Catedra de Calculatoare
Facultatea de Automatica si Calculatoare

Calculatoare Numerice

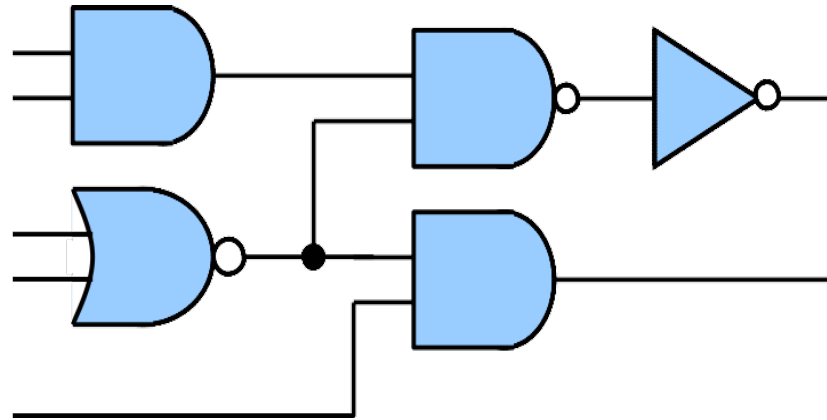
- Cursul 2 -

Introducere în Verilog HDL

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

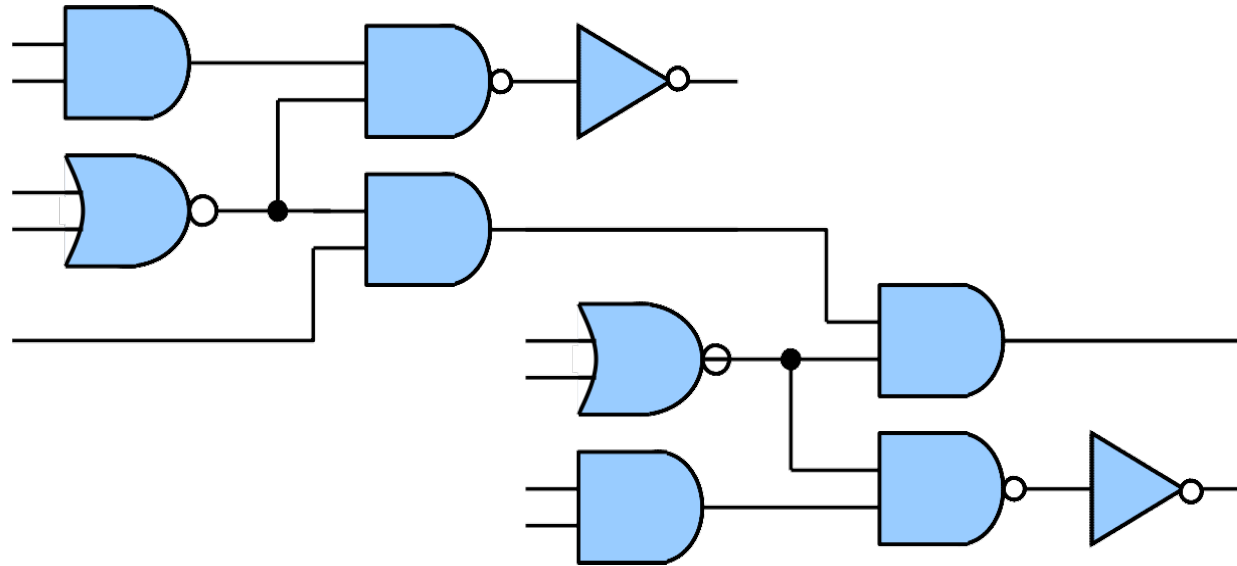
VERILOG DESIGN PROBLEMS
imgflip.com

Hardware Description Languages



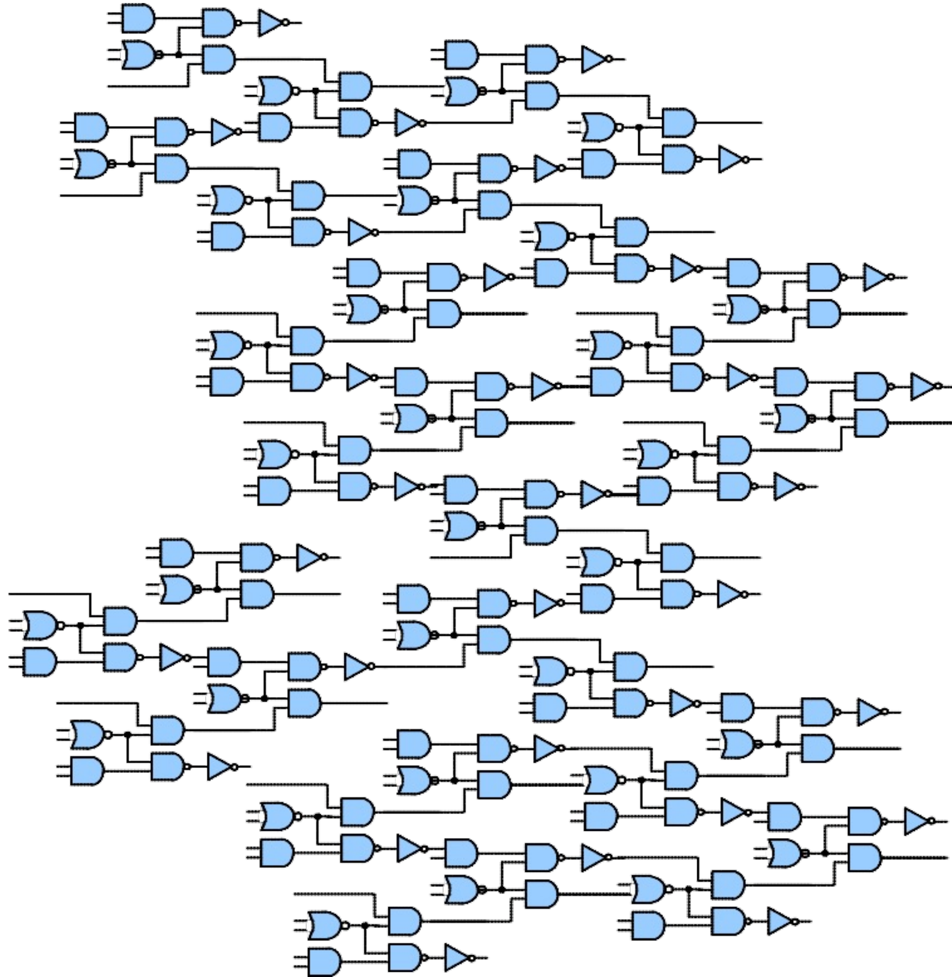
- La început, design-urile logice includeau doar câteva porți logice
- Era foarte ușor de verificat un astfel de design pe hârtie sau pe un breadboard

