S2. Introducere în limbajul Verilog

Modelarea circuitelor combinaționale utilizând limbaje de descriere hardware

1. Obiective

O1. Însușirea primelor noțiuni de Verilog, un limbaj de descriere hardware

O2. Implementarea circuitelor logice combinaționale folosind Verilog HDL

O3. Familiarizarea cu operatorii și constantele specifice limbajului Verilog

2. Considerații teoretice

2.1 Fundamentele modelării în Verilog

Verilog Hardware Description Language (HDL) furnizează o reprezentare textuală modulară și portabilă al arhitecturii unui sistem. Un sistem digital poate fi implementat la nivelul mai multor layere de abstractizare, începând de la nivelul de tranzistor, dependent de tehnologie până la nivelul de poartă logică, independentă de tehnologie, nivelul de registru de transfer (RTL) și algoritm (sau nivelul comportamental). Acest îndrumător de laborator se axează pe descrierea sistemelor digitale la nivelul RTL deoarece oferă o ilustrare eficace și condensată, rămânând în același timp puternic corelată cu implementarea fizică a sistemului. Codul Verilog exemplificat poate fi canalizat direct spre blocurile logice configurabile (CLBs) a unui dispozitiv FPGA (Field Programmable Gate Array) sau chiar unui ASIC (Application Specific Integrated Circuit).

2.1.1 Modulele Verilog

Unitatea de bază a limbajului Verilog este **modulul**, ce conține descrierea interfeței și a comportamentului unui circuit electronic. Un modul este cel mai apropiat de conceptul de “black box” ce constă în cunoașterea intrărilor și a ieșirilor, fără a ști însă detaliile de implementare și modul în care el funcționează.

Observație:
În alte limbaje de programare, unitatea de bază este funcția. Un modul, odată instanțiat, putem să îl vedem ca pe o ”funcție” care se autoapelează.
Noi nu putem să “apelăm” un modul pentru a executa o acțiune. Spre exemplu, nu putem să apelăm un modul sumator pentru a face suma a două numere. Ce putem face însă este să instanțiem un sumator, să îi legăm ca și intrări cele două numere și vom ști că la ieșirea sumatorului vom avea suma lor.

Declararea unui modul se face în felul următor:

```
module <nume_modul> (
    <tip_port_1> <nume_port_1>,
    <tip_port_2> <nume_port_2>,
    ...
    <tip_port_n> <nume_port_n>
    );
// Implementarea modulului.
endmodule
```

Această declarație constă în:

- Cuvântul cheie **module**
- **Numele modulului**
- **Lista de porturi:** elementele declarate aici sunt folosite pentru interfațarea cu exteriorul și pot fi
  - De intrare **(input)**
  - De ieșire **(output)**
  - Bidirecționale **(inout)**
- **; (punct și virgulă)**
- Implementarea modulului
- Cuvântul cheie **endmodule**

Interfața unui modul este formată din:

- Numele modulului: poate conține litere, cifre, $ și _. Primul caracter trebuie să fie o literă sau _.
- Lista de porturi: zero sau mai multe porturi, fiecare dintre acestea având o direcție (input, output sau inout) și un nume. Putem sa declarăm doar numele porturilor în acestă listă, însă, în acest caz, direcția porturilor trebuie declarată în corpul modulului.
  Se recomandă ca în această listă să fie declarate mai întâi intrările și apoi ieșirile.

```
module modulA ();

// Implementare modul.

endmodule


module modulB (a, b, c, d);

input a;

input b;

output c;

inout d;

// Implementare modul.

endmodule


module modulC (input a, output b);

// Implementare modul.

endmodule
```

Implementarea modului constă în descrierea funcționării acestuia utilizând:

- Declarații
  - Fire (**wire**): reprezintă conexiuni fizice între componente. Se folosesc pentru transmisia semnalelor (doar în descrierea logicii combinaționale) și nu au capacitate de reținere a informației (nu au o stare)
  - Registre (**reg**): sunt folosite pentru a stoca date, care persistă chiar dacă registrul este deconectat. Registrul este echivalentul unei variabile interne dintr-un limbaj de programare. Reține o valoare și i se poate atribuiri o valoare
- Construcții
  - Instanțieri de module
  - Atriburi continue (**assign**) pot fi folosite doar pe wire și implică faptul că acest **wire** este ieșirea unei combinații de porți logice
  - Blocuri **initial:** ne permit să definim o stare inițială. Acest bloc se va executa o singură dată, la inițializarea modulului.
  - Blocuri **always:** conțin acțiuni ce vor fi executate periodic

Intrările, ieșirile și porturile bidirecționale ale unui modul sunt implicit considerate ca având tipul **wire.** Putem să specificăm, dacă dorim, ca ieșirile să aibă tipul **reg**.

Toate variabilele de tip **wire** și **reg** (incluzând și cele de tip input, output și inout) au, în mod implicit, lățimea de bandă de 1 bit. Putem specifica, dacă dorim, o lățime mai mare de atât folosind construcția `[i:j]`. Atenție, i și j nu pot lua valoarea 0 simultan iar *i >=j* sau *j>=i*.

```
input a;          // Variabila fir de intrare, pe 1 bit.
output b;         // Variabila fir de ieșire, pe 1 bit.
output reg c;     // Variabila registru de iesire, pe 1 bit.
inout d;          // Variabila fir bidirecțională, pe 1 bit.
wire e;           // Variabila fir locală, pe 1 bit.
reg f;            // Variabila registru locală, pe 1 bit.
input [7:0] g;    // Variabila fir de intrare, pe 8 biți ordonați 7 → 0
wire [0:7] h;     // Variabila fir locală, pe 8 biți ordonați 0 → 7.
reg [2:6] i;      // Variabila registru locală, pe 5 biți ordonați 2 → 6.
```

Limbajul Verilog permite folosirea unui modul în descrierea altui modul prin instanțierea acestuia. Instanțierea modulelor se face în felul următor:

```
nume_modul <nume_instanta> (
    <nume_argument_1>,
    <nume_argument_2>,
    ...
    <nume_argument_n>,
    );
// Continuare dreapta
```

```
nume_modul <nume_instanta> (
    .nume_port_1(<nume_argument_1>),
    .nume_port_2(<nume_argument_2>),
...
    .nume_port_n(<nume_argument_n>),
    );
```

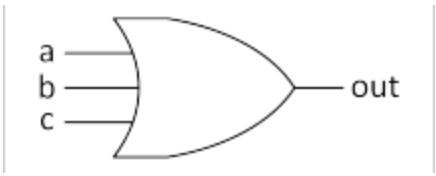
Mai jos avem un exemplu de declarație și instanțiere de modul.

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

    // Implementare modul.

endmodule

module modulB(input in1, input in2, output out);

    reg out1, out2, out3, out4, out5;

    modulA a1(in1, in2, out1, out2);

// Am scris argumentele in ordinea porturilor.

    modulA a2(.b(in2), .a(in1), .c(out3), .d(out4));

// Am scris argumentele intr-o ordine aleatoare, dar pentru fiecare am specificat
numele portului la care trebuie legat.

    modulA a3(in1, in2, out5);       // Nu am legat portul "d". Nu este o eroare.

// Implementare modul.

endmodule
```

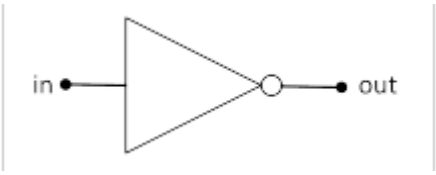
Observație:
Nu este necesar să legăm toate porturile unui modul atunci când îl instanțiem. Spre exemplu, un sumator poate avea o ieșire de date (suma celor două intrări) și o ieșire de transport (carry). Dacă ieșirea de transport nu ne folosește este obișnuit să nu o conectăm la nicio variabilă.

Pentru a descrie circuite folosind Verilog, avem la dispoziție și o serie de primitive care sunt asociate porților logice de bază. Fiecare primitivă are asociate porturi prin intermediul cărora se face legătura cu exteriorul. Astfel, există primitive predefinite care oferă posibilitatea conectării mai multor intrări (**and**, **or**, **nor**, **nand**, **xor**, **xnor**), sau a mai multor ieșiri (**buf**, **not**).

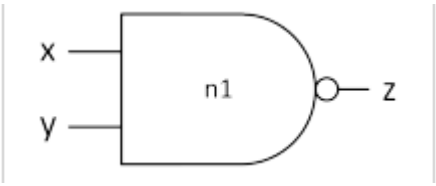
Folosirea unei primitive se face prin instanțierea cu lista de semnale care vor fi conectate la porturile ei.




