

# Laborator 3

## Verilog - Introducere

### Obiective:

- Introducerea în limbajul de descriere Hardware Verilog
- Introducerea conceptelor de magistrala și definirea acestora în Verilog
- Introducerea în blocuri continue și procedurale în Verilog
- Introducerea structurilor decizionale în Verilog
- Introducerea structuri repetitive în Verilog
- Înțelegerea conceptului de "Positive Test"

### Verilog Hardware Description Language (HDL):

- Descriere textuală a unei realizări hardware
- Permite proiectarea la diverse nivele de abstractizare: algoritmi versus tranzistori (biți)
- Facilitează verificarea înainte integrării fizice (crearea hardware-ului propriu-zis)
- Unele de sinteză translatează modelele Verilog în realizări hardware - creează schema hardware ce poate fi imprimată ulterior
- Descrierea Verilog a unui hardware este organizată în unul sau mai multe *module*

### Modul Verilog

Un modul Verilog este definit astfel:

- Numele modului, precedat de cuvântul rezervat *module*
- O listă a intrărilor și ieșirilor modului, între paranteze, separate prin virgulă
- Implementarea modului
- Cuvântul rezervat, final, *endmodule*
- Datele de intrare (definite prin cuvântul rezervat *input*) și datele de ieșire (definite prin cuvântul rezervat *output*) formează *porturile* unui modul.\*
  - \* Există situația în care un *port* poate fi și de intrare și de ieșire în același timp, caz în care acesta este definit prin cuvântul rezervat *inout*

Un exemplu de definire a unui modul poate fi observat în codul de mai jos:

```
module nume_modul(input a, input b, output c, output d);
```

```
// Implementarea modulului
endmodule
```

Implementarea unui modul poate cuprinde următoarele tipuri de declarații:

- *Instanțe* de module Verilog (**neacoperite în această lucrare de laborator**)
- Atribuiți continue, introduse prin cuvântul rezervat *assign* - *asignare continuă*
- Blocuri *initial*
- Blocuri *always* - *asignare procedurală*

De ce să folosim *assign* și nu *always* atunci când este posibil?

- *always* se folosește mereu cu semnale sau porturi de output de tipul registri, adică trebuie alocat un spațiu fizic pentru acel registru, în timp ce folosind instrucțiunea *assign* doar atribuim o valoare ieșirii (transmitem informația cu ajutorul unui fir).

## Porturi și semnale în Verilog

### Diferențe

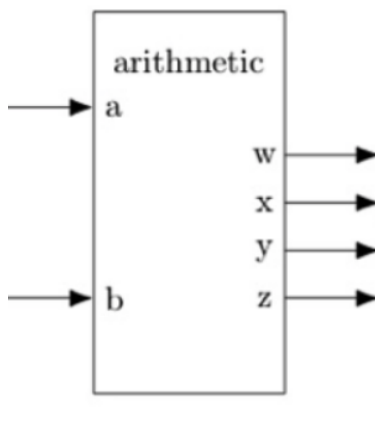
Distincția dintre *port* și *semnal* este dată de folosirea acestuia într-un modul Verilog. Un *semnal* este orice "variabilă" dintr-un modul Verilog, iar un *port* are și o "direcție" (fie *input*, *output* sau *inout*). În consecință, orice *port* este un *semnal*, dar nu orice *semnal* este neapărat și un *port*.

Ținând cont de acest aspect, putem avea *semnale* care sunt folosite intern într-un modul Verilog, dar nu apar în lista de *porturi*.

Exemplu:

```
module modul_exemplu(input a, output b, inout c); // porturi
wire d; // semnale
// Implementarea modulului
endmodule
```

### Exemplu de prim design în Verilog



Creăm un fișier nou **arithmetic.v** în care scriem următorul cod:

```

module arithmetic ( // lista de porturi
                    input a,b, // Linia 1
                    output w,x,y,z // Linia 2
                    );

endmodule

```

Codul nostru va descrie doar un pătrățel care are două intrări a și b și patru ieșiri w, x, y și z și **nu are funcționalitate** (aceasta va fi adăugată ulterior).

### Cum aplicam ce am învățat mai sus:

- **Pătrățel** - când vedem un pătrățel scriem **module** și **endmodule**
- **Nume pătrățel** - după **module** scriem numele pătrățelului, în cazul nostru vom scrie numele **arithmetic**
- **Intrări** - specificăm între paranteze ca **input** toate intrările
- **Ieșiri** - specificăm între paranteze ca **output** toate ieșirile

ATENȚIE la **lista de porturi**, adică tot ce scriem între paranteze. Fiind o listă, toate elementele sunt despărțite prin virgulă, altfel spus am putea scrie **a,b,w,x,y,z** dar mai specificăm și dacă sunt intrări sau ieșiri, ca urmare lista va deveni **input a,b, output w,x,y,z**, iar noi le scriem pe rânduri diferite să arate mai frumos codul și să fie mai ușor de citit. De observat faptul că după **z**, adică ultimul element din lista de porturi, **nu mai apare virgulă**.

După ce descriem acel pătrățel, trebuie să spunem și ce anume face, pentru asta vom introduce funcționalitate în cod astfel :

```

module arithmetic ( // lista de porturi
                    input a,b, // Linia 1
                    output reg w,x,y,z // Linia 2
                    );

    always @ ( a or b ) begin // Linia 3
        w = a + b;
        x = a - b;
        y = a * b;
        z = a / b;
    end // Linia 4

endmodule

```

O primă modalitate pentru a descrie cum funcționează un pătrățel este de a folosi un bloc **always** sau *initial*. Simplu spus blocul *initial* se execută **o singură dată** începând cu momentul 0 al simulării, în timp ce blocul *always* se execută ori de câte ori se schimbă valoarea oricărei variabile din **lista de senzitivitate**, începând cu momentul 0 al simulării și continuând atâta timp cât se mai modifică oricare variabilă din lista de senzitivitate. Sintaxa pentru blocul *always* este

**always @ (lista senzitivitate)**

Elementele din lista de senzitivitate pot fi precizate în 3 moduri:

1. Cu virgulă între elemente : **always @ (a,b)**
2. Cu **or** între elemente : **always @ (a or b)**
3. Cu \* în cazul în care în lista de senzitivitate apar **toate intrările** (oricâte intrări ar fi ) : **always @ (\*)**

Oricare din cele 3 metode de mai sus este valabilă pentru exemplul nostru.

Regulă pentru atribuire în blocuri initial și always:


*Dacă avem o variabilă la care îi atribuim o valoare în interiorul unui bloc *initial* sau *always* este o variabilă de tip *reg*.*


Pentru că le atribuim valori în interiorul unui bloc *always*, variabilele de **w**, **x**, **y** și **z** sunt declarate ca fiind de tip *reg*.

