Laboratorul 3 - Circuite combinaționale - descrierea comportamentală

În laboratoarele anterioare am studiat descrierea structurală, folosind primitive, precum și descrierea comportamentală, folosind atribuiri continue. Am remarcat faptul că generalizarea modulelor folosind parametri conduce la o capacitate de reutilizare mai mare, cu schimbări minime. Cu toate acestea, soluțiile prezentate nu sunt pretabile funcțiilor complexe, întrucât ele devin complicat de implementat sau de urmărit, în momentul când este găsit un bug în cod.

Laboratorul curent va prezenta elementele Verilog folosite pentru descrierea comportamentală la nivel procedural, ce se vor axa în continuarea pe conceptul de "**ce face** circuitul". Se folosesc construcții de nivel înalt, similare altor limbaje de programare întâlnite până în prezent, prin care putem descrie mai facil algoritmul care calculează ieșirile circuitului.

Structura limbajului Verilog - continuare

Tipul reg

În primele laboratoare a fost prezentat tipul *wire* pentru reprezentarea semnalelor din interiorul modulelor. Porturile unui modul erau *wires*, la fel și semnalele de legătură dintre instanțele primitivelor și porților. Deoarece acestea realizează conexiuni, nu au o stare și nu li se pot atribui valori. Pentru a putea reține stări/valori și a face atribuiri avem nevoie de tipul **reg**.

Declararea variabilelor de tip reg se poate face într-un mod similar variabilelor de tip wire, cum este exemplificat și mai jos:

Exemplu declarare variabilă de tip reg

```
reg x;
reg[7:0] m;
reg[0:4] n;
reg[7:0] a [3:0]; // array multidimensional cu 4 elemente de 8 biti
```

Declararea unei variabile de tip reg (deci o variabilă de tip registru în Verilog) **nu** implică sinteza unui registru hardware!

Construcții de control

În afară de folosirea atribuirilor continue, circuitele pot fi descrise comportamental și prin blocuri always. În interiorul acestora se pot folosi construcții de limbaj similare celor din limbajele procedurale.

Cod Verilog	Cod C

`if (sig == 0) begin`
`    a = 2;`
`end else if (sig == 1 ) begin`
`    a = 1;`
`end else begin`
`    a = 0;`
`end`
	`if (sig == 0) {`
	`    a = 2;`
	`} else if (sig == 1 ) {`
	`    a = 1;`
	`} else {`
	`    a = 0;`
	`}`

`case (sig)`
`    'b0: a = 2;`
`    'b1: a = 1;`
`    default: a = 0;`
`endcase`
	`switch (sig) {`
	`    case 0: a = 2; break;`
	`    case 1: a = 1; break;`
	`    default: a = 0;`
	`}`

`for (i = 0; i < 10; i = i + 1) begin`
`    a = a / 2;`
`end`
	`for(i = 0; i < 10; i = i + 1) {`
	`    a = a / 2;`
	`}`

`i = 0;`
`while (i < 10) begin`
`   a = a / 2;`
`   i = i + 1;`
`end`
	`i = 0;`
	`while(i < 10) {`
	`   a = a / 2;`
	`   i = i + 1;`
	`}`

`repeat (10) begin`
`   a = a / 2;`
`end`	

Construcțiile de repetiție sunt sintetizabile doar dacă ele au un număr **fix** de iterații. Trebuie acordată o deosebită atenție în momentul în care se implementează structuri repetitive utilizând *while* sau *repeat*, acestea fiind mult mai susceptibile la greșeli de implementare care vor genera, în final, un cod simulabil dar nesintetizabil.

Blocul always@

Blocurile *always* descriu un comportament ciclic, codul acestora fiind executat în continuu. Prezența operatorului @ face ca blocul să se “execute” doar la apariția unor evenimente. Evenimentele sunt reprezentate de modificarea unuia sau mai multor semnale.

În cadrul acestui laborator ne axăm doar pe descrierea circuitelor combinaționale, și vom folosi doar blocuri `always @(*)`, unde (*) se numește **sensitivity list**. Folosirea wildcard-ului * implică “execuția” blocului *always* la orice eveniment de modificare a oricărui semnal folosit în cadrul blocului.

Instrucțiunile din blocul *always* sunt încadrate între cuvintele cheie `begin` și `end` și sunt “executate” secvențial atunci când blocul este activat.

```
always @(*) begin
```

```
    b = 0;      // registrul b este inițializat cu 0 la orice
                // modificare a unui semnal
    c = b ^ a; // registrul c va primi valoarea expresiei din dreapta
                // la orice modificare a unui semnal (nu doar a sau b)
end
```

În locul wildcard-ului, *, sensitivity list-ul poate conține o listă de semnale la modificarea cărora blocul *always* să fie activat. Acestea se declară prin numele lor, folosind or sau , pentru a le separa.