Hardware Description Languages



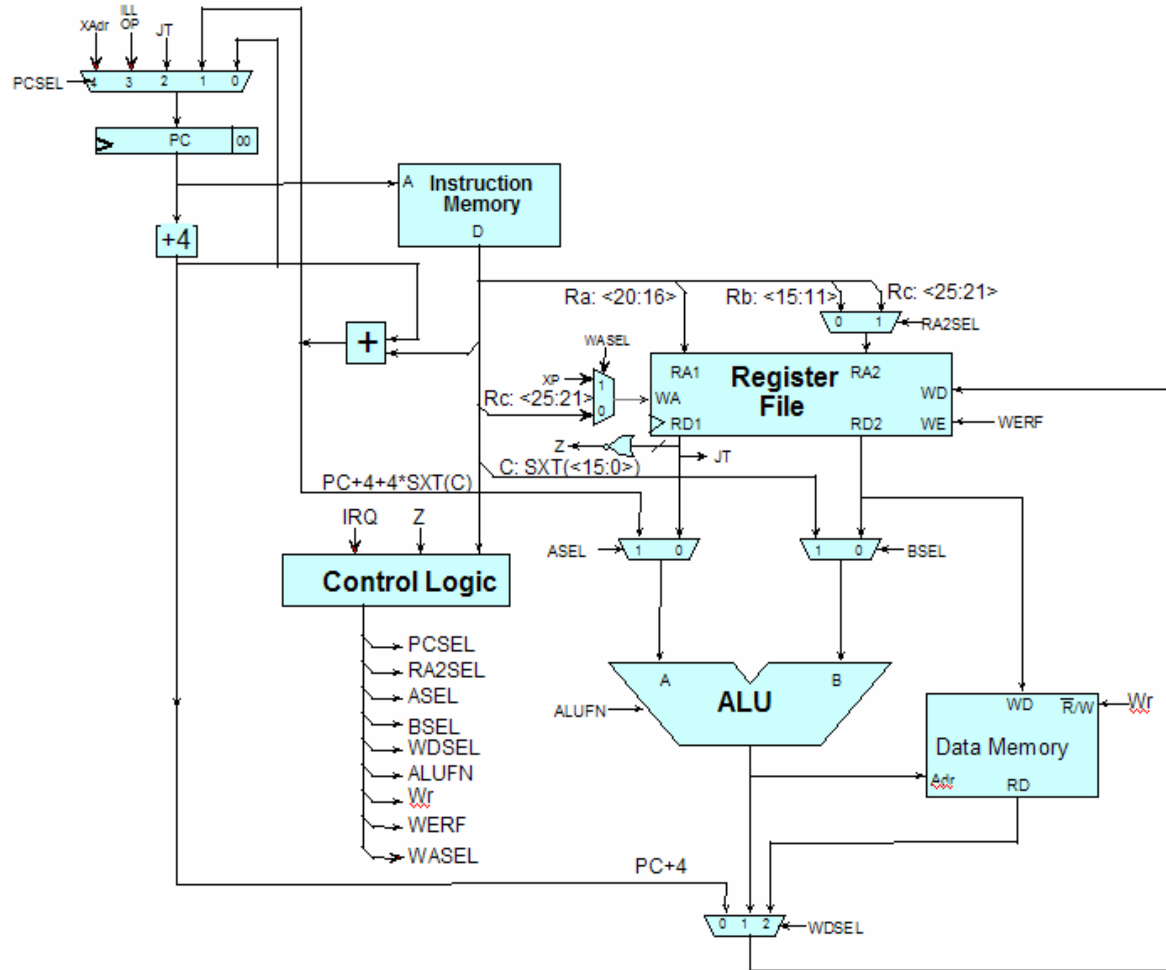
Pe măsură ce schemele logice ale dispozitivelor au devenit mai complicate, proiectanții au început să folosească modele logice la nivelul porților logice.
Acestea descriau într-un limbaj Hardware Description Language (HDL) comportamentul circuitului și ajutau la verificarea corectitudinii circuitului înainte de fabricație

Hardware Description Languages



- Atunci când proiectanții au început să lucreze cu design-uri de peste 100,000 porți logice, modelele la nivel de poartă logică erau prea simple.
- S-a trecut la limbaje de descriere hardware de nivel înalt.

Hardware Description Languages

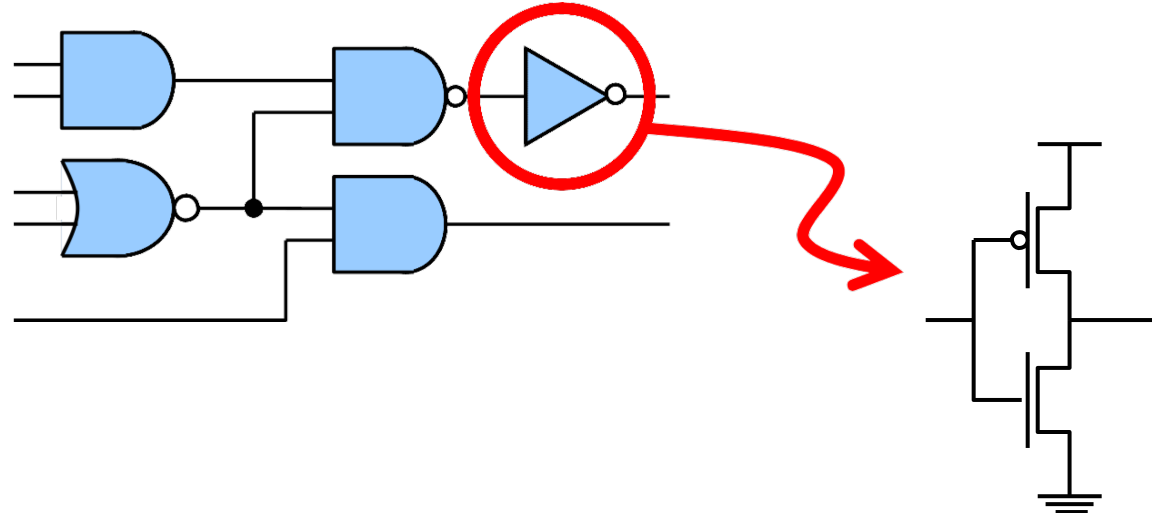


Design-ul HDL s-a dovedit util și pentru nivelul acesta de abstractizare pentru că un limbaj HDL poate genera o specificație precisă și un fundament pentru construirea de noi și noi design-uri, din ce în ce mai complicate.

