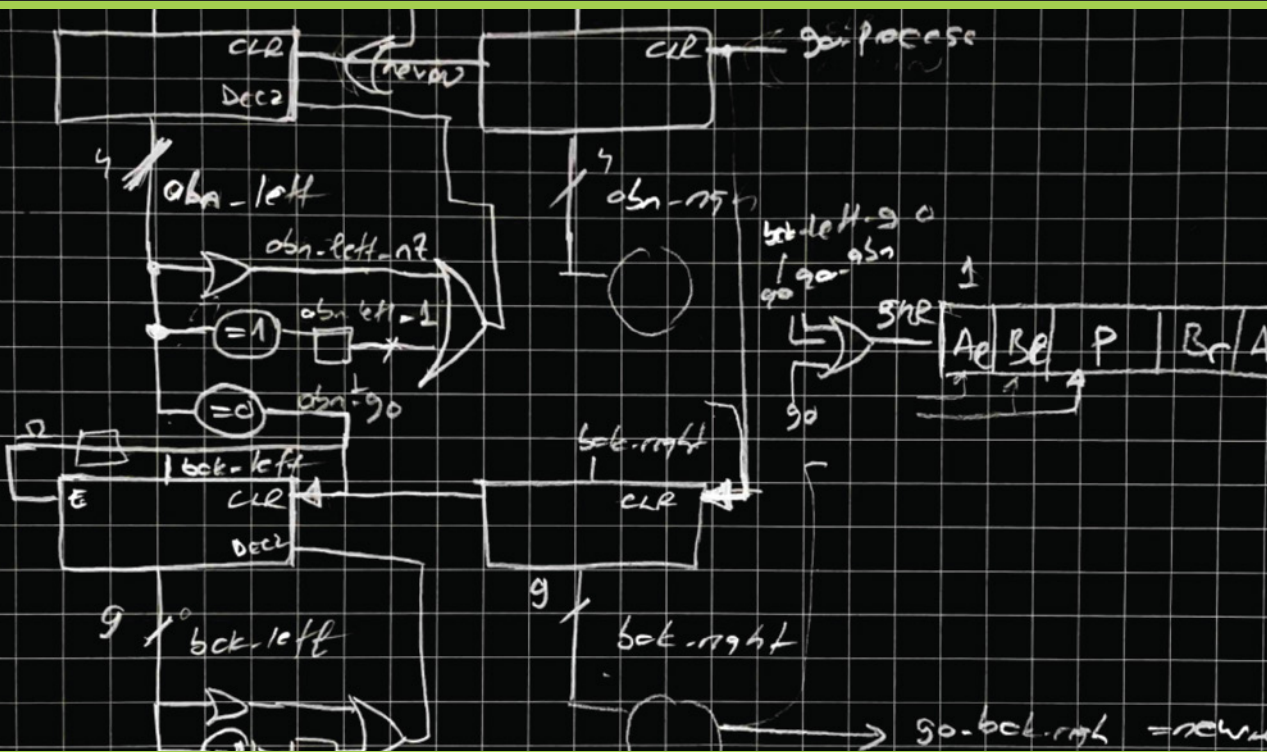


# TOMUL PENTRU ZĂBOVIT ȘI GÂNDIT

# DAN NICULA

# ELECTRONICĂ DIGITALĂ

# CARTE DE ÎNVĂȚĂTURĂ





**Dan NICULA**

# **ELECTRONICĂ DIGITALĂ**

**Carte de învățatură**



**EDITURA  
UNIVERSITĂȚII  
TRANSILVANIA  
DIN BRAȘOV**

2025

**EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV**

**Adresa: Str. Iuliu Maniu nr. 41A**

**500091 Brașov**

**Tel.: 0268 476 050**

**Fax: 0268 476 051**

**E-mail: editura@unitbv.ro**

**Editură acreditată de CNC SIS, cod 81**

**ISBN 978-606-19-1794-5 (ebook)**

**Copyright ©2025 Autorul**

Lucrarea a fost avizată de

Consiliul Departamentului de Electronică și Calculatoare din cadrul

Facultății de Inginerie Electrică și Știința Calculatoarelor,

Universitatea Transilvania din Brașov.



# Cuprins

---

|                                                                                   |           |
|-----------------------------------------------------------------------------------|-----------|
| <b>Electronică digitală</b>                                                       | <b>I</b>  |
| <b>Cuprins</b>                                                                    | <b>i</b>  |
| <b>Cuvântul cel dintâi către „cetitor”</b>                                        | <b>xi</b> |
| <b>Tomul pentru zăbovit și gândit</b>                                             | <b>1</b>  |
| <b>Începutul învățaturii „Tomului pentru zăbovit și gândit”</b>                   | <b>3</b>  |
| <b>1 De la matematică la inginerie</b>                                            | <b>5</b>  |
| 1.1 Lumea digitală. Între 0 și 1                                                  | 7         |
| 1.1.1 De ce este nevoie de „Electronică digitală”?                                | 8         |
| 1.1.2 Informația. Semnale                                                         | 12        |
| 1.2 Matematică binară                                                             | 16        |
| 1.2.1 Reprezentarea numerelor                                                     | 17        |
| 1.2.2 Conversia numerelor între bazele de numerație 2, 10, 16                     | 22        |
| 1.2.3 Reprezentarea numerelor întregi în complement față de 2                     | 28        |
| 1.2.4 Adunarea și scăderea numerelor întregi reprezentate în complement față de 2 | 31        |
| 1.2.5 Codul ASCII                                                                 | 33        |
| 1.2.6 Codul BCD                                                                   | 35        |
| 1.2.7 Algebra Booleană                                                            | 36        |
| 1.3 Inginerie binară                                                              | 38        |
| 1.3.1 Porți logice                                                                | 38        |
| 1.3.2 Justificarea noțiunii de poartă logică                                      | 40        |
| 1.3.3 Set complet de operatori logici                                             | 45        |
| 1.3.4 Parametrii porților și semnalelor logice                                    | 45        |
| 1.3.5 Timp de propagare                                                           | 51        |
| 1.4 Modelarea sistemelor digitale în HDL                                          | 54        |
| 1.4.1 De ce este nevoie de HDL?                                                   | 55        |
| 1.4.2 Metodologia de proiectare a sistemelor digitale                             | 57        |

|          |                                                    |            |
|----------|----------------------------------------------------|------------|
| 1.4.3    | Mediul de simulare HDL                             | 59         |
| 1.4.4    | Modele HDL                                         | 60         |
| 1.4.5    | Setul de valori                                    | 63         |
| 1.4.6    | Tipuri de date                                     | 63         |
| 1.4.7    | Codați cu stil!                                    | 65         |
| 1.4.8    | Hard-ul nu-i ca soft-ul!                           | 67         |
| <b>2</b> | <b>Circuite logice combinaționale</b>              | <b>69</b>  |
| 2.1      | Reprezentarea circuitelor logice combinaționale    | 72         |
| 2.1.1    | Tabele de adevăr                                   | 72         |
| 2.1.2    | Forme standard                                     | 74         |
| 2.1.3    | Diagrame Veitch-Karnaugh                           | 76         |
| 2.2      | Analiza circuitelor logice combinaționale          | 79         |
| 2.3      | Sinteza circuitelor logice combinaționale          | 84         |
| 2.3.1    | Minimizarea funcțiilor logice, diagrame V-K        | 84         |
| 2.3.2    | Minimizarea funcțiilor logice incomplet definite   | 86         |
| 2.3.3    | Minimizarea funcțiilor cu variabile reziduu        | 87         |
| 2.3.4    | Minimizarea funcțiilor logice cu ieșiri multiple   | 88         |
| 2.4      | CLC particulare                                    | 90         |
| 2.4.1    | Decodificator                                      | 91         |
| 2.4.2    | Codificator                                        | 99         |
| 2.4.3    | Multiplexor                                        | 101        |
| 2.4.4    | Demultiplexor                                      | 110        |
| 2.4.5    | Sumator                                            | 117        |
| 2.4.6    | Comparator                                         | 121        |
| 2.4.7    | Unitate logică și aritmetică                       | 123        |
| 2.4.8    | Look-Up-Table și memorii ROM                       | 126        |
| 2.5      | Modelarea circuitelor logice combinaționale în HDL | 129        |
| 2.5.1    | Operatori                                          | 129        |
| 2.5.2    | Operatori aritmetici                               | 131        |
| 2.5.3    | Operatori relaționali                              | 138        |
| 2.5.4    | Operatori de egalitate                             | 138        |
| 2.5.5    | Operatori logici                                   | 139        |
| 2.5.6    | Operatori logici pe vectori de biți                | 140        |
| 2.5.7    | Operatori de reducere                              | 141        |
| 2.5.8    | Operatori de deplasare                             | 143        |
| 2.5.9    | Operatorul condițional                             | 145        |
| 2.5.10   | Operatori de concatenare și replicare a biților    | 145        |
| 2.5.11   | Precedența operatorilor                            | 146        |
| 2.5.12   | Modelarea circuitelor logice combinaționale        | 149        |
| <b>3</b> | <b>Circuite logice secvențiale</b>                 | <b>157</b> |
| 3.1      | Circuite de memorare a stării                      | 159        |
| 3.1.1    | Latch                                              | 163        |
| 3.1.2    | Bistabile                                          | 167        |

|          |                                                                    |            |
|----------|--------------------------------------------------------------------|------------|
| 3.1.3    | Inițializarea bistabilelor                                         | 181        |
| 3.2      | Automate secvențiale                                               | 183        |
| 3.3      | Reprezentarea automatelor                                          | 186        |
| 3.3.1    | Tabele de tranziții                                                | 187        |
| 3.3.2    | Organigrame                                                        | 190        |
| 3.3.3    | Grafuri                                                            | 191        |
| 3.3.4    | Analiza circuitelor logice secvențiale                             | 193        |
| 3.3.5    | Sinteza automatelor                                                | 197        |
| 3.3.6    | Modelarea automatelor în Verilog                                   | 200        |
| 3.4      | Registre și numărătoare                                            | 206        |
| 3.4.1    | Registre cu accesare serie sau paralel                             | 207        |
| 3.4.2    | Numărătoare                                                        | 211        |
| 3.4.3    | Linear-Feedback-Shift-Register                                     | 211        |
| 3.4.4    | Modelarea HDL a registrelor și a numărătoarelor                    | 213        |
| 3.5      | Memorii                                                            | 225        |
| 3.5.1    | Memoria ROM sincronă                                               | 225        |
| 3.5.2    | Memoria RAM                                                        | 228        |
| 3.5.3    | Structuri de memorii                                               | 237        |
| <b>4</b> | <b>Considerente ingineresti ale sistemelor digitale</b>            | <b>245</b> |
| 4.1      | Timpul în sistemele digitale                                       | 248        |
| 4.1.1    | Căi de propagare, frecvența semnalului de ceas, slack              | 248        |
| 4.1.2    | Parametrii semnalului de ceas: skew, jitter, drift                 | 251        |
| 4.1.3    | Metastabilitatea bistabilelor                                      | 255        |
| 4.1.4    | Domenii de ceas, circuite de sincronizare                          | 256        |
| 4.1.5    | Structura pipeline                                                 | 259        |
| 4.2      | Protocoale de comunicare sincrone între module digitale            | 262        |
| 4.2.1    | Protocolul „request-acknowledge”                                   | 262        |
| 4.2.2    | Protocolul „valid-ready”                                           | 266        |
| 4.3      | Tehnologia circuitelor integrate digitale                          | 270        |
| 4.3.1    | Metodologia de proiectare a circuitelor integrate digitale         | 271        |
| 4.3.2    | Fabricarea circuitelor integrate                                   | 274        |
| 4.3.3    | Tehnologia CMOS                                                    | 275        |
| 4.4      | Noțiuni avansate de Verilog                                        | 279        |
| 4.4.1    | Modelarea comportamentală RTL                                      | 281        |
| 4.4.2    | Medii de testare                                                   | 288        |
|          | <b>Tomul pentru zăbovit și gândit - Final</b>                      | <b>300</b> |
|          | <b>Tomul pentru chibzuit și buchisit</b>                           | <b>301</b> |
|          | <b>Începutul învățaturii „Tomului pentru chibzuit și buchisit”</b> | <b>303</b> |
| <b>1</b> | <b>Reprezentarea datelor în sistemele digitale</b>                 | <b>305</b> |
| 1.1      | Pentru seminar                                                     | 305        |

|          |                                                        |            |
|----------|--------------------------------------------------------|------------|
| 1.2      | Pentru cei ce vor doar să promoveze examenul           | 306        |
| 1.3      | Pentru cei ce vor să învețe                            | 307        |
| 1.4      | Pentru cei ce vor să devină profesioniști              | 316        |
| <b>2</b> | <b>Algebră Booleană, porți logice, funcții logice</b>  | <b>319</b> |
| 2.1      | Pentru seminar                                         | 319        |
| 2.2      | Pentru cei ce vor doar să promoveze examenul           | 323        |
| 2.3      | Pentru cei ce vor să învețe                            | 327        |
| 2.4      | Pentru cei ce vor să devină profesioniști              | 344        |
| <b>3</b> | <b>Minimizare cu diagrame Veitch-Karnaugh</b>          | <b>351</b> |
| 3.1      | Pentru seminar                                         | 351        |
| 3.2      | Pentru cei ce vor doar să promoveze examenul           | 351        |
| 3.3      | Pentru cei ce vor să învețe                            | 352        |
| 3.4      | Pentru cei ce vor să devină profesioniști              | 356        |
| <b>4</b> | <b>Implementare CLC cu porți NAND</b>                  | <b>363</b> |
| 4.1      | Pentru seminar                                         | 363        |
| 4.2      | Pentru cei ce vor doar să promoveze examenul           | 365        |
| 4.3      | Pentru cei ce vor să învețe                            | 366        |
| 4.4      | Pentru cei ce vor să devină profesioniști              | 371        |
| <b>5</b> | <b>Multiplexoare, demultiplexoare și codificatoare</b> | <b>377</b> |
| 5.1      | Pentru seminar                                         | 377        |
| 5.2      | Pentru cei ce vor doar să promoveze examenul           | 378        |
| 5.3      | Pentru cei ce vor să învețe                            | 379        |
| 5.4      | Pentru cei ce vor să devină profesioniști              | 380        |
| <b>6</b> | <b>Modelarea circuitelor logice combinaționale</b>     | <b>393</b> |
| 6.1      | Pentru seminar                                         | 393        |
| 6.2      | Pentru cei ce vor doar să promoveze examenul           | 397        |
| 6.3      | Pentru cei ce vor să învețe                            | 398        |
| 6.4      | Pentru cei ce vor să devină profesioniști              | 399        |
| 6.5      | Concluzie: Inginer vs. ChatGPT                         | 409        |
| <b>7</b> | <b>Circuite cu bistabile</b>                           | <b>415</b> |
| 7.1      | Pentru seminar                                         | 415        |
| 7.2      | Pentru cei ce vor doar să promoveze examenul           | 418        |
| 7.3      | Pentru cei ce vor să învețe                            | 419        |
| 7.4      | Pentru cei ce vor să devină profesioniști              | 422        |
| <b>8</b> | <b>Analiza circuitelor logice secvențiale</b>          | <b>425</b> |
| 8.1      | Pentru seminar                                         | 425        |
| 8.2      | Pentru cei ce vor doar să promoveze examenul           | 426        |
| 8.3      | Pentru cei ce vor să învețe                            | 426        |
| 8.4      | Pentru cei ce vor să devină profesioniști              | 432        |

|                                                   |            |
|---------------------------------------------------|------------|
| <b>9 Sinteza circuitelor logice secvențiale</b>   | <b>439</b> |
| 9.1 Pentru seminar                                | 439        |
| 9.2 Pentru cei ce vor doar să promoveze examenul  | 441        |
| 9.3 Pentru cei ce vor să învețe                   | 441        |
| 9.4 Pentru cei ce vor să devină profesioniști     | 449        |
| <b>10 Modelarea automatelor în Verilog</b>        | <b>453</b> |
| 10.1 Pentru seminar                               | 455        |
| 10.2 Pentru cei ce vor doar să promoveze examenul | 457        |
| 10.3 Pentru cei ce vor să învețe                  | 457        |
| 10.4 Pentru cei ce vor să devină profesioniști    | 459        |
| <b>11 Registre și numărătoare</b>                 | <b>469</b> |
| 11.1 Pentru seminar                               | 469        |
| 11.2 Pentru cei ce vor doar să promoveze examenul | 471        |
| 11.3 Pentru cei ce vor să învețe                  | 471        |
| 11.4 Pentru cei ce vor să devină profesioniști    | 474        |
| <b>12 Memorii</b>                                 | <b>485</b> |
| 12.1 Pentru seminar                               | 485        |
| 12.2 Pentru cei ce vor doar să promoveze examenul | 486        |
| 12.3 Pentru cei ce vor să învețe                  | 486        |
| 12.4 Pentru cei ce vor să devină profesioniști    | 488        |
| <b>13 Proiectarea sistemelor digitale</b>         | <b>495</b> |
| 13.1 Pentru seminar                               | 495        |
| 13.2 Pentru cei ce vor doar să promoveze examenul | 496        |
| 13.3 Pentru cei ce vor să învețe                  | 496        |
| 13.4 Pentru cei ce vor să devină profesioniști    | 507        |
| <b>14 Probleme de examen</b>                      | <b>511</b> |
| 14.1 Subiectul 1                                  | 511        |
| 14.1.1A. Noțiuni generale                         | 511        |
| 14.1.2B. Circuite logice combinaționale           | 516        |
| 14.1.3C. Circuite logice secvențiale              | 518        |
| 14.1.4D. Proiectare sistem digital                | 521        |
| 14.2 Subiectul 2                                  | 523        |
| 14.2.1A. Noțiuni generale                         | 523        |
| 14.2.2B. Circuite logice combinaționale           | 528        |
| 14.2.3C. Circuite logice secvențiale              | 531        |
| 14.2.4D. Proiectare sistem digital                | 532        |
| 14.3 Subiectul 3                                  | 534        |
| 14.3.1A. Noțiuni generale                         | 534        |
| 14.3.2B. Circuite logice combinaționale           | 539        |
| 14.3.3C. Circuite logice secvențiale              | 540        |
| 14.3.4D. Proiectare sistem digital                | 541        |

|           |                                                                  |            |
|-----------|------------------------------------------------------------------|------------|
| 14.4      | Subiectul 4                                                      | 542        |
| 14.4.1A.  | Noțiuni generale                                                 | 543        |
| 14.4.2B.  | Circuite logice combinaționale                                   | 548        |
| 14.4.3C.  | Circuite logice secvențiale                                      | 550        |
| 14.4.4D.  | Proiectare sistem digital                                        | 552        |
| 14.5      | Subiectul 5                                                      | 553        |
| 14.5.1A.  | Noțiuni generale                                                 | 553        |
| 14.5.2B.  | Circuite logice combinaționale                                   | 558        |
| 14.5.3C.  | Circuite logice secvențiale                                      | 559        |
| 14.5.4D.  | Proiectare sistem digital                                        | 562        |
| <b>15</b> | <b>Subiecte pentru evaluat ingineri</b>                          | <b>565</b> |
| 15.1      | Ingineri începători                                              | 565        |
| 15.1.1    | Cunoștințe generale                                              | 565        |
| 15.1.2    | Definirea noțiunilor                                             | 566        |
| 15.1.3    | Circuite cu bistabile                                            | 575        |
| 15.1.4    | Proiectare circuite secvențiale pe baza specificațiilor          | 579        |
| 15.1.5    | Proiectare circuite secvențiale după model                       | 588        |
| 15.2      | Ingineri avansați                                                | 589        |
| 15.2.1    | Analiza circuitelor digitale                                     | 589        |
| 15.2.2    | Proiectare sisteme digitale de control                           | 596        |
|           | <b>Tomul pentru chibzuit și buchisit - Final</b>                 | <b>610</b> |
|           | <b>Tomul pentru meșterit și muncit</b>                           | <b>611</b> |
|           | <b>Începutul învățaturii „Tomului pentru meșterit și muncit”</b> | <b>613</b> |
| <b>1</b>  | <b>Modelarea circuitelor cu porți logice</b>                     | <b>615</b> |
| 1.1       | Desfășurarea lucrării                                            | 615        |
| 1.2       | Temă de casă                                                     | 618        |
| <b>2</b>  | <b>Modelarea circuitelor logice</b>                              | <b>623</b> |
| 2.1       | Desfășurarea lucrării                                            | 624        |
| 2.2       | Temă de casă                                                     | 626        |
| <b>3</b>  | <b>Desenare diagrame de timp, circuite și scheme</b>             | <b>629</b> |
| 3.1       | Desfășurarea lucrării                                            | 631        |
| 3.2       | Temă de casă                                                     | 633        |
| <b>4</b>  | <b>Modelarea circuitelor digitale în Verilog</b>                 | <b>635</b> |
| 4.1       | Noțiuni introductive                                             | 635        |
| 4.1.1     | Definirea termenilor                                             | 635        |
| 4.1.2     | Modele asociate unui proiect                                     | 637        |
| 4.1.3     | Organizarea fișierelor                                           | 638        |

|                                                                  |            |
|------------------------------------------------------------------|------------|
| 4.1.4 Metodologia de proiectare a circuitelor integrate digitale | 640        |
| 4.1.5 Metodologia de simulare HDL cu <i>ModelSim</i>             | 642        |
| 4.2 Desfășurarea lucrării                                        | 650        |
| 4.3 Temă de casă                                                 | 651        |
| <b>5 Multiplexoare, demultiplexoare, codificatoare</b>           | <b>653</b> |
| 5.1 Desfășurarea lucrării                                        | 654        |
| 5.2 Temă de casă                                                 | 656        |
| <b>6 Modelarea circuitelor logice combinaționale în Verilog</b>  | <b>659</b> |
| 6.1 Desfășurarea lucrării                                        | 660        |
| 6.2 Teme de casă                                                 | 663        |
| <b>7 Modelarea circuitelor cu bistabile</b>                      | <b>665</b> |
| 7.1 Desfășurarea lucrării                                        | 667        |
| 7.1.1 Mediu de testare pentru circuite cu bistabile              | 669        |
| 7.2 Teme de casă                                                 | 672        |
| <b>8 Modelarea circuitelor secvențiale cu bistabile</b>          | <b>677</b> |
| 8.1 Desfășurarea lucrării                                        | 677        |
| 8.2 Temă de casă                                                 | 680        |
| <b>9 Modelarea FSM</b>                                           | <b>683</b> |
| 9.1 Desfășurarea lucrării                                        | 684        |
| 9.2 Temă de casă                                                 | 686        |
| <b>10 Modelarea registrelor și a numărătoarelor</b>              | <b>689</b> |
| 10.1 Desfășurarea lucrării                                       | 689        |
| 10.1.1 Registru cu validare                                      | 689        |
| 10.1.2 Numărător                                                 | 690        |
| 10.1.3 LFSR                                                      | 691        |
| 10.2 Temă de casă                                                | 692        |
| <b>11 Modelarea memoriilor</b>                                   | <b>695</b> |
| 11.1 Noțiuni teoretice                                           | 695        |
| 11.1.1 Memorii                                                   | 696        |
| 11.1.2 Inițializarea memoriei cu o constantă                     | 698        |
| 11.1.3 Inițializarea memoriei                                    | 700        |
| 11.1.4 Funcții pentru accesul la fișiere de intrare/ieșire       | 700        |
| 11.1.5 Inițializarea memoriei din fișier                         | 701        |
| 11.1.6 Salvarea conținutului memoriei în fișier                  | 702        |
| 11.2 Desfășurarea lucrării                                       | 703        |
| 11.2.1 Testarea memoriei single-port                             | 703        |
| 11.2.2 Operațiuni cu fișiere                                     | 704        |
| 11.2.3 Modificări asupra mediului de testare                     | 705        |
| 11.3 Temă de casă                                                | 706        |

|                                                                   |            |
|-------------------------------------------------------------------|------------|
| 11.4 Fișiere cu modelul mediului de test                          | 708        |
| 11.4.1 Modulul generatorului de stimuli                           | 708        |
| 11.4.2 Modulul mediului de testare                                | 709        |
| <b>12 Modelarea sistemelor digitale complexe</b>                  | <b>713</b> |
| 12.1 Multiplicator secvențial                                     | 714        |
| 12.2 Circuit watchdog                                             | 719        |
| 12.3 Circuit pipeline pentru determinarea maximului dintre 3 date | 724        |
| <b>13 Subiecte pentru colocviul de laborator</b>                  | <b>735</b> |
| <b>Tomul pentru meșterit și muncit - Final</b>                    | <b>742</b> |
| <b>Tomul pentru grabă mare</b>                                    | <b>743</b> |
| <b>Începutul învățaturii „Tomului pentru grabă mare”</b>          | <b>745</b> |
| <b>1 Aritmetică binară</b>                                        | <b>749</b> |
| 1.1 Întrebări                                                     | 749        |
| 1.2 Răspunsuri                                                    | 758        |
| <b>2 Porți logice</b>                                             | <b>761</b> |
| 2.1 Întrebări                                                     | 761        |
| 2.2 Răspunsuri                                                    | 772        |
| <b>3 Funcții logice</b>                                           | <b>775</b> |
| 3.1 Întrebări                                                     | 775        |
| 3.2 Răspunsuri                                                    | 784        |
| <b>4 Circuite logice combinaționale</b>                           | <b>785</b> |
| 4.1 Întrebări                                                     | 785        |
| 4.2 Răspunsuri                                                    | 791        |
| <b>5 Circuite logice combinaționale particulare</b>               | <b>793</b> |
| 5.1 Întrebări                                                     | 793        |
| 5.2 Răspunsuri                                                    | 807        |
| <b>6 Elemente de memorare</b>                                     | <b>809</b> |
| 6.1 Întrebări                                                     | 809        |
| 6.2 Răspunsuri                                                    | 821        |
| <b>7 Automate cu număr finit de stări</b>                         | <b>823</b> |
| 7.1 Întrebări                                                     | 823        |
| 7.2 Răspunsuri                                                    | 828        |
| <b>8 Circuite logice secvențiale particulare</b>                  | <b>829</b> |



---

|                                                                  |            |
|------------------------------------------------------------------|------------|
| 8.1 Întrebări                                                    | 829        |
| 8.2 Răspunsuri                                                   | 851        |
| <b>9 Probleme pentru testat inteligența naturală/artificială</b> | <b>853</b> |
| 9.1 Probleme de proiectare                                       | 853        |
| <b>Tomul pentru grabă mare - Final</b>                           | <b>862</b> |
| <b>Grăirea de încheiere</b>                                      | <b>863</b> |
| <b>Electronică digitală - Final</b>                              | <b>866</b> |



---

## Cuvântul cel dintâi către „cetitor”

Omul a simțit dintotdeauna dorința să își transmită învățătura către generațiile următoare.

