

Laboratorul 2 - Limbajul Verilog: Circuite combinaționale

În laboratorul anterior au fost prezentate elementele Verilog folosite pentru descrierea structurală a circuitelor logice. Acestea constau în [wires](#), în [instanțierea de primitive](#) și în [instanțierea modulelor](#) declarate de utilizator. Folosirea descrierii structurale devine însă complicată și greu de înțeles pentru circuite cu mai mult de câteva componente.

Deși partiționarea circuitelor în cadrul unei arhitecturii duce la blocuri componente mai simple, acestea sunt rareori descrise la nivel de porți logice. Este folosită în loc o descriere comportamentală a circuitului, care permite o implementare mai rapidă. În plus, această metodă are și avantajul că este mai ușor de înțeles și de modificat.

Laboratorul curent va prezenta elementele Verilog folosite pentru descrierea comportamentală a circuitelor logice. Aceasta descrie **ce face** circuitul și nu **cum** va fi acesta implementat. Se folosesc construcții de nivel înalt, care pot defini direct funcția booleană a unui circuit sau chiar algoritmul care calculează ieșirile circuitului. Atât circuitele combinaționale, cât și cele secvențiale pot fi descrise comportamental. Acest laborator se va axa pe descrierea circuitelor combinaționale, urmând ca descrierea celor secvențiale să fie făcută în laboratoarele următoare.

Inițial, limbajul Verilog a fost conceput numai pentru simularea circuitelor logice. Din acest considerent el oferă o paletă largă de modalități de descriere a comportamentului unui circuit (orice instrucțiune poate fi relativ ușor simulată în software). Nu toate facilitățile oferite de Verilog pot fi însă implementate direct în hardware. Suportul pentru sintetizare (compilarea unei surse într-o listă de porți sau într-o mască de circuit integrat) a fost adăugat mai târziu în limbaj. De aceea, este nevoie de atenție la descrierea unui circuit care se vrea a fi sintetizat, pentru a nu folosi construcții care sunt ne-sintetizabile.

În cadrul laboratorului, scopul este întotdeauna de a obține un circuit sintetizabil, care poate fi apoi testat pe placa FPGA.

Assign

Pentru descrierea directă a unei funcții booleene, Verilog oferă o instrucțiune numită **atribuire continuă**. Aceasta folosește cuvântul cheie `assign` și "atribuie" direct, unei variabile de tip *wire* (nu de tip *reg*!), valoarea expresiei aflată în partea dreaptă a semnului egal.

Exemplu atribuire continua

```
module exemplu_assign(output out, input a, b, c);  
  
    assign out = (a && !b) || c;
```

endmodule

Această instrucțiune diferă însă de atribuiri normale oferite de limbajele de programare (C, C++, Java etc.) prin faptul că este executată continuu, valoarea semnalului asignat (partea stângă a semnului egal) fiind recalculată **de fiecare dată** când se modifică una dintre variabilele din partea dreaptă. Acest lucru contrastează cu funcționarea unei atribuiri obișnuite care modifică variabila asignată o singură dată.

Este o eroare să folosiți aceeași variabilă destinație pentru mai multe atribuiri continue. Ele vor încerca simultan să modifice variabila, lucru care nu este posibil în hardware și va duce la rezultate imprevizibile în simulare.

În partea stângă a unei *atribuiri continue* se poate afla orice variabilă declarată de tip wire sau orice ieșire a modulului care nu are altă declarație (ex. *reg*). Expresiile din partea dreaptă pot fi formate din orice variabile sau porturi de intrare și de ieșire și orice operatori suportați de Verilog.

Pentru a descrie un circuit combinațional folosind atribuiri continue evitați să creați bucle între semnale. Buclele pot duce la sintetizarea de elemente de memorare (eng. *latch*), sau chiar pot face circuitul nesintetizabil.

Se poate observa că o *atribuire continuă* este mult mai ușor de scris, de înțeles și de modificat decât o descriere echivalentă bazată pe instanțierea de primitive. Circuitul descris de o atribuire continuă poate fi însă relativ ușor sintetizat ca o serie de porți logice care implementează expresia dorită, unii operatori având o corespondență directă cu o poartă logică (ex. && - ȘI, || - SAU etc.).