Există două tipuri de variabile importante în Verilog *wire* și *reg*. Pentru *wire*, adică **fire**, avem doar o conexiune prin care se transmit semnale, altfel spus, firul acesta nu ține minte absolut nimic, **nu are memorie**, în timp ce *reg*, adică **registru**, are **capacitatea de a memora o**

**valoare.** Orice variabilă *input* sau *output* declarată în lista de porturi a unui modul are tipul predefinit *wire* (adică nu mai trebuie să specificăm noi că este de tip *wire*), iar în cazul în care dorim să avem o variabilă cu tipul *reg* acest lucru trebuie specificat scriind *input reg* sau *output reg*.

După *always @ (a or b)* la linia 3 în codul de mai sus apare *begin* , iar la linia 4 apare *end* care sunt același lucru ca acoladele { și } din limbajul C. Cu alte cuvinte, *begin* și *end* delimitează blocul de instrucțiuni care se va executa atunci când un element din lista de senzitivități a instrucțiunii *always* își modifică valoarea. Acel bloc de instrucțiuni se va executa în mod secvențial, adică instrucțiunile se vor executa pe rând, **una după alta**.

După scrierea codului se va salva fișierul dând click pe butonul **Save**  sau utilizând combinația de taste **Ctrl+S** (Atunci când apăsăm Ctrl+S trebui să fie selectată fereastra cu codul pe care dorim să îl salvăm. Important este să dispară acea steluță care ne indică faptul că nu a fost salvat codul).

**După salvare** vom compila fișierul dând click pe butonul **Compile All** 

Acesta a fost **design-ul** : un pătrățel care face ceva.

## Magistrale

Se poate observa că în modulul aritmetic, intrările a și b sunt intrări pe un bit (pot lua doar valorile 1 și 0). În cazul în care vrem să facem operații cu numere mai mari (care sunt reprezentate pe un număr mai mare de biți), nu este necesar să declarăm mai multe intrări/iesiri. Putem face acest lucru folosind magistrale. O magistrală este formată dintr-un grup de fire.

În Verilog, o magistrală mai este denumită și **vector**. Aceasta poate fi reprezentată folosind sintaxa:

*input [MSb:LSb] <nume\_variabila>* sau *output [MSb:LSb] <nume\_variabila>*, sau *output reg[MSb:LSb] <nume\_variabila>*, sau *wire[MSb:LSb] <nume\_variabila>*, sau *reg[MSb:LSb]<nume\_variabila>*

Unde MSb = Most Significant bit (cel mai semnificativ bit == cel mai din stânga bit) și LSb = Least Significant bit (cel mai puțin semnificativ bit == cel mai din dreapta bit, de obicei 0)

Exemplu:

```

input [31:0] d0,
input [31:0] d1,
input s,
output [31:0] o
);
wire [31:0] s_v;

```

**Regula pentru a determina de cati biți ai nevoie pentru a scrie un numar:**

**$2^n < \text{valoare maxima care poate fi atribuita portului/semnalului} < 2^{n+1}$**

Exemplu: Fie numărul n care poate lua valori in [0; 220]

$2^7 \leq 220 < 2^8$

**=> input [7:0] n**

Exercitiu: Modificati modulul aritmetic astfel incat a sa poata lua valori cuprinse in [0;1025] si b in [0;226]. Atentie calculati pe cati biți vor fi iesirile.

## Structuri decizionale

In verilog, ca si in orice alt limbaj de programare putem folosi urmatoarele structuri decizionale:

- IF

```

if ( conditie) begin
//cod pentru ramura de true
end
else
begin
//cod pentru ramura de false
end

```

- CASE

```

case ( variabila)
valoare1: //cod pentru acest caz
valoare2: // cod pentru valoarea2 a variabilei
valoare3, valoare4: //cod pentru valoarea3 sau valoarea4 a variabilei
valoare5: begin
/* cod pentru valoarea5 a variabilei */
end

```

```
default: // cod in cazul in care variabila nu are nicio valoare descrisa  
mai sus  
endcase
```

- operator tertial se utilizeaza in preponderent cu **assign**

**conditie?valoare\_ramura\_true:valoare\_ramura\_false**

Exemplu: Utilizand cele trei variante precizate mai sus se va rezolva problema urmatoare:

Sa spunem ca exista un semna de intrarel x, care poate lua valoarea 1, 2 sau 3. Semnalul de iesire y va lua valori dupa urmatoarele reguli:

- daca x va fi 2, y va fi 3,
- daca x va fi 3, y va fi 2.

**Varianta cu if:**

```
module conditie( input [1:0] x,  
output reg [1:0] y ) ;  
always@(*) begin // *- codul se va executa oricand apare vreo modificare  
if( x==2) begin  
y=3;  
end  
else  
if( x==3) begin  
y=2;  
end  
end  
endmodule
```

**Varianta cu case:**

```
module conditie( input [1:0] x,  
output reg [1:0] y ) ;  
always@(*) begin // *- codul se va executa oricand apare vreo modificare  
case(x)  
2: y=3;  
3: y=2;  
endcase  
end  
endmodule
```

**Varianta cu operator terial:**

```
module conditie( input [1:0] x,  
output [1:0] y ) ;  
assign y=(x==2)?3:2;  
endmodule
```

**Ce problema puteti gasi in codul de mai sus?**

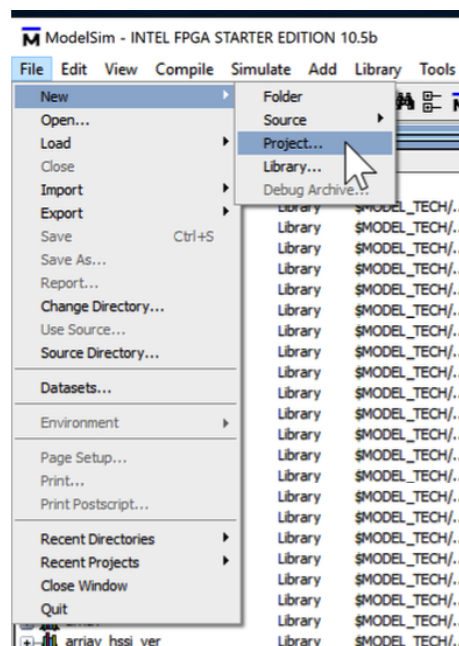
## Structuri repetitive

După cum probabil deja știți, structurile repetitive ne ușurează munca. În loc să afișăm de 10 ori un mesaj, cu ajutorul unei structuri repetitive care se repetă de 10 ori, vom afișa un mesaj.

În limbajul Verilog avem mai multe structuri repetitive, dar în acest laborator vom exemplifica doar 3 dintre ele: **for**, **while** și **repeat** pentru a ne obișnui cu crearea de fișiere în ModelSim, salvarea, compilarea și simularea acestora.