Prima ediție **„Electronică digitală - Carte de învățatură”** a apărut în anul 2012. Subtitlul arhaic își are originea în sentimentul meu că „*învățătura*”<sup>1</sup> devine învechită, fiind înlocuită treptat de „*descurcăreala*”<sup>2</sup>.

După 3 ani, în 2015, am publicat **„Electronică digitală - Carte de învățatură 2.0”**, o ediție revizuită, în care „*m-am adaptat pietei*”. Adică, am mai îndulcit tonul, am mai coborât ștacheta și am sortat problemele pentru a realiza o secțiune minimală, dedicată „*celor ce vor doar să promoveze examenul*”.

În 2019, pentru „*cei ce vor să învețe și să devină profesioniști*” am scris volumul **„Verilog - Carte de învățatură”**.

În anul 2020, am simțit nevoia să public un set de chestionare de electronică digitală, cu subtitlul **„Electronică digitală - Carte de învățatură în regim de urgență”**. Problemele au fost adaptate examinării online, prin transformarea problemelor de proiectare și gândire în chestionare potrivite pentru „*metoda ghici*”. Am vrut să marchez astfel perioada nefastă la nivel mondial și să îmi exprim convingerea că nu se pot pregăti profesioniști de electronică *în regim de urgență*.

După 2022, observând schimbarea evidentă a electronicii în ultimii 30 de ani, am actualizat conținutul cursului de Electronică digitală, prin aducerea limbajului de descriere hardware în strânsă conexiune cu circuitele electronice digitale. Am sesizat că studenții sunt mult mai pragmatici, dornici să rezolve repede problema, și mai puțin dornici să înțeleagă justificarea. După încă 2 ani, feed-back-ul primit din partea studenților a fost că „*nu există o bibliografie adaptată cerințelor mele*”.

---

<sup>1</sup>**învățatură** = Procesul de a dobândi cunoștințe. [dannicula.ro](http://dannicula.ro)

<sup>2</sup>**descurcăreală** = Arta de a rezolva o problemă, fără a o înțelege pe deplin, eventual „*asistat de inteligență artificială*”. [dannicula.ro](http://dannicula.ro)

Înțelegând corectitudinea acestei opinii, m-am decis să rescriu cartea de învățătură în integralitatea ei, sub forma a patru „tomuri”<sup>3</sup>:

- **Tomul pentru zăbovit și gândit**<sup>4</sup> suport de curs, noțiuni teoretice alăturate modelării Verilog.
- **Tomul pentru chibzuit și buchisit**<sup>5 6</sup> suport pentru seminar și probleme de proiectare.
- **Tomul pentru meșterit și muncit**<sup>7</sup> suport pentru laborator și mici proiecte pentru practică.
- **Tomul pentru grabă mare**<sup>8</sup> chestionare pentru auto-evaluare și evidențierea noțiunilor ce mai trebuie studiate.

Subtitlurile arhaice sunt intenționate, pentru că tot mai cred că „*a învăța dintr-o carte cu creionul în mână*” este ceva învechit. Mă aștept ca studenții să citească din această carte pe un monitor mare, în care pot vedea două pagini alăturate (similar cu o carte pe hârtie, deschisă), având în față un caiet (sau o tabletă de scris) și un creion (clasic sau „pen”). Din acest motiv, în carte am preferat să pun link-uri de acces rapid în versiunea electronică și să evit să dau detalii despre anumite subiecte cu o relevanță scăzută dar ușor de descoperit pe Internet.

Versiunea actuală, 2025, aduce o structurare și o corelare a întregului subiect al electronicii digitale, oferind suport pentru curs, seminar, laborator și studiu individual pentru studenții de la specializările:

- Electronică aplicată
- Telecomunicații
- Calculatoare
- Automatică
- Tehnologia Informației

---

<sup>3</sup>**tom** = Fiecare dintre părțile unei lucrări, ale unei opere de proporții mai întinse, formând adesea o unitate independentă și purtând, de cele mai multe ori, un număr de ordine. Volum, carte, lucrare, operă. [dexonline.ro](https://dexonline.ro)

<sup>4</sup>**a zăbovi** = A lucra prea încet, a nu se grăbi. [dexonline.ro](https://dexonline.ro)

<sup>5</sup>**a chibzui** = A reflecta asupra unei situații cumpănind toate eventualitățile. A se gândi. [dexonline.ro](https://dexonline.ro)

<sup>6</sup>**a buchisi** = A citi sau a învăța ceva cu efort și cu mîgală. [dexonline.ro](https://dexonline.ro)

<sup>7</sup>**a meșteri** = A lucra, a munci (cu pricepere) pentru a realiza ceva. [dexonline.ro](https://dexonline.ro)

<sup>8</sup>**grabă mare** = Termen generat de ChatGPT ca un arhaism pentru „stare de urgență”.

Deși subtitlurile sunt arhaice, conținutul este redactat într-o formă ce s-a dovedit acceptată de generația actuală de studenți, obișnuită (și nevoită) să citească mai mult de pe monitor decât de pe hârtie: paragrafe și cuvinte **cheie** evidențiate cu galben, **avertizări** marcate cu roșu, link-uri la pagini web, recomandări ale  ChatGPT și multe încurajări... 😊. <sup>9</sup>

Asemănările cu aplicațiile de socializare actuale sunt intenționate. 😞

Conceptul cărții este de a permite studenților, la finalul anului al doilea de studii, accesarea stagiilor de pregătire oferite de firme în domeniul hardware.

Fișierele necesare pentru efectuarea lucrărilor se află pe [site-ul autorului](http://site-ul autorului), [dannicula.ro](http://dannicula.ro).



Brașov, 2025.

Dan NICULA

»> CONTINUT

---

<sup>9</sup>Da, m-am gândit și la audio-book. Și la benzi desenate... 😊



---

## Începutul învățaturii „Tomului pentru zăbovit și gândit”

**Tomul pentru zăbovit și gândit**<sup>1</sup> prezintă noțiunile teoretice ale electronicii digitale, adaptate metodologiei de proiectare a circuitelor integrate digitale actuale.

Volumul are patru capitole:

- Un capitol introductiv, care face trecerea de la noțiunile dobândite la **cursurile de matematică** (reprezentarea numerelor, algebră Booleană, grafuri, diagrame V-K) la **inginerie** (circuite electronice pentru porți logice și bistabile).
- Un capitol despre **Circuite Logice Combinationale (CLC)**, care pune accentul pe modelarea circuitelor în Verilog. Se renunță la accentuarea proiectării structurii detaliate a circuitelor în favoarea modelării comportamentale care este urmată de sinteză de hardware, realizată cu programe specifice EDA („*Electronic Design Automation*”, engl.). S-au eliminat noțiunile legate de proiectarea cu circuite SSI/MSI („*Small/Medium Scale Integration*”, engl.). Capitolul explică necesitatea minimizării funcțiilor logice, fără a insista pe algoritmi de minimizare implementați de programele de sinteză de hardware.
- Un capitol despre **Circuite Logice Secvențiale (CLS)**, care insistă pe înțelegerea funcționării și modelării circuitelor cu bistabile. Accentul este pus pe înțelegerea formelor de undă generate de către simulatoare HDL („*Hardware Description Language*”, engl.). Proiectarea se concentrează pe modelarea comportamentală în Verilog, urmată de sinteză. S-a renunțat la accentuarea strategiilor de codificare a stărilor automatelor și alte detalii preluate spre a fi rezolvate de către programele de sinteză de hardware, în favoarea modelării structurilor des utilizate în practica actuală și a descrierii sistemelor digitale mari.

---

<sup>1</sup> a zăbovi = A lucra prea încet, a nu se grăbi. [dexonline.ro](http://dexonline.ro)

- Ultimul capitol grupează **noțiuni avansate** din domeniul electronicii digitale: chestiuni legate de temporizarea sistemelor digitale, protocoale de comunicare între blocuri digitale, tehnologia CMOS de fabricare a circuitelor integrate digitale și noțiuni avansate de Verilog.

Caracteristica esențială a prezentării teoriei o constituie alăturarea modelelor HDL (Verilog) fiecărei structuri de circuite logice combinaționale sau secvențiale prezentate.

Am gândit această carte ținând cont de predilectia studenților de a citi de pe monitor. Ar fi minunată utilizarea unui ecran mare (nu laptop), cu două pagini alăturate. Textul are multiple **evidențieri** pentru a ușura focalizarea pe noțiunile importante și a da senzația de parcurgere a unui text scris pe hârtie.

Desigur, mă aștept ca textul să fie parcurs încet, cu **ză-ba-vă** și aplecare asupra gândirii și înțelegerii noțiunilor.

Volumul conține și exemple de rezolvări de probleme, pentru că sunt conștient că unii studenți nu ajung niciodată să parcurgă și al doilea volum:

**Tomul pentru chibzuit și buchisit**



Brașov, 2025.

Dan NICULA



# Capitolul 1

De la matematică la inginerie

The whiteboard contains the following content:

**Truth Table:**

| $C_i$ | A | B | $C_o$ | S |
|-------|---|---|-------|---|
| 0     | 0 | 0 | 0     | 0 |
| 0     | 0 | 1 | 0     | 1 |
| 0     | 1 | 0 | 0     | 1 |
| 0     | 1 | 1 | 1     | 0 |
| 1     | 0 | 0 | 0     | 1 |
| 1     | 0 | 1 | 1     | 0 |
| 1     | 1 | 0 | 1     | 0 |
| 1     | 1 | 1 | 1     | 1 |

**Karnaugh Maps:**

For  $S$  (Sum):

| $C_i \backslash A \backslash B$ | 00 | 01 | 11 | 10 |
|---------------------------------|----|----|----|----|
| 0                               | 0  | 1  | 1  | 0  |
| 1                               | 1  | 0  | 0  | 1  |

For  $C_o$  (Carry Out):

| $C_i \backslash A \backslash B$ | 00 | 01 | 11 | 10 |
|---------------------------------|----|----|----|----|
| 0                               | 0  | 0  | 1  | 0  |
| 1                               | 1  | 1  | 1  | 1  |

**Equations:**

$$S = A \oplus B \oplus C_i$$
$$C_o = A \cdot B + A \cdot C_i + B \cdot C_i$$

**Logic Circuit:**

The circuit diagram shows a 1-bit full adder. It uses three 2-input AND gates to implement the carry-out equation  $C_o = A \cdot B + A \cdot C_i + B \cdot C_i$ . The outputs of these AND gates are combined using a 3-input OR gate to produce  $C_o$ . The sum output  $S$  is produced by a 3-input XOR gate with inputs A, B, and  $C_i$ .

Sumator?

$1+1+1=\{1,1\}$ ? Hm...



### Predoslovie la Capitolul Întâi:

#### Unde se face trecerea, cu băgare de seamă, de la socoteală la electron

Fii bine-venit la capul acestei cărți.

Dar să nu te arunci cu capul înainte ca într-o priză descoperită, căci prima oprire pe acest drum nu este cu leduri, ci cu... **matematică**.

Așa-i, n-avem ce face — până și **ingineria**, cu toată grandoarea ei de becuri și sârme, își trage rădăcina din biata matematică, oftată de mulți, înțeleasă de puțini.

În acest capitol de deschidere, vom așeza temeliile — nu din cărămidă, ci din simboluri, logici și legi care par simple până le încurci. Vei vedea cum „unu” și „zero” ajung să hotărască soarta lumii moderne, cum adevărul și minciuna se scriu cu fire și tensiuni, și cum, din câteva reguli matematice de toată jena (la prima vedere), se pot zidi aparaturi de mare iscusință.

Așadar, nu fugi de formulă, nu te spăria de definiție, căci toate-s mai blânde decât par. Cu puțină răbdare, vei înțelege cum din socoteală se face știință, și din știință — meșteșug.

Și ține minte, om bun:

**un inginer fără matematică e ca un fierar fără ciocan — mult zgomot, dar puțină treabă.**

Cu aceasta fiind zise, să ne deșteptăm mintea și să pornim cu pas calculat pe cărarea digitală, căci de aici începe meșteșugul cel adevărat!

*Text generat cu ChatGPT.*

## 1.1 Lumea digitală. Între 0 și 1

Lumea reală este analogică! Ar fi groaznic dacă nu ar fi așa.

Și totuși, echipamentele electronice actuale sunt preponderent digitale. De ce?

Pentru că mașinile nu sunt oameni. Mașinile sunt proiectate de oameni, pentru oameni, dar nu sunt oameni. Dacă proiectezi și construiești ceva, ai nevoie să simplifici lucrurile. Trecerea de la realitatea analogică la abstractul digital a fost determinată de nevoia de a gestiona sisteme din ce în ce mai complexe. Deci, a fost nevoie de simplificare.

Codarea informației cu două stări (adevărat și fals) asociate cu două simboluri (0 și 1) ușurează stocarea și procesarea acestora.

Să vă dau un exemplu. Plouă. Analogic. Auziți știrea: „A plouat  $X$  litri pe metru pătrat”. Numărul de litri este o valoare reală. Poate fi 2 litri, 2.27 litri sau 2.27123986 litri pe metru pătrat. Însă, pentru această știre, este relevantă precizia? Nu. Cred că toți suntem suficient de mulțumiți cu valoarea întreagă, fără zecimale. „A plouat 2 litri pe metru pătrat” Am putea redacta știrea ca „a plouat mult” sau „a plouat puțin”? Sau putem spune „azi a plouat” sau „azi nu a plouat”. Desigur, poate specialiștii în meteorologie ar vrea o precizie mai mare. De acord. Câte zecimale? 2 zecimale, 4 zecimale, 10 zecimale? Dar tot putem alocă un număr finit de numere pentru a exprima cantitatea de ploaie și să satisfacem cele mai mari cereri de precizie. Nu este nevoie de un număr infinit de zecimale pentru a exprima o mărime analogică foarte precisă. Dacă ne rezumăm să spunem dacă „a plouat” sau „nu a plouat”, avem avantajul codării informației cu un singur bit. Putem transmite informația și unui câine: „a plouat” = *mâna ridicată*, „nu a plouat” = *mâna coborâtă*. Am inventat deja negația...

Un exemplu mai frumos: Câte culori și nuanțe există în natură? 🌈

Am întrebat acest lucru pe  ChatGPT și am primit răspunsul:

„Numărul exact de culori și nuanțe existente în natură este aproape imposibil de cuantificat. Natura oferă o gamă infinită de culori, iar percepția umană a acestora este limitată de capacitatea ochilor și creierului de a distinge între diferite lungimi de undă ale luminii”.

Nu aș fi putut da un răspuns mai corect și mai frumos. Bravo ChatGPT!

Deci un număr infinit de nuanțe de culori.

Sunteți mulțumiți cu numărul de culori afișate de un laptop?

Am întrebat pe ChatGPT: „Câte culori și nuanțe poate afișa un laptop”?

Am primit răspunsul:

„Un laptop poate afișa un număr limitat de culori și nuanțe, în funcție de tehnologia ecranului și de capacitățile plăcii grafice. Acestea sunt determinate de adâncimea de culoare (color depth) și de spectrul de culori pe care ecranul poate să îl redea”.

Deci, un număr finit de culori.

În concluzie, nu toate culorile din natură pot fi afișate pe un laptop.

Vă întristează și frustrează faptul că nu vedeți toate culorile pe monitor? Puteți fi fericiți chiar dacă nu vedeți toate culorile din natură pe monitor? Ați auzit vreodată că „nu folosesc laptop-ul pentru că nu văd toate culorile”? Dar, cu siguranță ați auzit că un peisaj de munte este mai frumos în natură decât pe monitor! <sup>1</sup> Putem trăi cu o simplificare. Pentru că senzorii umani (ochi, urechi) sunt imperfecti și pot lucra cu mai puțină informație. Și dacă am vrea mai multă informație recepționată de ochi și urechi, ne-ar trebui un creier mai mare pentru procesare...

Ce încerc să spun este că simplificarea există în natură, iar inginerii nu fac decât, inspirați de natură, să simplifice lucrurile mai mult pentru a le putea înțelege, proiecta, realiza și folosi. Deci, lumea între 0 și 1 este o mare simplificare. Însă, sistemele electronice și utilizatorii acestora (oamenii) acceptă aceste simplificări.

Desigur, nu încercați să aplicați prea multe simplificări dacă vreți să exprimați sentimente. În realitate, declarația „Te iubesc” are absolut același efect ca și „Nu te iubesc” dacă timbrul vocii sau emoția transmisă nu este cea potrivită. 😊

Însă, această carte este despre proiectarea sistemelor digitale, care sunt foarte fericite cu informația codată cu numai două simboluri: „Te iubesc=1”, „Nu te iubesc=0”.

### 1.1.1 De ce este nevoie de „Electronică digitală”?

Această carte este dedicată studenților de la specializările de Inginerie Electronică, Telecomunicații, Calculatoare, Automatică și Tehnologia informației.

De ce un curs de electronică digitală pentru aceste specializări? Pentru

<sup>1</sup>O mamă, dulce mamă...

că electronica digitală reprezintă esența realizării **calculatoarelor** actuale. Calculatoarele sunt digitale iar programarea calculatoarelor necesită (într-o oarecare măsură) înțelegerea electronicii digitale.

Chestiunea cu „*software sau hardware*” nu este similară cu cea cu „*oul sau găina*”. Hardware a fost înainte de software, iar software se bazează pe hardware. Există și azi „*hardware fără software*” dar nu s-a inventat încă „*software fără hardware*”. Celor ce consideră că pot fi buni programatori software fără a avea cunoștințe de hardware le răspund astfel:

„Da, puteți fi un președinte de țară sau un ministru de externe, dar dacă nu știți istorie, nu veți fi buni”. *Atenție la subliniere: „buni”.*

Indiferent de specializare, este nevoie de electronică digitală, la fel cum înainte de cursul de istorie, ar trebui să fi promovat un curs de limba română.

Cursul de electronică digitală presupune existența unor cunoștințe inițiale dobândite de la cursurile anterioare (gimnaziu, liceu, facultate). Presupune și o atenție asupra lumii înconjurătoare (organizare, procese, etc.) Disciplinele necesare pentru înțelegerea electronicii digitale sunt:

- *Comunicare*, pentru a ne putea înțelege și colabora pentru proiectarea unor sisteme complexe.
- *Limba engleză*, pentru că electronica digitală conține multe noțiuni preluate din această limbă iar domeniul utilizează limba engleză ca limbă generală de comunicație.
- *Fizică*, pentru că inginerii realizează lucruri nu doar filozofează asupra lor. Noțiunile de electricitate sunt introduse în clasa a 6-a.
- *Dispozitive electronice*, pentru că electronica digitală nu este doar matematică cu 0 și 1 este despre realizarea circuitelor cu dispozitive electronice integrate.
- *Algebră liniară, geometrie analitică și diferențială* pentru că algebra Booleană reprezintă fundamentul teoretic al electronicii digitale.
- *Matematici speciale*, pentru că pentru a desena o linie pe un ecran, indiferent de software sau hardware, este necesară înțelegerea conceptului de dreaptă și a ecuațiilor ce o determină.
- *Electrotehnică*, un curs universitar peste capitolul de fizică din liceu, pentru că porțile digitale sunt circuite electronice caracterizate de legile fizice ale electricității. Ar trebui știut că două rezistențe de valoare egală  $R$  conectate în serie sunt echivalente cu o rezistență de valoare  $2 \times R$ , dar aceleași două rezistențe conectate în paralel sunt echivalente cu o rezistență de valoare  $R/2$ .

- *Măsurări electronice*, pentru a știi că „a măsura înseamnă a compara cu un etalon” și că mărimile fizice au unități de măsură. Adică, înălțimea mea nu este 1.85. Ar putea fi  $1.85m$  sau  $18.5dm$  sau  $185cm$ , pentru că înălțimea reprezintă o dimensiune, nu un număr.
- *Programarea calculatoarelor și limbaje de programare*, pentru că majoritatea sistemelor digitale de azi sunt complexe și necesită reconfigurare după realizare, lucru care implică aspecte de programare software.
- *Semnale și sisteme*, pentru a știi că „semnalul este o mărime variabilă în timp”, iar semnalul electric este tot un semnal, caracterizat de o mărime electrică (tensiune electrică), chiar dacă îl simplificăm la două simboluri matematice, 0 și 1.

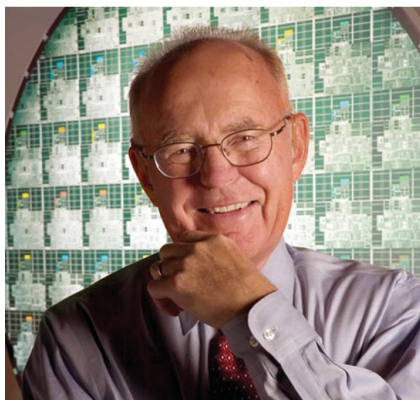
Deci, sunt multe cerințe pentru a înțelege cursul de electronică digitală. Însă, electronica digitală este obligatorie pentru înțelegerea cursurilor care urmează în pregătirea inginerească în specializările menționate.

