Proiectarea automatelor de stare cu VHDL

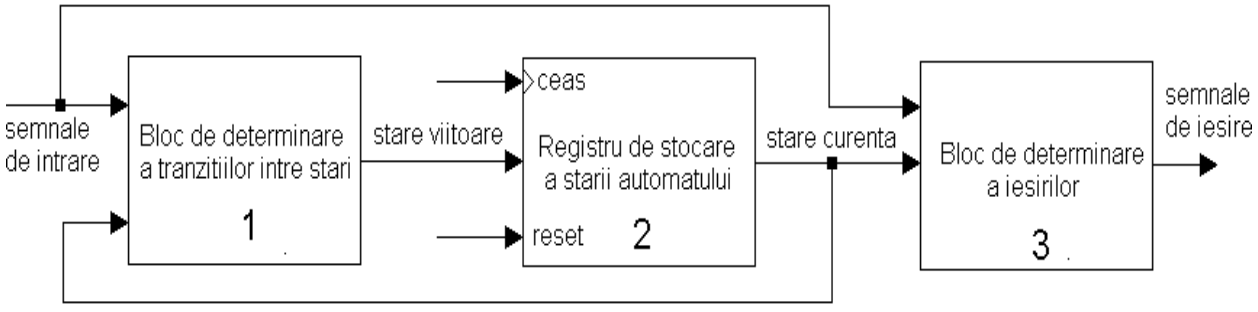
În domeniul circuitelor digitale, în majoritatea cazurilor, proiectanţii trebuie să proiecteze circuite care efectuează anumite secvenţe specifice de operaţii, de exemplu, controlere utilizate pentru comanda funcţionării altor circuite. Automatele de stare reprezintă o metodă foarte eficientă pentru modelarea circuitelor secvenţiale. Prin modelarea automatelor de stare într-un limbaj de descriere hardware şi utilizarea programelor de sinteză proiectanţii se pot concentra doar asupra modelării secvenţei dorite de operaţii fără a-şi pune probleme în privinţa implementării circuitului. Această operaţie este lăsată programelor de sinteză.

Un automat de stare este un circuit secvenţial cu mai multe stări interne. În comparaţie cu circuitele secvenţiale obişnuite (numărătoare, regiştri), tranziţiile dintr-o stare în alta şi secvenţa de evenimente este mult mai complicată. Deşi diagrama bloc de bază a unui automat de stare este similară cu cea a unui circuit secvenţial obişnuit, procedura de proiectare este diferită. Astfel, modelul unui automat de stare se construieşte plecând de la un model mult mai abstract, cum ar fi o diagramă de stări, care descrie în format grafic interacţiunile şi tranziţiile dintre stările interne.

Formal, un automat de stare este specificat de 5 elemente: stările simbolice, semnalele de intrare, semnalele de ieşire, funcţia pentru stabilirea stării următoare şi funcţia pentru stabilirea valorilor de la ieşire.

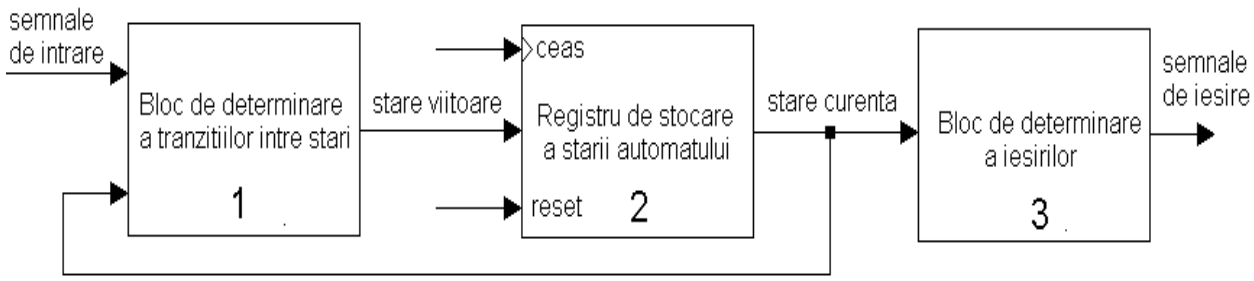
După tipul funcţiei pentru stabilirea valorilor de la ieşire, automatele de stare se împart în două categorii: *Mealy* şi *Moore*.

În cazul automatelor Mealy, semnalele de ieşire depind atât de starea curentă, cât şi de intrările prezente.



