

Laboratorul 1 - Limbajul Verilog: Introducere

Verilog

În cadrul laboratorului de Arhitectura Calculatoarelor vom studia un limbaj de descriere a hardware-ului (eng. *Hardware Description Language* - *HDL*) numit **Verilog**. Limbajele de descriere a hardware-ului sunt folosite în industrie pentru proiectarea și implementarea circuitelor digitale (eng. *Application-Specific Integrated Circuit* - *ASIC*). Cele mai folosite limbaje de descriere a hardware-ului sunt Verilog și VHDL. Din punct de vedere al sintaxei ele se aseamănă cu limbajele de programare de uz general (ex. C/C++/Java) însă oferă în plus anumite facilități pentru modelarea și simularea logicii combinaționale și secvențiale.

Proiectarea unui circuit digital într-un HDL începe printr-o descriere textuală a circuitului. Aceasta este compilată pentru a verifica sintaxa și a genera un model al circuitului iar apoi modelul poate fi rulat într-un simulator pentru a verifica funcționalitatea descrierii. O alternativă la rularea în simulator este sintetizarea unei configurații pentru programarea unui chip FPGA și testarea acestuia.

Verilog oferă mai multe alternative pentru descrierea unui circuit. Două dintre aceste alternative pe care le vom folosi în cadrul laboratorului sunt:

- **descrierea structurală** - porțile logice sunt legate între ele, asemănător cu o schemă logică, pentru a obține funcționalitatea dorită; aceasta este subiectul pe care îl vom aborda în laboratorul de față.
- **descrierea comportamentală** - se folosesc construcțiile de nivel înalt (ex. `if`, `for`, `while` etc.) pentru a descrie funcționalitatea dorită; vom aborda acest subiect în laboratoarele viitoare.

Primitive

Pentru descrierea structurală a circuitelor, Verilog oferă o serie de primitive (eng. *built-in primitives*) asociate porților logice de bază (nu există primitive secvențiale predefinite). O listă abreviată a primitivelor predefinite poate fi consultată în [Tab. 1](#).

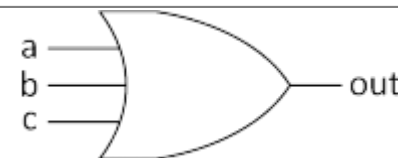
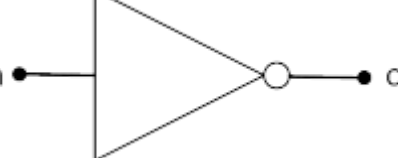
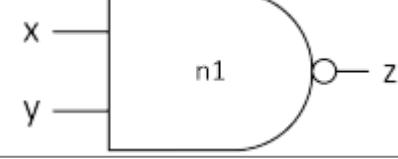
| N intrări | N ieșiri |
|-----------|----------|
| or | buf |
| and | not |
| nor | |
| nand | |
| xor | |
| xnor | |

Tab. 1: Primitive Verilog

Fiecare primitivă are porturi, prin care este conectată în exterior. Primitivele predefinite oferă posibilitatea conectării mai multor intrări (ex. `or`, `and`, `xor` etc.) sau mai multor ieșiri (ex. `buf`, `not`).

Folosirea unei primitive se face prin instanțierea cu lista de semnale care vor fi conectate la porturile ei. Pentru primitivele predefinite porturile de ieșire sunt declarate înaintea porturilor de intrare.

