Laboratorul 2 – Implementarea unui automat cu stari finite

Obiective

Acest laborator este o introducere in proiectarea logica a automatelor cu stari finite (Finite State Machines - FSM). In cadrul laboratorului vor fi prezentate diversele tipuri de FSM si moduri de proiectare in vederea implementarii. Scopul acestuia este de a va familiariza cu unele concepte de proiectare pe care le veti utiliza in viitor.
Deasemenea, in cadrul acestui laborator se vor introduce notiuni privind modularizarea unui proiect si reutilizarea codului.
Pentru a obtine punctajul acordat acestui laborator, trebuie sa prezentati laborantului functionarea corecta in hardware a proiectului dumneavoastra.

Considerente de proiectare

Exista mai multe metode de a proiecta automate cu stari finite. Totusi, urmand anumite metode de proiectare se asigura faptul ca unealta de sinteza identifica si optimizeaza algoritmul ce se doreste a fi implementat, imbunatatind simularea, intarzierile si depanarea circuitului.
Automatele cu stari finite se pot imparti in implementari: Mealy vs. Moore, One-hot vs. Binary sau Safe vs. Fast. Principalele compromisuri intre aceste implementari sunt explicate in sectiunile urmatoare. Recomandarea generala pentru alegerea unui tip de automat cu stari finite depinde de arhitectura dispozitivului tinta si de specificul automatului (dimeniune si comportament). Totusi, in automatele de tip Moore, One-hot se implementeaza mai bine in FPGA-uri, iar cele Mealy, Binary se implementeaza mai bine in CPLD-uri.

Mealy vs. Moore

Sunt doua tipuri bine cunoscute de implementari pentru automate cu stari finite: Mealy si Moore. Diferenta intre ele este ca automatele Mealy determina valorile iesirilor bazanduse si pe starea curenta si pe intrari, in timp ce automatele Moore determina valorile iesirilor numai pe baza starii curente. In general, automatele de tip Moore se implementeaza mai bine in FPGA-uri deoarece cel mai des se alege codarea One-hot si deci este foarte usor (chiar trivial) sa se obtina valorile iesirilor. Daca se alege codarea binara (Binary) este posibil sa se ajunga la un automat mai compact si cateodata mai rapid utilizand automate de tip Mealy. Totusi, acest lucru nu este intotdeauna valabil si nu este usor de determinat fara a cunoaste specificul automatului cu stari finite.

Codarea One-hot vs. Binary

Exista mai multe metode de a codifica starile unui automat. Totusi, doua sunt utilizate mai des in proiectarea cu FPGA-uri sau CPLD-uri. Acestea sunt One-hot si Binary. Pentru cele mai multe arhitecturi FPGA, codarea One-hot este mai buna datorita numarului mare de bistabili (se mapeaza mai bine in interiorul LUT-urilor). Cand dispozitivul tinta este un CPLD, codarea binara (Binary) poate fi de multe ori mai eficienta datorita structurii logice a CPLD-ului si a resurselor mai putin numeroase. In orice caz, majoritatea uneltelor de sinteza contin un algoritm de extragere a FSM-urilor ce poate identifica structura si poate alege cea mai buna metoda de codare pentru marimea, tipul si arhitectura tinta. Desi aceasta facilitate exista (la Xilinx este State Diagram), in cele mai multe cazuri este mai eficient sa aleaga de catre proiectant modul de codare pentru a permite un control mai ridicat si o depanare mai usoara. Este indicat sa consultati documentatia uneltei de sinteza pentru detalii privind capacitatea de extragere a automateleor.

Safe vs. Fast

Cand se proiecteaza un automat cu stari finite, in general sunt doua tinte care se contrazic si trebuiesc intelese: siguranta (Safe) si viteza (Fast). O implementare sigura se refera la cazul in care automatul ar primi un set de valori de intrare necunoscut sau ajunge intr-o stare necunoscuta si poate sa revina intr-o stare cunoscuta la urmatoarea perioada de ceas. Pe de alta parte, daca se renunta la aceasta cerinta, de cele mai multe ori automatul poate fi implementat utilizand mai putine resurse si mai rapid.

