CAPITOLUL 3

AUTOMATE DE STARE ÎN LIMBAJUL VHDL

În acest capitol se descriu diferite tehnici pentru proiectarea automatelor de stare utilizând limbajul VHDL. Se prezintă mai întâi deosebirea dintre automatele de stare Moore şi Mealy. În continuare, se descrie un exemplu simplu de proiectare, din care rezultă faptul că realizarea unei descrieri funcționale a unui automat de stare constă din simpla translatare a unei diagrame de stare în instrucțiuni `case` şi `if`. În ultima parte a capitolului, se prezintă tehnici de codificare a stărilor pentru automatele de stare.

3.1 Automate de stare de tip Moore şi Mealy

Există două tipuri de automate de stare, automate Moore şi automate Mealy. În cazul automatelor de stare Moore, ieşirile reprezintă funcții doar de starea prezentă. Automatele de tip Mealy pot avea ieşiri care sunt funcții atât de starea prezentă, cât şi de intrările prezente. Deosebirea dintre cele două tipuri de automate de stare este ilustrată în figura 3.1.

Operațiile suplimentare necesare pentru descrierea automatelor Mealy față de cele necesare pentru descrierea automatelor Moore sunt minime. Pentru implementarea unui automat de tip Mealy, trebuie să se descrie ieşirile ca funcții atât de biții de stare, cât şi de intrări. În general, utilitarele de sinteză permit utilizarea ambelor tipuri de automate de stare.

3.2 Exemplu de proiectare

Ca un exemplu, se va proiecta un automat de stare reprezentând un controler simplu de memorie, mai întâi prin metoda tradițională, iar apoi utilizând limbajul VHDL. Controlerul de memorie activează şi dezactivează semnalele *OE* (*Output*

Enable) și *WE* (*Write Enable*) ale unui buffer de memorie în timpul tranzacțiilor de citire și de scriere. Intrările principale ale controlerului de memorie sunt semnalele *Ready* și *RW* (*Read/Write*) care provin de la un microprocesor. Alte intrări ale controlerului sunt semnalul de ceas *Clk* și semnalul de resetare sincronă *Rst*. Ieșirile controlerului de memorie sunt semnalele *OE* și *WE*. O nouă tranzacție începe cu validarea semnalului *Ready* după terminarea unei tranzacții precedente (sau, la punerea sub tensiune, pentru tranzacția inițială). La un ciclu de ceas după începerea unei tranzacții, starea semnalului *RW* determină tipul tranzacției: de citire, dacă semnalul *RW* este activat, sau de scriere, în caz contrar. O tranzacție este terminată prin activarea semnalului *Ready*, după care poate începe o nouă tranzacție. Semnalul *OE* este activat în timpul unei tranzacții de citire, iar semnalul *WE* este activat în timpul unei tranzacții de scriere.

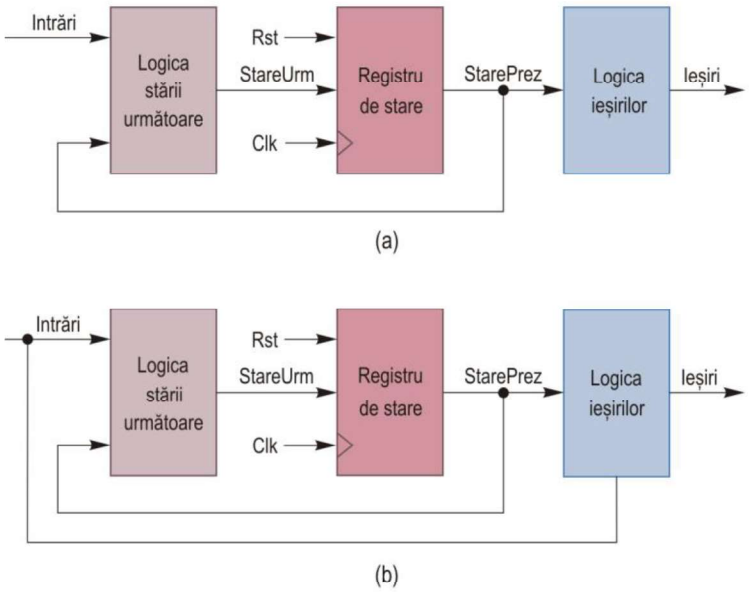
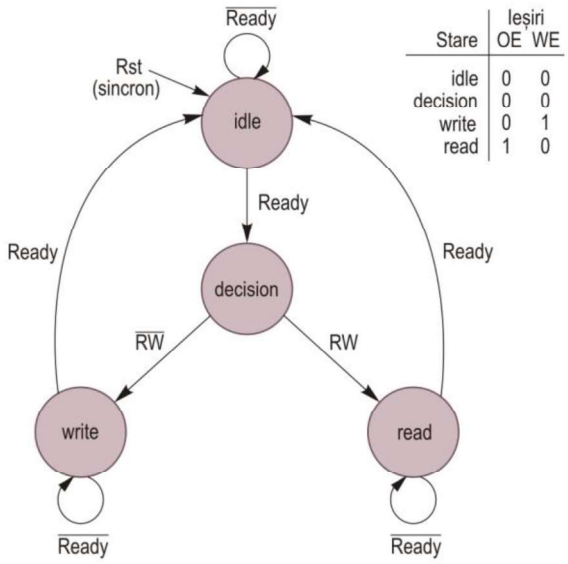


Figura 3.1. Deosebirea dintre automatele de stare de Moore (*a*) și automatele de stare Mealy (*b*).

3.2.1 Proiectarea tradițională

Conform metodologiei de proiectare tradiționale, se construiește mai întâi o diagramă de stare, pe baza căreia se poate întocmi apoi un tabel al stărilor. Se pot determina și elimina stările echivalente prin compararea liniilor din tabelul stărilor și utilizarea unui tabel al implicațiilor, dacă este necesar. În continuare, se asignează stările și se construiește un tabel de tranziție a stărilor, pe baza căreia se pot determina ecuațiile stării următoare și ecuațiile ieșirilor, ținând cont de tipul bistabilelor utilizate pentru implementare.

Considerăm că implementarea controlerului se va realiza printr-un automat Moore. Diagrama de stare a automatului este ilustrată în figura 3.2.



Stare	Ieșiri OE	WE
idle	0	0
decision	0	0
write	0	1
read	1	0

Figura 3.2. Diagrama de stare a controlerului de memorie.

Pentru acest automat de stare, nu există stări echivalente; toate stările necesită intrări diferite pentru tranziția în starea următoare, sau ieșirile sunt diferite. Pentru implementare se va utiliza numărul minim de bistabile necesare. Se alege implementarea cu bistabile de tip D. Se întocmește tabelul de tranziție a stărilor, combinat cu tabelul de asignare a stărilor (tabelul 3.1). Asignarea stărilor este indicată în coloana stării prezente (SP). Coloana stării următoare (SU) indică tranzițiile din starea prezentă în starea următoare pe baza valorii curente a celor două intrări, *RW* și *Ready*. Combinațiile valorilor acestor intrări sunt indicate prin 00, 01, 11 și 10. Deoarece implementarea se realizează cu bistabile D, nu este necesar să se prevadă o coloană pentru intrările bistabilelor, deoarece aceasta coincide cu coloana stării următoare. Ieșirile se indică în ultima coloană.