```
or(out, a, b, c);
```




```
not(out, in);
```




```
nand(z, x, y);
```

Observație:
În cazul primitivelor este obligatoriu să declarăm la început semnalele de ieșire, acestea fiind urmate de cele de intrare.
Pentru primitivele predefinite numele instanței este opțional.
Așa cum afirmația precedentă a lăsat să se întrevadă, în Verilog se pot defini și UDP-uri (User Defined Primitives), prin intermediul tabelei de adevăr.

Descrierile Verilog a unui sistem digital sunt constituite din module care, la nivel RTL, ilustrează comportarea componentei unui sistem. Modulul începe cu cu descrierea intrărilor și ieșirilor. La nivel global, intrările și ieșirile unui modul sunt cunoscute sub numele de porturi. La nivelul registrelor de transfer, un modul poate să conțină următoarele tipuri de instrucțiuni:

a) assign
b) always
c) instanțe ale altor module

Instrucțiunile de tip **assign** sunt numite *comenzi continue* tocmai datorită faptului că descriu comportamentul design-ului combinațional. Pe de altă parte, instrucțiunile de tip **always** pot fi folosite pentru a descrie atât circuitele combinaționale cât și cele secvențiale, în funcție de maniera în care comanda este folosită. Utilizarea neadecvată poate să conducă la efecte nedorite atunci când vine vorba de definirea unei descrieri combinaționale în locul uneia secvențiale și vice versa.

Modulul Verilog din Fig. 1 implementează un multiplexor de tip 2-la-1 cu datele de intrare *d0* și *d1*, linia de selecție *s* și ieșirea *o.* Un modul Verilog debutează cu interfața modulului care include numele modului și porturile sale, închise între paranteze rotunde. Implementarea modulului conține o linie de cod unde ieșirea *o* va fi definită de valoarea dată de **operatorul condițional** ”**?”**

```
condition_expression ? expression_true : expression_false
```

care evaluează fie *expression_true* sau *expression_false* în funcție de valoarea dată de *condition_expression* (1 sau 0).

```
1 module mux_1s_1b (
2   input d0,
3   input d1,
4   input s,
5   output o
6 );

8   assign o = s ? d1 : d0;
9 endmodule
```

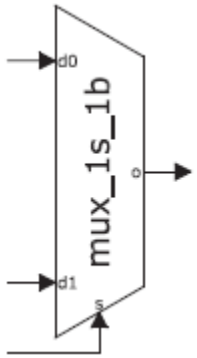


Fig. 1. Descrierea Verilog a unui multiplexor 2 la 1 pe 1 bit

Un aspect important de înțeles atunci când redactăm instrucțuini de tip **assign** este acela că **”instrucțiunile de tip assign nu sunt imperative în natura lor” [Stro05],** un aspect care diferențiază limbajul Verilog de alte limbaje de programare imperative. Linia de cod din Fig.1 nu atribuie o valoare variabilei o! În schimb ea conectează partea dreaptă și partea stângă a instrucțiunii printr-o legătură fizică **[Stro05].** De asemenea, modulul în sine definește conexiunea fizică a intrărilor modului cu ieșirile sale. Toate numele semnalelor ajung să devină fire într-o implementare fizică. Comentariile în Verilog au același format ca și în Java și C. Pentru vizibilitate ridicată și citire accesibilă se încurajează **indentarea corespunzătoare**.

2.1.2 Magistrale

Să luăm în considerare faptul că multiplexorul 2-la-1, definit anterior, necesită să fie modificat în așa fel încât să poată opera cu date pe 32 de biți. Pentru a elimina necesitatea de a lucra cu 65 de linii de intrare (32 de linii pentru intrarea *d0*, 32 de linii pentru intrarea *d1* și o linie de selecție) și pentru a evita scrierea a 32 de comenzi assign similare, Verilog incorporează conceptul de magistrală (bus), ca și o colecție de fire. O magistrală poate fi asimilată cu un **vector** și este definită specificând rangul superior și inferior a unui bit între paranteze drepte, în imediata apropiere a magistralei. Multiplexorul pe 32 de biți, de tip 2-la-1 este descris în Fig.2.

Este demn de menționat faptul că magistralele definite în liniile de cod 2,3 și 5 în Fig.2 au ranguri de la 31 la 0. Instrucțiunea assign din codul de la Fig.1 **rămâne neschimbată**.