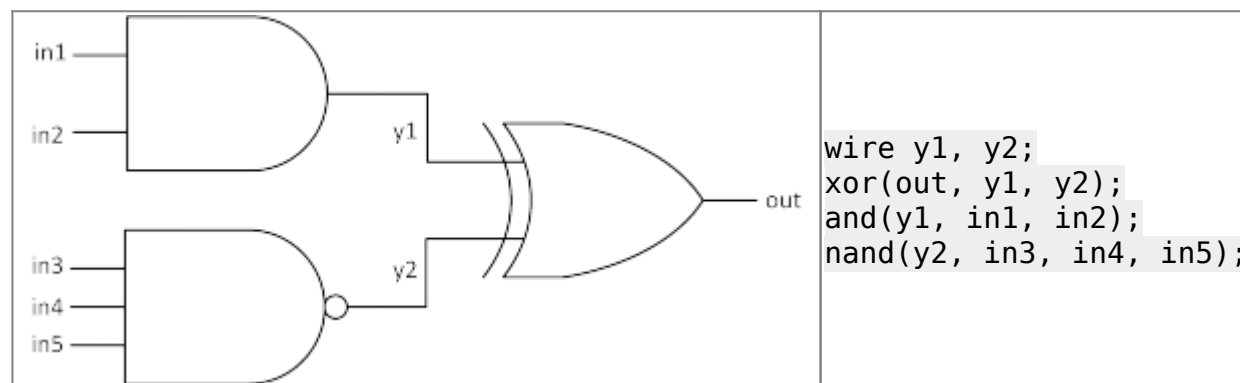
Tab. 2 oferă câteva exemple de instanțiere a unor porți în Verilog. Pentru primitivele predefinite numele instanței este opțional.

| Schema | Cod |
|---|--|
|  | <pre>or(out, a, b, c); // sau or ol(out, a, b, c);</pre> |
|  | <pre>not(out, in); // sau not my_not(out, in);</pre> |
|  | <pre>nand (z, x, y); // sau nand n(z, x, y);</pre> |

Tab. 2: Exemple de instanțiere a primitivelor

Wires

În Verilog, specificarea semnalelor dintr-o diagramă structurală se face prin **wires**. Acestea se declară prin cuvântul cheie `wire`.



În exemplul anterior `y1` și `y2` sunt niște semnale de câte 1 bit care leagă ieșirile porților `and` (`y1`) și `nand` (`y2`) la intrările porții `xor`.

Pentru a declara semnale pe mai mulți biți se pot folosi vectori precum în declarațiile următoare; `x` reprezintă un semnal de 8 biți, iar `y` reprezintă un semnal de 5 biți. Bitul cel mai semnificativ (eng. *most significant bit* - MSB) este situat întotdeauna în stânga, iar bitul cel mai puțin semnificativ (eng. *least significant bit* - LSB) în dreapta.

```

wire[7:0] x;           // 8 biti, MSB este bitul 7, LSB bitul 0
wire[0:4] y;           // 5 biti, MSB este bitul 0, LSB bitul 4

// vector de wires

```

```
wire[7:0] a[9:0];           // vector cu 10 elemente a câte 8 biti

// matrice de wires
wire[3:0] b[4:0][4:0];     // matrice de 5x5 elemente a câte 4 biti
```

În mod implicit semnalele care nu sunt declarate sunt considerate ca fiind de tip *wire* și având 1 bit (ex. *in1*, *in2*, ... din codul de mai sus).

Putem accesa individual biții dintr-un *wire* (ex. *x[0]*) sau într-un interval ca în limbajul Matlab (ex. *x[3:1]*, *x[7:2]*).

```
wire[7:0] x;               // 8 biti
wire[4:0] y;               // 5 biti

assign x[7:3] = y;         // bitii 7-3 din x o să fie egali cu biții din y
```

Module

Limbajul Verilog necesită ca toate elementele prezentate anterior (primitive și *wires*) să fie declarate într-un **modul**. Scopul modulului este să definească interfața și să încapsuleze funcționalitatea unui circuit.

