8: Σχεδίαση και Προσομοίωση Ολοκληρωμένων Συστημάτων με Verilog (I)

Vasileios Tenentes

University of Ioannina

Outline

Οδηγίες εγκατάστασης Modelsim
Εισαγωγή
Κατηγορίες ολοκληρωμένων συστημάτων – βασική ορολογία
Τι είναι η Verilog;
Ιστορία της Verilog
Διάγραμμα Ροής Σχεδιασμού Συστημάτων
Παράδειγμα 1 – Hello World
Παράδειγμα 2 – 4bit counter
Syntax and Semantics
Αναπαράσταση αριθμών
Modules
Παράδειγμα 3 – full adder -1bit counter
Παράδειγμα 4. 4bit-Adder
Παράδειγμα 5. Parity checker
Port connection rules
Παράδειγμα 6: unconnected ports
Hierarchical identifiers
Data types
Behavioural Modeling
Μοντελοποίηση Συμπεριφοράς

Οδηγίες εγκατάστασης Quartus και Modelsim

Οδηγίες εγκατάστασης του Modelsim θα έχετε στο αρχείο installation_quartus.pdf

Το modelsim είναι μέρος του Quartus και υπάρχει σε free άδεια χρήσης εδώ

https://fpgasoftware.intel.com/13.0sp1/?edition=web

Εισαγωγή

Κατηγορίες ολοκληρωμένων συστημάτων – βασική ορολογία

Application Specific Integrated Circuit (ASICs)

Αρχικά ο όρος ξεκίνησε για να περιγράψει κυκλώματα ειδικού σκοπού. Πλέον, ο όρος χρησιμοποιείται για όλα τα κυκλώματα που δεν είναι FPGAs.

Τα ASICs μόλις κατασκευαστούν παραμένουν ως είναι για όλη την διάρκεια της ζωής τους και δεν επιτρέπουν αλλαγή του σχεδιασμού τους

Field Programmable Gate Arrays (FPGAs)

Επανασχεδιάσιμα κυκλώματα που επιτρέπουν την αλλαγή του σχεδιασμού τους από τον χρήστη. Χρησιμοποιούνται για την κατασκευή πρωτοτύπων, για εκπαιδευτικούς λόγους, και για επιτάχυνση απαιτητικών υπολογιστικών αναγκών.

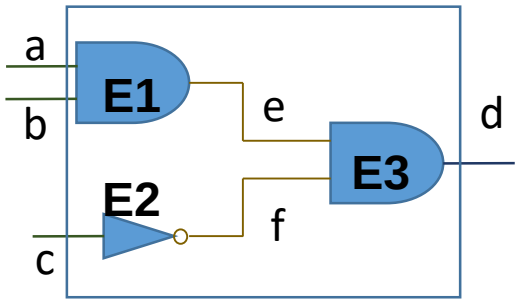
Υπάρχουν και **Ετερογενή Συστήματα**

Συστήματα που έχουν διαφορετικά ASIC στοιχεία, π.χ. CPU Cores, GPU, κτλ., μαζί με FPGA fabric για επαναπροσδιορισμό του σχεδιασμού τους

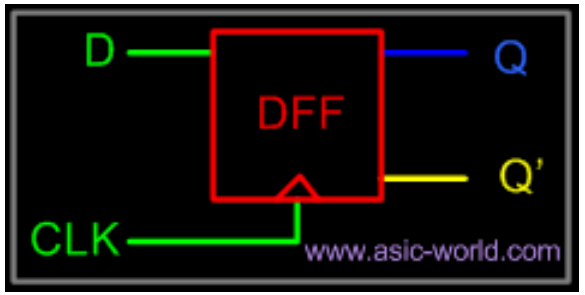
Διαδικασίες σχεδιασμού για ASIC και γλώσσες περιγραφής υλικού, όπως είναι η Verilog, χρησιμοποιούνται και στον προγραμματισμό των FPGAs

Τι είναι η Verilog;

Η Verilog είναι **HARDWARE DESCRIPTION LANGUAGE (HDL)**. HDL είναι γλώσσες που περιγράφουν ένα ψηφιακό σύστημα π.χ. ένα router, έναν μικροεπεξεργαστή, έναν ελεγκτή μνήμης, ένα απλό flip-flop. Χρησιμοποιώντας μια HDL γλώσσα κάποιος μπορεί να περιγράψει ένα ψηφιακό σύστημα σε οποιοδήποτε επίπεδο.




```
module example1(a,b,c,d);
    input a,b,c;
    output d;
    wire a,b,c,d,e,f;

    and E1(e,a,b);
    not E2(f,c);
    and E3(d,e,f);

endmodule
```




```
// D flip-flop Code
module d_ff (d, clk, q, q_bar);
    input d, clk;
    output q, q_bar;
    wire d, clk;
    reg q, q_bar;
    always @ (posedge clk)
    begin
        q <= d;
        q_bar <= ! d;
    end
endmodule
```

Ιστορία της Verilog

Αρχικά η Verilog ξεκίνησε ως ιδιωτική γλώσσα περιγραφής και προσομοίωσης υλικού από την εταιρεία Gateway Design Automation Inc. το 1984.

Ήταν ένα κράμα μιας γλώσσας περιγραφής υλικού που προϋπήρξε της HiLo και της C που γνωρίζουμε.

Η Verilog επισημοποιήθηκε ως γλώσσα με Standards που εμφανίστηκαν την περίοδο 1984-1990.

Σήμερα είναι η κυρίαρχη γλώσσα μοντελοποίησης υλικού

Προσομοιωτές με βάση τη Verilog αναπτύχθηκαν το 1985-1987. Ο Verilog simulator ήταν προϊόν της Gateway. Η πρώτη επέκταση είναι η Verilog-XL, στην οποία εφαρμόστηκε ο αλγόριθμος-XL στην προσομοίωση κυκλωμάτων στο επίπεδο πυλών (gate-level simulation).

Το 1990 η εταιρία **Cadence Design System**, η οποία μέχρι τότε ασχολούταν με Thin film process simulators, εξαγόρασε την Gateway Automation System. Η Cadence εμπορεύθηκε την γλώσσα Verilog και τον προσομοιωτή της προωθώντας μια **bottom-up μεθοδολογία σχεδιασμού**. Παράλληλα η εταιρεία **Synopsys** προωθούσε μια **top-down μεθοδολογία** σχεδιασμού χρησιμοποιώντας την Verilog.

Το Standard της Verilog

Το 1990 η Cadence αναγνώρισε πως αν η Verilog παρέμενε κλειστή γλώσσα τότε οι ανάγκες για standardization θα οδηγούσε στην υιοθέτησης της Γλώσσας VHDL. Επομένως, η Cadence οργάνωσε το Open Verilog International (OVI), στο οποίο το 1991 παρέδωσε τα ηνία της γλώσσας Verilog.

Το OVI βελτίωσε το Language Reference Manual (LRM) της γλώσσας και προσπάθησε να κάνει τη γλώσσα όσο γίνεται πιο πολύ **vendor-independent**.

Όταν η OVI σχημάτισε το LRM το 1992 πολλές εταιρείες έγραψαν προσομοιωτές Verilog. Ο πιο επιτυχημένος ήταν το VCS, Verilog Compiled Simulator, από την Chronologic Simulation, ο οποίος ήταν μεταφραστής, σε αντίθεση με το Verilog-XL που ήταν βασισμένο σε διερμηνέα της γλώσσας. Ως αποτέλεσμα, παρόλο που χρειαζόταν μεταγλώττιση, ήταν πολύ πιο γρήγορος κατά την προσομοίωση.

Το 1994, the IEEE 1364 working group, σχηματίστηκε το οποίο μετέτρεψε το OVI LRM σε IEEE standard (IEEE Std. 1364-1995). Τον Δεκέμβριο του 1995 η Verilog έγινε IEEE standard.

Νέα χαρακτηριστικά εμπλουτίζουν συνεχώς τη γλώσσα Verilog, ακολούθησε η Verilog 2001, IEEE Std. 1364-2001.

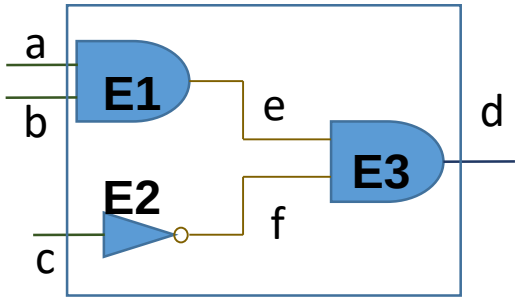
Ακολούθησε το IEEE Std. 1364-2005 και το 2009 η Verilog ενσωματώθηκε σε μια άλλη γλώσσα που είχε κάνει την εμφάνισή της την SystemVerilog. Η σημερινή έκδοση είναι η IEEE Std. 1800-2017.

Διάγραμμα Ροής Σχεδιασμού Συστημάτων

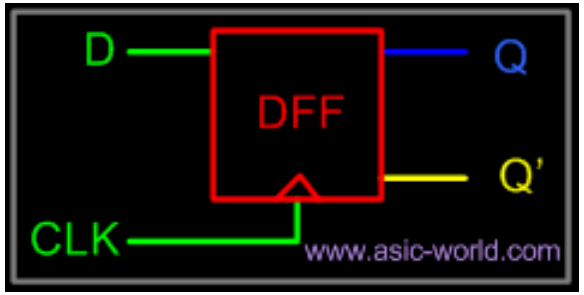
Μορφές/Αναπαραστάσεις Σχεδιασμού με Verilog

Η Verilog μας επιτρέπει να σχεδιάσουμε στα εξής επίπεδα αφαίρεσης:

- **Behavioral level:** η συμπεριφορά περιγράφεται με αλγορίθμους. Κάθε αλγόριθμος αποτελείται από διαδοχικές εντολές. Functions, Tasks και Always είναι τα βασικά στοιχεία στον κώδικα Verilog αυτού του επιπέδου.
- **Register-Transfer Level (RTL)**: Περιγράφονται τα χαρακτηριστικά του κυκλώματος (σήματα) και η μεταφορά των δεδομένων ανάμεσα σε καταχωρητές. Χαρακτηριστικό αυτού του επιπέδου είναι τα σαφή όρια χρονισμού των λειτουργιών κάθε στοιχείου, των σημάτων και των καταχωρητών. Γενικά μπορούμε να πούμε ότι «κάθε κώδικας που συντίθεται είναι RTL κώδικας».
- **Gate Level**: Όπως και στην RTL μορφή, κάθε σήμα και καταχωρητής και οι σχέσεις τους είναι σαφή, όπως και ο χρονισμός τους. Τα σήματα μπορούν να έχουν μόνο (`0', `1', `X', `Z`) τιμές. Τα elements μπορούν να είναι μόνο πρωτογενείς πύλες (AND, OR, NOT κτλ.). Συνήθως δεν γράφουμε σε αυτό το επίπεδο αλλά παράγεται αυτόματα από εργαλεία σύνθεσης.