- *Limbaje de descriere hardware*, limbajul de modelare pentru sisteme electronice integrate, înțeles de către toate programele EDA (*Electronic Design Automation*, engl.) utilizate pentru proiectarea circuitelor integrate digitale.
- *Arhitectura microprocesoarelor*, circuite integrate de procesare a datelor prezente în majoritatea echipamentelor electronice.
- *Arhitectura calculatoarelor*, proiectare de sistemele electronice de calcul complexe.
- *Tehnici de proiectare VLSI* (*Very Large Scale Integration*, engl.), Proiectarea Circuitelor Integrate.
- *Sisteme încorporate*, (*embedded systems*, engl.) sisteme de calcul electronice înglobate într-o mulțime de produse.
- *Prelucrarea digitală a semnalelor*.
- *Bazele comunicațiilor*, sisteme de telecomunicații.
- *Prelucrarea imaginilor*, procesarea numerică a datelor.
- *Grafică avansată pe calculator*.
- *Sisteme de operare*.
- *Inteligență artificială*.

Piața circuitelor integrate digitale este azi mult mai mare decât cea a circuitelor integrate analogice sau mixte (analog-digital). Sistemele electronice ce procesează semnale analogice se realizează în prezent... digital. Gândiți-vă cum se realizează un sistem audio, de la microfon la difuzor. Microfonul este un traductor care captează vocea (semnal analogic). Imediat semnalul este eșantionat și convertit în semnal digital, transmis (prin unde radio) spre o stație de bază digitală, acolo intră într-un mixer digital, se procesează alături de semnale din alte surse și, în final, se convertește în semnal analogic pentru fi redat într-un difuzor (boxă). Deci, semnalul parcurge o cale lungă de procesare, în care majoritatea timpului semnalele sunt codate digital. Și stocarea semnalului audio (înregistrarea) se face tot digital.

Domeniul circuitelor integrate a prezentat o creștere extraordinară a complexității în ultimul secol. Pentru a justifica dinamica extraordinară a domeniului, se citează foarte frecvent **Legea lui Moore**.

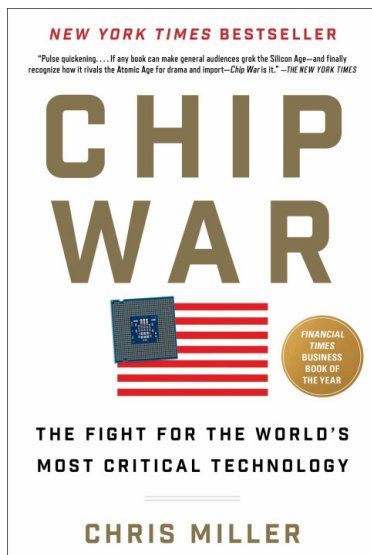
**Gordon Moore**, co-fondator al firmei Intel, a afirmat în 1965 (comparând complexitatea crescândă a microprocesoarelor) că „*numărul de componente din circuitele integrate s-a dublat anual de la invenția circuitului integrat, în 1958, până în 1965*”, adăugând estimarea că această tendință va mai continua. Există pe Internet reprezentări grafice ale exprimării acestei legi prin punctele marcate de microprocesoarele care au fost inventate de-a lungul istoriei.



**Figura 1.1** Gordon Earle Moore (1929 – 2023), inginer, fondator al firmei Intel Corporation a enunțat observația că numărul de tranzistoare dintr-un circuit integrat (IC) se dublează aproximativ la fiecare doi ani.

Și ar mai fi o motivație pentru a învăța despre electronică digitală indiferent de subdomeniul în care veți profesa. Este evident faptul că fabricarea

circuitelor integrate este din ce în ce mai scumpă. Iar cei mai mulți bani sunt în industria de război... Din păcate.



**Figura 1.2** Coperta cărții „*Chip War - The fight for the worlds's most critical technology*” (autor *Chris MILLER*) în care se explică legătură dintre industria de război și circuitele integrate.

În acest context, electronica va deveni mult mai necesară. Și pentru a înțelege dacă dispozitivele electronice importate sunt sau nu sigure din punct de vedere militar... Imaginați-vă că ajungeți să certificați că un anumit dispozitiv electronic este inofensiv pentru utilizator și nu devine o armă controlată de la distanță. Aveți nevoie de cunoștințe hardware și software. Și „*calul Troian*” are azi o implementare digitală.

Sună a „*Science Fiction*”? *Deja s-a întâmplat*. Din păcate...

### 1.1.2 Informația. Semnale

Conceptual, sistemele electronice digitale stochează, procesează și transmit informații.

Dar, **ce este informația?**

Conform **dexonline.ro**, cea mai potrivită definiție a informației este:

*Fiecare dintre elementele noi, în raport cu cunoștințele prealabile, cuprinse*



în semnificația unui simbol sau a unui grup de simboluri (text scris, mesaj vorbit, imagini plastice, indicație a unui instrument etc.).

De reținut atributul de „nou”, adică nu este o informație ceea ce deja se știe. Remarcăm din definiție și faptul că „informația este cuprinsă într-un grup de simboluri”. În electronica digitală ne vom focaliza pe aspectele **cantitative** ale informației.

Există și o latură **calitativă** de care se ocupă disciplinele umaniste. Mă refer la o comunicare plină de noutăți (știri), dar care sunt de calitate însoțită și evident fără utilitate.

De exemplu, știrea că „o găină a trecut strada pe trecerea de pietoni”. Este informație, conform definiției, pentru că este ceva nou, ce nu ați știut. Însă deținerea acestei informații și timpul petrecut cu procesarea ei nu vă aduc nimic benefic. Deci ar trebui să renunțați bucuros la a recepționa informații necalitative. Dar asta este deja o discuție care nu face obiectul electronicii digitale...

Din punct de vedere informațional, informația trebuie codată cu anumite simboluri pentru a putea fi stocată, transmisă și procesată. Electronica digitală poate ajuta aici.

Dacă informația este o mărime fizică, în ce se măsoară? Tot conform [dextonline.ro](http://dextonline.ro), unitatea de măsură a cantității de informație se numește *bit*.

Ne oprim un pic să reflectăm:

### Oare cum se măsoară calitatea informației?

Cuvântul **BIT** provine din 3 litere ale cuvintelor din limba engleză **Binary Digit**, adică „cifră binară” (în baza de numerație 2).

Electronica analogică codează informația în nivele de tensiune într-un domeniu continuu. De exemplu, orice valoare a tensiunii electrice între 0 și 5 volți.

Electronica digitală utilizează simboluri binare (0 și 1) și codează informația în biți. Dacă informația este răspunsul la o întrebare închisă (DA sau NU) este suficient un bit (DA=1, NU=0). Dacă informația este mai complexă (temperatura exterioră), atunci este necesar un cod și mai mulți biți pentru a coda informația. De exemplu, dacă dorim codarea temperaturilor între 0 și 40 de grade cu rezoluție de 1 grad, ar fi suficienți 6 biți pentru că  $2^6 \geq 40$ . Dacă dorim extinderea codului la temperaturi negative sau rezoluții de 0.1 grade, atunci trebuie să mai adăugăm biți de cod suplimentari.

**Semnalul este o mărime variabilă în timp.** De exemplu, temperatura de moment reprezintă o informație. Însă dacă monitorizăm temperatura în timp, succesiunea de temperaturi instantanee reprezintă un *semnal*. Sem-

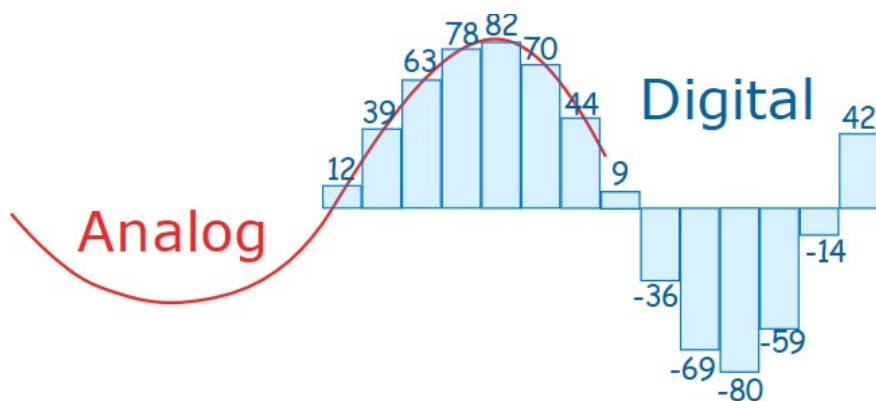
nalele pot fi *analogice* (au orice valoare, într-un anumit domeniu continuu) sau *digitale* (au doar anumite valori discrete). Antonimul noțiunii de „*semnal continuu*” (poate avea orice valoare) este „*semnal discret*” (poate avea doar anumite valori).

Altfel spus, semnalele analogice au valori continue iar semnalele digitale au valori discrete.

Am menționat însă și domeniul  *timp*. Semnalele pot fi continue în timp (adică au valori la orice moment de timp) sau discrete în timp (eșantionate, adică își modifică valoarea doar la anumite momente de timp). Pentru a vă imagina un semnal discret în timp, gândiți-vă la un ceas la care secundarul nu se învârtă cu viteză uniformă, ci sare discret din secundă în secundă.

Un semnal eșantionat își modifică valoarea doar la momentul eșantionării și și-o păstrează constantă până la următoarea eșantionare.

Figura 1.3 prezintă graficul unui semnal analogic (variație continuă în valoare și în timp) suprapus peste cel al unui semnal digital (valori discrete, eșantionat în timp).



**Figura 1.3**

Semnal analogic (variație continuă în valoare și în timp).

Semnal digital (valori discrete, eșantionat în timp).

Desigur, realitatea prezintă semnale analogice. Însă, transformarea acestora în semnale digitale determină câteva avantaje:

- *Imunitate la zgomot.* În cazul semnalelor analogice, orice mică abatere, zgomot, afectează valorile semnalului și posibila interpretare a acestuia. În cazul semnalelor digitale, se acceptă o doză de incertitudine deter-

minată de faptul că valorile sunt discrete, nu foarte apropiate unele de altele într-un spațiu abstract. De exemplu, dacă se consideră „0” = 0V și „1” = 5V, se pot accepta și tensiuni altele decât 0V și 5V dacă se face presupunea că „0” înseamnă „tensiune sub 1V” iar „1” înseamnă „tensiune mai mare de 4V”.

- *Codarea simplă, potrivită pentru operații numerice implementate hardware.* Implementarea unor algoritmi complecși necesită funcții matematice care implică numere. Este evident că valorile numerice ușurează calculele.
- *Stocarea și compactarea informației.* Semnalele trebuie stocate și restaurate într-un mod eficient. Desigur, codarea digitală facilitează atât stocarea și restaurarea precisă a informației cât și compactarea acesteia cu un consum mic de energie și un grad mare de compactare.

Figura 1.4 prezintă grafic prelucrarea unui semnal analogic pentru a deveni un semnal binar (semnal digital care poate avea doar două valori codate pe un singur bit).

Pe orizontală este reprezentată axa timpului (a).

Semnalul analogic are o variație continuă în timp și poate avea orice valori, într-un domeniu de valori (matematic, este ca o „funcție continuă”) (b).

Dacă se consideră o valoare de prag, între minim și maxim, semnalul analogic are momente când are valoare sub prag și momente când are valoare peste prag (c).

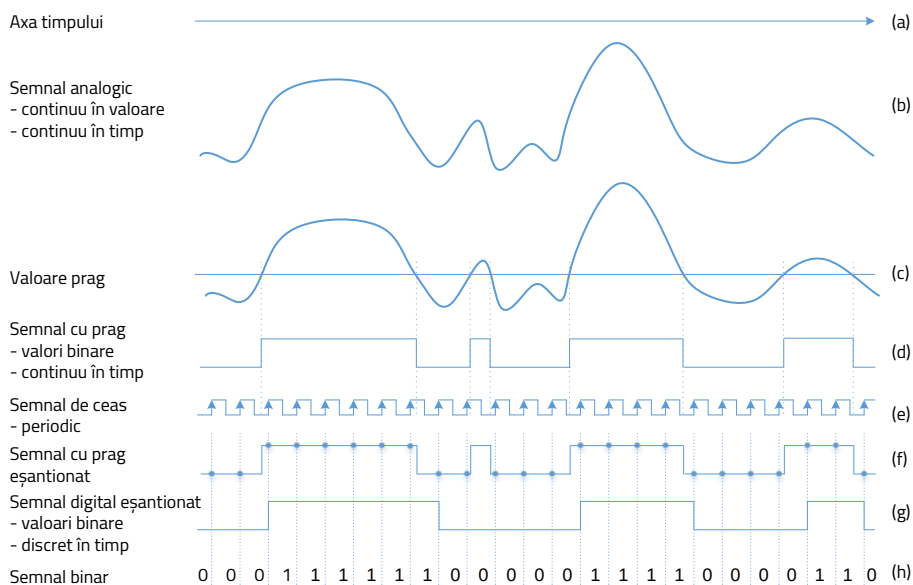
Semnalul cu prag are doar două valori (valori binare 0 sau 1) și este continuu în timp (d). Se observă că semnalul este similar cu semnalul original analogic, însă păstrează doar informația poziției valorii instantanee față de prag (mai mic sau mai mare), nu se mai poate reconstitui amplitudinea semnalului analogic.

Dacă se alătură un *semnal de ceas* (semnal periodic digital), acesta va defini momente periodice în timp (e).

Semnalul cu prag poate fi eșantionat la momentele determinate de frontul crescător al semnalului de ceas (o singură dată pe perioadă) obținându-se un semnal digital eșantionat (f), având valori binare (0 și 1) dar și discret în timp (valorile se schimbă numai o dată pe perioada semnalului de ceas) (g). Se observă că variațiile de scurtă durată (mai scurte decât perioada semnalului de ceas) pot fi omise iar momentele de comutare ale semnalului pot fi deplasate în timp, la momentele determinate de semnalul de ceas exterior.

În concluzie, semnalul digital își modifică valoarea (ori 0, ori 1) doar

la momentele periodice de timp. Putem transmite semnalul ca o succesiune de biți 0001111110000011110000110 (h). Dacă perioada semnalului de ceas este cunoscută, se poate reface semnalul digital eșantionat din succesiunea de biți.



**Figura 1.4** Transformarea unui semnal analogic într-un semnal binar.

## 1.2 Matematică binară

Matematică știe toată lumea! Sau... ar trebui să știe. Calculatoarele însă nu sunt oameni. Sunt mașini de calcul. Nu au (încă) o „*inteligență umană*”. Din acest motiv, calculatoarele trebuie instruite (învățate, programate) cu limbaje mai simple pentru ele.

Ca să predai matematica unui calculator trebuie să i-o „*explici*” simplu, pe înțelesul lui. De aceea s-a înțeles că matematica bazată doar pe două simboluri (numere binare), ar fi mai potrivită pentru o mașină de calcul. De aici a apărut noțiunea de „*matematică binară*”: înțeleasă de calculatoare și (mai greu) înțeleasă de (unii) oameni.

Dacă vreți să proiectați sau programați calculatoare, atunci este necesar să vorbiți „limba calculatoarelor”.



### Codul Domniei Tale în Lumea Firii Digitale:

#### Meșteșugul Binarului Deslușit

O, lume strălucitoare a dascăliei moderne, iată-ne ajunși în veacul de siliciu, unde mintea omului nu mai cioplește lemnul, ci codul binar!

Să nu-ți fie cu mirare, bădie, că astăzi, dacă voiești a porunci vreunui ghemotoc electronic – numit cu slavă „calculator” –, nu-i de-ajuns a-i da palme ori a-l blestema-n graiul satului. Trebuie, zic, să-nveți a-i grăi în limba lui de duhuri și scânteii, adică în matematică binară, cu nule și unuri, precum grăunțele dintr-o traistă fermecată. Căci altminteri, vai ție, te va privi cutia cu nepăsare de metal, și nu vei scoate din ea decât oftat și beznă de ecran.

Drept aceea, învață, dragă leat:

**căci fără binar, programarea e ca o horă fără lăutar!**

*Text generat cu ChatGPT.*

### 1.2.1 Reprezentarea numerelor

Există multiple formate de reprezentare a numerelor. Formatul de reprezentare potrivit depinde de tipul de numere (pozitive, întregi, fracționare), domeniul de reprezentare și operațiile realizate cu numere. De exemplu, operația de împărțire nu se poate realiza în mulțimea numerelor întregi decât dacă se acceptă erori. Adică „7 împărțit la 3 egal 2”. Dacă ne gândim la numere reale, atunci „7.0 împărțit la 3.0 egal 2.3333 sau 2.(3)”.

Dar, **infinitul nu este pentru ingineri...**

În continuare, vom considera doar modalitățile de reprezentare a numerelor utilizate în sistemele de calcul digitale (cu cifre binare) sau cele utilizate de oameni (cu cifre în baza 10).

## Reprezentarea numerelor în baza 10

Reprezentarea numerelor în baza 10 (numere zecimale) o cunoaște toată lumea (sau ar trebui), deși nu mulți știu noțiunea de „bază de numerație”. Totuși, de mici, ne numărăm degetele și, de multe ori, ajungem la un rezultat corect: 10.

Prima propoziție învățată la limba română este:

„Ana are mere.” 🍏 🍏 🍏

Simultan, la lecția de matematică ni se cere să-i numărăm merele: 1, 2, 3, ...

Dacă Ana are mai mult de 9 mere, toți copii sunt mulțumiți să continue numărarea cu două cifre: 10, 11, 12, ...

Implicit, se presupune o numărare cu 10 simboluri (0, 1, 2, ..., 9) adică „în baza 10”.

Până aici, toate bune. Dar calculatorul nu are (nici măcar) mintea unui copil. Să nu intrăm în subiectul „AI”. <sup>2</sup>

În liceu, dacă ați fost atenți, v-au fost prezentate la matematică „alte baze de numerație”. De exemplu, baza 5 are 5 simboluri, notate 0, 1, 2, 3, 4. Cinci se scrie cu două simboluri:  $5|_{10} = 10|_5$ . Expresia matematică de mai sus o citim: „cinci în baza zece este egal cu unu zero în baza cinci”. Adică:  $10|_5 = 1 \cdot 5^1 + 0 \cdot 5^0 = 5|_{10}$

Iar numărul 7 se scrie în baza 5 cu două simboluri ca  $12|_5$ . Adică „șapte în baza zece este egal cu unu doi în baza cinci”:

$$12|_5 = 1 \cdot 5^1 + 2 \cdot 5^0 = 7|_{10}$$

Să acceptăm că toată lumea înțelege ca suntem în anul „2025”. Matematicienii, justifică astfel:

$$2025|_{10} = 2 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0$$

Adică 2025 este scrierea alăturată a numerelor cu care se multiplică puterile bazei de numerație 10.

Nu vă așteptați să fiți înțeleși dacă spuneți că „suntem în anul 31100” chiar dacă menționați imediat că socotiți „în baza 5”. Adică:

$$31100|_5 = 3 \cdot 5^4 + 1 \cdot 5^3 + 1 \cdot 5^2 + 0 \cdot 5^1 + 0 \cdot 5^0 = 2025|_{10}$$

Dar, dacă spuneți că „suntem în anul 31100” gândind „în baza 7” atunci matematicienii vor înțelege anul 7597, justificând:

<sup>2</sup>Inteligență Artificială („Artificial Intelligence”, engl.)

$31100|_7 = 3 \cdot 7^4 + 1 \cdot 7^3 + 1 \cdot 7^2 + 0 \cdot 7^1 + 0 \cdot 7^0 = 7597|_{10}$ , dar ceilalți prieteni vor fi plecat de mult...

Ce am demonstrat cu asta? Că, pentru a fi înțeleasă valoarea unui număr, trebuie cunoscută baza de numerație în care este reprezentat numărul. Oamenii presupun implicit „baza 10”. În calculatoare, numerele sunt reprezentate în „baza 2”. Se pot face conversii între numerele reprezentate în diverse baze de numerație, dar nu este ușor pentru toți...

Matematic, pentru reprezentarea numerelor în baza 10, cu  $N$  cifre, formula este:

$$z_{N-1} \dots z_1 z_0 = \sum_{i=0}^{N-1} (z_i \cdot 10^i)$$

S-a notat cu  $z_i$  cifra zecimală cu index  $i$  a numărului considerat de  $N$  cifre.

Se remarcă faptul că s-a notat indexul cifrelor începând cu zero:

$$0, 1, \dots, N - 1$$

**M-ai pierdut?** Vestea bună este că teoria bazelor de numerație nu este necesară aici. O singură bază de numerație (diferită de 10) are o aplicabilitate practică: **baza 2**, cea „înțeleasă” de calculatoare și utilizată în toate sistemele electronice de calcul actuale.

Deci, dacă ai citit până aici, vă îndemn să continuați.



De acum începe frumusețea...

## Reprezentarea numerelor în baza 2

În sistemele de calcul, numerele sunt reprezentate în baza de numerație 2 (numere binare) pentru că există unele avantaje legate de stocarea și procesarea numerelor reprezentate cu numai două simboluri. În baza de numerație 2, numerele se codifică cu două simboluri, denumite 0 și 1.

Matematic, formula este:

$$b_{N-1} \dots b_1 b_0 = \sum_{i=0}^{N-1} (b_i \cdot 2^i)$$

S-a notat cu  $b_i$  cifra binară cu index  $i$  a numărului considerat de  $N$  cifre.

Dacă spuneți că suntem în anul „111\_1110\_1001” poate vă mai înțelege vreun coleg mai ager (dacă există unul mai ager ca voi 😊), dar cu siguranță vă va înțelege calculatorul, care gândește în baza 2:

$$111\_1110\_1001|_2 = 1 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 2025|_{10}$$

În această scriere caracterul de subliniere \_ are numai un rol de delimitator a 4 caractere binare (la fel cum se scrie 2.025 cu un punct la fiecare 3 caractere zecimale).

**„Codarea binară e ca *horinca*: grea la început, dar te încălzește când pricepi rostul ei.”**

*Text generat cu ChatGPT.*

## Reprezentarea numerelor în baza 16

Reprezentarea numerelor în baza 16 (numere hexazecimale) necesită 16 simboluri: 10 cifre (0 – 9) și 6 litere (*A, B, C, D, E, F*, minuscule sau majuscule).

De remarcat că celelalte litere *G, H, I, ...* nu au nicio valoare numerică asociată în baza 16. Și nici nu este vreo confuzie între litera *O* și cifra 0 (nici scrise de mână, nici scrise la tastatură), pentru că litera *O* nu are nicio semnificație în baza 16.

| Baza 10 | Baza 16 | Baza 2 |
|---------|---------|--------|
| 0       | 0       | 0000   |
| 1       | 1       | 0001   |
| 2       | 2       | 0010   |
| 3       | 3       | 0011   |
| 4       | 4       | 0100   |
| 5       | 5       | 0101   |
| 6       | 6       | 0110   |
| 7       | 7       | 0111   |
| 8       | 8       | 1000   |
| 9       | 9       | 1001   |
| 10      | A       | 1010   |
| 11      | B       | 1011   |
| 12      | C       | 1100   |
| 13      | D       | 1101   |
| 14      | E       | 1110   |
| 15      | F       | 1111   |



Matematic, formula este:

$$h_{N-1}...h_1h_0 = \sum_{i=0}^{N-1} (h_i \cdot 16^i)$$

S-a notat cu  $h_i$  cifra hexazecimală cu index  $i$  a numărului considerat de  $N$  cifre.

### Bytes, LSB, MSB, Multiplii

O cifră binară, 0 sau 1, este denumită „*bit*” („*Binary digIT*”, engl.).

Un bit se prescurtează în scris cu litera „*b*” minusculă.

Un grup de 8 biți este denumit „*byte*”. Un byte se prescurtează în scris cu litera „*B*” majusculă.

$1B = 8b$  se citește „*un byte este egal cu 8 biți*”.

1b poate avea  $2^1 = 2$  valori (0 și 1).

1B poate avea  $2^8 = 256$  de valori (0000\_0000 ... 1111\_1111).

Un grup de 4 biți se numește „*nibble*”.