În continuare, se determină ecuațiile stării următoare pentru biții de stare. Pentru aceasta, pe baza tabelului de tranziție a stărilor se întocmesc diagramele Karnaugh ale biților de stare Q_0, Q_1 și ale ieșirilor *OE*, *WE* în funcție de biții stării prezente (q_0, q_1) și de intrări (*RW*, *Ready*). Aceste diagrame se utilizează pentru determinarea ecuațiilor minime ale stării următoare și ieșirilor. Deoarece implementarea se realizează cu bistabile D, ecuațiile stării următoare reprezintă și ecuațiile de intrare ale bistabilelor.

Tabelul 3.1. Tabelul de tranziție a stărilor pentru controlerul de memorie.

SP (q_0q_1)		SU (Q_0Q_1)				Ieșiri	
Nume	Cod	00	01	11	10	OE	WE
idle	00	00	01	01	00	0	0
decision	01	11	11	10	10	0	0
write	11	11	00	00	11	0	1
read	10	10	00	00	10	1	0

Se obțin următoarele ecuații:

$$Q_0 = \overline{q}_0 q_1 + q_0 \cdot \overline{Ready} \tag{3.1}$$

$$Q_1 = \overline{q}_0 \overline{q}_1 \cdot Ready + \overline{q}_0 q_1 \cdot \overline{RW} + q_0 q_1 \cdot \overline{Ready} \tag{3.2}$$

$$OE = q_0 \overline{q}_1 \tag{3.3}$$

$$WE = q_0 q_1 \tag{3.4}$$

Pe baza acestor ecuații, se poate realiza implementarea, de exemplu, într-un circuit PLD care dispune de bistabile de tip D. Pentru implementarea cu alte bistabile, este necesară determinarea unui nou set de ecuații pe baza tabelului de tranziție și a tabelului de excitație a bistabilului ales.

3.2.2 Proiectarea utilizând limbajul VHDL

Diagrama de stare din figura 3.2 se poate translata în mod simplu într-o descriere VHDL de nivel înalt fără a fi necesară asignarea stărilor, întocmirea tabelului de tranziție a stărilor și determinarea ecuațiilor stării următoare pe baza tipului de bistabile disponibile. În limbajul VHDL, fiecare stare se poate translata într-o alternativă a unei instrucțiuni `case`. Tranzițiile între stări se pot specifica apoi prin instrucțiuni `if`.

Pentru descrierea automatului de stare, se pot utiliza două procese sau trei procese, în funcție de modul în care se descompune modelul automatului de stare. Aceste variante de descriere sunt prezentate în continuare.

3.2.2.1 Descrierea automatului de stare cu două procese

Mai întâi, se exemplifică descrierea automatului de stare în care se utilizează două procese. Pentru translatarea diagramei de stare în limbajul VHDL, se definește un tip enumerat constând din numele stărilor, iar apoi se declară un semnal de acest tip, în modul indicat în liniile următoare.

```
type TIP_STARE is (idle, decision, read, write);
signal Stare : TIP_STARE;
```

Dintre cele două procese, primul (`proc1`) va descrie logica stării următoare și registrul de stare, iar al doilea (`proc2`) va descrie logica de ieșire (figura 3.3).

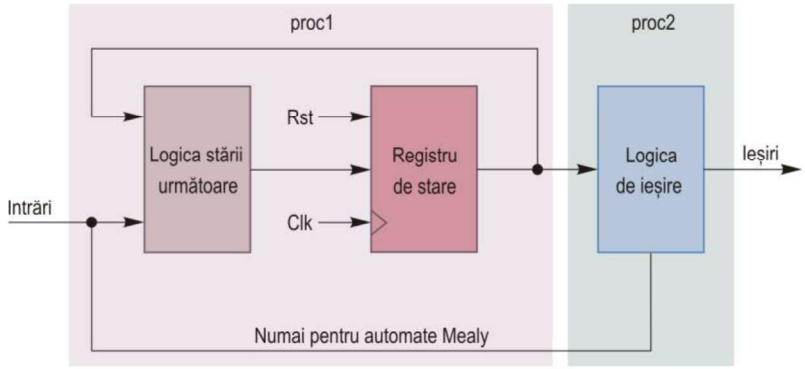


Figura 3.3. Automat de stare cu două procese.

În procesul `proc1` se descriu tranzițiile automatului de stare. Se poate utiliza o instrucțiune `case`, specificând pentru fiecare stare tranzițiile posibile și valorile ieșirilor. Procesul construit astfel indică tranzițiile executate pe baza stării prezente și a intrărilor prezente. Tranzițiile între stări au loc în mod sincron, pe frontul crescător al semnalului de ceas. Deoarece automatul de stare necesită un semnal de resetare sincronă, se va adăuga o instrucțiune `if` la începutul procesului `proc1` pentru a aduce automatul în starea `idle` în cazul în care semnalul *Rst* este activ.

În procesul combinațional `proc2` se specifică semnalele de ieșire ale automatului de stare. Pentru automatele de stare, stilul de descriere în care se utilizează un proces separat pentru specificarea semnalelor de ieșire este avantajos, deoarece ieșirile și tranzițiile stărilor se pot identifica în mod simplu. Aceste descrieri sunt mai ușor de creat și de înțeles, în special pentru automatele de stare de dimensiuni mari, automatele cu un număr mare de tranziții sau cu un număr mare de ieșiri.

Descrierea controlerului de memorie este prezentată în Exemplul 3.1.

Exemplul 3.1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity contr_mem is
   port (Clk   : in STD_LOGIC;
         Rst   : in STD_LOGIC;
         Ready : in STD_LOGIC;
         RW    : in STD_LOGIC;
         OE    : out STD_LOGIC;
         WE    : out STD_LOGIC);
end contr_mem;

architecture automat_stare of contr_mem is

type TIP_STARE is (idle, decision, read, write);
signal Stare : TIP_STARE;
```

```
begin
   proc1: process (Clk)
   begin
      if RISING_EDGE (Clk) then
         if (Rst = '1') then
            Stare <= idle;
         else
            case Stare is
               when idle =>
                  if (Ready = '1') then
                     Stare <= decision;
                  else                        -- else nu este necesar
                     Stare <= idle;
                  end if;
               when decision =>
                  if (RW = '1') then
                     Stare <= read;
                  else                        -- RW = '0'
                     Stare <= write;
                  end if;
               when read =>
                  if (Ready = '1') then
                     Stare <= idle;
                  end if;
               when write =>
                  if (Ready = '1') then
                     Stare <= idle;
                  end if;
            end case;
         end if;
      end if;
   end process;