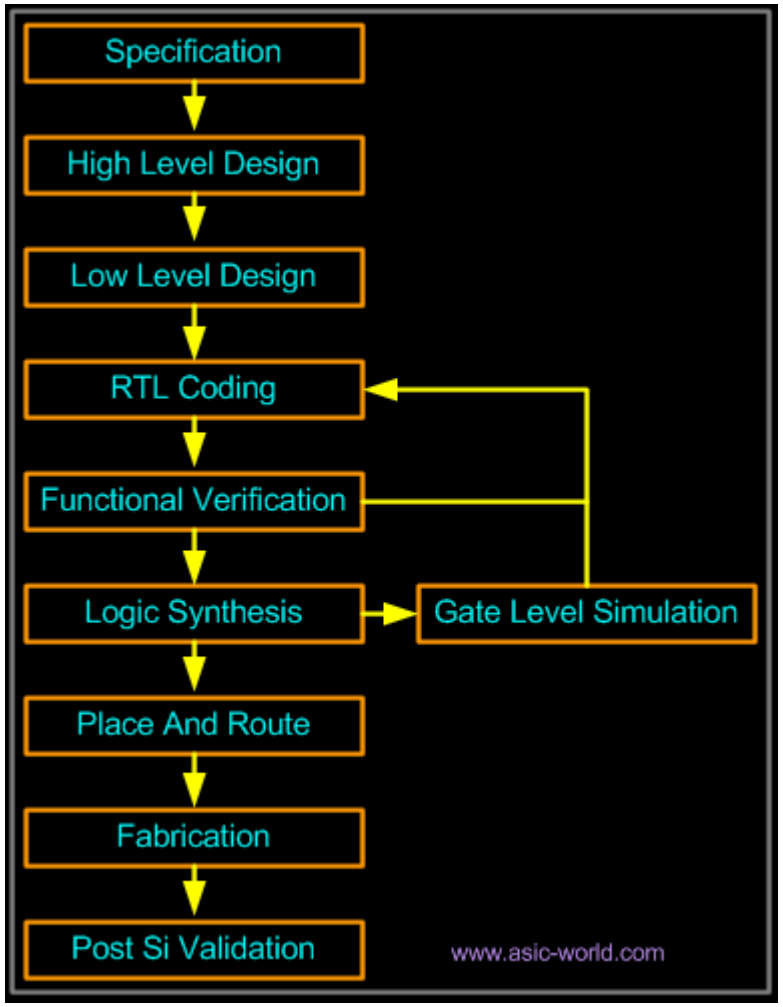



```
module example1(a,b,c,d);
    input a,b,c;
    output d;
    wire a,b,c,d,e,f;

    and E1(e,a,b);
    not E2(f,c);
    and E3(d,e,f);

endmodule
```



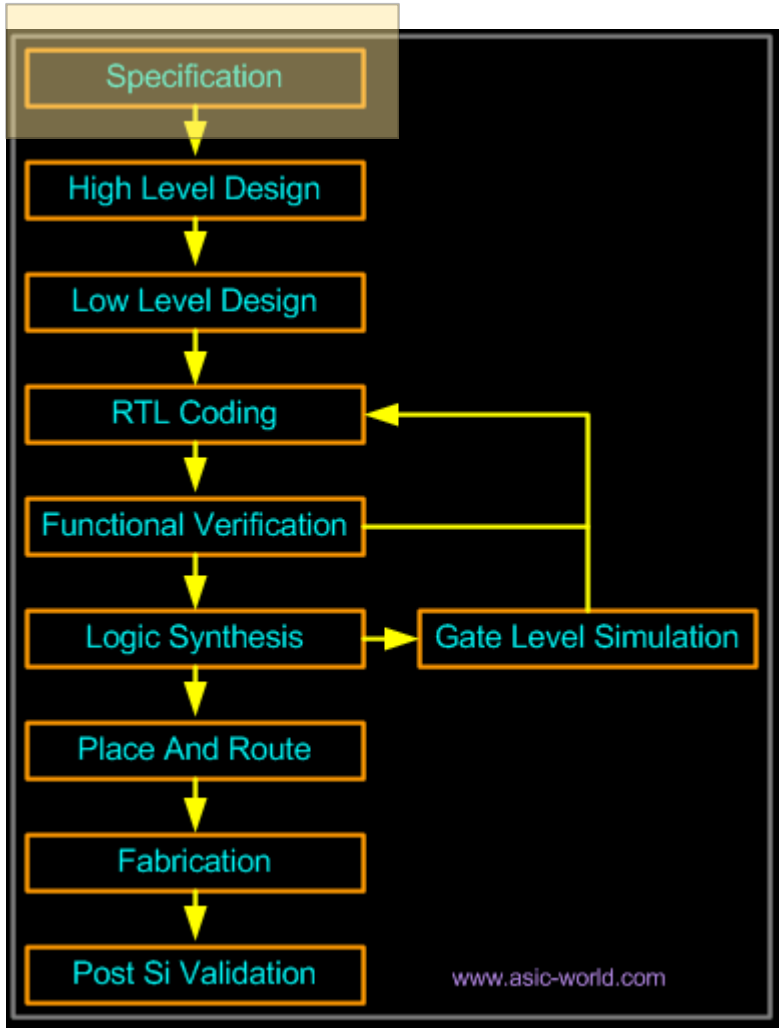

```
// D flip-flop Code
module d_ff ( d, clk, q, q_bar);
    input d ,clk;
    output q, q_bar;
    wire d ,clk;
    reg q, q_bar;
    always @ (posedge clk)
    begin
        q <= d;
        q_bar <= ! d;
    end
endmodule
```

Τυπικό Διάγραμμα Ροής Σχεδιασμού Ψηφιακών Κυκλωμάτων



Εργαλεία που απαιτούνται για κάθε στάδιο σχεδιασμού:
Specification: Επεξεργαστής κειμένου π.χ. Word, Open Office.
High Level Design: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.
Micro Design/Low level design: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.
RTL Coding: Ο αγαπημένος σας Text Editor π.χ. Notepad++, Vim, Emacs
Simulation: Modelsim, VCS, Vivado, Nsim, Verilog-XL, Veriwell, Finsim, Icarus.
Synthesis: Design Compiler, Leonardo Spectrum, Quartus, Vivado. FPGA vendors όπως η Intel και η Xilinx δίνουν τέτοια εργαλεία δωρεάν.
Place & Route: For FPGA specific P&R. ASIC απαιτούν ακριβά P&R εργαλεία όπως το IC compiler και το Encounter.
Post Si Validation: Για Application Specific ICs (ASIC) and Field Programmable Gate Arrays (FPGAs), το Τσιπ ελέγχετε σε κανονικές συνθήκες σε Motherboard με drivers των συσκευών.

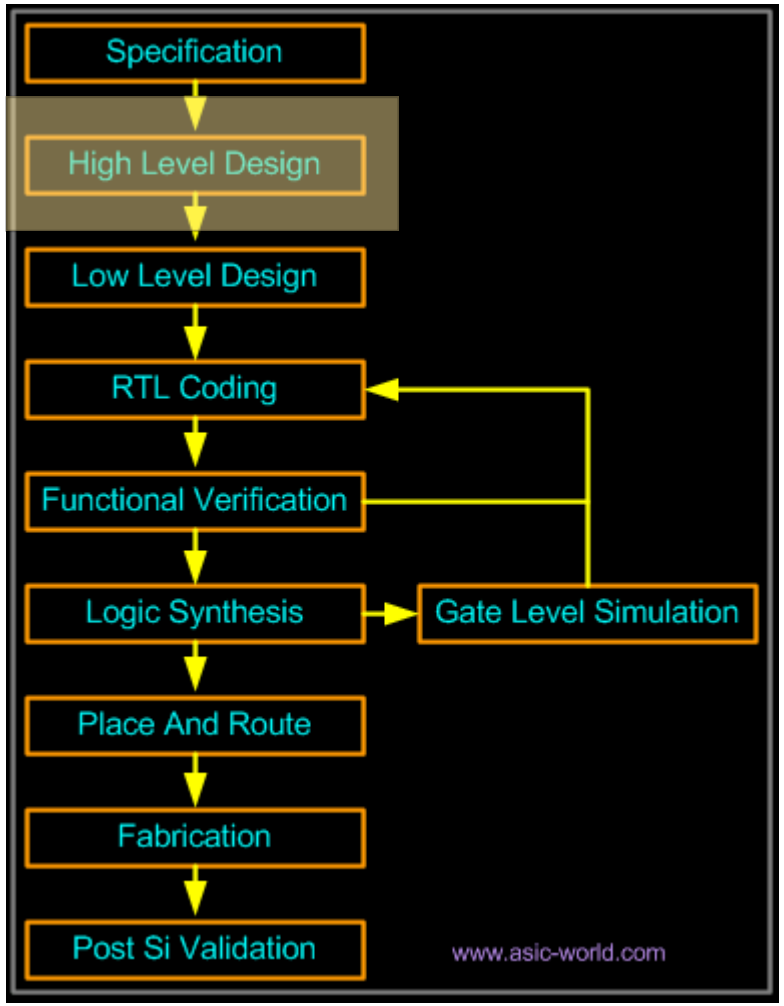
Specification



Specification: Το στάδιο που καθορίζονται οι βασικοί παράμετροι του συστήματος/σχεδιασμού μας.
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office.

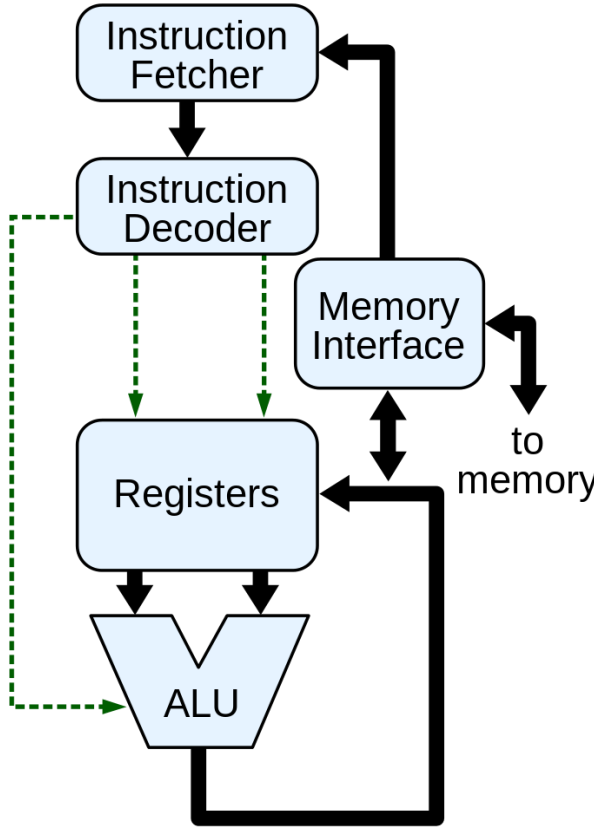
Παράδειγμα: θέλω να σχεδιάσω έναν counter, οποίος θα προσθέτει σήματα των 4 bit με σύγχρονο reset και θα είναι ακμοπυροδότητος (active high enable). Όταν το reset είναι ενεργό θα μηδενίζεται.

High Level Design

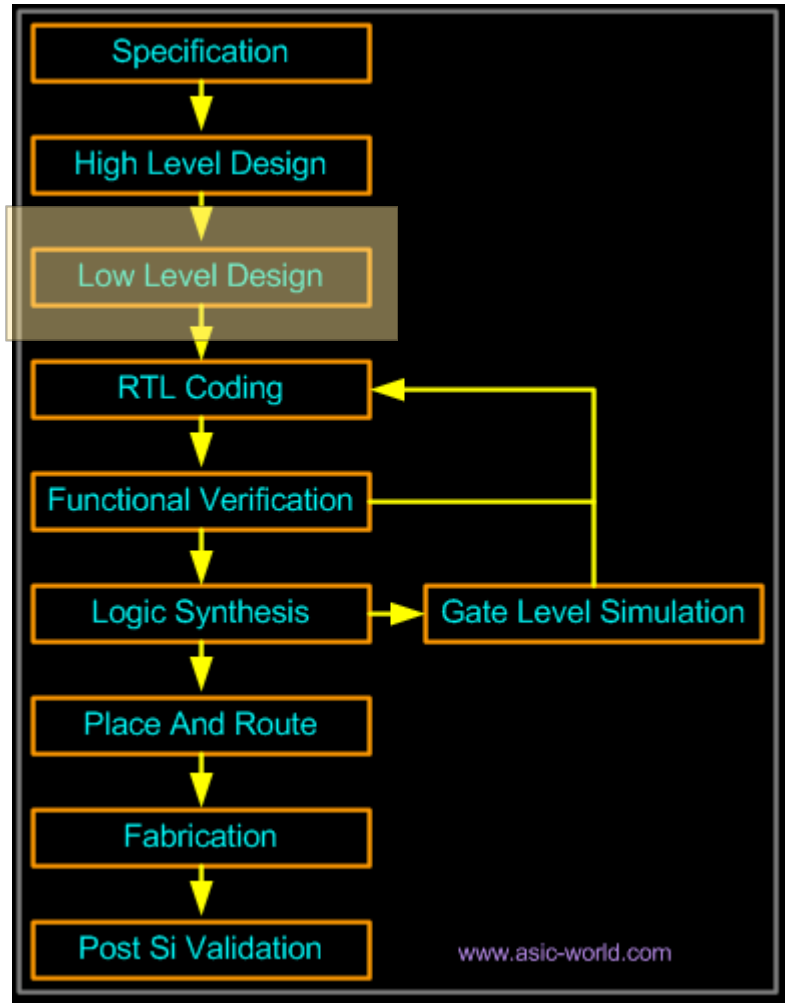


High Level Design: Διαχωρισμός του σχεδίου σε δομικά τμήματα με βάση τη λειτουργία τους και καθορισμός της επικοινωνίας τους. Προσοχή είναι γενικό και σχηματικό το στάδιο αυτό. Δεν εξετάζει πολλές λεπτομέρειες.
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλία σχεδίασης διαγραμμάτων π.χ. Powerpoint, visio. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.

Παράδειγμα: σε έναν επεξεργαστή έχουμε registers, ALU, Instruction Decoder, Memory Interface etc.