În cadrul unui șir de biți, cel mai puțin semnificativ bit (cu ponderea cea mai mică în valoarea reprezentată) se numește „*lsb*” („*least significant bit*”, engl.), iar cel mai semnificativ bit (cu ponderea cea mai mare în valoarea reprezentată) se numește „*msb*” („*most significant bit*”, engl.). Este recomandat să nu se utilizeze expresiile „*bitul din dreapta*” sau „*bitul din stânga*” ci „*bitul lsb*” sau „*bitul msb*” pentru că nu întotdeauna bitul mai puțin semnificativ este în partea dreaptă a șirului de biți.

Similar, în cazul unor mărimi reprezentate pe mai mult de 8 biți, dacă se consideră șirul împărțit în bytes, cel mai puțin semnificativ byte (cu ponderea cea mai mică în valoarea reprezentată) se numește „*LSB*” („*Least Significant Byte*”, engl.). iar cel mai semnificativ byte (cu ponderea cea mai mare în valoarea reprezentată) se numește „*MSB*” („*Most Significant Byte*”, engl.).

Pentru a exprima cantități mari de date (biți) este nevoie de a defini multiplii ai bitului. Desigur, nu există submultiplii ai bitului.

$$2^0 = 1b$$

$$2^1 = 2b$$

$$2^2 = 4b$$

$$2^3 = 8b = 1B$$

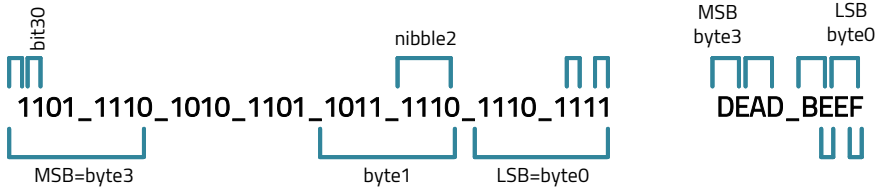
$$2^4 = 16b = 2B$$

$$2^{11} = 2.048b = 2Kb$$

$$2^{12} = 4.096b = 4Kb$$

$$2^{13} = 8.192b = 8Kb = 1KB$$

$$2^{14} = 16.384b = 16Kb = 2KB$$



**Figura 1.5** Definirea noțiunilor de „bit”, „byte”, „nibble”, „MSB”, „LSB”, „msb”, „lsb” pentru un șir de date binare și hexazecimale.

|                                      |                                           |
|--------------------------------------|-------------------------------------------|
| $2^5 = 32\ b = 4\ B$                 | $2^{15} = 32.768\ b = 32\ Kb = 4\ KB$     |
| $2^6 = 64\ b = 8\ B$                 | $2^{16} = 65.536\ b = 64\ Kb = 8\ KB$     |
| $2^7 = 128\ b = 16\ B$               | $2^{17} = 131.072\ b = 128\ Kb = 16\ KB$  |
| $2^8 = 256\ b = 32\ B$               | $2^{18} = 262.144\ b = 256\ Kb = 32\ KB$  |
| $2^9 = 512\ b = 64\ B$               | $2^{19} = 524.288\ b = 512\ Kb = 64\ KB$  |
| $2^{10} = 1.024\ b = 128\ B = 1\ kb$ | $2^{20} = 1.048.576\ b = 1\ Mb = 128\ KB$ |

|                                                    |                      |
|----------------------------------------------------|----------------------|
| $1\ b$                                             | $1\ bit$             |
| $1\ B = 2^3\ b = 8\ b$                             | $1\ Byte$            |
| $1\ Kb = 2^{10}\ b = 1.024\ b$                     | $\approx 10^3\ b$    |
| $1\ KB = 2^{10} \cdot 8\ b = 8.192\ b$             | $\approx 10^3\ B$    |
| $1\ Mb = 2^{20}\ b = 1.048.576\ b$                 | $\approx 10^6\ b$    |
| $1\ MB = 2^{20} \cdot 8\ b = 8.388.608\ b$         | $\approx 10^6\ B$    |
| $1\ Gb = 2^{30}\ b = 1.073.741.824\ b$             | $\approx 10^9\ b$    |
| $1\ GB = 2^{30} \cdot 8\ b = 8.589.934.592\ b$     | $\approx 10^9\ B$    |
| $1\ Tb = 2^{40}\ b = 1.099.511.627.776\ b$         | $\approx 10^{12}\ b$ |
| $1\ TB = 2^{40} \cdot 8\ b = 8.796.093.022.208\ b$ | $\approx 10^{12}\ B$ |
|                                                    | $1\ kilo\ bit$       |
|                                                    | $1\ kilo\ Byte$      |
|                                                    | $1\ mega\ bit$       |
|                                                    | $1\ mega\ Byte$      |
|                                                    | $1\ giga\ bit$       |
|                                                    | $1\ giga\ Byte$      |
|                                                    | $1\ tera\ bit$       |
|                                                    | $1\ tera\ Byte$      |

## 1.2.2 Conversia numerelor între bazele de numerație 2, 10, 16

Numerele reprezentate în diferite baze de numerație pot fi convertite dintr-o bază de numerație în alta.

### Conversie din baza 2 în baza 16

Cifrele binare se grupează câte 4 (pentru că  $2^4 = 16$ ) de la cele mai puțin semnificative spre cele mai semnificative (din dreapta spre stânga). Dacă numărul de cifre binare nu este multiplu de 4, se pot considera zerouri suplimentare în fața celor mai semnificative (partea stânga). Fiecare grup de 4 cifre binare generează o cifră în baza 16.

#### Exemplu

$$11010111|_2 = 1101\_0111|_2 = D7|_{16}, \text{ deoarece } 1101|_2 = D|_{16} \text{ și } 0111|_2 = 7|_{16}$$

$$1010101111001101|_2 = 1010\_1011\_1100\_1101|_2 = ABCD|_{16}$$

$$10110101101|_2 = 101\_1010\_1101|_2 = 0101\_1010\_1101|_2 = 5AD|_{16}$$

$$1110110101100|_2 = 1\_1101\_1010\_1100|_2 = 0001\_1101\_1010\_1100|_2 = 1DAC|_{16}$$

### Conversie din baza 16 în baza 2

Fiecare cifră hexazecimală generează 4 cifre binare.

#### Exemplu

$$6ABC|_{16} = 0110\_1010\_1011\_1100|_2, \text{ deoarece } 6|_{16} = 0110|_2, A|_{16} = 1010|_2, \\ B|_{16} = 1011|_2 \text{ și } C|_{16} = 1100|_2$$

$$ABCD|_{16} = 1010\_1011\_1100\_1101|_2$$

$$BEBE|_{16} = 1011\_1110\_1011\_1110|_2$$

$$B0DE|_{16} = 1011\_0000\_1101\_1110|_2$$

$$DAC1|_{16} = 1101\_1010\_1100\_0001|_2$$

#### Atenție!

0 este „zero” =  $0|_{16}$ , nu litera *O*

1 este „unu” =  $1|_{16}$ , nu litera *I*

### Conversie din baza 2 în baza 10

Numărul în baza 10 se determină prin adunarea ponderată a puterilor lui 2. Ponderea elementului  $2^i$  în suma finală este reprezentată de bitul corespunzător  $b_i$ . Altfel zis, dacă  $b_i = 1$ , la sumă se adaugă  $2^i$ , altfel nu.

#### Exemplu

$$101\_1001|_2 = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 0 + 16 + 8 + 0 + 0 + 1 = \\ = 89|_{10}$$

O variantă mai rapidă constă în însumarea puterilor lui 2 ale căror

indecși apar în reprezentarea binară. Este necesară identificarea rapidă a indecșilor cifrelor binare și cunoașterea valorilor puterilor lui 2.

*Exemplu*

$$1011001|_2 = 2^6 + 2^4 + 2^3 + 2^0 = 64 + 16 + 8 + 1 = 89|_{10}$$

O alternativă mai rapidă și mai puțin riscantă pentru conversia numerelor mari este de a se converti numărul din baza 2 în baza 16 și apoi din baza 16 în baza 10.

$$16^0 = 1, 16^1 = 16, 16^2 = 256, 16^3 = 4096.$$

*Exemplu*

$$1101\_0111|_2 = D7|_{16} = 13 \cdot 16^1 + 7 \cdot 16^0 = 13 \cdot 16 + 7 = 215|_{10}$$

$$1\_1101\_1010\_1100|_2 = 1DAC|_{16} = 1 \cdot 16^3 + 13 \cdot 16^2 + 10 \cdot 16^1 + 12 \cdot 16^0 = 1 \cdot 4096 + 13 \cdot 256 + 10 \cdot 16 + 12 = 4537|_{10}$$

### Conversie din baza 10 în baza 2

Metoda *scăderii ponderate* este o metodă iterativă care presupune listarea puterilor lui 2, mai mici decât numărul considerat. Iterațiile încep cu considerarea puterii lui 2 celei mai mari (mai mică decât numărul). Se scade numărul putere a lui 2 din numărul considerat. Cu restul obținut se repetă scăderea următoarei puteri ale lui 2 (mai mică decât cea anterioară). Dacă restul este mai mic decât puterea lui 2, se consideră în forma binară un 0 (zero). Altfel, se consideră un 1 și se obține un nou rest prin scăderea puterii lui 2 din valoarea restului. Algoritmul se termină la cea mai mică putere a lui 2 ( $2^0 = 1$ ).

*Exemplu*

$$130|_{10} = ?|_2.$$

- Cea mai mare putere a lui 2, mai mică sau egală cu 130 este  $2^7 = 128$ . Se consideră  $2^7$ .
- $130 - 128 = 2$ . Cea mai mare putere a lui 2, mai mică sau egală cu 2 este  $2^1 = 2$ . Se consideră  $2^0$ .
- $2 - 2 = 0$ . S-a terminat algoritmul.

$$\begin{array}{r|l} 130 & \\ 128 & 7 \\ \hline & \\ 2 & \\ 2 & 1 \end{array}$$

----

0 S-a terminat algoritmul

$$130|_{10} = 128 + 2 = 1 \cdot 2^7 + 1 \cdot 2^1 = 1000\_0010|_2.$$

### Exemplu

$$1999|_{10} = ?|_2.$$

- Cea mai mare putere a lui 2, mai mică sau egală cu 1999 este  $2^{10} = 1024$ . Se consideră  $2^{10}$ .
- $1999-1024=975$ . Cea mai mare putere a lui 2, mai mică sau egală cu 975 este  $2^9 = 512$ . Se consideră  $2^9$ .
- $975-512=463$ . Cea mai mare putere a lui 2, mai mică sau egală cu 463 este  $2^8 = 256$ . Se consideră  $2^8$ .
- $463-256=207$ . Cea mai mare putere a lui 2, mai mică sau egală cu 207 este  $2^7 = 128$ . Se consideră  $2^7$ .
- $207-128=79$ . Cea mai mare putere a lui 2, mai mică sau egală cu 79 este  $2^6 = 64$ . Se consideră  $2^6$ .
- $79-64=15$ . Cea mai mare putere a lui 2, mai mică sau egală cu 15 este  $2^3 = 8$ . Se consideră  $2^3$ .
- $15-8=7$ . Cea mai mare putere a lui 2, mai mică sau egală cu 7 este  $2^2 = 4$ . Se consideră  $2^2$ .
- $7-4=3$ . Cea mai mare putere a lui 2, mai mică sau egală cu 3 este  $2^1 = 2$ . Se consideră  $2^1$ .
- $3-2=1$ . Cea mai mare putere a lui 2, mai mică sau egală cu 0 este  $2^0 = 1$ . Se consideră  $2^0$ .
- $1-1=0$ . S-a terminat algoritmul.

$$\begin{array}{r}
 1999-| \\
 1024 \quad | \quad 10 \\
 ---- \\
 975-| \\
 512 \quad | \quad 9 \\
 ---- \\
 463-| \\
 256 \quad | \quad 8 \\
 ----
 \end{array}$$

```

207-|
128 | 7
----
 79-|
 64 | 6
----
 15-|
  8 | 3
----
  7-|
  4 | 2
----
  3-|
  2 | 1
----
  1-|
  1 | 0
----
  0  S-a terminat algoritmul

```

$1999|_{10} = 1024 + 512 + 256 + 128 + 64 + 8 + 4 + 2 + 1 = 1 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 111\_1100\_1111|_2$ .

O alternativă constă în *împărțirea succesivă a numărului zecimal la 2 și considerarea resturilor* (primul rest obținut reprezintă bitul cel mai puțin semnificativ, din dreapta).

### Exemplu

$130|_{10} = ?|_2$

|         |                           |
|---------|---------------------------|
| 130   0 | 130 : 2 = 65 rest 0       |
| 65   1  | 65 : 2 = 32 rest 1        |
| 32   0  | 32 : 2 = 16 rest 0        |
| 16   0  | 16 : 2 = 8 rest 0         |
| 8   0   | 8 : 2 = 4 rest 0          |
| 4   0   | 4 : 2 = 2 rest 0          |
| 2   0   | 2 : 2 = 1 rest 0          |
| 1   1   | 1 : 2 = 0 rest 1          |
| 0       | 0 S-a terminat algoritmul |

$130|_{10} = 1000\_0010|_2$ , resturile considerate, cu primul rest lsb și ultimul rest msb.

### Exemplu

$1999|_{10} = ?|_2$

|          |                           |
|----------|---------------------------|
| 1999   1 | 1999 : 2 = 999 rest 1     |
| 999   1  | 999 : 2 = 499 rest 1      |
| 499   1  | 499 : 2 = 249 rest 1      |
| 249   1  | 249 : 2 = 124 rest 1      |
| 124   0  | 124 : 2 = 62 rest 0       |
| 62   0   | 62 : 2 = 31 rest 0        |
| 31   1   | 31 : 2 = 15 rest 1        |
| 15   1   | 15 : 2 = 7 rest 1         |
| 7   1    | 7 : 2 = 3 rest 1          |
| 3   1    | 3 : 2 = 1 rest 1          |
| 1   1    | 1 : 2 = 0 rest 1          |
| 0        | 0 S-a terminat algoritmul |

$1999|_{10} = 111\_1100\_1111|_2$ , resturile considerate, cu primul rest lsb și ultimul rest msb.

Se poate face o conversie indirectă, din baza 10 în baza 16 și apoi din baza 16 în baza 2.

### Conversie din baza 10 în baza 16

O metodă constă în împărțirea succesivă a numărului zecimal la 16 și considerarea resturilor. Primul rest obținut reprezintă cifra cea mai puțin semnificativă a numărului în baza 16, cifra din dreapta.

#### Exemplu

$$130|_{10} = ?|_{16}$$

|         |                 |
|---------|-----------------|
| 130   8 | 130:16=8 rest 2 |
| 128     |                 |

-----  
2

|       |                                               |
|-------|-----------------------------------------------|
| 8   0 | 8:16=0 rest 8, cât=0, s-a terminat algoritmul |
| 0     |                                               |

-----  
8 |

$130|_{10} = 82|_{16}$ , resturile considerate cu cel mai puțin semnificativ primul obținut

#### Exemplu

$$1999|_{10} = ?|_2$$

|            |                              |
|------------|------------------------------|
| 1999   124 | 1999:16=124 rest 15 (F hexa) |
|------------|------------------------------|

---

```

1984 |
-----
    15

124 | 7      124:16=7 rest 12 (C hexa)
112 |
-----
    12

  7 | 0      7:16=0 rest 7, cât=0, s-a terminat algoritmul
  0 |
-----
    7

```

$1999|_{10} = 7CF|_{16}$ , resturile considerate cu cel mai puțin semnificativ primul obținut

### Conversie din baza 16 în baza 10

Numărul în baza 10 se determină prin suma ponderată a puterilor lui 16. Ponderele termenului  $16^i$  este cifra corespunzătoare  $h_i = 1$ .

#### Exemplu

$$BEEF|_{16} = 11 \cdot 16^3 + 14 \cdot 16^2 + 14 \cdot 16^1 + 15 \cdot 16^0 = 11 \cdot 4096 + 14 \cdot 256 + 14 \cdot 16 + 15 = 48879|_{10}$$

$$2025|_{16} = 2 \cdot 16^3 + 2 \cdot 16^1 + 5 \cdot 16^0 = 2 \cdot 4096 + 2 \cdot 16 + 5 = 8229|_{10}$$

### 1.2.3 Reprezentarea numerelor întregi în complement față de 2

Stocarea informațiilor în sisteme de calcul necesită codificarea acestora cu coduri binare, pe baza simbolurilor 0 și 1.

Numerele pozitive sunt codificate prin reprezentarea acestora în baza 2.

Numerele întregi (cu semn) pot fi codificate în diverse forme, însă cea mai răspândită este reprezentarea în **complement față de 2**.

Această codificare s-a impus datorită avantajelor acesteia în implementarea hardware a sistemelor de calcul digitale.

Operațiile aritmetice în baza 2 sunt similare operațiilor în baza 10, par mai grele pentru operatorul uman, însă sunt mai ușor de implementat în sisteme de calcul hardware digitale.



Semnul numerelor întregi (pozitive sau negative) este și el codat binar (de obicei, **0 = număr pozitiv, 1 = număr negativ**). Bitul de semn se adaugă la combinația binară înainte de cel mai semnificativ bit. Altfel spus, în reprezentările în complement față de 2, bitul cel mai semnificativ este bitul de semn. Operațiile aritmetice binare implementate hardware trebuie să țină cont de semnul operanzilor și să îl proceseze corect. În reprezentarea numerelor întregi, primul (cel mai semnificativ) bit are semnificația de „semn” al numărului întreg.

Reprezentarea în complement față de 2 a numerelor negative se realizează cu următorul algoritm:

- Se pornește de la reprezentarea binară a modulului numărului considerat (valoare pozitivă).
- Se neagă bit cu bit reprezentarea binară obținută.
- La combinația binară rezultată se adaugă 1, cu considerarea transportului. Bitul de semn se va obține direct din operație și va fi 1.

De remarcat faptul că, pentru determinarea numărului pozitiv asociat unui număr negativ, se folosește exact același algoritm: negare bit cu bit și adunare (nu scădere) cu 1.

La calcule trebuie avut în vedere domeniul de reprezentare al numerelor întregi.

În cazul numerelor pozitive, pe  $N$  biți pot fi reprezentate  $2^N$  numere pozitive, în domeniul  $[0 \dots (2^N - 1)]$ .

În cazul numerelor întregi, pe  $N$  biți pot fi reprezentate  $2^N - 1$  numere întregi, în domeniul  $[-(2^{N-1} - 1) \dots 0 \dots + (2^{N-1} - 1)]$ . Adică  $(2^{N-1} - 1)$  numere negative, 0 și  $(2^{N-1} - 1)$  numere pozitive.

Tabelul 1.1 prezintă numerele întregi reprezentate pe  $N = 4$  biți, în complement față de 2.

Se observă câteva proprietăți ale codurilor în complement față de 2:

- Cel mai semnificativ bit (de semn) este 0 pentru numere pozitive și 1 pentru numere negative.
- Zero are o singură reprezentare, cu toți biții egali cu 0.
- Indiferent de numărul de biți pe care este reprezentat, numărul  $-1$  are toți biții egali cu 1.
- Numerele pare au cel mai puțin semnificativ bit egal cu 0, indiferent de semn. Numerele impare au cel mai puțin semnificativ bit egal cu 1, indiferent de semn.

| Baza 10 | Complement față de 2 |
|---------|----------------------|
| 7       | 0111                 |
| 6       | 0110                 |
| 5       | 0101                 |
| 4       | 0100                 |
| 3       | 0011                 |
| 2       | 0010                 |
| 1       | 0001                 |
| 0       | 0000                 |
| -1      | 1111                 |
| -2      | 1110                 |
| -3      | 1101                 |
| -4      | 1100                 |
| -5      | 1011                 |
| -6      | 1010                 |
| -7      | 1001                 |

Tabelul 1.1

Reprezentarea numerelor în complement față de 2, pe 4 biți.

- Cel mai mare număr pozitiv are primul bit egal cu 0 (bitul de semn) și toți ceilalți biți egali cu 1. Cel mai mic număr negativ are primul bit egal cu 1 (bitul de semn) și toți ceilalți biți egali cu 0, cu excepția celui mai puțin semnificativ.

Un algoritm alternativ pentru determinarea mai rapidă a numerelor negative asociate numerelor pozitive și invers este următorul:

- Se pornește de la reprezentarea binară a numărului considerat (valoare pozitivă), considerat bit cu bit, din dreapta în stânga (începând cu bitul cel mai puțin semnificativ, spre bitul de semn).
- Se copiază biții până la primul 1, inclusiv.
- După primul 1, se continuă considerarea biților dar, ulterior, fiecare bit (inclusiv cel de semn) se complementează. Bitul de semn se va obține direct din operație și va fi 1.

Algoritmul expus poate fi folosit atât pentru determinarea numărului negativ pornind de la reprezentarea binară a numărului pozitiv, cât și pentru determinarea numărului pozitiv asociat unei reprezentări în complement față de 2 a unui număr negativ.

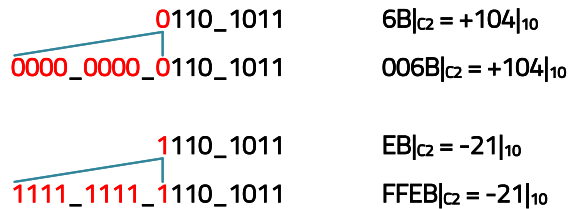
Pe un număr fix de biți se pot reprezenta un număr finit de numere. În cazul unor operații aritmetice, este posibil ca rezultatul să nu mai poată fi reprezentat pe numărul de biți ai operanzilor. În acest caz, este necesară

**extinderea de semn** a operandilor. Creșterea numărului de biți pe care este reprezentat un număr codat în complement față de 2 se face prin extinderea bitului de semn, aflat pe poziția cea mai semnificativă (din stânga) pe toate pozițiile superioare ale formatului cu mai mulți biți de cod.

Exemple de codare în complement față de 2, cu număr minim de biți, cu 8 biți și cu 16 biți:

$$\begin{aligned} 10|_{10} &= 1010|_2 = 01010|_{C2} = 0000\_1010|_{C2} = 0000\_0000\_0000\_1010|_{C2} \\ -10|_{10} &= 0\_1010|_2 + 1 = 1\_0101 + 1 = 1\_0110|_{C2} = 1111\_0110|_{C2} = \\ &1111\_1111\_1111\_0110|_{C2} \end{aligned}$$

Figura 1.6 prezintă extinderea de semn de la 8 la 16 biți pentru două numere (pozitiv și negativ).



**Figura 1.6** Extinderea de semn de la 8 la 16 biți pentru două numere (pozitiv și negativ).

#### 1.2.4 Adunarea și scăderea numerelor întregi reprezentate în complement față de 2

Adunarea numerelor reprezentate în complement față de 2 poate fi ușor implementată în hardware deoarece în această codificare semnul numerelor nu necesită o procesare diferită față de celelalte cifre binare.

Adunarea numerelor reprezentate în complement față de 2 (fie pozitive, fie negative) se face similar cu adunarea numerelor binare (similar cu cea a numerelor exprimate în baza 10): de la cea mai puțin semnificativă cifră (din dreapta) spre cea mai semnificativă (din stânga), cu considerarea transportului spre bitul superior în caz că suma a 2 biți (și transportul de la bitul inferior) este mai mare sau egală cu 2 (baza de numerație).

Scăderea numerelor reprezentate în complement față de 2 (fie pozitive, fie negative) se face ca o adunare cu cel de al doilea operand complementat (pozitiv/negativ) pe baza relației:  $A - B = A + (-B)$ .

*Exemplu*

Operații aritmetice cu numere codificate în complement față de 2, pe 8 biți.