Automat MEALY

În cazul circuitelor secvenţiale Moore, ieşirile sunt dependente numai de starea curentă, fără să depindă în mod direct de intrări.



Automat MOORE

Metoda Mealy permite implementarea unui anumit circuit printr-un număr minim de elemente de memorare (bistabile), însă eventualele variaţii necontrolate ale semnalelor de intrare se pot transmite semnalelor de ieşire. Proiectarea prin metoda Moore necesită mai multe elemente de memorare pentru acelaşi tip de comportament, dar funcţionarea circuitului este mai sigură.

Implementarea in VHDL

Spre deosebire de alte limbaje de descriere hardware, VHDL nu este prevăzut cu elemente de limbaj speciale pentru modelarea automatelor de stare. De aceea, pentru descrierea funcţională a automatelor de stare se utilizează o simplă translatare a diagramei de stare în instrucţiuni ***case*** şi ***if***.

Particularități:

1. Tipul stărilor este definit de utilizator. De obicei se utilizează tipul enumerare.
   TYPE *state_type* **IS** (idle, tap1, tap2, tap3, tap4);
2. Se folosesc semnale sau variabile interne în arhitectură pentru memorarea stărilor.
   SIGNAL filter : *state_type*;
3. Pentru a determina tranziția în starea următoare se folosește insrucțiunea **case.**
4. Pentru a determina semnalele de ieșire se folosește fie un **proces** combinațional cu instrucțiunea **case**, fie **asignarea condițională** sau **selectivă** de semnal.

Pentru descrierea automatului de stare, se pot utiliza diferite stiluri de descriere:

1. Un singur proces

- conţine atât tranziţia stărilor cât şi funcţiile de ieşire;

2. Două procese
   - proces combinaţional care conţine logica stării următoare şi logica de ieşire;
   - proces secvenţial pentru actualizarea stării prezente, sincron cu semnalul de ceas;
3. Trei procese
   - proces combinaţional care conţine logica stării următoare ;
   - proces secvenţial pentru actualizarea stării prezente, sincron cu semnalul de ceas;
   - proces combinaţional care conţine logica de ieşire;

Iniţializare sincronă şi asincronă

Orice automat de stare necesită o iniţializare. Dacă aceasta nu este inclusă, semnalele şi variabilele de tipul std_logic se vor afla în starea `U` (neiniţializat).

Vom analiza ***iniţializarea sincronă***. Considerăm că automatul de stare trece din orice stare în starea iniţială când semnalul reset este egal cu 0.

```
state_machine  :  process (clk)
 begin
   if (clk'EVENT AND clk = '1') then
    if (reset = '0') then
          state_vector <= idle;
         else
     -- condiţiile de tranziţie a stărilor
    end if;
  end if;
end process state_machine;
```

Deoarece semnalul ***reset*** nu a fost inclus în lista de sensibilitate, doar semnalul ***clk*** va activa procesul. A doua condiţie ***if*** va fi evaluată doar atunci când semnalul de ceas va trece din `0` în `1`. ***State_vector*** poate fi un semnal sau variabilă, care trece în starea ***idle*** atunci când semnalul reset este `0` pe frontul crescător al semnalului de ceas.

În cazul ***iniţializării asincrone*** semnalul ***reset*** va fi inclus în lista de sensibilitate a procesului. Astfel procesul se va activa când unul dintre semnalele clk şi reset se va modifica.

```
state_machine : process (clk, reset)
  begin
   if (reset = '0') then              -- resetare asincronă pe `0`
```

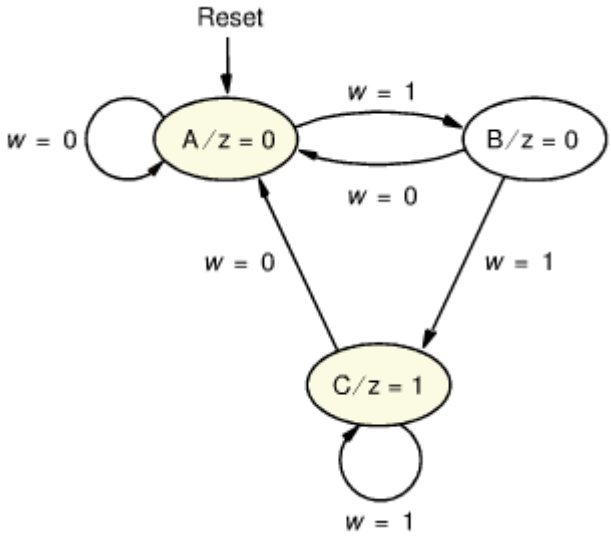
```
   state_vector <= idle;              -- trecere în starea inițială
  elsif (clk'EVENT AND clk = '1') then -- sincron cu frontul pozitiv
  -- condițiile de tranziție a stărilor
  end if;
end process state_machine;
```

Semnalul reset are prioritate față de semnalul clk, deoarece instrucțiunile se evaluează în ordine succesivă, deci condiția de resetare este analizată prima.