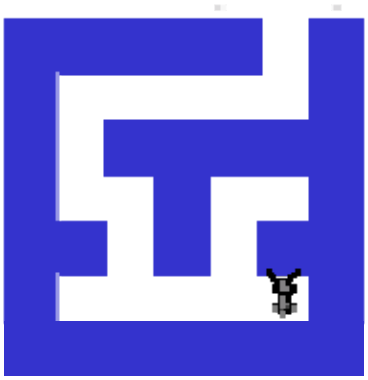
Descrierea problemei

Se cere sa se proiecteze, folosind circuite FPGA, automatul secvential reprezentand unitatea de comanda a unei “insecte electronice” (RoboAnt®), care se deplaseaza intr-un labirint. De asemenea, este realizat un simulator software/hardware, cu interfata grafica, pentru urmarirea comportarii insectei in labirint.
Specificatii de proiectare:

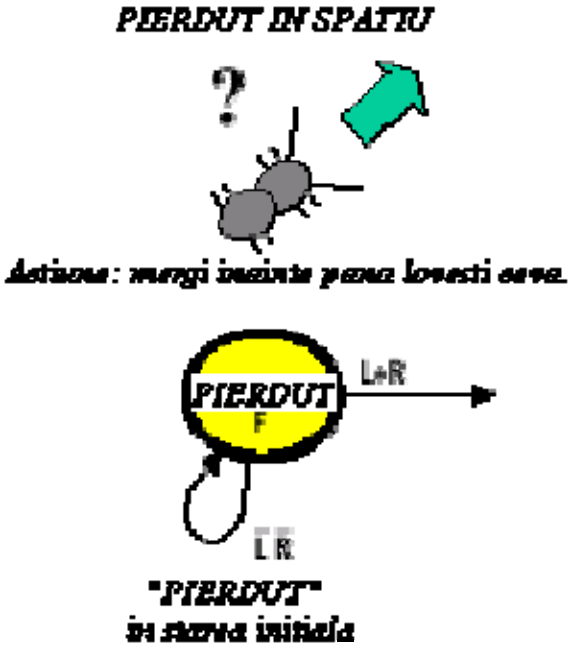
- Senzori: doua antene L (stanga) si R (dreapta), care vor furniza semnalul 1 atunci cand vin in contact cu un obstacol

- Organe de executie: Pas inainte F, Rotire cu un unghi stanga TL si dreapta TR.

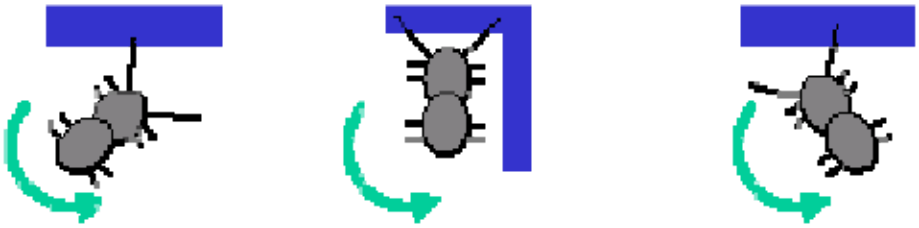
- Scopul urmarit: inzestrarea insectei cu „inteligenta”, astfel incat ea sa poata iesi din labirint.



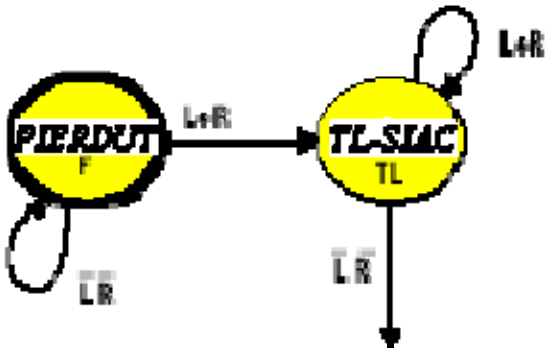
Strategia: “Antena din dreapta pe peretele labirintului”.
Situatii in care se poate afla insecta:



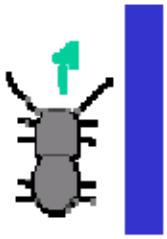
ATINGERE OBSTACOL



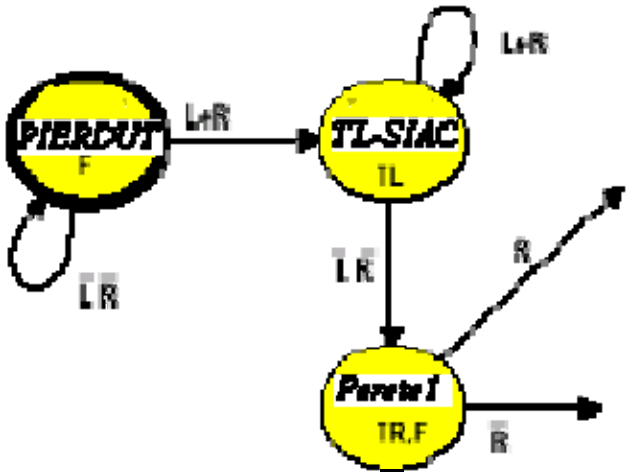
Actiune: TL (Stop Invers Analog de Contact -SIAC), pana cand obstacolul nu mai este atins



PUTIN LA DREAPTA..

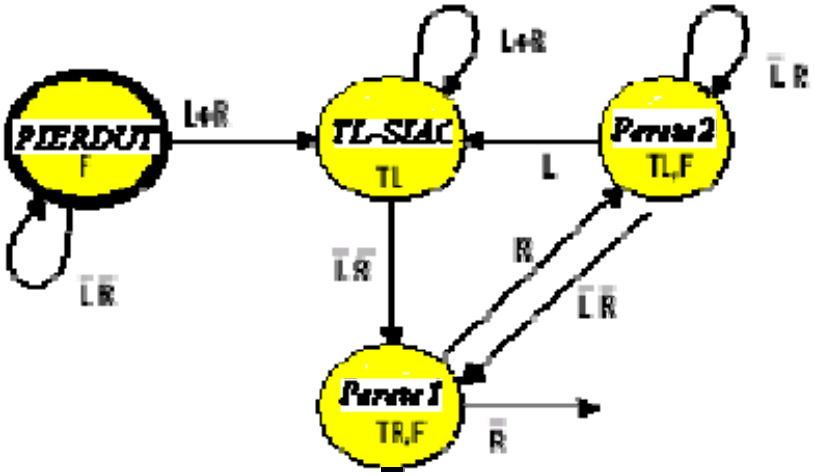


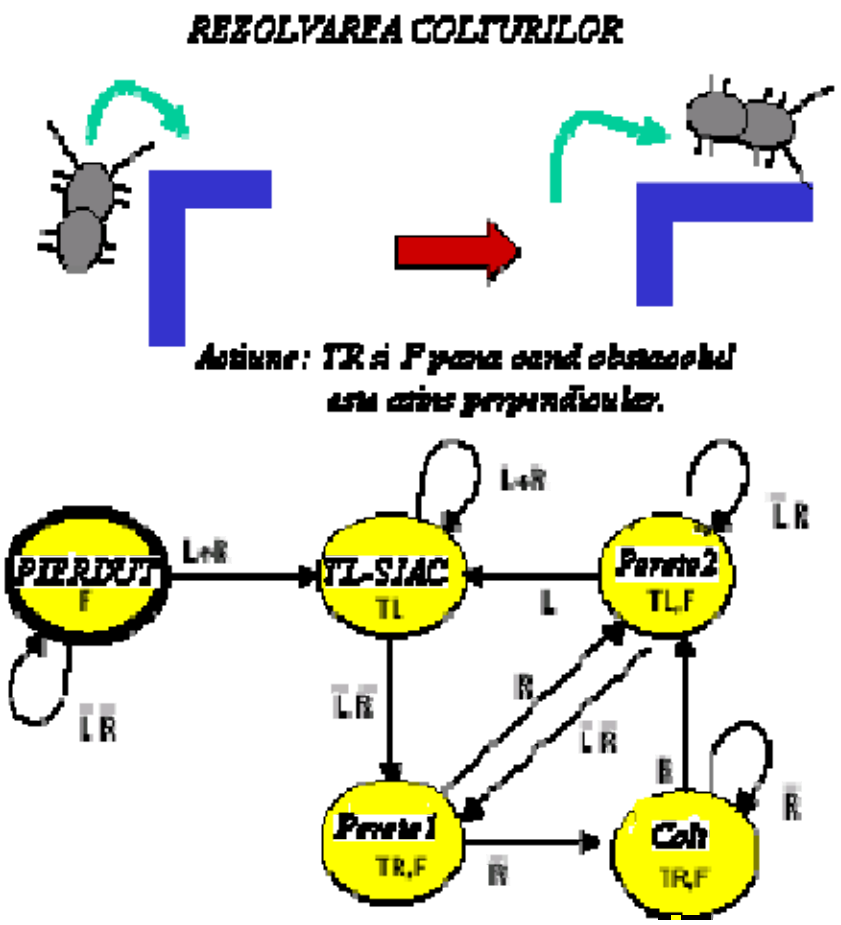
Actiune: TR si F, stabileste daca a fost atins un obstacol