Considerăm operandii 10, 20, 30 reprezentați pe 8 biți. Codarea acestora este:

- $10|_{10} = 1010|_2 = 0000\_1010|_{C2}$
- $20|_{10} = 10100|_2 = 0001\_0100|_{C2}$
- $30|_{10} = 11110|_2 = 0001\_1110|_{C2}$
- $-10|_{C2} = \overline{0000\_1010|_2} + 1 = 1111\_0101 + 1 = 1111\_0110|_{C2}$
- $-20|_{C2} = \overline{0001\_0100|_2} + 1 = 1110\_1011 + 1 = 1110\_1100|_{C2}$
- $-30|_{10} = \overline{0001\_1110|_2} + 1 = 1110\_0001 + 1 = 1110\_0010|_{C2}$

Rezultă operațiile:

- |                |     |            |
|----------------|-----|------------|
|                | 10+ | 0000_1010+ |
|                | 20  | 0001_0100  |
| • $10+20 = 30$ | --  | -----      |
|                | 30  | 0001_1110  |
- 
- |                              |     |            |
|------------------------------|-----|------------|
|                              | 10+ | 0000_1010+ |
|                              | -20 | 1110_1100  |
| • $10-20 = 10+(-20) = (-10)$ | --- | -----      |
|                              | -10 | 1111_0110  |
- 
- |                   |      |            |
|-------------------|------|------------|
|                   | -10+ | 1111_0110+ |
|                   | 20   | 0001_0100  |
| • $(-10)+20 = 10$ | ---  | -----      |
|                   | 10   | 0000_1010  |
- 
- |                                |      |            |
|--------------------------------|------|------------|
|                                | -10+ | 1111_0110+ |
|                                | -20  | 1110_1100  |
| • $-10-20 = (-10)+(-20) = -30$ | ---  | -----      |
|                                | -30  | 1110_0010  |

Se remarcă faptul că rezultatul, inclusiv cu semnul corect, se obține prin adunare bit cu bit de la *lsb* spre *msb*, cu considerarea bitului de transport. Eventualul transport generat de bitul *msb* se ignoră (vezi calculul bitului 7, *msb*, în cazul  $-10 - 20 = -30$ ).

### 1.2.5 Codul ASCII

Caracterele alfa-numerice sunt codate în **codul ASCII** (*American Standard Code for Information Interchange*, engl.). Codul ASCII asociază fiecărui caracter (tipăribil sau de control) un cod binar de 7 biți. La momentul inventării codării, codurile erau suficiente pentru caracterele prezente în limba engleză. Ulterior, codurile ASCII au fost extinse cu al 8-lea bit. Tabelele de coduri ASCII extinse (pe 8 biți) sunt diverse, pentru a facilita codarea caracterelor specifice diverselor limbi. Și mai târziu, codurile ASCII au fost extinse la 2 bytes. Să fie loc și pentru diacriticele românești...

Desigur, nu trebuie memorate codurile ASCII. Însă cunoașterea modului de codare a caracterelor în tabela ASCII vă permite să rezolvați rapid diverse probleme asociate sistemelor digitale.

#### Exemplu

Asocierea unui nume cu un cod ASCII binar.

Să presupunem numele: *Ion Stelescu*. Fiecare literă poate fi codată în cod ASCII și exprimată în hexazecimal sau binar, după cum urmează:

|              |                                             |
|--------------|---------------------------------------------|
| <i>I</i>     | = 49 <sub>16</sub> = 0100_1001 <sub>2</sub> |
| <i>o</i>     | = 6F <sub>16</sub> = 0110_1111 <sub>2</sub> |
| <i>n</i>     | = 6E <sub>16</sub> = 0110_1110 <sub>2</sub> |
| <i>space</i> | = 20 <sub>16</sub> = 0010_0000 <sub>2</sub> |
| <i>S</i>     | = 53 <sub>16</sub> = 0101_0011 <sub>2</sub> |
| <i>t</i>     | = 74 <sub>16</sub> = 0111_0100 <sub>2</sub> |
| <i>e</i>     | = 65 <sub>16</sub> = 0110_0101 <sub>2</sub> |
| <i>l</i>     | = 6C <sub>16</sub> = 0110_1100 <sub>2</sub> |
| <i>e</i>     | = 65 <sub>16</sub> = 0110_0101 <sub>2</sub> |
| <i>s</i>     | = 73 <sub>16</sub> = 0111_0011 <sub>2</sub> |
| <i>c</i>     | = 63 <sub>16</sub> = 0110_0011 <sub>2</sub> |
| <i>u</i>     | = 75 <sub>16</sub> = 0111_0101 <sub>2</sub> |

Numele *Ion Stelescu* este stocat în sisteme de calcul ca următoarea secvență de coduri binare (fiecare număr reprezintă un byte).

49 6F 6E 20 53 74 65 6C 65 73 63 75

Ion Stelescu

În binar, secvența este:

0100\_1001\_0110\_1111\_0110\_1110\_0010\_0000\_0101\_0011\_0111\_0100  
0110\_0101\_0110\_1100\_0110\_0101\_0111\_0011\_0110\_0011\_0111\_0101



### ASCII: taina numerelor ce dau glas tăcerii.

Într-această eră a firelor nevăzute și a socotelilor ce se fac într-un ceas mai iute decât clipitul, s-au ridicat, dintre toate limbile pământului, niște semne ce nu-s nici slove, nici chipuri, ci coduri tainice cu nume străin: ASCII.

Și iată cum, dintru 'nceput, aceste numere păreau a fi doar socoteli domnești, 0 până la 127, de la sârmanul nul până la mândrul tilde (~), fiecare purtând în sine duh de cuvânt și năluca de glas. Un fel de Miorită digitală, dar fără fluier, doar cu biți.

Astfel:

- Litera „A” (ce de altfel începe toată cartea) era 65 — semn că domnul ASCII avea simț de noblete și-a pus-o la loc de cinste.
- „Enter”, slujitorul ce face rânduială în slove, purta mantie de 13. Iar când pășea, toți se dădeau la o parte.
- Spațiul? Acel neînțeles al scrisului? Purtător al numărului 32 — nici tânăr, nici bătrân, dar mereu prezent. Tăcut, dar de trebuință mare!

Și astfel, poporul tastelor se rânduia în tăcere, până când vreun programator, meșter într-ale vrăjitoriei de cod, rostea incantația „printf” și tăcerea prindea grai.

Dar vai! Nu toți pot citi aceste semne. Mulți le privesc ca pe niște hieroglife ale viitorului, de zici că-s scrise de călugări pe hârtie de siliciu.

Iar când vreun neofit răstoarnă o literă cu diacritică și primește „ș” în loc de „â” sau „€” în loc de „ț”, atuncea blestemă cu foc și pară:

— **„Blestemat fie codul ăsta al vostru, care mi-a schimbat 'Mărioara' în 'MĂ€rioara'!”**

Și așa, ASCII rămâne în istorie ca un alfabet al tăcerii, ce șoptește dincolo de ochi, auz și simțire – vorbă mută, dar cu greutate numerică.

*Text generat cu ChatGPT.*

### 1.2.6 Codul BCD

**Codul BCD** (*Binary Code Decimal*, engl.) asociază 4 cifre binare fiecărei cifre din baza 10. Din acest motiv combinațiile binare asociate numerelor mai mari decât 9 (1010, 1011, 1100, 1101, 1110, 1111) nu au semnificație în codul BCD. Codul BCD are evidente avantaje la conversia numerelor din baza 10 în binar, însă implică o complexitate hardware mai mare la implementarea operațiilor aritmetice. Codul BCD este utilizat în diverse aplicații de control (care nu necesită procesări aritmetice) care se bazează pe numărări modulo 10.

#### *Exemplu*

- $175|_{10} = 0001\_0111\_0101|_{BCD}$ , combinațiile binare, pe 4 biți asociate cifrelor 1, 7, și 5.
- $2025|_{10} = 0010\_0000\_0010\_0101|_{BCD}$
- $0010\_1001\_0001\_0111|_{BCD} = 2917|_{10}$ , pentru că  $0010 = 2$ ,  $1001 = 9$ ,  $0001 = 1$ ,  $0111 = 7$ .  
De remarcat că  $2917|_{10} = 2048 + 512 + 256 + 64 + 32 + 4 + 1 = 1011\_0110\_0101|_2$ .
- $0010\_1001\_0001\_0111|_2 = 2^{13} + 2^{11} + 2^8 + 2^4 + 2^2 + 2^1 + 2^0 = 8192 + 2048 + 256 + 16 + 4 + 2 + 1 = 10519|_{10}$   
Iar  $10519|_{10} = 0001\_0000\_0101\_0001\_1001|_{BCD}$ .

În concluzie, un șir de cifre binare trebuie înțeles ținând cont de modul în care este codificată informația. Același șir de cifre binare poate reprezenta un număr pozitiv, un număr negativ, un caracter tipăribil sau orice altceva.

De exemplu, șirul de cifre binare 1001\_0001 poate fi înțeles ca:

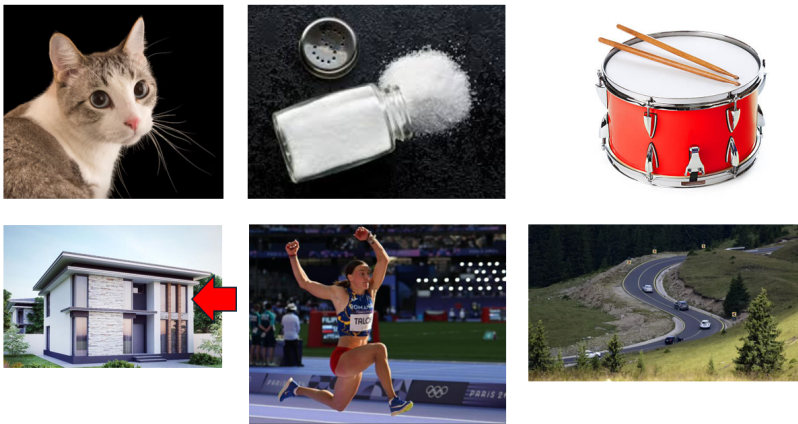
- $145|_{10}$ , dacă reprezintă în binar un număr pozitiv;
- $-111|_{10}$ , dacă reprezintă un număr întreg, în complement față de 2;
- $-0.03515625|_{10}$ , dacă reprezintă un număr real în format FP8 E4M3 (floating point, 8 biți, 1 bit de semn, 4 biți exponent și 3 biți mantisă);
- $9.0625|_{10}$  dacă reprezintă un număr în virgulă fixă (4 biți partea întreagă și 4 biți partea fracționară);
- caracterul ASCII  $91|_{16} = 145|_{10}$ , a cărui formă depinde de setul de caractere al calculatorului;
- $91|_{10}$ , dacă reprezintă un număr pozitiv în format BCD.

Aceasta nu ar trebui să fie o surpriză. Nici în viața de zi cu zi nu este altfel (vezi figura 1.7).

- Ce înțelegeți prin cuvântul „CAT”?<sup>3</sup>
- Ce înțelegeți prin cuvântul „SALT”?<sup>4</sup>
- Ce înțelegeți prin cuvântul „DRUM”?<sup>5</sup>

E confuzionant dacă (nu) știi limbi străine, nu-i așa?

Codarea binară este „limba nativă a calculatoarelor”. Nu toți oamenii o înțeleg cu ușurință.



**Figura 1.7** Decodarea valorilor binare este ca în viață: trebuie să știi formatul pentru a le înțelege. Cuvintele „CAT”,<sup>3</sup> „SALT”<sup>4</sup> și „DRUM”<sup>5</sup> au semnificații diferite, în limbi diferite.

### 1.2.7 Algebra Booleană

Algebra Booleană este definită pe mulțimea binară  $B = \{0, 1\}$  cu legile de compoziție internă:

Complement (NOT):  $\bar{X}$

Conjunție (AND):  $X \cdot Y$

Disjuncție (OR):  $X + Y$

<sup>3</sup>**CAT** = Parte a unei clădiri cuprinzând încăperile sau apartamentele situate pe același plan orizontal. (rom.) Pisică. (engl.)

<sup>4</sup>**SALT** = Mișcare bruscă prin care corpul se desprinde de la pământ. (rom.) Sare. (engl.)

<sup>5</sup>**DRUM** = Cale de comunicație terestră, șosea. (rom.) Tobă (instrument muzical). (engl.)





**Figura 1.8** George Boole (1815-1864). Matematician englez care a fundamentat logica simbolică modernă și a cărei algebră logică, cunoscută sub numele de „algebră Booleană”, constituie baza proiectării circuitelor integrate digitale.

Operatorul SAU EXCLUSIV, cu simbolul  $\oplus$ , este un operator complex definit pe baza operatorilor algebrei Boole conform regulii:

$$X \oplus Y = \overline{X} \cdot Y + X \cdot \overline{Y}$$

Acest operator nu este definit de către algebra booleană, însă are multe aplicații în aritmetica binară. Operatorul SAU EXCLUSIV (XOR) implementează funcția de adunare matematică în baza 2 și funcția de inversare comandată. Tabelul de adevăr al funcției logice XOR și al adunării pe 2 biți este prezentat în continuare.

| A | B | $A \oplus B$ (XOR) | $A + B$ (plus) | $f(A, B)$ , A comandă, B dată  |
|---|---|--------------------|----------------|--------------------------------|
| 0 | 0 | 0                  | 0              | $B$ (dacă $A = 0$ )            |
| 0 | 1 | 1                  | 1              |                                |
| 1 | 0 | 1                  | 1              | $\overline{B}$ (dacă $A = 1$ ) |
| 1 | 1 | 0                  | 0              |                                |

Se observă egalitatea coloanelor asociate funcției logice XOR și a funcției matematice de adunare.

Poarta logică AND implementează un circuit de înmulțire a doi biți. Tabelul de adevăr al funcției logice AND și al înmulțirii pe 2 biți este prezentat în continuare.

| $A$ | $B$ | $A \cdot B$ (AND) | $A \times B$ (înmulțire) | $f(A, B)$ , A comandă, B dată |
|-----|-----|-------------------|--------------------------|-------------------------------|
| 0   | 0   | 0                 | 0                        | 0 (dacă $A = 0$ )             |
| 0   | 1   | 0                 | 0                        |                               |
| 1   | 0   | 0                 | 0                        | $B$ (dacă $A = 1$ )           |
| 1   | 1   | 1                 | 1                        |                               |

Se observă egalitatea coloanelor asociate funcției logice AND și a funcției matematice de înmulțire.

În Verilog, operatorii logici au următoarele simboluri:

| Operator logic | Simbol Verilog |
|----------------|----------------|
| AND            | $\&$           |
| OR             | $ $            |
| NOT            | $\sim$         |
| XOR            | $\wedge$       |

**Teorema dualității:** Orice relație logică are o relație logică duală, cu aceeași valoare de adevăr. Relația logică duală se obține prin interschimbarea operatorilor AND  $\leftrightarrow$  OR și a valorilor 0  $\leftrightarrow$  1.

Tabelul 2.1 prezintă axiomele și teoremele algebrei Booleene, în forma directă și în forma duală.

## 1.3 Inginerie binară

### 1.3.1 Porți logice

O **poartă logică** este un circuit electronic care implementează o funcție logică. Porțile logice au asociate simboluri grafice.

| $A$ | $B$ | NOT<br><br>$\bar{A}$ | AND<br><br>$A \cdot B$ | NAND<br><br>$\overline{A \cdot B}$ | OR<br><br>$A + B$ | NOR<br><br>$\overline{A + B}$ | XOR<br><br>$A \oplus B$ | XNOR<br><br>$\overline{A \oplus B}$ |
|-----|-----|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| 0   | 0   | 1                                                                                                       | 0                                                                                                         | 1                                                                                                                     | 0                                                                                                    | 1                                                                                                                | 0                                                                                                          | 1                                                                                                                       |
| 0   | 1   | 1                                                                                                       | 0                                                                                                         | 1                                                                                                                     | 1                                                                                                    | 0                                                                                                                | 1                                                                                                          | 0                                                                                                                       |
| 1   | 0   | 0                                                                                                       | 0                                                                                                         | 1                                                                                                                     | 1                                                                                                    | 0                                                                                                                | 1                                                                                                          | 0                                                                                                                       |
| 1   | 1   | 0                                                                                                       | 1                                                                                                         | 0                                                                                                                     | 1                                                                                                    | 0                                                                                                                | 0                                                                                                          | 1                                                                                                                       |

Logica Booleană operează cu două valori: *adevărat* și *fals*. Acestor valori li se pot asocia pe mulțimea binară două valori numerice: 0 = fals, 1 = adevărat.

| Axioma                                                                         | Forma cu operatorul                              |                                              |
|--------------------------------------------------------------------------------|--------------------------------------------------|----------------------------------------------|
|                                                                                | AND                                              | OR                                           |
| Axioma 1. Mulțimea $B=\{0, 1\}$ este închisă în raport cu operatorii AND și OR | $\forall X, Y \in B \Rightarrow X \cdot Y \in B$ | $\forall X, Y \in B \Rightarrow X + Y \in B$ |
| Axioma 2. Asociativitatea                                                      | $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$      | $X + (Y + Z) = (X + Y) + Z$                  |
| Axioma 3. Comutativitatea                                                      | $X \cdot Y = Y \cdot X$                          | $X + Y = Y + X$                              |
| Axioma 4. Distributivitatea                                                    | $X \cdot (Y + Z) = X \cdot Y + X \cdot Z$        | $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$    |
| Axioma 5. Existența elementului neutru                                         | $X \cdot 1 = 1 \cdot X = X$                      | $X + 0 = 0 + X = X$                          |
| Axioma 6. Existența complementului                                             | $X \cdot \bar{X} = \bar{X} \cdot X = 0$          | $X + \bar{X} = \bar{X} + X = 1$              |

| Teorema                                           | Forma cu operatorul                                          |                                                      |
|---------------------------------------------------|--------------------------------------------------------------|------------------------------------------------------|
|                                                   | AND                                                          | OR                                                   |
| Teorema 1. Idempotența (tautologia)               | $X \cdot X = X$                                              | $X + X = X$                                          |
| Teorema 2. Legea lui 0 și a lui 1                 | $X \cdot 0 = 0 \cdot X = 0$                                  | $X + 1 = 1 + X = 1$                                  |
| Teorema 3. Dubla negație (involuția)              | $\bar{\bar{X}} = X$                                          | $\bar{\bar{X}} = X$                                  |
| Teorema 4. Absorbția directă<br>Absorbția inversă | $X \cdot (X + Y) = X$<br>$X \cdot (\bar{X} + Y) = X \cdot Y$ | $X + X \cdot Y = X$<br>$X + \bar{X} \cdot Y = X + Y$ |
| Teorema 5. Teorema lui De Morgan                  | $\overline{X \cdot Y} = \bar{X} + \bar{Y}$                   | $\overline{X + Y} = \bar{X} \cdot \bar{Y}$           |

Tabelul 1.2

Axiomele și teoremele algebrei Booleene.

Evaluarea valorii de adevăr a unei expresii formate din propoziții simple este posibilă prin considerarea unor operatori de *negare*, *conjunție* și *disjunție*.

Tabelele de adevăr al operatorilor sunt:

| NOT |           | AND |     |             | OR  |     |         |
|-----|-----------|-----|-----|-------------|-----|-----|---------|
| $A$ | $\bar{A}$ | $A$ | $B$ | $A \cdot B$ | $A$ | $B$ | $A + B$ |
| 0   | 1         | 0   | 0   | 0           | 0   | 0   | 0       |
| 1   | 0         | 0   | 1   | 0           | 0   | 1   | 1       |
|     |           | 1   | 0   | 0           | 1   | 0   | 1       |
|     |           | 1   | 1   | 1           | 1   | 1   | 1       |

Pe baza celor 3 operatori, se pot defini operatori compuși, conform tabelor de adevăr:

| NAND |     |                        | NOR |     |                    | XOR |     |              | XNOR |     |                         |
|------|-----|------------------------|-----|-----|--------------------|-----|-----|--------------|------|-----|-------------------------|
| $A$  | $B$ | $\overline{A \cdot B}$ | $A$ | $B$ | $\overline{A + B}$ | $A$ | $B$ | $A \oplus B$ | $A$  | $B$ | $\overline{A \oplus B}$ |
| 0    | 0   | 1                      | 0   | 0   | 1                  | 0   | 0   | 0            | 0    | 0   | 1                       |
| 0    | 1   | 1                      | 0   | 1   | 0                  | 0   | 1   | 1            | 0    | 1   | 0                       |
| 1    | 0   | 1                      | 1   | 0   | 0                  | 1   | 0   | 1            | 1    | 0   | 0                       |
| 1    | 1   | 0                      | 1   | 1   | 0                  | 1   | 1   | 0            | 1    | 1   | 1                       |

### 1.3.2 Justificarea noțiunii de poartă logică

O poartă logică este un circuit electronic care implementează o funcție logică. Dar de ce se numește circuitul „poartă”?

Justificarea semnificației de „poartă” pentru un circuit care implementează un operator logic devine facilă dacă se consideră porțile cu două intrări. Se poate considera că valoarea semnalului logic aplicat la una din intrări condiționează trecerea prin circuit a semnalului prezentat la cealaltă intrare.

Să considerăm intrările denumite *data* și *validare* și ieșirea denumită *rezultat*. Se pot găsi următoarele semnificații ale funcționării porților logice.

- **Poarta AND:**

Dacă *validare* = 1, trece *data* (poarta este deschisă, *rezultat* = *data*).

Dacă *validare* = 0, la ieșire este întotdeauna 0 (poarta este blocată pentru *data*, *rezultat* = 0).



**Figura 1.9** Justificarea noțiunii de „poartă”: valoarea unui semnal condiționează trecerea altui semnal prin circuit.

- **Poarta OR:**

Dacă  $validare = 0$ , trece  $data$  (poarta este deschisă,  $rezultat = data$ ).

Dacă  $validare = 1$ , la ieșire este întotdeauna 1 (poarta este blocată pentru  $data$ ,  $rezultat = 1$ ).

- **Poarta XOR:**

Dacă  $validare = 0$ , trece  $data$  (poarta este deschisă,  $rezultat = data$ ).

Dacă  $validare = 1$ , trece  $\overline{data}$  (poarta este deschisă ca un inversor,  $rezultat = \overline{data}$ ).

- **Poarta NAND:**

Dacă  $validare = 1$ , trece  $\overline{data}$  (poarta este deschisă ca un inversor,  $rezultat = \overline{data}$ ).

Dacă  $validare = 0$ , la ieșire este întotdeauna 1 (poarta este blocată pentru  $data$ ,  $rezultat = 1$ ).

- **Poarta NOR:**

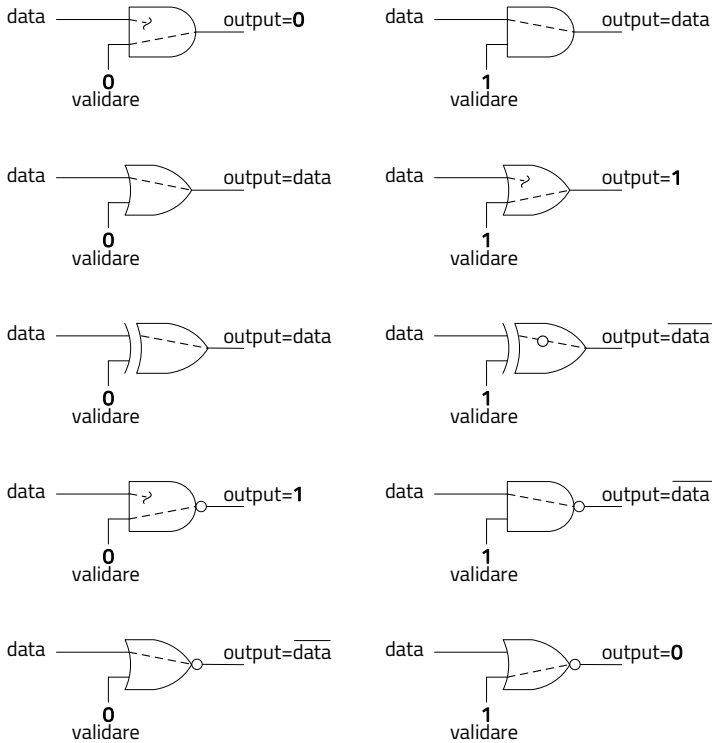
Dacă  $validare = 0$ , trece  $\overline{data}$  (poarta este deschisă ca un inversor,  $rezultat = \overline{data}$ ).