Este foarte important ca lista de semnale dată unui bloc *always@* să fie **completă**, altfel nu toate combinațiile de intrări sunt acoperite și unele variabile pot rămâne neatribuite corespunzător. Pentru a evita astfel de erori se recomandă folosirea wildcard-ului *.

Exemplu declarare listă de senzitivitate

```
always @(a or b or c) // sintaxa Verilog-1995
always @(a, b, c)     // sintaxa Verilog-2001, 2005
always @(a, b or c)   // permis dar nerecomandat, îngreunează
                      // lizibilitatea codului
always @(*)           // toate semnalele din modul (intrări +
                      // wires declarate în modul)
```

În modulul următor care implementează o poartă xor, ieșirea out se va schimba doar când semnalul a se schimbă, ceea ce duce la un comportament incorect care nu ia în considerare și schimbarea lui b. În plus, modulul generat nu va fi unul combinațional, deoarece este nevoie de memorie pentru a menține starea ieșirii atunci când b se modifică.

```
module my_xor(output reg out, input a, input b);
    always @(a) begin
        out = a ^ b;
    end
endmodule
```

Nu se pot instanția primitive și module în interiorul blocurilor *always* și *initial*.

În cadrul blocurilor *initial* și *always* atribuirile pot fi de două feluri, cu semantici diferite:

- **Atribuiri blocante**: sunt folosite pentru descrierea logicii combinaționale (porți logice). Atribuirile blocante sunt interpretate ca executându-se secvențial, semantica fiind identică cu cea din limbajele de programare procedurale.
- **Atribuiri non-blocante**: sunt folosite pentru descrierea logicii secvențiale (ne vom întâlni cu ele în laboratorul următor).

În Verilog putem declara variabile de tipul *reg* și *wire* pentru a crea circuite combinaționale. Atunci când se fac atribuirile există totuși următoarele restricții:

- Pe o variabilă de tip **wire** nu putem face decât **atribuiri continue** (*assign*). Acestea trebuie să fie

în exteriorul blocurilor always sau initial. Valoarea atribuită poate fi o constantă, o expresie, valoarea unui fir sau a unui registru.
- Pe o variabilă de tip **reg** putem face doar **atribuiri blocante/non-blocante**. Acestea trebuie să fie **în interiorul** unui bloc initial sau always. Valoarea atribuită poate fi o constantă, o expresie, valoarea unui fir sau a unui registru.

Exemplu sumator descris comportamental, folosind always@

```
module my_module(
    output reg[4:0] o, // o trebuie sa fie reg pentru a o putea atribui
                       // în blocul always
    input[3:0] a, b);

    reg[2:0] i;        // poate fi maxim 7; noi avem nevoie de maxim 4
    reg c;             // ținem minte transportul

    always @(*) begin
        i = 0; // la orice modificare a intrărilor, i va fi inițial 0
        c = 0; // transportul initial este 0

        // toți biții lui o sunt recalculati la modificarea intrărilor
        for (i = 0; i < 4; i = i + 1) begin
            {c, o[i]} = a[i] + b[i] + c;
        end
    end
endmodule
```

Pentru ca un bloc *always* să fie sintetizat într-un circuit combinațional este necesar ca orice “execuție” a blocului să atribuie cel puțin o valoare pentru fiecare ieșire a modulului.

Bineînțeles, acea valoare nu poate fi calculată pe baza ieșirilor sau valorilor anterioare ale variabilelor din interiorul modulului. Asta ar însemna că este necesară o memorie pentru a menține acele valori, transformând circuitul într-unul secvențial.

Nu este recomandată declararea variabilelor de tip reg în interiorul blocurilor de tip *always* sau *initial*. În Verilog ea este totuși posibilă, dacă blocul respectiv are un tag asociat. Variabila astfel declarată va avea vizibilitate doar în interiorul blocului.

Exemplu declarare variabilă de tip reg în interiorul blocului always

```
always @(*) begin :my_tag
    reg x;
    x = in1 ^ in2;
```

```
end
```