Pentru a declara un modul se folosesc cuvintele cheie *module* și *endmodule*. Pe lângă aceste cuvinte cheie, declarația unui modul mai conține:

- numele acestuia
- lista de porturi (pentru interfața cu exteriorul); pot fi porturi de intrare (*input*), porturi de ieșire (*output*) sau porturi de intrare-ieșire (*inout*) și pot avea mai mulți biți

Exemplu declarare modul

```
module exemplu(output o, input a, input[3:0] b);
    // descrierea functionalitatii
endmodule
```

Click pentru un mod alternativ de declarare a unui modul

Exemplu declarare modul în Verilog 1995)

```
module exemplu(o, a, b);
    output o;
    input a;
    input[3:0] b;
    // descrierea functionalitatii
endmodule
```

Spre deosebire de primitive, ordinea porturilor unui modul nu este restricționată. Intrările și ieșirile pot fi declarate în orice ordine, însă, pentru consistență, se preferă folosirea convenției de la primitive: prima dată se declară ieșirile, apoi intrările.

Implicit, toate porturile sunt de tip *wire*, însă acest lucru poate fi modificat pentru porturile de ieșire (porturile de intrare nu pot fi modificate). Vom discuta acest aspect în laboratoarele următoare.

Instanțiere module

Descrierea funcționalității unui modul poate folosi, pe lângă primitive, și instanțe ale altor module. Acest lucru se face asemănător cu instanțierea unei primitive, cu diferența că, în acest caz, numele instanței **nu** este opțional.

Exemplu instanțiere modul

```
module mux4_1(output out, input[3:0] in, input[1:0] sel);
    // implementare multiplexor 4:1
endmodule

module mux8_1(output out, input[7:0] in, input[2:0] sel);
    mux4_1 m1(out30, in[3:0], sel[1:0]);    // instantiere mux4_1
    mux4_1 m2(out47, in[7:4], sel[1:0]);    // alta instantiere mux4_1
    // logica aditionala
    not(n_sel2, sel[2]);
    and(y1, out30, n_sel2);
    and(y2, out47, sel[2]);
    or(out, y1, y2);
endmodule
```

În exemplul anterior definiția modulului *mux8_1* folosește instanțe ale modulului *mux4_1*, denumite *m1* și *m2*. Semnalele *out30*, *in[3:0]* și *sel[1:0]* sunt legate la porturile *out*, *in* și, respectiv, *sel* ale instanței *m1*, asemănător cu apelul unei funcții în C.