APOI PUTIN LA STANGA..

Actiune: TL si F,stabileste daca a fost atins un obstacol





Reducerea starilor echivalente:

Observatie: Si = Sj daca

1. Starile au iesiri identice;
2. Fiecare intrare -> stari echivalente.

Strategia de Reducere: Gaseste perechi de stari echivalente si efectueaza FUZIUNEA lor.

Observam ca strategia “Antena din dreapta pe peretele labirintului” nu conduce insecta spre iesire in cazul unui labirint cu insule (odata ce insecta ajunge in contact cu o insula, se va invarti la nesfarsit in jurul ei). Pentru a suplini acest dezavantaj se introduce un nou senzor S (Smell) si doua actiuni M si E (Mark si Erase). Scopul acestora este de a putea marca drumul pe care am mai fost si astfel de a evita cazul in care insecta se va invarti in jurul insulei. Cand insecta detecteaza (prin smell) ca a ajuns intr-o zona in care a facut anterior mark, ea nu va mai lua aceeasi „decizie” pentru a nu repeta parcusul anterior.

Pentru a putea testa intr-un mediu „real” unitatea de control (botezata in continuare mind), se pune la dispozitie un proiect ce contine o interfata VGA si un labirint memorat (alaturi de conexiunile necesare) la care se va adauga implementarea modulului cu urmatoarea interfata:

```
module mind(clk, clr, L, R, S, TL, TR, F, M, E);
   input clk;
   input clr;
   input L;
   input R;
   input S;
   output TL;
   output TR;
   output F;
   output M;
   output E;
   // Implementare
endmodule
```

In cazul in care nu se doreste implementarea astfel incat sa se poata trata si labirinturile ce contin insule, comenzile M si E vor fi permanent 0 iar intrarea S va fi ignorata in luarea deciziilor.

In continuare se prezinta diverse structuri pentru implementarea in Verilog a tipurilor de automate cu stari finite mentionate in paragrafele anterioare.

1. Mealy – One-Hot – Fast – 4 states

```
parameter <state1> = 4'b0001;
parameter <state2> = 4'b0010;
parameter <state3> = 4'b0100;
parameter <state4> = 4'b1000;

reg [3:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
```

```
  state <= <state1>;
end
else
  case (state)
    <state1> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
    end
    <state2> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
    end
    <state3> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
    end
    <state4> : begin
      if (<condition>)
```

```
            state <= <next_state>;
          else if (<condition>)
            state <= <next_state>;
          else
            state <= <next_state>;
        end
      endcase

assign <output1> = <logic_equation_based_on_states_and_inputs>;
assign <output2> = <logic_equation_based_on_states_and_inputs>;
```

2. Mealy – One-Hot – Safe – 4 states

```
parameter <state1> = 4'b0001;
parameter <state2> = 4'b0010;
parameter <state3> = 4'b0100;
parameter <state4> = 4'b1000;

reg [3:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
```

```
  else
    state <= <next_state>;
end
<state2> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
end
<state3> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
end
<state4> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
end
default : begin  // Fault Recovery
  state <= <state1>;
end
```

```
    endcase

assign <output1> = <logic_equation_based_on_states_and_inputs>;
assign <output2> = <logic_equation_based_on_states_and_inputs>;
```