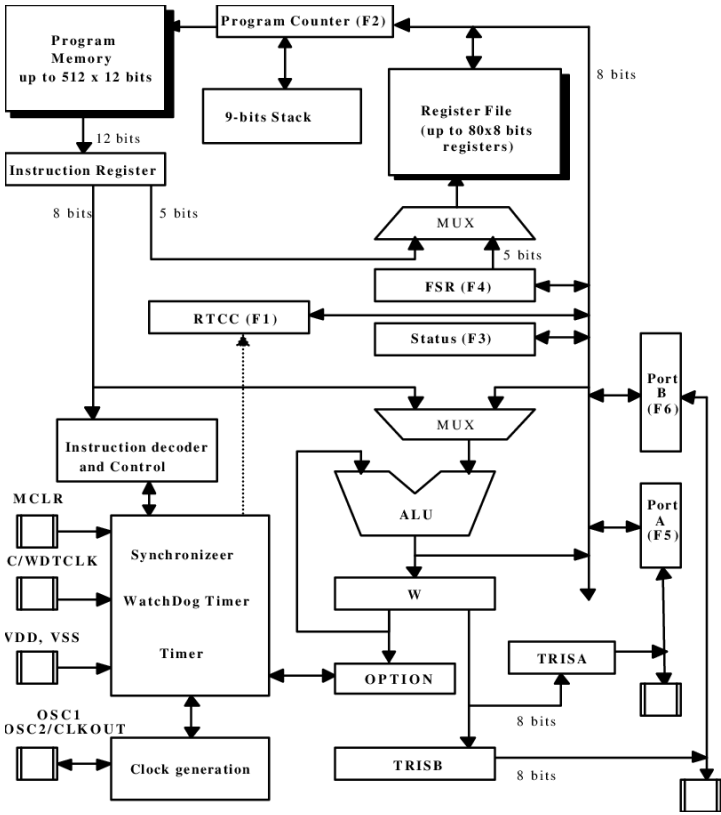
Micro Design - Low Level Design

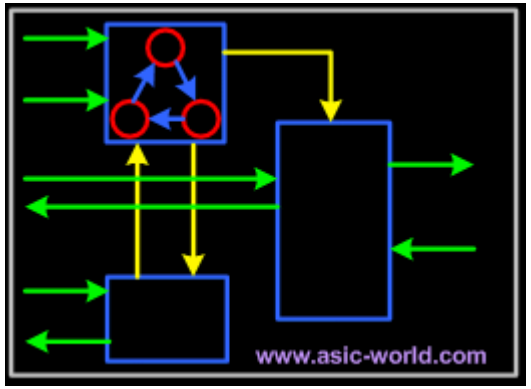


Micro Design ή Low Level Design: είναι το στάδιο που ο σχεδιαστής περιγράφει κάθε τμήμα πως θα υλοποιηθεί. Έχει λεπτομέρειες όπως State machines, counters, Mux, decoders, internal registers. Είναι καλό να σχεδιάζονται και ενδεικτικά waveforms -κυμματομορφές σε αυτό το στάδιο. Είναι το στάδιο που παίρνει τον περισσότερο χρόνο.

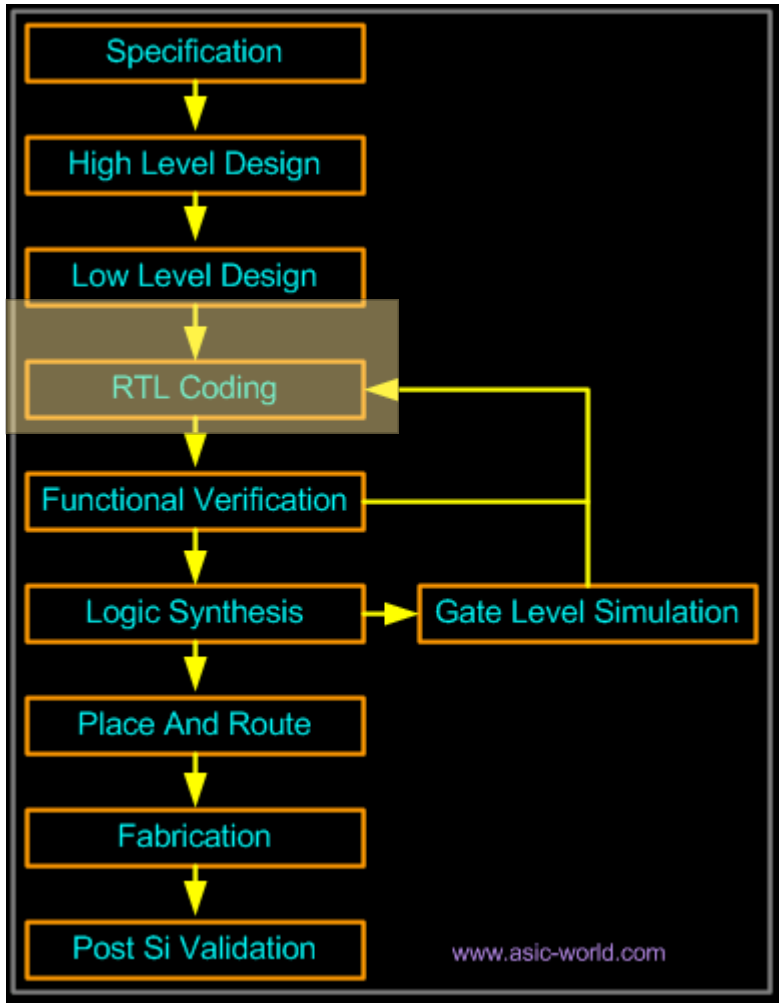
Εργαλεία: Επεξεργαστής κειμένου π.χ. Word, Open Office. Εργαλία σχεδίασης διαγραμμάτων π.χ. Powerpoint, visio. Εργαλεία προβολής χρονοσειρών π.χ. waveformer ή testbencher ή ακόμα και στο Word.

Παράδειγμα: σε έναν επεξεργαστή έχουμε τα βασικά στοιχεία όπως registers, ALU, Instruction Decoder, Memory Interface etc., αλλά έχουμε και control logic που αποτελείται από πολυπλέκτες και μηχανές καταστάσεων, και υλοποιεί τις λεπτομέρειες για τον συντονισμό των στοιχείων.





Register Transfer Level (RTL) Coding

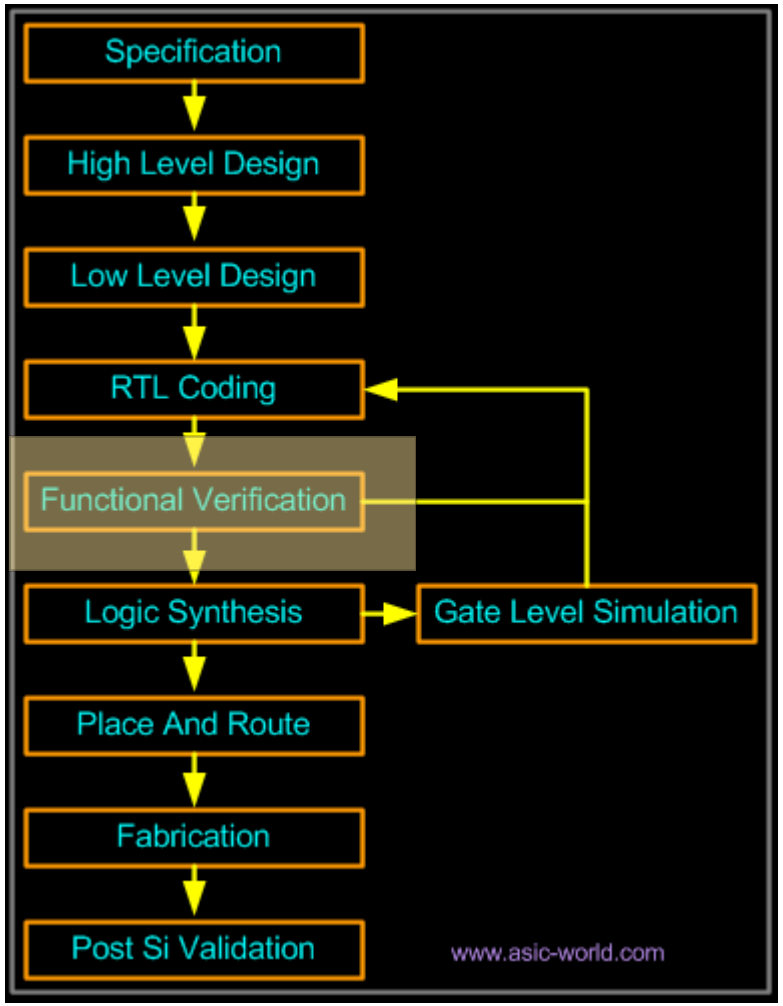


RTL Coding: Στο στάδιο αυτό το Micro design μετατρέπεται σε Verilog/VHDL κώδικα με χρήση συνθέσιμων/κατασκευάσιμων στοιχείων.
Εργαλεία: Ο αγαπημένος σας Text Editor π.χ. Notepad++, Vim, Emacs
Παράδειγμα: στο παράδειγμα βλέπουμε σε Verilog έναν full adder. Προσθέτει δύο bits και το κρατούμενο εισόδου και γράφει στην έξοδο το αποτέλεσμα και το κρατούμενο εξόδου.

```
module addbit (
a      , // first input
b      , // Second input
ci     , // Carry input
sum    , // sum output
co       // carry output
);
//Input declaration
input a, b, ci;
//Ouput declaration
output sum, co;
//Port Data types
wire  a, b, ci, sum, co;
//Code starts here
assign {co,sum} = a + b + ci;
endmodule // End of Module addbit
```

```
module addbit (a, b, ci, sum , co);
   input a, b, ci;
   output sum, co;
   wire  a, b, ci, sum, co;
   assign {co,sum} = a + b + ci;
endmodule // End of Module addbit
```

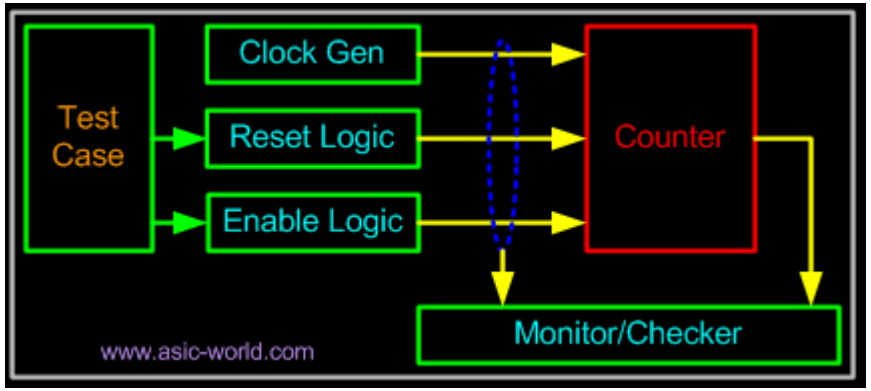
Functional verification



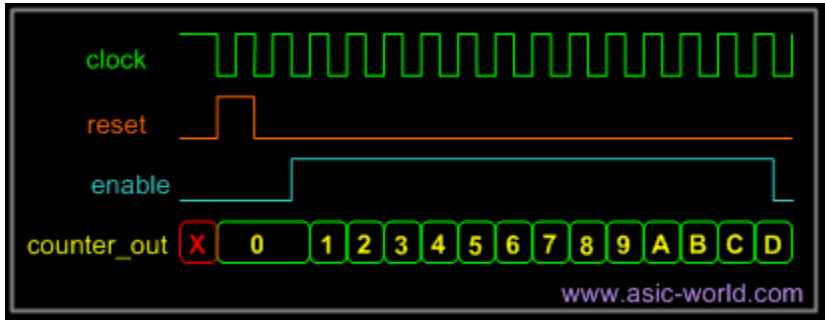
Functional verification : είναι το στάδιο που πιστοποιούμε ότι τα δομικά στοιχεία του σχεδιασμού μας λειτουργούν. Χρησιμοποιούμε προσομοιωτές. Για να ελέγξουμε ότι ο RTL κώδικάς μας πληροί τις προδιαγραφές, πρέπει κάθε δομικό RTL στοιχείο να λειτουργεί σωστά. Για τον λόγο αυτό, γράφουμε ένα testbench, που θέτει εισόδους και ελέγχει αποκρίσεις στο RTL στοιχείο που σχεδιάσαμε. 60-70% του χρόνου σχεδιασμού το σπαταλάμε εδώ.
Εργαλεία: Modelsim, VCS, Vivado, Nsim, Verilog-XL, Veriwell, Finsim, Icarus

Παράδειγμα: στο παράδειγμα βλέπουμε ένα **testbench** για έναν counter.

