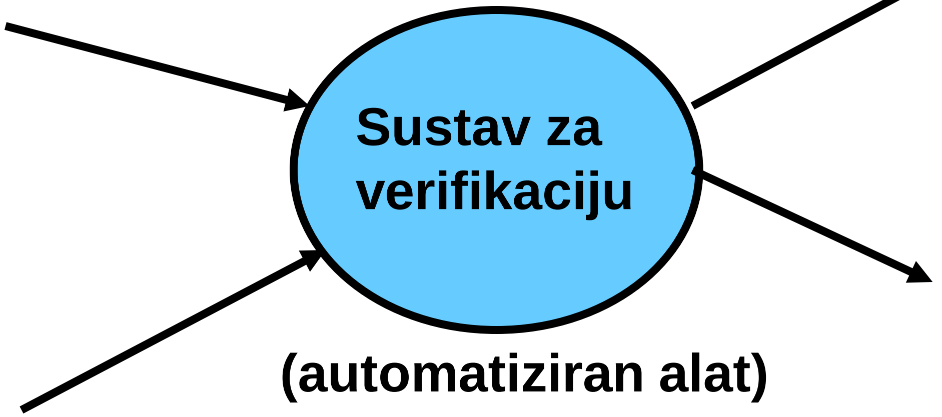
# Formalna verifikacija digitalnih sustava

## VIS i Verilog

## Sadržaj

## Formalna verifikacija provjerom modela

*I* = Implementacija
(model sustava koji želimo verificirati)



(automatiziran alat)

DA, implementacija **zadovoljava** specifikaciju

NE, implementacija **ne zadovoljava** specifikaciju (i ispis pogrešnog ponašanja)

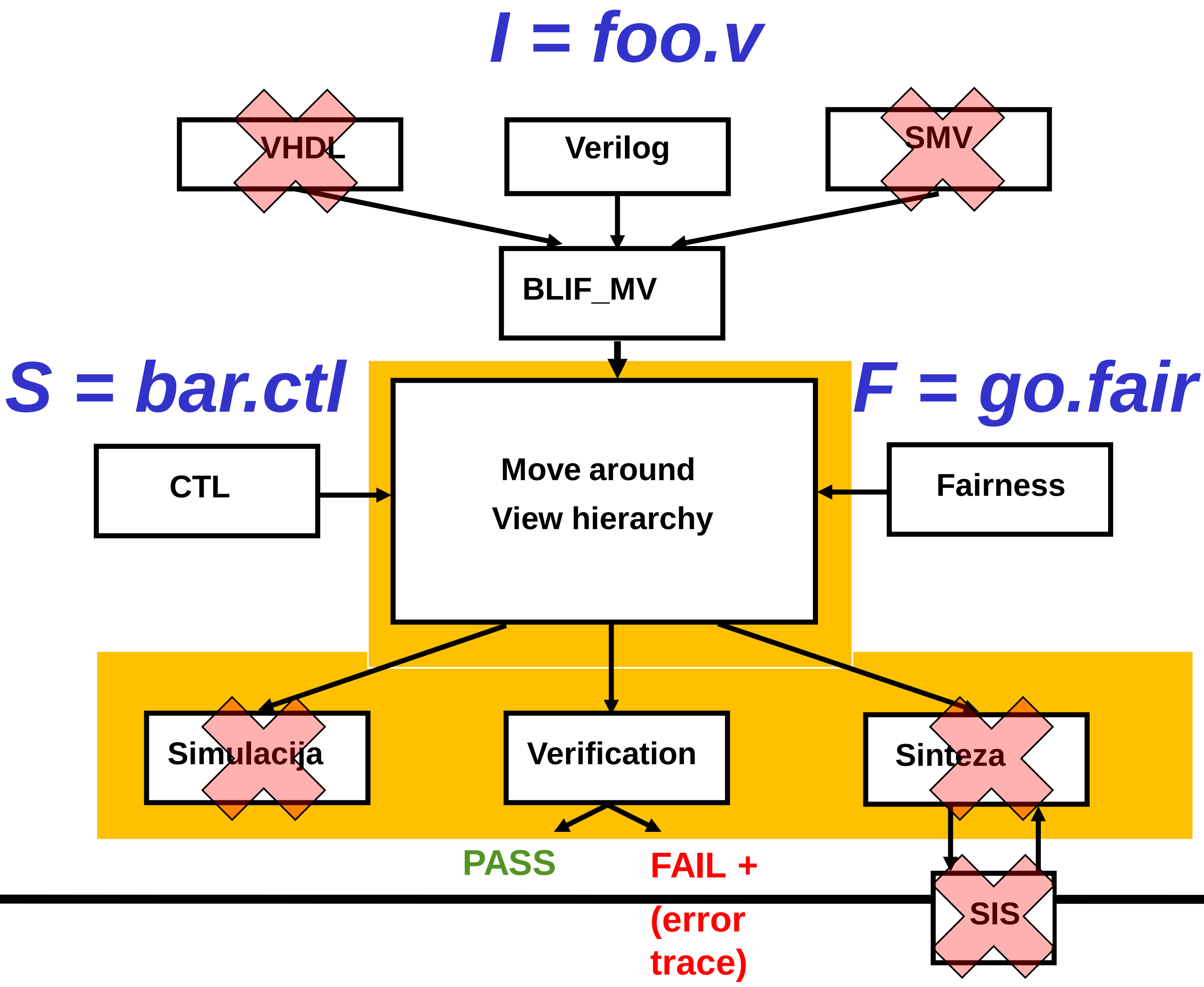
*S* = Specifikacija željenog ponašanja izražena u vremenskoj logici

zadovoljava =
logička zadovoljivost
(engl. satisfiability)

## I. Sustav za formalnu verifikaciju - VIS

- VIS (Verification Interacting with Synthesis) je sustav za formalnu verifikaciju, sintezu i simulaciju konačnih sustava.
- **Verification:** Are we building the product **right**?
- (Validation: Are we building the **right** product?)

## VIS (http://www-cad.eecs.berkeley.edu/~vis)



## VIS

- **Prihvaća opis implementacije u Verilogu (npr. datoteka `foo.v`).**
- **Prihvaća specifikaciju željenog obilježja u CTL vremenskoj logici (npr. datoteka `bar.ctl`).**
- **Prihvaća (kao opciju) ograničenje ponašanja u CTL vremenskoj logici (npr. datoteka `go.fair`).**
- **VIS operira na jednom posrednom formatu (BLIF_MV) opisa implementacije te se prije pokretanja VIS sustava datoteka `foo.v` mora prevesti u taj posredni format pozivom prijevodnika:**

```
vl2mv foo.v
```

**Generira se datoteka `foo.mv` koja se učita u sustav VIS te se uspostavlja interna struktura podataka koja omogućuje analizu, verifikaciju i sintezu ciljanog digitalnog sustava.**

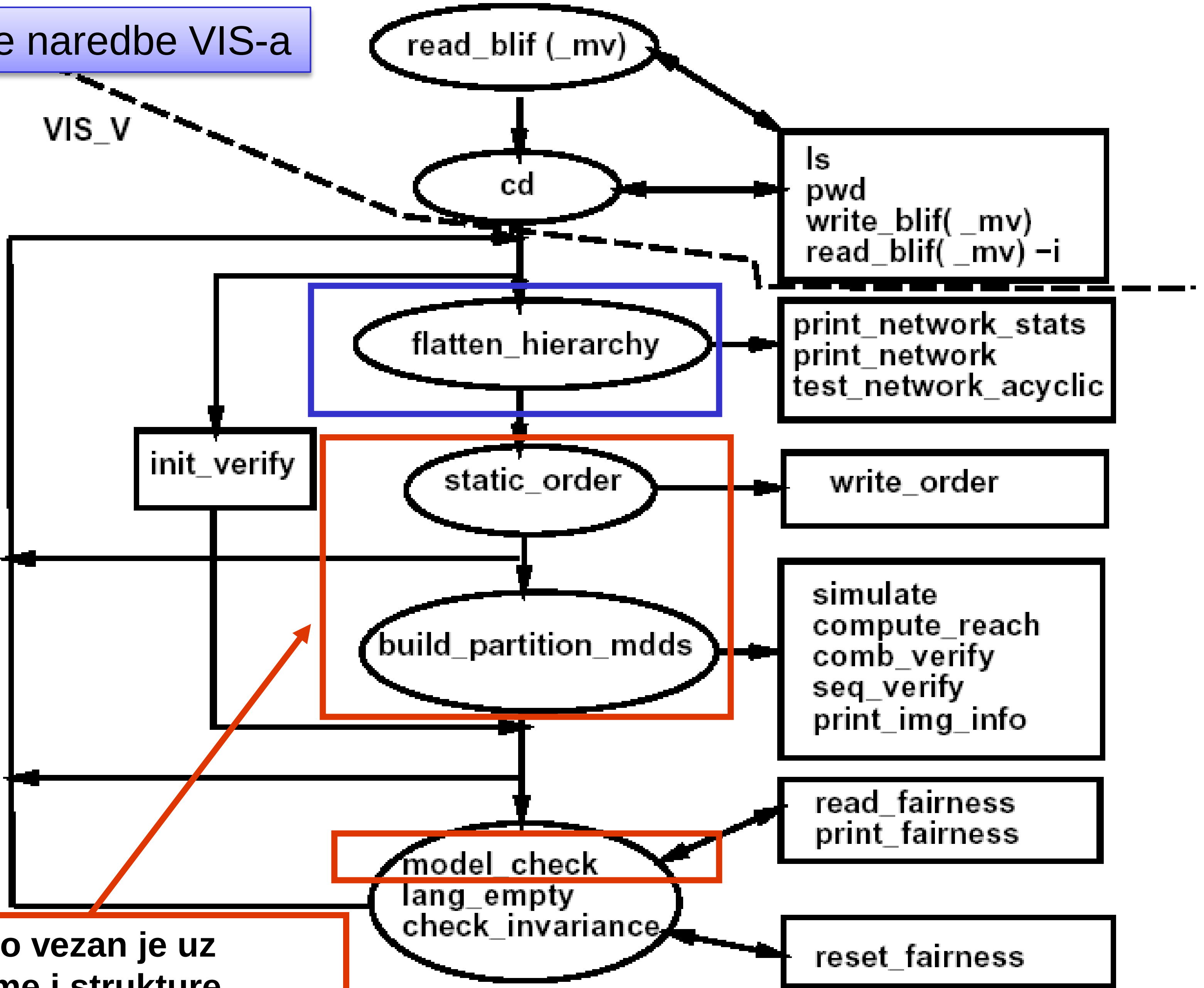
## VIS

`foo.mv` datoteka predstavljena je u sustavu VIS kao hijerarhija modula oblikovanog digitalnog sustava.

Što je moguće analizirati nakon upisa datoteke foo.mv u VIS:

- Prikaz hijerarhije modula. Po hijerarhiji se može "šetati" naredbama poput UNIX sustava (`pwd`, `cd` , …).
- Istražiti međudjelovanje modula preko zajedničkih varijabli.
- Prikazati strukture svih modula na izvedbenoj (niskoj) razini (registri, signali, …). Tako prikazana izravnana (engl. *flatten*) jedinstvena mrežna struktura sastavljena od temeljnih logičkih sklopova nije optimizirana. Optimizaciju izvedbe prikazanog digitalnog sustava izvode drugi alati.
- Izvesti simulaciju svakog modula determinističkim ili slučajnim vektorima ulaznih varijabli.
- Formalno verificirati digitalni sustav s obzirom na željeno obilježje.
- Tijekom verifikacije nametnuti ograničenja nepristranosti (engl. *fairness*).

## Važnije naredbe VIS-a



Ovaj dio vezan je uz algoritme i strukture podataka za model_check

## II. Verilog

- Jezik za opis strojevine (HDL)
- Zasnovan na C-u, ima slične jezične konstrukte
- Razvijen još 1984. godine, a normiran 1995. (Verilog-95)
- Promovirala ga je tvrtka Cadence
- Današnja norma je iz 2009.

  SystemVerilog/Verilog integrira jezik za opis i jezik za verifikaciju sklopovlja
- Tvrtke Cadence, Mentor i Synopsis, kao najveći proizvođači programske opreme za automatizaciju dizajna sklopova naveliko koriste Verilog prilikom simulacije novih elektroničkih komponenata

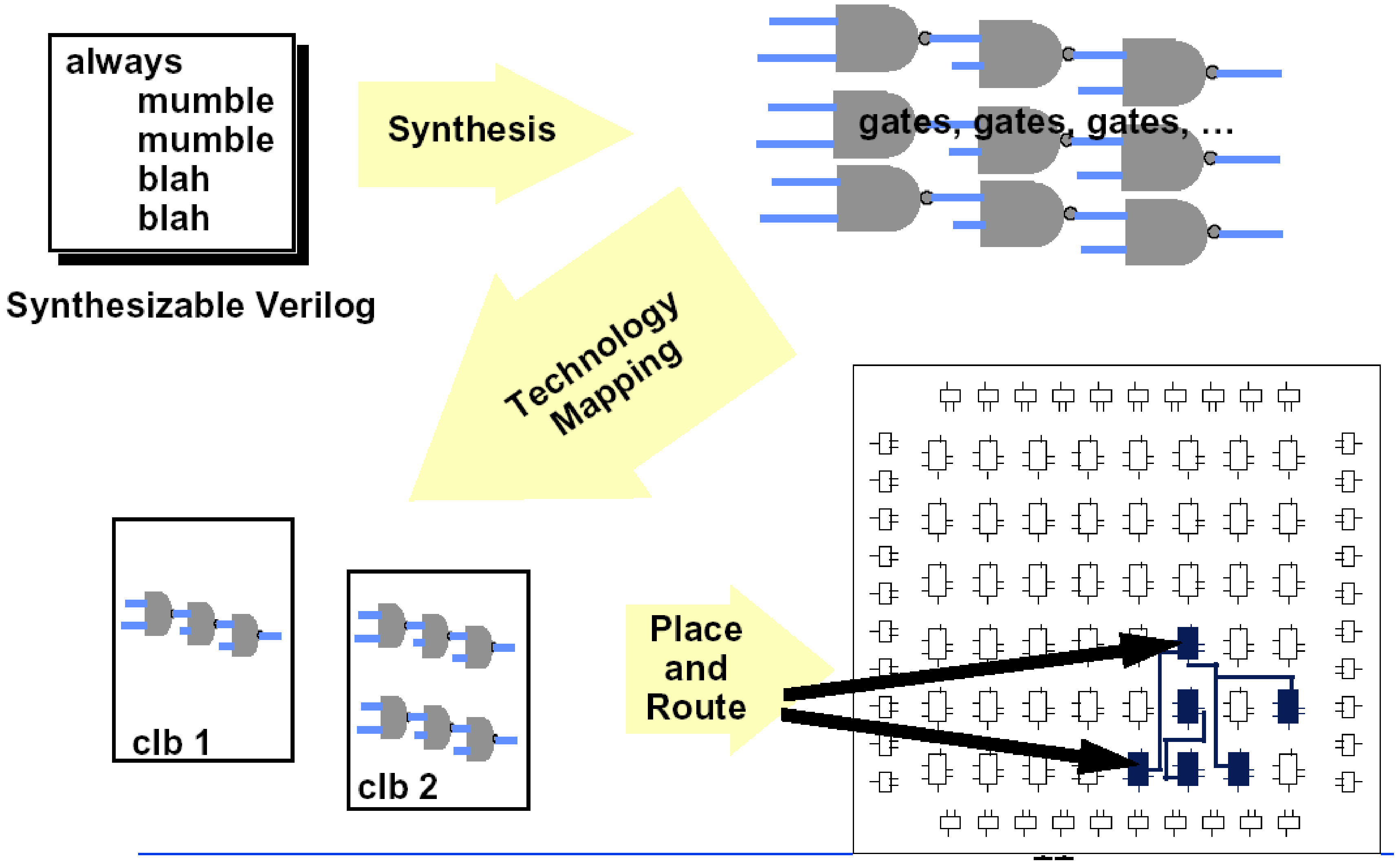
## Verilog

- Opis na više razina apstrakcije.
- Datoteke u Verilogu mogu se **verificirati, simulirati i sintetizirati**.