Avantajele HDL

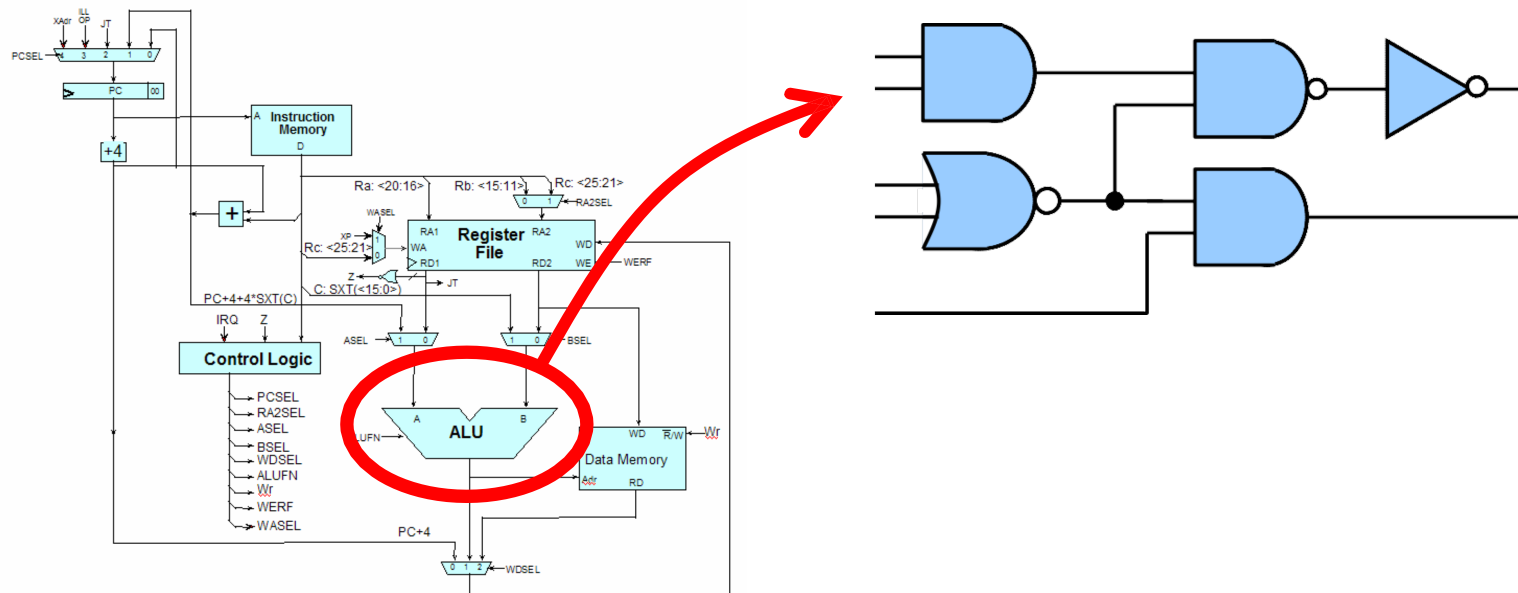
Permite proiectanților să discute despre ce ar trebui să facă hardware-ul fără să proiecteze hardware-ul în sine.
Cu alte cuvinte, un HDL separă comportamentul unui circuit de implementarea acestuia, prin diferite nivele de abstractizare.

HDL reușește asta prin module și interfețe



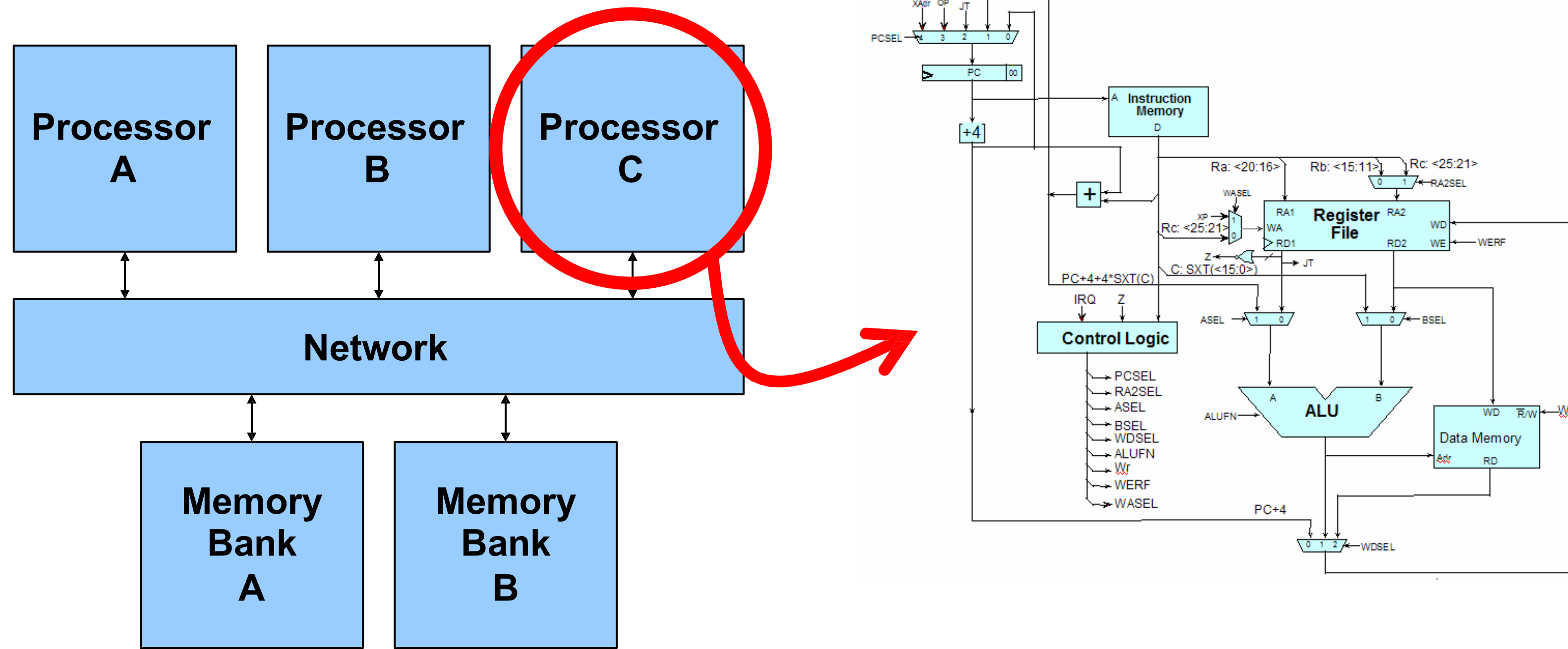
Avantajele HDL

Permite proiectanților să discute despre ce ar trebui să facă hardware-ul fără să proiecteze hardware-ul în sine. Cu alte cuvinte, un HDL separă comportamentul unui circuit de implementarea acestuia, prin diferite nivele de abstractizare.



Avantajele HDL

Permite proiectanților să discute despre ce ar trebui să facă hardware-ul fără să proiecteze hardware-ul în sine. Cu alte cuvinte, un HDL separă comportamentul unui circuit de implementarea acestuia, prin diferite niveluri de abstractizare.



