14.10.2024

Arhitectura calculatoarelor
(laborator 3 - S3)

Modelarea comportamentală este oferită prin două structuri de limbaj:

- blocuri `always`, și
- blocuri `initial`

Blocurile always și initial

Blocurile `initial` sunt executate o singură dată, la începutul simulării.

Execuția unui bloc `always` este declanșată de oricare dintre evenimentele specificate în lista de senzitivitate a blocului, listă specificată prin:
`always @ ( <sensitivity_list> )`.

Tranziția oricărui semnal din lista de senzitivitate declanșază reexecuția blocului `always`. Dacă semnalul este precedat de un specificator de front, `posedge` sau `negedge`, doar respectivul frontul va declanșa reexecuția blocului. Nu pot fi combinate în aceeași listă de senzitivitate semnale cu și fără specificatori. Evenimentele din listă sunt separate prin `or` sau prin virgulă. Dacă blocul `initial` sau `always` conține mai multe instrucțiuni, acestea sunt încadrate între `begin` și `end`.

Atribuiri procedurale

Atribuirile procedurale se execută în interiorul unui bloc `always` sau `initial` și, spre deosebire de cele atribuirile continue, sunt evaluate doar la execuția blocului de rezidență.

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 expresie *part-select* a cazurilor de mai sus, sau
- o concatenare a cazurilor de mai sus

Important: Semnalul folosit în partea stângă a atribuirii procedurale trebuie să fie declarat de tipul `reg`.

Dacă partea dreaptă a unei atribuiri procedurale are mai puțini biți decât cea stângă, aceasta va fi extinsă cu 0-uri în biții msb.

Există două tipuri de atribuiri procedurale:

- *cu blocare*: folosesc ca simbol de atribuire = cu forma `<left_hand_side> = <expression>`, respectiv
- *fără blocare*: folosesc ca simbol de atribuire <= cu forma`<left_hand_side> <= <expression>`

Important: Pentru componente combinaționale, blocul `always` va folosi doar atribuiri cu blocare!

Important: Pentru componente secvențiale sincrone, blocul `always` va folosi doar atribuiri fără blocare!

Utilizarea atribuirilor procedurale

Sistemele secvențiale sincrone declanșate pe front (componente de tip flip-flop) sunt modelate prin blocuri `always` în care sunt utilizate atribuiri fără blocare. Lista de senzitivitate include semnalul de tact (*clk*), precedat de specificatorul de front, și, eventual, un semnal asincron de inițializare, precedat și el de specificatorul de front.

Notă: La activitățile practice de la această disciplină, 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 secvențiale declanșate pe nivel (latch) sunt construite cu blocuri `always` conținând doar atribuiri fără blocare. Lista de senzitivitate conține semnalul de activare și, eventual, un semnal de inițializare. Nu se folosesc specificatori 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) q <= 1'd0;
  else if (en) q <= d ^ q;
end
```

Structurile combinaționale pe lângă atribuirile continue (`assign`), mai pot fi modelate prin blocuri `always` conținând doar atribuiri cu blocare. Lista de senzitivitate include toate semnalele a căror modificare necesită reevaluarea blocului `always`. Tipic, aceste semnale din lista de senzitivitate include semnalele care apar în părțile drepte ale atribuirilor sau în expresiile de tip condiții din bloc. În locul adăugării 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 selecție:

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

Instrucțiuni condiționale

Instrucțiunile *condiționale* au următorul format:

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

Ramura `else` este opțională. Expresia `condition` este evaluată și dacă este diferită de 0 se execută instrucțiunea *statement_true*, altfel se execută *statement_else*, dacă ramura este inclusă.

Dacă o ramură cuprinde mai multe instrucțiuni, acestea vor fi incluse într-un bloc `begin ... end`.

Studiu de caz

Registru cu încărcare paralelă pe 8 biți cu reset asincron, activ la 0 (stânga) și, respectiv, cu reset sincron, activ la 1 (dreapta):

```
1  module reg8_async_rst_b (
2    input clk,
3    input rst_b,
4    input [7:0] d,
5    output reg [7:0] q
6  );

8    always @ (posedge clk, negedge rst_b)
9      if (! rst_b) q <= 8'd0;
10     else q <= d;
11 endmodule
```