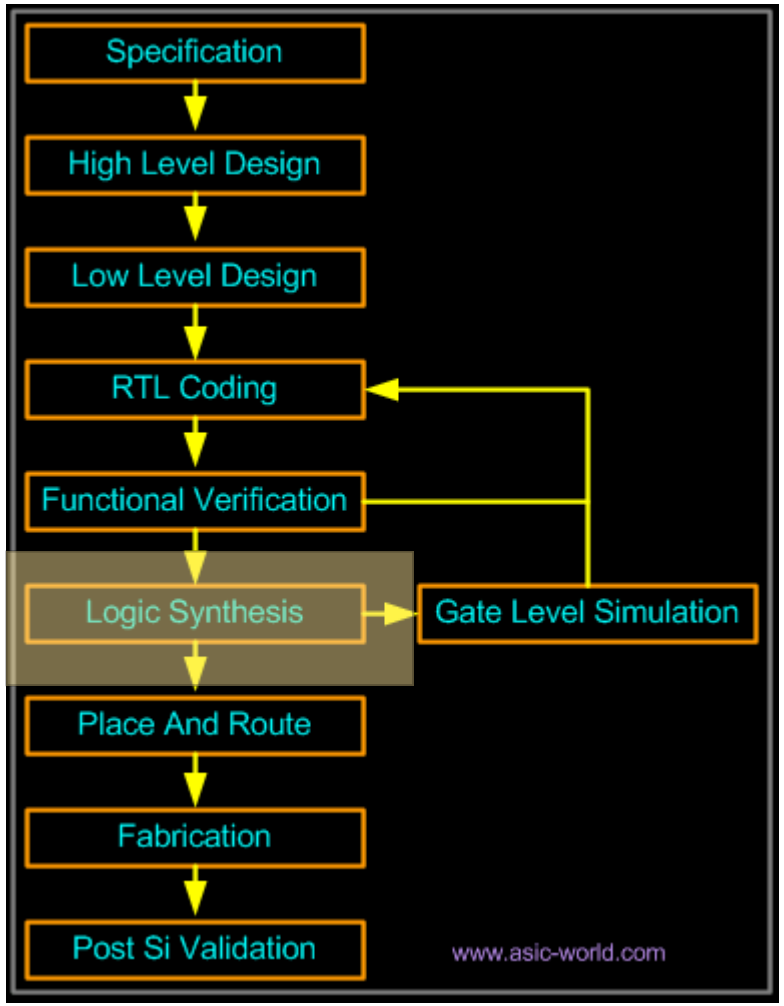
Testbench για τον Counter



Waveform viewer για να βλέπουμε τις χρονοσειρές των σημάτων Εισόδου-εξόδου κατά το Simulation



Logic Synthesis



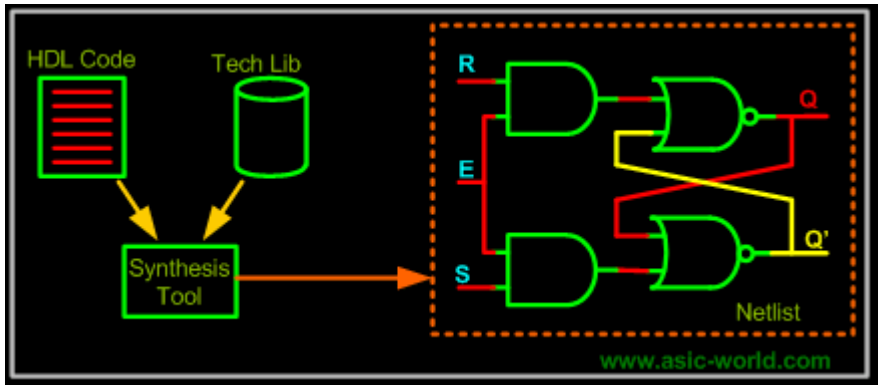
Logic Synthesis: Σύνθεση είναι μια αυτοματοποιημένη διαδικασία την οποία εκτελεί το λογισμικό σύνθεσης το οποίο δέχεται σαν είσοδο:

- RTL κώδικα του υλικού
- Περιγραφή της τεχνολογίας κατασκευής (παρέχεται από τα εργοστάσια κατασκευής)
- Περιορισμούς/προδιαγραφές π.χ. ελάχιστη ταχύτητα ρολογιού, μέγιστη κατανάλωση ισχύος, εμβαδό τσιπ κτλ. DRC (Design Rules Checking)

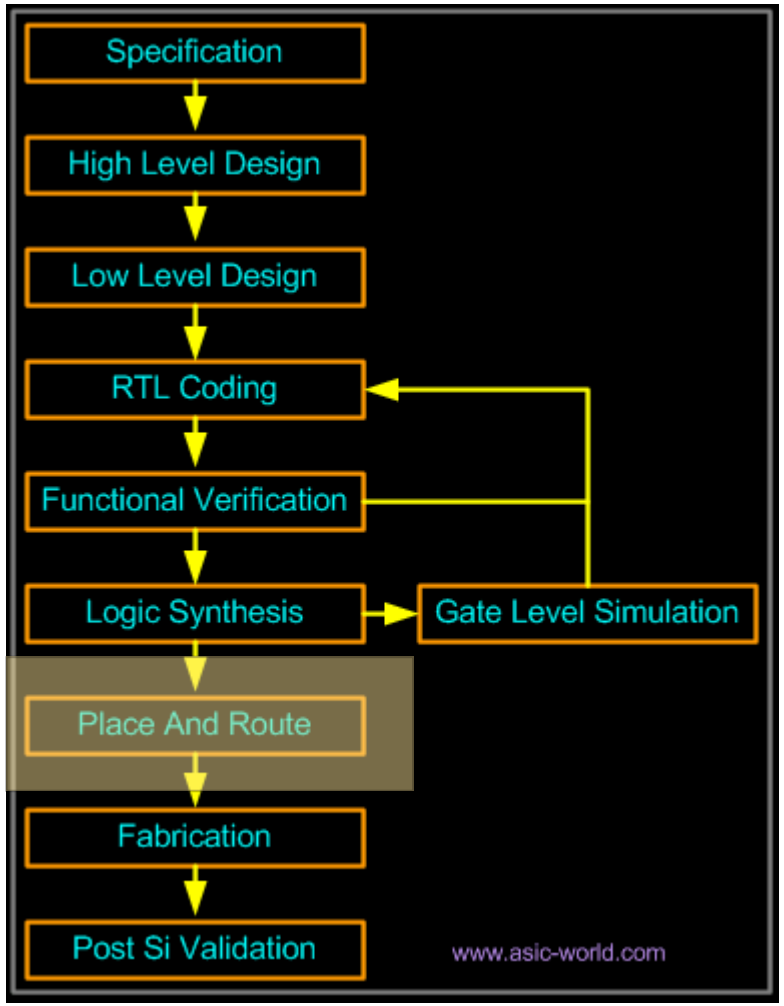
Το λογισμικό σύνθεσης προσπαθεί αυτόματα να παράγει ένα σχέδιο (το **gate-level netlist**) που έχει την ίδια λειτουργικότητα με το παρεχόμενο, αλλά χρησιμοποιώντας μόνο δομικά στοιχεία από την τεχνολογία κατασκευής. Παράλληλα προσπαθεί να ικανοποιήσει τους περιορισμούς.

Εργαλεία: Design Compiler, Leonardo Spectrum, **Quartus, Vivado**. FPGA vendors όπως η Intel και η Xilinx δίνουν τέτοια εργαλεία δωρεάν.

Παράδειγμα: στο παράδειγμα βλέπουμε ένα RTL design και ένα Tech lib να συνθέτουν ένα gate-level netlist ενός flip-flop.

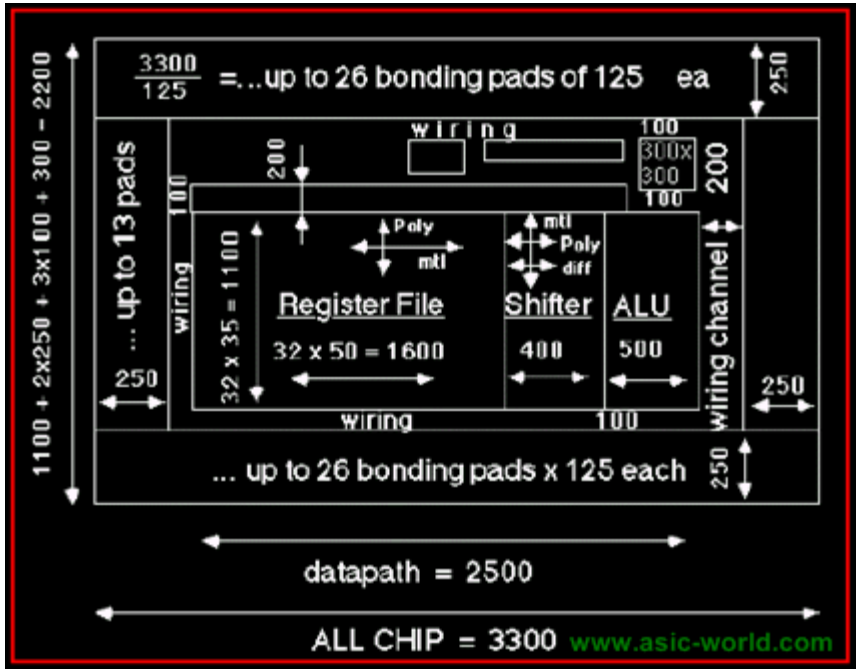


Place and Route

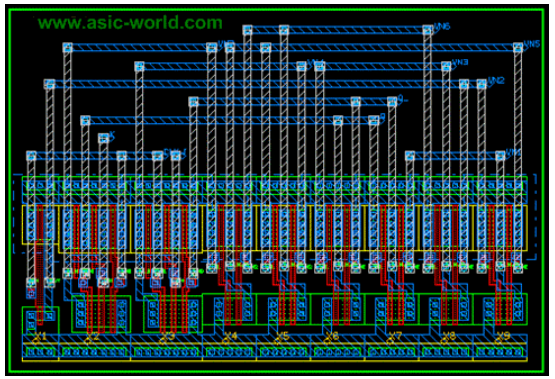


Place and Route : Το gate-level netlist από την σύνθεση επεξεργάζεται από το place and route λογισμικό (P&R ή **layout tools)** σε Verilog gate-level netlist μορφή. Τα gates και flip-flops τοποθετούνται στο τσιπ, το clock-tree συντίθεται και όλα τα στοιχεία συνδέονται με interconnections. Σε ASICs, η έξοδος από το P&R tool είναι το layout (αρχείο GDS) το οποίο χρησιμοποιείται από το foundry για κατασκευή. Σε FPGAs είναι ένα μικροπρόγραμμα (λέγεται bistream), το οποίο χρησιμοποιείται για να αλλάξει το σχεδιασμό του FPGA.
Εργαλεία: For FPGA specific P&R Vivado, Quartus. Τα ASIC απαιτούν ακριβά P&R εργαλεία όπως το IC compiler και το Encounter.

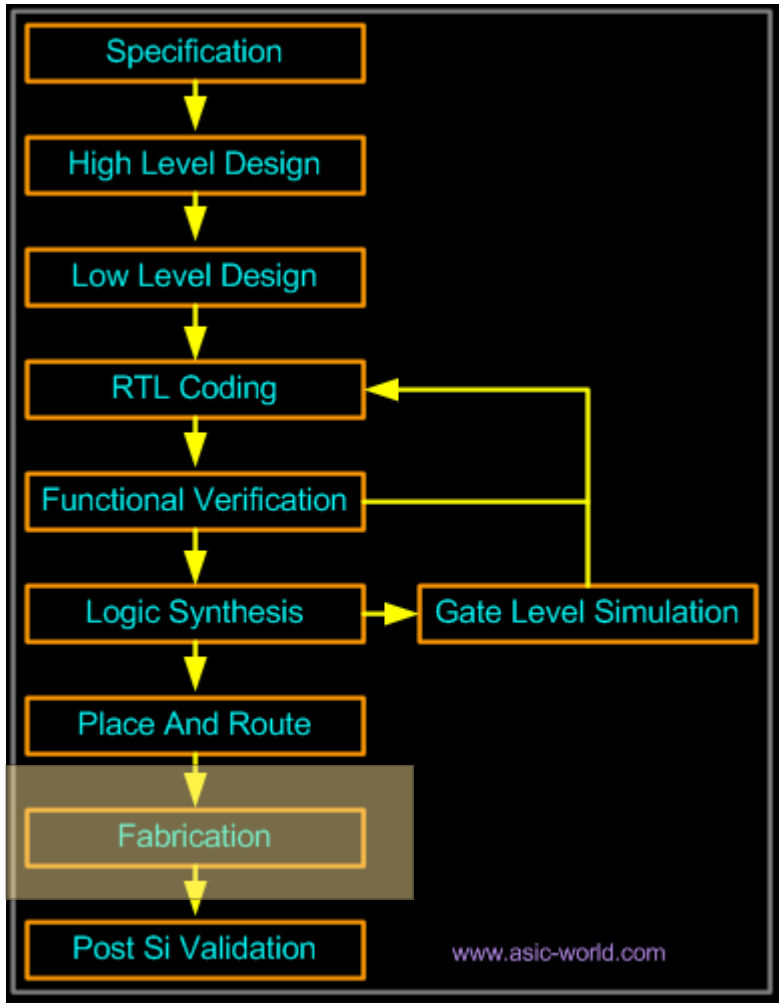
Παράδειγμα: Placement μικροεπεξεργαστή και flip-flop.