Avantajele HDL

- Proiectanții pot să dezvolte specificații funcționale complete care descriu cu exactitate comportamentul tuturor componentelor și interfețelor acestora
- Proiectanții pot să ia decizii legate de cost,performanță, consum de energie încă din primele etape ale proiectarii unui circuit logic
- Proiectanții pot să creeze diferite programe și unelte care să manipuleze automat design-ul în etapele de verificare, sinteză, optimizare etc.

A tale of two HDLs

VHDL	Verilog
Sintaxă ADA-like, multă redundanță	Sintaxă C-like, concisă
Tipuri extensibile și motor pentru simulare	Tipuri și reprezentări logice prestabilite
Un design este compus din entități, fiecare entitate poate avea arhitecturi multiple	Un design este compus din module care au o singură implementare
Gate-level, dataflow, modelare comportamentală. Sintetizabil	Gate-level, dataflow, modelare comportamentală. Sintetizabil
Mai greu de învățat și folosit	Ușor de învățat și folosit, simulare rapidă

Vom folosi Verilog…

Avantaje

- Alegerea majorității echipelor de design
- Sunteți deja familiarizați cu sintaxa C
- Sintaxă simplă modul/port organizează un proiect în mod intuitiv și ierarhic, chiar și dacă acesta descrie o structură complexă (procesor)
- Potrivit pentru verificarea și sinteza circuitelor logice

Dezavantaje

- Câteva ”chichițe” care sunt mai greu de înțeles de către începători
- Sintaxa C poate să facă pe începători să creadă că Verilog funcționează și la nivel semantic ca C-ul
- E foarte ușor să scrii cod foarte urât. Este mai important decât la C ca programele să fie ușor de înțeles și parcurs.

HDL NU ESTE un limbaj de programare software!!!

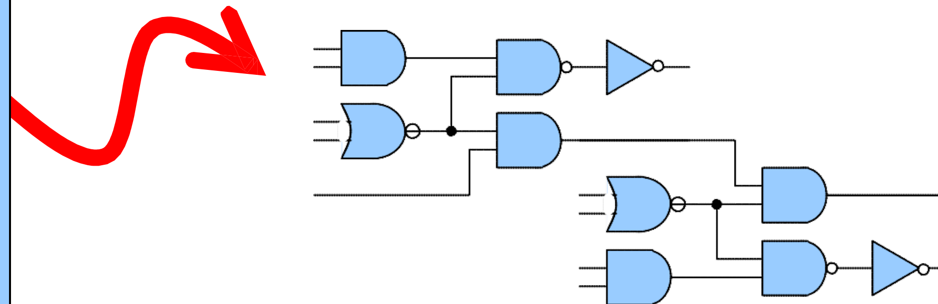
Limbaj programare software

- Limbaj ce poate fi translatat în instrucțiuni în cod mașină, ce poate fi apoi executat de un procesor.

Hardware Description Language

- Limbaj pentru structurii spațiale și a comportamentului temporal a unui sistem hardware

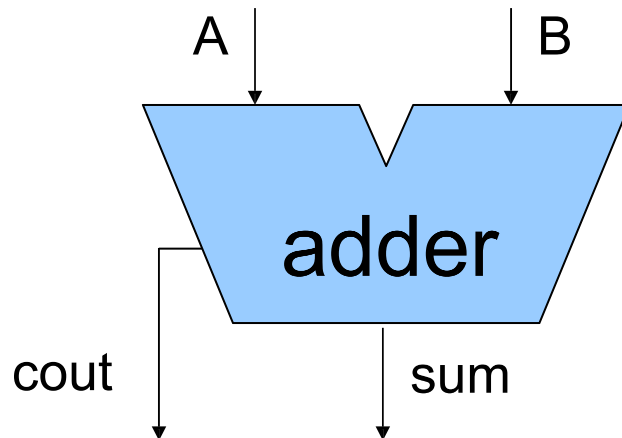
```
module foo(clk,xi,yi,done); input
   [15:0] xi,yi; output done;

   always @(posedge clk) begin:
         if (!done) begin
            if (x == y) cd <= x; else (x > y) x <=
            x - y;
         end
      end
endmodule
```



Modelare ierarhică cu Verilog

Un modul Verilog are un nume și o interfață sub forma unei liste de porturi

– Trebuie să specificăm direcția și lățimea pe biți a fiecărui port



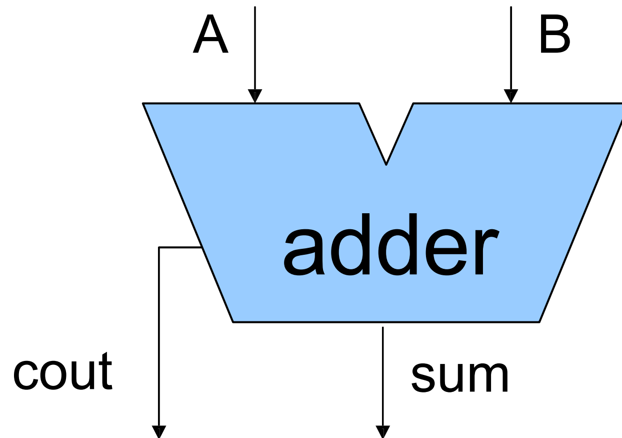

```
module adder( A, B, cout,  sum );
  input   [3:0] A, B;
  output        cout;
  output  [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```

Nu uitați punct și virgulă!!!

Modelare ierarhică cu Verilog

Un modul Verilog are un nume și o interfață sub forma unei liste de porturi

- Verilog-2001 a introdus o exprimare succintă tip ANSI C pentru porturi

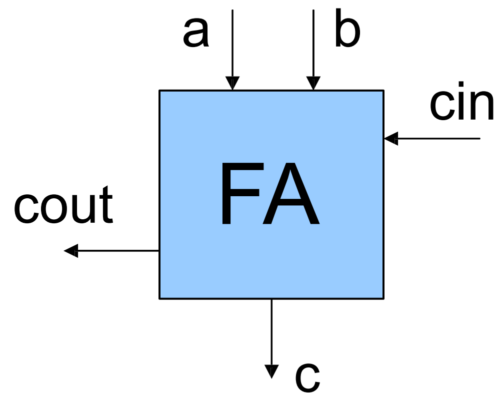



```
module adder( input   [3:0] A, B,
              output        cout,
              output  [3:0] sum );