```
1 module mux_1s_32b (
2   input [31:0] d0,
3   input [31:0] d1,
4   input s,
5   output [31:0] o
6 );

8   assign o = s ? d1 : d0;
9 endmodule
```

Fig. 2. Descrierea Verilog a unui multiplexor 2 la 1 pe 32 biți

Limbajul Verilog permite selectarea unui set de linii din cadrul unei magistrale. Să considerăm, spre exemplu, cazul construirii unui multiplexor care preia o intrare pe 64 de biți, iar în funcție de linia de selecție, va avea la ieșire fie jumătatea mai semnificativă sau mai puțin semnificativă. Acest lucru poate fi obținut folosind facilitatea **part-select** așa cum este descrisă în Fig. 3.

```
1 module mux_half_64b (
2   input [63:0] d,
3   input s,
4   output [31:0] o
5 );

7   assign o = s ? d[63:32] : d[31:0];
8 endmodule
```

Fig. 3. Un multiplexor 2 la 1 cu facilitatea part-select pentru o intrare pe 64 biți

Dacă linia de selecție ***s*** este activă, jumătatea mai semnificativă ai intrării ***d*** este generată la ieșire, altfel jumătatea mai puțin senificativă este transportată la ieșire. Expresia true a operatorului condițional, în linia 7, este un vector pe 32 de biți care include rangurile 63 până la 32 a intrării *d*. În mod similar Expresia false reprezintă jumătatea mai puțin semnificativă ai intrării ***d***.

Observație:
Atunci când se dorește livrarea unui pachet de date reprezentând jumătatea unui interval de biți declarat inițial este important să vă asigurați că lățimile expresiilor true și false a operatorului condițional sunt egale!

Magistralele pot fi de asemenea generate în Verilog prin intermediul unor constructori dedicați. Primul operator Verilog care permite construirea magistralelor este **operatorul de concatenare**. Concatenarea este construită ca o listă de semnale, separate de virgulă, închise între acolade. Semnalele concatenate sunt anexate una la alta, primul semnal ocupând cele mai semnificative poziții, urmat de al doilea semnal concatenat și așa mai departe. Ca și exemplu, să considerăm un modul care, în funcție de valoarea liniei de control, numită *swch,* comută cele două jumătăți a unei intrări pe 32 de biți. Implementarea modulului este ilustrată în Fig. 4.

Comutarea jumătăților este realizată plasând jumătatea mai puțin semnificativă a lui *i* (poziția 15 către 0) în fața jumătății celei mai semnificative în cadrul operatorului de concatenare. Dacă *swch* are valoarea 0, ieșirea va avea vectorul ordonat de biți așa cum a fost declarat la intrare.

```
1 module hlvs_swch (
2   input [31:0] i,
3   input swch,
4   output [31:0] o
5 );
6
7   assign o = swch ? {i[15:0], i[31:16]} : i;
8 endmodule
```

Fig. 4. Modul Verilog pentru comutarea jumătăților unei intrări pe 32 de biți

Operatorul de concatenare poate fi folosit de asemenea ca un factor în partea stângă a unei expresii. O astfel de construcție este folosită pentru a atribui valori mai multor semnale într-o singură propoziție. Lățimile părții stângi și a părții drepte au nevoie să fie egale. Implementarea comportamentală a unui sumator binar este un exemplu tipic în acest sens. Să considerăm un sumator binar pe 8 biți a cărui cod este descris în Fig.5.