# Uvod u Verilog

## Modern Design Methodology

Simulation and Synthesis are components of a design methodology



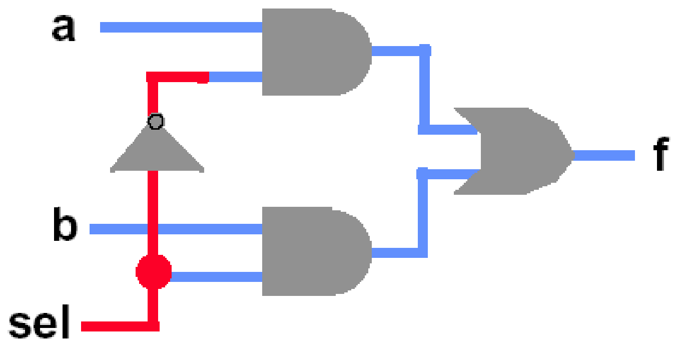
## Simulation of Digital Systems

- Simulation checks two properties
  - functional correctness —is the logic correct
    - correct design, and design correct
  - timing correctness —is the logic/interconnect timing correct
    - e.g. are the set-up times met?

- It has all the limitations of software testing
  - Have I tried all the cases?
  - Have I exercised every path? Every option?

## Representation: Structural Models

- **Structural models**
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates —much as you would see in an implementation of a circuit.
    - You could describe your lab1 circuit this way
- **Identify:**
  - Gate instances, wire names, delay from *a* or *b* to *f*.



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

        and #5   g1 (f1, a, nsel),
                 g2 (f2, b, sel);
        or   #5  g3 (f, f1, f2);
        not      g4 (nsel, sel);
endmodule
```

## Representation: Gate-Level Models

- Need to model the gate's:
  - Function
  - Delay
- Function
  - Generally, HDLs have built-in gate-level primitives
    - Verilog has NAND, NOR, AND, OR, XOR, XNOR, BUF, NOT, and some others
  - The gates operate on input values producing an output value
    - typical Verilog gate instantiation is:

optional "many"

```
and #delay  instance-name (out, in1, in2, in3, …);
```

## Four-Valued Logic

- **Verilog Logic Values**
  - The underlying data representation allows for any bit to have one of four values
  - 1, 0, x (unknown), z (high impedance)
  - x —one of: 1, 0, z, or in the state of change
  - z —the high impedance output of a tri-state gate.
- **What basis do these have in reality?**
  - 0, 1 … no question
  - z … A *tri-state* gate drives either a zero or one on its output. If it's not doing that, its output is high impedance (z). Tri-state gates are real devices and z is a real electrical affect.
  - x … not a real value. There is no *real* gate that drives an x on to a wire. x is used as a debugging aid. x means the simulator can't determine the answer and so maybe you should worry!

## Four-Valued Logic

- **Logic with multi-level logic values**
  - Logic with these four values make sense
    - Nand anything with a 0, and you get a 1. This includes having an x or z on the other input. That's the nature of the nand gate
    - Nand two x's and you get an x
  - Note: z treated as an x on input. Their rows and columns are the same
  - If you forget to connect an input ..it will be seen as an z.
  - At the start of simulation, *everything* is an x.

Input B

Input A

| Nand | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| x | 1 | x | x | x |
| z | 1 | x | x | x |

A 4-valued truth table for a Nand gate with two inputs

## How to build and test a module

- Construct a “test bench” for your design
  - Develop your hierarchical system within a module that has input and output ports (called “design” here)
  - Develop a separate module to generate tests for the module (“test”)
  - Connect these together within another module (“testbench”)

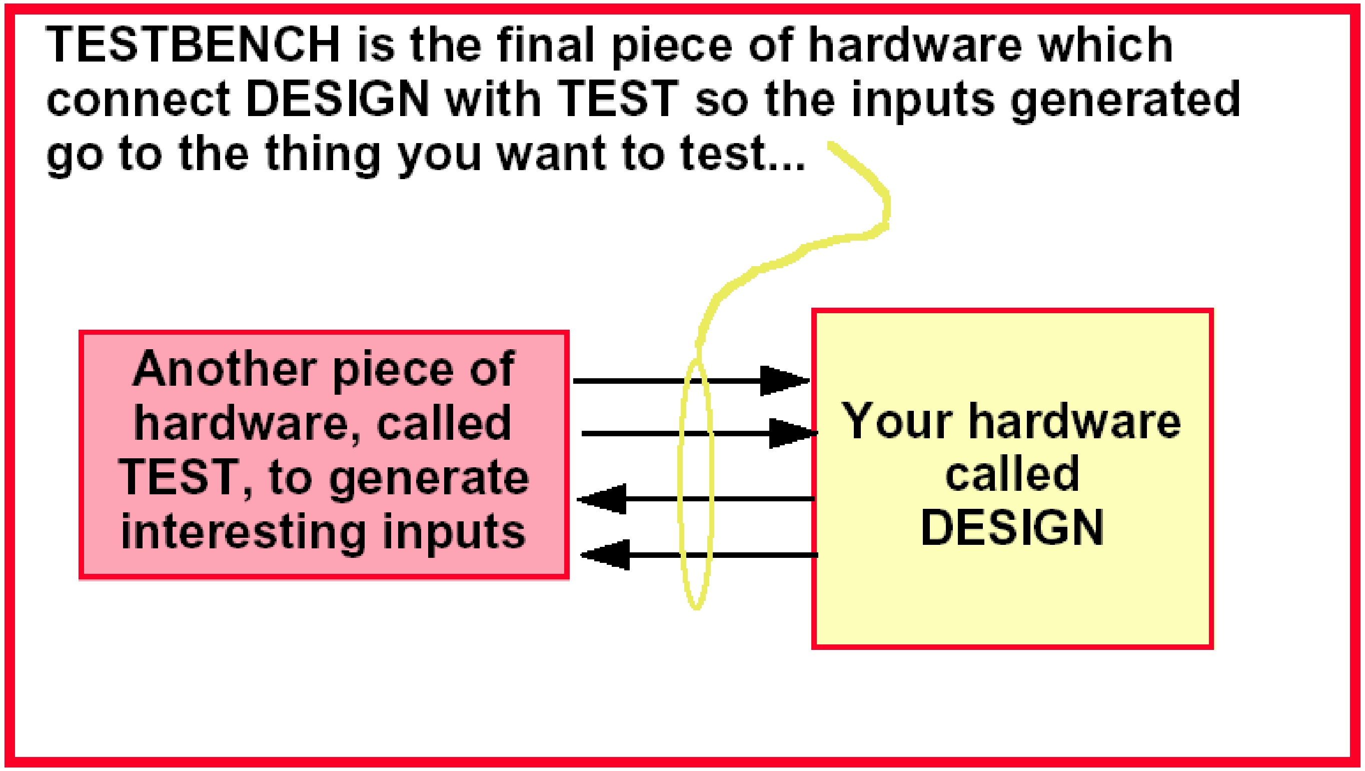
```
module design (a, b, c);
      input     a, b;
      output   c;
      …
```

```
module testbench ();
      wire        l, m, n;

      design   d (l, m, n);
      test        t (l, m);

      initial begin
            //monitor and display
            …
```

```
module test (q, r);
      output   q, r;

      initial begin
            //drive the outputs with signals
            …
```

## Another view of this

- 3 chunks of verilog, one for each of:



TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...

Another piece of hardware, called TEST, to generate interesting inputs
Your hardware called DESIGN

## Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the *design*

```
module tBench;
    wire      su, co, a, b;

    halfAdd       ad(su, co, a, b);
    testAdd       tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
    output    sum, cOut;
    input     a, b;

    xor #2    (sum, a, b);
    and #2    (cOut, a, b);
endmodule
```

```
module testAdd(a, b, sum, cOut);
    input    sum, cOut;
    output   a, b;
    reg      a, b;

    initial begin
        $monitor ($time,,
          "  a=%b, b=%b, sum=%b, cOut=%b",
           a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

## The test module

- It's the test generator
- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current simulation time
- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

```
module testAdd(a, b, sum, cOut);
    input    sum, cOut;
    output   a, b;
    reg      a, b;

    initial begin
        $monitor ($time,,
          "  a=%b, b=%b, sum=%b, cOut=%b",
          a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

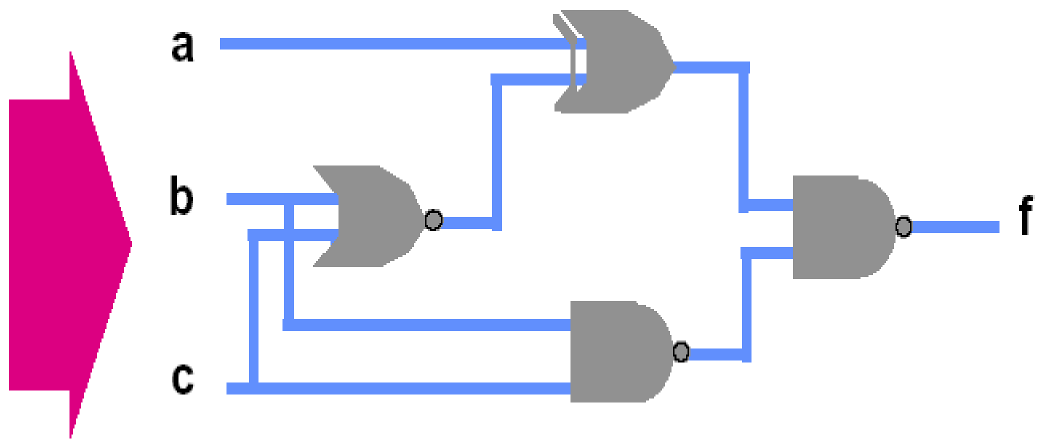
## What’s cool?

- You type the left, synthesis gives you the gates
- It used a different library than you did. (2-input gates only)
- One description suffices for a variety of alternate implementations!

## Hmmm …

- … but this assumes you know a gate level implementation —that’s not an “ abstract”Verilog description.

```
module gate (f, a, b, c);
    output    f;
    input     a, b, c;

    and    A (a1, a, b, c),
           B (a2, a, ~b, ~c),
           C (a3, ~a, o1);
    or     D (o1, b, c),
           E (f, a1, a2, a3);
endmodule
```

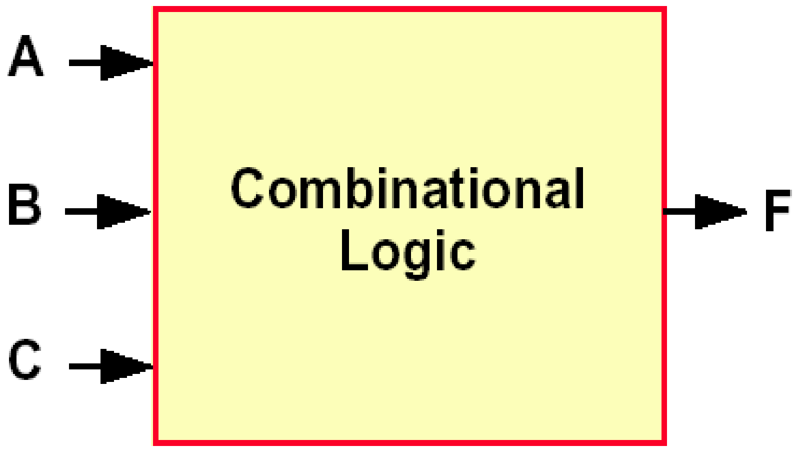


## Goal

- To specify a combination ckt, inputs->outputs…
- ..in a form of Verilog that synthesis tools will correctly read
- ..and then use to make the right logic

## And…

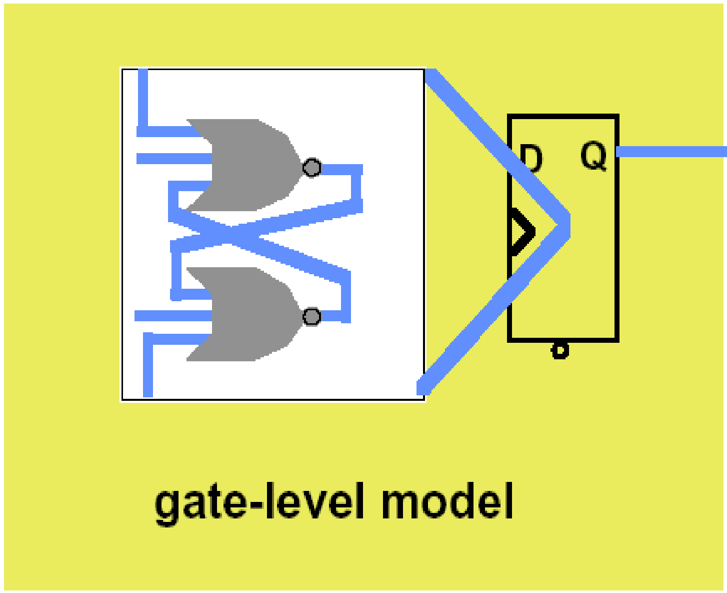
- We know the function we want, and can specify in C-like form…
- ..but we don't now the exact gates;  we want the tool to do this.

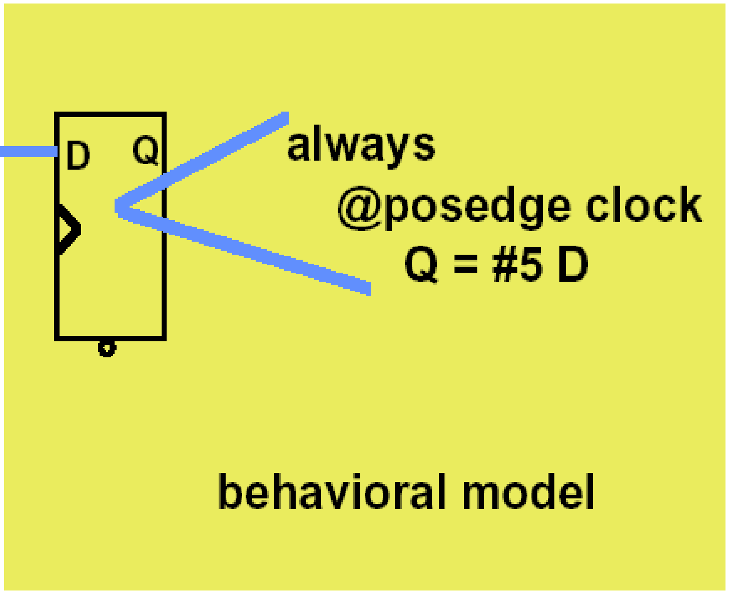


# Verilog Levels of Abstraction

- **Gate modeling**
  - the system is represented in terms of primitive gates and their interconections
    - NANDs, NORs, …
- **Behavioral modeling**
  - the system is represented by a program-like language

Modeliranje ponašanja (ponašajni model)





## Structural vs Behavioral Models

(hrv. ponašajni model)

- **Structural model**
  - Just specifies primitive gates and wires
  - i.e., the structure of a logical netlist
  - You basically know how to do this now.
- **Behavioral model**
  - More like a procedure in a programming language
  - Still specify a module in Verilog with inputs and outputs...
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  - i.e., you specify the behavior you want, not the structure to do it
- **Why use behavioral models**
  - For testbench modules to test structural designs
  - For high-level specs to drive logic synthesis tools (Lab 2)

## Napomena

- U ovom je predmetu žarište na ponašajnim modelima, jer oni u pravom smislu pokazuju oblikovanje kroz aktivnost modeliranja po principu “odozgo-prema-dolje” (od više razine apstrakcije prema nižoj).

## Behavioral Modeling

- *Procedural* statements are used
  - Statements using "always" Verilog construct
  - Can specify both combinational and sequential circuits
- Normally don't think of procedural stuff as "logic"
  - They look like C: mix of ifs, case statements, assignments …
  - ..but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)
- Current technology
  - You can do combinational (and later, sequential) design
  - Sizable designs can take hours ..days ..to run
  - Companies pay $50K - 80K per copy for such software
    - This ain't shrink-wrap software!
  - The software we'll use is more like $10-15K

## Behavioral Constructs

- Behavioral descriptions are introduced by initial and always statements

| Statement | Looks like | Starts | How it works | Use in Synthesis? |
|---|---|---|---|---|
| initial | initial<br>begin<br>…<br>end | Starts when simulation starts | Execute once and stop | Not used in synthesis |
| always | always<br>begin<br>…<br>end | | Continually loop—while (power on) do statements; | Used in synthesis |

- Points:
  - They all execute concurrently
  - They contain behavioral statements like if-then-else, case, loops, functions, …

## Statements, Registers and Wires

- **Registers**
  - Define storage, can be more than one bit
  - Can only be changed by assigning value to them on the left-hand side of a behavioral expression.
- **Wires (actually " nets")**
  - Electrically connect things together
  - Can be used on the right-hand side of an expression
    - Thus we can tie primitive gates and behavioral blocks together!
- **Statements**
  - left-hand side = right-hand side
  - left-hand side must be a register

Multi-bit registers and wires