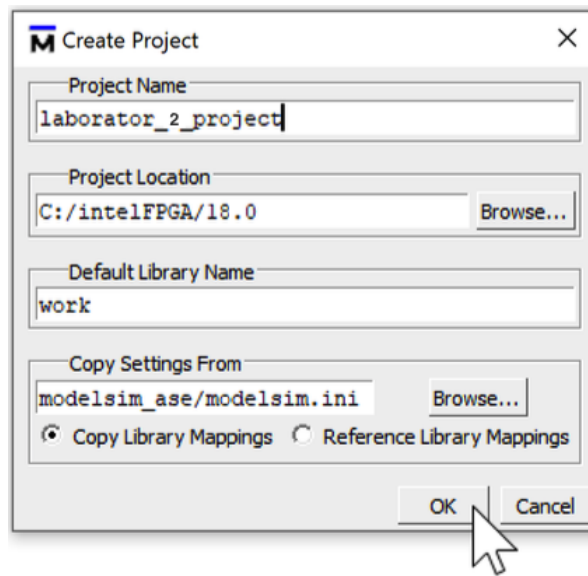
Vom deschide programul ModelSim și vom crea un nou proiect accesând

**File → New → Project...**

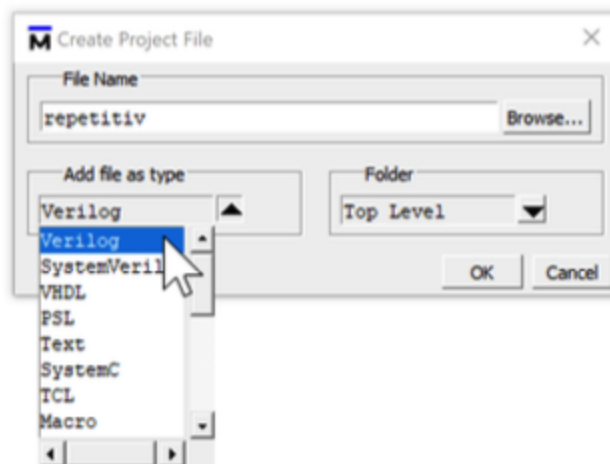
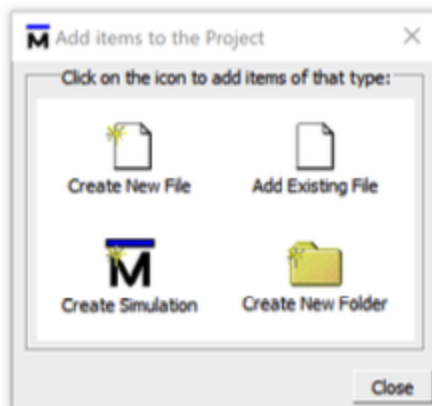


Vom denumi proiectul **laborator\_2\_project** și se va da click pe OK.





Se va adăuga un fișier în proiect utilizând meniul nou apărut **Add items to the project** și dând click pe **Create New File** . Va apărea o nouă fereastră **Create Project File** unde veți completa cu numele fișierului, în cazul nostru **repetitiv** și (**FOARTE IMPORTANT!!!**) veți selecta tipul fișierului ca fiind **Verilog** .



După putem da click pe **OK** și să dăm **Close** la fereastra **Add items to the project**. Dăm dublu click pe fișierul nou creat care apare în partea stângă și se va deschide o fereastră în care putem edita codul.

```

module repetitiv ();
    integer i; // Linia 1
    initial begin // Linia 2
        $display("FOR LOOP"); // Linia 3
        for (i=0;i<10;i=i+1) begin // Linia 4
            $display("i=%d",i); // Linia 5
        end

        i=0; // Linia 6
        $display("WHILE LOOP");
        while (i<10) begin
            $display("i=%d",i);
            i=i+1;
        end

        i=0;
        $display("REPEAT LOOP");
        repeat (10) begin
            $display("i=%0d",i); // Linia 7
            i=i+1; // Linia 8
        end
    end
endmodule

```

La linia 1 observăm o **declarație de variabilă**, variabilă care se numește **i** (pentru ușurință, Verilog mai definește alte tipuri de variabile des utilizate, cum ar fi **integer**, care este de fapt echivalent cu **reg[31:0]**), iar la linia 2 un bloc initial fără de care nu se poate nimic din punct de vedere funcțional în limbajul Verilog.

La linia 4, în cadrul instrucțiunii for observăm că scrie **i=i+1** în loc să scrie mai simplu **i++** din cauză că în limbajul Verilog nu există operatorii **++** sau **--**, deci se vor utiliza **i=i+1** sau **i=i-1**.

La linia 6 se reinițializează variabila **i** pentru a ne asigura că în cadrul următoarei structuri repetitive variabila **i** va porni de la valoarea 0.

La linia 8 apare  $i=i+1$  doar pentru că dorim să afișăm valori diferite pentru variabila  $i$ , nefiind necesară această instrucțiune ca în cazul lui while sau for care sunt bucle cu condiții de oprire. Structura repetitivă repeat se repetă de câte ori este specificat între paranteze, în cazul nostru, de 10 ori.

Funcția `$display` este o funcție care tipărește mesaje. Practic, la linia 3 dorim să afișăm mesajul „FOR LOOP”, iar la liniile 5 și 7 vom dori să afișăm variabila  $i$  care, în interiorul mesajului va fi pusă unde apare `%d`.

După scrierea codului se va salva fișierul dând click pe butonul Save sau utilizând combinația de taste `Ctrl+S`.

Se verifică salvarea fișierului uitându-ne dacă mai apare steluța în dreptul numelui fișierului sau nu.