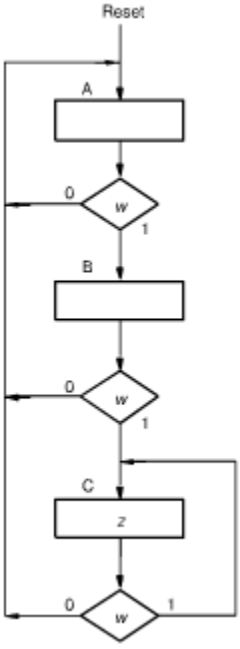
Reprezentarea automatelor finite

Automatele finite pot fi reprezentate prin digrame de stare (sisteme simple) sau scheme bloc (sisteme complexe). Ambele reprezentări utilizează notații simbolice care arată tranziția dintre stări și valorile de la ieșire în dependență de anumite condiții.

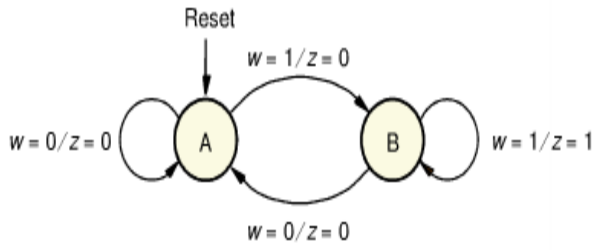
Exemplu de diagramă de stare pentru automatul Moore:



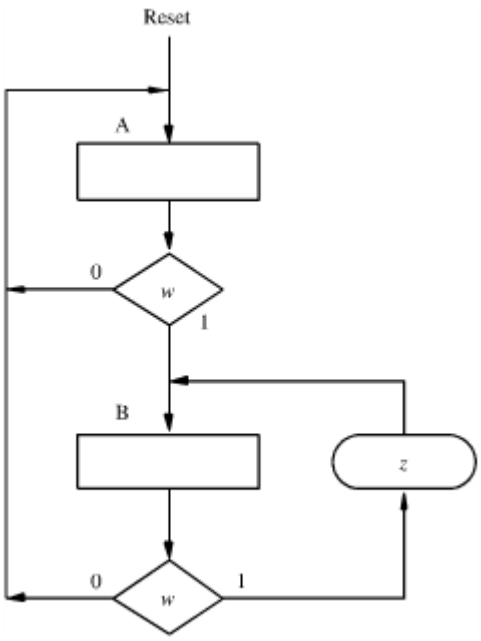
Exemplu de schemă bloc (Automatul Moore):



Pentru automatul Mealy:



Pentru automatul Mealy:

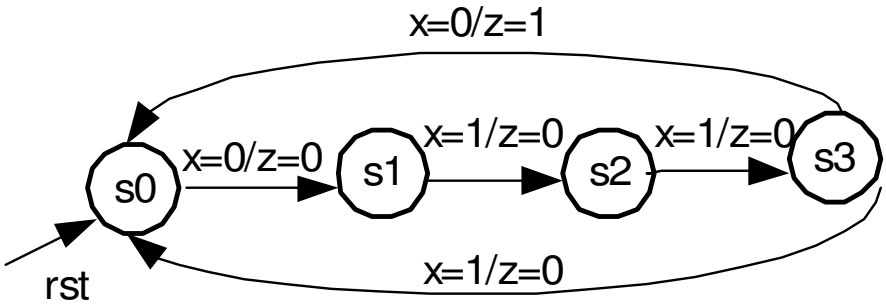


Exemple de automate.

Să se efectuieze sinteza unui detector de secvență. Detectorul generează la ieșire valoarea 1 logic doar atunci când la intrare este aplicată secvența 0110.

Automatul Mealy.