```
module silly (q, r);
   reg    [3:0]  a, b;
   wire   [3:0]  q, r;

   always begin
          ...
          a =  (b & r) | q;
          ...
          q = b;
          ...
   end
endmodule
```

Logic with registers and wires

Can't do —why?

## Behavioral Statements

- **if-then-else**
  - What you would expect, except that it's doing 4-valued logic. 1 is interpreted as True; 0, x, and z are interpreted as False

```
if (select == 1)
        f = in1;
else    f = in0;
```

- **case**
  - What you would expect, except that it's doing 4-valued logic
  - If " selector" is 2 bits, there are 4 possible case-items to select between
  - There is no *break* statement —it is assumed.

```
case (selector)
    2'b00: a = b + c;
    2'b01: q = r + s;
    2'bx1: r = 5;
    default: r = 0;
endcase
```

- **Funny constants?**
  - Verilog allows for sized, 4-valued constants
  - The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.

8'b00x0zx10

assume f, a, q, and r are registers for this slide

# Behavioral Statements

- Loops
  - There are restrictions on using these for synthesis — don't.
  - They are mentioned here for use in test modules
- Two main ones — for and while
  - Just like in C
  - There is also repeat and forever — see the book

```
reg   [3:0]   testOutput, i;
…
for (i = 0; i <= 15; i = i + 1) begin
        testOutput = i;
        #20;
end
```

```
reg   [3:0]   testOutput, i;
…
i = 0;
while (i <= 15)) begin
        testOutput = i;
        #20 i = i + 1;
end
```

!!!!! **Important: Loops must have a delay operator (or as we'll see later, an @ or wait(FALSE)). Otherwise, the simulator never stops executing them.**

## Concurrent Constructs

- We already saw #delay
- Others
  - @ ..Waiting for a *change* in a value —used in synthesis
    - @ (var) w = 4;
    - This says wait for var to change from its current value. When it does, resume execution of the statement by setting w = 4.
  - Wait ..Waiting for a value to be a certain level —not used in synthesis
    - wait (f == 0) q = 3;
    - This says that if f is equal to zero, then continue executing and set q = 3.
    - But if f is not equal to zero, then suspend execution until it does. When it does, this statement resumes by setting q = 3.
- Why are these concurrent?
  - Because the event being waited for can only occur as a result of the concurrent execution of some other always/initial block or gate.
  - They're happening concurrently

## FAQs: behavioral model execution

- **How does an always or initial statement start**
  - That just happens at the start of simulation —arbitrary order
- **Once executing, what stops it?**
  - Executing either a #delay, @event, or wait(FALSE).
  - All always blocks need to have at least one of these. Otherwise, the simulator will never stop running the model -- (it's an infinite loop!)
- **How long will it stay stopped?**
  - Until the condition that stopped it has been resolved
    - #delay ..until the delay time has been reached
    - @(var) ..until var changes
    - wait(var) ..until var becomes TRUE
- **Does time pass when a behavioral model is executing?**
  - No. The statements (if, case, etc) execute in zero time.
  - Time passes when the model stops for #, @, or wait.
- **Will an always stop looping?**
  - No. But an initial will only execute once.

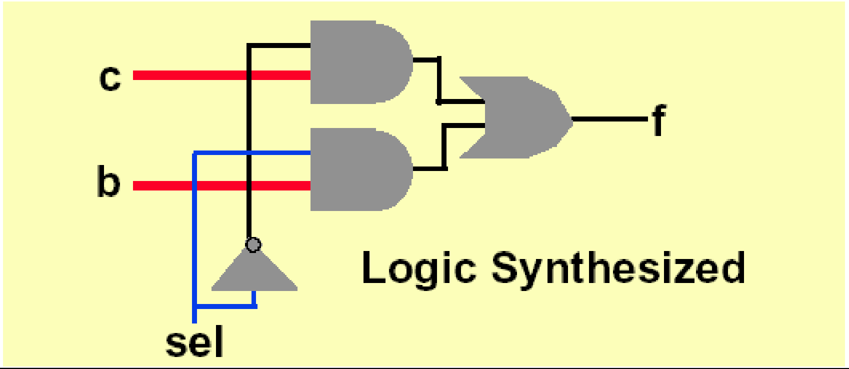
## A Combinational Circuit

- **Using behavioral constructs**
  - Logic for a simple MUX is specified procedurally here
  - This example is synthesizable

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

    always @ (sel or b or c)
              if (sel == 1)
                   f = b;
              else
                   f = c;
endmodule
```

Read this as follows:
Wait for any change on a, b, or c, then execute the begin-end block containing the if. Then wait for another change.

This " if"functionally describes the MUX



Logic Synthesized

## Typical Style

- Your Verilog for combination stuff will look like this:

```
module blah (<output names>, <input names>);
     output    <output names>;
     input     <input names>;
     reg       <output names>;

     always @ (<names of all input vars>)
          begin
               < LHS = RHS assignments>
               < if ... else  statements>
               < case statements >
          end
endmodule
```

- Yes...it's a pretty restricted subset of the langauge...

## What is the behavioral model sensitive to?

- The behavioral statements execute in sequence (one then the next)
- Therefore, what a behavioral model is sensitive to is context specific
  - i.e. it is only sensitive to what it is currently waiting for
  - time, edge, level —(#, @, wait)
- The model is not sensitive to a change on *y,* or *w.*

```
    always begin
        @ (negedge clock1)
            q = y;
        @ (negedge clock2)
            q = w;
        @ (posedge clock1)
            /*nothing*/  ;
        @ (posedge clock2)
            q = 3;
    end
```

Here, it is only sensitive to clock1

Here, it is only sensitive to clock2. A posedge on clock1 will have no effect when waiting here.

It is never sensitive to changes on y or w

# Behavioral Timing Model

- How does the behavioral model advance time?
  - # —delaying a specific amount of time
  - @ —delaying until an event occurs —e.g. @v
    - " posedge", " negedge", or any change
    - this is edge-sensitive behavior
    - When the statement is encountered, the value v is sampled. When v changes in the specified way, execution continues.
  - wait —delaying until an event occurs (" wait (f == 0)")
    - this is level sensitive behavior
  - While one model is waiting for one of the above reasons, other models execute —time marches on

## Wait vs. While

- **Are these equivalent?**
  - No: The left example is correct, the right one isn't —it won't work
  - *Wait* is used to wait for an expression to become TRUE
    - the expression eventually becomes TRUE because a variable in the expression is changed by another process
  - *While* is used in the normal programming sense
    - in the case shown, if the expression is TRUE, the simulator will continuously execute the loop. Another process will never have the chance to change " in". Infinite loop!
    - while can't be used to wait for a change on an input to the process. Need other variable in loop, or # or @ in loop.

```
module yes (in, .);
input     in;
...
          wait (in == 1);
          ...
endmodule
```

```
module no (in, .);
input     in;
...
          while (in != 1);
          ...
endmodule
```

## Blocking procedural assignments and #

- We've seen **blocking assignments** —they use =
  - Options for specifying delay

addition

```
#10 a = b + c;
a = #10 b + c;
```

The difference?

| | |
|---|---|
| + | addition |
| \|\| | logical OR |

  - The differences:

Second: temp store before assign

Note the action of the second one:

- an *intra-assignment* time delay
- execution of the always statement is blocked (suspended) in the middle of the assignment for 10 time units.
- how is this done?

## Non-blocking assignments (<=)

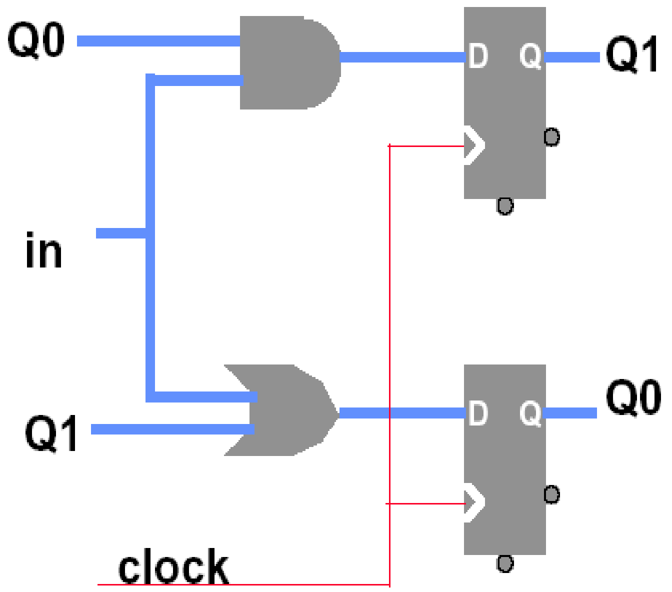
- Two important aspects to these
  - an intra-assignment time delay doesn't stop them (they're non-blocking)
  - they implement a concurrent assignment
- Example — intra-assignment time delay
  - non-blocking assignments use "<="

    a <= #10 b + c;
- What happens?
  - b + c is calculated
  - an update event for a is scheduled #10 in future
  - execution of the always *continues* in the *current* time
    - the execution of the always is not blocked by the delay
  - there is also a subtle difference in how *a* is updated …
    - we'll get to it, but first, an example

# Non-Blocking Concurrent Assignment

- **Concurrent Assignment —primary use of <=**
  - The assignment is " guarded" by an edge
  - All assignments guarded by the edge happen concurrently
    - All right-hand sides are evaluated before any left-hand sides are updated
    - Like this

```
module fsm (Q1, Q0, in, clock);
     output    Q1, Q0;
     input     clock, in;
     reg       Q1, Q0;

     always @(posedge clock) begin
          Q1 <= in & Q0;
          Q0 <= in | Q1;
     end
endmodule
```



## Non-Blocking Concurrent transfers

- Across the whole design,

  all right-hand sides are evaluated

  before any left-hand sides are updated.

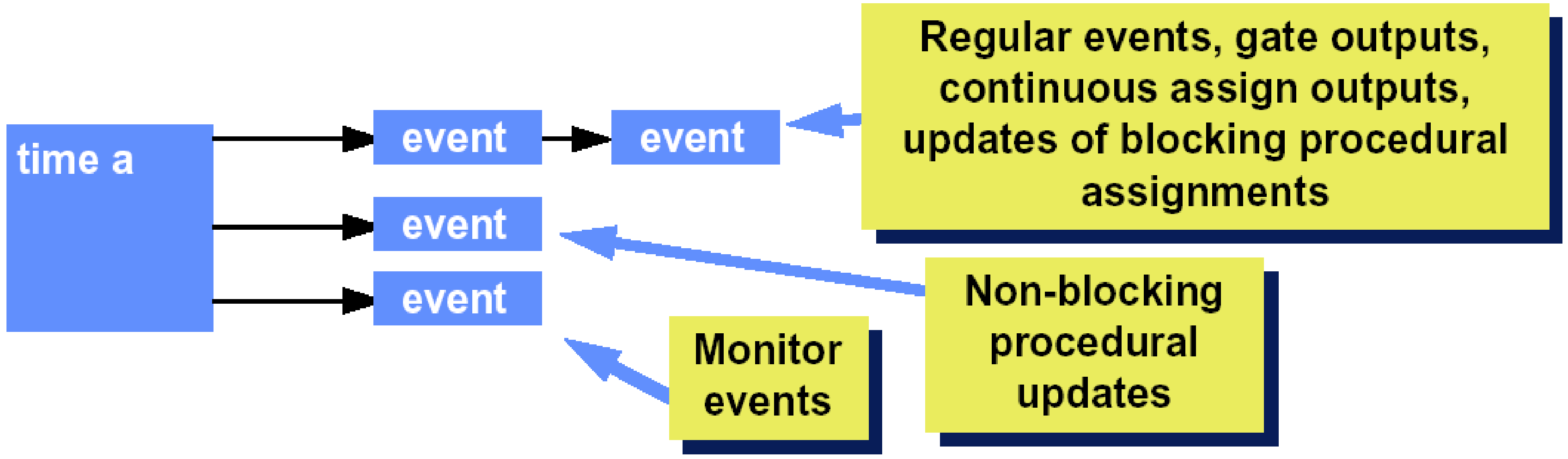
- Thus, the order of r-hs's evaluated and l-hs's updated can be arbitrary (but separate)

## This allows us to ...

- handle concurrent specification in major systems
- reduce the complexity of our descriptions
- attach lots of actions to one event —the clock

## What gets scheduled when/where

- Now
  - While there are *regular* events:
    - "  retrieve all regular events for current time and execute in arb. order"
    - Note: These may produce more regular events for current time
  - Retrieve all non-blocking events for the current time and execute
    - these may produce more regular events for current time, if so
  - When no more events, do monitor events. No new events produced


time a
event
event
event
event
Regular events, gate outputs, continuous assign outputs, updates of blocking procedural assignments
Non-blocking procedural updates
Monitor events

## Names of things

- **Thus far, we've seen names of…**
  - registers, variables, inputs, outputs, instances, integers
  - Their scope is the begin-end block within which they were defined
    - module —endmodule
    - task —endtask
    - function —endfunction
    - begin:name —end
  - ..nothing else within that scope may already have that name
- **Types of references**
  - Forward referenced —Identifiers for modules, tasks, functions, and named begin-end blocks may be used before being defined
  - Not Forward referenced —must be defined before use
    - wires and registers
  - Hierarchical references —named through the instantiation hierarchy
    - " a.b" references identifier *b* in namespace *a*
    - forward referenced (anything can be accessed, bad style)