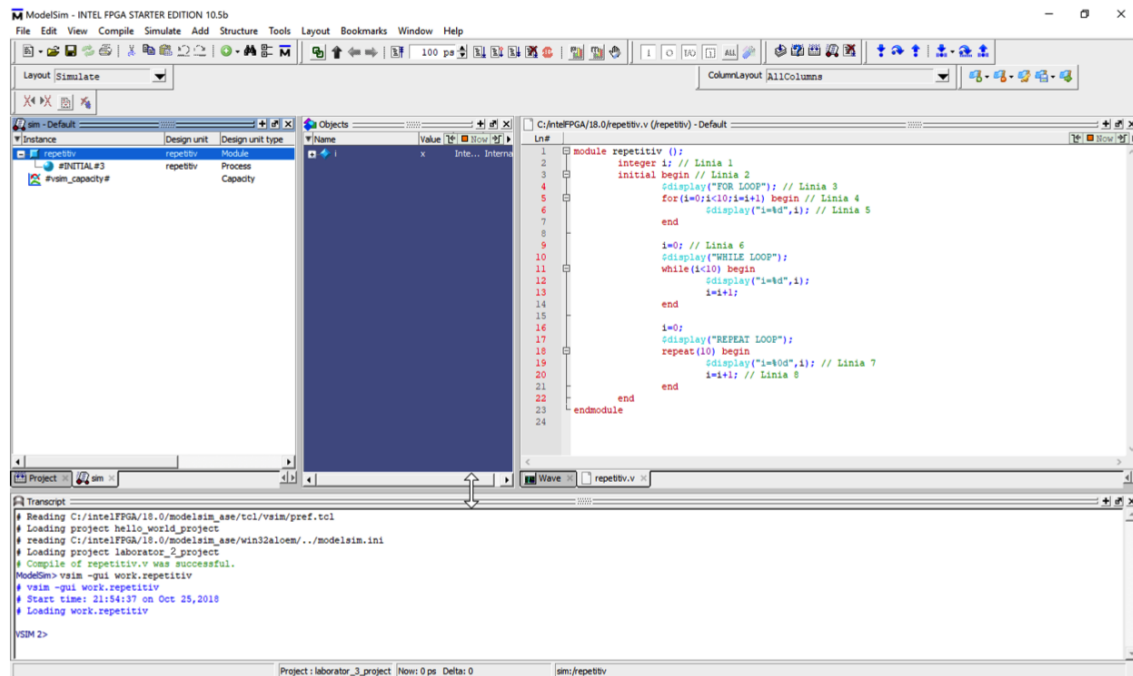
După salvare vom compila fișierul dând click pe butonul Compile All

După compilare dacă rezultă erori le corectăm, salvăm fișierul modificat și îl compilăm încă o dată. Dacă nu avem erori, simulăm dând click pe butonul Simulate

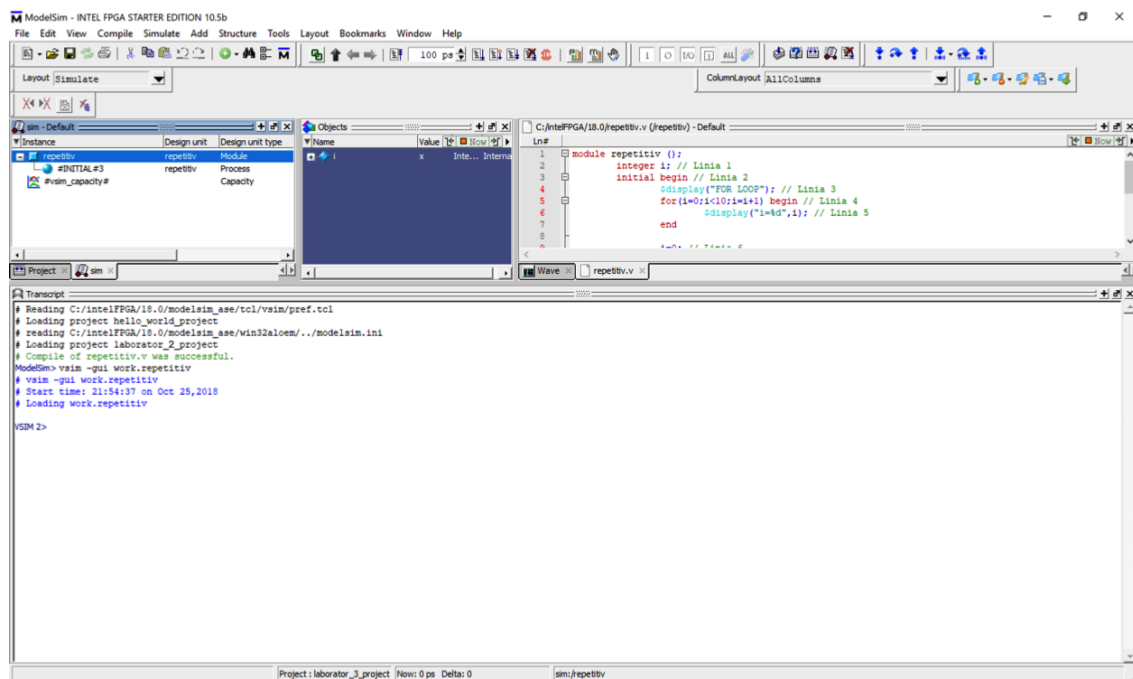
Va apărea o fereastră Start Simulation unde se va da click pe simbolul + din stânga folderului work pentru deschiderea conținutului acestuia (ca să vedem fișierele din el). După se va selecta fișierul repetitiv și se va da click pe OK.

După “transformarea” ModelSim-ului (adică după apariția noilor ferestre sim și Object) ne asigurăm că vedem fereastra Transcript. Dacă nu o vedem, o putem vedea accesând meniul View și selectând View → Transcript.

Dacă vedem fereastra Transcript o putem face mai mare poziționând mouse-ul pe bara ca în imagine și ținând apăsat click stânga tragem de bară în sus.

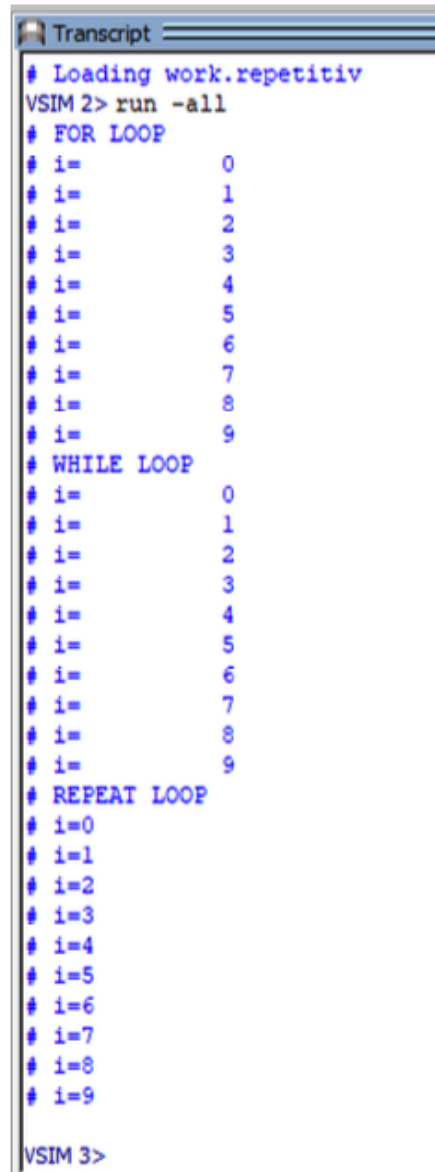


Astfel vom vedea mai bine fereastra Transcript unde vor apărea toate mesajele tipărite de funcția \$display .



