Circuite secvenţiale - utilizare sincronă şi asincronă

Sincron - asincron

- Sincronizare
  - circuitele lucrează după un semnal de ceas comun
  - evoluează în aceleaşi momente
- Asincronism
  - fiecare circuit evoluează pe baza unor comenzi provenite de la alte circuite
- pot apărea ambele moduri de comandă în acelaşi sistem

Utilizare asincronă (1)

- exemplu - comanda unui ascensor
- intrări
  - etajul destinaţie
  - etajul curent
- ieşiri
  - urcare
  - coborîre

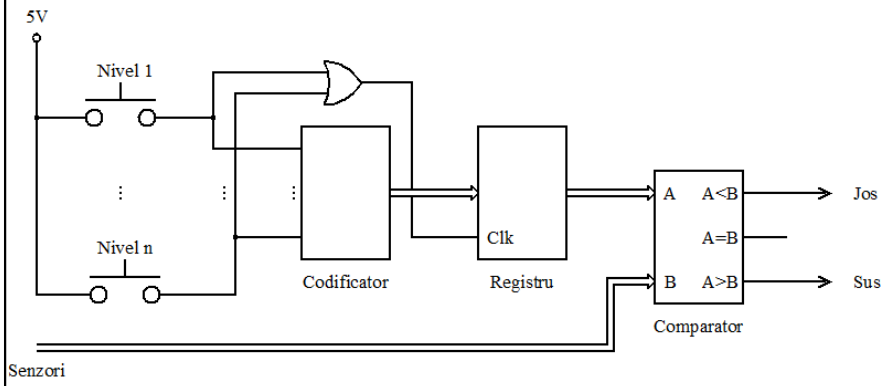
Utilizare asincronă (2)

- comanda - dată de relaţia dintre intrări
  - etaj destinaţie < etaj curent → coborîre
  - etaj destinaţie > etaj curent → urcare
  - etaj destinaţie = etaj curent → nici o acţiune
- modificarea intrării impune modificarea automată a comenzii
- nu este necesară reţinerea unei stări curente

Utilizare asincronă (3)

- sunt totuşi necesare circuite secvenţiale?
- controlul informaţiilor de la intrări
- etajul curent - provenit de la senzori
- etajul destinaţie - indicat de utilizator
  - apăsarea unui buton
  - informaţia se poate pierde → trebuie păstrată
  - putem folosi un registru

Utilizare asincronă (4)



Utilizare asincronă (5)

- problemă cu schema anterioară
  - este apăsat un nou buton în timp ce liftul se deplasează către etajul destinaţie deja stabilit
  - deci etajul destinaţie este schimbat - *în timpul deplasării*
- proiectarea asincronă este foarte dificilă
  - mai utilă pentru componente, nu pentru sisteme întregi

Utilizare sincronă

- este apăsat un buton corespunzător unui etaj
  - liftul este nefolosit - ne deplasăm spre acel etaj
  - liftul are deja o comandă - ignoră noua apăsare a butonului
  - răspunsuri diferite la aceeaşi valoare a intrării
- soluţia - sistem cu stare
- comanda dată - depinde de starea curentă
  - şi de ultimele informaţii de la intrări (senzori)

Proiectarea sistemelor secvenţiale simple

Automate

- stări
- tranziţii între stări
- 3 categorii de variabile
  - de intrare
  - de stare
  - de ieşire

Ecuaţii de funcţionare

- pornind de la valorile pentru
  - intrări
  - starea curentă
- dorim să obţinem valorile pentru
  - starea următoare
  - ieşiri

Proiectare - paşi (1)

- analiza problemei - foarte important!
- stabilirea variabilelor de intrare şi de ieşire
- graful de fluenţă
  - stările
  - tranziţiile
  - valorile variabilelor de intrare care condiţionează tranziţiile
  - valorile ieşirilor corespunzătoare tranziţiilor

Proiectare - paşi (2)

- matricea (tabelul) de fluenţă
  - exprimare formală a grafului de fluenţă
  - liniile - stările curente
  - coloanele - combinaţiile de valori ale variabilelor de intrare
  - conţinutul unei celule
    - starea următoare
    - valorile variabilelor de ieşire

Proiectare - paşi (3)

- reducerea stărilor echivalente
  - pe baza analizei matricii de fluenţă
  - 2 stări sunt neechivalente dacă
    - produc valori diferite ale ieşirilor pentru cel puţin o combinaţie de valori ale intrărilor
    - echivalenţa lor depinde de echivalenţa altor 2 stări, care sunt neechivalente (recursiv)
  - altfel sunt echivalente
  - stările echivalente pot fi reunite

Proiectare - paşi (4)

- codificarea stărilor
  - rezultă şi numărul de biţi necesar pentru a reţine starea (numărul variabilelor de stare)
- rescrierea matricii de fluenţă cu stările în forma codificată
- aplicarea minimizării
- obţinerea ecuaţiilor de funcţionare
  - câte o ecuaţie booleană pentru fiecare variabilă de stare, respectiv ieşire

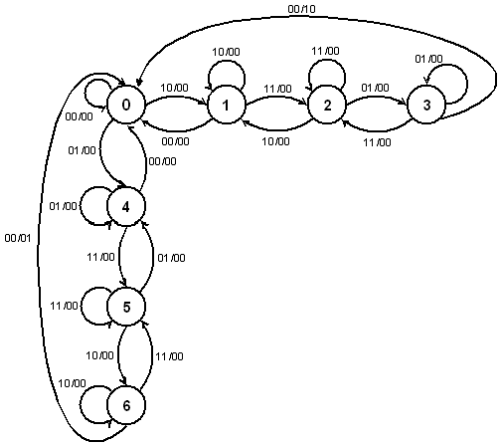
Exemplu

- problema - numărarea persoanelor care intră şi ies printr-un culoar
- semnale de intrare - 2 celule fotoelectrice
- semnale de ieşire - incrementarea a 2 numărătoare (intraţi, respectiv ieşiţi)
- premise
  - pe culoar nu pot trece două persoane una pe lângă alta
  - mişcarea persoanelor - relativ lentă (în comparaţie cu automatul)

Detectarea intrărilor/ieşirilor

- secvenţa semnalelor de intrare
  - 00→10→11→01→00 - a intrat o persoană
  - 00→01→11→10→00 - a ieşit o persoană
- secvenţă incompletă - nu a intrat/ieşit nimeni
- la completarea unei secvenţe, se activează ieşirea corespunzătoare
  - primeşte valoarea 1 pentru o perioadă de ceas

Graful de fluenţă



Matricea de fluenţă

	00	01	10	11
0	0/00	4/00	1/00	*/**
1	0/00	*/**	1/00	2/00
2	*/**	3/00	1/00	2/00
3	0/10	3/00	*/**	2/00
4	0/00	4/00	*/**	5/00
5	*/**	4/00	6/00	5/00
6	0/01	*/**	6/00	5/00

