LUCRAREA NR. 11
MODULE DE SIMULARE

1. Scopul lucrării

Lucrarea abordează problematica simulării în VHDL. Se prezintă noţiunile fundamentale legate de simularea dispozitivelor numerice hardware în general, după care se adaptează aceste informaţii în contextul limbajului VHDL. Se definesc *modulele de simulare* în VHDL (contextul în care se utilizează, elementele lor componente etc.), după care se detaliază fiecare aspect, ilustrându-se prin exemple semnificative noţiunile noi expuse.

Se evidenţiază importanţa generării rezultatelor simulării şi modurile în care se pot obţine aceste rezultate în VHDL. Se insistă pe utilizarea instrucţiunii **assert**, subliniindu-se universalitatea acesteia şi ilustrându-se modul său de aplicare cu exemple instructive.

2. Consideraţii teoretice

2.1 Module de simulare

Procesul de proiectare a sistemelor hardware ar fi incomplet fără *verificarea* proiectelor. La fel ca şi în cazul altor medii integrate de dezvoltare, există mai multe modalităţi de verificare a unui proiect VHDL. Însă dintre toate metodele, soluţia cea mai des utilizată o constituie *modulele de simulare* (*test benches*).

Un *modul de simulare* este un mediu integrat, în care un proiect (numit şi *unitate supusă testării - UST*) este verificat prin aplicarea unor semnale numite *stimuli* şi observarea *răspunsurilor* generate de sistem. Cu alte cuvinte, un modul de simulare se substituie mediului în care urmează să se desfăşoare ciclul de viaţă al proiectului, astfel încât să putem observa şi analiza comportamentului sistemului proiectat.

Un modul de simulare este alcătuit din următoarele elemente:

- *un soclu* (*socket*) în care va fi plasat sistemul supus testării;
- *un generator de stimuli* (un subsistem care aplică stimuli proiectului aflat în testare, fie generându-i intern, fie preluându-i de la o sursă exterioară de semnal);
- *instrumente de monitorizare a răspunsurilor* la stimuli, generate de către sistemul supus testării.

Noţiunea de module de simulare a fost adaptată şi pentru proiectele scrise în VHDL. Adaptările se referă la următoarele aspecte: un modul de simulare nu este un sistem de sine stătător, ci o specificaţie VHDL care va fi simulată de către simulatorul integrat în sistemul de dezvoltare. Modulul de simulare este alcătuit din:

- instanţierea unităţii supuse testării;
- procese „sensibile" la stimulii aplicaţi unităţii supuse testării.

În acest fel, se creează o specificaţie hibridă, care combină atât instrucţiuni specifice tipului de descriere structurală cât şi instrucţiuni specifice tipului de descriere comportamentală. O asemenea abordare este validă în VHDL deoarece atât instanţierea componentelor cât şi procesele reprezintă instrucţiuni concurente.

Stimulii destinaţi unităţii supuse testării (UST) sunt specificaţi în interiorul arhitecturii modulului de simulare (*test bench*) sau pot fi citiţi dintr-un fişier extern. Pe de altă parte, reacţiile unităţii testate pot fi observate în mai multe feluri:

a) prin intermediul ieşirilor generate de simulatorul VHDL (adică a formelor de undă care apar pe ecran);
b) prin intermediul fişierelor de raport conţinând mesaje generate de simulator;
c) prin intermediul mesajelor generate de simulator la consolă;
d) prin intermediul scrierii în fişiere folosind operaţiile de intrări / ieşiri în mod text care sunt disponibile în VHDL (în pachetul standard TEXTIO).

Numărul mare de opţiuni puse la dispoziţie de limbajul VHDL oferă o mare flexibilitate în vederea scrierii de module de simulare. „Reversul medaliei" l-ar putea constitui însă scrierea de module de simulare cu o complexitate mai mare chiar decât cea a unităţii testate...

2.2 Elementele unui modul de simulare

Aşa cum am menţionat anterior, în VHDL un modul de simulare constituie de fapt o specificaţie VHDL care dispune de propria sa entitate şi arhitectură. Cu toate acestea, modulul de simulare are o structură specială care conţine o serie de elemente caracteristice:

- **entitatea modulului de simulare**: *aceasta nu are porturi*;
- **instanţierea componentei unităţii supuse testării (UST)** – relaţia dintre modulul de simulare şi unitatea supusă testării este specificată prin intermediul *instanţierii componentei* şi al *specificaţiei de tip structural*;
- **stimulii** – reprezintă un set de semnale care sunt *declarate intern* în cadrul *arhitecturii modulului de simulare* şi care sunt *asignate porturilor unităţii testate* la *instanţierea* acesteia. Stimulii sunt definiţi ca *forme de undă* în cadrul unuia sau mai multor *procese comportamentale*.

