 b + c; // a=b+c at time 5
d = a;        // d=anew at time 5
```

- With non-blocking assignments:

```
a <= #5 b + c; // a=b+c at time 5
d <= a;        // d=aold at time 0
```

Events Example

- Κάθε έκφραση συνδέεται με έναν αρχικό χρόνο
- Initial και always: εσωτερικά σειριακά
 - εκτός από non-blocking assignments



```
initial begin 0
  a = 0; b = 0; c = 0;
  clk = 0;
end

always begin 10, 20, 30, 40, 50
  clk = #10 1;
  clk = #10 0;
end 4, 14, 34

wire #4 [3:0] comb = a + b;

always @(posedge clk) 10, 30
  a <= b + 1;
always @(posedge clk) 10, 30
  b <= c + 1;
always @(posedge clk) 15, 35
  c <= #5 a + 1;
```


Sensitivity lists

- Λογικές εκφράσεις με **or**
- **posedge** και **negedge**
 - Ρολόγια, reset
- Παράλειψη παραγόντων RHS και αυτών που γίνονται “read” δίνουν λάθη στην προσομοίωση
- Προσοχή στο hardware που θέλουμε να περιγράψουμε...

```
always @(posedge clk or negedge rst_)  
...  
  
always @(opcode or b or c)  
  if (opcode == 32'h52A0234E)  
    a = b ^ (~c);  
  
always @(posedge a or posedge b)  
...
```

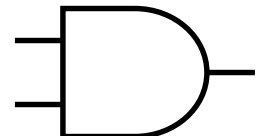


Τιμές σημάτων

- **Four-valued logic**
- 0 ή 1
- Z
 - Έξοδος τρικατάστατου οδηγητή
 - Καλώδιο χωρίς ανάθεση
- X
 - Αρχική τιμή των regs
 - Έξοδος πύλης με είσοδο/ους Z
 - Ταυτόχρονη ανάθεση 0 και 1 από δύο ή περισσότερες πηγές (multi-source logic) [πηγή = always block]
- Προσοχή στην αρχικοποίηση (regs)

```
initial ...
```

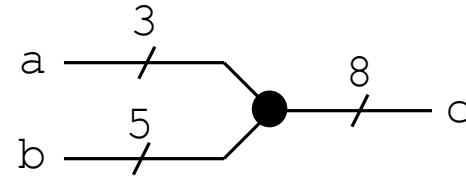
```
always @(posedge clk)  
  if (reset) ...  
  else ...
```



	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Concatenation

- «Hardwired» πράξεις...
- ... απαραίτητες σε μια HDL



```
wire [2:0] a;  
wire [4:0] b;  
  
wire [7:0] c = {a , b};
```

```
wire [7:0] unsigned;  
wire [15:0] sign_extend = {  
    (unsigned[7] ? 8'hFF : 8'h0), unsigned  
};
```

For ... While ...

- ... τα γνωστά
- Μόνο μέσα σε blocks !
- Δεν υπάρχει break ουτε continue!!!
- Δεν υπάρχει i++, ++i κτλ!
- Κυρίως για testbenches !!!

```
integer i;  
// the famous i variable :)  
initial begin  
    for ( i=0; i<10; i=i+1 )begin  
        $display ("i= %d",i);  
    end  
end
```

```
integer j; //reg [3:0] j is OK!  
initial begin  
    j=0;  
    while (j < 10)begin  
        $display ("j= %b",j);  
        j=j+1;  
    end  
end
```

Παραμετρικά modules (1/2)

- Μπορούμε να έχουμε παραμέτρους σε ένα module
- Default μέγεθος
- ... πολύ βολικό!

```
module RegLd #(
    parameter N = 8,
    parameter dh = 2)
(
    input          clk,
    input          load,
    input          [N-1:0] D,
    output reg [N-1:0] Q
);

always @(posedge clk)
    if (load)
        Q = #dh D;

endmodule
```

Παραμετρικά modules (2/2)

```
wire clk, ld;  
wire [3:0] d2;  
wire [3:0] q2;  
  
RegLd      reg2 (clk, ld, d2, q2);  
defparam reg2.N = 4;  
defparam reg2.dh = 4;
```

ή

```
RegLd # (  
    .N      (4),  
    .dh     (2)  
) reg2 (  
    .clk    (clk)  
    .load   (ld),  
    .D      (d2),  
    .Q      (q2)  
);
```

Τρικατάστατοι οδηγητές

- Εκμετάλλευση της κατάστασης Z
- Χρήση του τύπου inout

```
module tristate(en, clk, data);  
  input      en, clk;  
  inout [7:0] data;  
  
  wire [7:0] data = (en) ? data_out : 8'bz;  
  
  always @(posedge clk)  
  begin  
    if (!en)  
      case (data)  
        ...  
      endcase  
  end  
endmodule
```

```
wire [7:0] bus;  
  
tristate tr0(en0, clk, bus);  
tristate tr1(en1, clk, bus);  
tristate tr2(en2, clk, bus);
```

Μνήμες

- Αναδρομικά: array of array
- Can be synthesized
- Αρχικοποίηση από αρχείο:
 - \$readmemh(filename, array)
 - \$readmemb(filename, array)