Stări echivalente - etapa 1

(0,1): e
(0,2): (3,4)
(0,3): n
(0,4): e
(0,5): (1,6)
(0,6): n
(1,2): e
(1,3): n
(1,4): (2,5)
(1,5): (1,6) (2,5)
(1,6): n
(2,3): e
(2,4): (3,4) (2,5)
(2,5): (3,4) (1,6)
(2,6): (1,6) (2,5)
(3,4): n
(3,5): (3,4) (2,5)
(3,6): n
(4,5): e
(4,6): n
(5,6): e

Stări echivalente - etapa 2

(0,1): e	(1,3): n	(2,6): n
(0,2): n	(1,4): n	(3,4): n
(0,3): n	(1,5): n	(3,5): n
(0,4): e	(1,6): n	(3,6): n
(0,5): n	(2,3): e	(4,5): e
(0,6): n	(2,4): n	(4,6): n
(1,2): e	(2,5): n	(5,6): e

Reducerea stărilor echivalente

- grupări posibile: (0,1), (0,4), (1,2), (2,3), (4,5), (5,6)
- nu se poate forma nici un grup de 3 sau mai multe stări
- noile stări (variantă)
  - (0,1) → A
  - (2,3) → B
  - (4,5) → C
  - 6 → D

Codificarea stărilor

- 4 stări → 2 variabile de stare (biţi)
- variantă
  - A ↔ 00
  - B ↔ 01
  - C ↔ 10
  - D ↔ 11
- ecuaţiile finale depind de codificare
  - dar nu putem şti care variantă va fi mai bună

Diagrama de minimizare

y_1y_0 \ x_1x_0	00	01	11	10
00	00/00	10/00	01/00	00/00
01	00/10	01/00	01/00	00/00
11	00/01	**/**	10/00	11/00
10	00/00	10/00	10/00	11/00

Minimizare - $y_{1,n+1}$

y_1y_0 \ x_1x_0	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	0	*	1	1
10	0	1	1	1

Minimizare - $y_{0,n+1}$

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	0	*	0	1
10	0	0	0	1

Minimizare - z_1

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	*	0	0
10	0	0	0	0

Minimizare - z_0

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	*	0	0
10	0	0	0	0

Ecuaţiile de funcţionare

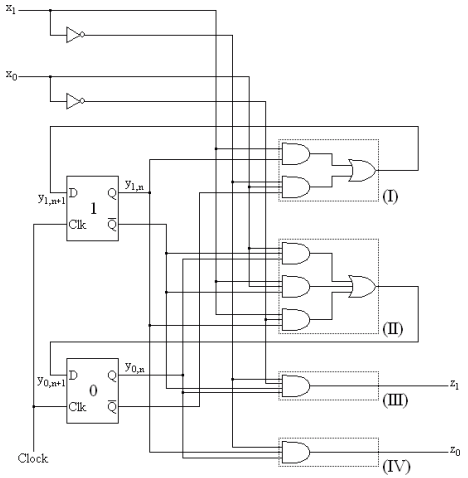
(I) $y_{1,n+1} = \left(y_1 \cdot x_1 + \overline{y_0} \cdot \overline{x_1} \cdot x_0 \right)_n$

(II) $y_{0,n+1} = \left(\overline{y_1} \cdot y_0 \cdot x_0 + \overline{y_1} \cdot x_1 \cdot x_0 + y_1 \cdot x_1 \cdot \overline{x_0} \right)_n$

(III) $z_{1,n} = \left(\overline{y_1} \cdot y_0 \cdot \overline{x_1} \cdot \overline{x_0} \right)_n$

(IV) $z_{0,n} = \left(y_1 \cdot y_0 \cdot \overline{x_1} \right)_n$

Implementarea circuitului



Automate hardware

Complexitate (1)

- sistemele discutate la cursul anterior permit rezolvarea de probleme simple
  - nu utilizează nici măcar circuite combinaționale și secvențiale predefinite
  - sumatoare, comparatoare, decodoare etc.
  - regiștri, numărătoare etc.
- în general trebuie abordate probleme mai complexe

Complexitate (2)

- logica este separată în două părți
  - elementele de acționare
  - secvențiatorul
- acesta din urmă comandă elementele de acționare
- modelare
  - tot sub forma unui automat
  - dar determinarea stărilor se face în alt mod

Componente (1)

- elemente de acţionare
  - circuite combinaţionale şi secvenţiale
  - realizează operaţiile cerute de algoritmul implementat
  - operaţiile trebuie realizate la anumite momente
    - implementarea algoritmului
  - materializare - circuitele secvenţiale primesc comenzi la anumite momente

Componente (2)

- secvenţiatorul
  - automatul de control
  - trimite comenzi către elementele de acţionare la momentele potrivite
  - implementare cablată - automat simplu
  - stările - decodificate
    - câte un bistabil pentru fiecare stare
    - la fiecare moment - un singur bistabil are valoarea 1

Intrările secvenţiatorului

- semnale care pot fi testate
- în funcţie de valoarea lor se pot lua decizii
- provenienţă
  - semnale din exteriorul sistemului
  - semnale generate de elementele de acţionare (comparatoare, numărătoare - terminal count, porţi logice etc.)

Ieşirile secvenţiatorului

- semnale de comandă către elementele de acţionare
  - comenzi pentru circuitele secvenţiale
  - validări pentru circuitele combinaţionale
- exemple
  - semnale de resetare
  - încărcare regiştri
  - incrementare/decrementare numărătoare
  - validare decodoare etc.

Descrierea algoritmului

- schema logică
- 2 tipuri de blocuri
  - testări
    - semnalele de intrare
  - acţiuni
    - comenzi către elementele de acţionare

Implementarea secvenţiatorului

- delimitarea stărilor
  - o stare poate include simultan testări şi acţiuni
- determinarea ecuaţiilor
  - variabile de stare
    - precizează starea următoare în funcţie de starea curentă şi de intrări
    - uzual, stările nu sunt codificate
    - câte o variabilă de stare (bistabil) pentru fiecare stare
  - ieşiri

Exemplu

- măsurarea perioadei unui semnal
- testarea repetată a valorii semnalului de intrare (INPUT)
  - automatul este mult mai rapid decât variaţiile semnalului măsurat
  - determinarea duratei unei perioade
    - incrementarea unui numărător
    - cât timp intrarea este pe 0, apoi cât este pe 1
    - sau invers

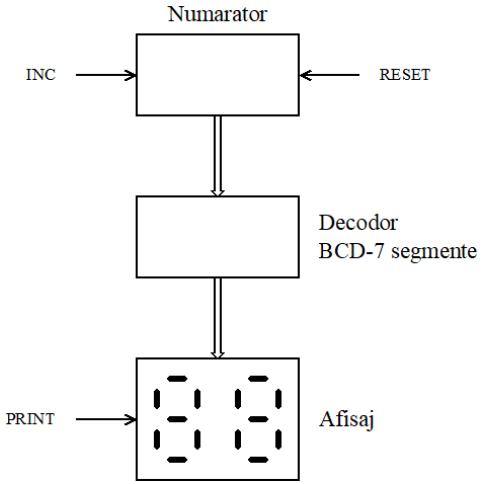
Exemplu (continuare)