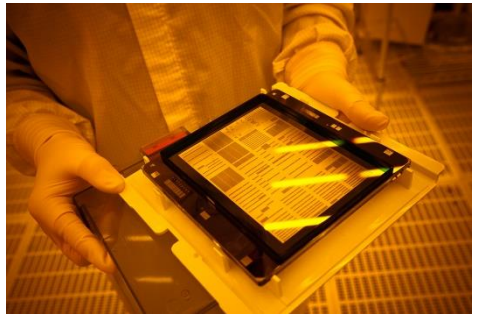


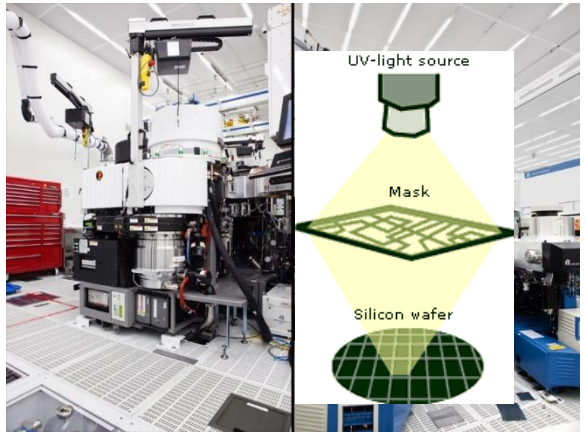
Στο στάδιο αυτό παράγονται και SPEF (standard parasitic exchange format) αρχεία από τα layout tools, όπως το ASTRO, τα οποία μπορούν αν χρησιμοποιηθούν για πιο προσομοιώσεις καλύτερης ακρίβειας



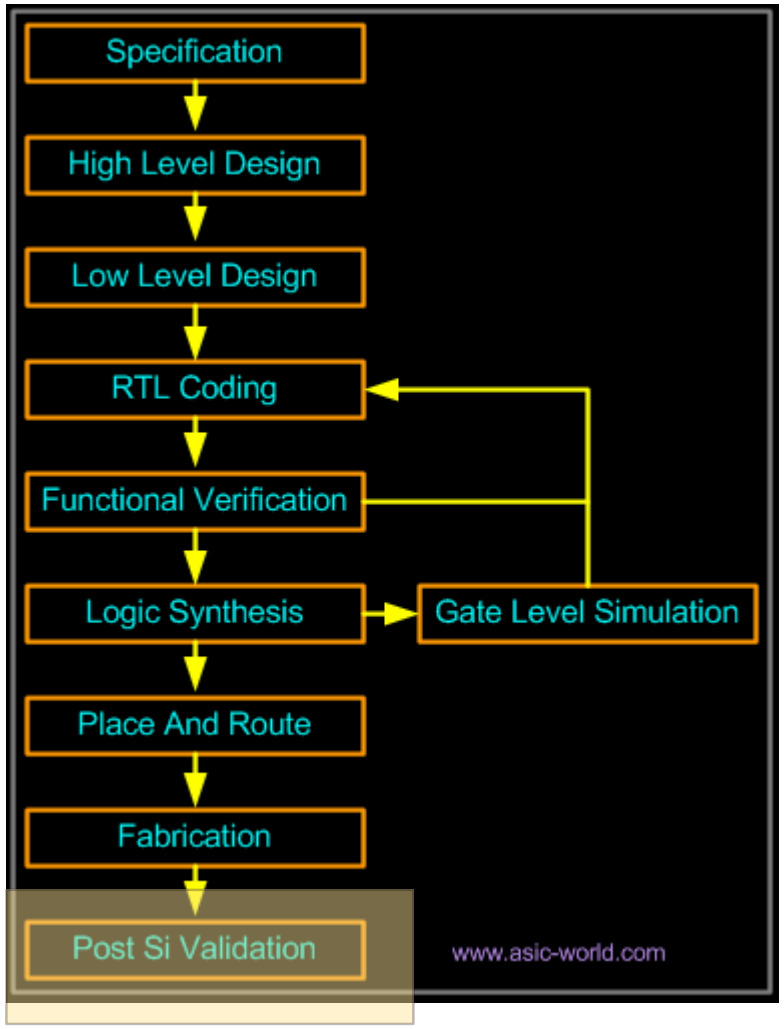
Fabrication







Post Si Validation



Post Silicon Validation: Μόλις κατασκευαστεί το chip (silicon) στο εργαστήριο ελέγχεται σε πραγματικές συνθήκες πριν προωθηθεί στην αγορά. Από τη διαδικασία αυτή μπορεί να εντοπιστούν κάποια bugs στο design, οπότε θα πρέπει να διορθωθεί στο σχεδιασμό.

Παράδειγμα 1 – Hello World

```
module hello_world ;
initial begin
  $display ("Hello World");
  #10 $finish;
end
endmodule // End of Module hello_world
```

Με πράσινο είναι σχόλια και με μπλε είναι δεσμευμένες εκφράσεις.

Κάθε πρόγραμμα ξεκινάει με 'module' <module_name>.

Στο παραπάνω παράδειγμα έχουμε module hello_world. (προσέξτε: μπορούμε να έχουμε pre-processor statements όπως `include', `define' πριν από τη δήλωση του module)

Στη 2η γραμμή έχουμε το ξεκίνημα ενός **μπλοκ κώδικα initial** : εκτελείτε μόνο μια φορά αμέσως μετά το ξεκίνημα της προσομοίωσης τη χρονική στιγμή time=0 (0ns). Περιέχει 2 εντολές ανάμεσα στο begin και το end.

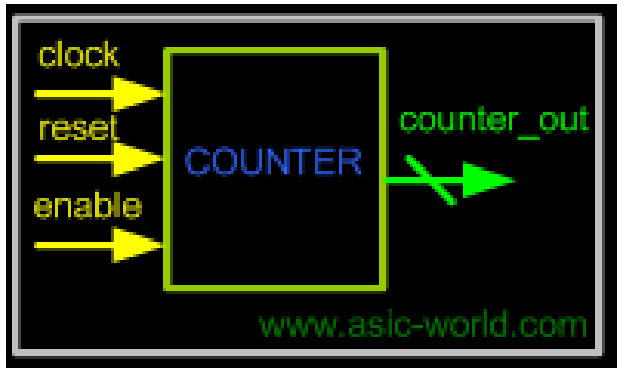
Όλα τα μπλοκ ξεκινούμε με begin και τελειώνουν με end.

Το ‘Module’ τελειώνει με 'endmodule'.

Παράδειγμα 2 – 4bit counter

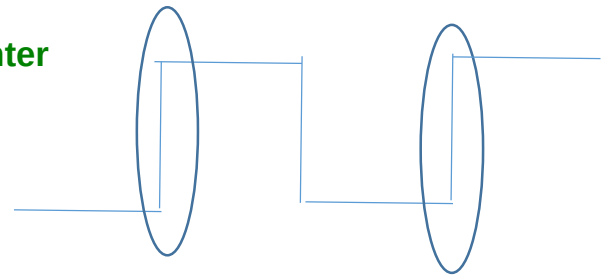
Specification
•4-bit synchronous up counter.
•active high, synchronous reset.
•Active high enable.

- Δηλώνουμε όλες τις εισόδους εξόδους
- Δηλώνουμε και τον τύπο τους είτε wire είτε reg
  - Το wire είναι απλό καλώδιο
  - Το reg θα μετατραπεί σε flip-flop, είναι σαν να βάζουμε καταχωρητή 1 bit
- Τα «always συνθήκη» μπλοκς εκτελούνται όταν ικανοποιείτε η συνθήκη τους – κατ'επανάληψη




```
//------------------------------------------------------
// This is my second Verilog Design
// Design Name : first_counter
// File Name : first_counter.v
// Function : This is a 4 bit up-counter with
// Synchronous active high reset and
// with active high enable signal
//------------------------------------------------------
module first_counter (
clock , // Clock input of the design
reset , // active high, synchronous Reset input
enable , // Active high enable signal for counter
counter_out // 4 bit vector output of the counter
); // End of port list
//-------------Input Ports----------------------------
input clock ;
input reset ;
input enable ;
//-------------Output Ports---------------------------
output [3:0] counter_out ;
//-------------Input ports Data Type-------------------
// By rule all the input ports should be wires
wire clock ;
wire reset ;
wire enable ;
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [3:0] counter_out ;
```

```

//------------Code Starts Here------------------------
// Since this counter is a positive edge trigged one,
// We trigger the below block with respect to positive
// edge of the clock.
always @ (posedge clock)
begin : COUNTER // Block Name
// At every rising edge of clock we check if reset is active
// If active, we load the counter output with 4'b0000
if (reset == 1'b1) begin
  counter_out <= #1 4'b0000;
end
// If enable is active, then we increment the counter
else if (enable == 1'b1) begin
  counter_out <= #1 counter_out + 1;
end
end // End of Block COUNTER

endmodule // End of Module counter
```



Syntax and Semantics

Αναπαράσταση αριθμών

Σταθερές μπορούν να αναπαρασταθούν σε δεκαδικό, δεκαεξαδικό, οκταδικό ή δυαδικό σύστημα βάσης (μορφή).

Αρνητικοί αναπαρίστανται με το συμπλήρωμα ως προς 2.

Όταν χρησιμοποιείται σε αριθμό το (?) τότε είναι το ‘z’ σύμβολο που σημαίνει unconnected/high impedance.

Ο χαρακτήρας underscore (_) είναι νόμιμος μέσα σε αριθμό, εκτός από το πρώτο ψηφίο όπου αγνοείται.

Ακέραιοι Αριθμοί (Integers)

Ακέραιοι Αριθμοί (**Integers**) ορίζονται ως:

- Sized or unsized numbers (Unsized size is 32 bits)
- In a radix of binary, octal, decimal, or hexadecimal
- Radix and hex digits (a,b,c,d,e,f) are case insensitive
- Spaces are allowed between the size, radix and value

Syntax: <size>'<radix><value>;

- **Radix**
  - **h for hexadecimal**
  - b for binary
  - d for decimal

Integer	Stored as
1	00000000000000000000000000000001
8'hAA	10101010
6'b10_0011	100011
'hF	00000000000000000000000000001111

Verilog expands <value> filling the specified <size> by working from right-to-left
When <size> is smaller than <value>, then leftmost bits of <value> are truncated
When <size> is larger than <value>, then leftmost bits are filled, based on the value of the leftmost bit in <value>.
Leftmost '0' or '1' are filled with '0'
Leftmost 'Z' are filled with 'Z'
Leftmost 'X' are filled with 'X

Note : X Stands for unknown and Z stands for high impedance, 1 for logic high or 1 and 0 for logic low or 0

Πραγματικοί Αριθμοί (Real)

Verilog supports real constants and variables
Verilog converts real numbers to integers by rounding
Real Numbers can not contain 'Z' and 'X'
Real numbers may be specified in either decimal or scientific notation
< value >.< value >
< mantissa >E< exponent >
Real numbers are rounded off to the nearest integer when assigning to an integer

Real Number	Decimal notation
1.2	1.2
0.6	0.6
3.5E6	3,500000.0

Προσημασμένοι αριθμοί

Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus they become signed numbers. Verilog internally represents negative numbers in 2's complement format. An optional signed specifier can be added for signed arithmetic.