Exemplul următor ilustrează noţiunile expuse până în acest moment:

```
-- Entitatea modulului de simulare
entity MODUL_DE_SIMULARE is
end MODUL_DE_SIMULARE;
architecture ARH_MODUL_DE_SIMULARE of MODUL_DE_SIMULARE is
-- Declararea componentei
component POARTA_ŞI is
      port (A, B: in BIT;
                Y: out BIT);
end component;
-- Declararea stimulilor
signal A, B, C: BIT;
begin
-- Instanţierea unităţii supuse testării (UST)
UST: POARTA_ŞI port map (A, B, C);
-- Semnalele de stimulare
      A <= '0', '1' after 20 ns, '0' after 40 ns;
      B <= '1', '0' after 40 ns, '1' after 80 ns;
end ARH_MODUL_DE_SIMULARE;
```

În exemplul de mai jos este prezentată o *generalizare* a utilizării modulelor de simulare în VHDL:

```
-- Entitatea modulului de simulare
entity MODUL_DE_SIMULARE is
end MODUL_DE_SIMULARE;
architecture ARH_MODUL_DE_SIMULARE of MODUL_DE_SIMULARE is
-- Declararea componentei
component COMP is
      port (...);
end component;
-- Declararea stimulilor
signal A, B, C...: BIT;
signal Z...
begin
-- Instanţierea unităţii supuse testării (UST)
UST: entity WORK.COMP(COMP_ARH) port map (...);
-- Definirea stimulilor
stimuli: process
            begin
                  A <= ...;
                  B <= ...;
                  ...
                  wait for...
                  ...
                  wait;
            end process stimuli;
end ARH_MODUL_DE_SIMULARE;
```

2.3 Utilizarea modulelor de simulare

De fiecare dată când creăm o nouă specificaţie VHDL, trebuie să avem în vedere faptul, deosebit de important, că proiectul nostru va trebui să fie (cât mai) uşor de verificat. Evident, fiecare proiect poate fi simulat, dar în cazul proiectelor de dimensiuni foarte mari, procesul de simulare *on-line* se poate dovedi extrem de costisitor din punct de vedere al timpului. Este mult mai convenabil să utilizăm un modul de simulare pentru verificarea proiectului. Întrucât scrierea unui modul de simulare se poate dovedi o sarcină foarte complexă, este recomandabil să scriem (sub formă de comentarii sau de notiţe separate) o serie de indicaţii generale referitoare la modul viitor de dezvoltare al stimulilor, pe măsură ce procesul de proiectare a sistemului avansează. Unii proiectanţi cu experienţă recomandă chiar ca să se scrie fişiere de stimuli (sau vectori de test) complete în acest moment.

După ce încheiem complet scrierea unităţii supuse testării şi a modulului de simulare, putem demara verificarea. Trebuie să ne fie foarte clar faptul că vom simula modulul de simulare şi NU unitatea supusă testării: aceasta este doar una dintre componentele instanţiate în interiorul modulului de simulare.

Nu există limitări în privinţa dimensiunii modulului de simulare: singura limitare o constituie capacitatea şi performanţele simulatorului VHDL.

Exemplul de mai jos ilustrează o posibilitate de descriere a unui stimul „semnal de tact (CLOCK)":

```
...
-- Constantă declarată în arhitectura modulului de simulare
constant CONSTANTA_TACT: TIME := 20 ns;
signal CLOCK: STD_LOGIC;
...
-- Perioada tactului va fi de: 2*CONSTANTA_TACT = 40 ns;
GENERATOR_TACT: process(CLOCK)
begin
      if CLOCK = 'U' then CLOCK <= '0';
      else
            CLOCK <= not CLOCK after CONSTANTA_TACT;
      end if;
end process GENERATOR_TACT;
```

Noţiunile expuse până în acest punct sunt reluate într-o formă grafică în figura 11.1.