Constante

Pentru specificarea valorilor întregi este folosită următoarea sintaxă:

```
[size] ['radix'] constant_value
```

- numerele conțin doar caracterele bazei lor și caracterul '_'
- pentru a ușura citirea, se poate folosi caracterul '_' ca delimitator
- caracterele 'z' sau '?' specifică impedanță mare
- caracterul 'x' specifică valoare necunoscută
- se poate specifica dimensiunea numărului în biți dar și baza acestuia (b,B,d,D,h,H,o,O - binar, zecimal, hexa, octal)

Exemple:

```
8'b1;           // echivalent cu 1 sau 8'b00000001 etc.
8'b1010_0111;  // echivalent cu 167 sau 8'b10100111 etc.
8'b 10;        // echivalent cu 2 sau 8'b00000010 etc.
```

```
126;
16'habcd;
```

Operatori

Verilog pune la dispoziție mai multe tipuri de operatori. Unii dintre aceștia sunt cunoscuți din limbajele de programare precum C, C++, Java, și au aceeași funcționalitate. Alții sunt specifici limbajului Verilog și sunt folosiți în special pentru a descrie ușor circuite logice. [Tab. 1](#) conține operatorii suportați de Verilog, împreună cu nivelul lor de precedență.

Simbol	Funcție	Precedență
! ~ + - (unari)	Complement, Semn	1
**	Ridicare la putere	2
* / %	Înmulțire, Împărțire, Modulo	3
+ - (binari)	Adunare, Scădere	4
<< >> <<< >>>	Shiftare	5
< <= > >= == !=	Relaționali	6
& ^	Bitwise	7
& ~& ^ ~^ ^~ ~	Reducere	8
&&	Logici	9
?:	Condițional	10
{,} {n{}}	Concatenare	

Tab. 1: Operatori Verilog

În continuare sunt prezentați operatorii mai neobișnuiți suportați de Verilog:

- Operatorii de shiftare aritmetică; realizează shiftarea cu păstrarea bitului de semn, pentru variabilele declarate ca fiind cu semn.

Exemplu operatori de shiftare aritmetica

```
wire signed[7:0] a, x, y;

assign x = a >>> 1; // daca bitul de semn al lui a este 0 bitul nou
                    //      introdus este 0
                    //      // daca bitul de semn al lui a este 1 bitul nou
                    //      introdus este 1
assign y = a <<< 1; // bitul nou introdus este tot timpul 0, asemănător
                    //      cu operatorul <<
```

- Operatorii de reducere; se aplică pe un semnal de mai multi biți și realizează operația logică între toți biții semnalului.

Exemplu operatori de reducere

```
wire[7:0] a;
```

```
wire x, y, z;

assign x = &a; // realizează SI intre toti bitii lui a
assign y = ~&a; // realizează SI-NU intre toti bitii lui a
assign z = ^a; // realizează XNOR intre toti bitii lui a, echivalent
cu ^~
```

- Operatorul de concatenare; realizează concatenarea a două sau mai multe semnale, într-un semnal de lăţime mai mare.

Exemplu operator de concatenare

```
wire[3:0] a, b;
wire[9:0] x;

assign x = {b, 2'b01, a[2:1], 2'b00}; // bitii 9:6 din x vor fi egali
cu bitii 3:0 ai lui b // bitii 5:4 din x vor fi egali
cu 01 // bitii 3:2 din x vor fi egali
cu bitii 2:1 ai lui a // bitii 1:0 din x vor fi egali
cu 00
```

Tipul reg

În laboratorul trecut a fost prezentat tipul *wire* pentru reprezentarea semnalelor. Porturile unui modul erau wires, la fel şi semnalele de legătură dintre instanţele primitivelor şi porţilor. Deoarece acestea realizează conexiuni, nu au o stare şi nu li se pot atribui valori. Pentru a putea reţine stări/valori şi a face atribuiri avem nevoie de tipul **reg**.

Declararea variabilelor de tip *reg* se poate face într-un mod similar variabilelor de tip *wire*, cum este exemplificat şi mai jos:

```
reg [7:0] a;
reg [0:7] b;
reg c;

// vector de reg
reg [7:0] d[3:0]; // array 2D de 4 linii x 8 biţi

// matrice de reg
reg [7:0] e[3:0][3:0]; // array 3D de 4 linii x 4 coloane x 8 biţi
```

Declararea variabilelor *reg* şi *wire* se face în afara blocurilor *always* şi *initial*, iar atribuiri acestora se fac doar în interiorul acestor blocuri, prezentate în secţiunile [Blocul always](#) şi [Blocul initial](#).

Blocul `always@`

În afară de folosirea atribuirilor continue, circuitele pot fi descrise comportamental și prin blocuri `always`. În interiorul acestora se pot folosi construcții de limbaj similare celor din limbajele procedurale, decrie în secțiunea [Construcții de control](#).

Blocurile `always` descriu un comportament ciclic, codul acestora fiind executat în continuu. Prezența operatorului `@` face ca blocul să se “execute” doar la apariția unor evenimente. Evenimentele sunt reprezentate de modificarea unuia sau mai multor semnale.