Number	Description
32'hDEAD_BEEF	Unsigned or signed positive number
-14'h1234	Signed negative number

```
1  module signed_number;
2
3  reg [31:0]  a;
4
5  initial begin
6    a = 14'h1234;
7    $display ("Current Value of a = %h", a);
8    a = -14'h1234;
9    $display ("Current Value of a = %h", a);
10   a = 32'hDEAD_BEEF;
11   $display ("Current Value of a = %h", a);
12   a = -32'hDEAD_BEEF;
13   $display ("Current Value of a = %h", a);
14    #10  $finish;
15 end
16
17 endmodule
```

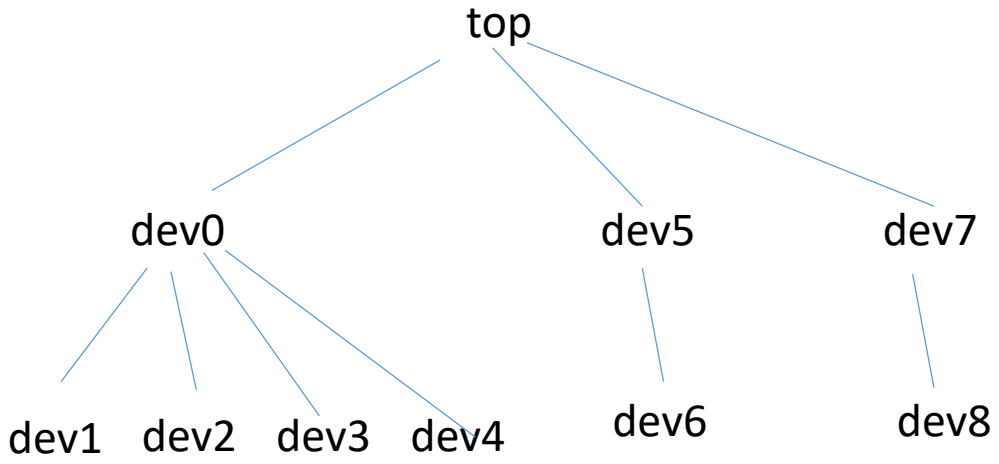
```
Current Value of a = 00001234
Current Value of a = ffffedcc
Current Value of a = deadbeef
Current Value of a = 21524111
```

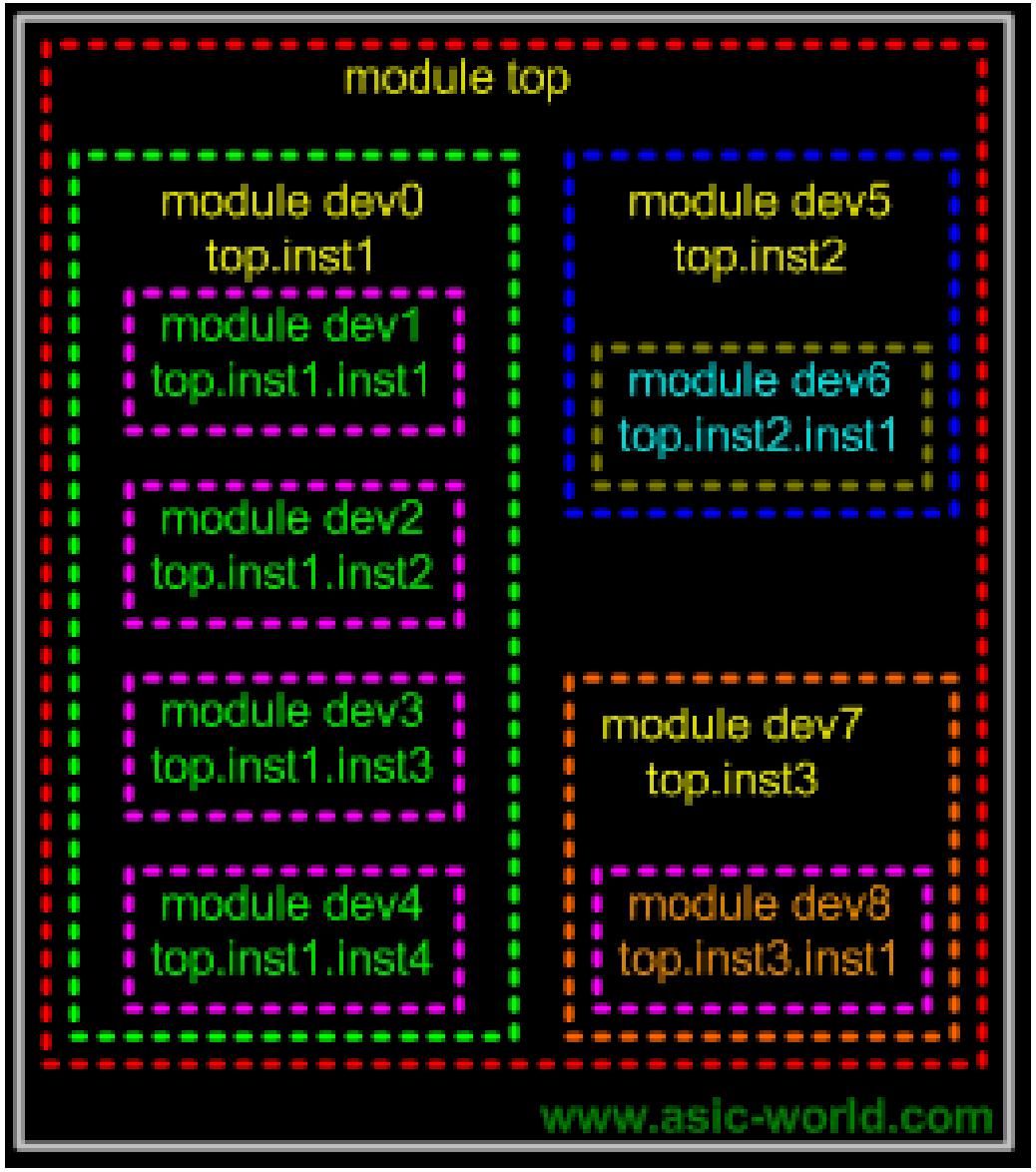
Modules

Modules are the building blocks of Verilog designs

You create the design hierarchy by instantiating modules in other modules

You instance a module when you use that module in another, higher-level module





Ports

Ports allow communication between a module and its environment.

All but the top-level modules in a hierarchy have ports.

Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is :

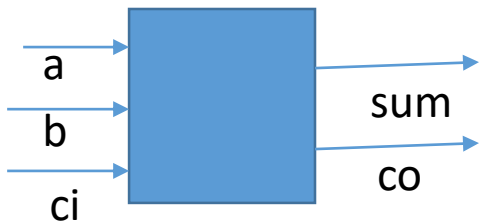
input [range_val:range_var] list_of_identifiers;
output [range_val:range_var] list_of_identifiers;
inout [range_val:range_var] list_of_identifiers;

Examples : Port Declaration

```
1 input          clk      ; // clock input
2 input  [15:0]  data_in  ; // 16 bit data input bus
3 output [7:0]   count    ; // 8 bit counter output
4 inout          data_bi  ; // Bi-Directional data bus
```

Παράδειγμα 3 – full adder -1bit counter

```
1  module addbit (
2  a      , // first input
3  b      , // Second input
4  ci     , // Carry input
5  sum    , // sum output
6  co       // carry output
7  );
8  //Input declaration
9  input a;
10 input b;
11 input ci;
12 //Ouput declaration
13 output sum;
14 output co;
15 //Port Data types
16 wire   a;
17 wire   b;
18 wire   ci;
19 wire   sum;
20 wire   co;
21 //Code starts here
22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit
```

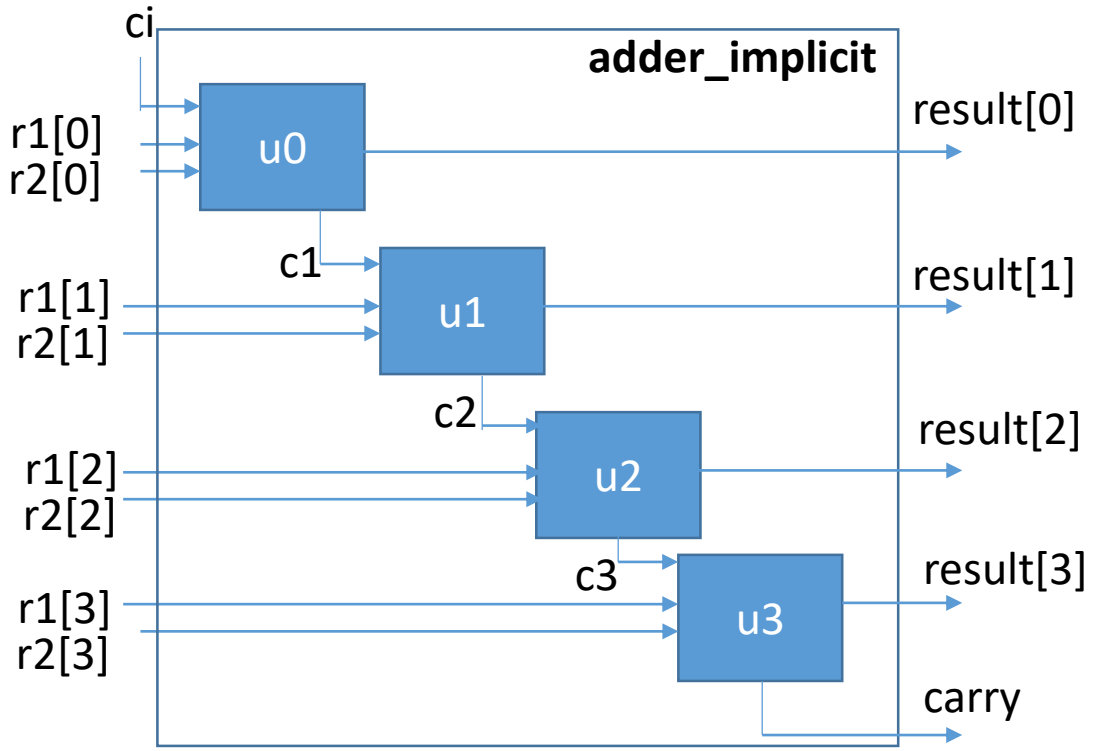


Παράδειγμα 4. 4bit-Adder (αθροιστής ριπής)

```
1  //------------------------------------------------
2  // This is simple adder Program
3  // Design Name : adder_implicit
4  // File Name   : adder_implicit.v
5  // Function    : This program shows how implicit
6  //               port connection are done
7  // Coder       : Deepak Kumar Tala
8  //------------------------------------------------
9  module adder_implicit (
10 result        , // Output of the adder
11 carry         , // Carry output of adder
12 r1            , // first input
13 r2            , // second input
14 ci              // carry input
15 );
16
17 // Input Port Declarations
18 input  [3:0]  r1        ;
19 input  [3:0]  r2        ;
20 input         ci        ;
21
22 // Output Port Declarations
23 output [3:0]  result    ;
24 output        carry     ;
25
26 // Port Wires
27 wire   [3:0]  r1        ;
28 wire   [3:0]  r2        ;
29 wire          ci        ;
30 wire   [3:0]  result    ;
31 wire          carry     ;
32
33 // Internal variables
34 wire          c1        ;
35 wire          c2        ;
36 wire          c3        ;
37
```

Modules connected by port order (implicit)

```
38 // Code Starts Here
39 addbit u0 (
40 r1[0]          ,
41 r2[0]          ,
42 ci             ,
43 result[0]      ,
44 c1
45 );
46
47 addbit u1 (
48 r1[1]          ,
49 r2[1]          ,
50 c1             ,
51 result[1]      ,
52 c2
53 );
54
55 addbit u2 (
56 r1[2]          ,
57 r2[2]          ,
58 c2             ,
59 result[2]      ,
60 c3
61 );
62
63 addbit u3 (
64 r1[3]          ,
65 r2[3]          ,
66 c3             ,
67 result[3]      ,
68 carry
69 );
70
71 endmodule // End Of Module adder
```



Modules instantiation connectivity

```
1  //------------------------------------------------------
2  // This is simple adder Program
3  // Design Name : adder_implicit
4  // File Name   : adder_implicit.v
5  // Function    : This program shows how implicit
6  //               port connection are done
7  // Coder       : Deepak Kumar Tala
8  //------------------------------------------------------
9  module adder_implicit (
10 result      , // Output of the adder
11 carry       , // Carry output of adder
12 r1          , // first input
13 r2          , // second input
14 ci            // carry input
15 );
16
17 // Input Port Declarations
18 input  [3:0]  r1      ;
19 input  [3:0]  r2      ;
20 input         ci      ;
21
22 // Output Port Declarations
23 output [3:0]  result  ;
24 output        carry   ;
25
26 // Port Wires
27 wire   [3:0]  r1      ;
28 wire   [3:0]  r2      ;
29 wire         ci      ;
30 wire   [3:0]  result  ;
31 wire         carry   ;
32
33 // Internal variables
34 wire         c1      ;
35 wire         c2      ;
36 wire         c3      ;
37
```