După vom rula programul click pe butonul **Run -All**

În fereastra Transcript vor apărea toate mesajele tipărite.



```
# Loading work.repetitiv
VSIM 2> run -all
# FOR LOOP
# i=      0
# i=      1
# i=      2
# i=      3
# i=      4
# i=      5
# i=      6
# i=      7
# i=      8
# i=      9
# WHILE LOOP
# i=      0
# i=      1
# i=      2
# i=      3
# i=      4
# i=      5
# i=      6
# i=      7
# i=      8
# i=      9
# REPEAT LOOP
# i=0
# i=1
# i=2
# i=3
# i=4
# i=5
# i=6
# i=7
# i=8
# i=9
VSIM 3>
```

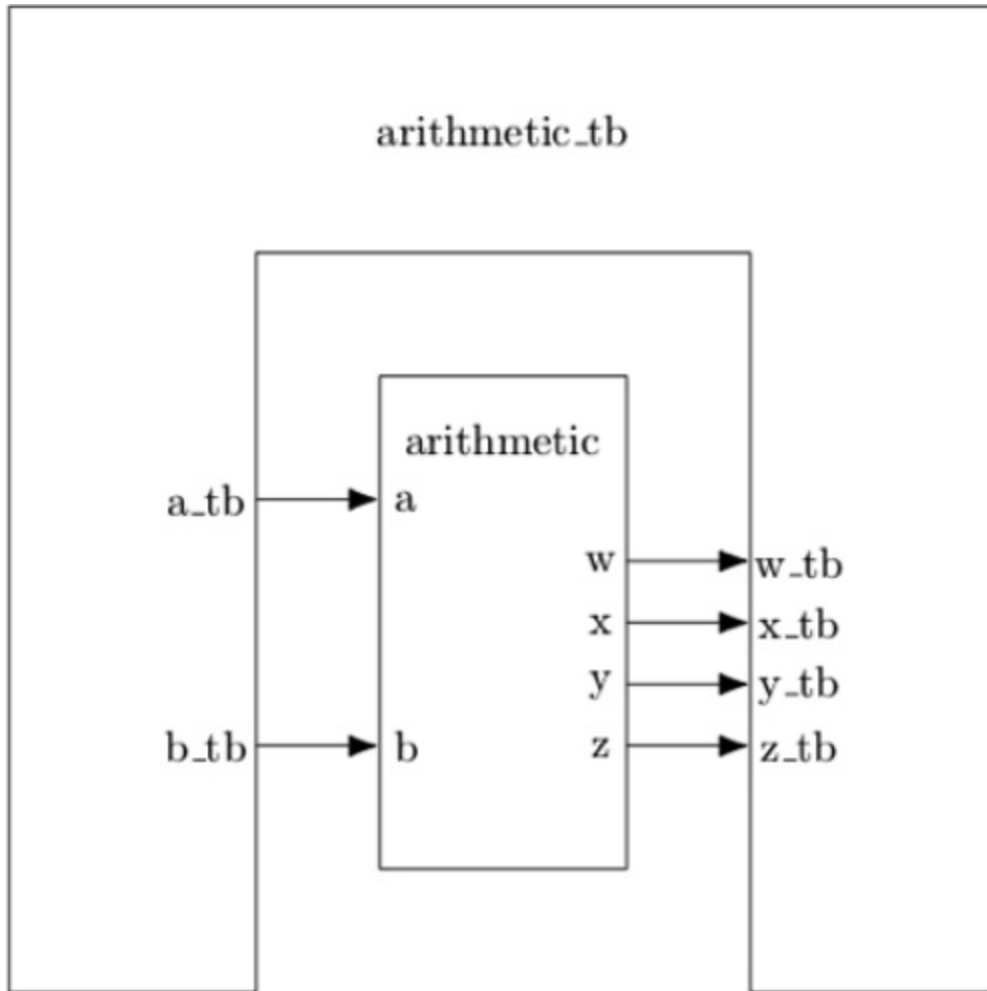
A se observa în mod special efectul liniei 7 față de efectul liniei 5 din cod. Introducerea lui %0d în loc de %d de fapt va însemna cu 0 spații.

## Simbolistica specifică pentru reprezentarea numerelor în diferite baze:

%d or %D	Decimal format
%b or %B	Binary format
%h or %H	Hexadecimal format
%o or %O	Octal format
%c or %C	ASCII character format
%v or %V	Net signal strength
%m or %M	Hierarchical name
%s or %S	As a string
%t or %T	Current time format

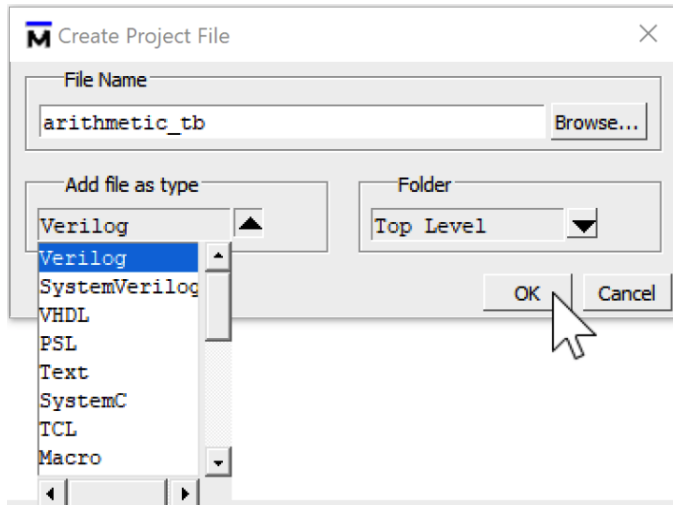
## Primul testbench

Testbench-ul este în formă de “potcoavă” și înconjoară DUT-ul. Dacă ni se dă un design trebuie să putem să scriem automat testbench-ul. Cum? Urmează în continuare.