```
1 module add_8b (
2   input [7:0] x,
3   input [7:0] y,
4   output [7:0] z,
5   output co
6 );
7
8   assign {co, z} = x + y;
9 endmodule
```

Fig. 5. Sumator pe 8 biți cu transport la ieșire

Interfața sumatorului include operanzii pe 8 biți x și y, ieșirea z, tot pe 8 biți, pentru sumă și o ieșire de transport pe 1 bit. Adăugând două numere pe n-biți va rezulta într-o sumă de n+1 biți. În consecință, membrul stâng al expresiei din linia de cod 8 (vezi Fig.5) are 9 biți, bitul de transport aflându-se pe poziția cea mai semnificativă. O remarcă ar trebui făcută în legătură cu operanzii de tip modulo 2^n . Rezultatul acestei adunări este reprezentat doar de n biți (fără transport). În acest caz, membrul stâng al expresiei adoptă o lățime de bandă de n biți iar rezultatul este

generat astfel: **assign** *z = a + b.* Simulatorul sau ustensila pentru sinteză va selecta fie un sumator tipic pe n biți sau un sumator de tip modulo 2^n bazat pe lățimea membrului stâng al expresiei.

Cea de-a doua facilitate pentru construirea magistralelor este **operatorul de replicare**, care, așa cum sugerează numele, realizează duplicarea unei expresii de un număr de ori, concatenând toate copiile acelei expresii într-un vector.

Operatorul de replicare se poate construi după următorul format:

```
{<number of replications>{<value to be replicated>}}
```

De exemplu, luați în considerare proiectarea unui modul Verilog pentru extinderea unui număr cu semn pe 16 biți la unul 32 de biți. Numerele negative au valoarea 1 ca fiind cei mai semnificativi biți (MSB), în timp ce pozițiile pozitive alocă valoarea 0. Reprezentarea largă a semnăturii dominate în sistemele digitale este **Complementul de doi** (C2) **[Vlad12]** iar extensia semnelor se realizează prin adăugarea semnalului bit în cele mai semnificative poziții.

```
1 module sgn_extd (
2   input [15:0] i,
3   output [31:0] o
4 );

6   assign o = {{16{i[15]}}, i};
7 endmodule
```

Fig. 6. Modul Verilog pentru extinderea semnului

Fragmentul de cod din Fig. 6 implementează extinderea semnului realizând duplicarea intrărilor de 16 ori în cele mai semnificative poziții, ca mai apoi să treacă la concatenarea numărului de la intrare.

2.1.3 Tipuri de date

Verilog operează cu două tipuri de date:

- fizice
- abstracte

Tipurile fizice de date au o corespondență exactă în hardware-ul folosit în modelarea RTL. Numai tipurile de date fizice pot forma magistrale. Tipurile de date fizice cele mai uzuale sunt:

a) **wires -** descriu conexiunile dintre componentele interne ale unui modul. Firele sunt utilizate la orice nivel de abstractizare Verilog. Un fir transportă o valoare dintr-un loc în altul fără a stoca valoarea respectivă. Valorile firelor sunt setate fie prin declarații de alocare sau prin ieșiri de module.

b) **reg** - descrie depozitele care păstrează ultima valoare atribuită acestora. În consecință, semnalele de tip reg nu trebuie să fie conduse în mod continuu, spre deosebire de fire. Ambele semnale interne sau modulele pot fi de tip reg. Ele sunt utilizate la RTL sau la nivel de algoritm ca precum și în testbenches. Valoarea lor este setată într-un bloc **always** sau unul **initial.**

Tipurile de date abstracte, deși au o corespondență hardware într-o anumită măsură, depind pe instrumentul de simulare / sinteză utilizat. Ele sunt folosite în implementările comportamentale Verilog, și în testbenches, facilitând testarea celorlalte module Verilog. Tipurile abstracte includ:

a) ***integer*** – reprezentând numere întregi pe 32 de biți
b) ***time*** – reprezentând valori fără semn, pe 64 de biți, stocând timpul de simulare
c) ***real*** – reprezentând numere de tip floating-point
d) ***parameter*** – reprezentând o constantă specifică modulului, precum lățimea, inițializarea valorilor

2.1.4 Operatorii

Operatorul condițional, prezentat anterior, este singurul operator Verilog de tip ternar. Operatorii de tip **bitwise** au vectori ca și argumente. Simbolurile lor, funcțiile și numărul de operanzi pe care un operator îl poate prelua, sunt prezentate în tabelul de mai jos:

Tabel 1 - Operatorii Verilog bitwise

Symbol	Function	Arity
~	bitwise complementation	unary
&	bitwise AND	binary
—	bitwise OR	binary
^	bitwise EXOR	binary
^~	bitwise XNOR	binary

Operatorul biwise NAND se obține prin negarea rezultatului unui bitwise AND precum (a | b). Într-un mod similar operatorul NOR este definit. Folosind operatorii de tip bitwise, multiplexorul pe 32 de biți 2-la-1, implementat în Fig.2 poate fi rescris precum în Fig.7.