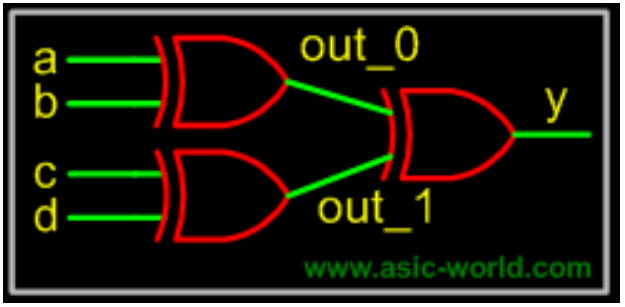
Modules connected by port order (implicit)

```
38 // Code Starts Here
39 addbit u0 (
40 r1[0]          ,
41 r2[0]          ,
42 ci             ,
43 result[0]      ,
44 c1
45 );
46
47 addbit u1 (
48 r1[1]          ,
49 r2[1]          ,
50 c1             ,
51 result[1]      ,
52 c2
53 );
54
55 addbit u2 (
56 r1[2]          ,
57 r2[2]          ,
58 c2             ,
59 result[2]      ,
60 c3
61 );
62
63 addbit u3 (
64 r1[3]          ,
65 r2[3]          ,
66 c3             ,
67 result[3]      ,
68 carry
69 );
70
71 endmodule // End Of Module adder
```

Modules connected by name

```
38 // Code Starts Here
39 addbit u0 (
40 .a            (r1[0])        ,
41 .b            (r2[0])        ,
42 .ci           (ci)           ,
43 .sum          (result[0])    ,
44 .co           (c1)
45 );
46
47 addbit u1 (
48 .a            (r1[1])        ,
49 .b            (r2[1])        ,
50 .ci           (c1)           ,
51 .sum          (result[1])    ,
52 .co           (c2)
53 );
54
55 addbit u2 (
56 .a            (r1[2])        ,
57 .b            (r2[2])        ,
58 .ci           (c2)           ,
59 .sum          (result[2])    ,
60 .co           (c3)
61 );
62
63 addbit u3 (
64 .a            (r1[3])        ,
65 .b            (r2[3])        ,
66 .ci           (c3)           ,
67 .sum          (result[3])    ,
68 .co           (carry)
69 );
70
71 endmodule // End Of Module adder
```

Παράδειγμα 5. Parity checker




```
1  //-----------------------------------------------------
2  // This is simple parity Program
3  // Design Name : parity
4  // File Name   : parity.v
5  // Function    : This program shows how a verilog
6  //               primitive/module port connection are done
7  // Coder       : Deepak
8  //-----------------------------------------------------
9  module parity (
10 a       , // First input
11 b       , // Second input
12 c       , // Third Input
13 d       , // Fourth Input
14 y         // Parity  output
15 );
16
17 // Input Declaration
18 input     a       ;
19 input     b       ;
20 input     c       ;
21 input     d       ;
22 // Ouput Declaration
23 output    y       ;
24 // port data types
25 wire      a       ;
26 wire      b       ;
27 wire      c       ;
28 wire      d       ;
29 wire      y       ;
30 // Internal variables
31 wire      out_0 ;
32 wire      out_1 ;
33
34 // Code starts Here
35 xor u0 (out_0,a,b);
36
37 xor u1 (out_1,c,d);
38
39 xor u2 (y,out_0,out_1);
40
41 endmodule  // End Of Module parity
```

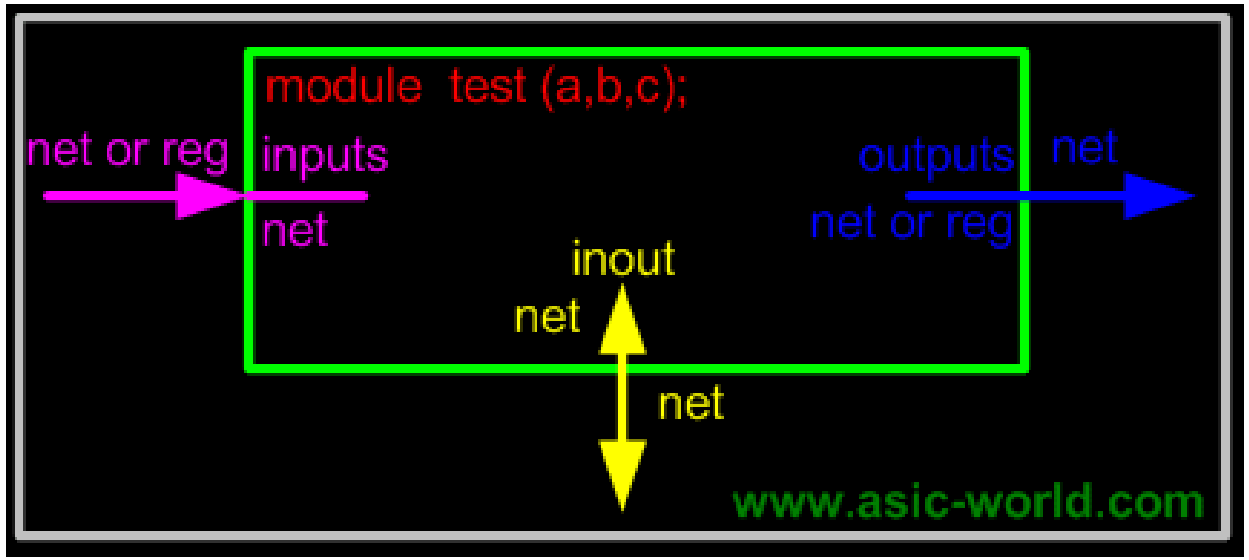
Question : What is the difference between u0 in module adder and u0 in module parity?

Port connection rules

Inputs : internally must always be of type net, externally the inputs can be connected to a variable of type reg or net.

Outputs : internally can be of type net or reg, externally the outputs must be connected to a variable of type net.

Inouts : internally or externally must always be type net, can only be connected to a variable net type.



Width matching : It is legal to connect internal and external ports of different sizes. But beware, synthesis tools could report problems:
Unconnected ports: unconnected ports are allowed by using a ",".
The **net/wire** data type is used to connect structure.
A **net/wire** type is required if a signal is driven.

Παράδειγμα 6: unconnected ports

Implicit Unconnected Port

```
1  module implicit();
2  reg clk,d,rst,pre;
3  wire q;
4
5  // Here second port is not connected
6  dff u0 ( q,,clk,d,rst,pre);
7
8  endmodule
9
10 // D fli-flop
11 module dff (q, q_bar, clk, d, rst, pre);
12 input clk, d, rst, pre;
13 output q, q_bar;
14 reg q;
15
16 assign q_bar = ~q;
17
18 always @ (posedge clk)
19 if (rst == 1'b1) begin
20   q <= 0;
21 end else if (pre == 1'b1) begin
22   q <= 1;
23 end else begin
24   q <= d;
25 end
26
27 endmodule
```

Explicit Unconnected Port

```
1  module explicit();
2  reg clk,d,rst,pre;
3  wire q;
4
5  // Here q_bar is not connected
6  // We can connect ports in any order
7  dff u0 (
8  .q          (q),
9  .d  (d),
10 .clk        (clk),
11 .q_bar      (),
12 .rst        (rst),
13 .pre        (pre)
14 );
15
16 endmodule
17
18 // D fli-flop
19 module dff (q, q_bar, clk, d, rst, pre);
20 input clk, d, rst, pre;
21 output q, q_bar;
22 reg q;
23
24 assign q_bar = ~q;
25
26 always @ (posedge clk)
27 if (rst == 1'b1) begin
28   q <= 0;
29 end else if (pre == 1'b1) begin
30   q <= 1;
31 end else begin
32   q <= d;
33 end
34
35 endmodule
```

Hierarchical identifiers

Hierarchical path names are based on the top module identifier followed by module instant identifiers, separated by periods.

This is useful basically when we want to see the signal inside a lower module, or want to force a value inside an internal module. The example below shows how to monitor the value of an internal module signal.

```
1  //------------------------------------------------------
2  // This is simple adder Program
3  // Design Name : adder_hier
4  // File Name   : adder_hier.v
5  // Function    : This program shows verilog hier path works
6  // Coder       : Deepak
7  //------------------------------------------------------
8  `include "addbit.v"
9  module adder_hier (
10 result        , // Output of the adder
11 carry         , // Carry output of adder
12 r1            , // first input
13 r2            , // second input
14 ci              // carry input
15 );
16
17 // Input Port Declarations
18 input    [3:0]   r1        ;
19 input    [3:0]   r2        ;
20 input            ci        ;
21
22 // Output Port Declarations
23 output   [3:0]  result     ;
24 output          carry      ;
25
26 // Port Wires
27 wire     [3:0]   r1        ;
28 wire     [3:0]   r2        ;
29 wire             ci        ;
30 wire     [3:0]   result    ;
31 wire             carry     ;
32
33 // Internal variables
34 wire             c1        ;
35 wire             c2        ;
36 wire             c3        ;
37
38 // Code Starts Here
39 addbit u0 (r1[0],r2[0],ci,result[0],c1);
40 addbit u1 (r1[1],r2[1],c1,result[1],c2);
41 addbit u2 (r1[2],r2[2],c2,result[2],c3);
42 addbit u3 (r1[3],r2[3],c3,result[3],carry);
43
44 endmodule  // End Of Module adder
45
46 module tb();
47
48 reg [3:0] r1,r2;
49 reg  ci;
50 wire [3:0] result;
51 wire  carry;
52
53 // Drive the inputs
54 initial begin
55   r1 = 0;
56   r2 = 0;
57   ci = 0;
58   #10  r1 = 10;
59   #10  r2 = 2;
60   #10  ci = 1;
61   #10  $display("+-----------------------------------------------------+");
62   $finish;
63 end
64
65 // Connect the lower module
66 adder_hier U (result,carry,r1,r2,ci);
67
68 // Hier demo here
69 initial begin
70   $display("+-----------------------------------------------------+");
71   $display("| r1 | r2 | ci | u0.sum | u1.sum | u2.sum | u3.sum |");
72   $display("+-----------------------------------------------------+");
73   $monitor("| %h  | %h  | %h  |  %h  | %h  | %h  | %h  |",
74   r1,r2,ci, tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum, tb.U.u3.sum);
75 end
76
77 endmodule
```

Simulator output:

```
+-----------------------------------------------------+
| r1 | r2 | ci | u0.sum | u1.sum | u2.sum | u3.sum |
+-----------------------------------------------------+
| 0  | 0  | 0  |  0  | 0  | 0  | 0  |
| a  | 0  | 0  |  0  | 1  | 0  | 1  |
| a  | 2  | 0  |  0  | 0  | 1  | 1  |
| a  | 2  | 1  |  1  | 0  | 1  | 1  |
+-----------------------------------------------------+
```

Data types

Verilog Language has two primary data types:

Nets - represent structural connections between components.
Registers - represent variables used to store data.

Every signal has a data type associated with it:
Explicitly declared with a declaration in your Verilog code.
Implicitly declared with no declaration when used to connect structural building blocks in your code.
Implicit declaration is always a net type "wire" and is one bit wide.

Types of Nets

Each net type has a functionality that is used to model different types of hardware (such as PMOS, NMOS,

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate).

Note : Of all net types, wire is the one which is most widely used.

Example - wor

```
1  module test_wor();
2
3  wor a;
4  reg b, c;
5
6  assign a = b;
7  assign a = c;
8
9  initial begin
10   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
11   #1 b = 0;
12   #1 c = 0;
13   #1 b = 1;
14   #1 b = 0;
15   #1 c = 1;
16   #1 b = 1;
17   #1 b = 0;
18   #1 $finish;
19 end
20
21 endmodule
```

Simulator Output

```
0 a = x b = x c = x
1 a = x b = 0 c = x
2 a = 0 b = 0 c = 0
3 a = 1 b = 1 c = 0
4 a = 0 b = 0 c = 0
5 a = 1 b = 0 c = 1
6 a = 1 b = 1 c = 1
7 a = 1 b = 0 c = 1
```

Nets wor and wand

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate).

Example - wor

```
1  module test_wor();
2
3  wor a;
4  reg b, c;
5
6  assign a = b;
7  assign a = c;
8
9  initial begin
10   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
11   #1 b = 0;
12   #1 c = 0;
13   #1 b = 1;
14   #1 b = 0;
15   #1 c = 1;
16   #1 b = 1;
17   #1 b = 0;
18   #1 $finish;
19 end
20
21 endmodule
```

Simulator Output

```
0 a = x b = x c = x
1 a = x b = 0 c = x
2 a = 0 b = 0 c = 0
3 a = 1 b = 1 c = 0
4 a = 0 b = 0 c = 0
5 a = 1 b = 0 c = 1
6 a = 1 b = 1 c = 1
7 a = 1 b = 0 c = 1
```

Example - wand

```
1  module test_wand();
2
3  wand a;
4  reg b, c;
5
6  assign a = b;
7  assign a = c;
8
9  initial begin
10   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
11   #1 b = 0;
12   #1 c = 0;
13   #1 b = 1;
14   #1 b = 0;
15   #1 c = 1;
16   #1 b = 1;
17   #1 b = 0;
18   #1 $finish;
19 end
20
21 endmodule
```

Simulator Output

```
0 a = x b = x c = x
1 a = 0 b = 0 c = x
2 a = 0 b = 0 c = 0
3 a = 0 b = 1 c = 0
4 a = 0 b = 0 c = 0
5 a = 0 b = 0 c = 1
6 a = 1 b = 1 c = 1
7 a = 0 b = 0 c = 1
```

Nets tri and trireg

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate).