Dacă  $validare = 1$ , la ieșire este întotdeauna 0 (poarta este blocată pentru  $data$ ,  $rezultat = 0$ ).

Se observă că, dacă poarta este deschisă, semnalul de date se propagă de la intrare la ieșire, posibil procesat (negat). Dacă poarta este blocată,

semnalul de date nu se propagă de la intrare la ieșire, iar ieșirea porții are o stare fixă, 0 sau 1, independentă de semnalul de date.

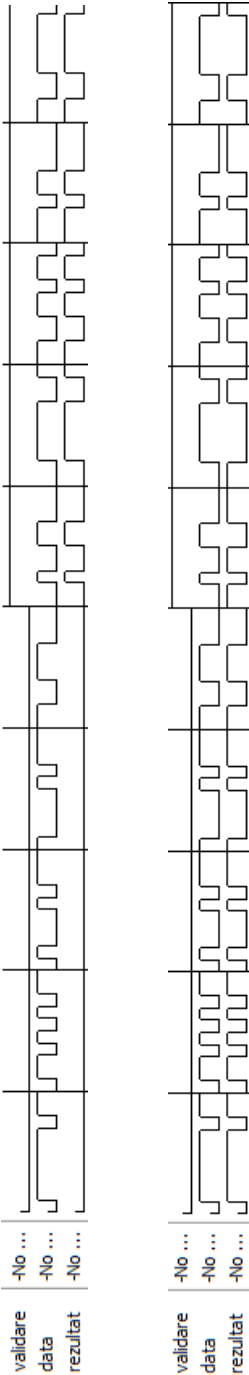
Figura 1.10 prezintă justificarea noțiunii de „poartă” pentru porțile logice AND, OR, XOR, NAND, NOR.



**Figura 1.10** Justificarea noțiunii de „poartă” pentru porțile logice AND, OR, XOR, NAND, NOR.

Și pe formele de undă se poate observa justificarea noțiunii de „poartă”. Ca exemplu, figura 1.11 (sus) prezintă formele de undă simulate pentru poarta logică AND.

Similar, pentru poarta XOR se obțin formele de undă prezentate în figura 1.11 (jos).



**Figura 1.11** Justificarea noțiunii de „poartă” pe formele de undă.  
Pentru poarta AND (sus), se observă că:  
dacă *validare* = 0, la ieșire este întotdeauna 0 (poarta este blocată pentru *data*, *rezultat* = 0), iar  
dacă *validare* = 1, trece *data* (poarta este deschisă, *rezultat* = *data*).

Pentru poarta XOR (jos), se observă că:  
dacă *validare* = 0, trece *data* (poarta este deschisă, *rezultat* = *data*), iar  
dacă *validare* = 1, trece  $\overline{data}$  (poarta este deschisă ca un inversor, *rezultat* =  $\overline{data}$ ).



## Odă Porții Logice

### Acolo unde logica prinde contur

În lumea digitală, acolo unde ordinea și haosul se întâlnesc la fiecare semnal, poarta logică stă la mijloc, un pilon de stabilitate și rațiune. Mică și discretă, ea este inima oricărui circuit digital, cea care asigură că fiecare decizie, oricât de mică, se ia cu claritate și determinare. Fără ea, totul ar fi o mare confuzie de semnale fără sens, un haos total unde 0 și 1 s-ar pierde unul pe altul, iar procesarea datelor ar fi imposibilă.

Poarta logică, indiferent de tipul său – AND, OR, NOT, XOR sau altele – are o singură misiune nobilă: să decidă. Simplu și eficient, ea ia intrările pe care le primește și le transformă în ieșiri clare și bine definite. Ceea ce pare un simplu „DA” sau „NU” într-o propoziție devine, într-un circuit, o chestiune de viață și de moarte pentru performanța sistemului. Fiecare decizie pe care o ia o poartă logică are un impact direct asupra fluxului de informație, iar fără ea, chiar și cele mai complexe procese ar deveni imposibil de realizat.

Poarta logică este fundamentală, dar, paradoxal, adesea rămâne invizibilă pentru majoritatea celor care lucrează cu circuite. Deși pare mică și simplă, ea este, de fapt, motorul care face ca întregul sistem digital să funcționeze. Fără poarta logică, întregul univers digital ar fi doar o înșiruire de 0-uri și 1-uri, fără nicio legătură între ele, fără sens și fără direcție. Într-o lume în care viteza și precizia sunt esențiale, poarta logică își îndeplinește rolul cu o eficiență desăvârșită, fără să ceară laude sau recunoaștere.

Așadar, deși poate părea un element modest al arhitecturii digitale, poarta logică merită toată admirația pentru că este, în realitate, temelia pe care se clădesc toate procesele complexe, fiecare calcul, fiecare decizie.

**Fără poarta logică, digitalul ar fi doar o fantezie fără coordonate.**

*Text generat cu ChatGPT.*



### 1.3.3 Set complet de operatori logici

Pentru realizarea unei funcții logice nu este necesară disponibilitatea tuturor porților logice. Există seturi *complete* de operatori logici prin care se pot realiza orice funcții logice. Acest lucru se poate demonstra matematic, utilizând axiomele și teoremele algebrei Booleene. Noțiunea de „*set complet de operatori logici*” stă la baza programelor software de sinteză de hardware pentru a realiza o implementare automată a oricărei funcții logice cu un subset de primitive existente într-o tehnologie. Există următoarele seturi complete de operatori logici:

- (AND, NOT)
- (NAND)
- (OR, NOT)
- (NOR)
- (XOR, AND)

Utilizând exclusiv porți logice asociate fiecărui set complet de operatori se pot realiza toți operatorii logici. În figura 1.12 este prezentat modul de realizare a operatorilor logici NOT, AND, NAND, OR, NOR, XOR și XNOR utilizând doar porțile din seturile complete (AND, NOT), (NAND), (OR, NOT), (NOR) și (XOR, AND).

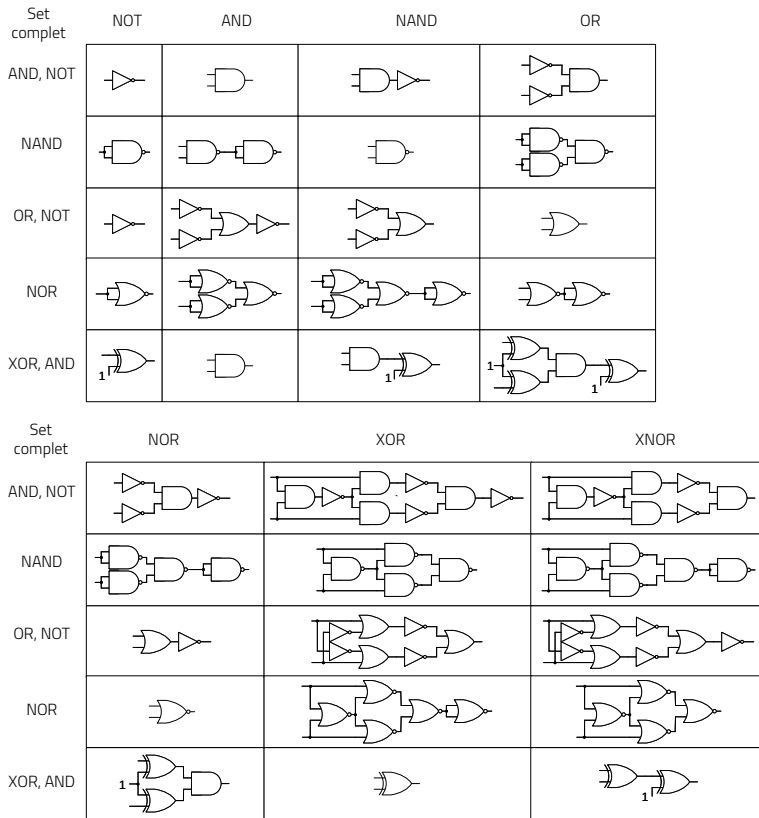
### 1.3.4 Parametrii porților și semnalelor logice

#### Nivele logice

Matematica, algebra Booleană, operează cu valori logice (adevărat și fals) cărora le asociază (abstract) simboluri numerice (0 și 1). Funcțiile logice operează cu valorile logice, într-un mod abstract. Dacă însă vrem să realizăm *circuite electronice* care să aibă ca funcții de transfer funcții logice, trebuie să asociem valorile logice cu mărimi electrice.

Cel mai frecvent, valorile logice, 0 și 1, sunt asociate cu nivele de tensiune electrică (0 = tensiune mică, 1 = tensiune mare).

Se poate face o convenție de „*logică negată*” în care valorile logice se asociază invers cu nivelele de tensiune electrică (1 = tensiune mică și 0 = tensiune mare). Însă, în acest curs vom considera asocierea cea mai frecventă:



**Figura 1.12** Realizarea operatorilor logici cu operatori din seturi complete.

- Stare logică „0” = Nivel mic de tensiune electrică
- Stare logică „1” = Nivel mare de tensiune electrică

Dar, ca ingineri, trebuie să avem definiții clare pentru ce înseamnă „*tensiune electrică mică*” și „*tensiune electrică mare*”.

Apropo, vă aduceți aminte de noțiunea de „*semnal mic*”? Există întrebarea (nu fără răspuns): **Cât de mic este „semnalul mic”?** Mai știți răspunsul la această întrebare?

Se consideră următoarele notații, asociate valorilor limită ale tensiunilor de intrare și de ieșire:

$V_{OH}$  - tensiune minimă *garantată* la ieșirea porții aflate în stare logică 1

$V_{OL}$  - tensiune maximă *garantată* la ieșirea porții aflate în stare logică 0  
 $V_{IH}$  - tensiune minimă *acceptată* la intrarea porții aflate în stare logică 1  
 $V_{IL}$  - tensiune maximă *acceptată* la intrarea porții aflate în stare logică 0

Atributele **garantată** și **acceptată** sunt noțiuni ingineresti care consideră aspecte dincolo de matematică. De exemplu, propagarea semnalelor se face prin medii zgomotoase și cu pierderi. Altfel zis, tensiunea de la emițător s-ar putea să ajungă diminuată la receptor. Pentru a garanta funcționarea corectă a circuitelor electronice în condiții de zgomot și prin linii de transmisiune cu pierderi, emițătorul trebuie să *garanteze* un minimum de tensiune pentru starea de 1 iar receptorul trebuie să *accepte* un minimum de tensiune pentru aceeași stare. Similar pentru starea de 0.

Este evident că, în starea de 1, nivelul de tensiune *garantată* la ieșire trebuie să fie mai mare decât nivelul *acceptat* la intrare (pentru a nu se interpreta greșit starea logică 1 propagată între poarta emițătoare/sursă și poarta receptoare/destinație):

$$V_{OH} > V_{IH}$$

În mod similar, în starea de 0, nivelul de tensiune *garantată* la ieșire trebuie să fie mai mic decât nivelul *acceptat* la intrare (pentru a nu se interpreta greșit starea logică 0 propagată între poarta emițătoare/sursă și poarta receptoare/destinație):

$$V_{OL} < V_{IL}$$

Figura 1.13 prezintă grafic valorile de tensiune de ieșire generate de o poartă emițătoare și cele ale tensiunii de intrare la poarta receptoare. Se observă faptul că domeniul tensiunilor de ieșire *garantate* este mai restrâns față de domeniul tensiunilor de intrare *acceptate*.

### Margine de zgomot

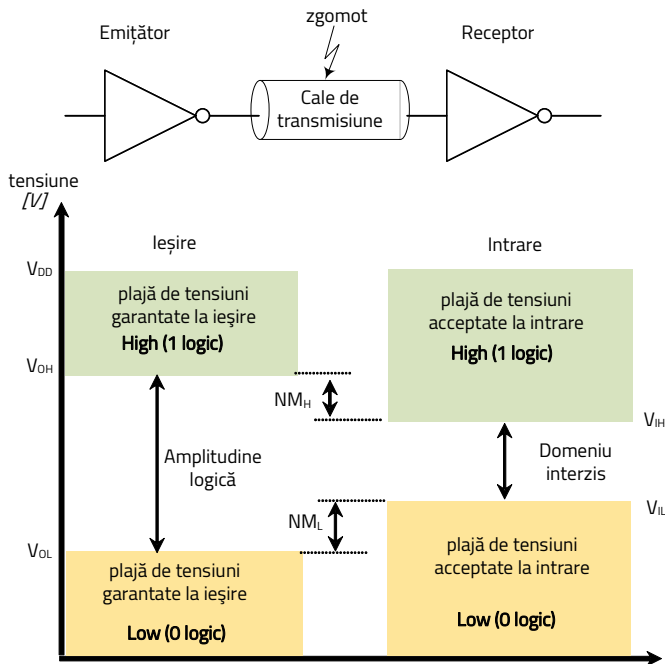
Marginea de zgomot („Noise Margin”, engl.) reprezintă nivelul maxim al zgomotului (considerat o tensiune electrică suplimentară) care poate afecta o linie de transmisiune digitală, astfel încât valoarea logică detectată de receptor să fie identică cu valoarea logică emisă de emițător. Putem considera că zgomotul se adună la semnalul generat de emițător pentru a determina semnalul recepționat. Figura 1.13 prezintă circuitul menționat, graficul cu valorile de tensiune de ieșire generate de o poartă emițătoare și tensiunile de intrare la poarta receptoare, cât și definirea marginilor de zgomot.

Se definesc noțiunile:

Margine de zgomot în starea H (High = 1 logic):  $NM_H = V_{OH} - V_{IH}$

Margine de zgomot în starea L (Low = 0 logic):  $NM_L = V_{IL} - V_{OL}$

Margine de zgomot :  $NM = \min(NM_H, NM_L)$



**Figura 1.13** Definirea nivelelor de tensiune și a marginii de zgomot pentru porțile logice.

### Fan-in

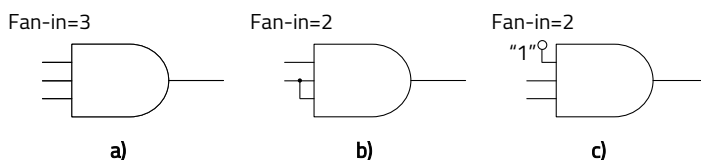
Noțiunea de „fan-in” reprezintă număr de intrări disponibile pentru o poartă logică. Nu se consideră intrările care nu poartă informație (conectate la constante sau intrări echivalente).

În general, o poartă logică cu  $N$  intrări are fan-in= $N$ . Însă, sunt situații

când unele intrări nu sunt utilizate, caz în care acestea se conectează la alte intrări sau la valorile logice 0 sau 1. În acest caz, fan-in este diminuat.

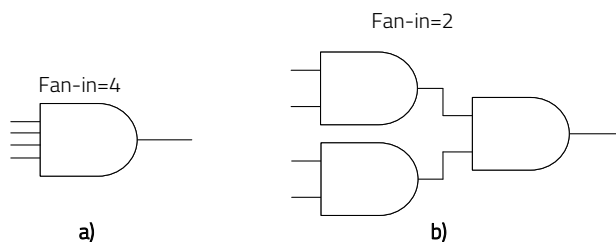
Porțile logice cu fan-in mai mare sunt mai lente decât cele cu fan-in mai mic. Din acest motiv, o poartă logică cu un număr mare de intrări se realizează printr-un circuit logic format din mai multe porți logice cu fan-in mai mic, conectate în cascadă.

Figura 1.14 prezintă circuite logice în care poarta AND cu 3 intrări prezintă fan-in fie egal cu 3, fie egal cu 2.



**Figura 1.14** a) Poartă AND cu 3 intrări și fan-in=3. b) și c) Poartă AND cu 3 intrări și fan-in=2.

Figura 1.15 prezintă o funcție logică AND cu 4 intrări realizată cu o singură poartă logică AND cu fan-in=4, sau cu un circuit logic echivalent format din 3 porți logice AND cu fan-in=2.



**Figura 1.15** a) Funcție logică AND cu 4 intrări realizată cu o singură poartă logică AND cu fan-in=4.

**b)** Funcție logică AND cu 4 intrări cu un circuit format din 3 porți logice AND cu fan-in=2.

## Fan-out

Noțiunea de „fan-out” desemnează parametrul unei porți logice numeric egal cu numărul de porți de același tip ce pot fi comandate de o ieșire. Fiecare intrare într-o poartă logică conectează o capacitate suplimentară pe ieșirea porții care generează semnalul. La o încărcare mare, ieșirea unei porți logice nu mai poate genera un curent suficient de mare pentru a încărca sarcina capacitivă, ceea ce va determina deteriorarea fronturilor semnalelor digitale (fronturi mai puțin abrupte) și menținerea porții logice de destinație cu un semnal de intrare în domeniul interzis (între cele două nivele de tensiune asociate stărilor logice 0 și 1).

Cantitativ, fan-out-ul unei porți se definește ca raportul între curentul de ieșire și curentul de intrare al unei porți logice.

Fan-out în starea H (High = 1 logic):  $FO_H = I_{OH}/I_{IH}$

Fan-out în starea L (Low = 0 logic):  $FO_L = I_{OL}/I_{IL}$

Fan-out:  $FO = \min(FO_H, FO_L)$

S-a notat:

$I_{OH}$  = curent de ieșire când poarta are ieșirea în starea 1 (High)

$I_{IH}$  = curent de intrare când poarta are intrarea în starea 1 (High)

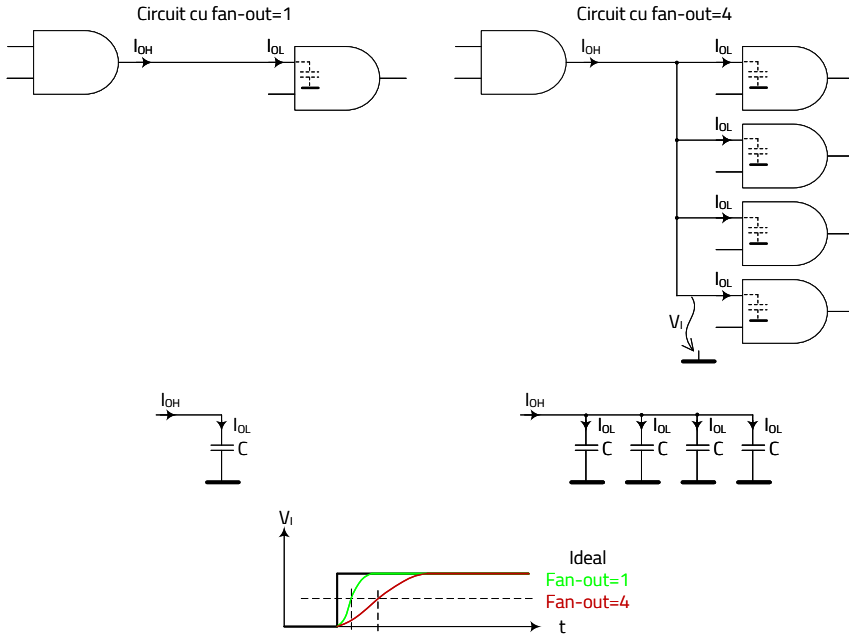
$I_{OL}$  = curent de ieșire când poarta are ieșirea în starea 0 (Low)

$I_{IL}$  = curent de intrare când poarta are intrarea în starea 0 (Low)

Depășirea parametrului fan-out pentru o poartă logică, adică încercarea de a comanda cu o poartă logică mai multe porți logice decât cele admise, va determina o degradare a fronturilor semnalului și, ca efect, o întârziere suplimentară a semnalului. Altfel spus, dacă se depășește fan-out, se micșorează frecvența de funcționare a sistemului digital.

Considerând o analogie între curent electric și apă, depășirea de fan-out este similară cu încercarea de a distribui apă din aceeași țevă (același debit, aceeași presiune) unui bloc de locuințe, deși racordul a fost dimensionat pentru a distribui apă doar unei case. Toți vor primi apă, însă mai lent, și dacă vor dori toți să își umple cada de baie în același timp (analogie cu încărcarea sarcinii capacitive a porților logice comandate) o vor putea face, dar într-un timp mai lung. Deci, nu e bine, pentru că toți ne grăbim, nu-i așa?

Figura 1.16 prezintă explicații pentru înțelegerea noțiunii de fan-out și în ce mod este afectat semnalul electric de către numărul de porți conduse de către o poartă logică.



**Figura 1.16** Explicarea noțiunii de „fan-out”, la comutare din starea 0 în starea 1. Circuite cu fan-out=1 vs. circuite cu fan-out=4. Circuite echivalente considerând capacitatea de intrare a porților receptoare.

### 1.3.5 Timp de propagare

Așa cum „*iarna nu-i ca vara*” și „*practica nu-i ca teoria*” este evident că nici „*ingineria nu-i ca matematica*”. Există multiple deosebiri între abordările și noțiunile matematice și chestiunile ce țin de realizarea fizică a unor sisteme digitale. Una din diferențe între matematică și inginerie este legată de tratarea timpului.

Problema clasică de matematică „*Din orașul A pleacă un tren spre orașul B cu viteza ...*” poate fi rezolvată indiferent de valoarea numerică a vitezei trenului. Însă, inginereste, dacă vrem să rezolvăm problema practic, vom constata că trenul nu poate merge chiar cu orice viteză! Ca efect, trenul nu poate să se deplaseze instantaneu din orașul A în orașul B (nici dacă se teleportează). Deși, matematic și filozofic, ar putea.

Similar se întâmplă cu porțile logice. Putem desena o poartă logică ce

conduce 100 de alte porți logice, însă în realitate, noțiunea de fan-out ne limitează numărul de porți conduse, pentru că poarta logică are un curent de ieșire limitat ce nu permite conducerea unui număr mare de alte porți logice.

**Imaginați-vă un profesor care ține un curs.** Nu-i așa că ar fi fost mai ușor dacă ați fi fost la curs? Poate vorbește profesorul într-o sală cu 30 de studenți? Poate. Dar, dacă în sală sunt 100 de studenți? Se schimbă ceva pentru profesor? Poate că reușește să se facă auzit, dacă vorbește mai tare. Dar ar putea să țină cursul pe un stadion? Nu, fără mijloace de amplificare. Deși profesorul ar vorbi la fel, datorită distanțelor și a numărului mare de receptori (al distanței pe care trebuie să o parcurgă sunetul), semnalul ar ajunge la destinatari atenuat în așa măsură încât nu mai poate fi înțeles.

Deci, deși „matematic” ne putem imagina o persoană care vorbește pe un stadion, aspectele „ingineresti”, ca timpul de propagare al semnalului de la emițător la receptor și zgomotul existent pe linia de transmisiune, fac imposibilă comunicarea.

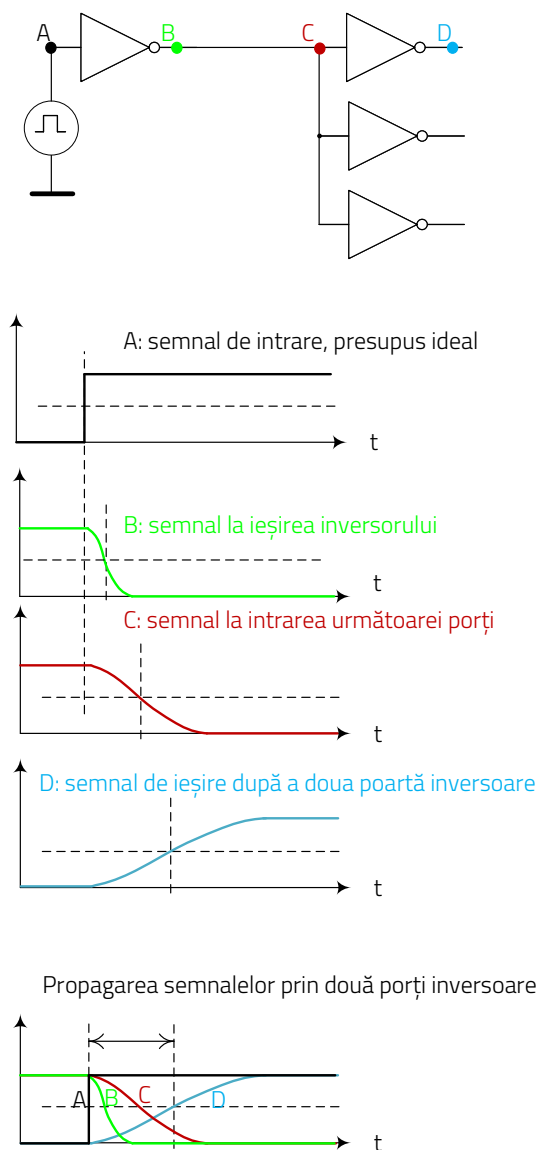
Dacă ați înțeles asta, puteți accepta că orice poartă logică are un timp de propagare între intrare și ieșire. Acest timp este dependent de:

- Tehnologia de realizare a porților logice,
- Fan-out porților logice,
- Caracteristicile electrice ale porților logice (curenți, tensiuni),
- Plasarea porților logice pe suprafața de siliciu.