```
module mux_1s_32b (
  input [31:0] d0,
  input [31:0] d1,
  input s,
  output [31:0] o
);
  wire [31:0] s_v;

  assign s_v = {32{s}};
  assign o =  (~s_v & d0) | (s_v & d1);
endmodule
```

Fig. 7. Multiplexor pe 32 biți implementat cu operanzi bitwise

Operatorii de reducere sunt operatori de tip unar luând o intrare de vector și generând o ieșire obținută prin aplicarea respectivului operator peste toate liniile vectorului de intrare. Ca și exemplu, testarea unui vector a că este nul poate fi realizată prin următoarea afirmație **assign is_0 = ~(|a);.** Operatorul de reducere aplică un OR peste toți biții vectorului a obținând un rezultat pe un singur bit care este negat ca **is_0** să fie 1 când a este 0...00.

Verilog mai oferă de asemenea numere aritmetice, relaționale, operatori de deplasare și logici precum +, -, *, /, %, <, >, <=, >=, ==, !=, <<, >>, !, &&, ||. Operatorii logici și relaționali returnează fie valoarea logică booleană 1 dacă o expresie este adevărată, sau 0 pentru o expresie falsă.

2.1.5 Constantele

Constantele Verilog au următorul format `<bit width>'<radix specifier><value>` pentru care:

a) **bit_width** este un număr zecimal reprezentând numărul de biți alocați constantei. Deși opțional, este o bună practică să se precizeze lungimea constantă a biților
b) **radix_specifier** poate fi *b* pentru binar, *o* pentru octal, *d* pentru zecimal si *h* pentru hexazecimal. Această zonă este opțională, iar radixul zecimal este cel implicit.
c) **value** este valoarea constantei menționată în radixul specificat.

Tabel 2 – Constante Verilog

Verilog Constant	Stored as
3'b110	110
8'b0010_1101	00101101
5'd6	00110
12'ha9e	101010011110

3. Referințe bibliografice

[Ashe07] P. J. Ashenden, *Digital Design (Verilog): An Embedded Systems Approach Using Verilog.* Morgan Kaufmann, 2007.

[Nava05] Z. Navabi, *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification*, 2nd ed. McGraw-Hill Professional, 2005.

[Paln03] S. Palnitkar, *Verilog Hdl: A Guide to Digital Design and Synthesis*, 2nd ed. Prentice Hall, 2003.

[Vlad12] M. Vlăduțiu, *Computer Arithmetic: Algorithms and Hardware Implementations*, 2012th ed. Springer, 2012.

[Stro05] L. Strozek. Verilog Tutorial - Edited for CS141. [Online]. Available: Verilog Tutorial