```
wire [ 9:0] addr;  
wire [15:0] word_in;  
reg [15:0] word_out;  
reg [15:0] memory [1023:0];  
  
always @(posedge clk) begin  
    if (we)  
        memory[addr] = word_in;  
    else  
        word_out = memory[addr];  
end
```

```
initial begin  
    $readmemh ("memory.dat", memory);  
end
```

```
memory.dat:  
0F00 00F1  
0F02
```


Συναρτήσεις – Functions (1/3)

- Δήλωση (declaration):

```
function [ range_or_type ] fname;  
    input_declarations  
    statements  
endfunction
```

- Επιστρεφόμενη τιμή (return value):

- Ανάθεση στο σώμα του function

```
fname = expression;
```

- Κλήση (function call):

```
fname ( expression,... )
```

Συναρτήσεις - Functions (2/3)

- Χαρακτηριστικά συναρτήσεων:
 - Επιστρέφει 1 τιμή (default: 1 bit)
 - Μπορεί να έχει πολλαπλά ορίσματα εισόδου (πρέπει να έχει τουλάχιστον ένα)
 - Μπορούν να καλούν άλλες functions αλλά όχι tasks.
 - Δεν υποστηρίζουν αναδρομή (non-recursive)
 - Εκτελούνται σε μηδέν χρόνο προσομοίωσης
 - **Δεν** επιτρέπονται χρονικές λειτουργίες (π.χ. delays, events)
- Χρησιμοποιούνται για συνδυαστική λογική και είναι synthesizable
 - προσοχή στον κώδικα για να γίνει σωστά σύνθεση

Συναρτήσεις - Functions (3/3)

- Function examples:

```
function calc_parity;  
input [31:0] val;  
begin  
    calc_parity = ^val;  
end  
endfunction
```

```
function [15:0] average;  
input [15:0] a, b, c, d;  
begin  
    average = (a + b + c + d) >> 2;  
end  
endfunction;
```

Verilog Tasks (1/2)

- Τυπικές procedures
- Πολλαπλά ορίσματα `input`, `output` και `inout`
- Δεν υπάρχει συγκεκριμένη τιμή επιστροφής (χρησιμοποιεί τα ορίσματα `output`)
- Δεν υποστηρίζουν αναδρομή (non-recursive)
- Μπορούν να καλούν άλλες tasks και functions
- Μπορούν να περιέχουν `delays`, `events` και χρονικές λειτουργίες
 - Προσοχή στη σύνθεση

Verilog Tasks (2/2)

- Task example:

```
task ReverseByte;  
  input [7:0] a;  
  output [7:0] ra;  
  integer j;  
begin  
  for (j = 7; j >= 0; j=j-1) begin  
    ra[j] = a[7-j];  
  end  
end  
endtask
```

Functions and Tasks

- Ορίζονται μέσα σε modules και είναι τοπικές
- Δεν μπορούν να έχουν always και initial blocks αλλά μπορούν να καλούνται μέσα από αυτά
 - Μπορούν να έχουν ότι εκφράσεις μπαίνουν σε blocks

Functions vs Tasks

Functions	Tasks
Μπορούν να καλούν άλλες functions αλλά όχι tasks	Μπορούν να καλούν άλλες tasks και functions
Εκτελούνται σε μηδενικό χρόνο προσομοίωσης	Μπορούν να διαρκούν μη μηδενικό χρόνο προσομοίωσης
Δεν μπορούν περιέχουν χρονικές λειτουργίες (delay, events κτλ)	Μπορούν να περιέχουν χρονικές λειτουργίες (delay, events κτλ)
Έχουν τουλάχιστον 1 είσοδο και μπορούν να έχουν πολλές	Μπορούν να έχουν μηδέν ή περισσότερα ορίσματα εισόδων, εξόδων και inout
Επιστρέφουν μια τιμή, δεν έχουν εξόδους	Δεν επιστρέφουν τιμή αλλά βγάζουν έξοδο από τα ορίσματα εξόδου output και inout

System Tasks and Functions

- Tasks and functions για έλεγχο της προσομοίωσης
 - Ξεκινούν με "\$" (e.g., \$monitor)
 - Standard – της γλώσσας

- Παράδειγμα system task: \$display

```
$display("format-string", expr1, ..., exprn);
```

format-string - regular ASCII mixed with formatting

characters %d - decimal, %b - binary, %h - hex, %t - time, etc.

other arguments: any expression, including `wires` and `regs`

```
$display("Error at time %t: value is %h, expected %h", $time,  
actual_value, expected_value);
```


Χρήσιμες System Tasks

- `$time` – τρέχον χρόνος προσομοίωσης
- `$monitor` – τυπώνει όταν αλλάζει τιμή ένα όρισμα (1 μόνο κάθε φορά νέες κλήσεις ακυρώνουν τις προηγούμενες)
`$monitor("cs=%b, ns=%b", cs, ns)`
- Έλεγχος προσομοίωσης
 - `$stop` - διακοπή simulation
 - `$finish` - τερματισμός simulation
- Υπάρχουν και συναρτήσεις για file I/O
 - `$fopen, $fclose, $fwrite ... etc`