2.3.1 Entitatea modulului de simulare

Modulul de simulare arată la fel ca oricare altă specificaţie VHDL: el este alcătuit dintr-o entitate şi o arhitectură. Există însă o deosebire majoră faţă de celelalte situaţii: *această entitate nu conţine nici declaraţii de porturi şi nici declaraţii de parametri generici*.

Motivul este evident: modulul de simulare nu este un dispozitiv real sau un sistem care să fie nevoit să comunice cu mediul său înconjurător; prin urmare, el nu are nevoie de intrări sau ieşiri. Toate valorile destinate porturilor de intrare ale unităţii supuse testării (UST) sunt specificate în interiorul arhitecturii modulului de simulare, ca *stimuli*. Ieşirile sunt

monitorizate (de exemplu, cu ajutorul simulatorului) şi apoi pot fi eventual salvate într-un fişier.

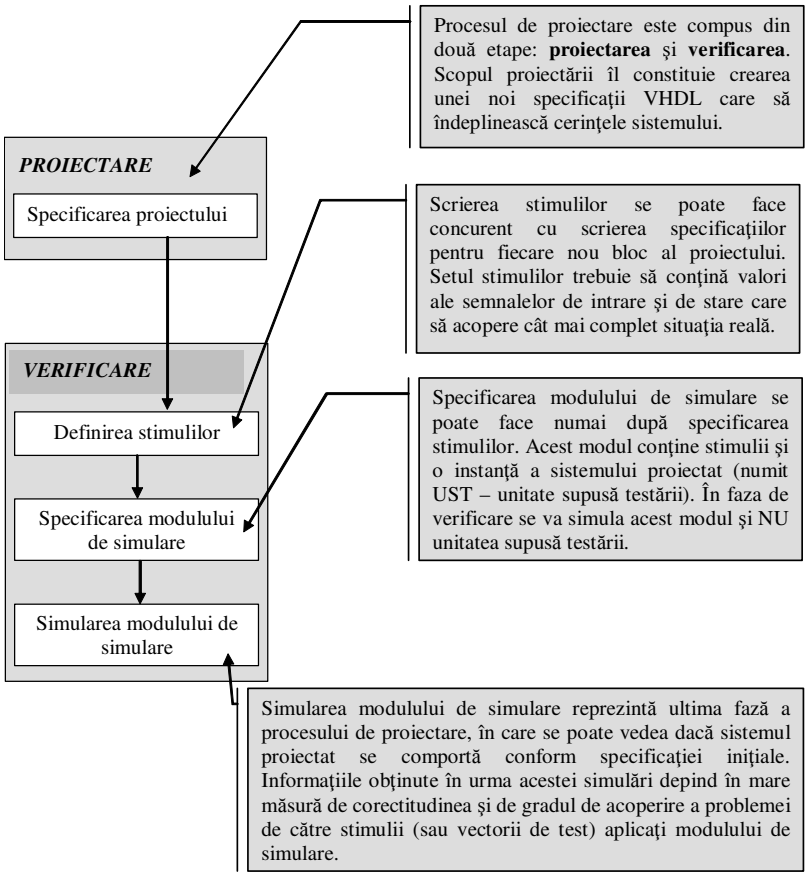

Figura 11.1 *Procesul de proiectare şi verificare a proiectului în VHDL*

De ce este oare nevoie de o entitate a modulului de simulare, când el este practic numai o arhitectură? Răspunsul ne este dat de către regulile de construcţie a specificaţiilor VHDL: nici o arhitectură nu poate fi specificată fără entitatea sa – această regulă se aplică inclusiv modulelor de simulare.

În exemplul de mai jos prezentăm un modul de simulare care se opreşte singur:

```
-- Entitatea modulului de simulare
entity MS_BISTABIL_D is
end MS_BISTABIL_D;

architecture ARH_MS_BISTABIL_D of MS_BISTABIL_D is
-- Declararea componentei
component COMP_D is
      port (...);
end component;
shared variable  END_SIM: BOOLEAN := FALSE;
signal CLK, D, Q, NQ: BIT;
-- Perioada semnalului de tact generat
constant PERIOADA_CLK: TIME := 20 ns;

begin
-- Instanţierea unităţii supuse testării (UST)
UST: COMP_D port map (...);
-- Definirea stimulilor
GENERATOR_TACT: process
                  begin
                  if not END_SIM then
                   CLK <= '0';
                   wait for PERIOADA_CLK / 2;
                   CLK <= '1';
                   wait for PERIOADA_CLK / 2;
                  else wait;
                  end if;
            end process GENERATOR_TACT;
-- Stimulul asignat porturilor de intrare ale COMP_D
STIMUL: process
begin
      D <= '1';
      wait for 200 ns;
      D <= '0';
      wait for 200 ns;
      END_SIM := TRUE; -- când END_SIM devine TRUE, procesul
                       -- GENERATOR_TACT este oprit
      wait;
end process STIMUL;
end ARH_MS_BISTABIL_D ;
```