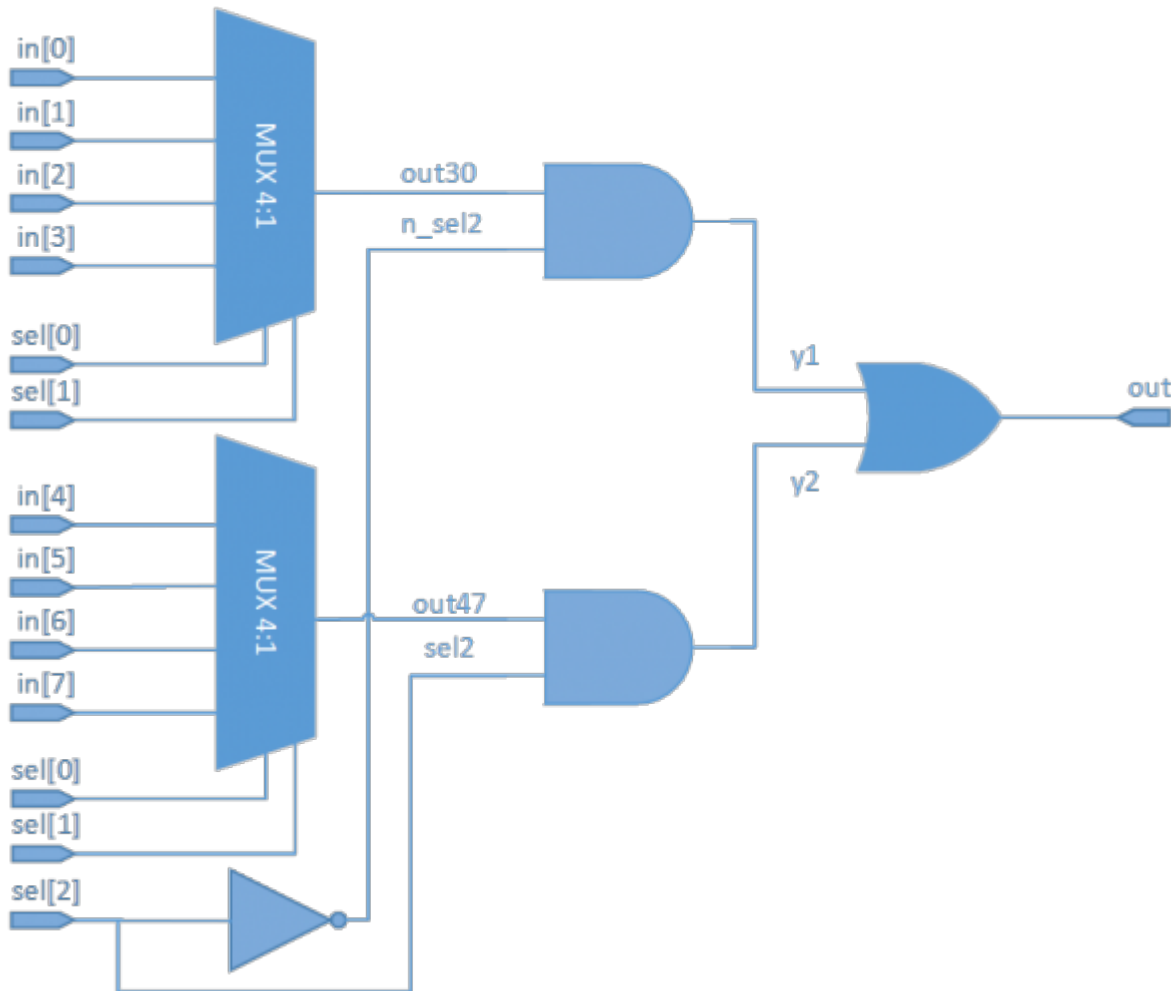


Fig. 1:

Multiplexorul 8:1

Click pentru un mod alternativ de instanțiere a unui modul

În Verilog, legătura dintre porturile unei instanțe și semnalele legate la aceste porturi poate fi făcută și pe baza numelor, nu doar a poziției. Această metodă de instanțiere a unui modul poate fi observată în exemplul următor.

Exemplu instanțiere modul

```
module mux4_1(output out, input[3:0] in, input[1:0] sel);
    // implementare multiplexor 4:1
endmodule

module mux8_1(output out, input[7:0] in, input[2:0] sel);
    mux4_1 m1(.out(out30), .in(in[3:0]), .sel(sel[1:0])); //
    // instanțiere mux4_1
    mux4_1 m2(.sel(sel[1:0]), .in(in[7:4]), .out(out74)); // alta
    // instanțiere mux4_1
    // logica aditionala
    not(n_sel2, sel[2]);
    and(y1, out30, n_sel2);
    and(y2, out74, sel[2]);
    or(out, y1, y2);
endmodule
```

endmodule

Această metodă de instanțiere nu este suportată pentru primitive, porturile acestora neavând asociate nume.

Proiectarea Top-Down

Proiectarea top-down se referă la partiționarea sistematică și repetată a unui sistem complex în unități funcționale mai simple, a căror proiectare poate fi făcută mai facil. O partiționare și organizare la nivel înalt a unui sistem reprezintă arhitectura acestuia. Unitățile funcționale individuale ce rezultă în urma partiționării sunt mai ușor de proiectat și de testat decât întregul sistem. Strategia divide-et-impera a proiectării top-down ne permite proiectarea de circuite care conțin milioane de porți.

Instanțierea unui modul în definiția unui alt modul este numită imbricare (eng. *nested module*). Modulele imbricate reprezintă mecanismul oferit de Verilog pentru proiectarea top-down, deoarece imbricarea creează automat o partiționare a sistemului.