Exerciții

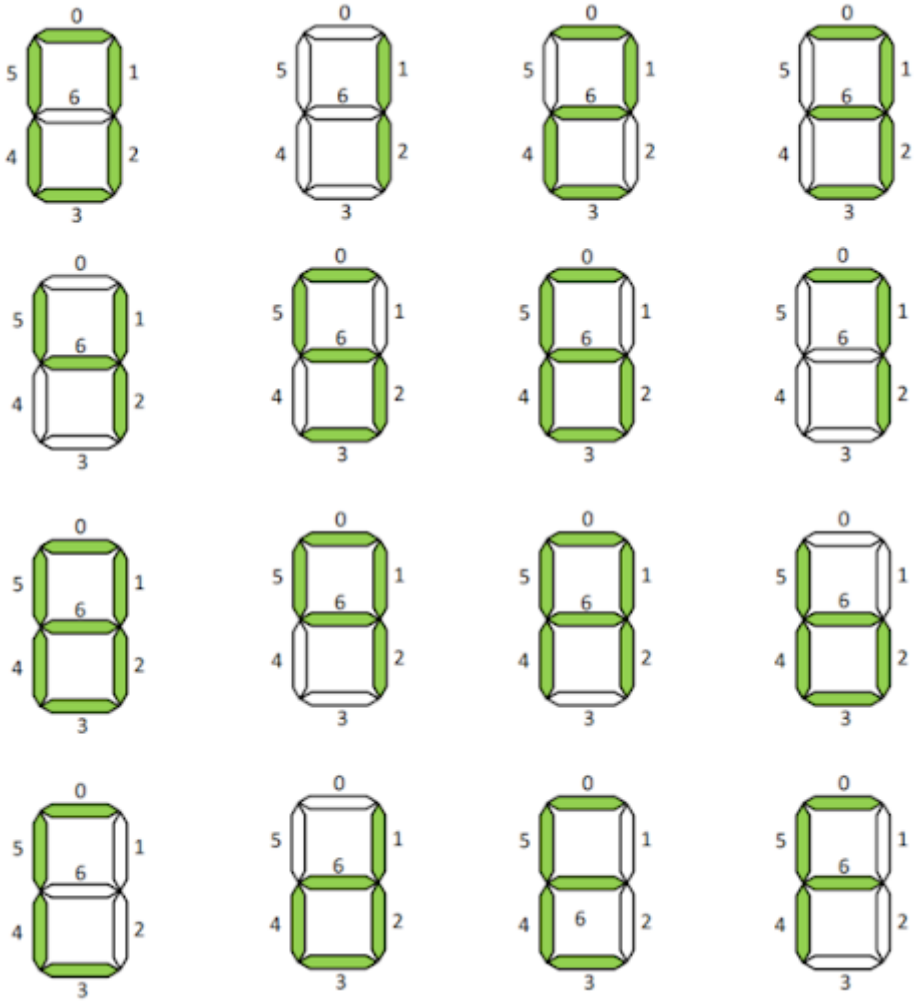
Pentru implementarea exercițiilor se vor utiliza scheletele de cod din arhiva laboratorului. Scheletele de cod conțin deja un proiect Xilinx ISE, iar unele din ele și un modul de testare. Urmăriți cerința și zonele marcate cu TODO.

1. **(4p)** Implementați și simulați un **multiplicator pe 4 biți** fără a folosi operatorul * (înmulțire).
   - *Hint*: Folosiți convenția Verilog pentru interfața modulului. Câți biți are ieșirea?
   - *Hint*: Înmulțiți pe hârtie, în baza 2, numerele 1001 și 1011. Transpuneți în limbajul Verilog algoritmul folosit.
2. **(2p)** Implementați și simulați un modul de **afișaj cu 7 segmente** pentru numere în baza 10. Descrierea sumară a funcționalității acestuia se poate regăsi în Anexă.
   - *Hint*: Există o ieșire validă pentru fiecare intrare? Nu uitați de cazul `default`.
   - *Hint*: Se vor testa doar cifrele de la 0 la 9.
3. **(4p)** Implementați o **unitate aritmetico-logică** simplă (UAL), pe **4** biți, cu 2 operații: adunare și înmulțire. Folosiți o intrare de selecție de 1 bit pentru a alege între cele două operații astfel: 0 - adunare, 1 - înmulțire.
   - *Hint*: Este necesar să creați un nou proiect. Se poate folosi tutorialul.
   - *Hint*: Câți biți au ieșirea sumatorului și a multiplicatorului? Dar a UAL-ului?
   - *Hint*: Pentru selecția dintre ieșirea sumatorului și cea a multiplicatorului se poate folosi atribuirea continuă sau se poate implementa un modul multiplexor 2:1
   - *Hint*: Pentru testarea UAL trebuie creat un scenariu de test, folosind tutorialul. Presupunând că sumatorul și multiplicatorul sunt testate temeinic la exercițiile anterioare, creați un scenariu relevant funcționării UAL-ului ca un ansamblu.
4. **(3p)**: Pentru o utilizare mai generală, implementați un UAL cu operatori cu dimensiune variabilă.
   - *Hint*: Pentru a-l implementa, este necesară implementarea unui multiplicator parametrizat - atenție la dimensiunea semnalelor!

Anexă

Seven-segment display

Acest tip de dispozitiv de afișare este alcătuit, așa cum vă puteți da seama și din numele acestuia, din 7 segmente controlabile individual. Așadar, putem aprinde (1 logic) sau stinge (0 logic) fiecare segment din componența acestuia.



Reprezentarea numerelor în format hexazecimal considerând un afișaj cu 7 segmente

De exemplu, dacă dorim să afișăm cifra 6, ieșirea noastră va avea valoarea `7'b111_1101`.

Pentru mai multe detalii asupra acestui tip de modul, consultați pagina de Wiki.

Resurse

- Schelet de cod
- PDF laborator
- Soluție laborator