   proc2: process (Stare)
   begin
      case Stare is
         when idle     => OE <= '0'; WE <= '0';
         when decision => OE <= '0'; WE <= '0';
         when read     => OE <= '1'; WE <= '0';
         when write    => OE <= '0'; WE <= '1';
      end case;
   end process proc2;

end automat_stare;
```

Descrierea funcțională a automatului de stare se realizează într-un mod mai simplu decât descrierea prin specificarea ecuațiilor. În același timp, printr-o asemenea descriere se reduce posibilitatea apariției erorilor. Prin sinteza descrierii funcționale rezultă o structură similară cu cea obținută prin implementarea ecuațiilor (3.1) – (3.4).

3.2.2.2 Descrierea automatului de stare cu trei procese

Automatul de stare poate fi descris prin utilizarea a trei procese, dacă modelul automatului de stare este descompus în modul ilustrat în figura 3.4. Procesul combinațional `proc1` descrie logica stării următoare. Procesul secvențial `proc2` conține

descrierea registrului de stare, specificând sincronizarea tranzițiilor de stare cu semnalul de ceas. Procesul combinațional `proc3` specifică semnalele de ieșire ale automatului de stare.

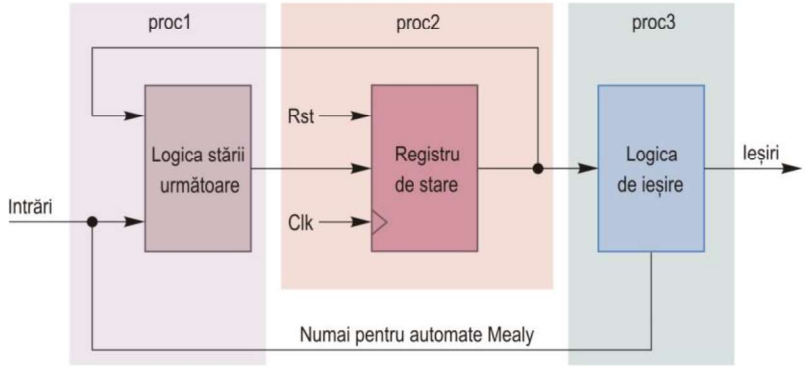


Figura 3.4. Automat de stare cu trei procese.

În cazul acestui tip de descriere, vor fi necesare două semnale de stare, unul pentru starea prezentă (*StarePrez*) și un altul pentru starea următoare (*StareUrm*). Starea următoare este determinată ca o funcție de starea prezentă și de intrări. Deci, lista de sensibilitate a primului proces trebuie să includă aceste semnale, după cum se indică în liniile următoare.

```
proc1: process (StarePrez, Ready, RW)
begin
   ...
end process proc1;
```

Observație

Procesul `proc1` este un proces combinațional, astfel încât este important să se specifice starea următoare în toate condițiile, utilizând clauze `else`, pentru a se evita inserarea circuitelor *latch* de către utilitarul de sinteză.

Procesul `proc1` nu indică momentul în care starea următoare devine stare prezentă. Actualizarea stării prezente este descrisă în al doilea proces, `proc2`, care conține registrul de stare. Modificarea stării prezente se realizează în mod sincron, pe frontul crescător al semnalului de ceas:

```
proc2: process (Clk)
begin
   if RISING_EDGE (Clk) then
      StarePrez <= StareUrm;
   end if;
end process proc2;
```

Automatul de stare necesită un semnal de resetare sincronă. Pentru aceasta, se poate adăuga o instrucțiune `if` la începutul procesului `proc1` pentru ca automatul să treacă în starea `idle` în cazul în care semnalul *Rst* este activ. Secvența necesară pentru resetarea sincronă este următoarea:

```
proc1: process (Rst, StarePrez, Ready, RW)
begin
   if (Rst = '1') then
      StareUrm <= idle;
   else
      case StarePrez is
         ...
      end case;
   end if;
end process proc1;
```

Condiția de resetare sincronă se poate specifica și în procesul secvențial `proc2`, astfel:

```
proc2: process (Clk)
begin
   if RISING_EDGE (Clk) then
      if (Rst = '1') then
         StarePrez <= idle;
      else
         StarePrez <= StareUrm;
      end if;
   end if;
end process proc2;
```

Descrierea controlerului de memorie în care se utilizează trei procese este prezentată în Exemplul 3.2. În această descriere, resetarea sincronă se realizează în procesul `proc2`.

Exemplul 3.2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity contr_mem is
   port (Clk   : in STD_LOGIC;
         Rst   : in STD_LOGIC;
         Ready : in STD_LOGIC;
         RW    : in STD_LOGIC;
         OE    : out STD_LOGIC;
         WE    : out STD_LOGIC);
end contr_mem;

architecture automat_stare of contr_mem is

type TIP_STARE is (idle, decision, read, write);
signal StarePrez, StareUrm : TIP_STARE;

begin
   proc1: process (StarePrez, Ready, RW)
```

```
   begin
      case StarePrez is
         when idle =>
            if (Ready = '1') then
               StareUrm <= decision;
            else                        -- Ready = '0'
               StareUrm <= idle;
            end if;
         when decision =>
            if (RW = '1') then
               StareUrm <= read;
            else                        -- RW = '0'
               StareUrm <= write;
            end if;
         when read =>
            if (Ready = '1') then
               StareUrm <= idle;
            else                        -- Ready = '0'
               StareUrm <= read;
            end if;
         when write =>
            if (Ready = '1') then
               StareUrm <= idle;
            else                        -- Ready = '0'
               StareUrm <= write;
            end if;
      end case;
   end process proc1;

   proc2: process (Clk)
   begin
      if RISING_EDGE (Clk) then
         if (Rst = '1') then
            StarePrez <= idle;
         else
            StarePrez <= StareUrm;
         end if;
      end if;
   end process proc2;

   proc3: process (StarePrez)
   begin
      case StarePrez is
         when idle     => OE <= '0'; WE <= '0';
         when decision => OE <= '0'; WE <= '0';
         when read     => OE <= '1'; WE <= '0';
         when write    => OE <= '0'; WE <= '1';
      end case;
   end process proc3;

end automat_stare;
```

În locul procesului `proc3`, se pot utiliza instrucțiuni condiționale de asignare a semnalelor pentru specificarea semnalelor de ieșire, ca în liniile următoare:

```
   OE <= '1' when StarePrez = read else '0';
   WE <= '1' when StarePrez = write else '0';
```

3.3 Tehnici pentru codificarea stărilor

3.3.1 Ieşiri codificate prin biţii de stare

Tehnica de codificare a ieşirilor prin biţii de stare permite o transmisie mai rapidă a ieşirilor automatului la pinii circuitului. În acest caz, ieşirile automatului de stare sunt reprezentate chiar de biţii de stare. Un numărător este un exemplu de automat de stare pentru care ieşirile reprezintă şi biţii de stare. Această tehnică necesită ca stările să fie codificate cu atenţie, astfel încât ieşirile să corespundă cu valorile păstrate în registrul de stare, după cum se arată în figura 3.5.