2.3.2 Unitatea supusă testării

Un sistem care va fi verificat cu ajutorul unui modul de simulare nu are nevoie de modificări sau declaraţii adiţionale. Din acest motiv, oricărei specificaţii VHDL (chiar şi uneia provenită dintr-o sursă externă) i se poate aplica un modul de simulare. O astfel de situaţie poate să survină atunci

când este necesară efectuarea unei simulări de mare anvergură în care sunt implicate mai multe dispozitive.

În mod obligatoriu, unitatea supusă testării trebuie să fie instanţiată în interiorul arhitecturii modulului de simulare. Acest lucru poate fi realizat în acelaşi mod ca în cazul oricărei specificaţii structurale, adică fie prin instanţiere directă, fie prin instanţierea unei componente însoţită de declaraţia componentei şi de configuraţie. Porturilor instanţei unităţii supuse testării din cadrul modulului de simulare le vor fi asignate semnalele-stimul.

Întrucât atât procesele cât şi instrucţiunea de instanţiere a unei componente reprezintă instrucţiuni concurente, nu are importanţă dacă unitatea supusă testării va fi instanţiată prima şi stimulii vor fi definiţi mai târziu, sau invers.

Exemplul de mai jos prezintă specificaţia unui demultiplexor 1 la 4 şi modulul de simulare asociat lui:

```
entity DMUX is
      port(X: in BIT;
           S: in BIT_VECTOR (1 downto 0);
           Y: out BIT_VECTOR (3 downto 0));
end DMUX;
architecture ARH_DMUX of DMUX is
begin
      Y <= ('0', '0', '0', X) when S = "00" else
           ('0', '0', X, '0') when S = "01" else
           ('0', X, '0', '0') when S = "10" else
           (X, '0', '0', '0') when S = "11";
end ARH_DMUX;
-- Entitatea modulului de simulare
entity MS_DMUX is
end MS_DMUX;

architecture ARH_MS_DMUX of MS_DMUX is
-- Declararea componentei
component DMUX is
      port(X: in BIT;
           S: in BIT_VECTOR (1 downto 0);
           Y: out BIT_VECTOR (3 downto 0));
end component DMUX;
signal X: BIT;
signal S: BIT_VECTOR (1 downto 0);
signal Y: BIT_VECTOR (3 downto 0);
```

```
begin
-- Instanţierea unităţii supuse testării (UST)
UST: DMUX port map (X => X, S => S, Y => Y);
STIM: process
       begin
        X <= '1';
        S <= "00";
        wait for 40 ns;
        S <= "01";
        wait for 40 ns;
        S <= "10";
        wait for 40 ns;
        S <= "11";
        wait for 40 ns;
        wait;
      end process STIM;
end ARH_MS_DMUX ;
```

2.3.3 Stimuli

Un element esenţial al oricărui modul de simulare îl reprezintă *setul de stimuli*: o secvenţă de valori aplicate în timp fiecărui semnal de intrare al unităţii supuse testării. De vreme ce modulul de simulare nu comunică cu mediul său înconjurător prin semnale, toţi stimulii trebuie să fie declaraţi intern în arhitectura modulului de simulare. Ei vor fi declaraţi la fel ca oricare alt semnal, în zona declarativă a arhitecturii.

Stimulii pot fi specificaţi astfel:

- *ca instrucţiuni concurente de asignare* de valori unor semnale (în care modificările survenite pe semnalele de intrare sunt date ca forme de undă, cu ajutorul cuvântului cheie **after** şi alegând modelul de propagare dorit, transport sau inerţial);
- *în interiorul unui proces care conţine instrucţiuni de asignare* de valori unor semnale separate prin intermediul instrucţiunilor **wait for** (care introduc întârzieri între execuţia instrucţiunilor de asignare menţionate mai sus). În acest caz, este necesară adăugarea unei instrucţiuni **wait** vide (fără nici o condiţie) pe ultima poziţie a listei de instrucţiuni secvenţiale executate în cadrul procesului. Această instrucţiune va suspenda procesul pe termen nelimitat (altminteri - aşa cum am arătat în lucrarea nr. 6 dedicată proceselor - procesul ar fi executat din nou de la început).

