S4. Utilizarea procedurilor structurate

Construirea blocurilor *always* şi *initial* în Verilog

1. Scopul laboratorului

Construirea logicii secvențiale sincrone în Verilog

2. Desfășurarea laboratorului

2.1 Blocuri *always* şi *initial*

Modelarea comportamentală este oferită prin două structuri de limbaj:

- blocuri *always* şi
- blocuri *initial*

Blocurile initial sunt executate o singură dată, la începutul simulării Verilog însă suportul oferit de uneltele de sinteză pentru integrarea lor în hardware nu este consistent (cu precădere pentru platformele FPGAs).

Execuţia unui bloc always este declanşată de oricare dintre evenimentele specificate în lista de senzitivitate a blocului, listă specificată astfel: `always @ ( <sensitivity list> )`.

Pentru un semnal din listă, oricare tranziţie declanşează execuţia blocului always. Dacă semnalul este precedat de un specificator de front, posedge sau negedge, atunci frontul crescător, respectiv descrescător, al său declanşează execuţia blocului. Dacă un semnal are specificator de front, toate semnalele din listă trebuie să le aibă precizate. În listă, evenimente multiple sunt separate de cuvântul rezervat ***or*** sau de virgulă. Dacă blocul **initial** sau **always** conţine mai multe instrucţiuni, acestea sunt încadrate între **begin** şi **end**.

2.2 Atribuiri procedurale

Sunt atribuiri executate în interiorul unui bloc **always** sau **initial** şi, spre deosebire de atribuirile continue, sunt evaluate doar la execuţia blocului.

Partea stângă a unei atribuiri procedurale poate fi:

- un semnal declarat cu tipul reg,
- o variabilă de tip întreg,
- o variabilă de tip real,
- o variabilă de tip time,
- un bit sau o selecţie part-select a cazurilor de mai sus, sau
- o concatenare a cazurilor de mai sus

Observație:
Semnalul folosit în partea stângă a atribuirii procedurale trebuie să fie declarat de tipul **reg**. Dacă partea dreaptă a unei atribuiri procedurale are mai puţini biţi decât cea stângă, va fi extinsă cu biţi de 0.

Există două tipuri de atribuiri procedurale:

- ***Cu blocare***, care folosesc ca simbol e atriburie **=** şi având forma **<left_hand_side> = <expression>,** respectiv
- ***Fără blocare***, care folosesc ca simbol de atriburie **<=** si avand forma **<left_hand_side> <= <expression>**

Important: Un bloc *initial* sau *always* poate conţine fie doar atriburiri cu blocare, fie doar fără blocare, nu o combinaţie a celor două **[Cumm00]**.

Pentru atriburiri cu blocare, evaluarea părţii drepte şi actualizarea părţii stângi se realizează imediat, execuţia continuând cu următoarea instrucţiune doar ulterior.

Pentru atriburie fără blocare, părţile drepte ale tuturor atribuirilor din bloc sunt evaluate secvenţial, dar actualizările părţilor drepte corespunzătoare este amânată la finalul execuţiei blocului.

2.2.1 Atribuiri cu blocare versus fără blocare

Următoarele fragmente ilustrează cele două tipuri de atribuiri:

```
always @ (*) begin
        blk_b = blk_a ;
        blk_c = blk_b ;
        blk_d = blk_c ;
end
```

```
always @ (posedge clk) begin
     nonblk_b <= nonblk_a ;
     nonblk_c <= nonblk_b ;
     nonblk_d <= nonblk_c ;
end
```

Atribuirile lor fizice echivalente sunt prezentate mai jos:

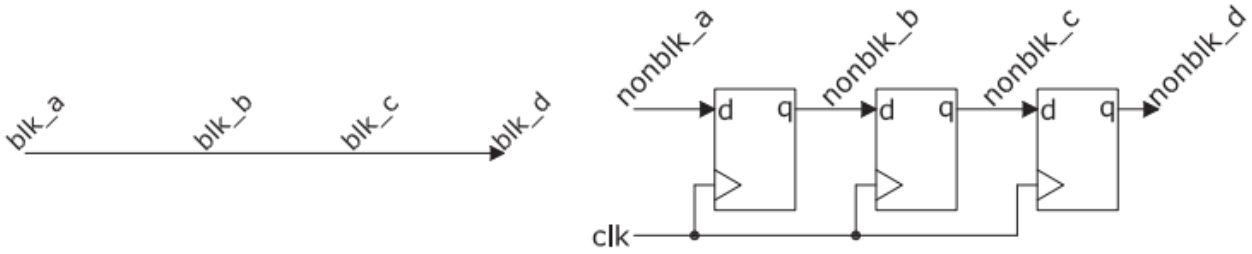

Fig. 1 - Blocking vs non-blocking elements

Blocul din stânga are atriburie combinaţională (folosind doar fire de legătură) pe când a celui din dreapta are atriburie secvenţială sincronă (un registru de deplasare).