Example - tri

```
1  module test_tri();
2
3  tri a;
4  reg b, c;
5
6  assign a = (b) ? c : 1'bz;
7
8  initial begin
9    $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
10   b  = 0;
11   c  = 0;
12   #1  b = 1;
13   #1  b = 0;
14   #1  c = 1;
15   #1  b = 1;
16   #1  b = 0;
17   #1  $finish;
18 end
19
20 endmodule
```

Simulator Output

```
0 a = z b = 0 c = 0
1 a = 0 b = 1 c = 0
2 a = z b = 0 c = 0
3 a = z b = 0 c = 1
4 a = 1 b = 1 c = 1
5 a = z b = 0 c = 1
```

Example - trireg

```
1  module test_trireg();
2
3  trireg a;
4  reg b, c;
5
6  assign a = (b) ? c : 1'bz;
7
8  initial begin
9    $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
10   b  = 0;
11   c  = 0;
12   #1  b = 1;
13   #1  b = 0;
14   #1  c = 1;
15   #1  b = 1;
16   #1  b = 0;
17   #1  $finish;
18 end
19
20 endmodule
```

Simulator Output

```
0 a = x b = 0 c = 0
1 a = 0 b = 1 c = 0
2 a = 0 b = 0 c = 0
3 a = 0 b = 0 c = 1
4 a = 1 b = 1 c = 1
5 a = 1 b = 0 c = 1
```

Types of Registers

Registers store the last value assigned to them until another assignment statement changes their value.
Registers represent data storage constructs.
You can create regs arrays called memories.
register data types are used as variables in procedural blocks.
A register data type is required if a signal is assigned a value within a procedural block.
Procedural blocks begin with keyword initial and always.

Data Types	Functionality
reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Note : Of all register types, reg is the one which is most widely used

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators.

When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column together with the escape sequence that represents the character in the left-hand column.

Special characters in strings

Character	Description
\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Example of strings:

```
module strings();
// Declare a register variable that is 21 bytes
reg [8*21:0] string ;

initial begin
  string = "This is sample string";
  $display ("%s \n", string);
end

endmodule
```

Behavioural Modeling
Μοντελοποίηση Συμπεριφοράς

Procedural Blocks – Block συναρτήσεις

Verilog behavioral code is inside procedure blocks, but there is an exception: some behavioral code also exist outside procedure blocks. We can see this in detail as we make progress.

There are two types of procedural blocks in Verilog:
initial : initial blocks execute only once at time zero (start execution at time zero).
always : always blocks loop to execute over and over again; in other words, as the name suggests, it executes always.

If a procedure block contains more than one statement, those statements must be enclosed within
Sequential **begin - end** block
Parallel **fork - join** block

When using begin-end, we can give name to that group. This is called named blocks.

Example of ***initial*** block:

```
1  module initial_example();
2  reg clk,reset,enable,data;
3
4  initial begin
5    clk = 0;
6    reset = 0;
7    enable = 0;
8    data = 0;
9  end
10
11 endmodule
```

Example of ***always*** block:

```
1  module always_example();
2  reg clk,reset,enable,q_in,data;
3
4  always @ (posedge clk)
5  if (reset)  begin
6     data <= 0;
7  end else if (enable) begin
8     data <= q_in;
9  end
10
11 endmodule
```

Procedural Blocks – Sequential and Parallel

If a procedure block contains more than one statement, those statements must be enclosed within
Sequential **begin - end** block
Parallel **fork - join** block

When using begin-end, we can give name to that group. This is called named blocks.

Example sequential:

```
1  module initial_begin_end();
2  reg clk,reset,enable,data;
3
4  initial begin
5   $monitor(
6     "%g clk=%b reset=%b enable=%b data=%b",
7     $time, clk, reset, enable, data);
8   #1   clk = 0;
9   #10  reset = 0;
10  #5   enable = 0;
11  #3   data = 0;
12  #1  $finish;
13 end
14
15 endmodule
```

Simulator Output

```
0 clk=x reset=x enable=x data=x
1 clk=0 reset=x enable=x data=x
11 clk=0 reset=0 enable=x data=x
16 clk=0 reset=0 enable=0 data=x
19 clk=0 reset=0 enable=0 data=0
```

Example parallel:

```
1  module initial_fork_join();
2  reg clk,reset,enable,data;
3
4  initial begin
5   $monitor("%g clk=%b reset=%b enable=%b data=%b",
6     $time, clk, reset, enable, data);
7   fork
8     #1   clk = 0;
9     #10  reset = 0;
10    #5   enable = 0;
11    #3   data = 0;
12  join
13  #1  $display ("%g Terminating simulation", $time);
14  $finish;
15 end
16
17 endmodule
```

Simulator Output

```
0 clk=x reset=x enable=x data=x
1 clk=0 reset=x enable=x data=x
3 clk=0 reset=x enable=x data=0
5 clk=0 reset=x enable=0 data=0
10 clk=0 reset=0 enable=0 data=0
11 Terminating simulation
```

Procedural assignments

Procedural assignment statements assign values to reg, integer, real, or time variables and can not assign values to nets (wire data types)

You can assign to a register (reg data type) the value of a net (wire), constant, another register, or a specific value.

Example ***wrong*** assignment:

```
1  module initial_bad();
2  reg clk,reset;
3  wire enable,data;
4
5  initial begin
6    clk = 0;
7    reset = 0;
8    enable = 0;
9    data = 0;
10 end
11
12 endmodule
```

Example of ***correct*** assignment:

```
1  module initial_good();
2  reg clk,reset,enable,data;
3
4  initial begin
5    clk = 0;
6    reset = 0;
7    enable = 0;
8    data = 0;
9  end
10
11 endmodule
```

```
module initial_wire_correct()
reg clk,reset;
wire enable,data;

initial begin
   clk = 0;
   reset = 0;
   enable <= 0;
   data <= 0;
end
endmodule
```

Wires can be assigned using nonblocking assignment <= instead of blocking assignment =

Blocking and nonblocking assignments

Blocking assignments are executed in the order they are coded, hence they are sequential. Since they block the execution of next statement, till the current statement is executed, they are called **blocking assignments.** Assignments are made with "=" symbol. Example a = b;

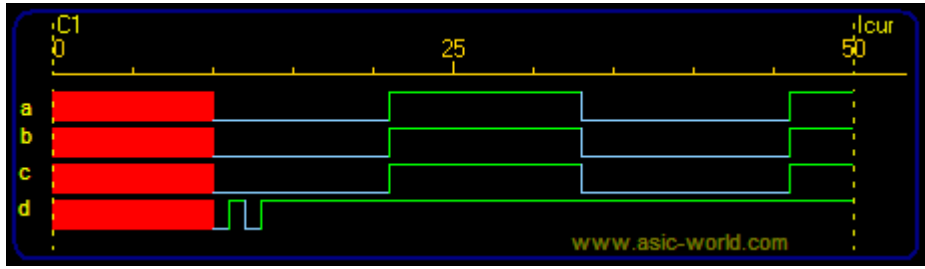
Nonblocking assignments are executed in parallel. Since the execution of next statement is not blocked due to execution of current statement, they are called **nonblocking statements**. Assignments are made with "<=" symbol. Example a <= b;

Wires can be assigned using nonblocking assignment <= instead of blocking assignment =

```
1  module blocking_nonblocking();
2
3  reg a,b,c,d;
4  // Blocking Assignment
5  initial begin
6    #10 a = 0;
7    #11 a = 1;
8    #12 a = 0;
9    #13 a = 1;
10 end
11
12 initial begin
13   #10 b <= 0;
14   #11 b <= 1;
15   #12 b <= 0;
16   #13 b <= 1;
17 end
18
19 initial begin
20   c = #10 0;
21   c = #11 1;
22   c = #12 0;
23   c = #13 1;
24 end
25
26 initial begin
27   d <= #10 0;
28   d <= #11 1;
29   d <= #12 0;
30   d <= #13 1;
31 end
32
33 initial begin
34   $monitor("TIME = %g A = %b B = %b C = %b D = %b",$time, a, b, c, d);
35   #50 $finish;
36 end
37
38 endmodule
```

Simulator Output

```
TIME = 0 A = x B = x C = x D = x
TIME = 10 A = 0 B = 0 C = 0 D = 0
TIME = 11 A = 0 B = 0 C = 0 D = 1
TIME = 12 A = 0 B = 0 C = 0 D = 0
TIME = 13 A = 0 B = 0 C = 0 D = 1
TIME = 21 A = 1 B = 1 C = 1 D = 1
TIME = 33 A = 0 B = 0 C = 0 D = 1
TIME = 46 A = 1 B = 1 C = 1 D = 1
```



Assign and deassign

The ***assign*** and ***deassign*** **procedural assignment statements** allow continuous assignments to be placed onto registers for controlled periods of time.
The *assign* statement overrides previous assignments to a register.
The *deassign* procedural statement ends a continuous assignment to a register.

```
1  module assign_deassign ();
2
3  reg clk,rst,d,preset;
4  wire q;
5
6  initial begin
7    $monitor("@%g clk %b rst %b preset %b d %b q %b",
8      $time, clk, rst, preset, d, q);
9    clk = 0;
10   rst = 0;
11   d  = 0;
12   preset = 0;
13   #10  rst = 1;
14   #10  rst = 0;
15   repeat (10) begin
16     @ (posedge clk);
17     d <= $random;
18     @ (negedge clk) ;
19     preset <= ~preset;
20   end
21   #1  $finish;
22 end
23 // Clock generator
24 always  #1  clk = ~clk;
25
```

```
26 // assign and deassign q of flip flop module
27 always @(preset)
28 if (preset) begin
29   assign U.q = 1; // assign procedural statement
30 end else begin
31  deassign U.q;     // deassign procedural statement
32 end
33
34 d_ff U (clk,rst,d,q);
35
36 endmodule
37
38 // D Flip-Flop model
39 module d_ff (clk,rst,d,q);
40 input clk,rst,d;
41 output q;
42 reg q;
43
44 always @ (posedge clk)
45 if (rst) begin
46   q <= 0;
47 end else begin
48   q <= d;
49 end
50
51 endmodule
```

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
@1  clk 1 rst 0 preset 0 d 0 q 0
@2  clk 0 rst 0 preset 0 d 0 q 0
@3  clk 1 rst 0 preset 0 d 0 q 0
@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 1
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```

Force and release

Another form of procedural continuous assignment is provided by the ***force*** and ***release*** procedural statements. These statements have a similar effect on the assign-deassign pair, but a force can be applied to nets as well as to registers.

They can be used during gate level simulation to work around reset connectivity problems.

Also can be used insert single and double bit errors on data read from memory.

They are not synthesizable.

```
1  module force_release ();
2
3  reg clk,rst,d,preset;
4  wire q;
5
6  initial begin
7    $monitor("@%g clk %b rst %b preset %b d %b q %b",
8      $time, clk, rst, preset, d, q);
9    clk = 0;
10   rst = 0;
11   d  = 0;
12   preset = 0;
13   #10  rst = 1;
14   #10  rst = 0;
15   repeat (10) begin
16     @ (posedge clk);
17     d <= $random;
18     @ (negedge clk) ;
19     preset <= ~preset;
20   end
21   #1  $finish;
22 end
23 // Clock generator
24 always  #1  clk = ~clk;
25
```

```
26 // force and release of flip flop module
27 always @(preset)
28 if (preset) begin
29   force U.q = preset; // force procedural statement
30 end else begin
31   release U.q;     // release procedural statement
32 end
33
34 d_ff U (clk,rst,d,q);
35
36 endmodule
37
38 // D Flip-Flop model
39 module d_ff (clk,rst,d,q);
40 input clk,rst,d;
41 output q;
42 wire q;
43 reg q_reg;
44
45 assign q = q_reg;
46
47 always @ (posedge clk)
48 if (rst) begin
49   q_reg <= 0;
50 end else begin
51   q_reg <= d;
52 end
53
54 endmodule
```

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
@1  clk 1 rst 0 preset 0 d 0 q 0
@2  clk 0 rst 0 preset 0 d 0 q 0
@3  clk 1 rst 0 preset 0 d 0 q 0
@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 0
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```