În matematică, timpul de propagare este considerat egal cu zero. Însă în realitate, acest timp nu este nul și determină limitarea frecvenței maxime de operare a porților logice. Ca să înțelegeți asta, imaginați-vă că vi se cere să faceți operații de adunare cu o cifră.  $2+1=?$   $3+2=?$  Simplu, nu-i așa? Dar dacă vi se cer 100 de asemenea operații pe secundă? Reușiți? Ce este de făcut? Dacă vreți să dați răspunsurile corecte, trebuie să vă evaluați capacitățile și să impuneți o „limită de frecvență” (sau un interval de timp minim între două întrebări). Aceasta se datorează faptului că deși operația este foarte simplă, totuși nu o puteți face instantaneu.

Figura 1.17 prezintă cazul unui inversor (poartă logică ce implementează funcția logică NOT) pentru care se consideră un timp de propagare. Semnalul de ieșire B este întârziat față de semnalul de intrare A (considerat ideal). Semnalul B se propagă prin sârme și mai acumulează întârzieri până la semnalul C, la intrarea în următoarea poartă. Ieșirea D este nu numai negată, dar și întârziată față de semnalul C.

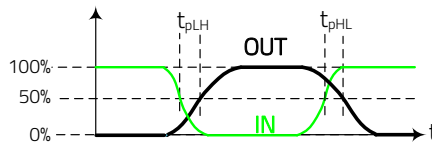




**Figura 1.17** Explicarea timpului de propagare printr-o poartă logică inversoare. Se observă că, deși din punct de vedere logic,  $D=A$ , practic, semnalul din nodul D are altă formă și este întârziat față de semnalul din nodul A.

Astfel, deși din punct de vedere logic, matematic,  $D=A$ , se observă că semnalul din nodul D are altă formă și este întârziat față de semnalul din nodul A. În cazul conectării mai multor porți logice una după alta, întârzierea semnalelor prin porți și fire de interconexiune se acumulează și nu poate fi neglijată.

Timpul de propagare între intrarea și ieșirea unei porți logice se măsoară la 50% din amplitudinea semnalelor de la intrare și ieșire, așa cum este prezentat în figura 1.18.



**Figura 1.18** Măsurarea timpului de propagare printr-o poartă inversoare, pe formele de undă ale intrării și ieșirii, la 50% din amplitudinea acestora.

$t_{PLH}$  - timp de propagare între intrare și ieșire când ieșirea comută din 0 în 1,

$t_{PHL}$  - timp de propagare între intrare și ieșire când ieșirea comută din 1 în 0.

## 1.4 Modelarea sistemelor digitale în HDL

Realizarea unui circuit integrat este foarte costisitoare (milioane de dolari). Nici proiectarea acestora nu este ieftină. Proiectarea necesită munca multor ingineri și pachete software (EDA - Electronic Design Automation) cu licențe foarte scumpe, plătite anual. Mai mult, sistemele digitale actuale sunt foarte complexe (miliarde de porți logice). Este evident că nu se mai poate gestiona complexitatea proiectului cu abordările inițiale. Dacă la nivelul anilor 1970 (acum 50 de ani) primul microprocesor, Intel 4004, s-a proiectat „manual” (fiind primul procesor, nu existau calculatoare, ca azi), metodologia actuală, anii 2025, permite proiectarea pe un singur chip a peste 1 miliard de porți logice.

Acest lucru este posibil datorită suportului adus de metodologia de proiectare asistată de calculator, în care sistemul este descris textual, prin limbaje de descriere hardware.

### 1.4.1 De ce este nevoie de HDL?

Limbajele de descriere hardware (HDL - *Hardware Description Language*, engl.) sunt limbaje de modelare (nu limbaje de programare) a comportamentului sau structurii unui sistem digital. Există extensii ale limbajelor de descriere hardware utilizate pentru a modela circuite analogice și mixte. Există și extensii utilizate intensiv pentru testarea și verificarea sistemelor digitale, înainte și după manufacturare.

În continuare, prin HDL ne vom referi la limbaje de descriere utilizate cu precădere pentru modelarea sistemelor digitale în vederea implementării sub formă de circuite integrate. Există două limbaje HDL utilizate pe scară largă: **VHDL** și **Verilog**.

**VHDL** (*VHSIC HDL, Very High Speed Integrated Circuit, Hardware Description Language*, engl.) a fost dezvoltat de către Departamentul Apărării Statelor Unite (din nou bani de la industria de război...) în anii 1980 și standardizat de către IEEE (*Institute of Electrical and Electronics Engineers*, engl.) cu numele IEEE 1076. Sintaxa VHDL este moștenită din [limbajul de programare ADA](#) și prezintă multiple redundanțe. Scopul inițial al limbajului VHDL a fost să documenteze sistemele digitale. Ulterior a apărut ideea de a realiza o simulare a modelului și, mai târziu, ideea de a sintetiza modelul, adică de a-l transforma automat dintr-o descriere comportamentală într-o structură de primitive (porți logice și bistabile) existente într-o bibliotecă de tehnologie. Mai târziu limbajul a evoluat prin includerea specificațiilor utilizate pentru modelarea circuitelor analogice și mixte **VHDL-AMS** (*Analogue and Mixt Signals*, engl.).

**Verilog** (*VERY LOGic*, engl.) este un alt limbaj de descriere hardware, lansat în 1985 de către o companie privată și ulterior, în 1995, standardizat cu numele IEEE-1364. Verilog, față de VHDL, este un limbaj mai pragmatic, conceput din start pentru a fi utilizat pentru sinteză de hardware. Ulterior, limbajul a evoluat prin includerea specificațiilor utilizate pentru verificarea circuitelor integrate digitale și a devenit **SystemVerilog**.

În prezent, pachetele EDA oferă suport foarte bun pentru modelarea sistemelor digitale cu oricare din limbajele Verilog sau VHDL, inclusiv utilizarea pentru același proiect a ambelor limbaje pentru modelarea și reutilizarea unor sub-module ale sistemului.

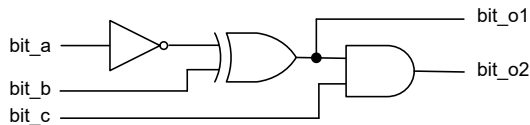
Verilog și VHDL se bazează pe aceleași concepte care caracterizează sistemele digitale: porți/bistabile, combinațional/secvențial, comportament/structură, simulare/sinteză, timp/paralelism. Decizia de a folosi unul sau celălalt este lăsată de obicei pe seama... clientului.

De reținut că VHDL nu este o abreviere pentru „*Verilog HDL*”. Verilog și VHDL sunt două limbaje de descriere hardware diferite. Între acestea există multe asemănări (ambele descriu sisteme hardware digitale) dar există

și deosebiri (sintaxa celor două limbaje este total diferită).

Însă, datorită ușurinței mai mari a scrierii codului (mai intuitiv), înțelegerea mai ușoară a codului și a utilizării mai largi în România, s-a optat pentru utilizarea limbajului Verilog ca prim limbaj pentru învățarea descrierii comportamentului sistemelor digitale.

Limbajele de descriere hardware sunt obligatorii de a fi folosite pentru a reuși gestionarea unor sisteme digitale mari. De exemplu, un circuit format din câteva porți logice, ca cel prezentat în figura 1.19 poate fi descris cu un desen schematic. Poate fi și simulat cu anumite aplicații (inclusiv online), dar cam atât. Trecerea spre un circuit integrat nu se mai face pornind de la schemă.



**Figura 1.19** Schemă cu câteva porți logice, posibil de desenat manual și de înțeles ușor.

Dacă se dorește includerea unui asemenea circuit ca un modul al unui circuit integrat (nimeni nu mai realizează azi circuite integrate cu numai câteva porți logice, asta a fost în anii 1970), ar trebui scris un model HDL ca cel prezentat în continuare.

```

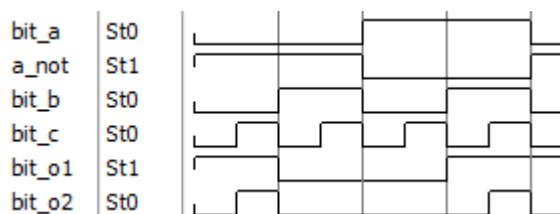
module bit_logic (
input  bit_a  ,
input  bit_b  ,
input  bit_c  ,
output bit_o1 ,
output bit_o2
);

wire a_not ;

assign a_not  = ~bit_a      ;
assign bit_o1 = a_not ^ bit_b ;
assign bit_o2 = bit_o1 & bit_c ;

endmodule // bit_logic
  
```

Circuitul simplu prezentat în figura 1.19 poate fi simulat în conjuncție cu un set de stimuli minimal pentru a se obține formele de undă prezentate în figura 1.20. Prin verificarea formelor de undă, se poate decide dacă circuitul proiectat este cel corect. Desigur, mediului de simulare, generator de stimuli de intrare pentru circuitul verificat, este modelat tot în limbajul Verilog.



**Figura 1.20** Forme de undă obținute prin simularea modelului Verilog al circuitului simplu prezentat în figura 1.19.

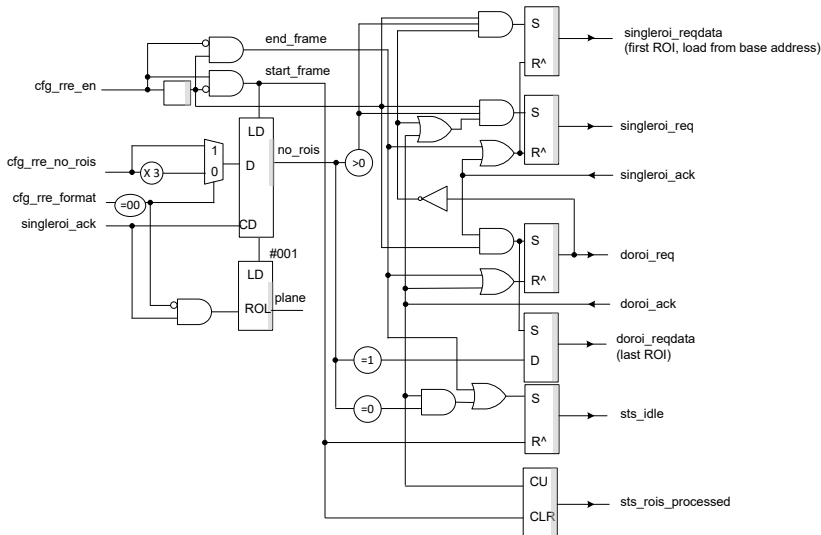
Dacă circuitul este mai complex, ca cel din figura 1.21, situația înțelegerii circuitului se schimbă și modelul HDL ar fi nu numai mai intuitiv dar și mai valoros prin simularea acestuia și preluarea de către programele software de proiectarea automată.

În concluzie, Verilog este un limbaj HDL care oferă suport pentru:

- *Modelarea comportamentală* a sistemului digital proiectat.
- *Modelarea structurală* a sistemului digital proiectat, prin instanțierea mai multor module de complexitate mai mică.
- *Modelarea mediului de testare* care include generatoare de stimuli și specificații de verificare și monitorizare a funcționării modulului proiectat.
- *Simularea* modulului proiectat în cadrul mediului de testare pentru a depana modulul și a certifica corectitudinea proiectării.
- *Suport pentru metodologia de proiectare* asistată de calculator prin modelarea sistemului digital pe diverse nivele de detaliere, în toate etapele de proiectare, până la lansarea manufacturării.

### 1.4.2 Metodologia de proiectare a sistemelor digitale

Metodologia de proiectare a circuitelor integrate digitale este reprezentată grafic în figura 4.14. Se observă că modelul RTL (*Register Transfer Level*,

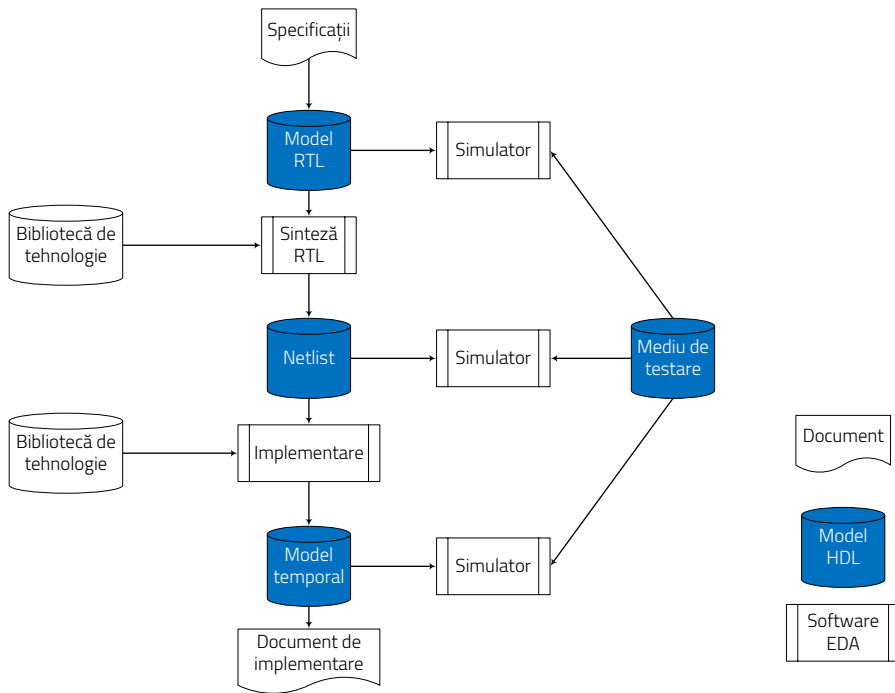


**Figura 1.21** Schemă mai complexă, mai greu de desenat manual și de înțeles. Și totuși sunt doar câteva porți logice, nu miliarde... Modelul HDL este de preferat.

engl.), scris de proiectant, parcurge o succesiune de etape (asistate de produse software specifice) până la livrarea modelului spre a fi fabricat. Metodologia are două părți, denumite în jargonul profesional: „*front-end*” și „*back-end*”.

Partea de „*front-end*” constă în modelarea sistemului de proiectat (comportamental sau structural cu instanțierea și re-utilizarea unor module) și simularea acestuia într-un mediu de verificare. După verificarea funcțională, modelul sistemului se sintetizează pe baza unei biblioteci de tehnologie specifice obținându-se un „*netlist*” (*NETwork LISTing*, engl.). Netlist-ul este generat automat de către software de sinteză, într-un HDL, ca o descriere structurală a proiectului cu instanțe de componente din biblioteca de tehnologie. Acest netlist se recomandă a fi din nou simulat pentru a se evita posibilele erori de sinteză (porțiuni de circuit sintetizate diferit de așteptările proiectantului).

Partea de „*back-end*” pornește de la netlist-ul obținut din sinteză și urmează etape similare cu proiectarea unui cablaj imprimant, însă pentru o implementare sub forma de circuit integrat pe un chip de siliciu. Netlist-ul (descriere de celule digitale primitive interconectate) este plasat pe chip, pe baza unei biblioteci cu informații despre dimensiunile fizice ale celulelor logice. Ulterior, celulele sunt interconectate (rutate) conform informațiilor



**Figura 1.22** Metodologia de proiectare a circuitelor integrate digitale: modelare, sinteză, implementare.

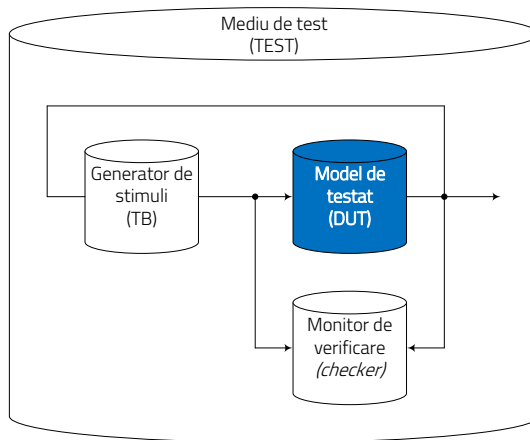
din netlist. Se obține tot un netlist, generat automat de către software de plasare și rutare, care conține adițional informații despre timpul de propagare al semnalelor prin circuit, considerând întârzierile reale prin firele de conexiune (sârme) și logică.

### 1.4.3 Mediul de simulare HDL

Desigur, această descriere a metodologiei de proiectare este mult simplificată. În realitate, metodologia include multe etape de verificare și decizii ale proiectatului (constrângeri de timp și arie, proiectarea dispunerii sistemului pe chip, floorplan, etc.). Simularea sistemului proiectat se face pe mai multe nivele de detaliere, nu doar modelul RTL (scris de proiectant) cât și diverse modele netlist (generate automat de software). Indiferent de nivelul de abstractizare al modelului sistemului digital proiectat, mediul de verificare este similar și conține următoarele componente:

- **Modelul sistemului proiectat**, de obicei denumit DUT (*Device Under Test*, engl.). După verificare, acest model este utilizat ca model de referință care se sintetizează și în final se realizează circuitul integrat proiectat.
- **Generator de stimuli**, model HDL al sistemului exterior circuitului proiectat, utilizat exclusiv pentru verificarea DUT-ului.
- **Mediul de simulare** în care se instanțiază celelalte module menționate, cel puțin DUT-ul și generatorul de stimuli. Mediul de simulare poate fi refolosit la simularea netlist-urilor. Acest modul nu are porturi de intrare sau ieșire și reprezintă modelul HDL care se lansează în simulare (cu ajutorul programelor software de simulare, simulatoare HDL).
- **Monitoare de verificare** pot fi instanțiate, opțional, în mediul de verificare pentru a automatiza verificările în timpul simulării.

Modulele utilizate într-o simulare sunt incluse unele în altele, așa cum sunt prezentate în figura 1.23.



**Figura 1.23** Mediul de simulare HDL.

#### 1.4.4 Modele HDL

În limbajele de descriere hardware (HDL) se realizează modele (comportamentale sau structurale) ale unor sisteme hardware digitale. Descrierile proiectelor în Verilog HDL se numesc **„modele”**, nu „programe” (programele reprezintă descrierea unor algoritmi executați de un procesor).



Modelele reprezintă o abstractizare a unui obiect real (în acest caz un sistem electronic digital).

HDL permite atât modelarea comportamentului (a funcției de transfer) unui bloc de dimensiuni mici, cât și descrierea structurii unui bloc mai mare.

**Modulul** este blocul elementar de construcție în Verilog.

Indiferent de funcție (modelare de hardware, model de test, monitor de date, generator de stimul), toate blocurile funcționale se descriu în Verilog ca **module**.

Modulele sunt descrise prin specificații incluse între cuvintele rezervate `module` și `endmodule`, scrise întotdeauna cu minuscule.

Toate construcțiile sintactice Verilog sunt incluse în module.

Verilog este un limbaj „*case-sensitive*”: toate cuvintele rezervate sunt minuscule și toate denumirile date de proiectați (semnale, porturi, etc.) fac distincție între caracterele majuscule și cele minuscule (semnalul „*Abc*” este diferit de semnalul „*abc*”).

În schimb, VHDL nu este un limbaj „*case-sensitive*”: cuvintele rezervate și denumirile date de proiectat sunt tratate identic, indiferent dacă sunt scrise cu majuscule sau minuscule (semnalul „*Abc*” este același cu semnalul „*abc*”).

Modulele conțin atât informații despre interfața acestora (modul de interconectare a acestora cu alte module) cât și comportamentul sau structura acestora. Formatul modulului facilitează proiectarea „*top-down*” (de la abstract la detalii) și „*bottom-up*” (de la blocuri mici spre sistem complet).

Un modul conține modelul unui proiect sau a unei părți de proiect. Modulele pot încorpora alte module pentru a forma un model ierarhic care descrie cum se includ componentele unui proiect în ansamblul proiectului.

Sintaxa unui modul Verilog prezintă între cuvintele rezervate `module` și `endmodule`:

- *Declarații de parametrii* (constante posibil de suprascris în momentul instanțierii modulului), introduse cu cuvântul rezervat *parameter*.
- *Declarații de porturi*, care reprezintă interfața modulului cu lumea exterioară acestuia.
- *Declarații de date locale* (interne modulului).
- *Specificații concurente*, care se execută în paralel.
- *Instanțieri de module*, pentru a realiza o descriere structurală.

```

module <nume modul> #(
  <declaratii parametrui>
) (
  <declaratii porturi>
);

<declaratii de date>      ;
<specificatii concurente> ;
<instantieri de componente> ;

endmodule

```

De exemplu, următorul cod Verilog descrie un circuit logic care implementează funcția de vot majoritar cu 3 intrări.

```

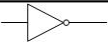
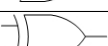
module votMajoritar (
input  i1      , // intrare de vot 1
input  i2      , // intrare de vot 2
input  i3      , // intrare de vot 3
output decizie // decizie de vot majoritar
);

// & = AND, | = OR
assign decizie = (i1 & i2) | (i1 & i3) | (i2 & i3);

endmodule // votMajoritar

```

Verilog are operatorii logici asociați cu simboluri de tip caracter.

| Operator | Poartă logică                                                                       | Simbol Verilog |
|----------|-------------------------------------------------------------------------------------|----------------|
| NOT      |  | ~              |
| AND      |  | &              |
| NAND     |  | ~ &            |
| OR       |  |                |
| NOR      |  | ~              |
| XOR      |  | ^              |
| XNOR     |  | ~ ^            |

Operatorii logici pe vectori operează „bit cu bit”. Rezultatul unui operator pe vector este tot un vector de dimensiunea operanzilor.

| Operator              | Operator | Verilog                    | Exemplu<br>$a = 3'b101$<br>$b = 3'b110$<br>$c = 3'b111$<br>$d = 1'b1$ |
|-----------------------|----------|----------------------------|-----------------------------------------------------------------------|
| $\sim$                | NOT      | $\sim a$<br>$\sim d$       | $\bar{a} = 3'b010$<br>$\bar{d} = 1'b0$                                |
| $\&$                  | AND      | $b\&c$<br>$b\&d$           | $b \cdot c = 3'b110$<br>$b \cdot d = 3'b000$                          |
| $ $                   | OR       | $a b$<br>$a d$             | $a + b = 3'b111$<br>$a + d = 3'b101$                                  |
| $\wedge$              | XOR      | $a\wedge b$<br>$a\wedge d$ | $a \oplus b = 3'b011$<br>$a \oplus d = 3'b100$                        |
| $\sim$ sau $\sim\sim$ | XNOR     | $a \sim b$                 | $\overline{a \oplus b} = 3'b100$                                      |

#### 1.4.5 Setul de valori

Verilog are un set de valori simplu, predefinit în limbaj, cu 4 valori.

- **0**: valoare logică 0 sau condiție „fals”
- **1**: valoare logică 1 sau condiție „adevărat”
- **x**: valoare logică necunoscută sau indiferentă, poate fi orice, 0 sau 1
- **z**: stare de înaltă impedanță (ieșire în gol)

Starea logică x este utilizată pentru a modela stări indiferente sau necunoscute.