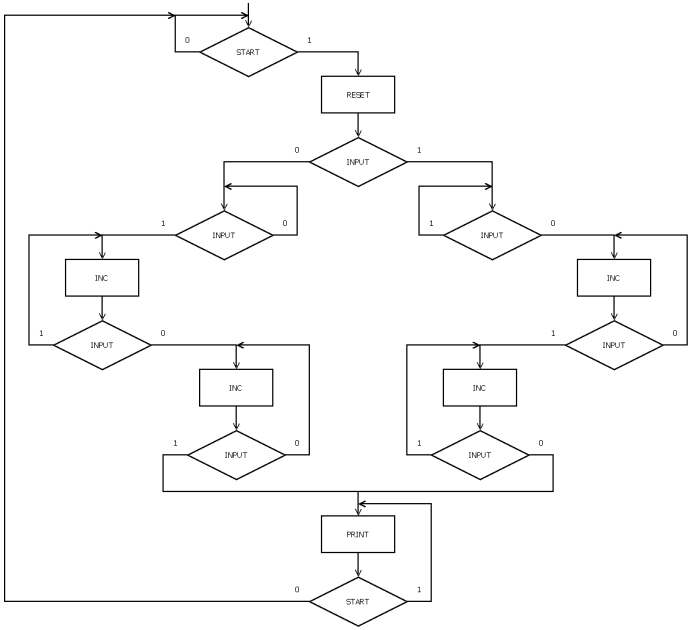
- iniţial
  - se aşteaptă apăsarea unui buton (START)
- final
  - afişare (PRINT)
  - se aşteaptă terminarea apăsării butonului

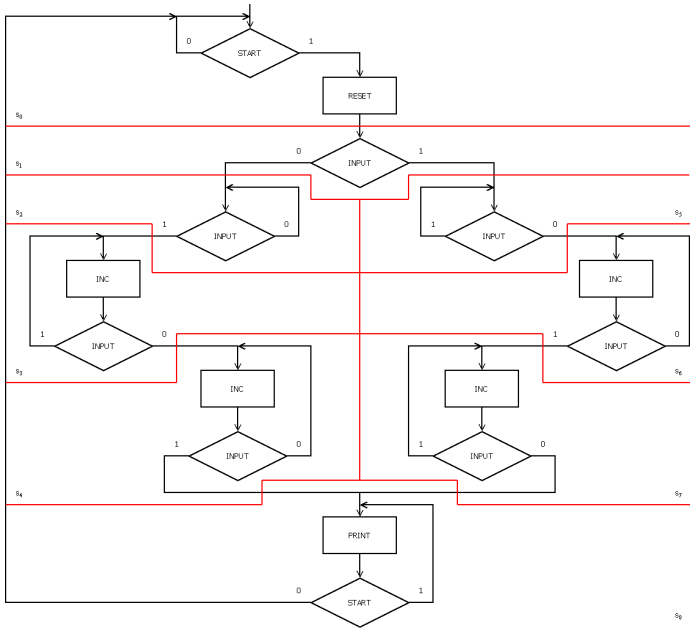
Elementele de acţionare

- determinarea valorii - numărător
- pentru afișare de folosește un display cu 7 segmente
- conectare
  - valoarea din numărător este trimisă către display
  - pentru adaptarea informației - decodor BCD-7 segmente

Elementele de acţionare - schema







Ecuaţii de funcţionare

$$s_{0,n+1} = s_{0,n} \cdot \overline{START} + s_{8,n} \cdot \overline{START}$$

$$s_{1,n+1} = s_{0,n} \cdot START$$

$$s_{2,n+1} = s_{1,n} \cdot \overline{INPUT} + s_{2,n} \cdot \overline{INPUT}$$

$$s_{3,n+1} = s_{2,n} \cdot INPUT + s_{3,n} \cdot INPUT$$

$$s_{4,n+1} = s_{3,n} \cdot \overline{INPUT} + s_{4,n} \cdot \overline{INPUT}$$

$$s_{5,n+1} = s_{1,n} \cdot INPUT + s_{5,n} \cdot INPUT$$

$$s_{6,n+1} = s_{5,n} \cdot \overline{INPUT} + s_{6,n} \cdot \overline{INPUT}$$

$$s_{7,n+1} = s_{6,n} \cdot INPUT + s_{7,n} \cdot INPUT$$

$$s_{8,n+1} = s_{4,n} \cdot INPUT + s_{7,n} \cdot \overline{INPUT} + s_{8,n} \cdot START$$

$$RESET = s_{0,n} \cdot START$$

$$PRINT = s_{8,n}$$

$$INC = s_{3,n} + s_{4,n} + s_{6,n} + s_{7,n}$$

Microprogramare

Implementarea funcţiilor booleene

- combinaţional
  - minimizare
- cu ajutorul unei memorii ROM
  - intrările - biţii de adresă
  - ieşirile - biţii de date
  - conţinutul memoriei ROM urmează direct tabelul de adevăr

Exemplu

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Adresa	Valoare
0	1
1	0
2	0
3	1
4	0
5	1
6	0
7	1

Microprogramare

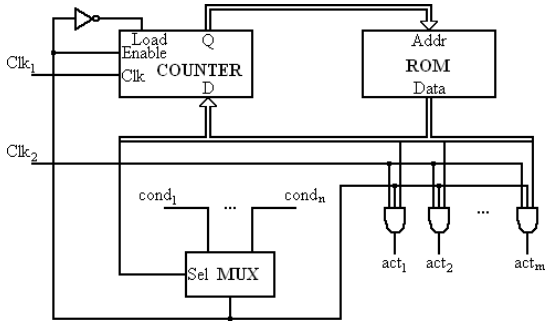
- implementarea secvenţiatorului
  - memorie ROM
    - fiecare locaţie corespunde unei stări
    - conţine informaţiile necesare execuţiei (microinstrucţiune)
  - numărător program
    - adresa locaţiei corespunzătoare următoarei stări
    - actualizat prin incrementare sau salt

Microprogramare orizontală (1)

- câmpurile unei microinstrucţiuni
  - codul condiţiei testate
    - poate fi şi 1 (+5V - întotdeauna adevărat) sau 0 (0V - întotdeauna fals)
  - semnalele de ieşire (acţiunile)
    - 1 - activat
    - în cazul în care condiţia testată este adevărată
  - adresa de salt
    - în cazul în care condiţia testată este falsă
    - altfel - incrementarea numărătorului program

Microprogramare orizontală (2)