## A few examples

```
module a (.);
    reg e;
    task b;
        reg c;
        begin : d
            reg e;
            e = 1;
            a.e = 0;
        end
    endtask
    always
        begin : f
            reg g;
            a.b.d.e = 2;
            g = q.a.b.d.e;
            e = 3;
        end
endmodule
```

named begin-end block

e's hierarchical name is .a.b.d.e

This e is different (it is top e)

g's hierarchical name is .a.f.g

assumes a is instantiated in q

same as a.e

since no local e

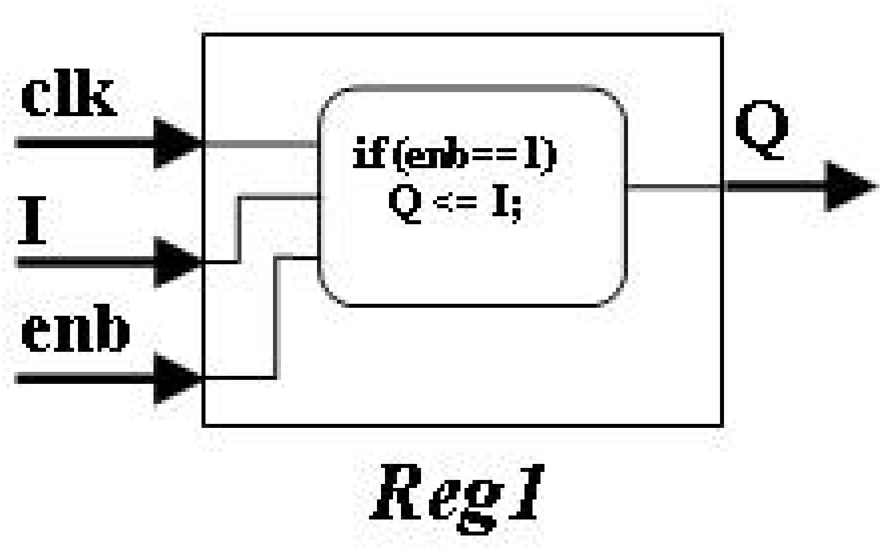
## Verilog primjer

Tip dugog i jednolinijskog komentara

```
/* 4 to 1 MUX (16 data in-out) */

module mux_4to1(Y, A, B, C, D, sel);

output [15:0] Y;
input [15:0] A, B, C, D;
input [1:0] sel;
reg [15:0] Y;              // izlaz tipa reg

always @(A or B or C or D or sel)
      case ( sel )              // ovisno o 2 bita sel
            2'b00: Y = A;
            2'b01: Y = B;
            2'b10: Y = C;
            2'b11: Y = D;
            default: Y = 16'hxxxx;  //inače hex nepoznato
      endcase
endmodule
```

## Verilog primjer

```
// 1-bit Register



module Reg1(clk, I, enb, Q);

input clk;
input I, enb;
output Q;
reg Q;


always @ (posedge clk)
   begin
      if( enb == 1 )
         Q <= I;
   end

endmodule
```



nonblocking assign

(isti vremenski trenutak za cijeli oblikovani sustav)

## Verilog primjer

```
/* 32 bit register with an asynchronous reset (active
low) */

module Reg32(Q, D, clk, reset_);

output [31:0] Q;
input [31:0] D;
input clk, reset_;
reg [31:0] Q;                   // izlaz tipa reg

always @(posedge clk or negedge reset)
      if (!reset_)              // ako reset (neg) stavi 0
            Q <= 32'b0;         // non blocking assign
      else
            Q <= D;       // inače stavi što je na ulazu
endmodule
```

**! logical NOT**

## Verilog primjer

```
// 2x4 Decoder
// Truth table for 2x4 Decoder
//  A   B  |  D3  D2  D1  D0
//---------+-----------------
//  0   0  |   0   0   0   1
//  0   1  |   0   0   1   0
//  1   0  |   0   1   0   0
//  1   1  |   1   0   0   0


module Decoder(A, B, D);
      input A, B;
      output [3:0] D;
      reg [3:0] D;


// nastavak na slijedećoj slici
```

## Verilog primjer

```
// nastavak 2x4 Dekoder

always @ (A or B)
begin
      if( A == 0 && B == 0 )
            D <= 4'b0001;      // non blocking assign
      else if ( A == 0 && B == 1 )
            D <= 4'b0010;
      else if ( A == 1 && B == 0 )
            D <= 4'b0100;
      else
            D <= 4'b1000;
end
endmodule
```

## Verilog primjer

```
// 4-bit Adder

module Adder(A, B, Result);
  input [3:0] A;
  input [3:0] B;
  output [3:0] Result;
  reg [3:0] Result;

  always @ (A or B)
      begin
            Result <= A + B;
      end
endmodule
```

*****************************************

**Napomena:**

- **“Register” operandi su “unsigned”**
- **Ako je jedan operand nepoznat (“x”), rezultat je nepoznat (“x”)**
- **“Carry” se ignorira u ovom primjeru**

## Verilog primjer

```
// Adder, uporaba “parameter” i “assign” naredbe

module adder (sum, a, b);
  parameter WIDTH = 7;
            // ovdje 8 bita, može se mijenjati
  input [WIDTH:0] a, b;
  output [WIDTH:0] sum;

  assign sum = a + b;
endmodule
************************************************
```

**Napomena:**

- **Naredbom “`assign`” će se promijeniti “`sum`” ako se promijene “`a`” ili “`b`” BILO KADA (a ne kao što to u drugim prikazanim slučajevima određuje “blocking” i “nonblocking” pridruživanje) . Ovdje korištena naredba “assign” prikazuje doslovnu karakteristiku kombinacijske logike (kao da su elementi čvrsto i stalno povezani žicama).**

## Verilog primjer

```
// jednobitno zbrajalo, 3 bita na ulazu

module oneBitFullAdder(cOut, sum, aIn, bIn, cIn);
      output cOut, sum;
      input aIn, bIn, cIn;

assign sum = aIn ^ bIn ^ cIn,
       cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);
endmodule
// operator  “^“ je “bitwise XOR”
// operator “|” je “bitwise” OR
// operator “&” je bitwise AND
// operator “~” je bitwise komplement
```

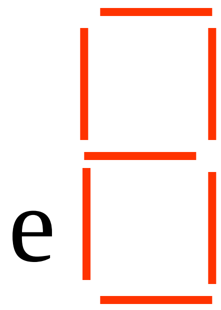
*****************************************************

**Napomena:**
**Naredba “`assign`” je niže razine apstrakcije nego “blocking” i “nonblocking” pridruživanje. Bliže je opisu izvedbe digitalnih sustava logičkim sklopovima.**

## Verilog primjer

```
/* 4-bit Multiplier, consists of 2 4-bit inputs A and B,
and an 8-bit output */


module Multiplier(A, B, Result);
      input [3:0] A;
      input [3:0] B;
      output [7:0] Result;
      reg [7:0] Result;

      always @ (A or B)
            begin
                  Result <= A * B;
            end
endmodule
```

# Verilog primjer



Bin inputs: A B C D

A=$2^3$, B=$2^2$, C=$2^1$, D=$2^0$

Output: e LED only

for HEX character

Ex. 1 1 0 1

= $13_{DEC}$ = HEX char. “d”

e LED = 1 (ON)

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 1 | 1 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

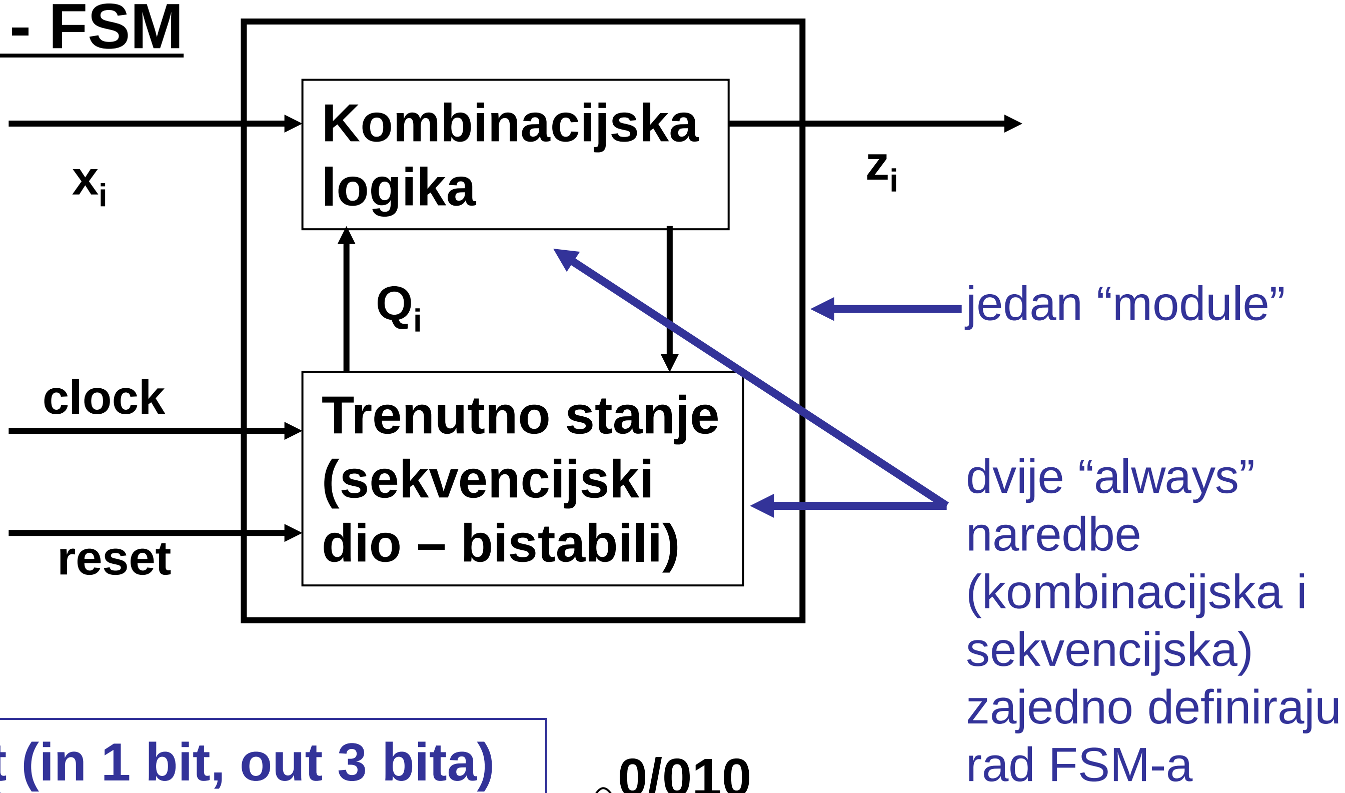
```
module binaryToESeg(eSeg, A, B, C, D);
    output eSeg;
    input  A, B, C, D;
    reg    eSeg;
        always @(A or B or C or D) begin
            eSeg = 1;     // initial set = on
            if(~A & D)    // if ~A&D then off
                eSeg = 0;
            if(~A & B & ~C)
                eSeg = 0;
            if(~B & ~C & D)
                eSeg = 0;
    end
endmodule
```

# Verilog primjer - FSM

$Q_{NEXT} = f(x, Q)$

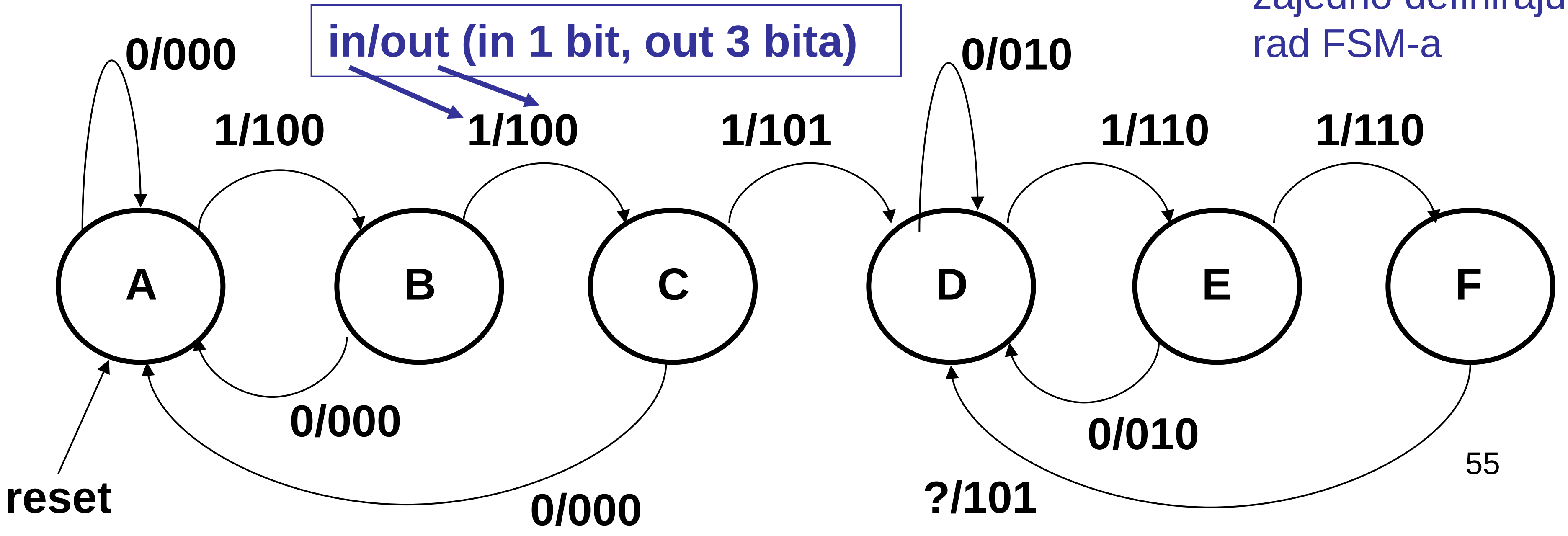
Mealy: $z = g(Q, x)$

Moore $z = h(Q)$



jedan “module”

dvije “always” naredbe (kombinacijska i sekvencijska) zajedno definiraju rad FSM-a