Diagrama de stare:

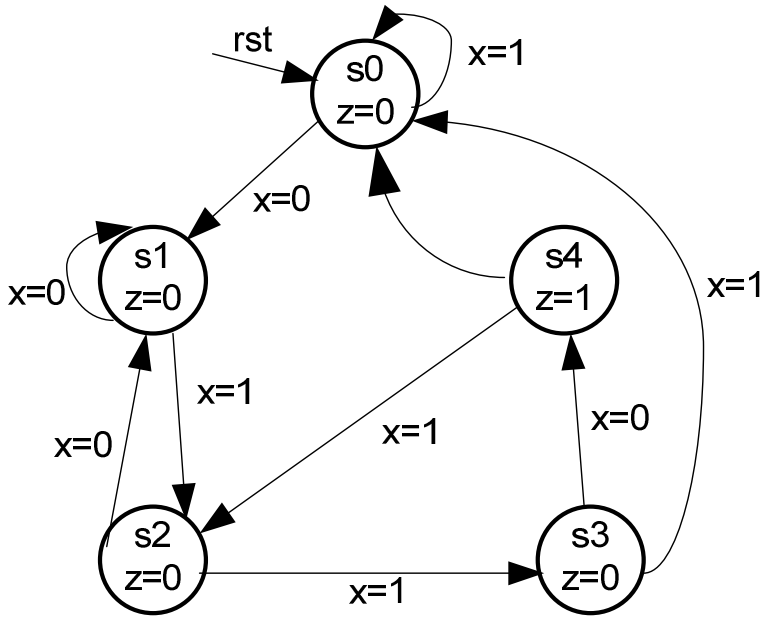


Codul VHDL pentru automatul Mealy descris cu 2 procese.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity mealy2 is
Port (CLK, rst,x: in std_logic;
```

```
        Z: out std_logic);
End entity mealy2;
Architecture Mealy2_arch of mealy2 is
Type State is (s0, s1, s2, s3);
Signal cs, ns: state;
Begin
Secv: process (CLK, rst) is
begin
If (rising_edge(clk)) then
If (rst='1') then cs<=s0;  else cs<=ns;
end if;
end if;
end process secv;
com: process (cs, x) is
begin
z<='0';
case cs is
when s0 =>  if (x='0')  then ns<=s1; else ns<=s0; end if;
when s1=>  if (x='0')  then ns<=s1; else ns<=s2; end if;
when s2 =>  if (x='0')  then ns<=s1; else ns<=s3; end if;
when s3 =>  if (x='0')  then ns<=s1; z<='1'; else ns<=s0; end if;
end case;
end process com;
end architecture Mealy2_arch;
```

Diagrama de stare pentru automatul Moore



Codul VHDL pentru automatul Moore descris cu 2 procese.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity moore2 is
Port (CLK, rst,x: in std_logic;
            Z: out std_logic);
End entity moore2;
Architecture moore2_arch of moore2 is
Type State is (s0, s1, s2, s3, s4);
Signal cs, ns: state;
Begin
Secv: process (CLK, rst) is
begin
If (rising_edge(clk)) then
If (rst='1') then cs<=s0;  else cs<=ns;
end if;
end if;
end process secv;
com: process (cs, x) is
begin
z<='0';
case cs is
```

```
when s0 =>  if (x='0')  then ns<=s1; else ns<=s0; end if;
when s1=>  if (x='0')  then ns<=s1; else ns<=s2; end if;
when s2 =>  if (x='0')  then ns<=s1; else ns<=s3; end if;
when s3 =>  if (x='0')  then ns<=s4; else ns<=s0; end if;
when s4 =>  z <='1'; ns <= s0;
end case;
end process com;
end architecture Moore2_arch;
```

Codul VHDL pentru automatul Moore descris cu 3 procese.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity Moore3 is
Port (CLK, rst,x: in std_logic;
            Z: out std_logic);
End entity Moore3;
Architecture Moore3_arch of Moore3 is
Type State is (s0, s1, s2, s3, s4);
Signal cs, ns: state;
Begin
Secv: process (CLK) is
begin
If (rising_edge(clk)) then
If (rst='1') then cs<=s0;  else cs<=ns;
end if;
end if;
end process secv;
com: process (cs, x) is
begin
case cs is
when s0 =>  if (x='0')  then ns<=s1; else ns<=s0; end if;
when s1=>  if (x='0')  then ns<=s1; else ns<=s2; end if;
when s2 =>  if (x='0')  then ns<=s1; else ns<=s3; end if;
when s3 =>  if (x='0')  then ns<=s4; else ns<=s0; end if;
when s4 =>   ns <= s0;
end case;
end process com;
```