2.2.2 Utilizarea atribuirilor procedurale

Sistemele secvenţiale sincrone declanşate pe front sunt modelate prin blocuri always având în lista de senzitivitate semnalul de ceas şi utilizând doar atribuiri fără blocare. Semnalul de tact va avea precizat specificatorul de front.

O potenţială intrare sincronă de reset este adăugată în lista de senzitivitate prin negedge rst_b sau posedge rst, pentru o linie de reset activă la 0, respectiv la 1.

Notă: În acest laborator, semnalele active la 0 sunt marcate cu sufixul **_b**.

Fragmentul de cod de mai jos descrie un bistabil de tip D cu intrarea de reset asincronă activă la 0, *rst_b*:

```
always @ (posedge clk, negedge rst_b) begin
        if (!rst_b) q <= 1'd0 ;
         else
        q <= d ;
end
```

Sistemele secvenţiale declanşate pe nivel sunt construite cu blocuri always conţinând doar atribuiri fără blocare şi având semnalul de activare specificat în lista de senzitivitate. Nu se va folosi specificator de front. O posibilă intrare asincronă de reset se adaugă la lista de senzitivitate, de asemenea fără specificator de front.

Fragmentul de cod de mai jos descrie un latch de tip T cu un semnal de reset asincron, activ la 1, *rst:*

```
always @ (en, d, rst) begin
        if (rst_b) q <= 1'd0 ;
        else if (en) q <= d ^ q ;
end
```

Structurile combinaţionale pot fi modelate fie utilizând atribuiri continue, fie prin blocuri always în care sunt utilizate doar atribuiri cu blocare. În lista de senzitivitate vor fi adăugate toate semnalele care apar în părţile drepte ale atribuirilor din bloc sau în codiţiile utilizate în blocul always. În locul adăugirii tuturor semnalelor necesare în lista de senzitivitate, Verilog permite utilizarea simbolului * ca listă de senzitivitate.

Fragmentul de cod de mai jos descrie un multiplexor cu o linie de selecţie:

```
always @ (*) begin
        if (sel) o = d1 ;
        else o = d0 ;
end
```

Instrucţiunile condiţionale au următorul format:

```
if (<condition>)
    <statement_true>;
else
    <statement_false>
```

Ramura ***else*** este opţională. Expresia condition este evaluată şi dacă este diferită de 0 se execută instrucţiunea ***statement_true***, altfel se execută ***statement_false***, dacă ramura este inclusă.

Dacă o ramură cuprinde mai multe instrucţiuni, acestea vor fi mărginite de construcţia ***begin ...end.***

2.3 Studiu de caz

Se consideră un registru cu încărcare paralelă pe 8 biţi cu reset asincron, activ la 0 (stânga) şi, respectiv, cu reset sincron, activ la 1 (dreapta):

```
module reg8_async_rst_b (
input clk,
input rst_b,
input [7:0] d,
output reg [7:0] q );
always @ (posedge clk, negedge rst_b) begin
          if (! rst_b) q <= 8'd0 ;
          else q <= d ;
endmodule
```

```
module reg8_sync_rst (
 input clk,
 input rst,
 input [7:0] d,
 output reg [7:0] q );
 always @ (posedge clk)
           if (rst)  q <= 8'd0 ;
           else  q <= d ;
 endmodule
```

În partea stângă, ***rst_b*** afectează registrul indiferent de momentul activării, deci este inclusă în lista de senzitivitate. În dreapta, ***rst*** afectează registrul doar când este activă pe frontul de tact nefiind inclusă în lista de senzitivitate.

Important: Orice intrare sincronă a registrului este tratată similar liniei de reset sincrone prin evaluarea valorii sale în corpul blocului ***always*** fără includerea sa în lista de senzitivitate.

2.4 Instrucţiunea case

Reprezintă un mecanism de decizie multiplă care verifică potrivirea unei expresii selector în raport cu mai multe ramuri, având formatul:

```
case (<expression>)
        <value_1> : <statement>;
        ....
        <value_n> : <statement>;
        default : <statement_default>;
endcase
```

Selectorul ***expression*** este comparat cu valorile executându-se instrucţiunea corespunzătoare primei potriviri. Verificarea se face ordonat, începând de la *value_1*. Dacă nu s-a găsit nicio potrivire şi este inclus cazul default, va fi executat instrucţiunea acestuia.

Instrucţiunea ***casex*** şi ***casez***, cu acelaşi format, tratează biţii nedefiniţi, respectiv, în impedanţă ridicată, ca biţi don't cares, care nu influenţează comparaţia selectorului. Simbolul **?** poate fi folosit de asemenea, pentru a marca poziţii binare **don't care**.

Exerciţiu: Implementaţi un decodificator 2-la-4 cu intrare de enable şi ieşiri active la 0.