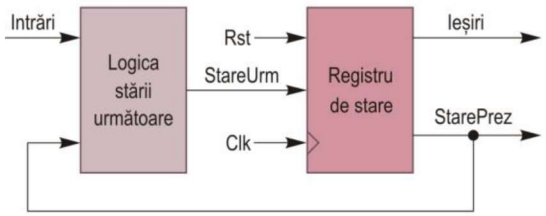


Figura 3.5. Automat Moore cu ieşirile codificate în registrul de stare.

Prin utilizarea acestei tehnici, proiectul va fi mai dificil de înţeles şi modificat, astfel încât tehnica este recomandată numai în cazurile în care sunt necesare anumite optimizări ale resurselor disponibile şi ale performanţelor, optimizări care nu sunt asigurate de sistemul de sinteză în mod automat sau prin utilizarea directivelor de sinteză.

Pentru ilustrarea codificării ieşirilor prin biţii de stare, se va utiliza exemplul controlerului de memorie. Deoarece există patru stări distincte ale automatului, sunt necesari minim doi biţi de stare, însă pot fi necesari biţi suplimentari pentru codificarea ieşirilor. Problema este de a realiza o codificare a stărilor astfel încât ieşirile să reprezinte o parte a biţilor de stare. Pentru această codificare, se întocmeşte mai întâi un tabel cu stările prezente şi ieşirile care trebuie codificate (tabelul 3.2).

Tabelul 3.2. Ieşirile controlerului de memorie ca funcţii de starea prezentă.

Stare	OE	WE
idle	0	0
decision	0	0
read	1	0
write	0	1

Se examinează apoi tabelul, căutându-se combinaţia ieşirilor care apare cu frecvenţa cea mai mare. În cazul în care toate combinaţiile ieşirilor ar fi unice, codificarea ar fi terminată. În acest caz, combinaţia 00 apare de două ori (pentru stările `idle` şi `decision`). Pentru a realiza distincţia dintre cele două stări, se adaugă un bit suplimentar de stare, `St0`. Acest bit se stabileşte în mod arbitrar ca fiind 0 pentru starea

`idle` și 1 pentru starea `decision`. Pentru celelalte stări, valoarea bitului `St0` se stabilește în mod arbitrar ca fiind 0. Codificarea finală a stărilor este prezentată în tabelul 3.3.

Tabelul 3.3. Ieșirile controlerului de memorie codificate prin biții de stare.

Stare	OE	WE	St0
idle	0	0	0
decision	0	0	1
read	1	0	0
write	0	1	0

Pentru descrierea automatului de stare utilizând această tehnică, codificarea stărilor trebuie definită în mod explicit cu ajutorul constantelor. În Exemplul 3.3, semnalul de stare este declarat de tip **`STD_LOGIC_VECTOR`** în locul tipului enumerat `TIP_STARE`, iar codificarea stărilor este specificată prin constante. Definiția entității este aceeași și nu mai este reprodusă.

Exemplul 3.3

```
architecture automat_stare of contr_mem is

signal   Stare    : STD_LOGIC_VECTOR (2 downto 0);
constant idle     : STD_LOGIC_VECTOR (2 downto 0) := "000";
constant decision : STD_LOGIC_VECTOR (2 downto 0) := "001";
constant read     : STD_LOGIC_VECTOR (2 downto 0) := "100";
constant write    : STD_LOGIC_VECTOR (2 downto 0) := "010";

begin
   tranz_st: process (Clk)
   begin
      if RISING_EDGE (Clk) then
         if (Rst = '1') then
            Stare <= idle;
         else
            case Stare is
               when idle =>
                  if (Ready = '1') then
                     Stare <= decision;
                  end if;          -- fara else; memorie implicita
               when decision =>
                  if (RW = '1') then
                     Stare <= read;
                  else             -- RW = '0'
                     Stare <= write;
                  end if;
               when read =>
                  if (Ready = '1') then
                     Stare <= idle;
                  end if;          -- fara else; memorie implicita
               when write =>
                  if (Ready = '1') then
                     Stare <= idle;
                  end if;          -- fara else; memorie implicita
               when others =>
```

```
                Stare <= idle;
            end case;
         end if;
      end if;
   end process tranz_st;

   -- Iesiri asociate cu continutul registrelor
   WE <= Stare(1);
   OE <= Stare(2);

end automat_stare;
```

În acest exemplu, se utilizează un singur proces pentru a descrie și a sincroniza tranzițiile stărilor. Acest proces este similar celui din Exemplul 3.1, cu excepția decodificării ieșirilor, dar în acest caz instrucțiunea `case` conține o ramură suplimentară `when others`. Această ramură este necesară pentru a acoperi toate valorile posibile ale semnalului de stare (acest semnal, de tip **`STD_LOGIC_VECTOR`** cu 3 biți, are 9^3 valori posibile). Deoarece sunt definite numai patru stări, celelalte patru stări din cele 8 posibile cu 3 biți de stare sunt stări ilegale. Deoarece codificarea stărilor a fost declarată în mod explicit și a fost aleasă astfel încât să conțină ieșirile stării prezente, ieșirile pot fi asignate direct din semnalul de stare.

3.3.2 Codificarea cu un bistabil pe stare

Tehnica de codificare cu un bistabil pe stare utilizează *n* bistabile pentru a reprezenta un automat cu *n* stări. Pentru fiecare stare există câte un bistabil, un singur bistabil fiind setat la un moment dat. Decodificarea stării prezente constă în simpla identificare a bistabilului care este setat. Tranziția dintr-o stare în alta constă în modificarea stării bistabilului corespunzător stării vechi din 1 în 0 și a stării bistabilului corespunzător stării noi din 0 în 1.

Avantajul principal al automatelor care utilizează codificarea cu un bistabil pe stare este că numărul de porți necesare pentru decodificarea informației de stare pentru semnalele de ieșire și pentru tranzițiile stărilor este mult mai redus decât numărul de porți necesare în cazul utilizării altor tehnici. Această diferență de complexitate crește pe măsură ce numărul de stări devine mai mare. Pentru ilustrarea acestui aspect, considerăm un automat cu 18 stări codificate prin două tehnici, codificarea secvențială și codificarea cu un bistabil pe stare. Pentru codificarea secvențială, sunt necesare cinci bistabile, iar pentru codificarea cu un bistabil pe stare sunt necesare 18 bistabile. Presupunem că figura 3.6 reprezintă o porțiune a diagramei de stare indicând toate tranzițiile posibile în starea `Stare15`.