Sintaxă

- Verilog este un limbaj case-sensitive; `x_in` și `x_In` sunt tratate ca două semnale diferite.
- Numele identificatorilor (ex. nume de module, semnale și porturi) pot conține doar *litere*, *cifre*, `_` și `$`; ele trebuie să înceapă cu `_` sau cu o *literă*.
- Comentariile se marchează cu `//` sau se încadrează între `/`.

Xilinx ISE

În cadrul laboratorului vom folosi mediul de dezvoltare **Xilinx ISE**, varianta WebPACK, pentru simularea codului Verilog și programarea plăcilor cu FPGA. WebPACK este versiunea gratuită a Xilinx ISE disponibilă pentru download pe site-ul [Xilinx](#). Un [tutorial](#) pentru instalarea versiunii 14.6 (folosite în laborator) găsiți în secțiunea tutorială. ⚠ Sistemul de operare Windows 8 nu este suportat în mod oficial.

Pentru crearea proiectelor și modulelor folosind Xilinx ISE urmăriți [tutorialul](#) de pe wiki.

Exerciții

1. **(3p)** Simulați sumatorul elementar complet din scheletul de cod. Corectați cele 6 erori de sintaxă din implementare.

- Hint: Descărcați scheletul de cod pentru exerciții.
 - Hint: Scheletul de cod conține deja un proiect Xilinx ISE și un modul de testare.
 - Hint: Urmăriți [tutorialul](#) pentru a realiza simularea (săriți peste adăugarea modulului de test, pașii 2-6, deoarece acesta este deja adăugat). Simularea nu va fi realizată din prima deoarece codul conține erori de sintaxă.
 - Hint: Comanda de verificare a sintaxei, *Behavioral Check Syntax*, situată deasupra celei de simulare, este utilă pentru detectarea mai rapidă a erorilor. ⚠️ Rulați comanda selectând în prealabil fișierul `full_adder.v`.
 - Hint: Erorile de sintaxă găsite vor apărea în fereastra *Errors*.
 - Hint: Este bine să începeți rezolvarea cu prima eroare, deoarece toate erorile următoare pot fi cauzate de aceasta. Navigați la **începutul** ferestrei de erori și citiți primul mesaj. Apăsând pe link-ul roz veți fi duși la linia de cod care a cauzat eroarea.
2. **(3p)** Implementați și simulați un multiplexor 4:1. Urmăriți diagrama de semnale generată.
 - Hint: Consultați [laboratorul 0](#) pentru implementarea unui multiplexor 4:1.
 - Hint: Respectați interfața cerută în scheletul de cod.
 3. **(2p)** Implementați un sumator pe 4 biți. Verificați corectitudinea sumatorului vizualizând semnalele în baza 10.
 - Hint: Consultați [laboratorul 0](#) pentru implementarea unui sumator pe mai mulți biți.
 - Hint: Folosiți sumatorul implementat la exercițiul 1, adăugându-l la proiect din meniul *Project→Add Copy of Source...*
 - Hint: Modificați afișarea unui semnal cu *click-dreapta→Radix→Unsigned Decimal*.
 4. **(2p)** Implementați și simulați un sumator pe 6 biți. Folosiți un sumator pe 4 biți și două sumatoare pe 1 bit.
 - Hint: Respectați interfața cerută în scheletul de cod.
 5. **(2p)** Implementați și simulați un comparator pe un bit. Acesta are două intrări și 3 ieșiri (pentru mai mic, egal și mai mare).
 - Hint: Respectați interfața cerută în scheletul de cod.

Resurse

- [Schelet de cod](#)
- [Soluție laborator](#) (disponibilă începând cu 05.10.2018)
- [PDF laborator](#)

Referințe

- Ciletti, Michael D. "Advanced digital design with the Verilog HDL". Prentice Hall, 2011

From:

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

Permanent link:

<https://elf.cs.pub.ro/ac/wiki/lab/lab01>

Last update: **2018/09/27 10:08**