În cadrul acestui laborator ne axăm doar pe descrierea circuitelor combinaționale, și vom folosi doar blocuri `always @(*)`, unde `(*)` se numește *sensitivity list*. Folosirea wildcard-ului `*` implică “execuția” blocului `always` la orice eveniment de modificare a semnalelor folosite în cadrul blocului.

Instrucțiunile din blocul `always@` sunt încadrate între cuvintele cheie `begin` și `end` și sunt “executate” secvențial atunci când blocul este activat.

Exemplu bloc `always`

```
always @(*)
begin
    b = 0;      // registrul b este inițializat cu 0 la orice
               // modificare a unui semnal
    c = 0;      // registrul c este inițializat cu 0 la orice
               // modificare a unui semnal
    c = b ^ a;  // registrul c va primi valoarea expresiei din dreapta
               // la orice
               // modificare a unui semnal (nu doar a sau b)
end
```

Pentru ca blocul `always` să fie combinațional este necesar ca toate variabile care trebuie calculate să fie inițializate la începutul blocului `always`. În caz contrar variabilele care nu au fost atribuite niciodată într-o anumită execuție a blocului `always` vor trebui memorate de la execuția anterioară ceea ce va duce la sintetizarea unui circuit secvențial asincron

Click pentru informații adiționale despre sensitivity list

În locul wildcard-ului, `*`, sensitivity list-ul poate conține o listă de semnale la modificarea cărora blocul `always` să fie activat. Acestea se declară prin numele lor, folosind `or` sau `,` pentru a le separa.

Exemplu specificare lista de semnale

```
always @(a or b or c)    // sintaxa Verilog-1995
always @(a, b, c)        // sintaxa Verilog-2001, 2005
always @(a, b or c)      // permis dar nerecomandat, ingreuneaza
                          // lizibilitatea codului
always @(*)              // toate semnalele din modul (intrari +
```

```
wires declare in modul)
```

Este foarte important ca lista de semnale dată unui bloc *always@* să fie **completă**, altfel nu toate combinațiile de intrări vor fi acoperite și este posibil ca unele variabile de ieșire să nu fie recalulate. Pentru a evita astfel de erori se recomandă folosirea wildcard-ului ***.

În modulul următor care implementează o poartă XOR, ieșirea *out* se va schimba doar când semnalul *a* se schimbă, ceea ce duce la un comportament incorect care nu ia în considerare și schimbarea lui *b*. În plus, modulul generat nu va fi unul combinațional, deoarece este nevoie de memorie pentru a menține starea ieșirii atunci când *b* se modifică.

Folosire incorecta sensitivity list

```
module my_xor(output reg out, input a, input b);

always @(a)
begin
    out = a ^ b;
end
endmodule
```

La fel și varianta *always @(b)* ar fi incorectă pentru că *out* s-ar modifica doar când semnalul *b* se modifică.

Pentru a evita astfel de greșeli folosiți *always (*)*.

Nu se pot instanția primitive și module în interiorul blocurilor *always* și *initial*.

Construcții de control

Cod Verilog	Cod C
<pre>if(a == 0) begin b = 2; end else begin b = 4; end</pre>	<pre>if(a == 0) { b = 2; } else { b = 4; }</pre>

<pre>case(sig) 0: a = 2; 1: a = 1; default: a = 0; endcase</pre>	<pre>switch(sig) { case 0: a = 2; break; case 1: a = 1; break; default: a = 0; }</pre>
<pre>for(i = 0; i < 10; i = i + 1) begin a = a % i; end</pre>	<pre>for(i = 0; i < 10; i = i + 1) { a = a % i; }</pre>
<pre>i = 0; while(i < 10) begin a = a % i; i = i + 1; end</pre>	<pre>i = 0; while(i < 10) { a = a % i; i = i + 1; }</pre>
<pre>repeat(10) begin a = a + 1; end</pre>	

Construcțiile de repetiție trebuie folosite cu atenție, deoarece pot duce foarte ușor la circuite nesintetizabile. Pentru a obține un circuit sintetizabil o condiție necesară pentru o buclă este ca numărul de iterații să fie cunoscut în momentul sintetizării. În alte cuvinte, numărul de iterații trebuie să fie fix și nu poate depinde de variabilele de intrare (direct sau indirect).

Exemplu modul descris comportamental

```
module my_module(
  output reg[3:0] o, // o trebuie să fie reg ca să îl folosim în
    atribuiri în always
  input[3:0] a, b);

  reg[2:0] i; // poate fi maxim 7; noi avem nevoie de maxim 4
  reg c; // ținem minte transportul

  always @(*)
  begin
    i = 0; // la orice modificare a intrărilor i va fi inițializat
    cu 0
    c = 0; // transportul inițial este 0
    o = 0; // la orice modificare a intrărilor o va fi inițializat
    cu 0

    // toti bitii lui o sunt recalculați la modificarea intrărilor
    for(i = 0; i < 4; i = i + 1) begin
      {c, o[i]} = a[i] + b[i] + c;
    end
  end
end
```

endmodule

Pentru ca un bloc *always* să fie sintetizat într-un circuit combinațional este necesar ca orice “execuție” a blocului să atribuie **cel puțin** o valoare pentru fiecare ieșire a modulului.