Fragmentul de cod reprezentând această porțiune a diagramei de stare este următorul:

```
case StarePrez is
   when Stare2 =>
      if Cond1 = '1' then
         StareUrm <= Stare15;
      else ...
```

```
   when Stare15 =>
      if ...
      elsif Cond3 = '0' then
         StareUrm <= Stare15;
      else ...
   when Stare17 =>
      if ...
      elsif Cond2 = '1' then
         StareUrm <= Stare15;
      else ...
end case;
```

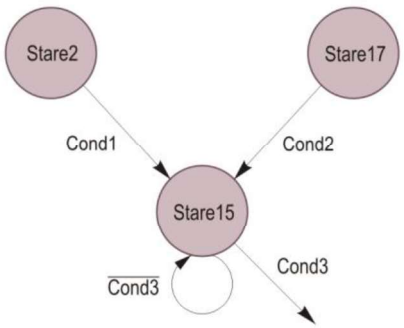


Figura 3.6. Porțiune a diagramei de stare a unui automat indicând toate tranzițiile posibile în Stare15.

Vom examina mai întâi logica stării următoare pentru codificarea secvențială. Notăm vectorul de stare pentru codificarea secvențială cu S, acest vector fiind de cinci biți, $S_4S_3S_2S_1S_0$. Codificarea secvențială pentru starea `Stare15` este 01111. Pe baza diagramei de stare din Figura 3.6, se poate scrie, pentru fiecare din cei cinci biți ai vectorului S, o ecuație reprezentând condițiile care determină setarea bitului respectiv datorită unei tranziții în starea `Stare15`:

$$\begin{aligned} S_{i,15} \quad = & \overline{S_4} \cdot \overline{S_3} \cdot \overline{S_2} \cdot S_1 \cdot \overline{S_0} \cdot Cond1 + \\ & S_4 \cdot \overline{S_3} \cdot \overline{S_2} \cdot \overline{S_1} \cdot S_0 \cdot Cond2 + \\ & \overline{S_4} \cdot S_3 \cdot S_2 \cdot S_1 \cdot S_0 \cdot \overline{Cond3} \end{aligned} \tag{3.5}$$

unde $0 \leq i \leq 4$, S_i reprezintă unul din cei cinci biți ai vectorului S, iar $S_{i,15}$ reprezintă ecuațiile care determină ca bitul S_i să fie setat datorită tranziției în starea `Stare15`. Expresia booleană corespunzătoare bitului S_4 ($S_{4,15}$) este 0, deoarece nicio tranziție în starea `Stare15` nu determină setarea bitului cel mai semnificativ (S_4) al vectorului de stare.

Deși ecuația (3.5) definește bitul S_i pentru tranzițiile în starea `Stare15`, aceasta nu este suficientă pentru a specifica biții S_0, S_1, S_2 și S_3. De exemplu, ecuația pentru $S_{0,15}$ acoperă numai cazurile pentru care bitul S_0 este setat datorită tranzițiilor în starea `Stare15`. Bitul S_0 este setat și datorită tranzițiilor în celelalte stări cu număr impar. Ecuațiile booleene asociate cu aceste tranziții pot fi similare ca și complexitate cu

ecuațiile pentru $S_{0,15}$. Pentru obținerea ecuației complete corespunzătoare bitului S_i, trebuie să se însumeze fiecare din ecuațiile $S_{i,n}$, unde n este un întreg de la 0 la 17, reprezentând cele 18 stări. Rezultă că logica pentru bitul S_i poate fi complexă, chiar și pentru o diagramă de stare relativ simplă. În general, prin codificarea secvențială vor rezulta cinci ecuații complexe pentru S_i.

Notăm vectorul de stare pentru codificarea cu un bistabil pe stare cu T, acest vector fiind de 18 biți. Bitul 15 al acestui vector, corespunzător stării `Stare15`, are ecuația (3.6):

$$T_{15} = T_2 \cdot Cond1 + T_{17} \cdot Cond2 + T_{15} \cdot \overline{Cond3} \tag{3.6}$$

Această ecuație poate fi dedusă în mod simplu pe baza diagramei de stare. Figura 3.7 prezintă implementarea logicii pentru tranzițiile în starea `Stare15`. În timp ce codificarea secvențială necesită cinci ecuații complexe pentru logica stării următoare, codificarea cu un bistabil pe stare necesită 18 ecuații simple. În funcție de arhitectura circuitului utilizat pentru implementare, un automat de stare care utilizează codificarea cu un bistabil pe stare poate necesita o cantitate semnificativ mai redusă de resurse pentru implementare decât un automat care utilizează alte metode de codificare. De asemenea, logica stării următoare necesită, de obicei, un număr mai redus de nivele logice între registrele de stare, ceea ce permite o frecvență mai ridicată de funcționare.

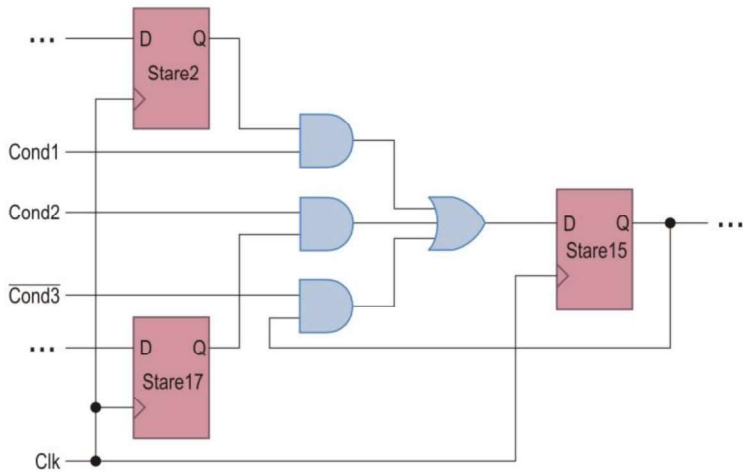


Figura 3.7. Logica stării următoare pentru tranzițiile în Stare15 în cazul automatului care utilizează codificarea cu un bistabil pe stare.

Codificarea cu un bistabil pe stare nu reprezintă însă soluția optimă în toate cazurile, mai ales datorită faptului că necesită un număr mai mare de bistabile decât codificarea secvențială. În general, codificarea cu un bistabil pe stare este avantajoasă atunci când arhitectura circuitului programabil utilizat conține un număr relativ mare

de bistabile și un număr relativ redus de porți logice între bistabile. De exemplu, această codificare este cea mai avantajoasă pentru automatele de stare implementate cu circuite FPGA, care conțin un număr mai mare de bistabile decât circuitele CPLD. Codificarea cu un bistabil pe stare poate fi soluția optimă chiar și pentru circuitele CPLD atunci când logica stării următoare necesită treceri multiple prin rețeaua logică.