- structura secvenţiatorului

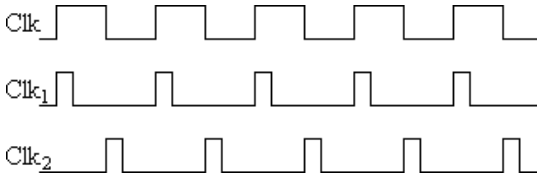


Microprogramare orizontală (3)

- dacă trebuie testate atât condiţia x, cât şi NOT(x)
  - este necesar să avem câte o intrare în multiplexor pentru x şi NOT(x)
- dacă 2 sau mai multe acţiuni nu apar niciodată simultan
  - în memoria ROM se poate stoca doar un cod al acţiunii activate, care este introdus într-un decodor → mai puţini biţi necesari în ROM

Microprogramare orizontală (4)

- semnalele de ceas
  - Clk_1 - delimitarea stărilor
  - Clk_2 - validarea semnalelor de comandă (acţiuni)
  - obţinute dintr-un semnal periodic



Probleme

- delimitarea stărilor - mai puţin flexibilă decât la secvenţiatorul cablat
- testare şi acţiune în aceeaşi stare
  - dacă acţiunea trebuie efectuată numai când condiţia testată are valoarea 1
  - altfel - ce valoare se trece în microinstrucţiune pentru acţiunea respectivă?
- starea următoare - uneori este mai greu de specificat

Probleme - exemplificare

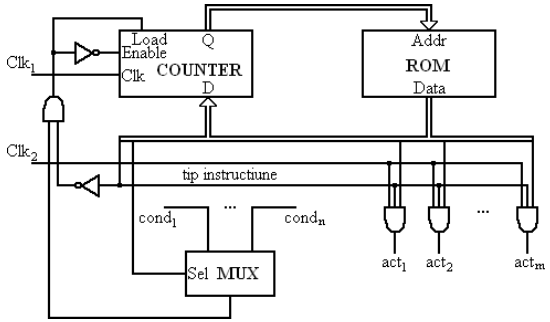
- aceeaşi problemă ca la cursul anterior
  - starea s_8 nu poate urma şi după s_4, şi după s_7
  - similar, s_0 nu urmează după s_8
  - soluţii
    - introducerea unor stări noi - nu realizează nici o acţiune, doar salt
    - duplicarea stării s_8 (una urmează după s_4, cealaltă după s_7)
    - codificarea stărilor astfel ca s_0 să urmeze după s_8

Microprogramare verticală (1)

- locaţiile din ROM - dimensiuni mai mici
- 2 tipuri de microinstrucţiuni
  - de salt
    - bit de identificare al tipului (valoarea 0)
    - codul condiţiei de testat
    - adresa de salt
  - de acţiune
    - bit de identificare al tipului (valoarea 1)
    - acţiuni

Microprogramare verticală (2)

- structura secvenţiatorului



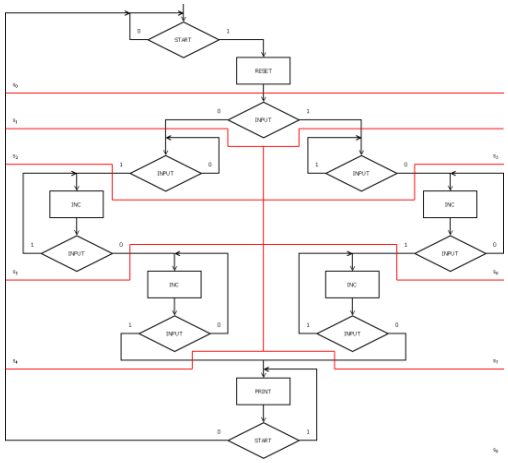
Microprogramare verticală (3)

- delimitarea stărilor - inflexibilă
- nu se pot grupa teste şi acţiuni în aceeaşi stare
- mai multe stări → mai multe cicluri de ceas → viteză mai mică

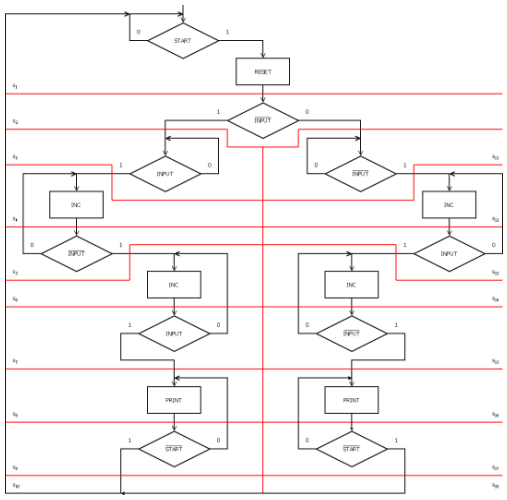
Exemplu

- măsurarea perioadei unui semnal (la fel ca la cursul 2)
- nu pot fi acţiuni şi salturi simultan → multe stări
- atenţie la perioada determinată
  - înainte: numărul înregistrat × perioada ceasului automatului
  - acum: incrementările nu se mai fac la fiecare perioadă de ceas

Secvenţiator cablat - stări



Microinstrucţiuni orizontale - stări



Microinstrucţiuni (1)

s_1: START; RESET; s_1
s_2: NOT(INPUT); ∅; s_{11}
s_3: INPUT; ∅; s_3
s_4: TRUE; INC; ∅
s_5: NOT(INPUT); ∅; s_4
s_6: TRUE; INC; ∅
s_7: INPUT; ∅; s_6
s_8: TRUE; PRINT; ∅

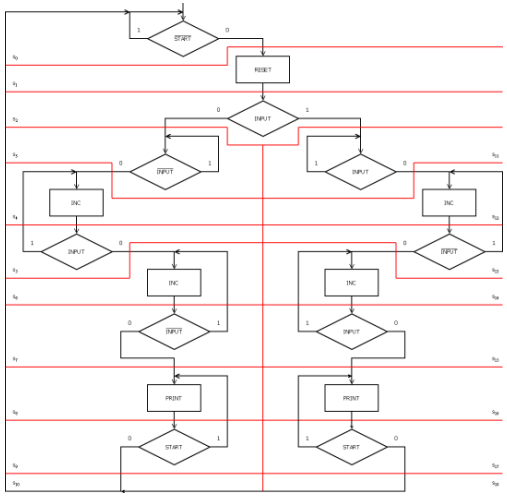
Microinstrucţiuni (2)

s_9: NOT(START); ∅; s_8
s_{10}: FALSE; ∅; s_1
s_{11}: NOT(INPUT); ∅; s_{11}
s_{12}: TRUE; INC; ∅
s_{13}: INPUT; ∅; s_{12}
s_{14}: TRUE; INC; ∅
s_{15}: NOT(INPUT); ∅; s_{14}
s_{16}: TRUE; PRINT; ∅

Microinstrucţiuni (3)

s_{17}: NOT(START); ∅; s_{16}
s_{18}: FALSE; ∅; s_1

Microinstrucţiuni verticale - stări



Microinstrucţiuni (1)

s_0: 0; NOT(START); s_0
s_1: 1; RESET
s_2: 0; INPUT; s_{11}
s_3: 0; NOT(INPUT); s_3
s_4: 1; INC
s_5: 0; INPUT; s_4
s_6: 1; INC
s_7: 0; NOT(INPUT); s_6