## Verilog primjer - FSM

```
module fsm (i, clock, reset, out);
      input i, clock, reset;
      output [2:0] out;        // output 3 bita
      reg [2:0] out;
      reg [2:0] currentState, nextState; //3 bita OK
      parameter [2:0]   A = 0, // state assignments
                        B = 1, // for current/next
                        C = 2,
                        D = 3,
                        E = 4,
                        F = 5;

//1st always: comb logic, out and nextState functions
always @(i or currentState)
      case (currentState)
            A: begin
                  nextState = (i == 0) ? A : B;
                  out = (i == 0) ? 3'b000 : 3'b100;
               end
```

ako se promijeni

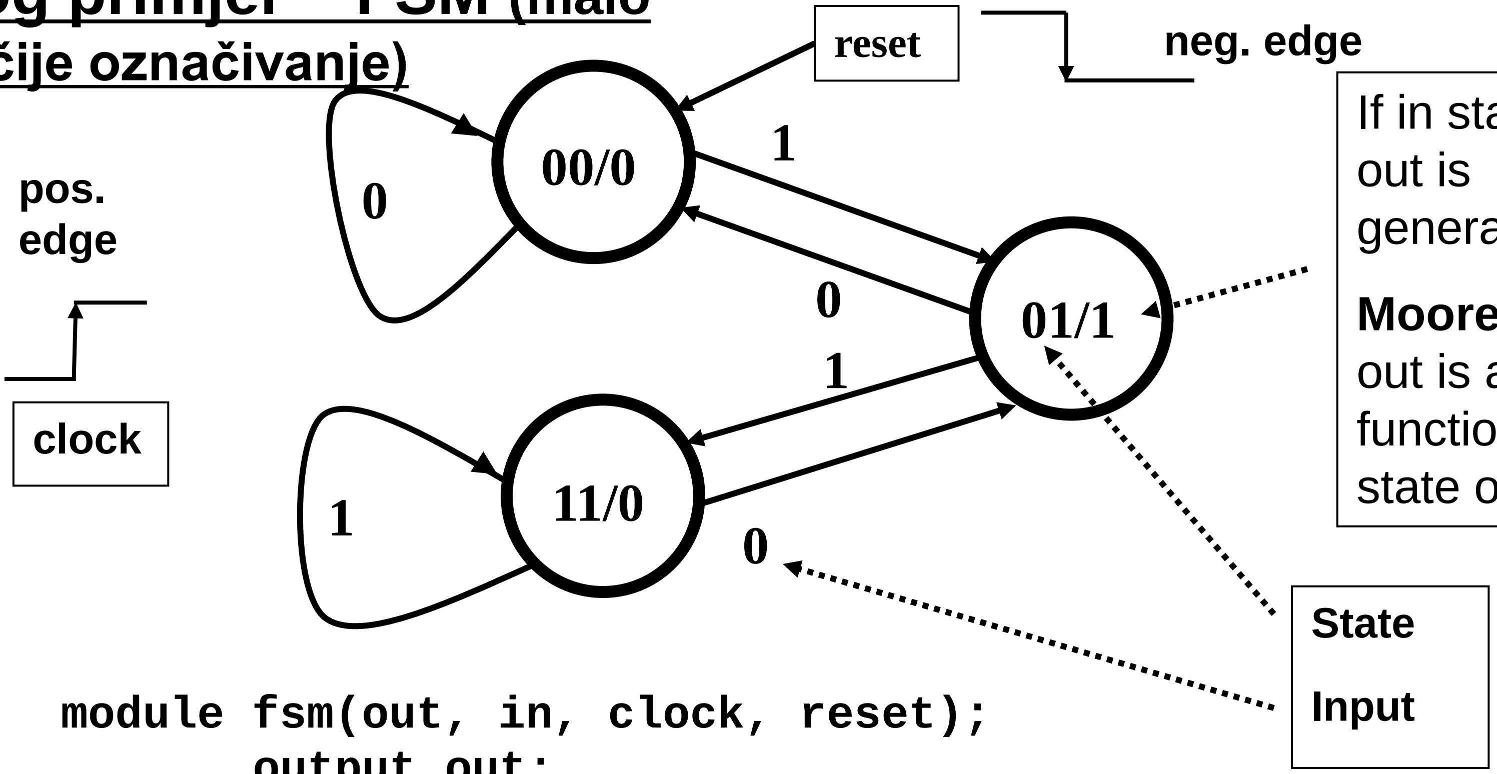
If true then, else

## Verilog primjer - FSM

```
            B:          // analogno za B, C, D, E
            C:
            D:
            E:
            F: begin
                  nextState = D;
                  out = (i == 0) ? 3'b101 : 3'b101;
               end
            default: begin // if undeined states, go to A
                        nextState = A;
                        out = (i == 0) ? 3'bxxx : 3'bxxx;
                     end             //out = don’t care
      endcase

// sequential part
      always @(posedge clock or negedge reset)
            if (~reset)                  // reset = low
                  currentState <= A;
            else                   // posedge clock
                  currentState <= nextState;
endmodule
```

nonblocking = acress the whole design

## Verilog primjer – FSM (malo drugačije označivanje)



If in state 1 out is generated.

**Moore FSM**: out is a function of state only

```
module fsm(out, in, clock, reset);
       output out;
       input  in, clock, reset;
       reg           out;
       reg           [1:0]  currentState, nextState;

              // combination portion

              // * * *

              // sequential portion

              // * * *

endmodule
```

```
// combination portion
always @(in or currentState) begin
      out = ~currentState[1] & currentState[0];
      // out = 1 only for state 01
      nextState = 0;
      if (currentState == 0)
            if(in) nextState = 1;   //else stay in 0
      if (currentState == 1)
            if (in) nextState = 3;  //else go to 0
      if (currentState == 3)begin
            if (in) nextState = 3;
            else nextState = 1;
            end
      end
// the sequential portion
always @(posedge clock or negedge reset)  begin
   if (~reset)
      currentState <= 0;             // as long as res=0
   else
      currentState <= nextState;   // as D type bistable
end
```

Bit select = 01

non blocking

## Verilog extensions – enumerated types

(similar to C)

```
/* def new type */
typedef enum {IDLE, READY, BUSY} controller_state;
/* contr._state is an enum type */


controller_state reg state;
/* state is a register variable of the type
“controller_state” */
```

## Verilog extensions – non-determinism

**There exist state-input pair for which the next state and output are not unique.**

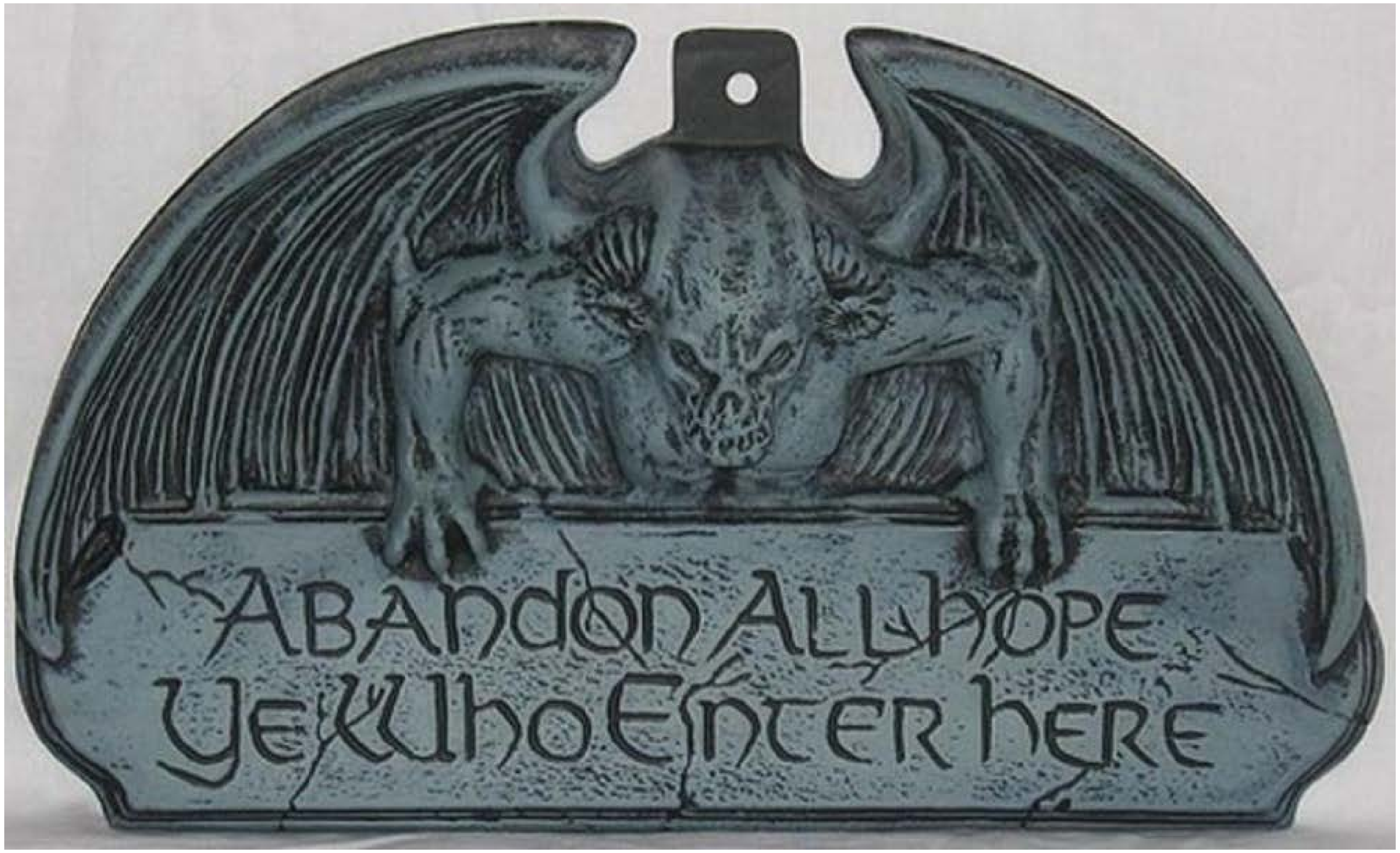
**$ND construct**

- **creates a nondeterministic signal source**
- **should only be used in an *assign* statement**

```
wire r;        /* def of a wire variable */
assign r=$ND(GO, NOGO);   /* nondeterminism */
.
.
always@(posedge clk) begin
.
.
state = r;
/* the state is nondeterm. GO or NOGO */
.
.
end
```

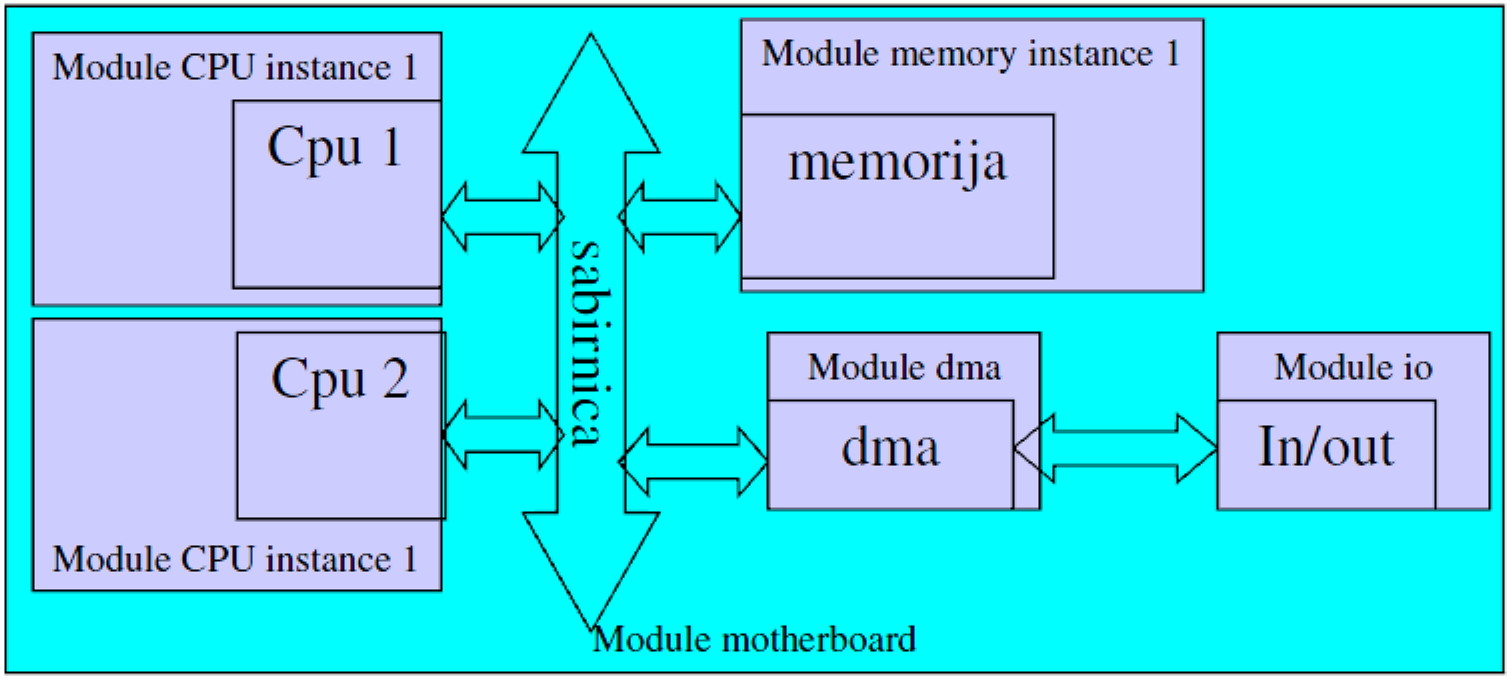
## SRANJE OD NOVIH PRIKAZNICA

**(to sve valjda piše negdje prije, neke su izbačene ili premještene)**



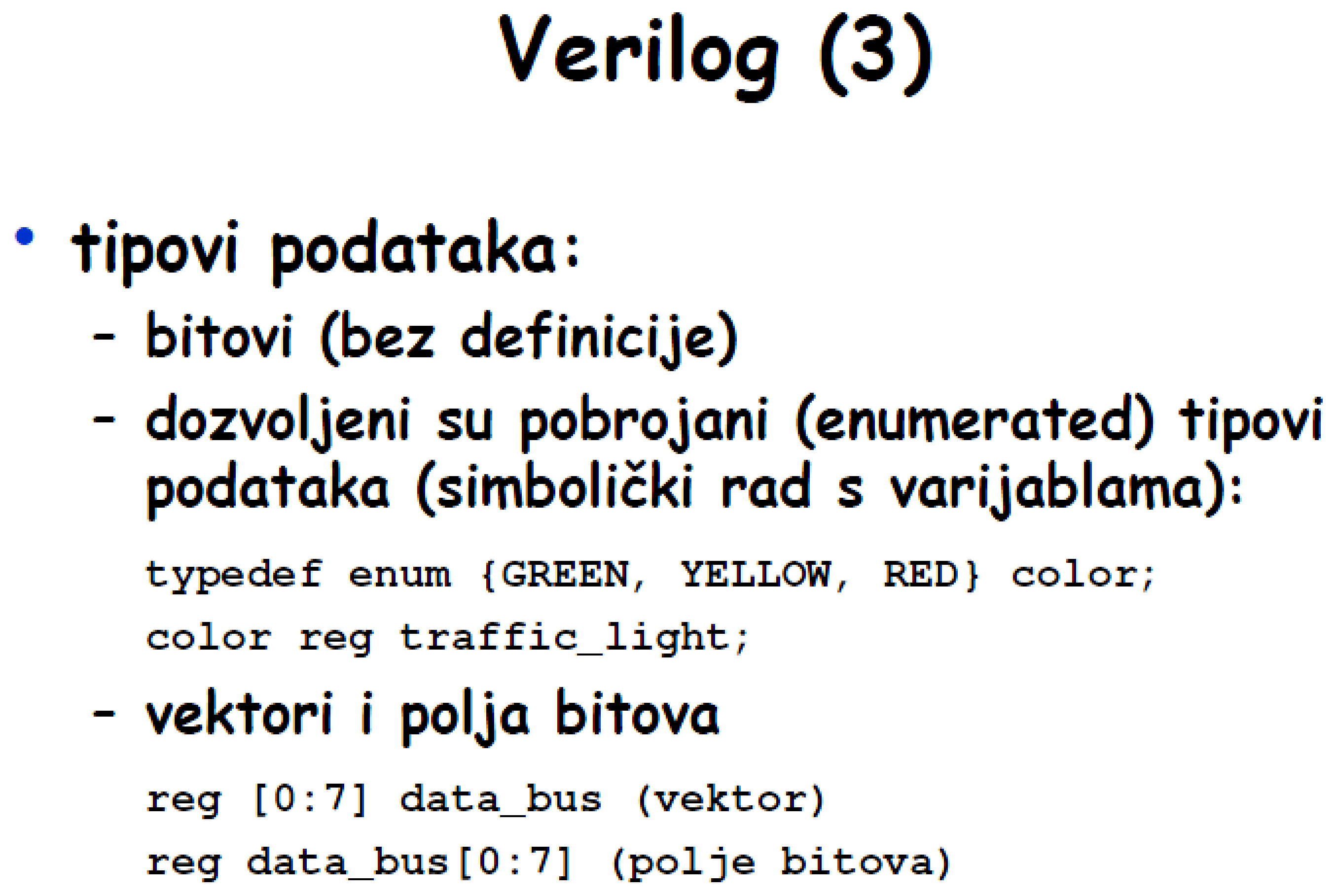
## Verilog (1)