  // HDL modeling of 4 bit
  // adder functionality

endmodule
```

Modelare ierarhică cu Verilog

Un modul poate conține alte module prin instanțierea lor. Asftel, se creează o ierarhie de module

- Modulele sunt conectate prin linii de legatură
- Porturile sunt atașate la aceste linii fie prin poziționare, fie prin numele lor

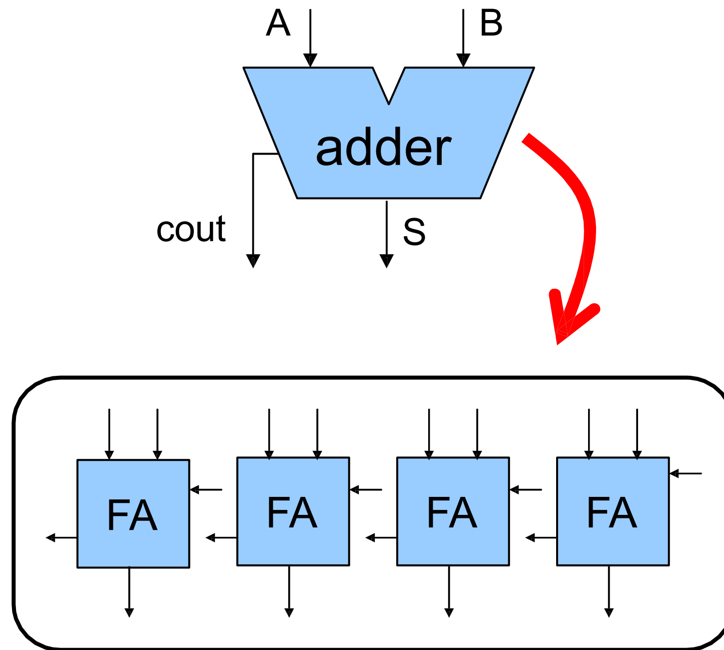



```
module FA( input   a, b, cin
           output  cout, sum );

  // HDL modeling of 1 bit
  // adder functionality

endmodule
```

Modelare ierarhică cu Verilog

Un modul poate conține alte module prin instanțierea lor. Asftel, se creează o ierarhie de module

- Modulele sunt conectate prin linii de legatură
- Porturile sunt atașate la aceste linii fie prin poziționare, fie prin numele lor

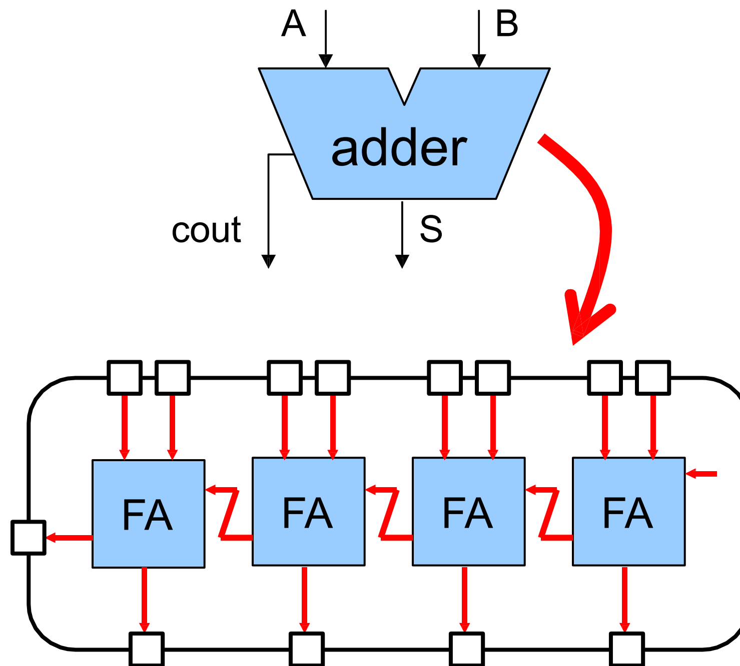



```
module adder(  input [3:0] A,B,
               output cout,
               output [3:0] S );
  FA fa0( ... );
  FA fa1( ... );
  FA fa2( ... );
  FA fa3( ... );

endmodule
```

Modelare ierarhică cu Verilog

Un modul poate conține alte module prin instanțierea lor. Asftel, se creează o ierarhie de module

- Modulele sunt conectate prin linii de legatură
- Porturile sunt atașate la aceste linii fie prin poziționare, fie prin numele lor



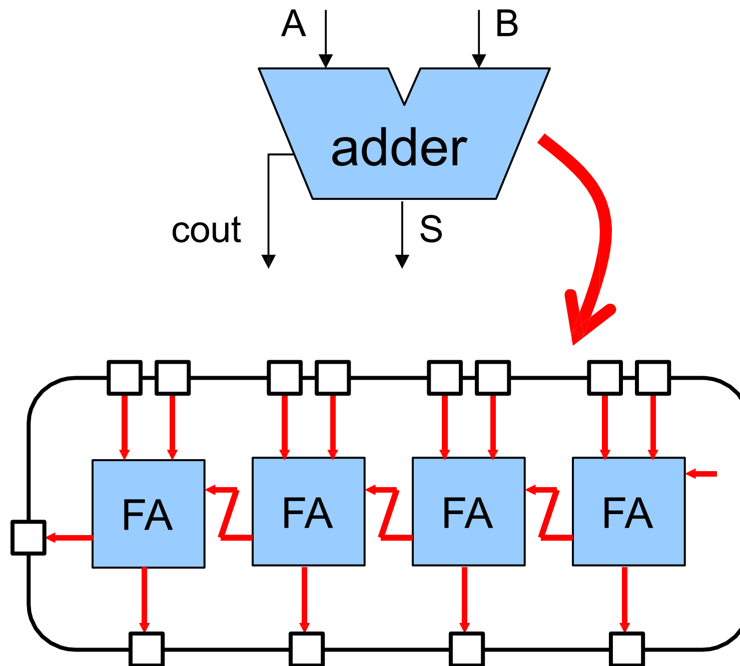

```
module   adder( input   [3:0] A, B,
                 output        cout,
                 output [3:0] S );

  wire c0, c1, c2;
  FA fa0( A[0], B[0], 0,  c0, S[0] );
  FA fa1( A[1], B[1], c0, c1, S[1] );
  FA fa2( A[2], B[2], c1, c2, S[2] );
  FA fa3( A[3], B[3], c2, cout, S[3] );

endmodule
```

Carry Chain

Modelare ierarhică cu Verilog

Un modul poate conține alte module prin instanțierea lor. Asftel, se creează o ierarhie de module

- Modulele sunt conectate prin linii de legătură
- Porturile sunt atașate la aceste linii fie prin poziționare, fie prin numele lor




```
module adder( input   [3:0] A, B,
                output        cout,
                output  [3:0] S );

  wire c0, c1, c2;
  FA fa0( .a(A[0]), .b(B[0]),
          .cin(0), .cout(c0),
          .sum(S[0] );

  FA fa1( .a(A[1]), .b(B[1]),
          ...