Microinstrucţiuni (2)

s_8: 1; PRINT
s_9: 0; START; s_8
s_{10}: 0; TRUE; s_0
s_{11}: 0; INPUT; s_{11}
s_{12}: 1; INC
s_{13}: 0; NOT(INPUT); s_{12}
s_{14}: 1; INC
s_{15}: 0; INPUT; s_{14}

Microinstrucţiuni (3)

s_{16}: 1; PRINT
s_{17}: 0; START; s_{16}
s_{18}: 0; TRUE; s_0

Comparaţie

- microinstrucţiuni orizontale
  - structură mai clară
  - se pot grupa mai bine stările - mai puţine stări
    - test urmat de acţiune - în aceeaşi stare
    - acţiune urmată de test - nu se poate în aceeaşi stare
- microinstrucţiuni verticale
  - cuvântul de memorie - mai scurt
  - mai multe stări - mai multe locaţii de memorie

Implementarea sistemelor secvenţiale

Variante tehnologice (1)

- ASIC
  - *Application-Specific Integrated Circuits*
  - proiectarea - realizată în mod specific pentru un anumit circuit sau clasă de circuite
  - avantaj - optimizare pentru rezolvarea problemei abordate → performanţe superioare
  - tipuri
    - full-custom
    - semi-custom

Variante tehnologice (2)

- circuite programabile
  - permit implementarea a diferite automate
    - în funcţie de problema abordată
    - pot fi reconfigurate după necesităţi
  - avantaje
    - flexibilitate în proiectare şi depanare
    - preţ redus - datorită producţiei de serie
  - dezavantaj
    - nu pot asigura obţinerea de performanţe maxime

Circuite programabile (1)

- arii de porţi (*Gate Arrays*)
  - număr mare de celule
  - fiecare celulă - sumă (OR) de termeni produs (AND)
    - de obicei se folosesc doar o parte din intrări (câte sunt necesare)
  - corespunde cu modul de proiectare al circuitelor combinaţionale
  - celulele sunt interconectate - circuite complexe

Circuite programabile (2)

- CPLD
  - *Complex Programmable Logic Devices*
  - fiecare celulă include şi un bistabil
    - la ieşirea părţii combinaţionale
    - poate fi folosit sau nu (comportament pur combinaţional)
    - poate exista şi reacţie înapoi - de la ieşirea bistabilului spre partea combinaţională
  - modelează comportamentul secvenţial

Circuite programabile (3)

- FPGA
  - *Field Programmable Gate Arrays*
  - celulele - de obicei mai simple decât la CPLD
  - partea combinaţională - LUT (*Look-Up Table*)
  - implementare LUT - variante
    - multiplexor
    - memorie ROM
    - memorie RAM - mai uşor de reconfigurat

Implementare (1)

Probleme

1. proiectarea logică a circuitului
2. implementarea fizică
   - configurarea şi interconectarea celulelor
   - utilizarea circuitelor disponibile în cadrul celulelor
     - structura fizică poate diferi de cea logică
- dificil de gestionat

Implementare (2)

Soluţia

- compilatoare de hardware
  - realizează implementarea fizică
  - ţin cont de structura hardware concretă pe care se face implementarea
- se folosesc limbaje dedicate de descriere a hardware-ului
  - VHDL, Verilog etc.

Limbajul Verilog

Organizare

- circuitele - implementate ca module
- moduri de descriere a circuitelor
  - structurală
    - nivel jos
    - indicarea componentelor folosite
  - comportamentală
    - nivel înalt
    - se descrie comportamentul dorit al circuitului
  - pot fi combinate

Module

- nume modul
- lista parametrilor (semnale de intrare-ieşire)
- declararea tipului parametrilor
  - input
  - output
  - inout (mai rar)
- descrierea modulului
  - diverse moduri

Exemplu

- proiectarea unui multiplexor $2 \rightarrow 1$
  - intrări
    - de date: I0, I1
    - de selecţie: Sel
  - ieşire: E
- pot fi folosite ambele moduri de descriere, inclusiv combinaţii

Varianta 1

```
module MUX(Sel,I0,I1,E);
input Sel,I0,I1;
output E;
assign E=(I1&Sel)|(I0&~Sel);
endmodule
```

- `assign` - asignare continuă
  - orice schimbare la intrare duce la schimbarea ieşirii (combinaţional)

Varianta 2

```
module MUX(Sel,I0,I1,E);
input Sel,I0,I1;
output E;
wire x,y,z;
not n0(z,Sel);
and a1(x,I1,Sel);
and a0(y,I0,z);
or o0(E,x,y);
endmodule
```

Varianta 2 (continuare)

- `wire` - declară semnale interne
- `not`, `and`, `or` - funcţii (module) predefinite
  - primul parametru - ieşirea
  - număr variabil de intrări
- se pot utiliza şi alte module definite de utilizator

Varianta 3

```
module MUX(Sel,I0,I1,E);
input Sel,I0,I1;
output E;
reg E;
always @(Sel,I0,I1)
  E=(I1&Sel)|(I0&~Sel);
endmodule
```

Varianta 3 (continuare)

- `reg` - variabila poate reţine o valoare între două asignări
  - de obicei secvenţial (dar nu neapărat)
- `always` - buclă infinită
  - se execută de fiecare dată când se modifică una din intrările din lista asociată
  - asignarea pentru variabila E nu este continuă
    - valoarea variabilei E se schimbă doar când se execută instrucţiunea respectivă

Varianta 4

```
module MUX(Sel,I0,I1,E);
input Sel,I0,I1;
output E; reg E;
always @(Sel,I0,I1)
begin
  if(Sel==0) E=I0;
  else E=I1;
end
endmodule
```

Elemente ale limbajului Verilog

Vectori (1)

- folosiţi pentru semnalele care constau din mai mulţi biţi
  - numere, adrese etc.
  - domeniul din care fac parte indicii - flexibil

```
input [7:0]x;
wire [0:7]y;
reg [10:3]z;
```

  - toate definesc semnale pe 8 biţi

Vectori (2)

- pot fi accesaţi
  - integral (toţi biţii simultan)
  - bit cu bit
  - o parte din biţi

```
z=x;
z[6]=y[4];
z[5:3]=x[6:4];
```

Tablouri (1)

- similare celor din limbajele software
- la un moment dat poate fi accesat un singur element al tabloului

```
wire x[3:0]; //tablou cu 4
  elemente pe 1 bit fiecare
a=x[0]; //corect
b=x; //eroare
c=x[2:0]; //eroare
```

Tablouri (2)

- elementele unui tablou pot fi vectori
- pot fi accesate direct şi la nivel de bit

```
reg [7:0]y[15:0]; // tablou de
  16 elemente pe 8 biti fiecare
reg [7:0]z;
z=y[5];
```