Soluţie

```
module dec_2x4 (
input [1:0] s,
input e,
output reg [3:0] y
);
always @ ( * )
casez ( { e , s } )
3'b100 : y = 4'b1110 ;
3'b101 : y = 4'b1101 ;
3'b110 : y = 4'b1011 ;
3'b111 : y = 4'b0111 ;
3'b0?? : y = 4'b1111 ;
endcase
endmodule
```

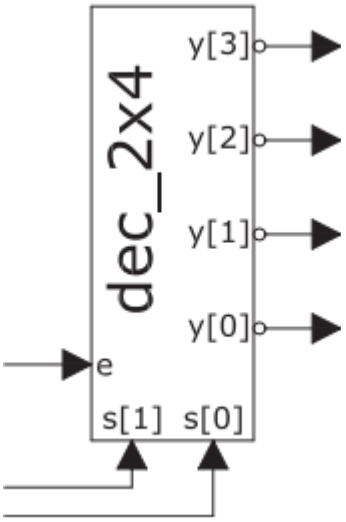

Fig. 2 - 2 to 4 Line Decoder

Ultima ramură maschează intrarea s prin simboluri don't care.

2.5 Afişarea informaţiilor de simulare

Apelul de sistem (system task) **$display()** tipăreşte date în consola simulatorului, având formatul: **$display(expression_1, … expression_n)**;. Argumentele numerice sunt tipărite în zecimal. În argumentele şir de caractere (încadrate în ghilimele), sunt recunoscute un număr de specificatori de format:

- %b - valoare binară
- %c –caracter ASCII, pe 8 biţi
- %d – valoare zecimală, %0d specifică lăţimea minimă
- %e, %f şi %g – valori reale
- %h – valoare hexazecimală
- %m – nume ierarhic de modul
- %o – valoare octală
- %s – şir de caractere
- %t – timpul de simulare furnizat de apelul sistem $time
- %u – date neformatate folosind 2 valori (1 şi 0)
- %z – date neformatate folosind 4 valori (1, 0, z şi x)

Apelul **&display()** recunoaşte următoarele secvenţe în argumentele şiruri de caractere:

- \n – linie nouă
- \t – tabulare
- \\ – caracter backslash
- \” – caracter ghilimele
- %% - caracter procent

Apelul system **$monitor** cu acelaşi format ca **$display**, tipăreşte date formatate ori de câte ori unul din argumente se modifică pe parcursul simulării. Se va apela o singură dată iar funcţionarea sa poate fi inhibată prin apelul system **$monitoroff,** respectiv poate fi reactivată prin apelul **$monitoron**.

2.6 Construcţii repetitive

Limbajul Verilog oferă 4 construcţii repetitive: ***forever***, ***repeat***, ***while*** şi ***for***. Construcţia forever are formatul **forever statement;** şi execută instrucţiunea statement indefinit. În acest laborator, forever va fi folosită doar pentru generarea tactului în fişiere testbench. În fragmentul următor se construieşte un semnal de ceas cu factor de umplere de 50 % şi perioadă de 100ns :

```
reg clk ;
initial begin
clk = 1'd0 ;
forever #50 clk = ~clk ;
end
```

Construcţia repetitivă este folosită într-un bloc ***initial*** unde semnalul este mai întâi iniţializat şi apoi basculat continuu la fiecare 50 ns.

Construcţia repeat are formatul **repeat (<number_of_times>) statement ;** şi execută instrucţiunea un număr dat de ori fiind folosită, de asemenea, în testbench-uri. Următorul cod tipăreşte toate numerele dintre 60 şi 63, inclusiv, în zecimal şi binar:

```
reg [5:0] n ;
initial begin
n = 6'd60 ;
repeat (4) begin
    $display ("%d(10)  = %b(2)" , n, n) ;
    n = n+1 ;
    end
end
```

Construcţia while are formatul **while (condition) statement ;** şi execută instrucţiunea cât timp expresia *condition* este adevărată.

Similar, construcţia for, cu formatul **for (loop_init ; loop_condition ; loop_update) statement ;** execută instrucţiunea cât timp condiţia de repetiţie este adevărată. Această structură repetitivă oferă facilităţi de iniţializare şi actualizare. Fragmentul de cod de mai jos tipăreşte toate numerele între limitele 90 şi 99, inclusiv, în zecimal şi binar:

```
reg [6:0] n ;
initial begin
for ( n= 'd90 ; n < 100 ; n = n+1 )
#50 $display ("%d (10) = %b (20)", n, n);
end
```

3. Referințe bibliografice

[Stro05] L. Strozek. Verilog Tutorial - Edited for CS141. [Online]. Available: https://wiki.eecs.yorku.ca/course_archive/2013-14/F/3201/_media/verilog-tutorial_harvard.pdf (Last accessed 20/07/2016).

[Cumm00] C. Cummings. Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf (Last accessed 17/04/2016).