- Hijerarhija modula
- Definirani moduli se mogu vise puta instancirati! = oprimjerivati
- U nadređenom modulu definira se povezivanje modula



## Verilog (2)

```
module main(clk); // MotherBoard
input clk;
tip_podataka vrsta_varijable ime_var1,ime_var2;
...                                  = primjerka
naziv_modula naziv_instance_modula(ime_var1,...); // CPU CPU1
...
endmodule

module naziv_modula(...) // CPU
...
endmodule
```



## Verilog (3)

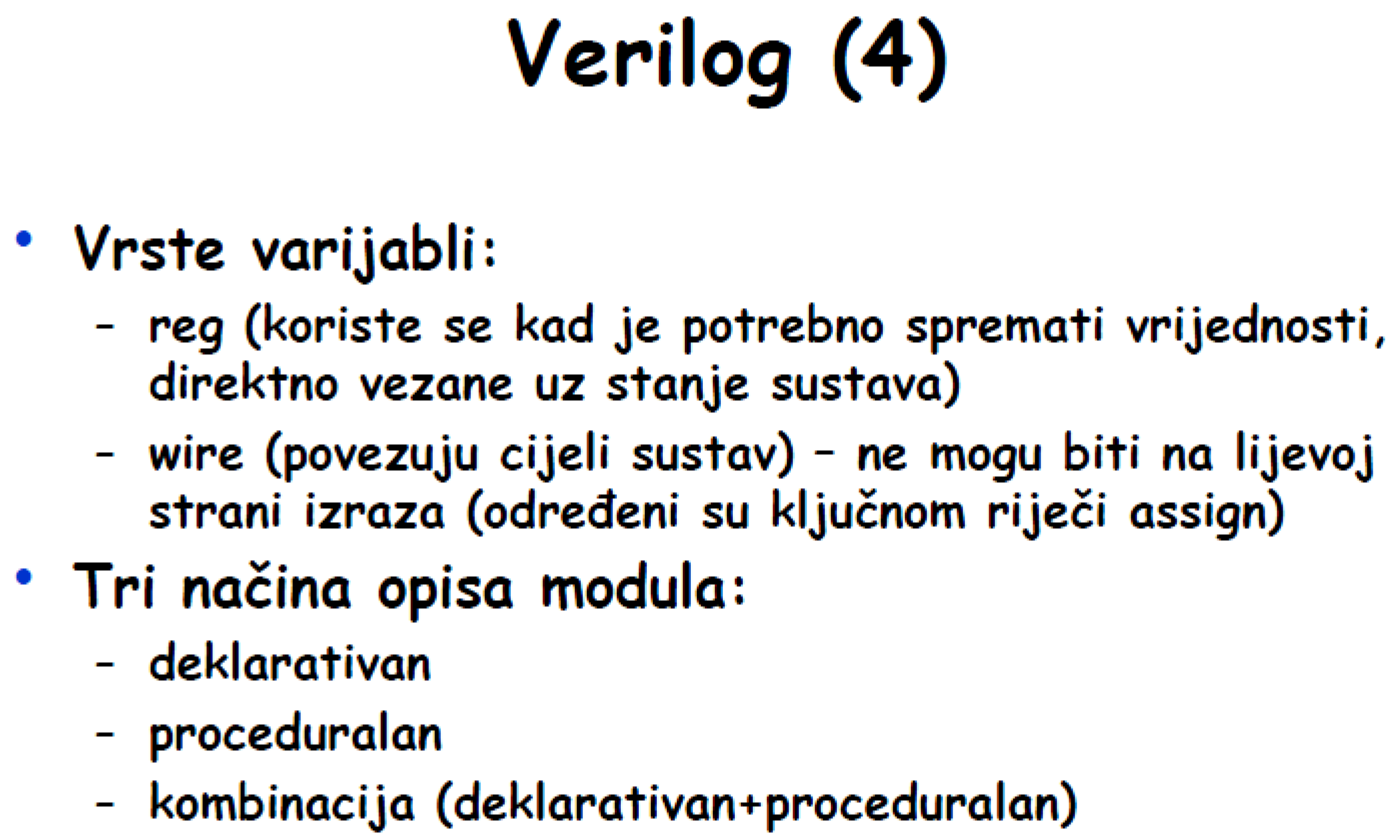
- tipovi podataka:
  - bitovi (bez definicije)
  - dozvoljeni su pobrojani (enumerated) tipovi podataka (simbolički rad s varijablama):

```
typedef enum {GREEN, YELLOW, RED} color;
color reg traffic_light;
```

  - vektori i polja bitova

```
reg [0:7] data_bus (vektor)
reg data_bus[0:7] (polje bitova)
```

## Sažetak 4

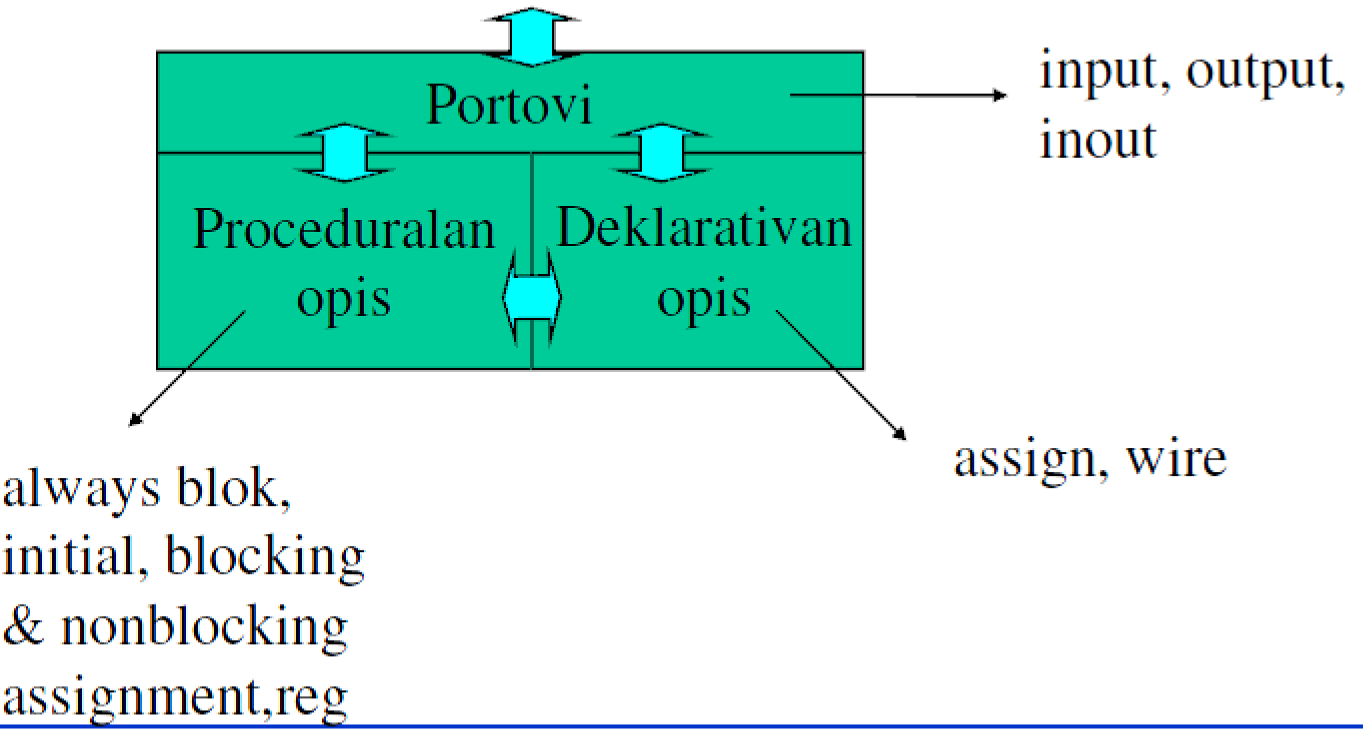


### Verilog (4)

- Vrste varijabli:
  - reg (koriste se kad je potrebno spremati vrijednosti, direktno vezane uz stanje sustava)
  - wire (povezuju cijeli sustav) – ne mogu biti na lijevoj strani izraza (određeni su ključnom riječi assign)
- Tri načina opisa modula:
  - deklarativan
  - proceduralan
  - kombinacija (deklarativan+proceduralan)

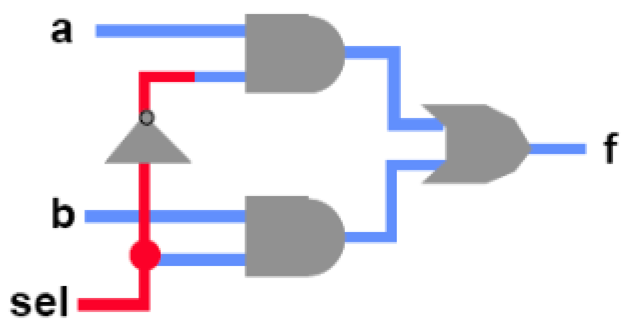
## Verilog (5)

- MODUL:



## Deklarativan opis 1 (structural model)

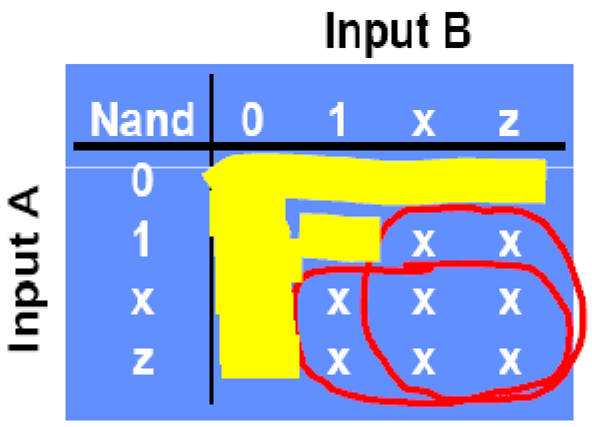
- Izgrađen je od logičkih vrata i drugih modula
- Prepoznati:ulaze, izlaze, instance vrata, kašnjenja



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

      and #5   g1 (f1, a, nsel),
               g2 (f2, b, sel);
      or   #5  g3 (f, f1, f2);
      not      g4 (nsel, sel);
endmodule
```

## Deklarativan opis 2 (4-valued logic)

- Strukture podataka simulatora dozvoljavaju
- 1,0,x,z
- 1,0 - očit smisao
- z - visoka impedancija (ima električki smisao)
- x - nepoznata vrijednost (značajno za simulator, ne ide u sintezu)



Input B

Input A

| Nand | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | x | x |
| x | | x | x | x |
| z | | x | x | x |

## Proceduralan ili ponašajni opis (behavioral model)

- Može se specificirati kombinacijska i sekvencijska logika
- Izgleda slično programiranju u C-u
- Sadrže initial i always blokove

| Statement | Looks like | Starts | How it works | Use in Synthesis? |
|---|---|---|---|---|
| initial | initial<br>begin<br>…<br>end | Starts when simulation starts | Execute once and stop | Not used in synthesis |
| always | always<br>begin<br>…<br>end | | Continually loop—while (power on) do statements; | Used in synthesis |

Pogledaj *clock.v*

## Sažetak 9

### Primjer - clock.v

```
module main();
   reg clock;

   initial begin
     $dumpfile("clock.vcd");
     $dumpvars(0,main);
     $dumpon;
     #2000 $finish;
   end
```

```
always begin
      clock=1;
      #10
      clock=0;
      #10
      clock=1;
   end
endmodule.
```

## Modeliranje vremena u Verilogu

- Bez kašnjenja u *always* bloku nastala bi beskonačna petlja u simulatoru
- # kašnjenje za *n* vremenskih jedinica (ne ide u sintezu) (npr. `#5` )
- `wait` čekanje da neka vrijednost dosegne određenu razinu (ne ide u sintezu)

  (npr. `wait(f==0) q=3` )
- @ čekanje na promjenu vrijednosti (koristi se u sintezi) (npr. `@(var) w=4` )

## Izrazi u Verilogu

- Mogu biti navedeni pomoću ključne riječi assign (samo wire) ili dodijeljeni proceduralno (initial ili always blok)
- 2 vrste dodjeljivanja
  - Blokirajuće =
  - Neblokirajuće <=
- operatori:
  - relacijski operatori: <,>,<=,>=,==,!=,===,!==
  - bit-operatori (poput C-a):~,&,|,^,~^, ^~,<<,>>
  - aritmetički operatori: +,-,*,/,%
  - logički operatori: ||,&&,!
- operator konkatenacije (npr. d = {a, b[3:0], c, 4'b1001})
- operator replikacije (npr. c={4{b}} je isto što i: c={b, b, b, b}

Pogledaj *blocking.v, non-blocking.v*

http://www.asic-world.com/verilog/operators2.html

Saž

## Primjer - blocking.v