Atribuiri (1)

- utilizate în descrierile comportamentale
  - nu şi pentru `assign`
- două tipuri
  1. blocante (=)
  - atribuirea curentă va începe doar după terminarea celei anterioare
  2. neblocante (<=)
  - atribuirea curentă poate începe înainte de terminarea celei anterioare

Atribuiri (2)

- `a` are iniţial valoarea 7

```
a=5;
b=a+3;
```

  - `b` va primi valoarea 8

```
a<=5;
b<=a+3;
```

  - `b` va primi valoarea 10

Întârzieri (1)

- instrucţiunile din exemplul anterior sunt raportate ca executându-se la acelaşi moment
  - indiferent de tipul de atribuire utilizat
- trebuie să specificăm faptul că o instrucţiune se execută mai târziu decât cea dinaintea sa
  - şi după cât timp

Întârzieri (2)

```
    a=5;
#5  b=a+3;
```

- a doua instrucţiune se execută la 5 unităţi de timp după prima
- deci nu mai putem avea execuţie în paralel
  - chiar dacă se folosesc atribuiri neblocante
- dacă nu specificăm o întârziere - similar cu #0

Baze de numeraţie

- valori exprimate în diferite baze de numeraţie
  - baza 2: `7'b0101110`
  - baza 8: `8'o247`
  - baza 10: `8'd25` sau `25` (implicit)
  - baza 16: `12'hA0F`
- numărul dinaintea semnului `'` exprimă întotdeauna numărul de biţi al reprezentării

Înaltă impedanţă

- valorile pe care le poate lua un semnal
  - 0
  - 1
  - `x` (nedeterminat)
  - `z` (înaltă impedanţă)
- exemplu de utilizare
  - `a <= 8'bzzzzzzzz;`
  - valoarea `z` nu poate fi testată

Concatenări de semnale

- sintaxa: **{***semnal***,***semnal***,...}**
- exemplu
  - `reg [3:0]a,b;`
  - `reg [7:0]c,d;`
  - `{a[2],c} <= {d[6:1],b[2:0]};`

Utilizarea limbajului Verilog în proiectarea hardware-ului

Semnale

- elemente de bază în orice implementare fizică
- fiecare semnal poartă 1 bit de informație
- deci în mod fundamental lucrăm cu biți
  - sau cu tablouri de biți

Tipuri de semnale (1)

- pot fi declarate în diverse moduri
  - în funcție de unghiul din care le privim
- intrare-ieșire pentru module
  - `input` - semnal de intrare pentru modulul curent
  - `output` - semnal de ieșire pentru modulul curent
  - `inout` - atât intrare, cât și ieșire

Tipuri de semnale (2)

- semnale interne modulelor
  - `reg` - poate memora valori preluate la anumite momente
    - uzual - implementare secvențială
  - `wire` - realizează legătura între intrarea unui circuit și ieșirea altuia
    - nu poate memora valori
- cele două categorii se pot suprapune

Tipuri de semnale (3)

- semnalele declarate ca `input` sunt automat de tip `wire`
  - nu putem impune valori unui semnal de intrare
- semnalele declarate ca `output` sunt de tip `wire` în mod implicit
  - pot fi conectate combinațional la alte semnale, de exemplu prin `assign`
  - le putem declara ca `reg` în mod explicit

Exemplu

```
module semnale(i,e1,e2);
input i;
output e1,e2;
reg x,e2;
assign e1=~x;
...
endmodule
```


Exemplu (cont.)

- semnalul `i` este intrare, deci nu poate fi `reg`
- semnalele `e1` și `e2` sunt ieșiri
- `e2` este `reg`
  - poate fi modificat direct
- `e1` nu este declarat `reg`
  - poate fi modificat prin intermediul semnalului `x`

Depinde de unde privim semnalul

- același semnal poate fi ieșire pentru un circuit (modul) și intrare pentru altul
  - situația apare de fapt foarte des
- deci și modul de declarare a semnalului poate diferi între cele două module
  - un modul îl poate modifica, celălalt îi poate doar citi valoarea

Exemplu

```
module iesire(s_out);
output s_out;
reg s_out;
...
s_out = 1;
...
endmodule
```


Exemplu (cont.)

```
module intrare(s_in);
input s_in;
reg x; // semnal intern
...
x = s_in;
...
endmodule
```


Exemplu (cont.)

```
module main();
wire s;
iesire a(s);
intrare b(s);
...
endmodule
```


Exemplu (cont.)

- ieșirea primului modul este conectată la intrarea celui de-al doilea
  - deci este de fapt același semnal
  - primul modul îi modifică valoarea
  - al doilea poate doar să o citească
  - la fel și modulul `main`
    - în mod obișnuit nu putem modifica valoarea unui semnal de ieșire a unui modul din afara acestuia

Semnale `inout`

- reutilizarea acelorași pini pentru intrări și ieșiri
  - la momente diferite de timp
  - util doar la module top-level
- de fapt sunt componente diferite care lucrează cu intrarea, respectiv ieșirea
  - ceea ce scoatem pe ieșire nu trebuie să interfereze cu ceea ce primim pe intrare

Exemplu

```
module in_out(Clk,data);
input Clk;
inout [7:0]data;
reg [7:0]d,x;
assign data=d;
initial
  d<=8'bzzzzzzzz;
```


Exemplu (cont.)

```
always @(posedge Clk)
begin
  x=data;
  #5 d<=x+1;
  #5 d<=8'bzzzzzzzz;
end
endmodule
```


Exemplu (cont.)

- semnalul `data` privit ca ieșire
  - legat (combinațional) la ieșirea variabilei `d`
  - deci orice modificare a valorii `d` se regăsește automat pe ieșirea `data`
  - când nu avem nimic de transmis pe ieșire, `d` trebuie să fie în înaltă impedanță
- semnalul `data` privit ca intrare
  - este citit în variabila `x` la anumite momente

Exemplu (cont.)