endmodule
```

Verilog Basics

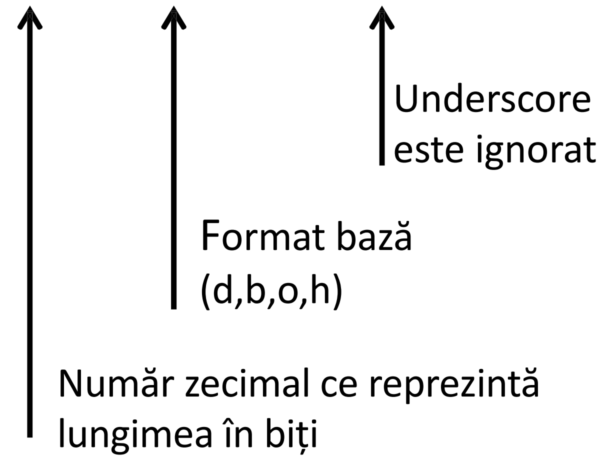
Valori pentru date

0 1

X Z

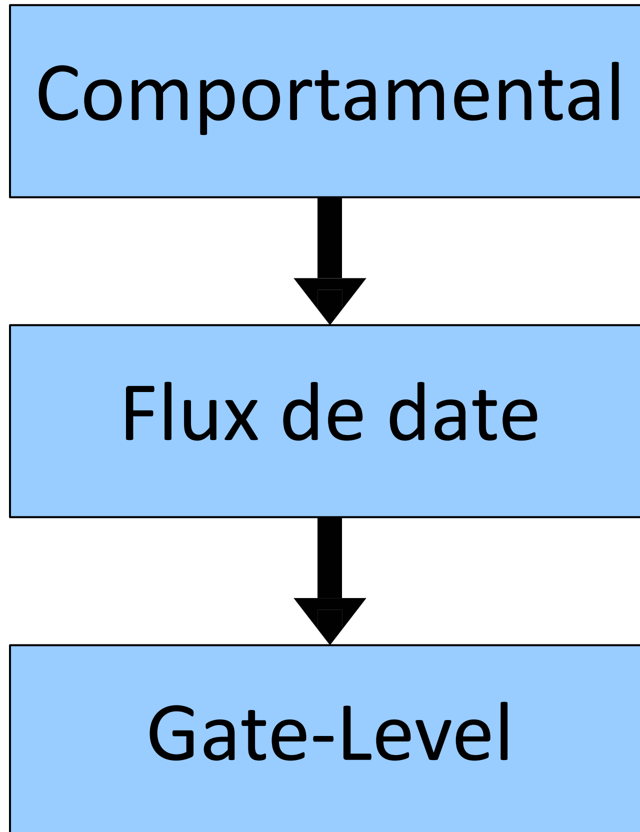
Constante numerice

4'b10_11



32'h8XXX_XXA3

3 Niveluri comune de abstractizare

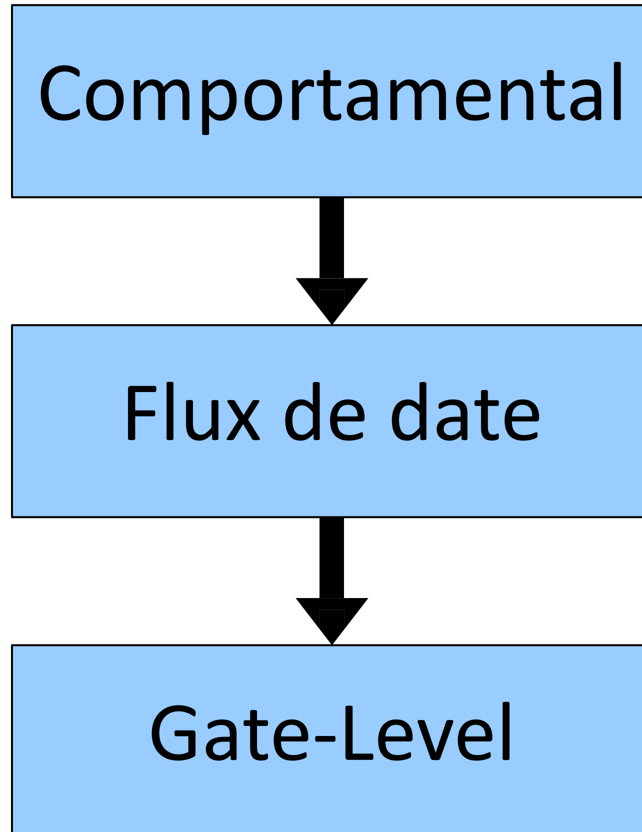


Algoritmul de nivel înalt al modulului este implementat fără să se țină prea mult cont de hardware

Modulul este implementat prin specificarea felului în care datele curg între registre

Modulul este implemetat folosind porți logice (AND, OR, NOT) și interconectările lor

3 Niveluri comune de abstractizare



Proiectanții pot crea modele de nivel scăzut din modelele de nivel înalt, fie manual sau automat

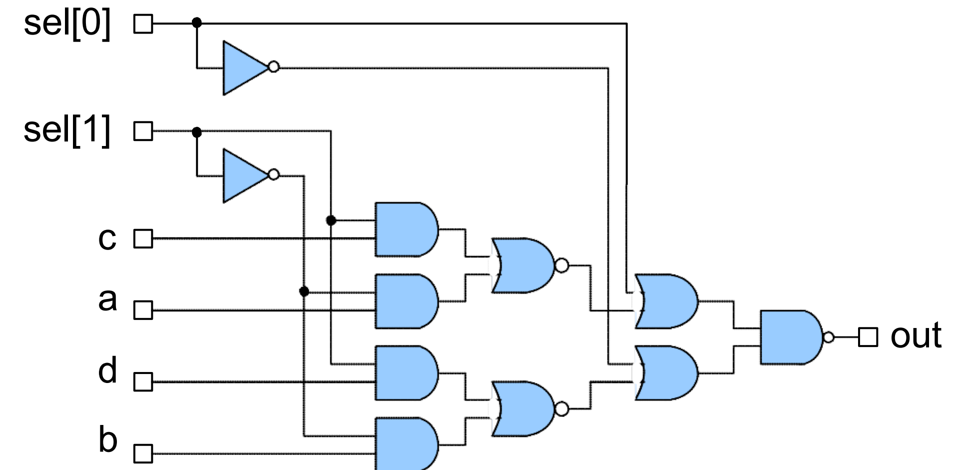
Procesul generării automate a unui model la nivel de porți logice dintr-un nivel comportamental sau de flux de date se numește

Sinteză Logică

Gate-Level : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire [1:0] sel_b;
  not not0( sel_b[0], sel[0] );
  not not1( sel_b[1], sel[1] );

  wire n0, n1, n2,  n3;
  and  and0( n0, c, sel[1]    );
  and  and1(  n1, a, sel_b[1] );
  and  and2(  n2, d, sel[1]    );
  and and3(  n3, b, sel_b[1] );

  wire x0, x1;
  nor nor0( x0,  n0, n1 );
  nor nor1( x1,  n2, n3 );

  wire y0, y1;
  or or0( y0,  x0, sel[0]    );
  or or1( y1,  x1, sel_b[0] );

  nand nand0(  out, y0, y1 );

endmodule
```

Porțile logice de bază sunt declarate ca primitive, deci nu este nevoie să definim module pentru ele



Dataflow : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire out, t0, t1;
  assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

