Laboratorul 2 - Operatori. Atribuire continuă. Parametrizare. Testare

În laboratorul anterior au fost prezentate elementele Verilog necesare pentru descrierea structurală a circuitelor logice. Aceasta poate deveni complicată și dificil de înțeles pentru circuite ce îndeplinesc o funcționalitate complexă.

Laboratorul curent va prezenta elementele Verilog folosite pentru descrierea comportamentală. Aceasta poate descrie **ce face** circuitul și nu **cum** va fi acesta implementat. Mai mult, vom completa un modul funcțional cu un modul de testare, astfel încât să avem posibilitatea de a verifica implementarea pe care o concepem.

Structura limbajului Verilog - continuare

Assign

Deși partiționarea circuitelor în cadrul unei arhitecturii duce la simplificarea implementării unui modul, implementarea acestuia la nivel de porți logice este rareori folosită, întrucât aceasta devine complicată și dificil de înțeles.

Primul pas este reprezentat de ușurarea modalității de scriere a unei funcții logice. Pentru aceasta, Verilog oferă o instrucțiune numită **atribuire continuă**. Aceasta folosește cuvântul cheie `assign` și "atribuie" unei variabile de tip `wire`, valoarea expresiei aflată în partea dreaptă a semnului egal. Atribuirea are loc la fiecare moment de timp, deci orice schimbare a valorii expresiei din partea dreaptă se va propaga imediat. 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.

Bazându-ne pe circuitul descris în figura de mai jos, acesta se poate scrie sub formă de atribuiri continue în următoarea formă:

```
    wire y1, y2;
  xor(out, y1, y2);
  and(y1, in1, in2);
nand(y2, in3, in4, in5);
```

Exemplu atribuire continuă

```
module my_beautiful_module (
    output out,
```

```
    input  i1,i2,i3,i4,i5);

          assign y1 = i1 & i2;
          assign y2 = ~(i3 & i4 & i5);

          assign out = y1 ^ y2;

endmodule
```

sau, mai concis:

```
module my_beautiful_module (
    output out,
    input  i1,i2,i3,i4,i5);

          assign out = (i1 & i2) ^  (~(i3 & i4 & i5));

endmodule
```

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.

Este o eroare să folosiți aceeași variabilă destinație pentru mai multe atribuiri continue. Ele vor încerca simultan să modifice variabila, lucru ce nu este posibil în hardware.

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
- caracterul '?' specifică impedanță mare (z)
- 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)

```
8'b1;         //binar, pe 8 biti, echivalent cu 1 sau 8'b00000001
8'b1010_0111; //binar, echivalent cu 167 sau 8'b10100111
4'b10;        //binar, pe 4 biti, echivalent cu 2 sau 4'b0010 etc.
126;          //scriere in decimal
16'habcd;     //scriere in hexazecimal
```

Operatori

Descrierea comportamentală la nivelul fluxului de date, descrisă anterior, presupune în continuare cunoașterea schemei hardware la nivelul porților logice sau, măcar, expresia logică. Deși reprezintă o variantă mai simplă decât utilizarea primitivelor, nu este cea mai facilă.

Pentru a ușura implementarea, 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. Cu ajutorul acestora putem simplifica implementarea, apelând la construcții folosind limbajul de nivel înalt.

Tabelul de mai jos 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
& ~& ^ ~^ ^~ | ~|	Reducere	7
&& ||	Logici	8
?:	Condițional	9
{,}	Concatenare	

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.

```
wire signed[7:0] a, x, y;
assign x = a >>> 1;  // dacă bitul de semn al lui a este 0 bitul nou
                     //introdus este 0
                     // dacă 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 mulți biți și realizează operația logică între toți biții semnalului

```
wire[7:0] a;
wire x, y, z;
assign x = &a;  // realizeaza AND între toți biții lui a
assign y = ~&a; // realizează NAND între toți biții lui a
assign z = ~^a; // realizeaza XNOR între toți biții lui a,
                // echivalent cu ^~
```

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

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

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

Parametrizarea modulelor

Parameter

Cuvântul rezervat `parameter` este o construcție de limbaj în verilog care permite unui modul să fie reutilizat cu specificații diferite. Spre exemplu, un sumator poate fi parametrizat să accepte o valoare pentru numărul de biți care poate să fie configurată diferit de la o simulare la alta. Comportamentul lor este similar cu cel al argumentelor unor funcții în alte limbaje de programare cunoscute. Folosind `parameter` este declarată o valoare constantă, prin urmare este ilegală modificarea valorii acesteia în timpul simulării. De asemenea, este ilegal ca un alt tip de dată să aibă același nume ca unul dintre parametri.

```
parameter MSB = 7;       // MSB este un parametru cu valoarea constantă 7
parameter [7:0] number = 2'b11;    // o valoare de 2 biți este convertită
                                   // într-o valoare de 8 biți
```

O variabilă de tip parametru este vizibilă local, în modulul ce a fost declarată.

Construirea și instanțierea modulelor parametrizabile

Instanțierea modulelor a fost folosită și în laboratorul anterior pentru a invoca logica implementată într-un alt modul. În acel context, era necesar să cunoaștem dimensiunea semnalelor din interfață pentru a le potrivi cu variabilele conectate la instanță. În cazul în care un modul are dimensiunile porturilor parametrizate, acesta poate fi instanțiat cu valori particulare ale parametrilor (diferite de cele predefinite). Să considerăm ca exemplu un modul de mai jos:

```
module my_beautiful_module (out, a, b);
    output [7:0] out;
    input [3:0] a;
    input [4:0] b;
```

```
    …// some logic
endmodule
```

Pentru a instanția acest modul, vom avea nevoie de 3 variabile de 8, 4, respectiv 5 fire pe care le vom conecta astfel:

```
My_beautiful_module inst1(out, a, b);
```

Pe de altă parte, având modulul:

```
module my_beautiful_parameterized_module(out, a, b);
    parameter a_width = 4;
    parameter b_width = 5;
    parameter out_width = 8;

    output [out_width-1:0] out;
    input [a_width-1:0] a;
    input [b_width-1:0] b;

    …// some logic
endmodule
```

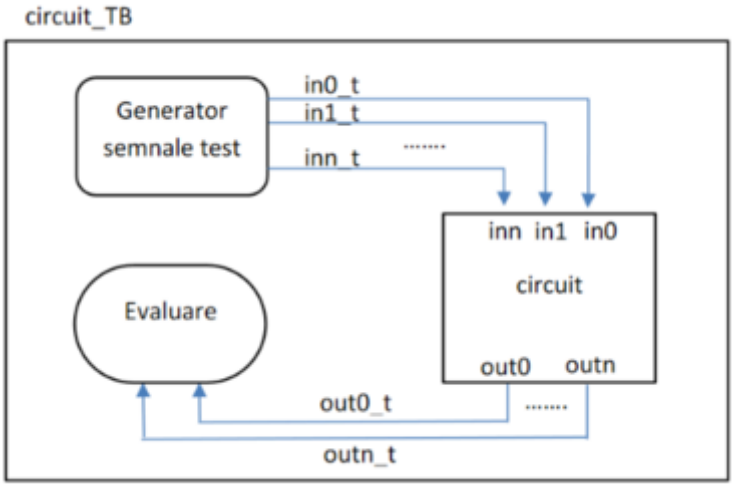
Îi putem utiliza logica fără a depinde de o dimensiune predefinită a semnalelor din interfață

```
wire [4:0] out1;
wire [4:0] out2;
wire [2:0] a;
wire [1:0] b;

my_beautiful_parameterized_module #(.a_width(3),
                                    .b_width(2),
                                    .out_width(5)) inst2(out, a, b);

// Sau, menținându-se ordinea parametrilor, doar prin specificarea noilor //
dimensiuni:

my_beautiful_parameterized_module #(3, 2, 5) inst3(out, a, b);
```

Testare

Pentru testarea unui modul folosind simulatorul se creează module speciale de test, în care, printre altele, se vor atribui valori intrărilor. Simularea permite detecția rapidă a erorilor de implementare și corectarea acestora.

Pentru a creea un modul de test și a-l simula puteți urma tutorialul de simulare aici, iar 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 începerea/activarea simulării și sunt folosite pentru inițializări și în module de test. 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
```

Blocurile `initial` nu sunt sintetizabile, fiind folosite doar în simulări.

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.

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*.

```
    a = 1; b = 4;

    $display("suma=%d", a+b);
```

Exerciții

Pentru implementarea exercițiilor se vor utiliza scheletele de cod din arhiva laboratorului. Scheletele de cod conțin deja un proiect Xilinx ISE, iar unele din ele și un modul de testare. Urmăriți cerința și zonele marcate cu TODO.

1. **(4p)** Sumatorul pe 4 biți. Implementare și testare.
   1. (1p) Implementați și simulați un **sumator pe 4 biți**, cu două intrări și o ieșire
      - *Hint*: Utilizați atribuirea continuă pentru implementare
      - *Hint*: Atenție la dimensiunea semnalelor de ieșire
   2. (2p) Folosind tutorialul de simulare, implementați un modul de test care să stimuleze sumatorul în cât mai multe situații posibile
      - *Hint*: Variabilele pe care le atribuim în modulul de test vor fi de tip *reg*.
      - *Hint*: Testați atât situații obișnuite de adunare, cât și situații speciale (ex. carry = 1)
      - *Hint*: Consultați forma de undă pentru a determina corectitudinea implementării.Verificați corectitudinea sumatorului vizualizând semnalele în baza 10.
   3. (1p) Analizând implementarea din Laboratorul 1 și varianta curentă, analizați cele două tipuri de implementări.
      - *Hint*: Comparați puncte forte, puncte slabe pentru fiecare din cele două variante
2. **(3p)** Implementați și simulați un **sumator parametrizat pe n biți**, cu două intrări și o ieșire. Parametrizarea se va efectua asupra dimensiunii variabilelor.
   - *Hint*: De câți parametri este nevoie? Observați dependența între dimensiunea variabilelor de între și cea de ieșire.
   - *Hint*: Luând exemplul modulului de test implementat la exercițiul 1, instanțiați un sumator pe 6 biți și adăugați stimuli corespunzători pentru a-i testa întreaga plajă de valori.
3. **(3p)** Implementați și simulați un **comparator** pe 4 biți. Acesta are două intrări și 3 ieșiri (pentru mai mic, egal și mai mare).
   - *Hint*: Unei variabile îi poate fi atribuită valoarea unei expresii logice.
   - *Hint*: Considerând experiența exercițiului 2, există vreo posibilitate să parametrizați comparatorul?
4. **(2p)** 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.
   1. (1p) Implementați multiplexorul folosind ecuația logică dedusă din tabelul de adevăr.
   2. (1p) Implementați multiplexorul folosind operatorul condițional '?'
      - *Hint*: Operatorul poate apărea de mai multe ori într-o expresie. ex: `assign x = (a == 0) ? 1 : ( (a == 1) ? : 2 : 0 );`

Resurse

- Schelet de cod
- PDF laborator
- Soluție laborator

From:
http://ocw.cs.pub.ro/courses/ - **CS Open CourseWare**

Permanent link:
http://ocw.cs.pub.ro/courses/ac-is/lab/lab02

Last update: **2023/10/21 12:51**