Relaţia dintre stimuli şi unitatea supusă testării este specificată prin intermediul asignărilor realizate în cadrul clauzei **port map** a instrucţiunii de instanţiere a unităţii testate.

Se prezintă în continuare exemplul unui modul de simulare al unui numărător pe 8 biţi, în care stimulii sunt specificaţi atât în formă secvenţială cât şi în formă concurentă:

```
-- Entitatea modulului de simulare
entity MS_NUMĂRĂTOR_8 is
end MS_NUMĂRĂTOR_8;
architecture ARH_MS_NUMĂRĂTOR_8 of MS_NUMĂRĂTOR_8 is
-- Declaraţia componentei
component NUMĂRĂTOR_8 is
      port (CLK, RESET: in STD_LOGIC;
            CE: in STD_LOGIC;                -- Clock enable
            LOAD: in STD_LOGIC;              --        Încărcare
paralelă
            DIR: in STD_LOGIC;               --  Numărare  sus  /
jos
            DIN: in INTEGER range 0 to 255;  -- Intrări de date
            COUNT: out INTEGER range 0 to 255-- Ieşiri
            );
end component NUMĂRĂTOR_8;
signal CLK, RESET, CE, LOAD, DIR: STD_LOGIC;
signal DIN: INTEGER range 0 to 255;
signal COUNT: INTEGER range 0 to 255;
begin
-- Instanţierea unităţii supuse testării (UST)
UST: NUMĂRĂTOR_8 port map (...);
process(CLK)
  begin
      if CLK = 'U' then CLK <= '0';
      else CLK <= not CLK after 15 ns;
      end if;
end process;
RESET <= '1','0' after 15 ns;   -- Iniţializare, apoi
                                -- funcţionare în regim normal
DIN <= 250;                     -- Intrările de date
STIM1: process
            begin
             LOAD <= '0';
             wait for 5 us;
             LOAD <= '1'; -- Se încarcă starea 250
             wait for 100 ns;
             LOAD <= '0';
             wait;
       end process STIM1;
```

```
STIM2: process
       begin
              DIR <= '1'; -- Numărare jos
              CE <= '1';  -- Clock Enable permite funcţionarea
              wait for 2 us;
              DIR <= '0'; -- Numărare sus
              wait for 1 us;
              CE <= '0';  -- Clock Enable nu permite funcţionarea
              DIR <= '1'; -- Numărare jos
              wait for 1 us;
              DIR <= '0'; -- Se reia Numărarea crescătoare
              wait;
        end process STIM2;
end ARH_MS_NUMĂRĂTOR_8;
```

*2.3.4 Instrucţiunea **assert***

Ultimul element al oricărei verificări încununate de succes îl reprezintă afişarea sau raportarea rezultatelor simulării. Acest lucru poate fi realizat în mai multe moduri: prin afişarea listei valorilor semnalelor care se modifică în timp (echivalentă cu afişarea formelor de undă), prin scrierea rezultatelor simulării într-un fişier „jurnal al simulării" (*log file*) sau prin utilizarea instrucţiunii VHDL **assert**.

Ultima opţiune este uşor de aplicat şi este folosită adeseori pentru afişarea unui mesaj atunci când se detectează o eroare în funcţionarea sistemului proiectat.

Atât instrucţiunea concurentă **assert**, cât şi instrucţiunea secvenţială **assert**, au fost deja prezentate în detaliu în cadrul lucrărilor nr. 7, respectiv 9.

Instrucţiunea **assert** se întrebuinţează cel mai adesea, în cadrul modulelor de simulare, pentru raportarea răspunsurilor eronate generate de unitatea supusă testării la recepţionarea stimulilor definiţi. Iată câteva moduri posibile (cele mai frecvent întâlnite) de utilizare a acestei instrucţiuni:

- se aplică o instrucţiune **assert** de fiecare dată când se aşteaptă o nouă valoare a unui semnal de ieşire al unităţii supuse testării;
- valoarea prognozată se specifică drept condiţie în cadrul instrucţiunii **assert**;
- se vor folosi mesaje de eroare cât mai precise şi detaliate: un simplu mesaj de genul „Eroare!" nu va oferi prea multe informaţii cu privire la ce nu funcţionează corect în simulare...