Aceasta este o atribuire continuă deoarece partea dreaptă este evaluată tot timpul și rezultatul este atribuit la orice moment de timp părții stângi a expresiei

Dataflow : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  wire t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  wire out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

O atribuite continuă implicită combină declararea liniei de date cu o atribuite de tip assign, deci este mai succintă

Dataflow : 4-input Mux & Adder

```
// Four input muxltiplexor
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  assign out = ( sel == 0 ) ? a :
               ( sel == 1 ) ? b :
               ( sel == 2 ) ? c :
               ( sel == 3 ) ? d : 1'bx;

endmodule

// Simple four bit adder
module adder( input  [3:0] op1, op2,
              output [3:0] sum );

  assign sum = op1 + op2;

endmodule
```

Exprimarea la nivel de flux de date permite descrieri mai abstracte decât cele la nivel de porți logice în Verilog

Dataflow : Key Points

Modelarea la nivel de flux de date permite proiectantului să se concentreze asupra stării design-ului și asupra felului în care datele circulă în circuit fără a se poticni în detalii de implementare la nivelul porților logice

- **Atribuirile continue sunt folosite pentru a conecta logica combinațională**
- **Este disponibilă o mare varietate de operatori:**

```
Arithmetic:     + - * / % **
Logical:        ! && ||
Relational:     > < >= <=
Equality:       == != === !===
Bitwise:        ~ & | ^ ^~
Reduction:      & ~& | ~| ^ ^~
Shift:          >> << >>> <<<
Concatenation:  { }
Conditional:    ?:
```

Evitați acești operatori pentru că se sintetizează prost!

Dataflow : Key Points

Modelarea la nivel de flux de date permite proiectantului să se concentreze asupra stării design-ului și asupra felului în care datele circulă în circuit fără a se poticni în detalii de implementare la nivelul porților logice

- **Atribuirile continue sunt folosite pentru a conecta logica combinațională**
- **Este disponibilă o mare varietate de operatori:**

```
Arithmetic:      + - * / % **
Logical:         ! && ||
Relational:      > < >= <=
Equality:        == != === !===
Bitwise:         ~ & | ^ ^~
Reduction:       & ~& | ~| ^ ^~
Shift:           >> << >>> <<<
Concatenation:   { }
Conditional:     ?:
```

```
assign signal[3:0]
       = { a, b, 2'b00 }
```

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a or b or c or d or sel)
  begin
    if ( sel == 0 )
      out =  a;
    else if  ( sel  ==  1 )
      out =  b;
    else if  ( sel  ==  2 )
      out =  c;
    else if  ( sel  ==  3 )
      out =  d;
  end

endmodule
```

Un bloc always este un bloc comportamental ce conține expresii ce sunt de obicei evaluate secvențial Codul dintr-un astfel de bloc poate fi foarte abstract (similar cu C) – aici implementăm un MUX cu declarații if/else

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a or b or c or d or sel )
  begin
    if ( sel == 0 )
        out = a;
       else if ( sel == 1 )
        out = b;
      else if  ( sel == 2 )
        out = c;
       else if ( sel == 3 )
        out = d;
  end

endmodule
```

Un bloc always poate include un sensitivity list – dacă oricare din aceste semnale se schimbă, atunci se execută blocul always

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a, b, c, d, sel )
  begin
    if ( sel == 0 )
      out = a;
     else if (sel == 1 )
      out = b;
     else if (sel == 2 )
      out = c;
     else if (sel == 3 )
      out = d;
  end

endmodule
```

În Verilog-2001 putem folosi o virgulă în loc de or

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a, b, c, d, sel )
  begin
    if ( sel == 0 )
      out =   a;
    else if    (  sel   ==   1  )
      out =   b;
    else if    (  sel   ==   2  )
      out =   c;
    else if    (  sel   ==   3  )
      out =   d;

  end
endmodule
```

Ce se întâmplă dacă uităm din greșeală un semnal din listă?

Blocul always nu se va executa dacă doar *d* se schimbă - deci dacă *sel==3* și *d* se schimbă, *out* nu va fi actualizat
Asta va crea discrepanțe între hardware-ul simulat și cel sintetizat - un hardware real nu are sensitivity lists, deci în realitate va funcționa cum trebuie!

Comportamental: 4-input Multiplexer

```
module mux4( input   a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    if ( sel == 0 )
        out = a;
      else if ( sel == 1 )
       out = b;
      else if ( sel == 2 )
       out = c;
      else if ( sel == 3 )
       out = d;
  end

endmodule
```

În Verilog-2001 putem să folosim o exprimare *@(*)* care creează un sensitivity list pentru toate semnalele din acel bloc always

whenever

always @(*)

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

Blocurile always pot conține declarații tip case statements, bucle for și while, chiar și funcții – în acest fel permit modelarea comportamentală de nivel înalt

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

În Verilog, o declarație reg este doar o variabilă – de fiecare dată când vedeți reg trebuie să vă gândiți la o variabilă, NU la un registru hardware!

TOATE atribuirile dintr-un bloc always TREBUIE să aibă ca destinație o variabilă reg!!!

Computer Science & Engineering Department

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

Ce se întâmplă în situația asta?
Hardware-ul sintetizat va
include un latch pentru out?

Computer Science & Engineering Department

Comportamental: 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

Poate! Ce se întâmplă dacă *sel == xx*? Out va fi neinițializat și hardware-ul va trebui să mențină cumva ultima valoare a lui *out*!

Computer Science & Engineering Department

Comportamental: 4-input Multiplexer

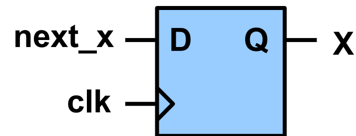
```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      default : out  = 1'bx;
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

Putem repara asta cu o clauză default în case – atunci nu avem nevoie de nici un latch hardware

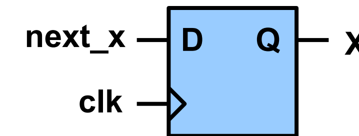
Computer Science & Engineering Department

Atribuiri blocante (=) și non-blocante (<=)

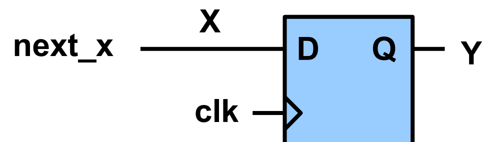
```
always @( posedge  clk )
begin
  x = next_x;
end
```



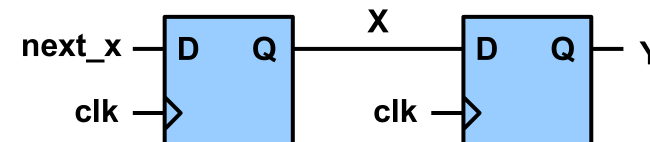