Bineînțeles, acea valoare nu poate fi calculată pe baza ieșirilor sau valorilor anterioare ale variabilelor din interiorul modulului. Asta ar însemna că este necesară o memorie pentru a menține acele valori, transformând circuitul într-unul secvențial.

Testare

Pentru testarea unui modul folosind simulatorul se creează module speciale de test, în care, printre altele, se vor atribui valori intrărilor. Pentru a crea un modul de test și a-l simula puteți urma [tutorialul de simulare](#). Această secțiune va prezenta câteva din construcțiile de limbaj pe care le puteți folosi într-un astfel de modul.

Blocul initial

Blocurile *initial* descriu un comportament executat o singură dată la începutul modulelor și sunt folosite pentru inițializări. Instrucțiunile sale trebuie încadrate între cuvintele cheie *begin* și *end* și sunt executate secvențial.

```
initial begin
    a = 0;
    b = 1;
    #10;      // delay 10 unități de timp de simulare
    a = 1;
    b = 0;
end
```

Sincronizarea prin întârziere

Folosind operatorul # se poate specifica o durată de timp între apariția instrucțiunii și momentul executării acesteia. Aceasta este utilă pentru a separa temporal diversele atribuiri ale intrărilor. Durata de timp este reprezentată prin unități de timp de simulare. De exemplu, dacă simularea folosește un *timescale* în nanosecunde, #*n* va reprezenta *n* nanosecunde.

```
initial begin
    a = 0;
    #100;    // delay 100 unități de timp de simulare
    a = 1;
```



```
end
```

Afișare

Atât în modulele de test cât și în modulele testate se pot folosi construcții pentru afișare în interiorul blocurilor *initial* și *always*. Una dintre aceste instrucțiuni este `$display`:

```
$display(arguments)
```

Argumentele acestei comenzi sunt similare cu cele ale funcției *printf* din C, ca în exemplul de mai jos, iar specificația completă o puteți găsi [aici](#). `$display` adaugă o linie nouă, iar dacă nu se dorește acest lucru se poate folosi comanda `$write`.

```
for(i = 0; i < 10; i = i + 1) begin
    $display("i=%d", i);
end
```

Exerciții

1. **(3p)** Implementați și simulați un sumator elementar complet folosind o singură atribuire continuă (`assign`).
 - Hint: Verilog are un operator pentru sumă.
 - Hint: Care este suma maximă care poate fi calculată cu un sumator elementar complet? De câți biți este nevoie pentru a o reprezenta?
 - Hint: Folosiți operatorul de concatenare.
2. **(3p)** Implementați și simulați un multiplexor 4:1 folosind un bloc comportamental ciclic (`always`).
 - Hint: Care este comportamentul unui multiplexor?
 - Hint: Folosiți instrucțiunea `case`.
3. **(4p)** Implementați și simulați un multiplicator pe 4 biți fără a folosi operatorul `*` (înmulțire).
 - Hint: Folosiți convenția Verilog pentru interfața modulului. Câți biți are ieșirea?
 - Hint: Înmulțiți pe hârtie, în baza 2, numerele 1001 și 1011. Transpuneți în limbajul Verilog algoritmul folosit.
4. **(2p)** Implementați o unitate aritmetico-logică simplă, pe 4 biți, cu 2 operații: adunare și înmulțire. Folosiți o intrare de selecție de 1 bit pentru a alege între cele două operații astfel: 0 - adunare, 1 - înmulțire. Utilizați multiplicatorul implementat la execuțiul anterior.
 - Hint: Puteți găsi un sumator pe 4 biți [în soluția laboratorului 1](#).
 - Hint: Câți biți au ieșirea sumatorului și a multiplicatorului? Dar a UAL-ului?
 - Hint: Folosiți o atribuire continuă pentru a selecta între ieșirea sumatorului și a multiplicatorului.

Resurse

- [Schelet de cod](#)
- [Soluție laborator](#) (disponibilă începând cu 12.10.2018)

- [PDF laborator](#)

Referințe

- Ciletti, Michael D. "Advanced digital design with the Verilog HDL". Prentice Hall, 2011
- [Verilog: always@ Blocks](#)
- [Verilog: wire vs. reg](#)

From:

<https://elf.cs.pub.ro/ac/wiki/> - **AC Wiki**

Permanent link:

<https://elf.cs.pub.ro/ac/wiki/lab/lab02>

Last update: **2018/10/01 08:38**