Trebuie scris CE nu funcţionează şi CÂND a avut loc respectivul eveniment (de exemplu, în ce condiţii de intrare).

Reamintim că noile valori vor fi asignate semnalelor numai în momentul suspendării proceselor; ne vom aştepta să obţinem nişte rezultate ale simulării corelate cu acest aspect.

Exemplul unui multiplexor 2:1, de semnale pe 2 biţi va ilustra modul de utilizare a instrucţiunii **assert** pentru generarea rezultatelor simulării, în cadrul proceselor din interiorul arhitecturii modulului de simulare:

```
-- Entitatea Multiplexor 2:1 de semnale pe 2 biţi

entity MUX2_LA_1_PE_2_BIŢI is

      generic(MUX_DELAY: TIME := 5 ns);

      port(A, B: in STD_LOGIC_VECTOR(1 downto 0);
           SEL: in STD_LOGIC;
           Y: out STD_LOGIC_VECTOR(1 downto 0));
end entity MUX2_LA_1_PE_2_BIŢI;

architecture ARH of MUX2_LA_1_PE_2_BIŢI is

begin
      with SEL select
            Y <= A after MUX_DELAY when '0',
                 B after MUX_DELAY when '1',
                 "XX" when others;
end architecture ARH;

entity MS is
end entity MS;

architecture ARH_MS of MS is

signal A, B, Y: STD_LOGIC_VECTOR(1 downto 0);
signal SEL: STD_LOGIC;

component MUX2_LA_1_PE_2_BIŢI is

      generic(MUX_DELAY: TIME := 5 ns);

      port(A, B: in STD_LOGIC_VECTOR(1 downto 0);
           SEL: in STD_LOGIC;
           Y: out STD_LOGIC_VECTOR(1 downto 0));
end component;
```

```
begin

UST: entity MUX2_LA_1_PE_2_BIŢI port map (A, B, SEL, Y);

STIMULI: process
begin
       SEL <= 'X';
      A <= "00";
      B <= "11";
       wait for 0 ns;
       assert (Y = "XX") report "Testul a eşuat pt. SEL = X";
       SEL <= '0';
       wait for 40 ns;
       assert (Y = "00") report "Testul a eşuat pt. SEL = 0";
       A <= "01";
       wait for 20 ns;
       assert (Y = "01") report "Testul a eşuat pt. SEL = 0 -
       Y nu s-a modificat";
       SEL <= '1';
       wait for 20 ns;
       assert (Y = "11") report "Testul a eşuat pt. SEL = 1";
       B <= "10";
       wait for 20 ns;
       assert (Y = "10") report "Testul a eşuat pt. SEL = 1 -
       Y nu s-a modificat";
       ...
end process STIMULI;

end architecture ARH_MS;
```

3. Desfăşurarea lucrării

3.1 Se vor testa şi implementa toate exemplele de module de simulare din lucrare.

3.2 Se vor determina stimulii de test optimi pentru exemplele din această lucrare.

3.3 Se vor crea module de simulare, inclusiv alegerea setului adecvat de stimuli de test, pentru exemplele din lucrările nr. 6 şi 7.

3.4 Pentru exemplul de mai jos, se va crea un modul de simulare şi se vor alege vectorii de test cei mai adecvaţi:

```
library IEEE;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_1164.all;

entity COUNTER is
       port( ZERO : out STD_LOGIC;
             CLK, LOAD_COUNT, RESET : in STD_LOGIC;
             COUNT : in STD_LOGIC_VECTOR(3 downto 0); );
end COUNTER;

architecture COUNTER of COUNTER is

signal Q : STD_LOGIC_VECTOR(3 downto 0);
signal EMPTY : STD_LOGIC;

begin

process (CLK, RESET)
      begin
            if RESET='1' then
                  Q <= (others => '0');
                  EMPTY <= '0';
            else
              if CLK'EVENT and CLK='1' then
               if LOAD_COUNT = '1' then
                  Q <= COUNT;
                  EMPTY <= '0';
               else
                  if Q = 0 then
                        EMPTY <= '1';
                  else
                        Q <= Q - 1;
                  end if;
               end if;
              end if;
            end if;
end process;

      ZERO <= EMPTY;

end COUNTER;
```