```
always @( posedge clk )
begin
  x <= next_x;
end
```




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




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

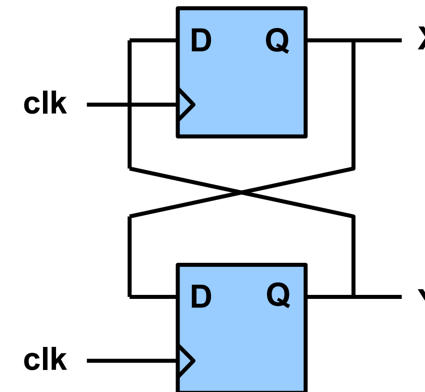


Atribuiri blocante (=) și non-blocante (<=)

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

X	Y

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



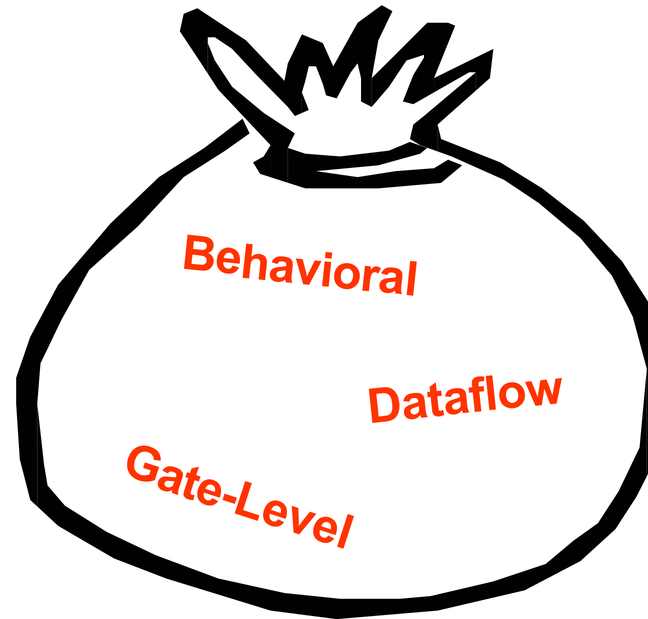
Take Away Point – trebuie să vă întrebați întotdeauna: ”Am nevoie de atribuiri blocante sau non-blocante pentru acest bloc always?”
Nu le puneți la întâmplare!

Atribuiri blocante (=) și non-blocante (<=)

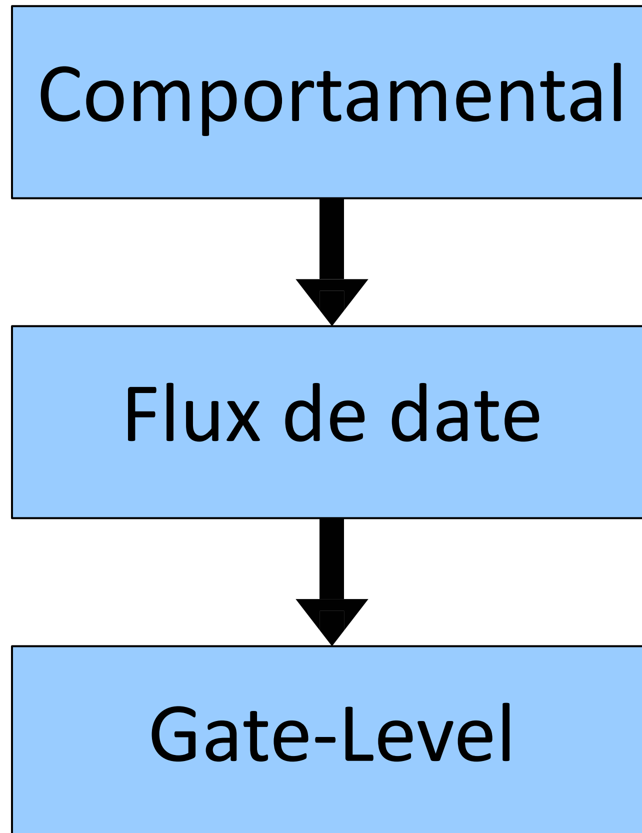
- Dacă vreți să creați logică **secvențială** – always @(posedge clk) + atribuiri **non-blocante**
- Dacă vreți să creați logică **combinațională** – always @(posedge clk) + atribuiri **blocante**
- Încercați să **nu amestecați cele două tipuri de atribuiri** în același block always!
  – Nu este interzis, dar e dificil de prezis ce va face compilatorul

Care abstractizare este cea mai potrivită?

Proiectanții folosesc de obicei o combinație de toate trei! La începutul etapei de design este posibil să folsoească mai mult modele comportamentale. Pe măsură ce design-ul este rafinat, acestea sunt înlocuite de modele data-flow. În cele din urmă, proeictanții folosesc programe automate pentru a sintetiza un model gate-level.



Sinteza Logică



Compilatoarele moderne sunt capabile să sintetizeze din ce în ce mai mult cod comportamental Verilog în cod gate-level

Problema este că este foarte greu de prezis cum va arăta hardware-ul generat în urma sintezei

Acest lucru îngreunează foarte mult procedeul de proiectare a unui circuit și găsirea formei optime a acestuia

Modele parametrizate

```
module mux4 #( parameter width )
             (  input [width-1:0] a, b, c, d
                input   [1:0] sel,
                output [width-1:0] out );

  ...

endmodule

// Specify parameters at instantiation time
mux4 #( .width(32) )
      alu_mux( .a(op1), .b(bypass), .c(32'b0),  .d(32'b1),
               .sel(alu_mux_sel), .out(alu_mux_out) );
```

Parametrii permit configurarea statică a modulelor la instanțiere și pot să mărească foarte mult utilitatea moduleleor respective.

Conectarea modelelor parametrizate

```
module adder #( parameter width )
              ( input  [width-1:0] op1,op2,
                output cout,
                output [width-1:0] sum );

  wire [width:0] carry;
  assign carry[0] = 0;
  assign cout = carry[width];

  genvar i;
  generate
    for ( i = 0; i < width; i = i+1 )
    begin : ripple
      FA fa( op1[i], op2[i],
             carry[i], carry[i+1] );
    end
  endgenerate

endmodule
```

Blocurile generate pot folosi parametrii pentru a instanția un număr variabil de submodule pentru a crea un număr variabil de legături

FPGA Engineer

Just send the signal to both ports.

C++ Developer

Oh no ! Two of my threads tried to access this data at the same time and my output is random !

imgflip.com

Drept concluzie

Limbajele de descriere hardware constituie o parte esențială a proiectării circuitelor digitale

- HDL poate furniza o specificație funcțională executabilă
- HDL permite o flexibilitate mărită în alegrea design-ului potrivit
- HDL încurajează dezvoltarea de programe automate de sinteză
- HDL ajută la micșorarea complexității unui design digital modern

Verilog nu este un limbaj de programare software, deci trebuie să fiți tot timpul conștienți de faptul că programul vostru se va translata într-un circuit hardware real!

Planificați-vă cu atenție ierarhia modulelor, deoarece asta va influența multe alte părți ale design-ului vostru