- atunci când scriem la ieșire, semnalul de la intrarea `data` trebuie să fie la rândul său în înaltă impedanță
  - provine de la alt modul
  - deci trebuie să existe un protocol de comunicare
  - sarcina proiectantului

Operații cu semnalele

- operatorii - sintaxă similară cu limbajul C
- operațiile pe biți - mult mai des utilizate
- unele operații pe biți se pot realiza mult mai eficient în hardware
- dacă un singur bit dintr-o variabilă are valoarea `x` (nedeterminat), toată variabila are valoarea `x`

Operatori de reducere

- funcţii booleene de aritate 2 (sau mai mare)
- operatori: `| & ^`
- pot fi aplicaţi asupra biţilor dintr-un vector
- exemplu: test dacă o variabilă are valoarea 0
  - `reg [7:0]a;`
  - `reg b;`
  - `b <= ~(|a);`

Tipuri auxiliare

- `integer`
- `real`
- `time`
- variabilele din aceste tipuri folosite nu sunt pentru sinteza circuitelor, ci pentru
  - exprimarea operațiilor paralele
  - simulare

Șiruri de caractere

- nu există un tip dedicat
- constante - între caracterele "..."
- variabile - declarate ca `reg`
  - trebuie să conțină un număr suficient de biți
  - dacă sunt mai mulți - completate cu 0 la dreapta

```
reg [20*8:1]s; // 20 de octeti
s="sir de caractere";
```


Bucle

- implementare
  - nivel jos
  - test + salt
- implementarea salturilor
  - prin stabilirea stării următoare
- structuri repetitive (`while`/`for`/`repeat`)
  - nu sunt destinate descrierii buclelor secvenţiale
  - rol - descrierea operaţiilor care se pot realiza în paralel

Exemplu

- inversarea ordinii biților într-o variabilă

```
reg [7:0]a;
integer i;
a<=8'b01010000;
#5 for(i=0;i<8;i=i+1)
  a[i]<=a[7-i];
```


Exemplu (cont.)

- atenție la precizarea întârzierilor

```
a<=8'b01010101;
for(i=0;i<8;i=i+1)
   #5 a[i]<=a[7-i];
```

- în acest caz, fiecare atribuire din buclă se face la 5 unități de timp după cea anterioară
- nu se fac în paralel - rezultat incorect

Proiectarea sistemelor secvenţiale

Principii

- modelarea - în principal comportamentală
  - se pot utiliza şi elemente structurale pentru unele componente
- trebuie luat în considerare semnalul de ceas
  - în mod explicit

Exemplu 1

- bistabil D - latch

```
module Latch(Clk,D,Q);
input Clk,D;
output Q;
reg Q;
always @(Clk,D)
  if(Clk==1) Q<=D;
endmodule
```


Exemplu 2

- bistabil D - flip-flop
- considerăm şi intrările R şi S
  - asincrone (nu depind de semnalul de ceas)
  - prioritare
  - necesare pentru iniţializare şi alte operaţii
- tratarea semnalului de ceas
  - contează doar frontul crescător
  - trebuie detectat

Exemplu 2 - varianta 1

```
module FlipFlop(Clk,D,R,S,Q);
input Clk,D,R,S;
output Q;
reg Q;
always @(posedge Clk,D,R,S)
  if(R==1) Q<=0;
  else if(S==1)
     Q<=1;
```


Exemplu 2 - varianta 1 (cont.)

```
  else if(posedge Clk)
     Q<=D;
endmodule
```


Exemplu 2 - varianta 1 (cont.)

- detectarea frontului crescător - `posedge`
  - front descrescător - `negedge`
- varianta NU ESTE CORECTĂ
  - frontul crescător poate fi precizat în lista de sensibilitate a structurii `always`
  - dar nu poate fi testat într-o instrucţiune `if`
  - pot fi testate doar valori ale semnalelor, nu şi tranziţii ale acestora

Exemplu 2 - varianta 2

```
module FlipFlop(Clk,D,R,S,Q);
input Clk,D,R,S;
output Q;
reg Q;
always @(R,S)
  if(R==1) Q<=0;
  else if(S==1)
     Q<=1;
```

Exemplu 2 - varianta 2 (cont.)

```
always @(posedge Clk)
  if(R==0 && S==0)
     Q<=D;
endmodule
```


Exemplu 2 - varianta 2 (cont.)

- se pot folosi 2 sau mai multe structuri `always`
  - şi `initial`
- toate se execută în paralel
- intrările R şi S trebuie să fie prioritare
  - sunt testate în cealaltă structură `always`
- combinaţiile sincron-asincron - dificil de tratat

Automate

- două componente
  - evoluţia stării
    - pe baza semnalului de ceas
  - elemente de acţionare
    - descriere de obicei comportamentală
    - calculează valorile pentru starea următoare şi ieşirile
- izolare - structuri `always` separate
  - modularitate - mai uşor de înţeles şi controlat

Structura de bază

```
module nume_modul(Clk,intrari,iesiri);
input Clk,intrari;
output iesiri;
reg iesiri;
reg stare_curenta,stare_urmatoare;
always @(posedge Clk)
  stare_curenta<=stare_urmatoare;
  // eventual reset, ...
```

Structura de bază (cont.)

```
always @(intrari,stare_curenta)
  begin
  // calcul iesiri, stare urmatoare
  end
endmodule
```


Structura de bază (cont.)

- partea combinaţională răspunde imediat la orice modificare a intrărilor sale
  - starea curentă
  - intrări
- partea secvenţială permite modificarea doar pe frontul crescător al ceasului
- nu e obligatoriu ca `stare_curenta` şi `stare_urmatoare` să fie implementate ca regiştri separaţi

Exemplu

- măsurarea perioadei unui semnal
- digitizare - transformare într-un semnal cu valori doar 0 şi 1
  - perioada rămâne aceeaşi
  - nu este sigur că valorile de 0 şi respectiv 1 au aceeaşi durată în cadrul unei perioade
- aceleaşi obiective ca la cursul 4

Varianta 1

```
module P(start,i,digit,print);
input start,i;
output reg [3:0]digit;
output reg print;
always @(start,i)
begin
  digit=0;
  print=0;
```

Varianta 1 (cont.)

```
if(i==0) begin
   while(i==0) ;
   while(i==1) digit=digit+1;
   while(i==0) digit=digit+1;
   print=1;
end
else begin
   while(i==1) ;
```


Varianta 1 (cont.)

```
   while(i==0) digit=digit+1;
   while(i==1) digit=digit+1;
   print=1;
  end
end
endmodule
```


Varianta 1 (cont.)

- problema
  - nu apare semnalul de ceas
  - la ce momente au loc toate acţiunile?
- semnalul de ceas trebuie să apară explicit ca intrare
- o instrucţiune poate fi întârziată până la apariţia unui eveniment
  - ex.: `@(posedge clk) digit=digit+1;`

Varianta 1 (cont.)