În primul rând creăm un nou fișier dând click dreapta în fereastra **Project** și apoi accesând **Add to Project** → **New File...** . Acest fișier va avea același nume ca design-ul, doar că îi mai adăugăm `_tb` pentru a ști că este vorba despre testbench-ul pentru design-ul `arithmetic`





Vom scrie codul de mai jos pentru acest testbench:

```
module arithmetic_tb;
    reg a_tb, b_tb; // Linia 1
    wire w_tb, x_tb, y_tb, z_tb; // Linia 2

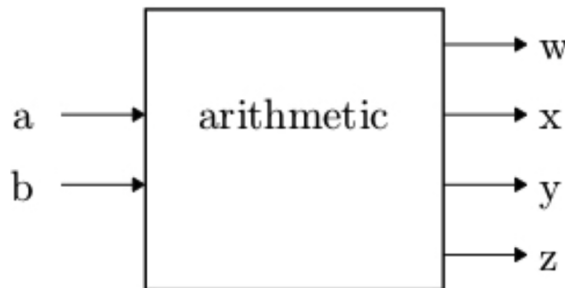
    initial begin // Monitor
        $monitor("Time=%d a=%b b=%b w=%b x=%b y=%b z=%b",
            $time, a_tb, b_tb, w_tb, x_tb, y_tb, z_tb);
    end

    initial begin // Generare stimuli
        a_tb = 1;
        b_tb = 1;
        #1 a_tb = 0;
        #1 b_tb = 0;
        #1;
    end

    arithmetic aRandomName ( // Instantiere DUT
        .a(a_tb),
        .b(b_tb),
        .w(w_tb),
        .x(x_tb),
        .y(y_tb),
        .z(z_tb)
    );

endmodule
```

Uitându-ne strict la pătrățelul de mai jos, putem scrie cu ușurință codul de mai sus pentru că are același nume, doar că mai are **\_tb** , **nu are listă de porturi** pentru că este testbench, **intrările** în pătrățel vor fi **reg** pentru că la etapa de **generare de stimuli** li se atribuie valori în cadrul unui bloc **initial** , **ieșirile** vor fi declarate ca **wire** .

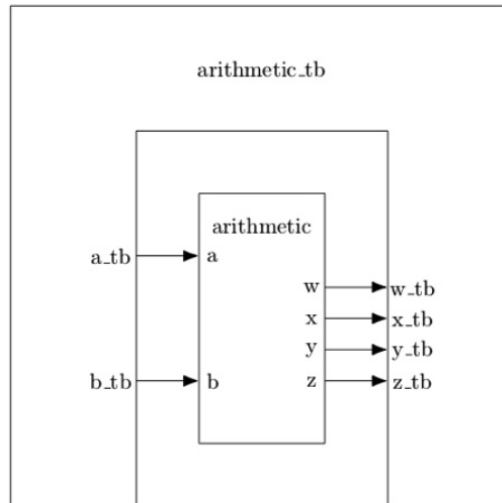


După aceste rânduri de cod, vedem că avem 3 mari etape într-un testbench: **monitor**, **generare de stimul** și **instanțierea DUT-ului**.


În etapa **monitor** avem pur și simplu funcția **\$monitor** care nu face altceva decât să afișeze un mesaj ori de câte ori se modifică oricare variabilă din interiorul său. Funcția **\$display** afișează un mesaj o singură dată, pe când funcția **\$monitor** afișează mesajul ori de câte ori se modifică o variabilă din interiorul său, exact ca diferența dintre **initial** și **always** . În această etapă este important să afișăm 3 lucruri:  **timpul** curent al simulării ( folosind funcția sistem **\$time** ), **intrările** și **ieșirile** din DUT.


La etapa de **generare de stimuli** important este să generăm cazuri de test pentru DUT astfel încât să vedem ce ieșiri scoate ca răspuns la anumite intrări. Timpul de simulare este 0 la orice început de bloc inițial, iar noi suntem responsabili cu înaintarea sa. Ca să modificăm timpul, care dacă nu facem asta va rămâne pe loc, îi punem întârzieri **#n** unde **n** este numărul de unități de timp care vor fi așteptate până să se execute instrucțiunile care urmează celui **#n** . Vom detalia în laboratorul următor mai bine aceste aspecte.

În etapa de **instanțiere a DUT-ului** este foarte important să folosim numele DUT-ului ca să spunem pe cine instanțiem, apoi îi dăm un nume instanței, în cazul nostru **aRandomName** și apoi urmează lista de conexiuni de porturi.



În figura de mai sus observăm că la portul **a** din DUT îi corespunde portul **a\_tb** din testbench, iar ca să arătăm acest lucru scriem în lista de conexiuni de porturi **.a(a\_tb)** ca să arătăm că la portul **a** din DUT-ul pe care îl instanțiem îi conectăm semnalul **a\_tb** din testbench-ul nostru.

**După scrierea codului** se va salva fișierul dând click pe butonul **Save**  sau utilizând combinația de taste Ctrl+S. Se verifică salvarea fișierului uitându-ne dacă mai apare steluța în dreptul numelui fișierului sau nu.

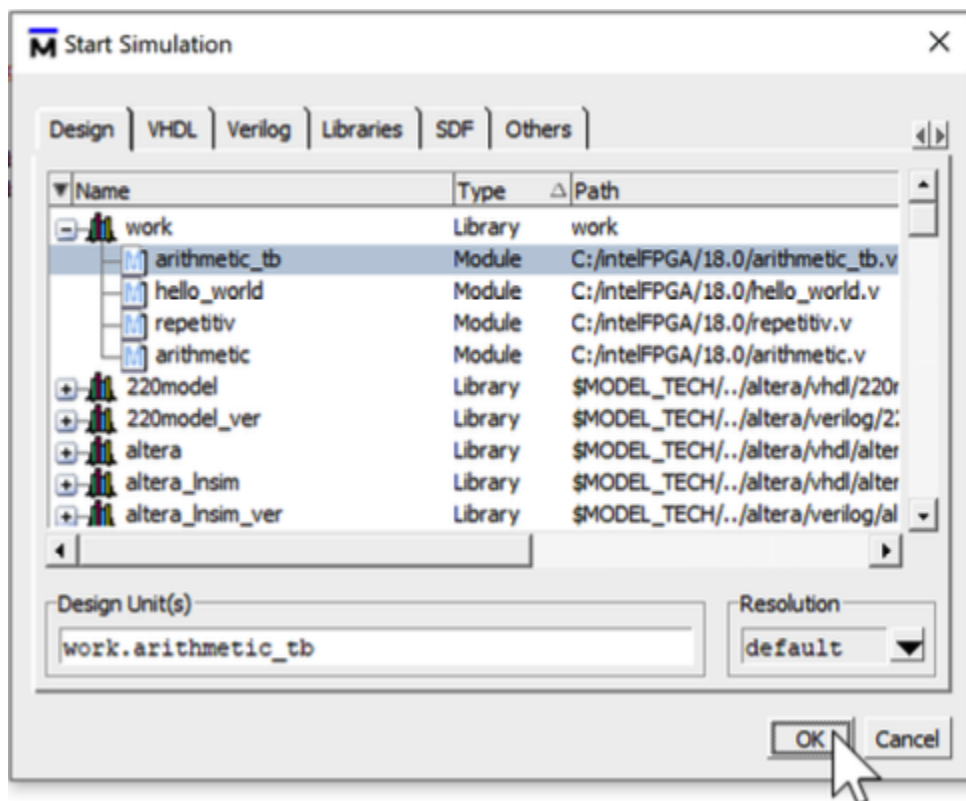
**După salvare** vom compila fișierul dând click pe butonul **Compile All** .