3. Mealy – Binary – Fast – 4 states

```
 parameter <state1> = 2'b00;
parameter <state2> = 2'b01;
parameter <state3> = 2'b10;
parameter <state4> = 2'b11;

reg [1:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
      <state2> : begin
        if (<condition>)
          state <= <next_state>;
```

```
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
      <state3> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
      <state4> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
    endcase

  assign <output1> = <logic_equation_based_on_states_and_inputs>;
  assign <output2> = <logic_equation_based_on_states_and_inputs>;
```

4. Mealy – Binary – Safe – 4 states

```
  parameter <state1> = 2'b00;
  parameter <state2> = 2'b01;
  parameter <state3> = 2'b10;
```

```
parameter <state4> = 2'b11;

reg [1:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
      <state2> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
      end
      <state3> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
```

```
            state <= <next_state>;
          else
            state <= <next_state>;
        end
        <state4> : begin
          if (<condition>)
            state <= <next_state>;
          else if (<condition>)
            state <= <next_state>;
          else
            state <= <next_state>;
        end
        default : begin  // Fault Recovery
          state <= <state1>;
        end
      endcase

    assign <output1> = <logic_equation_based_on_states_and_inputs>;
    assign <output2> = <logic_equation_based_on_states_and_inputs>;
```

5. Moore – One-Hot – Fast – 4 states

```
    parameter <state1> = 4'b0001;
    parameter <state2> = 4'b0010;
    parameter <state3> = 4'b0100;
    parameter <state4> = 4'b1000;

    reg [3:0] state = <state1>;

    always@(posedge <clock>)
      if (<reset>) begin
```

```
  state <= <state1>;
  <outputs> <= <initial_values>;
end
else
  case (state)
    <state1> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
      <outputs> <= <values>;
    end
    <state2> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
      <outputs> <= <values>;
    end
    <state3> : begin
      if (<condition>)
        state <= <next_state>;
      else if (<condition>)
        state <= <next_state>;
      else
        state <= <next_state>;
```

```
        <outputs> <= <values>;
      end
      <state4> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
        <outputs> <= <values>;
      end
    endcase
```

6. Moore – One-Hot – Safe – 4 states

```
parameter <state1> = 4'b0001;
parameter <state2> = 4'b0010;
parameter <state3> = 4'b0100;
parameter <state4> = 4'b1000;

reg [3:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
    <outputs> <= <initial_values>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
```

```
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  <state2> : begin
    if (<condition>)
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  <state3> : begin
    if (<condition>)
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  <state4> : begin
    if (<condition>)
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
```

```
        else
          state <= <next_state>;
        <outputs> <= <values>;
      end
      default: begin  // Fault Recovery
        state <= <state1>;
        <outputs> <= <values>;
          end
    endcase
```

7. Moore – Binary – Fast – 4 states

```
 parameter <state1> = 2'b00;
parameter <state2> = 2'b01;
parameter <state3> = 2'b10;
parameter <state4> = 2'b11;

reg [1:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
    <outputs> <= <initial_values>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
```

```
  else
    state <= <next_state>;
  <outputs> <= <values>;
end
<state2> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
  <outputs> <= <values>;
end
<state3> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
  <outputs> <= <values>;
end
<state4> : begin
  if (<condition>)
    state <= <next_state>;
  else if (<condition>)
    state <= <next_state>;
  else
    state <= <next_state>;
  <outputs> <= <values>;
```

```
    end
  endcase
```

8. Moore – Binary – Safe – 4 states

```
parameter <state1> = 2'b00;
parameter <state2> = 2'b01;
parameter <state3> = 2'b10;
parameter <state4> = 2'b11;

reg [1:0] state = <state1>;

always@(posedge <clock>)
  if (<reset>) begin
    state <= <state1>;
    <outputs> <= <initial_values>;
  end
  else
    case (state)
      <state1> : begin
        if (<condition>)
          state <= <next_state>;
        else if (<condition>)
          state <= <next_state>;
        else
          state <= <next_state>;
        <outputs> <= <values>;
      end
      <state2> : begin
        if (<condition>)
          state <= <next_state>;
```

```
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  <state3> : begin
    if (<condition>)
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  <state4> : begin
    if (<condition>)
      state <= <next_state>;
    else if (<condition>)
      state <= <next_state>;
    else
      state <= <next_state>;
    <outputs> <= <values>;
  end
  default : begin  // Fault Recovery
    state <= <state1>;
    <ouputs> <= <values>;
  end
endcase
```