În general, utilizarea codificării cu un bistabil pe stare nu necesită modificarea codului sursă. Multe utilitare de sinteză permit utilizarea unei directive de sinteză pentru specificarea codificării cu un bistabil pe stare, ca și a altor tehnici de codificare. Aceste directive pot fi sub forma unor opțiuni ale interfeței grafice, parametri ai liniei de comandă sau atribute ale limbajului. Aceste atribute sunt specifice utilitarului de sinteză. Secțiunea 3.3.3 prezintă modul în care se poate specifica utilizarea codificării cu un bistabil pe stare pentru utilitarul de sinteză al mediului de proiectare Vivado.

Generarea semnalelor de ieșire pentru automatele codificate cu un bistabil pe stare este similară cu generarea acestor semnale la automatele la care ieșirile sunt decodificate din registrul de stare. Decodificarea este însă foarte simplă, deoarece stările sunt reprezentate prin câte un singur bit și nu printr-un vector întreg. Logica de ieșire constă dintr-o poartă SAU, deoarece automatele Moore au ieșiri care sunt funcții de stările automatului, iar fiecare stare este reprezentată printr-un singur bit. Decodificarea ieșirilor adaugă un nivel logic suplimentar și o întârziere corespunzătoare, ca și în cazul în care biții de stare sunt codificați.

Dacă o anumită ieșire este activată într-o singură stare, aceasta va fi codificată în mod automat prin biții de stare. De exemplu, considerând controlerul de memorie, semnalul *WE* este activat numai în timpul stării `write`:

```
WE <= '1' when StarePrez = write else '0';
```

Există un bistabil asociat în mod direct cu starea `write`, astfel încât valoarea semnalului *WE* va fi dată de starea acestui bistabil. În consecință, acest semnal va fi disponibil la pinii circuitului sau pentru logica internă fără întârzierea suplimentară asociată cu decodificarea ieșirilor.

3.3.3 Codificarea stărilor în mediul de proiectare Vivado

Utilitarul de sinteză al mediului de proiectare Xilinx Vivado Design Suite permite utilizarea tehnicilor de codificare a stărilor care sunt prezentate în continuare.

- **Auto**: codificarea automată; este tehnica de codificare utilizată în mod implicit. Atunci când se utilizează această opțiune, modulul de sinteză încearcă să selecteze tehnica de codificare cea mai avantajoasă din algoritmii disponibili pentru fiecare automat de stare.
- **One-Hot**: codificarea cu un bistabil pe stare; la această codificare, în fiecare ciclu de ceas este setat un singur bit al registrului de stare. Această tehnică este avantajoasă pentru cele mai multe circuite FPGA, deoarece permite reducerea

logicii necesare, creșterea frecvenței de funcționare și reducerea puterii consumate.

- **Sequential**: codificarea secvențială, care asignează coduri binare succesive stărilor, ceea ce minimizează ecuațiile stării următoare.
- **Gray**: codificarea Gray, prin care va comuta un singur bit al registrului de stare între două stări consecutive, ceea ce minimizează hazardurile. Tehnica de codificare Gray este avantajoasă în mod special pentru automate de stare cu secvențe lungi de stări, fără ramificații între stări.
- **Johnson**: codificarea Johnson, care garantează faptul că va comuta un singur bit al registrului de stare între două stări consecutive, similar cu codificarea Gray.

În mediul de proiectare Vivado, tehnica de codificare a stărilor poate fi specificată prin setarea unei opțiuni de sinteză, prin care este setată o constrângere numită `fsm_encoding`. Pentru aceasta, trebuie selectată opțiunea **Settings** din panoul *Flow Navigator*, apoi trebuie selectată linia **Synthesis** din zona *Project Settings* și trebuie selectată o valoare diferită de **off** pentru parametrul *fsm_extraction*. Dacă pentru acest parametru este selectată valoarea **off**, atunci constrângerea `fsm_extract` va fi setată la **no** și pentru sinteză se va utiliza metoda de codificare implicită (**auto**).

În codul VHDL, tehnica de codificare a stărilor poate fi specificată prin constrângerea *FSM Encoding Algorithm* (`fsm_encoding`). Pentru a specifica o opțiune pentru această constrângere, trebuie validată mai întâi constrângerea *Automatic FSM Extraction* (`fsm_extract`) utilizând următoarele atribute:

```
attribute fsm_extract : STRING;
attribute fsm_extract of {nume_entitate | nume_semnal} :
          {entity | signal} is "yes";
```

După validarea constrângerii `fsm_extract`, poate fi specificată constrângerea `fsm_encoding` utilizând următoarele atribute:

```
attribute fsm_encoding : STRING;
attribute fsm_encoding of {nume_entitate | nume_semnal} :
          {entity | signal} is "{auto | one_hot |
          sequential | johnson | gray}";
```

3.4. Toleranța la defecte a automatelor de stare

Atunci când se realizează sinteza automatelor de stare ale căror stări sunt definite utilizând tipuri enumerate, semnalele de stare sunt convertite în vectori. Fiecărei valori a tipului stărilor i se asignează un cod. Pentru codificarea cu un bistabil pe stare, fiecare valoare a semnalului de stare corespunde unui bistabil. Pentru alte tehnici de codificare, cum este cea secvențială, numărul minim de stări necesare este valoarea întreagă cea mai mare pentru $\log_2 n$, $\lceil \log_2 n \rceil$, unde n este numărul stărilor. În cazul în care $\log_2 n$ nu este o valoare întreagă, vor exista stări nedefinite. Asemenea

stări pot exista și în cazul automatelor la care stările sunt codificate explicit, cum este automatul de stare din Exemplul 3.3.

Dacă automatul de stare ar trece într-o stare nedefinită sau ilegală, acesta nu ar funcționa într-un mod predictibil. Comportamentul automatului de stare atunci când acesta trece într-o stare ilegală depinde de ecuațiile de tranziție ale stărilor. Este posibilă specificarea tranzițiilor dintr-o stare ilegală sub forma unor condiții indiferente. Avantajul utilizării unor asemenea condiții constă în faptul că nu este nevoie de o logică suplimentară pentru a asigura că automatul va ieși din această stare. O asemenea logică suplimentară poate necesita resurse substanțiale ale circuitului pentru implementare, în special dacă există un număr mare de stări nedefinite. Dezavantajul utilizării condițiilor indiferente este că automatul de stare este mai puțin tolerant la defecte. Proiectantul trebuie să decidă dacă aceasta este o soluție acceptabilă pentru o anumită aplicație.

În practică, anumite hazarduri, zgomote sau combinații ilegale ale intrărilor pot determina modificarea stării unuia sau a mai multor bistabile, ceea ce poate avea ca efect tranziția automatului într-o stare ilegală. Automatul poate rămâne definitiv în această stare ilegală, sau poate activa o combinație ilegală a ieșirilor, ceea ce poate cauza alte efecte nedorite.

Automatele de stare pot fi proiectate astfel încât să fie tolerante la defecte prin adăugarea unei logici care să asigure ieșirea din stările ilegale. Pentru adăugarea acestei logici, se poate proceda în modul descris în continuare.

