Laboratorul 1 - Introducere in Verilog. Descrierea structurala

Verilog

În cadrul laboratorului de Arhitectura Calculatoarelor vom studia un limbaj de descriere a hardware-ului (*eng. Hardware Description Language* - ***HDL***) numit **Verilog**. Îl vom folosi pe tot parcursul laboratorului pentru a implementa noțiuni legate de arhitectura calculatoarelor. Limbajele de descriere a hardware-ului sunt folosite în industrie pentru proiectarea și implementarea circuitelor digitale. Cele mai folosite limbaje de descriere a hardware-ului sunt **Verilog** și **VHDL**.

Deși din punct de vedere sintactic se aseamănă foarte mult cu un limbaj de programare de uz general (C/C++/Java), trebuie ținut cont că instrucțiunile nu se execută secvențial, ca pe un procesor. Ținta unui cod scris în Verilog este implementarea sa pe un **FPGA** sau dezvoltarea unui **ASIC** (Application Specific Integrated Circuit).

De ce Verilog?

Un limbaj de descriere hardware conține o serie de abstractizări sau moduri de a genera, prin intermediul codului, porți logice. În comparație cu a proiecta “de mână” circuitele integrate, tocmai aceste abstractizări sunt cele care au permis electronicii digitale să se dezvolte în ritm rapid, odată cu progresul tehnologiei de fabricație. Cu ajutorul lor putem descrie relativ ușor structuri complexe, divizându-le în componentele lor comune și de bază.

Însă apare întrebarea naturală: Ce aș putea face cu un FPGA și nu aș putea face cu un procesor? Pe scurt, există trei răspunsuri:

- Un FPGA poate fi reconfigurat într-un timp foarte scurt. Asta înseamnă că, dacă am greșit ceva în design-ul nostru, dacă dorim să-l modificăm sau să-l extindem, timpul și costul acestei acțiuni sunt foarte mici;
- Un FPGA, prin construcția lui, oferă un grad extrem de ridicat de paralelism, lucru pe care codul scris pentru un procesor (deci cod secvențial) îl oferă într-o măsură mai redusă și mai greu de controlat;
- Un FPGA este de preferat oricând se dorește interfațarea unui dispozitiv (un senzor, un dispozitiv de afișare, etc.) care are nevoie de timpi foarte stricți în protocolul de comunicatie (exemplu: așteaptă 15 nanosecunde înainte să schimbi linia de ceas, apoi activează linia de enable pentru 25 de nanosecunde, apoi pune datele pe linia de date și ține-le cel puțin 50 de nanosecunde, etc). Pe un procesor acest lucru este iarăși dificil de controlat, fiindcă majoritatea instrucțiunilor se execută într-un număr diferit de cicli de ceas.

Întrucât au fost puse în discuție atât proiectarea prin porți logice a unui circuit cât și descrierea lui la un nivel mai abstract, putem clasifica alternative de descriere a unui circuit:

- **descrierea structurală** - mai puțin folosită, ea reprezintă o implementare asemănătoare cu o schemă logică a unui circuit, folosind primitive și module pentru implementarea funcționalității
- **descrierea comportamentală** - divizată în descriere la nivel de flux de date și descriere la nivel procedural, folosește construcții de nivel înalt, întâlnite și în alte limbaje de programare.

Ce tipuri de circuite putem construi?

Circuitele logice combinaționale aplică funcții logice pe intrări pentru a obține ieșirile. Valorile de ieșire depind astfel doar de valorile curente de intrare, iar când starea unei intrări se schimbă, se reflectă imediat asupra ieșiri.

Diagrama bloc pentru un circuit combinațional cu n intrări și m ieșiri

Logica combinațională poate fi reprezentată prin:

- diagrame structurale la nivel de porți logice,
- tabele de adevăr,
- expresii booleene (funcții logice).

Spre deosebire de **circuitele** logice combinaționale, cele **secvențiale** (eng: sequential logic) nu mai depind exclusiv de valoarea curentă a intrărilor, ci și de stările anterioare ale circuitului. Logica secvențială poate fi de două tipuri: sincronă și asincronă.

Schema bloc a unui circuit secvențial sincron

În primul caz, cel cu care vom lucra și la laborator, este folosit un semnal de ceas care comandă elementul/elementele de memorare, acestea schimbându-și starea doar la impulsurile de ceas. În al doilea caz, ieșirile se modifică atunci când se modifică și intrările, neexistând un semnal de ceas pentru elementele de memorare. Circuitele secvențiale asincrone sunt mai greu de proiectat, pot apărea probleme de sincronizare și sunt folosite mai rar. În continuare ne vom referi doar la circuitele secvențiale sincrone.

Pentru mai multe detalii și exemple privind circuitele combinaționale și secvențiale, studiați Laboratorul 0.

În laboratorul curent ne vom concentra asupra asimilării acestui nou limbaj, începând cu **descrierea structurală a unui circuit combinațional**.

Structura limbajului Verilog

Atunci când proiectăm un circuit digital folosind un HDL, începem prin a face o descriere textuală a circuitului, adică scriem cod. Acesta este compilat, iar în urma procesului va rezulta un model al circuitului care poate fi apoi rulat într-un simulator cu scopul de a verifica funcționalitatea descrierii. O alternativă la simulare este folosirea unui utilitar de sintetizare, care preia codul HDL și generează fișiere de configurare pentru FPGA.

Însă proiectarea circuitelor poate deveni complexă. Datorită acestui motiv, se preferă proiectarea de tip top-down, o modalitate de partiționare sistematică și repetată a unui sistem complex în unități funcționale mai simple, a căror implementare poate fi făcută mai facil. O partiționare și organizare la nivel înalt a unui sistem reprezintă arhitectura acestuia. Unitățile funcționale individuale ce rezultă în urma partiționării sunt mai ușor de proiectat și de testat decât întregul sistem. Strategia divide-et-impera a proiectării top-down ne permite proiectarea de circuite care conțin milioane de porți.