```
module main();
  reg clock;
  reg [4:0] a,b;
  initial begin
    $dumpfile("blocking.vcd");
    $dumpvars(1,main);
    $dumpon;
    a=3;
    b=7;
    clock=0;
    #2000 $finish;
  end
  always @(posedge clock) begin
      a=b;
      b=a;
  end
  always begin
    #10  clock=!clock;
  end
endmodule
```

## Primjer - non-blocking.v

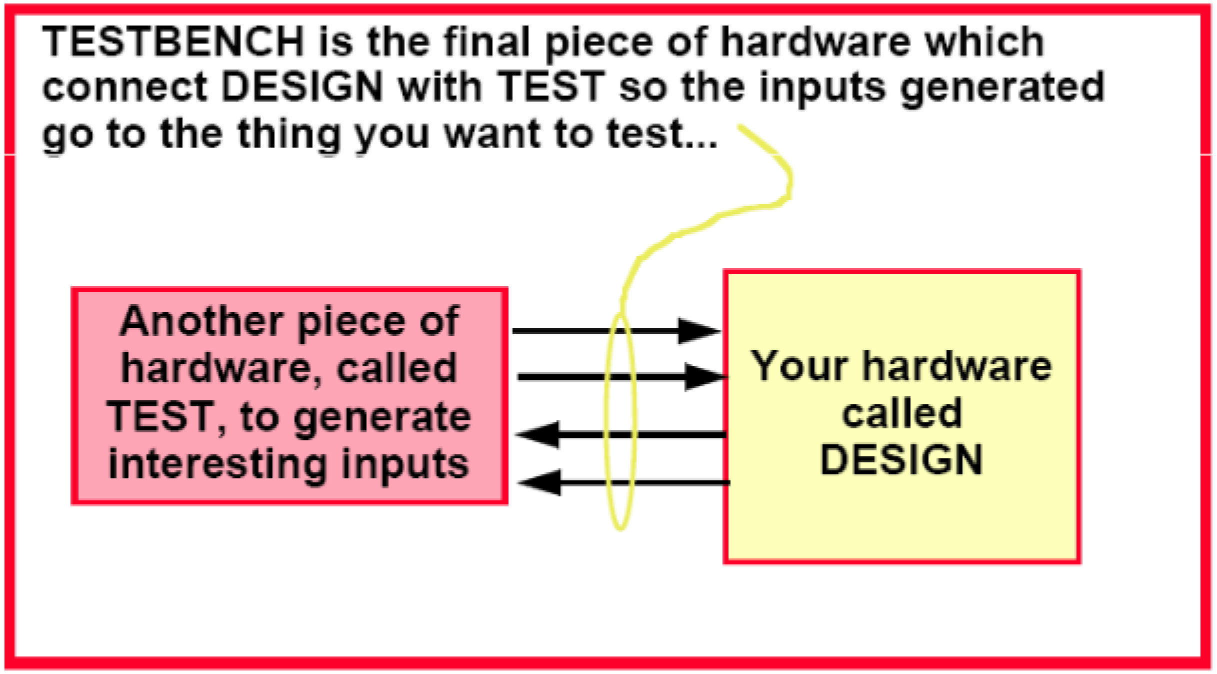
```
module main();
  reg clock;
  reg [4:0] a,b;
  initial begin
    $dumpfile("blocking.vcd");
    $dumpvars(1,main);
    $dumpon;
    a=3;
    b=7;
    clock=0;
    #2000 $finish;
  end
  always @(posedge clock) begin
      a<=b;
      b<=a;
  end
  always begin
    #10  clock=!clock;
  end
endmodule
```

## Ograničenja portova u modulima, petlje

- Za portove modula mora vrijediti sljedeće:
  - Input i inout port moraju biti wire
  - Output port mora biti wire ako ga je generirao podmodul
  - Output port mora biti wire ako je određen deklarativno (assign)
  - Output port mora biti reg ako je određen proceduralno
- Postoje petlje for i while (ne koriste se u sintezi, služe za testiranje modula)

## Testiranje modula I

- 3 dijela u Verilogu


TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...
Another piece of hardware, called TEST, to generate interesting inputs
Your hardware called DESIGN

## Testiranje modula II (primjer)

```
module tBench;
    wire      su, co, a, b;

    halfAdd     ad(su, co, a, b);
    testAdd     tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
    output    sum, cOut;
    input     a, b;

    xor #2    (sum, a, b);
    and #2    (cOut, a, b);
endmodule
```

```
module testAdd(a, b, sum, cOut);
    input    sum, cOut;
    output   a, b;
    reg      a, b;

    initial begin
        $monitor ($time,,
          "a=%b, b=%b, sum=%b, cOut=%b",
           a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

## III. Formalna verifikacija digitalnih sustava

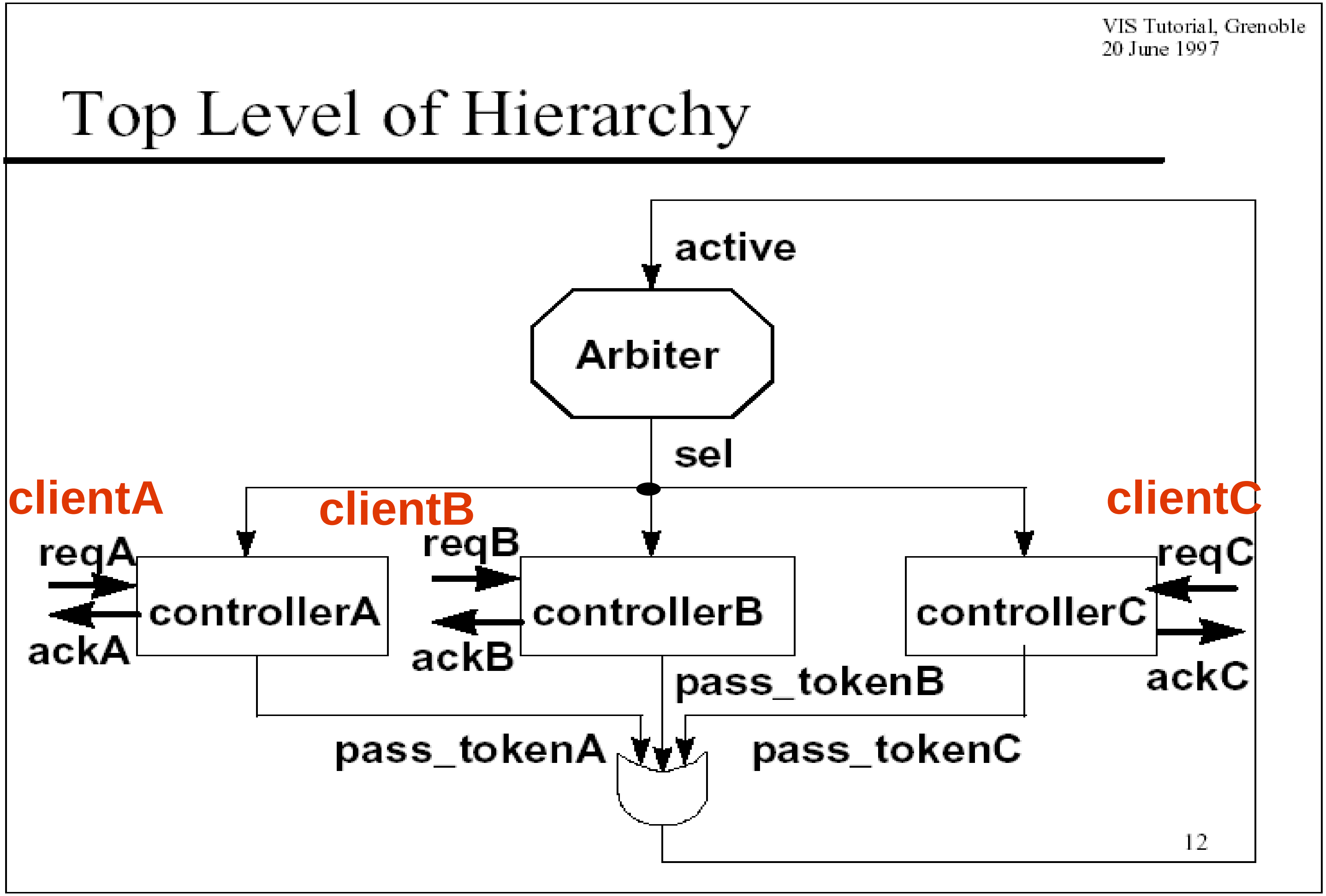
**Ciljani sustav: dodjeljivač sabirnice (engl.** *bus arbiter***)**

**Opis implementacije ( *I* ): Verilog**

**Sustav za verifikaciju: VIS**

**Opis specifikacije ( *S* ): CTL vremenska logika**

## Implementacija dodjeljivača sabirnice


VIS Tutorial, Grenoble
20 June 1997

Top Level of Hierarchy

## Dodjeljivač sabirnice – opis s najviše razine apstrakcije

- Korisnici (clientA, clientB, clientC) spojeni su na tri upravljačka sklopa (controllerA, controllerB, controllerC) preko kojih nastoje zauzeti sabirnicu.
- U jednom trenutku samo jedan upravljački sklop (controller) može upravljati sabirnicom kao zajedničkim sredstvom (resursom).
- Upravljački sklopovi (controller A, B, C) komuniciraju sa središnjim sklopom "Arbiter" koji mora osigurati da u jednom trenutku samo jedan upravljački sklop (controller) ima nadzor nad sabirnicom.
- Međusobna isključivost upravljanja sabirnicom ostvaruje se postojanjem jedne i samo jedne značke (engl. *token*) koju može ekskluzivno posjedovati controllerA, ili controllerB, ili controllerC, ili nijedan upravljački sklop (controller).

## Dodjeljivač sabirnice – opis s najviše razine apstrakcije

- Izlazni signal “sel” iz sklopa “Arbiter” govori kojem upravljačkom sklopu će biti ponuđena značka (token).
- Moguće stanje izlaza “sel”:

  A – controllerA će moći prihvatiti značku.

  B – controllerB će moći prihvatiti značku.

  C – controllerC će moći prihvatiti značku.

  X – Arbiter ne nudi značku (značka je u posjedu jednog od controllera A exili B exili C).

  sel $\in$ {A, B, C, X}
- Arbiter, controllerA, controllerB, controllerC, clientA, clientB, clientC upravljani su zajedničkim globalnim signalom takta (clk).

## Dodjeljivač sabirnice – ponašanje upravljačkog sklopa

- Svaki upravljački sklop (controller) ima odgovarajući izlazni signal: pass_tokenA, pass_tokenB, pass_tokenC.
- Svaki upravljački sklop može postaviti pass_token = 1 ako:
    - Imao je značku i više mu nije potrebna
    - Ponuđena mu je značka, ali mu nije potrebna
- Sva tri pass_token izlaza iz upravljačkih sklopova (controllerA, controllerB, controllerC) spojena su preko logičkog ILI sklopa (engl. OR) koji generira signal "active".
- active = 1 znači da upravljački sklopovi (controller A ili B ili C) šalju (prosljeđuju) značku, tj. nije im potrebna.
- active = 0 znači da značka nije dostupna (drži je neki od upravljačkih sklopova) , te Arbiter na svom izlazu sel generira X (sel = X).

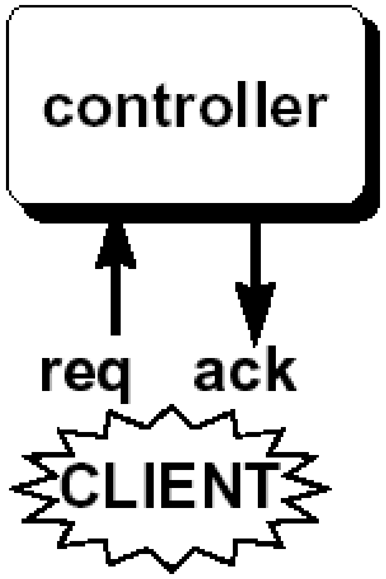
## Dodjeljivač sabirnice – ponašanje Arbitra

- Arbitar u svakom taktu (engl. *clock*) ciklički nudi značku (token): značka_za_A → značka_za_B → značka_za_C → nema _značke →…
- Trenutno (sadašnje) stanje Arbitra pokazuje njegov izlazni signal sel:

| | |
|---|---|
| sel = A | značka_za_A |
| sel = B | značka_za_B |
| sel = C | značka_za_C |
| sel = X | nema_značke |

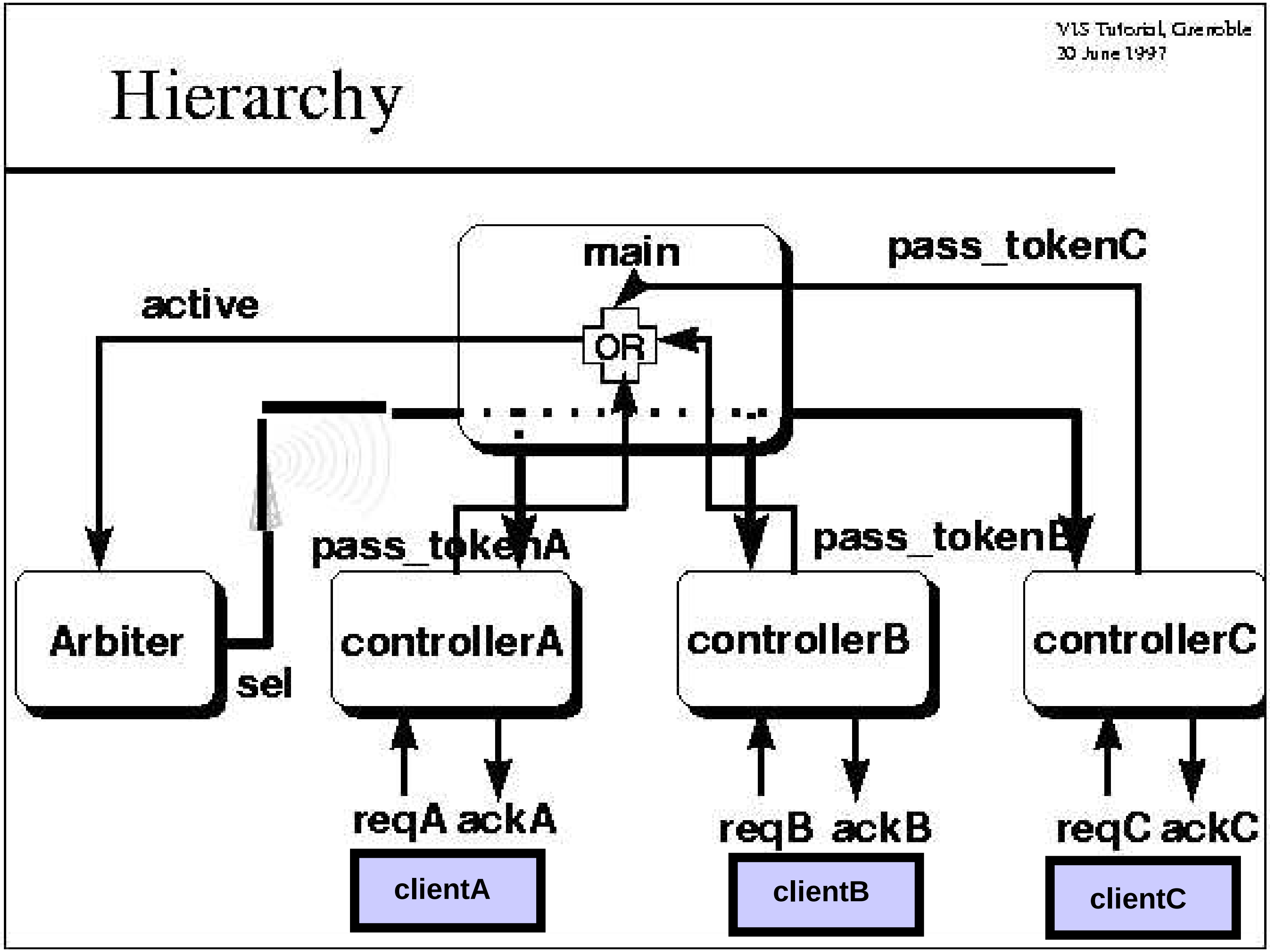
- Prikazani koncept omogućuje svakom upravljačkom sklopu (controller A, B, C) da se dočepa značke, tj. sabirnice. Kada jedan upravljački sklop otpusti značku drugi upravljački sklopovi je mogu prihvatiti. Barem se tako nadamo.