**După compilare** dacă rezultă erori le corectăm, **salvăm** fișierul modificat și în **compilăm** încă o dată.

Dacă nu avem erori, simulăm dând click pe butonul **Simulate** .

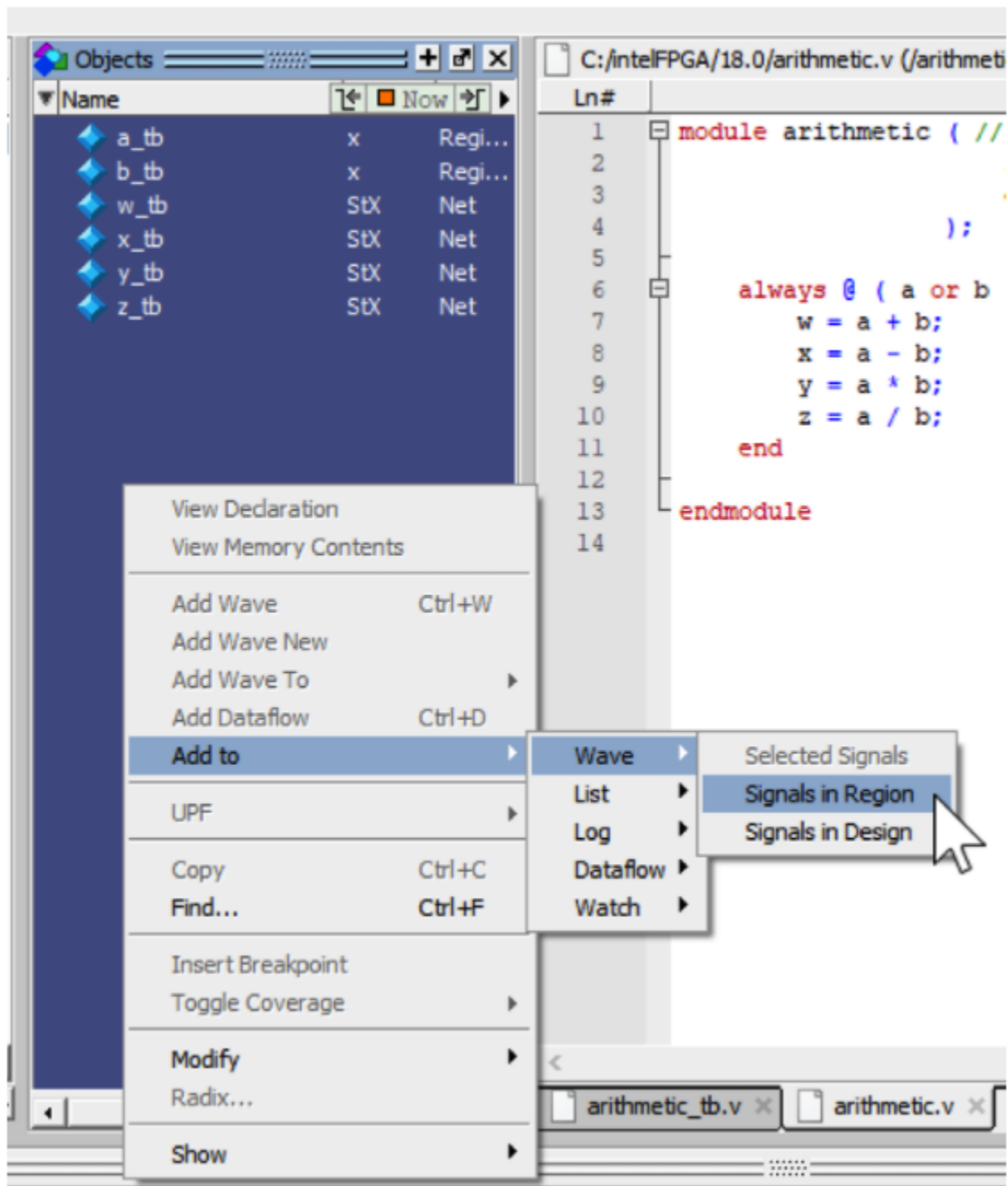
Va apărea o fereastră **Start Simulation** unde se va da click pe simbolul + din stânga folderului **work** pentru deschiderea conținutului acestuia ( ca să vedem fișierele din el) . După se va selecta fișierul **arithmetic\_tb** și se va da click pe **OK**.


**ATENȚIE!!!** Nu se simulează niciodată fișiere de design, doar fișiere de testbench.



Trebuie să vedem fereastra albastră cu numele **Objects**, dacă nu apare fereastra **Objects** mergem la meniul **View** și dăm click pe **View** → **Objects**. Avem aici două variante. În fereastra **Objects** dăm click dreapta în zona albastră (dăm click dreapta în gol, nu pe semnalele de acolo) și mergem pe calea **Add to** → **Wave** → **Signals in Region**

Altă variantă este să dăm un click în fereastra **Objects** să fie selectată acea fereastră, după care apăsăm combinația de taste **Ctrl+A** ca să selectăm toate semnalele și apoi apăsăm combinația de tasta **Ctrl+W** pentru a adăuga semnalele selectate în fereastra **Wave**. Important este că trebuie să ne apară semnalele declarate în testbench în fereastra **Wave**. Dacă nu vedem fereastra **Wave** mergem la meniul **View** și dăm click pe **View** → **Wave**.