- o soluţie ar fi deci ca toate acţiunile să fie întârziate în acest mod

```
while(i==0) @(posedge clk);
while(i==1)
   @(posedge clk) digit=digit+1;
while(i==0)
   @(posedge clk) digit=digit+1;
```

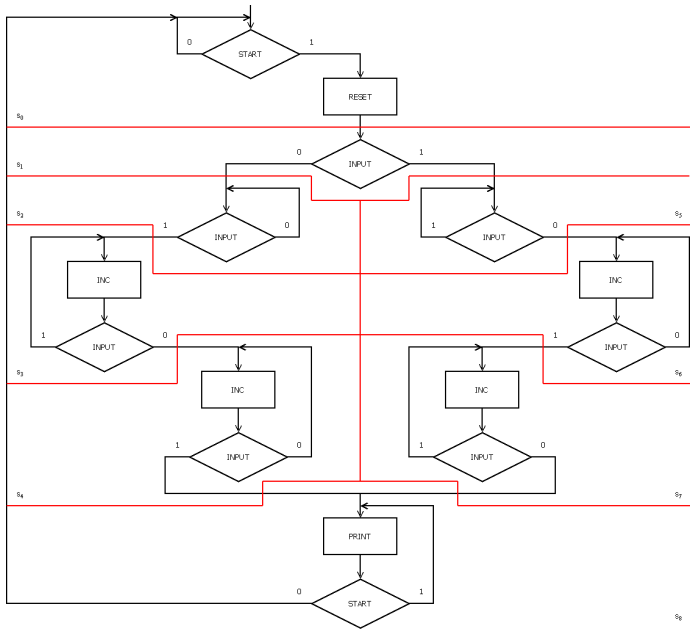
- foarte greu de gestionat

Varianta 2

- revenim la proiectarea automatului
  - la fel ca la cursul 4
  - orientativ - ne interesează stările, nu exact semnalele de comandă
    - implementarea acestora o realizează compilatorul
    - ex.: în loc de RESET vom scrie

```
digit<=0; display<=0;
```

    - descrierea - comportamentală



Varianta 2 - implementare

```
module P(Clk,start,i,digit,print);
input Clk,start,i;
output reg [3:0]digit;
output reg print;
reg [3:0]state;
reg [3:0]state_next;
always @(posedge Clk)
  state<=state_next;
```


Varianta 2 - implementare (cont.)

```
always @(start,i,state)
  if(state==0)
  begin
     if(start==0) state_next<=0;
     else state_next<=1;
     digit<=0;
     print<=0;
  end
```

Varianta 2 - implementare (cont.)

```
  else if(state==1) begin
     if(i==1) state_next<=2;
     else state_next<=5;
  end
  ... // celelalte stari
initial
  state<=0; state_next<=0;
endmodule
```


Modelarea prin automate (1)

- limbajul Verilog este util mai ales pentru partea de implementare
- descrierea algoritmului de funcţionare rămâne sarcina proiectantului
  - modelare pe bază de stări (automat)
- atenţie la atribuirea de valori pentru ieşiri
  - teoretic - toate ieşirile primesc valori la fiecare stare
  - practic - atunci când se schimbă valorile

Modelarea prin automate (2)

Avantaj

- separă partea de memorare a stării (secvenţială) de cea combinaţională
  - două blocuri `always` independente
    - unul actualizează starea
    - celălalt realizează calculul valorilor pentru ieşiri şi starea următoare
  - descriere clară

Modelarea prin automate (3)

Probleme

- la descrierea părţii combinaţionale
  - blocul `always` reacţionează la modificarea unei variabile
  - nu este potrivit pentru integrarea într-un sistem secvenţial

Exemplu

```
module ex(Clk,x,y,z);
input Clk,x,y;
output z;
reg z;
//starea - cati biti sunt necesari
reg [2:0]state,state_next;
always @(posedge Clk)
  state<=state_next;
```


Exemplu (cont.)

```
always @(x,y,state)
  if(state==0)
     if(x==1) begin
          z<=z+1;
          state_next<=0;
     end
     else state_next<=1;
  ...
```


Problema 1

- starea 0 - se incrementează `z` cât timp `x` are valoarea 1
  - buclă de program
- suntem în starea 0 şi starea următoare este tot 0
- dacă `x` şi `y` nu variază, `z` nu e incrementat
  - `always` nu detectează vreo schimbare pe nici una din variabile, deci nu se activează

Problema 2

- starea 0 - apare o modificare a valorii variabilei `y`
- nu ar trebui să influenţeze cu nimic evoluţia sistemului
  - deoarece `y` nu intervine în starea 0
  - dar `always` detectează o schimbare pe una din variabile, deci se activează
  - ca urmare, `z` are o incrementare parazită

Soluţii (1)

- `always` să reacţioneze şi la schimbările semnalului de ceas
  - sau numai la frontul crescător al acestuia
  - pot apărea interferenţe cu blocul `always` care se ocupă de actualizarea stării
    - atribuirile neblocante (<=) se execută în paralel şi atunci când sunt în blocuri `always/initial` diferite
  - nu rezolvă a doua problemă

Soluţii (2)

- toate variabilele care apar în lista de sensibilitate a blocului `always` să fie testate prin instrucţiuni `if` în toate stările
  - s-ar rezolva a doua problemă
  - foarte greu de realizat practic

Soluţii (3)

- descriere structurală (de nivel jos) a părţii combinaţionale
  - fără blocuri `always`
  - ecuaţii booleene şi/sau circuite predefinite
  - ar rezolva ambele probleme
  - puterea limbajului Verilog se pierde în foarte mare măsură dacă renunţăm la descrierile comportamentale

Soluţii (4)

- un singur vector de stare (în loc de doi)
- un singur bloc `always`
  - activat doar de frontul crescător al semnalului de ceas
  - toate calculele se (re)fac pe frontul crescător
  - deci toate variabilele sunt utilizate şi actualizate numai pe frontul crescător
- ambele probleme sunt rezolvate

Implementare

```
module nume_modul(Clk,intrari,iesiri);
input Clk,intrari;
output iesiri;
reg iesiri;
reg stare;
always @(posedge Clk)
  if(stare==0)
     //descriere stare 0
```


Implementare (cont.)

```
     //actualizare stare
  else if(stare==1)
     //descriere stare 1
     //actualizare stare
  ... //alte stari
endmodule
```


Limitare

- pentru variabilele de intrare se iau în considerare doar valorile de la momentele când apare un front crescător al ceasului
  - dacă o variabilă de intrare este activată doar între două fronturi crescătoare consecutive, va fi ignorată
  - în general nu este o problemă
    - variabilele de intrare variază lent
    - la nevoie, intrările care provin de la alte circuite pot fi sincronizate după acelaşi semnal de ceas

Exemplu

- calculul celui mai mare divizor comun a două numere
- algoritmul lui Euclid
- numerele sunt primite simultan din exterior
- semnale de notificare
  - `start` - numerele sunt disponibile la intrare
  - `ack` - rezultatul este disponibil la ieşire

Implementare

```
module
  cmmdc(Clk,start,x,y,r,ack);
input Clk,start;
input [7:0]x,y;
output reg [7:0]r;
output reg ack;
reg [7:0]a,b;
reg [1:0]state;
```


Implementare (cont.)

```
always @(posedge Clk)
  if(state==0)
     if(start==1) begin
          a<=x;
          b<=y;
          ack<=0;
          state<=1;
     end
     else state<=0;
```


Implementare (cont.)

```
  else if(state==1)
     if(a==b) begin
          r<=a;
          state<=2;
     end
     else if(a>b) begin
          a<=a-b;
          state<=1;
     end
```


Implementare (cont.)

```
     else begin
          b<=b-a;
          state<=1;
     end
  else if(state==2) begin
          ack<=1;
          state<=0;
     end
```

Implementare (cont.)

```
initial
begin
  state<=0;
  ack<=0;
end
endmodule
```