- Mai întâi, proiectantul trebuie să determine numărul posibil al stărilor ilegale. Acest număr este egal cu diferența dintre pătratul numărului bistabilelor utilizate pentru codificarea stărilor și numărul stărilor automatului.
- Dacă se utilizează un tip enumerat, proiectantul trebuie să includă un nume de stare în cadrul tipului enumerat pentru fiecare stare nedefinită (de exemplu, `nedefinit`).
- În sfârșit, proiectantul trebuie să specifice o tranziție a automatului de stare pentru ca acesta să iasă din starea ilegală. Această tranziție poate fi specificată sub forma următoare:

```
case StarePrez is
   ...
   when nedefinit => Stare <= idle;
end case;
```

În Exemplul 3.3, tranziția din stările nedefinite în starea inițială este specificată sub forma următoare:

```
when others => Stare <= idle;
```

Prin specificarea tranziției din stările ilegale într-o stare cunoscută se va genera o logică suplimentară. Există cazuri în care costul acestei soluții nu este justificat de

necesitatea unui automat tolerant la defecte. În aceste cazuri, se poate specifica în mod explicit faptul că tranziția dintr-o stare ilegală este o condiție indiferentă (deci, nu are importanță starea în care se efectuează tranziția dintr-o stare ilegală, deoarece nu este de așteptat ca automatul să treacă într-o asemenea stare). În cazul codificării explicite, condițiile indiferente pot fi declarate sub forma:

```
when others => Stare <= "---";
```

unde s-a presupus că semnalul *Stare* este un vector de trei biți.

În timp ce condițiile indiferente sunt implicite pentru automatele la care se utilizează tipuri enumerate atunci când nu se specifică toate combinațiile posibile ale valorilor, condițiile indiferente trebuie definite în mod explicit pentru automatele cu stările codificate explicit. Utilizarea constantelor permite definirea explicită atât a condițiilor indiferente, cât și a tranzițiilor din stările ilegale.

Posibilitatea ca automatul să treacă într-o stare ilegală crește atunci când se utilizează codificarea cu un bistabil pe stare sau când ieșirile sunt codificate prin biții de stare. În ambele cazuri, poate exista un număr mare de stări ilegale sau nedefinite. De exemplu, în cazul codificării cu un bistabil pe stare, există 2^n valori posibile ale vectorului de stare de n biți, în timp ce automatul are doar n stări. Deși codificarea cu un bistabil pe stare se alege, de obicei, pentru a obține o implementare eficientă a automatului de stare, dacă se include logica pentru tranziția din oricare stare ilegală în starea inițială sau într-o altă stare cunoscută, va rezulta o implementare ineficientă. De exemplu, pentru specificarea completă a tranzițiilor stărilor unui automat cu 18 stări la care se utilizează codificarea cu un bistabil pe stare, trebuie decodificate în plus $2^{18} - 18 = 262.126$ tranziții.

În locul adăugării unei logici pentru tranziția din toate stările ilegale într-o stare cunoscută, se poate include o logică pentru detectarea situațiilor în care este setat mai mult decât un bistabil la un moment dat. Se poate genera un semnal de coliziune pentru a detecta setarea mai multor bistabile în același timp. Pentru cazul în care există patru stări notate cu `Stare1`, …, `Stare4`, semnalul de coliziune va avea forma următoare:

```
Coliziune <= (Stare1 and (Stare2 or Stare3 or Stare4)) or
             (Stare2 and (Stare1 or Stare3 or Stare4)) or
             (Stare3 and (Stare1 or Stare2 or Stare4)) or
             (Stare4 and (Stare1 or Stare2 or Stare3));
```

Această tehnică se poate extinde pentru un număr oarecare de stări. Dacă nu este necesar ca semnalul de coliziune să fie generat într-un singur ciclu de ceas, se poate utiliza tehnica *pipeline* pentru generarea acestui semnal. Prin utilizarea acestei tehnici, frecvența de funcționare poate fi menținută la valoarea maximă.

Proiectantul trebuie să decidă care este cantitatea de resurse care poate fi utilizată pentru creșterea toleranței la defecte a automatului realizat, sau în ce măsură scăderea vitezei de funcționare datorită logicii suplimentare este acceptabilă.

Aplicații

3.1 Răspundeți la următoarele întrebări:

a. Care este principiul codificării ieșirilor prin biții de stare? Care sunt avantajele și dezavantajele acestei tehnici?

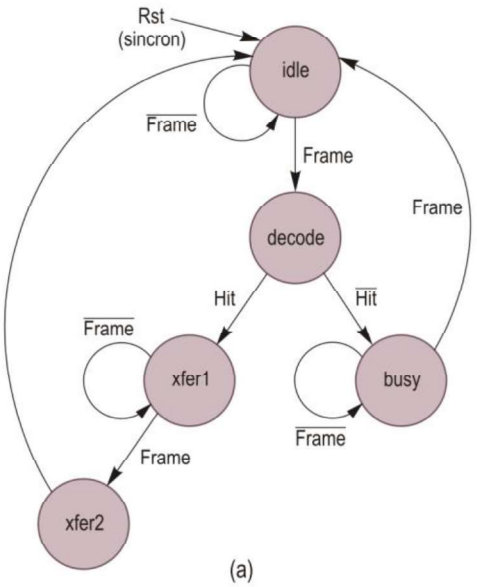
b. Care este principiul codificării cu un bistabil pe stare și care este avantajul principal al acestei tehnici?

c. Cum pot fi proiectate automatele de stare pentru a fi tolerante la defecte?

3.2 Considerăm un automat de stare cu trei stări (S0, S1, S2) și trei ieșiri (*OutA*, *OutB*, *OutC*). Ieșirea *OutA* trebuie activată numai în starea S0, ieșirea *OutB* trebuie activată numai în starea S1, iar ieșirea *OutC* trebuie activată numai în starea S2. O secvență posibilă pentru generarea ieșirilor este următoarea:

```
if (StarePrez = S0) then
   OutA <= '1';
elsif (StarePrez = S1) then
   OutB <= '1';
else                    -- StarePrez = S2
   OutC <= '1';
end if;
```

Explicați motivul pentru care această secvență corectă din punct de vedere sintactic nu este corectă din punct de vedere funcțional.



Stare	OE	GO	ACT
idle	0	0	0
decode	0	0	0
busy	0	0	1
xfer1	1	1	1
xfer2	1	0	1

(b)

Figura 3.8. Automat de stare pentru aplicația 3.3: (*a*) diagrama de stare; (*b*) semnalele de ieșire.

3.3 Proiectați automatul de stare din figura 3.8 utilizând un tip enumerat pentru definirea stărilor. Creați un modul al bancului de test și simulați funcționarea automatului de stare cu simulatorul Vivado.

3.4 Modificați descrierea automatului de stare din aplicația 3.3 astfel încât ieșirile să fie codificate prin biții de stare.