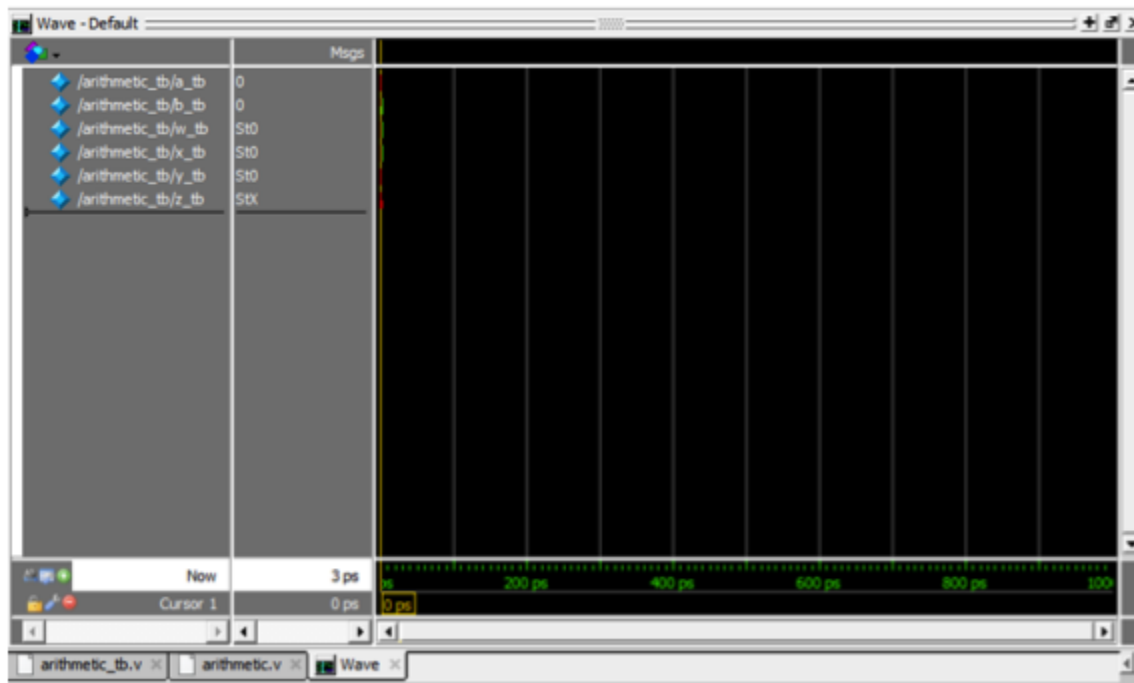


Numai după ce adăugăm semnalele în **Wave** atunci vom rula programul click pe butonul **Run -All** 

În fereastra **Transcript** vom vedea clar că a mers simularea și vedem mesajele tipărite de 3 ori, pentru că am introdus 3 întârzieri la etapa de generare de stimuli.

```
Transcript
# Start time: 16:33:00 on Oct 27, 2016
# Loading work.arithmetic_tb
# Loading work.arithmetic
add wave sim:/arithmetic_tb/*
VSIM 3> run -all
# Time=                0 a=1 b=1 w=0 x=0 y=1 z=1
# Time=                1 a=0 b=1 w=1 x=1 y=0 z=0
# Time=                2 a=0 b=0 w=0 x=0 y=0 z=x
VSIM 4>
```

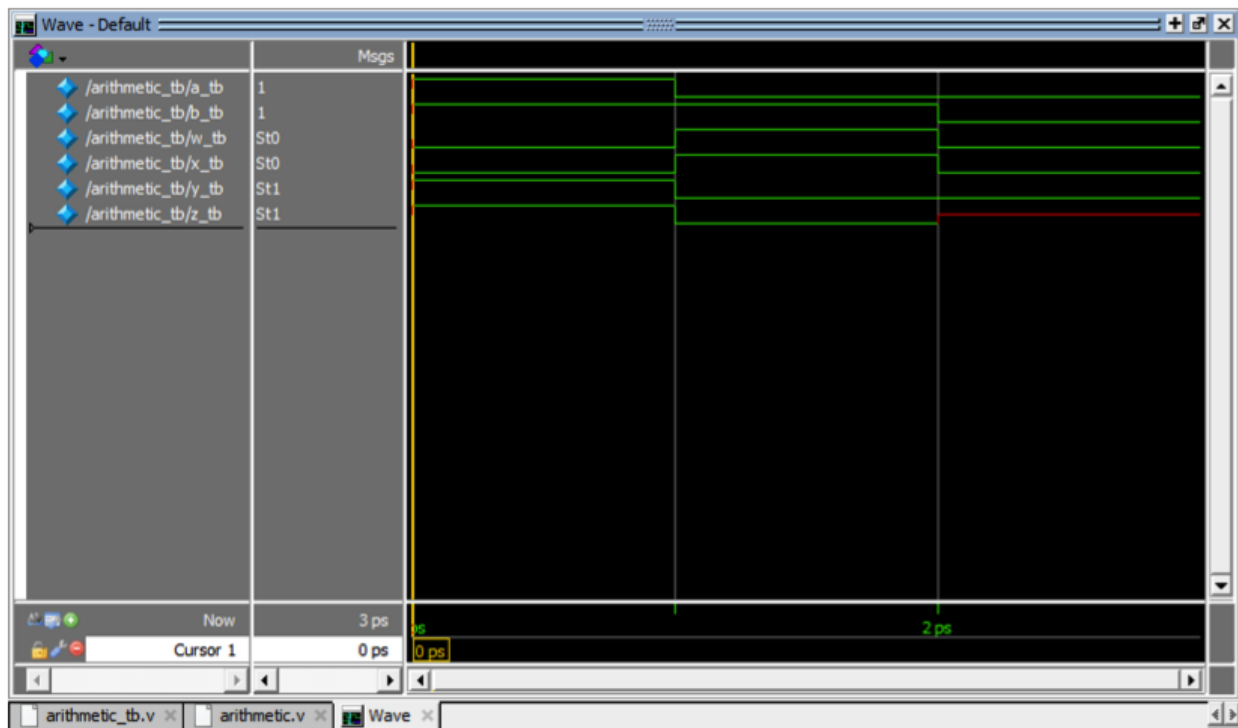
În cazul ferestrei **Wave** vedem ceva și nu prea. Semnalele sunt extrem de scurte și imaginea trebuie mărită ca să le vedem corespunzător.



Pentru a vizualiza mai ușor unde le dă click oriunde în zona neagră din fereastra **Wave** și dă click dreapta selectând **Zoom Full** sau dacă selectăm fereastra **Wave** putem simplu apăsa tasta **F**.

Rezultatul simulării este un **waveform**, adică niște semnale care sunt sus (adică la 1 logic) sau sunt jos (adică la 0 logic), altfel spus, un mod de vizualizare a simulării

care afișează în fereastra **Wave** semnalele pe care și funcția **\$monitor** le afișează în fereastra **Transcript**.



Astfel vom putea vedea dacă la anumite semnale de intrare le corespunde o anumită ieșire sau nu. Desigur, ne putem uita doar la mesajele afișate de funcția **\$monitor** în fereastra **Transcript**, dar totul va fi mai interesant vizualizând semnalele în fereastra **Wave**.

CONGRATS !!!

Tocmai ai rulat pentru prima dată cu succes un testbench în care ai simulat un design!

**Positive Test**

Testarea pozitivă este acea testare care încearcă să arate că un anumit modul al unei aplicații face ceea ce a fost proiectat să facă.

Exemplu aplicat pentru modulul arithmetice:

În fișierul arithmetic\_tb porțiunea următoare de cod trebuie modificată pentru a putea aplica tehnica de testare pozitivă:

```
initial begin // Generare stimuli
    a_tb = 1;
    b_tb = 1;
    #1 a_tb = 0;
    #1 b_tb = 0;
    #1;
end
```

Astfel:

```
initial begin
    a_tb = 4;
    b_tb = 3;
    if ( w_tb == 7 && x_tb == 1 && y_tb == 12 && z_tb == 1 )
        $display(" test passed");
    else
        $display(" test failed");
    end
```