## Dodjeljivač sabirnice – ponašanje klijenta (Client)



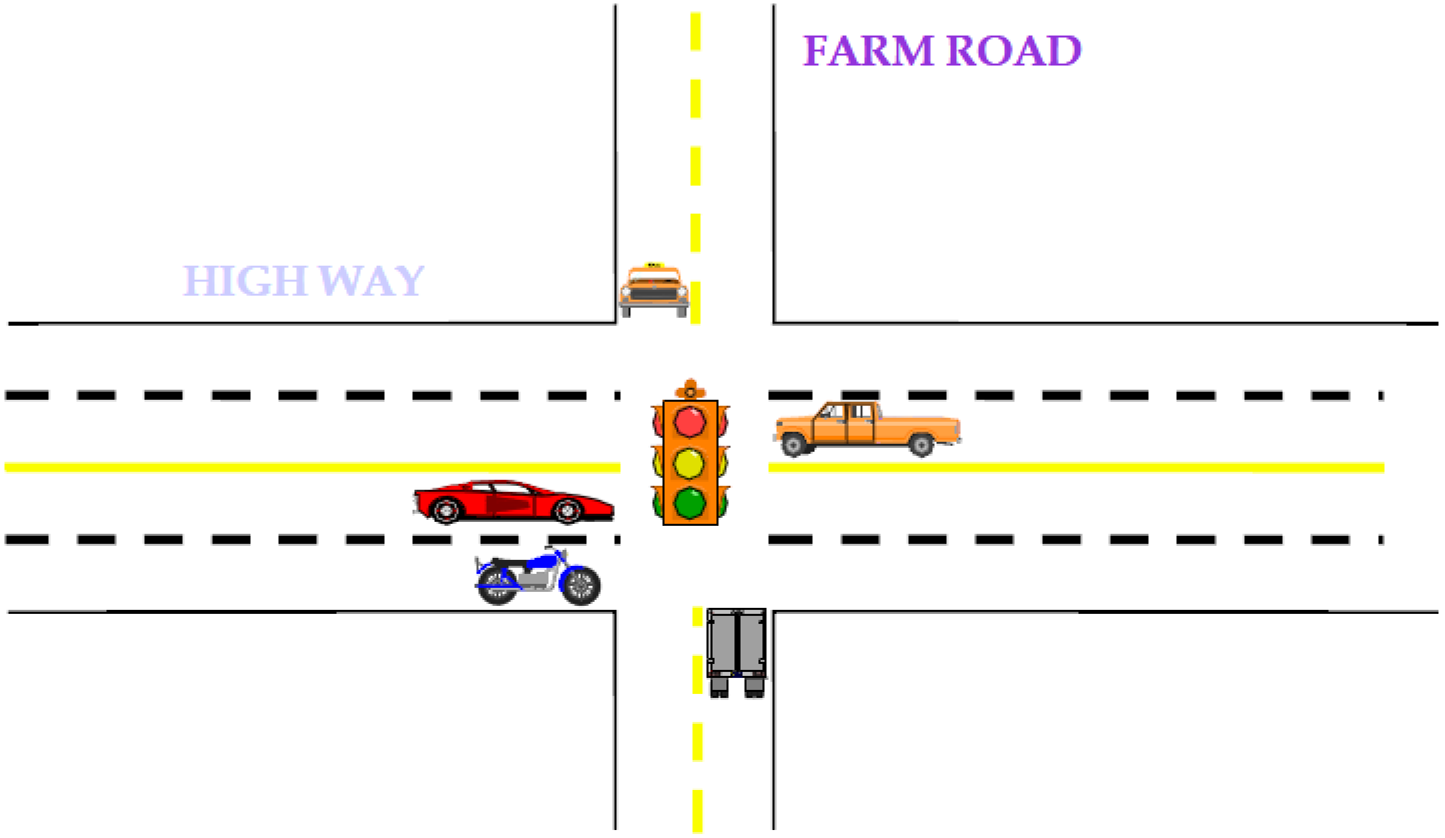
- Kada klijent (clientA, clientB, clientC) zatraži sabirnicu postavlja signal req u logičku jedinicu.
- Klijent može držati signal req u logičkoj jedinici po volji dugo.
- Upravljački sklop (controller A ili B ili C) postavlja i drži signal ack u logičkoj jedinici točno za vrijeme dok posjeduje značku (token).

## Example: Arbiter

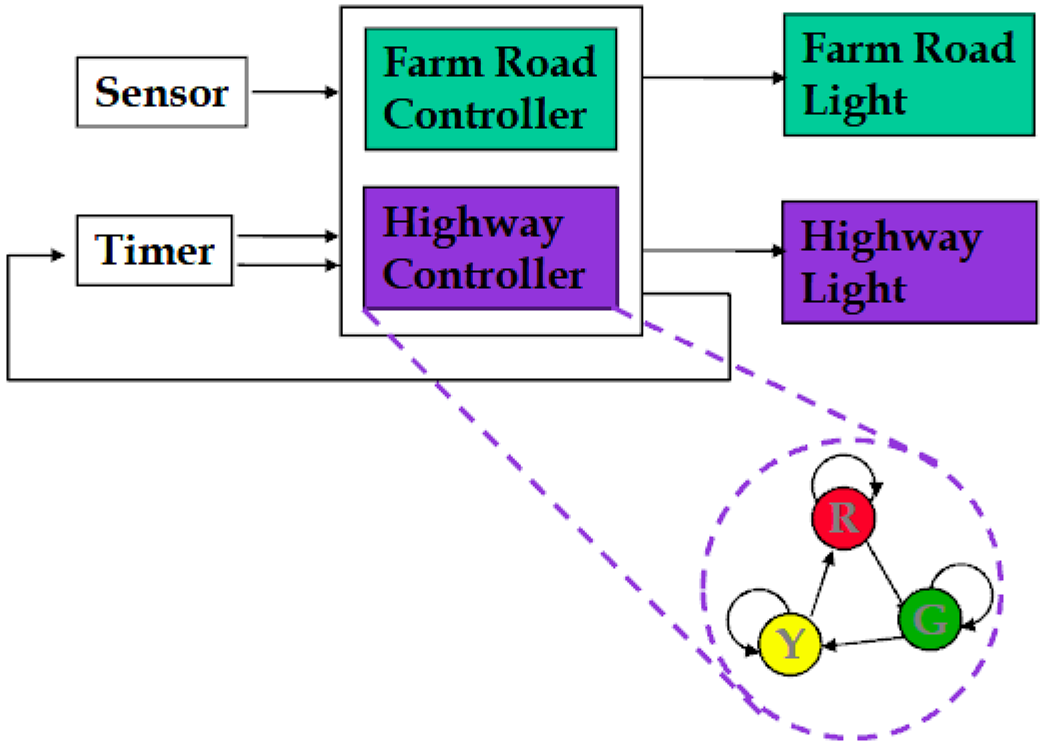


```
module main(clk);
          …         // typedef
          …         // input, output, wire, reg
          ...
          controller controllerA(clk, reqA, ackA, sel, pass_tokenA, A);
          controller controllerB(clk, reqB, ackB, sel, pass_tokenB, B);
          controller controllerC(clk, reqC, ackC, sel, pass_tokenC, C);
          arbiter arbiter(clk, sel, active);
          client clientA(clk, reqA, ackA);
          client clientB(clk, reqB, ackB);
          client clientC(clk, reqC, ackC);
endmodule

module controller(clk, req, ack, sel, pass_token, id);
          input clk, req, sel, id;
          output ack, pass_token;
          ….
endmodule

module arbiter(clk, sel, active);
          input clk, active;
          output sel;
          ...
endmodule

module client(clk, req, ack);
          input clk, ack;
          output req;
          ...
endmodule
```

## IV. Upravljač semaforom (TLC)



## Upravljač semaforom (TLC)



## Moore, Mealy

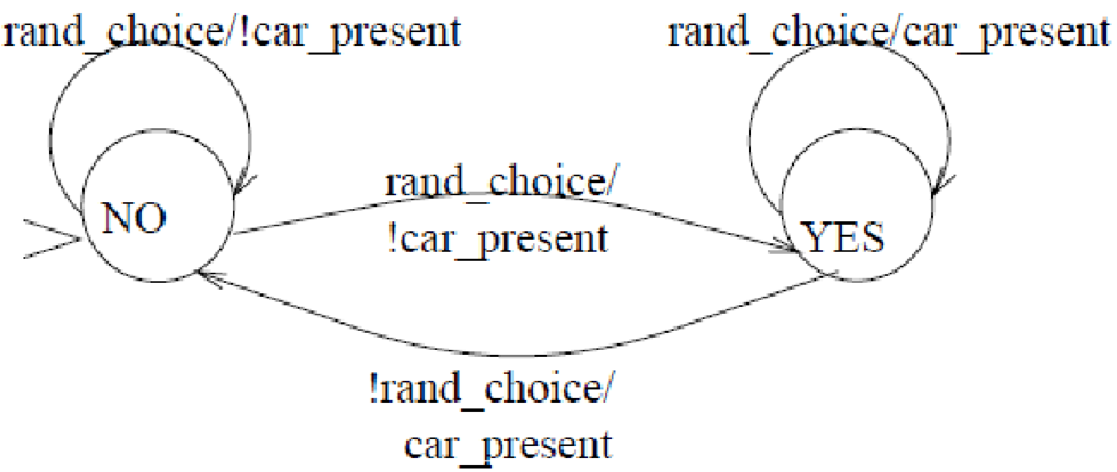
$$Q_{NEXT} = f(x, Q)$$

Mealy: $z = g(Q, x)$

Moore: $z = h(Q)$

SENSOR

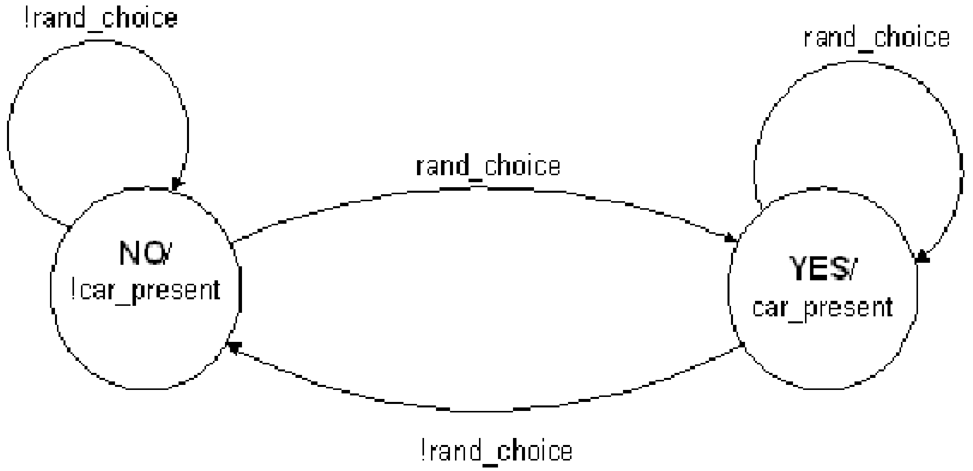
MEALY:



Kod Mealyjevog automata, **izlaz** iz stanja je ovisan o ulaznom podatku i o trenutnom stanju.

Kod Mooreovog automata, **izlaz** iz stanja je ovisan samo o trenutnom stanju.

MOORE:



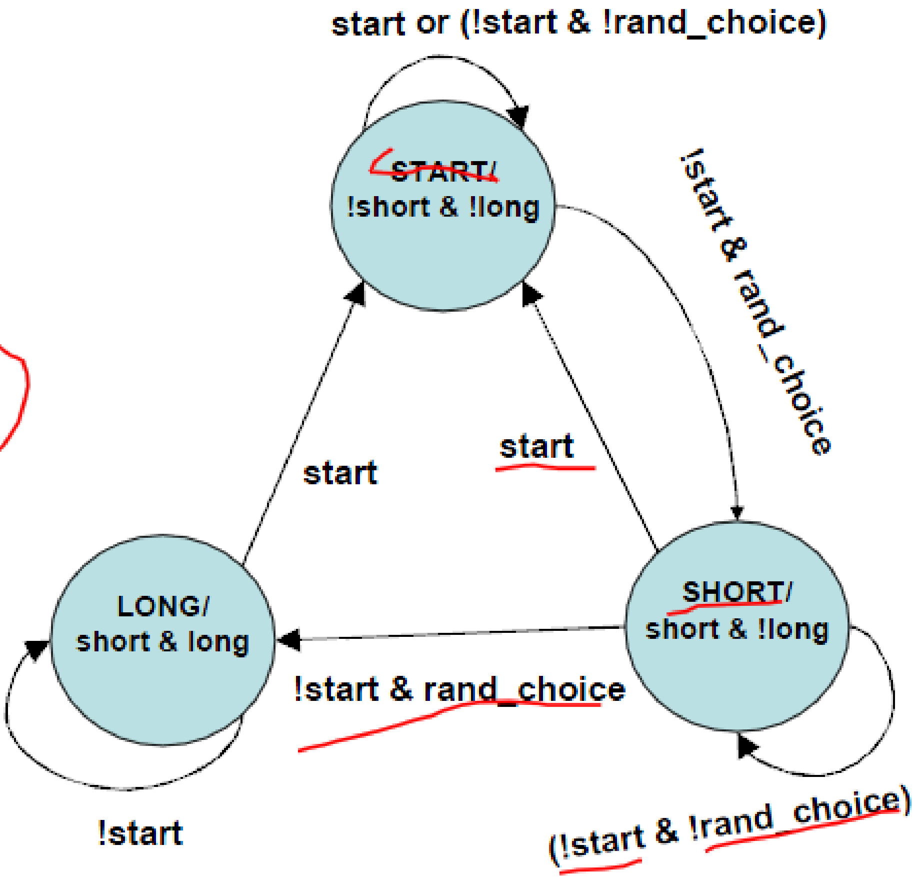
## Upravljač semaforom (TLC)

```
module timer(clk, start, short, long);

input clk;
input start;
output short;
output long;

wire rand_choice;
wire start, short, long;
timer_state reg state;

initial state = START;

assign rand_choice = $ND(0,1);

/* short could as well be assigned to be just (state == SHORT) */
assign short = ((state == SHORT) || (state == LONG));
assign long = (state == LONG);
```

```
always @(posedge clk) begin
        if (start) state = START;
        else
                begin
                   case (state)
                   START:
                        if (rand_choice == 1) state = SHORT;
                   SHORT:
                        if (rand_choice == 1) state = LONG;
        /* if LONG, remains LONG until start signal received */
                        endcase
                end
end
endmodule
```

## Verilog → FSM

```
module timer(clk, start, short, long);

input clk;
input start;
output short;
output long;

wire rand_choice;
wire start, short, long;
timer_state reg state;

initial state = START;

assign rand_choice = $ND(0,1);

assign short = ((state == SHORT) || (state == LONG));
assign long = (state == LONG);

always @(posedge clk) begin
  if (start) state = START;
  else
    begin
     case (state)
     START:
        if (rand_choice == 1) state = SHORT;
     SHORT:
        if (rand_choice == 1) state = LONG;
        /* if LONG, remains LONG until start signal received */
     endcase
    end
end
endmodule
```



## V. Zadatci

### Pretvorba Verilogova modela u Kripkeovu strukturu

- Za dva zadana odsječka Verilog koda:

```
module main(clk);
input clk;
reg a;
wire b;
     initial a=0;
     assign b = $ND(0,1);
     always @(posedge clk) begin
          a=b;
     end
endmodule
```

```
module main(clk);
reg a;
input clk;
    initial a=0;
    always @(posedge clk) begin
          a=!a;
    end
endmodule
```

**odredi istinitost sljedećih CTL specifikacija: AG(a=0); EF(a=0); EG (a=0); AG(AF(a=0)).**

**Obrazloži odgovor.**

## Zadatci

### Pretvorba Kripkeove strukture u Verilogov model

- **Potrebno je izgraditi jedan Verilog modul koji zadovoljava sljedeće CTL specifikacije:**
  - AG(p->EX(EG(q)))
  - AG(q->EX(EG(p)))
  - $AG((p \land \neg q) \lor (\neg p \land q))$

  **Zadovoljava li izgrađeni modul specifikaciju AF(p)**

- **Potrebno je izgraditi jedan Verilog modul koji zadovoljava sljedeće CTL specifikacije:**
  - AG(p->EX(q))
  - AG(q->EX(EG(r)))

  **Zadovoljava li izgrađeni modul specifikaciju AG(p->EF(r))**