```
1  module reg8_sync_rst (
2    input clk,
3    input rst,
4    input [7:0] d,
5    output reg [7:0] q
6  );

8    always @ (posedge clk)
9      if (rst) q <= 8'd0;
10     else q <= d;
11 endmodule
```

Important: pentru componentele secvențiale sincrone, intrările sincrone nu sunt incluse în lista de senzitivitate, spre deosebire de intrările asincrone.

Astfel, în partea stângă, intrarea asincronă *rst_b* este inclusă în lista de senzitivitate în timp ce intrarea sincronă *rst* din partea dreaptă nu este inclusă.

Instrucțiunea case

Mecanism de decizie multiplă care verifică potrivirea (*matching*) unei expresii selctor în raport cu mai multe ramuri, având formatul:

```
case (<expression>)
    <case_value_1> : <statement_1>;
    ...
    <case_value_n> : <statement_n>;
    default : <statement_default>;
endcase
```

Selectorul *expression* este comparat cu cele *n case_values*, executându-se instrucțiunea corespunzătoare primei potriviri. Verificarea se face ordonat, începând de la *case_value_1*. Dacă nu s-a găsit nicio potrivire și este inclusă clauza `default`, va fi executată instrucțiunea acesteia.

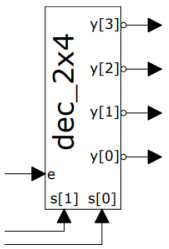
Dacă se dorește specificarea unor poziții binare care nu vor influența potrivirea, se va folosi simbolul ? în expresia binară a *case_value*-urilor.

Exercițiu rezolvat

Exercițiu: Implementați un decodificator 2-la-4 cu intrare de enable și ieșiri active la 0

Soluție:

```
1  module dec_2x4 (
2    input [1:0] s,
3    input e,
4    output reg [3:0] y
5  );
6    always @ ( * )
7      casez ({e, s})
8        3'b100 : y = 4'b1110;
9        3'b101 : y = 4'b1101;
10       3'b110 : y = 4'b1011;
11       3'b111 : y = 4'b0111;
12       3'b0?? : y = 4'b1111;
13     endcase
14 endmodule
```



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

Afișarea informațiilor de simulare

`$display()` tipărește informații în consola, având formatul:
`$display("format", expr_1, ... , expr_n);`
În șirul *format*, sunt recunoascute un număr de specificatori de format:

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

Apelul `$display()` recunoaște următoarele secvențe în șirul *format*:

- `\n` - linie nouă
- `\t` - tabulare
- `\\` - caracter backslash
- `\"` - caracter ghilimele
- `%%` - caracter procent

`$monitor` cu același format ca `$display`, tipărește informații formatate reluând afișarea ori de câte ori unul din semnalele tipărite își modifică valoarea pe parcursul simulării.

Construcții repetitive

Verilog oferă 4 construcții repetitive: `forever`, `repeat`, `while` și `for`.

Construcția `forever` are formatul `forever statement;` și execută instrucțiunea `statement` indefinit. La activitățile practice de la această disciplină, `forever` va fi folosit pentru generarea tactului în fisiere testbench. În fragmentul următor se construiește un semnal de ceas cu factor de umplere de 50% și perioadă de 100ns:

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

Construcția repetitivă este folosită într-un bloc `initial` unde semnalul este mai întâi inițializat și apoi basculat continuu la fiecare 50ns.

Construcția `repeat` are formatul `repeat (<number_of_times>) statement;` și execută instrucțiunea un număr dat de ori fiind folosită, de asemenea, în testbench-uri. Următorul cod tipărește toate numerele dintre 60 și 63, inclusiv, în zecimal și 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
```

Construcția `while` are formatul `while (condition) statement;` și execută instrucțiunea cât timp expresia *condition* este adevărată.

Similar, construcția `for`, cu formatul `for (loop_init; loop_condition; loop_update) statement;`, execută instrucțiunea cât timp condiția de repetiție este adevărată. Această structură repetitivă oferă facilități de inițializare și actualizare. Fragmentul de cod de mai jos tipărește toate numerele între limitele 90 și 99, inclusiv, în zecimal și binar:

```
reg [6:0] n;
initial begin
  for (n = 'd90; n < 100; n = n+1)
    #50 $display("%d(10) = %b(2)", n, n);
end
```