Module

Modulul este unitatea de bază a limbajului Verilog, element ce încapsulează inferfața și comportamentul unui circuit. Modelul *black-box* este cel mai apropiat de definiția unui modul, întrucât se cunosc elementele de legătură: intrările și ieșirile din modul precum și funcționalitatea precisă a modulului, cu un accent mai redus asupra detaliilor de implementare și a modului în care acesta funcționează.

Pentru declararea unui modul, se folosesc cuvintele cheie `module` și `endmodule`. Pe lângă aceste cuvinte cheie, declarația unui modul mai conține:

- numele acestuia,
- lista de porturi (pentru interfața cu exteriorul): pot fi de intrare (input), ieșire (output) sau intrare - ieșire (inout) și pot avea unul sau mai mulți biți,
- și, desigur, implementarea funcționalității modulului.

Ordinea porturilor unui modul nu este restricționată. Intrările și ieșirile pot fi declarate în orice ordine, însă, pentru consistență, o regulă de bună practică este folosirea convenției (obligatorie la primitive): prima dată se declară ieșirile, apoi intrările.

Exemplu declarare modul

```
module my_beautiful_module (
    output out,
    input [3:0] a,
    input b);
/* descrierea funcționalității */
endmodule
```

Pentru implementarea modului avem la dispoziție câteva elemente, care sunt descrise în ceea ce urmează.

Primitive

Element ce stă la baza descrierii structurale a circuitelor, primitiva este o funcție asociată unei porți logice de bază. Verilog are o suită de primitive predefinite:

- primitive asociate porților logice: and, or, nand, nor, xor, xnor;
- primitive asociate porților de transmisie: not, buf, etc;
- primitive asociate tranzistorilor: pmos, tranif, etc.

Fiecare primitivă are porturi, prin care este conectată în exterior. Primitivele predefinite oferă posibilitatea conectării mai multor intrări (ex. or, and, xor etc.) sau mai multor ieșiri (ex. buf, not). Folosirea unei primitive se face prin instanțierea sa cu lista de semnale care vor fi conectate la porturile ei. Pentru primitivele predefinite porturile de ieșire sunt declarate **înaintea** porturilor de intrare.

Tabelul de mai jos oferă câteva exemple de instanțiere a unor porți în Verilog. Pentru primitivele predefinite numele instanței este opțional.

Schema	Cod

	`or(out, a, b, c);`
	`// sau`
	`or o1(out, a, b, c);`

	`not(out, in);`
	`// sau`
	`not my_not(out, in);`

	`nand (z, x, y);`
	`// sau`
	`nand n(z, x, y);`

Exemple de instanțiere a primitivelor

Wires

Un singur modul sau o singură primitivă nu poate îndeplini singură funcția cerută. Astfel, apare necesitatea interconectării modulelor sau a primitivelor. Specificarea semnalelor dintr-o diagramă se face prin **wires**, care se declară prin cuvântul cheie `wire`.

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

În exemplul anterior y1 și y2 sunt semnale de câte 1 bit care leagă ieșirile porților and (y1) și nand (y2) la intrările porții xor.

Pentru a declara semnale pe mai mulți biți se pot folosi vectori precum în declarațiile următoare: m reprezintă un semnal de 8 biți, iar n reprezintă un semnal de 5 biți. Bitul cel mai semnificativ (eng. most significant bit - MSB) este situat întotdeauna în stânga, iar bitul cel mai puțin semnificativ (eng. least significant bit - LSB) în dreapta.

În mod implicit semnalele care nu sunt declarate sunt considerate ca fiind de tip wire și având 1 bit (ex. in1, in2, ... din codul de mai sus). Putem accesa individual biții dintr-un wire sau putem accesa un grup consecutiv de biți specificând intervalul (ex. m[0], m[3:1], m[7:2]).

```
wire[7:0] m; // 8 biti, MSB este bitul 7, LSB bitul 0
wire[0:4] n;  // 5 biti, MSB este bitul 0, LSB bitul 4
wire[7:0] a [9:0]; // array multidimensional cu 10 elemente de 8 biti
```

Exerciții

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

1. **(4p)** Implementați și simulați un **sumator elementar complet**, utilizând sumatoare elementare parțiale.
   - *Hint*: Urmăriți tutorialul pentru a realiza simularea (săriți peste adăugarea modulului de test, deoarece este deja adăugat).
   - *Hint*: Consultați laboratorul 0 pentru implementare.
2. **(3p)** Implementați și simulați un **sumator pe 4 biți**, cu două intrări și două ieșiri. Verificați corectitudinea sumatorului vizualizând semnalele în baza 10.
   - *Hint*: Consultați laboratorul 0 pentru implementarea unui sumator pe mai mulți biți.
   - *Hint*: Folosiți sumatorul implementat la exercițiul 1, adăugându-l la proiect din meniul Project→Add Copy of Source... .
   - *Hint*: Modificați afișarea unui semnal cu click dreapta→Radix→Unsigned Decimal.
3. **(3p)** Implementați și simulați un **sumator pe 6 biți**, cu două intrări și o ieșire. Câți biți va avea ieșirea? De ce?
   - *Hint*: Folosiți atât sumatoare pe 1 bit, cât și sumatoare pe 4 biți.
4. **(2p)** Implementați și simulați un **comparator** pe un bit. Acesta are două intrări și 3 ieșiri (pentru mai mic, egal și mai mare).
   - *Hint*: Respectați interfața cerută în scheletul de cod.

Resurse

- Schelet de cod
- PDF laborator
- Soluție laborator