```
outputz: process (cs) is
begin
case cs is
when s0 | s1 | s2 | s3 => z<='0';
when s4 => z<='1';
end case;
end process outputz;
end architecture Moore3_arch;
```

Definirea şi codificarea stărilor

În mod implicit majoritatea utilitarelor de sinteză folosesc codificarea automată a stărilor. Pot fi utilizate mai multe stiluri de codificare:

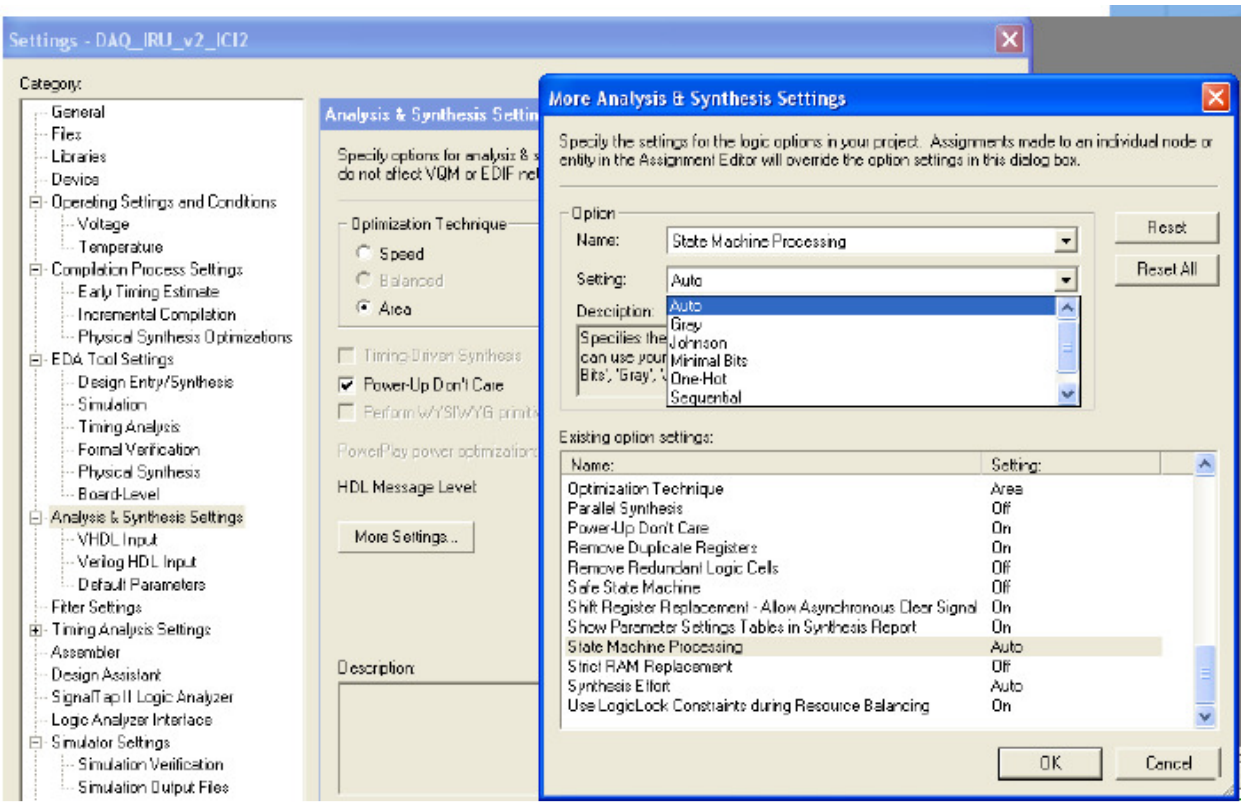
1. Codificare binară consecutivă.
2. Codificarea Gray.
3. Codificarea cu un bistabil pe stare.
4. Codificarea definită de utilizator.

Utilizatorul poate schimba această codificare folosind atribute predefinite sau direct în mediul de programare.

În Quartus II se utilizează atributul **`syn_encoding`**.