3.5 Proiectați un automat de stare pentru verificarea parității pare a fiecărui octet recepționat pe o linie serială. Intrările automatului sunt semnalul de ceas, un semnal de resetare asincronă, linia *Data* pe care se recepționează datele în mod serial (bit cu bit) și un semnal de sincronizare *Sync*. Recepția unui octet de date începe la următorul front crescător al semnalului de ceas după ce se detectează că semnalul *Sync* are valoarea logică 0 și se continuă la fiecare front crescător al semnalului de ceas. Ieșirea automatului este semnalul *PError*, care este setat la valoarea logică 1 la terminarea recepției unui octet dacă paritatea octetului este impară (octetul conține un număr impar de biți de 1) și la valoarea logică 0 în caz contrar. Semnalul *PError* rămâne activat până la începutul recepției următorului octet. Desenați diagrama de stare a automatului. Descrieți automatul utilizând un tip enumerat pentru definirea stărilor. Creați un modul al bancului de test și simulați funcționarea automatului de stare cu simulatorul Vivado.

3.6 Proiectați un automat de stare pentru detectarea secvenței de biți "11101011". Secvența se aplică serial la intrarea automatului începând cu bitul cel mai semnificativ. Desenați diagrama de stare a automatului, iar apoi descrieți automatul utilizând un tip enumerat pentru definirea stărilor. Creați un modul al bancului de test și simulați funcționarea automatului de stare cu simulatorul Vivado.

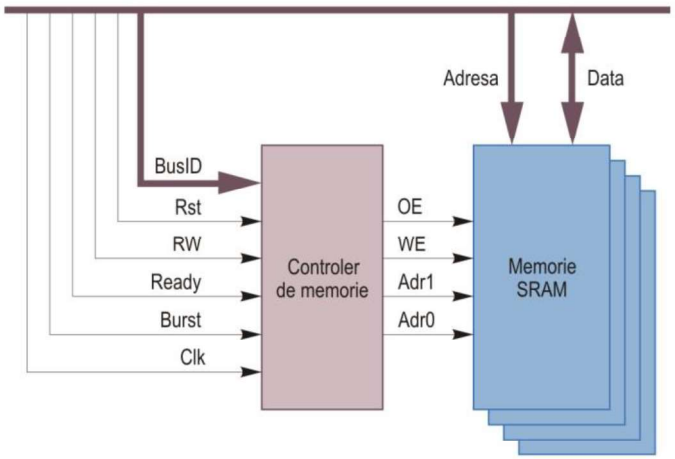


Figura 3.9. Schema bloc a sistemului conținând controlerul de memorie care permite citirea în mod exploziv.

3.7 Proiectați un controler de memorie care permite citirea din memorie a unui număr de patru cuvinte în mod exploziv. Schema bloc a unui sistem care utilizează acest controler de memorie este ilustrată în figura 3.9.

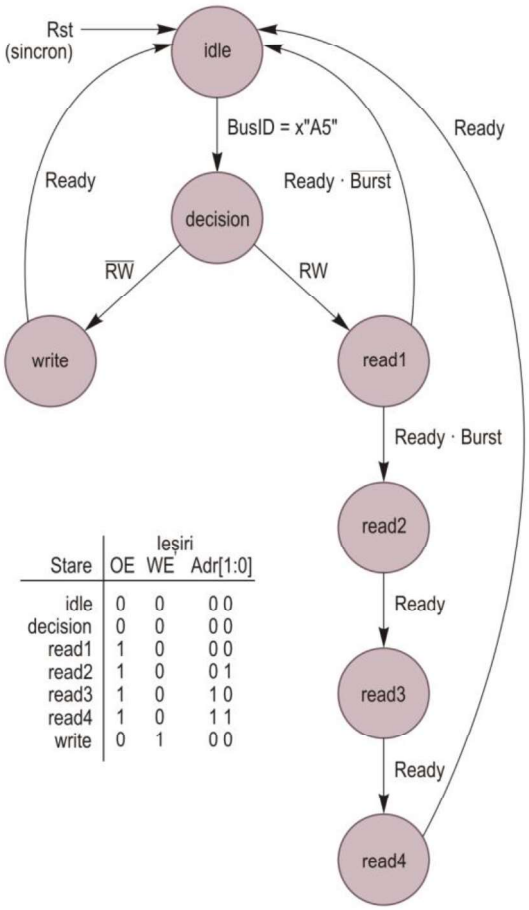


Figura 3.10. Diagrama de stare a controlerului de memorie care permite citirea în mod exploziv.

Circuitele conectate la magistrală inițiază un acces la memorie prin setarea identificatorului de magistrală *BusID* la valoarea x"A5". În următorul ciclu de ceas, se activează semnalul *RW* pentru a indica o citire din memorie, sau se dezactivează acest semnal pentru a indica o scriere în memorie. Dacă operația executată este una de citire, se poate citi fie un singur cuvânt, fie un grup de patru cuvinte în mod exploziv. O citire în mod exploziv este indicată prin activarea semnalului *Burst* în timpul primului ciclu de citire, după care controlerul accesează patru locații de memorie. Locațiile consecutive sunt accesate după activarea succesivă

a semnalului *Ready*. Controlerul activează semnalul de ieșire *OE* în timpul unei operații de citire și incrementează cei doi biți mai puțin semnificativi ai adresei în timpul unei citiri în mod exploziv.

La o operație de scriere în memorie, se scrie întotdeauna un singur cuvânt. În timpul unei operații de scriere, controlerul activează semnalul *WE*, permițând scrierea datelor în locația de memorie specificată de adresa *Adr*. Un acces de citire sau scriere este terminat la activarea semnalului *Ready*.

Figura 3.10 prezintă diagrama de stare a controlerului de memorie. Un semnal de resetare sincronă plasează automatul în starea `idle`. Atunci când memoria nu este accesată, controlerul rămâne în starea `idle`. Dacă vectorul *BusID* este setat la valoarea x"A5" în timp ce controlerul este în starea `idle`, automatul trece în starea `decision`. În următorul ciclu de ceas, controlerul trece fie în starea `read1`, fie în starea `write`, în funcție de starea semnalului *RW*. Dacă accesul este pentru citire, citirea unui singur cuvânt este indicată prin activarea semnalului *Ready* în starea `read1`, fără activarea semnalului *Burst*. În acest caz, controlerul revine în starea `idle`. O citire în mod exploziv este indicată prin activarea ambelor semnale *Ready* și *Burst* în starea `read1`. În acest caz, automatul trece prin fiecare din stările de citire (`read2`, `read3`, `read4`). În timpul fiecărei stări de citire este activat semnalul *OE* și este incrementată adresa *Adr*. Dacă accesul este pentru scriere, ceea ce se indică prin dezactivarea semnalului *RW*, controlerul activează semnalul *WE*, așteaptă semnalul *Ready* de la magistrală, iar apoi revine în starea `idle`.

Creați un modul al bancului de test și simulați funcționarea acestui automat de stare cu simulatorul Vivado.