```
TYPE count_state is (zero, one, two, three);
ATTRIBUTE syn_encoding              : STRING;
ATTRIBUTE syn_encoding OF count_state : TYPE IS "11 01 10 00";
SIGNAL present_state, next_state      : count_state;
```

Pentru a schimba codificarea în mediul de programare se accesează: **Assignments –Settings Analysis & Synthesis Settings –More Settings**.



1. Codificare binară consecutivă (Sequential).

Avantajul: număr minim de bistabile ($\log_2 n$);
Dezavantajul: Funcţii complexe de tranziţie a stărilor; logică complicată de decodare a stării curente; mai multe bistabile îşi pot schimba starea în acelaşi timp.

Să presupunem că avem un automat cu şase stări: idle, state1, state2, state3, state4, state5. Atunci codificarea stărilor va fi următoare:

idle 000
state1 001
state2 010
state3 011
state4 100
state5 101

2. Codul Gray.

Avantajul: Este cea mai sigură implementare, deoarece tranziţia de la o stare la alta va fi determinată de schimbarea valorii unui singur bistabil, deci este

exclusă apariţia unor stări intermediare. Este important în special în cazul automatelor asincrone.

Dezavantaje: Funcţii complexe de tranziţie a stărilor; logică complicată de decodare a stării curente;

idle 000
state1 001
state2 011
state3 010
state4 110
state5 111

3. Codificarea cu un bistabil pe stare (one hot).

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. Tranziţia dintr-o stare în alta constă în modificarea stării bistabilului corespunzător stării vechi din 1 în 0 şi 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 porţi necesare pentru decodificarea informaţiei de stare pentru ieşiri şi pentru tranziţiile stărilor este cu mult mai redus decât numărul de porţi necesare în cazul utilizării altor tehnici. Această diferenţă de complexitate creşte pe măsură ce numărul de stări devine mai mare.

idle 000001
state1 000010
state2 000100
state3 001000
state4 010000
state5 100000

În funcţie 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 frecvenţă mai ridicată de funcţionare. Codificarea cu un bistabil pe stare nu reprezintă însă soluţia optimă în toate cazurile, în principal datorită faptului că necesită un număr mai mare de bistabile decât codificarea secvenţială. În general, codificarea cu un bistabil pe stare este avantajoasă atunci când arhitectura circuitului programabil utilizat conţine un număr relativ mare de bistabile şi un număr relativ redus de porţi logice între bistabile. De exemplu, această codificare

este cea mai avantajoasă pentru automatele de stare implementate cu circuite FPGA, care conţin un număr mai mare de bistabile decât circuitele CPLD.

Toleranţa la defecte a automatelor de stare

În practică, anumite hazarduri, zgomote sau combinaţii ilegale ale intrărilor pot determina modificarea stării unuia sau a mai multor bistabile, ceea ce poate avea ca efect tranziţia automatului într-o stare ilegală.

Automatul poate rămâne definitiv în această stare ilegală, sau poate activa o combinaţie ilegală a ieşirilor, 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 ieşirea din stările ilegale.

Posibilităţi de proiectare a automatelor de stare sigure.

1. Utilizarea construcţiei **when others.**

when others => stare <= idle;

Prin specificarea tranziţiei din stările ilegale într-o stare cunoscută se va genera o logică suplimentară.

Există cazuri în care costul acestei soluţii nu este justificat de necesitatea unui automat tolerant la defecte.

În aceste cazuri, se poate specifica în mod explicit faptul că tranziţia dintr-o stare ilegală este o condiţie indiferentă (deci, nu are importanţă starea în care se efectuează tranziţia dintr-o stare ilegală, deoarece nu este de aşteptat ca automatul să treacă într-o asemenea stare).

În cazul codificării explicite, condiţiile indiferente pot fi declarate sub forma:

when others => stare <= "---";

unde s-a presupus că semnalul stare este un vector de 3 biţi.

Condiţia **when others** poate fi necesară şi atunci când toate stările automatului sunt descrise, deoarece tipul vectorului de stare este **std_logic_vector,** deci sunt 9 valori posibile, şi nu doar valorile '1' sau '0'.

2. Suplimentarea numărului de stări până la o putere a lui 2.

```
 type stare is (s0, s1, s2, nedefinit);
signal cs, ns: stare;
…
case   cs is
 …
when nedefinit => ns<=s0;
end case;
```

```
type stare is (s0, s1, s2, s3, s4, u1, u2, u3);
  signal cs, ns: stare;
  …
  case    cs is
   …
 when others => ns<=s0;
end case;
```