Starea logică z este utilizată pentru a modela un semnal fără sursă.

Deși semnalele reale pot avea doar două nivele logice, 0 sau 1, modelele Verilog consideră că orice semnal poate fi în una din cele 4 stări.

#### 1.4.6 Tipuri de date

În Verilog, valorile numerice se declară, împreună cu baza de numerație, cu sintaxa:

[dimensiune] ' <baza> <valoare>

Dimensiunea este opțională și este exprimată în număr de biți, indiferent de baza de numerație.

În lipsa bazei de numerație explicite, valorile se consideră a fi exprimate în baza 10.

Bazele de numerație acceptate în Verilog sunt:

- baza 2 :** desemnată de litera b sau litera B  
**baza 8 :** desemnată de litera o sau litera O  
**baza 10:** desemnată de litera d sau litera D  
**baza 16:** desemnată de litera h sau litera H

Numerele negative sunt precedate de semnul minus –.

| Valoare      | Semnificație                                                                                                                              |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| 13           | numărul 13 exprimat în baza 10 (în binar 1101)                                                                                            |
| -13          | numărul -13 exprimat în baza 10 (în binar, pe 5 biți 1_0011, iar pe 6 biți 11_0011)                                                       |
| 4'b1010      | numărul 10 exprimat în binar pe 4 biți                                                                                                    |
| 12'o327      | număr exprimat pe 12 biți în baza 8                                                                                                       |
| 16'hBEEF     | număr exprimat pe 16 biți în baza 16                                                                                                      |
| 8'd-3        | număr exprimat <b>ilegal</b> (valoarea nu poate fi negativă)                                                                              |
| -8'd3        | număr exprimat legal ca fiind complementul față de 2 al numărului pozitiv 3, reprezentat pe 8 biți (echivalent cu 8'b1111_1101)           |
| 16 'h DEAD   | formă legală cu spații între dimensiune și caracterul ' și între bază de numerație și valoare                                             |
| 4' b0010     | formă <b>ilegală</b> cu spații între caracterul ' și baza de numerație                                                                    |
| width'b1010  | formă <b>ilegală</b> , dimensiunea nu poate fi un parametru sau o expresie                                                                |
| 'b1010       | formă legală, dimensiunea poate lipsi (numărul este reprezentat pe numărul maxim de biți, în acest caz 4)                                 |
| 'hFF         | număr exprimat în baza 16 reprezentat pe 8 biți                                                                                           |
| 8'b101       | număr exprimat în baza 2 reprezentat pe 8 biți ca 8'b00000101                                                                             |
| 8'b0000_0101 | număr echivalent cu forma anterioară (caracterele underscore _ sunt ignorate)                                                             |
| 3'b0110_0101 | număr echivalent cu 3'b101 (caracterele din stânga sunt trunchiate dacă valoarea are mai mulți biți decât dimensiunea precizată explicit) |

|                |                                                                                                                                         |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| 8'b0_11_0_0101 | folosire legală a caracterului de subliniere pentru delimitare                                                                          |
| 8'b_0110_0101  | folosire <b>ilegală</b> a caracterului de subliniere la începutul valorii                                                               |
| 8'd260         | număr echivalent cu 4 în baza 10 (provenit din trunchierea caracterului cel mai semnificativ al reprezentării în binar 9'b1_0000_0100.) |
| 8'bx           | valoare nedefinită extinsă pe 8 biți xxxxxxxx                                                                                           |
| 4'bz           | valoare z extinsă pe 4 biți zzzz                                                                                                        |

### 1.4.7 Codați cu stil!

Una din definițiile [demonline.ro](http://demonline.ro) pentru „**stil**” este:

*„Talentul, arta de a exprima ideile și sentimentele într-o formă aleasă, personală”.*

Stilul se vede și după modul în care scrieți cod. Eu vă recomand „să scrieți cu stil” astfel încât să fiți mândrii de codul HDL scris, iar peste ani să ajungeți măcar odată să spuneți: „dar a fost evident, este stilul meu”.

Cum arată un cod HDL „scris cu stil”?

- **Un fișier, un modul.** Dacă se scrie câte un singur modul într-un fișier și numele fișierului este identic cu numele modulului conținut, viața devine mult mai simplă în cazul lucrului cu multe (zeci, sute) de fișiere. De exemplu, pentru această carte, modulul *bit\_logic* se află în fișierul *bit\_logic.v*. **Very-logic.**
- **Fișierul are un header.** Headerul reprezintă primele linii dintr-un fișier, introduse sub forma de comentarii în codul Verilog (linii precedate de caracterele //). Informațiile din header ar trebui să fie relevante: nume proiect, autor, data, descrierea funcționalității modulului, modificări. De exemplu, pentru această carte, fișierul *bit\_logic.v* are următorul header (primele 11 linii):

```
//-----
// Proiect      : Electronica Digitala
//              : Carte de invatatura
// Autor       : Dan NICULA
// An          : 2025
//-----
// Descriere   : Modul exemplificare schema simpla
```

```
//-----
// Modificari :
// Oct. 1, 2024 (DN): Initial
//-----
```

De remarcat că, anumite simulatoare și programe EDA nu acceptă caractere diacritice în cod, nici măcar în comentarii. În consecință, recomandarea mea este de a scrie toate comentariile, inclusiv cele din header, fără diacritice. Desigur, uneori este o soluție mai bună să scrieți tot codul, inclusiv comentariile și headerul, în limba engleză.

- **Comentariile sunt peste tot.** Nu cădeți în cursa „*mă grăbesc cu codul, scriu comentariile la sfârșit*”. Este doar o scuză. Scrieți comentariile imediat ce scrieți codul. Puneți un comentariu la fiecare port de intrare sau ieșire. Desigur, comentariile să fie relevante. Nu puneți comentariu „*ceas*” după semnalul *clk*, sau „*numărător*” după semnalul *counter*. Puneți comentarii relevante pentru funcționalitatea semnalului.

```
input          clk          , // clock, active rising edge, 1MHz
reg    [8 -1:0] count_access , // counter for number of accesses
                                // between 2 consecutive events
```

Puneți comentarii în descrierea modulelor sau a oricăror specificații mai complicate. Comentați raționamentul expresiilor de decizie complexe.

- **Spațiere și aliniere.** Codul aliniat este mai ușor de urmărit. Există editoare capabile să selecteze și să proceseze blocuri de texte nu doar orizontale ci și verticale. Dacă există un cod aliniat, este mult mai rapid de modificat dacă se apelează la selecție verticală.

```
bit_logic i_bit_logic(
    .bit_a  (bit_a  ),
    .bit_b  (bit_b  ),
    .bit_c  (bit_c  ),
    .bit_o1 (bit_o1 ),
    .bit_o2 (bit_o2 )
);
```

Nu folosiți caractere tabulatori (TAB). Diverse editoare asociază diferite numere de caractere de spațiere unui TAB. Ca efect, un cod ce pare aliniat pe un editor apare nealiniat pe alt editor. Setati editorul ca să înlocuiască automat caracterul TAB cu 2 (nu mai multe) caractere spațiu. Scrieți codul cu alinieri, utilizând tasta TAB, care se va converti în două caractere spațiu. Veți realiza rapid un cod frumos aliniat.

Este necesară o rigurozitate în alinierea specificațiilor cu mai multe cuvinte rezervate (*begin/end, if/else, case/default/endcase*).

- **Convenții pentru nume.** Verilog este un limbaj „*case sensitive*” (minusculile sunt considerate caractere diferite de corespondentele majuscule).

Deși sunt diferite, evitați să denumiți semnale cu aceleași nume dar caractere minuscule/majuscule diferite („*abc*” și „*Abc*”). Motivul este că veți confuziona atât colegii cât și programele EDA (pentru că unele software convertesc toate denumirile la minuscule, ceea ce, în acest caz, va genera erori).

În această carte, semnalele se vor denumi cu minuscule. Ca delimitator, se va utiliza caracterul de subliniere „\_”. Pentru nume de parametri se vor utiliza majuscule.

Desigur, diferite firme au diferite seturi de convenții pentru nume. Eu recomand să fii flexibili și, înainte de a scrie cod pentru o firmă, să întrebați dacă există acest tip de convenții și să le respectați.

- **Asocieri explicite la instanțiere.** Verilog acceptă asocierea numelor formale de porturi cu semnalele actuale atât implicit (prin poziție) cât și explicit (cuplu „port formal” - „semnal actual”). Există multiple argumente pentru a utiliza asocierea explicită a porturilor cu semnale în cadrul instanțierilor de module. De multe ori, lista de porturi este modificată ulterior. În cazul asocierii implicite există riscul de a greși prin editarea portului pe o poziție și a semnalului asociat pe altă poziție. Deși poate părea mai simplu, recomand evitarea instanțierilor implicite. Adică, nu așa:

```
bit_logic i_bit_logic(bit_a, bit_b, bit_c, bit_o1, bit_o2);
```

Deși pe mai multe rânduri, este recomandată asocierea explicită între port și semnal. Rămâne și ocazia unui comentariu la finalul fiecărui rând. Adică așa:

```
bit_logic i_bit_logic(
    .bit_a (bit_a ),
    .bit_b (bit_b ),
    .bit_c (bit_c ),
    .bit_o1 (bit_o1 ),
    .bit_o2 (bit_o2 )
);
```

### 1.4.8 Hard-ul nu-i ca soft-ul!

Modelarea hardware are două mari deosebiri față de programarea software.

1. **Hardware modelează timpul.** În limbajele HDL există clar definit conceptul de timp. Timpul de propagare este inherent oricărui sistem hardware. Din punctul de vedere al proiectantului, trebuie reținute regulile de utilizare a specificațiilor legate de timp. HDL lucrează cu

unități numerice de timp. Cu directiva „*timescale*” se asociază unitatea de timp cu o valoare de timp fizică. Se recomandă ca această asociere să se realizeze exclusiv în mediul de verificare. În modulele RTL (care urmează să fie sintetizate pentru că modelează hardware) nu se vor utiliza specificații care modelează întârzieri în timp, deoarece acestea nu sunt posibil de realizat fizic.

În sistemele sincrone există un semnal periodic de ceas. Se recomandă ca perioada acestuia să fie modelată într-un mod în care se poate modifica rapid, pentru a modela frecvențe diferite ale semnalului de ceas. Întârzierile stimulilor necesari sistemelor digitale ar trebui să se realizeze exclusiv în perioade ale semnalului de ceas, nu ca întârzieri de valori absolute.

2. **Hardware este paralel.** În Verilog comportamentul se modelează cu doar 3 specificații, care sunt executate în paralel (în același timp și independent una de alta): *initial*, *always*, *assign*.

Specificația *initial* nu este sintetizabilă și se folosește doar pentru modelarea mediilor de testare.

Specificația *always* este o specificație concurentă ce include alte specificații cu un comportament secvențial.

Specificația *assign* este o specificație continuă care modelează exclusiv circuite combinaționale.

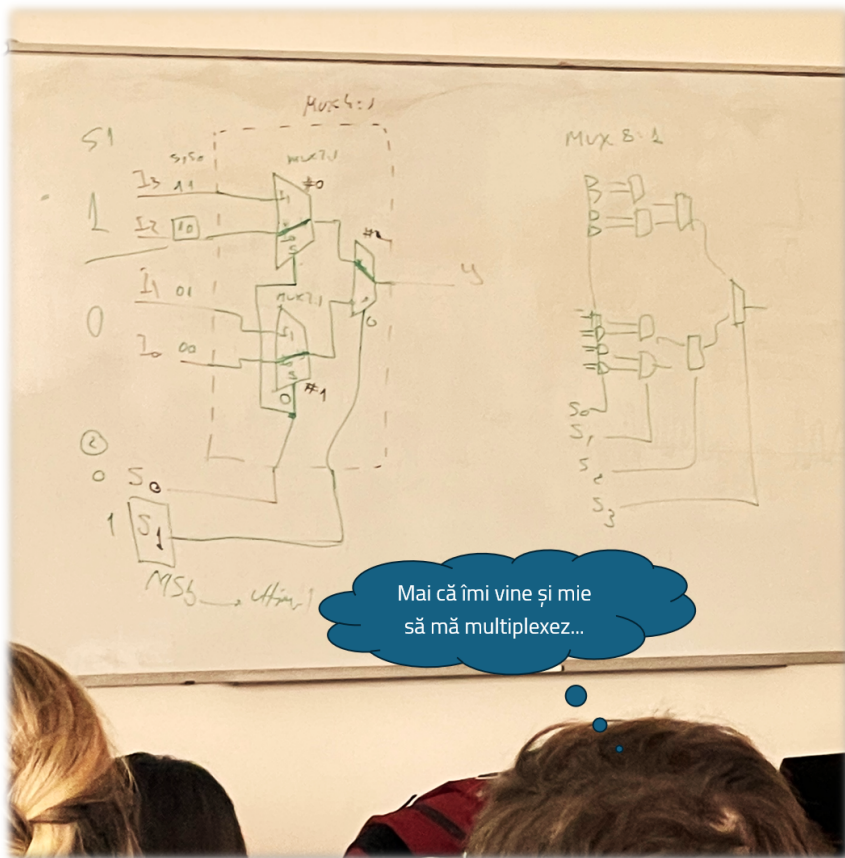
Cele 3 tipuri de specificații se execută în *paralel*. Ordinea lor în cod este indiferentă.

Și instanțierile de module pot fi considerate ca fiind paralele. De fapt, într-o structură de porți logice (de exemplu cea din figura 1.19) fiecare poartă funcționează independent, fără a fi influențată de prezența celorlalte porți. Fiecare poartă își re-evaluează starea de ieșire dacă se modifică semnalele de la intrarea acesteia. De exemplu, poarta AND nu își re-evaluează starea ieșirii la modificarea intrării *bit\_a* ci doar dacă i se modifică intrările proprii, prin propagarea semnalului *bit\_a* prin poarta NOT și prin poarta XOR.



## Capitolul 2

### Circuite logice combinaționale





### **Predoslovie la Capitolul al Doilea:**

#### **Unde circuitele nu țin minte nimic, dar se prefac că gândesc**

După ce ne-am prăfuit mințile prin târâmurile abstracte ale matematicii, iată-ne ajunși la o parte mai practică a cărții, unde intră în scenă circuitele logice combinaționale, numite așa nu pentru că s-ar combina la horă sau la suetă, ci pentru că își scot rezultatul în lume numai după cum le intră datele pe ușă. Și-atât.

Aceste circuite, să le fie numele curat de silicon, sunt precum niște oameni foarte eficienți, dar total lipsiți de memorie: le zici ceva — îți răspund pe loc, da' dacă-i întrebi după două secunde ce-ai spus, se uită la tine ca la o diodă arsă. N-au habar de trecut, nici interes pentru viitor. Ei sunt ai prezentului, domnule!

Și-n această parte a scrierii, vom învăța cum se încropește un adevăr logic din porți de felurite soiuri — AND, OR, NOT și altele, cu nume mai ciudate ca al neamurilor de pe linia maternă. Vom grăi despre diagrame de Karnaugh, care nu sunt niște dansuri orientale, ci niște scurtături în gândirea binară, pentru cei ce vor să simplifice fără să se piardă în pădurea logicii.

Așadar, strânge-ți neuronii ca pe niște șuruburi rătăcite și deschide bine ochii (și eventual schema electrică), căci intrăm în lumea firelor care gândesc... dar fără să știe ce-au gândit mai devreme.

Să purcedem, dară, cu minte ageră și zâmbet în colțul gurii, căci:

**ingineria e mai ușor de dus când nu e mohorâtă!**

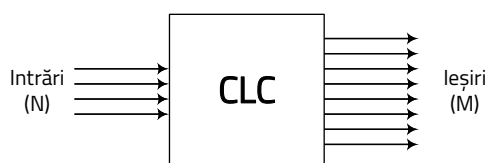
*Text generat cu ChatGPT.*

Circuitele logice combinaționale (**CLC**) sunt circuite electronice care implementează funcții logice. Matematic, circuitele logice combinaționale sunt definite ca o mulțime având 3 elemente:

$$CLC = \{X, Y, f\}$$

unde s-a notat:

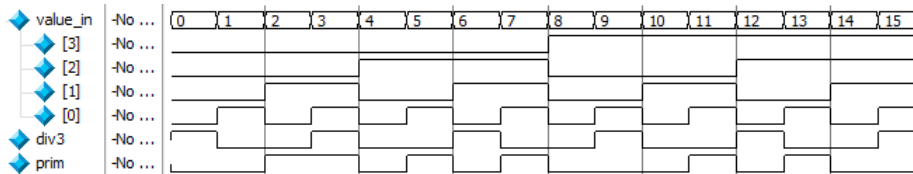
- $X$  este mulțimea combinațiilor de intrare. Dacă CLC are  $N$  intrări, numărul maxim de combinații de intrare este  $2^N$ , însă este posibil ca anumite combinații să nu fie acceptate.
- $Y$  este mulțimea combinațiilor de ieșire. Dacă CLC are  $M$  ieșiri, numărul maxim de combinații de ieșire este  $2^M$ , însă este posibil ca anumite combinații să nu fie necesare.
- $f : X \rightarrow Y$  este funcția de transfer a CLC.



**Figura 2.1** Simbolul bloc al unui circuitul logic combinațional (CLC).

Fizic, circuitele logice combinaționale sunt circuite electronice, deci prezintă un timp de propagare între intrare și ieșire, indiferent de simplitatea funcției de transfer. CLC se caracterizează prin faptul că valorile logice ale ieșirilor depind exclusiv de valorile prezente la intrare. Desigur, dacă o valoare de la intrare se modifică, CLC re-evaluează valoarea de la ieșire. O reprezentare a variației în timp a intrărilor și ieșirilor (forme de undă) este foarte utilă pentru înțelegerea funcției de transfer a unui CLC.

De exemplu, în figura 2.2 sunt prezentate formele de undă asociate unui circuit care determină dacă valoarea prezentă la intrare este divizibilă cu 3 sau este număr prim. Formele de undă sunt obținute prin modificarea intrării astfel încât să fie atinse toate combinațiile posibile ale intrării de 4 biți (numere în domeniul 0...15).



**Figura 2.2** Forme de undă ale CLC (divizibil cu 3 sau un număr prim).

## 2.1 Reprezentarea circuitelor logice combinaționale

Comportamentul circuitelor logice combinaționale poate fi reprezentat în diferite moduri.

Tabelele de adevăr sunt o metodă foarte intuitivă de reprezentare a CLC. Din păcate, pentru un număr mare de intrări (peste 4), tabelele de adevăr au dimensiuni mari și sunt greu de utilizat de către proiectanți.

Formele canonice (standard) sunt utile pentru demonstrații matematice analitice.

Diagramele **Veitch-Karnaugh (V-K)** sunt o formă grafică de reprezentare și sunt utilizate pentru minimizarea unor funcții logice cu un număr mic de intrări (maximum 6).

Funcțiile logice pot fi descrise în limbaje de descriere hardware, cu ajutorul operatorilor logici definiți de limbaj și specificații de atribuire continuă pentru semnale.

### 2.1.1 Tabele de adevăr

Pentru un CLC cu  $N$  intrări și  $M$  ieșiri, tabelul de adevăr listează, pentru fiecare din cele  $2^N$  combinații de intrare, combinația de ieșire asociată, pe  $M$  biți. Tabelul de adevăr este format din:

- $2^N$  rânduri, fiecare combinație de intrare are asociat un rând,
- $N$  coloane pentru cei  $N$  biți de intrare și
- $M$  coloane pentru cei  $M$  biți de ieșire.

Altfel spus, un CLC care are  $N$  intrări și  $M$  ieșiri este caracterizat de un tabel de adevăr de dimensiune  $2^N$  rânduri și  $N + M$  coloane. Pentru

un număr de intrări mare, tabelul de adevăr se dovedește a fi nepractic pentru a fi utilizat. De exemplu, pentru un CLC cu 10 intrări este nevoie de un tabel de adevăr cu  $2^{10} = 1024$  de rânduri.

Pentru un număr mic de intrări, tabelul de adevăr reprezintă o formă ușoară de reprezentare sau de rezolvare a unor probleme logice.

### Exemplu

Utilizați tabele de adevăr pentru a demonstra identitatea:

$$\overline{A \oplus B} = A \oplus \overline{B}$$

### Soluție

Expresia prezintă două variabile  $A$  și  $B$ . Se construiește tabelul de adevăr cu toate combinațiile de intrare  $2^2 = 4$  și se adaugă coloane suplimentare pentru funcțiile necesare,  $A \oplus B$ ,  $\overline{A \oplus B}$ ,  $\overline{B}$ ,  $A \oplus \overline{B}$ .

Rezultă tabelul:

| $A$ | $B$ | $A \oplus B$ | $\overline{A \oplus B}$ | $\overline{B}$ | $A \oplus \overline{B}$ |
|-----|-----|--------------|-------------------------|----------------|-------------------------|
| 0   | 0   | 0            | 1                       | 1              | 1                       |
| 0   | 1   | 1            | 0                       | 0              | 0                       |
| 1   | 0   | 1            | 0                       | 1              | 0                       |
| 1   | 1   | 0            | 1                       | 0              | 1                       |

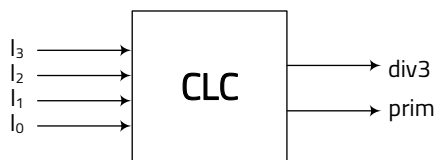
Se observă identitatea coloanelor asociate expresiilor  $\overline{A \oplus B}$  și  $A \oplus \overline{B}$ .

### Exemplu

Să se construiască tabelul de adevăr al unui CLC care acceptă la intrare un număr pozitiv în domeniul  $0 \dots 15$  și determină dacă numărul este divizibil cu 3 sau este un număr prim.

### Soluție

Din enunțul problemei se deduce schema bloc a circuitului logic combinațional, numărul de intrări, numărul de ieșiri și funcția de transfer. Cele 4 intrări sunt denumite  $I_3 I_2 I_1 I_0$  iar ieșirile sunt denumite *div3* și *prim*.



Tabelul de adevăr prezintă  $2^4 = 16$  rânduri și  $4 + 2 = 6$  coloane. A mai fost adăugată, în partea stângă, o coloană cu indexul rândului.

| index | $I_3$ | $I_2$ | $I_1$ | $I_0$ | $div3$ | $prim$ |
|-------|-------|-------|-------|-------|--------|--------|
| 0     | 0     | 0     | 0     | 0     | 1      | 0      |
| 1     | 0     | 0     | 0     | 1     | 0      | 0      |
| 2     | 0     | 0     | 1     | 0     | 0      | 1      |
| 3     | 0     | 0     | 1     | 1     | 1      | 1      |
| 4     | 0     | 1     | 0     | 0     | 0      | 0      |
| 5     | 0     | 1     | 0     | 1     | 0      | 1      |
| 6     | 0     | 1     | 1     | 0     | 1      | 0      |
| 7     | 0     | 1     | 1     | 1     | 0      | 1      |
| 8     | 1     | 0     | 0     | 0     | 0      | 0      |
| 9     | 1     | 0     | 0     | 1     | 1      | 0      |
| 10    | 1     | 0     | 1     | 0     | 0      | 0      |
| 11    | 1     | 0     | 1     | 1     | 0      | 1      |
| 12    | 1     | 1     | 0     | 0     | 1      | 0      |
| 13    | 1     | 1     | 0     | 1     | 0      | 1      |
| 14    | 1     | 1     | 1     | 0     | 0      | 0      |
| 15    | 1     | 1     | 1     | 1     | 1      | 0      |

### 2.1.2 Forme standard

Funcțiile logice pot fi reprezentate în două forme standard:

**Forma Canonică Normală Disjunctivă (FCND)**, sumă de produse („Sum Of Products”, SOP, engl.):

$$SOP = \sum_{i=0}^{2^N-1} (d_i \cdot m_i)$$

**Forma Canonică Normală Conjunctivă (FCNC)**, produs de sume („Product of Sums”, POS, engl.):

$$POS = \prod_{i=0}^{2^N-1} (d_i + M_i)$$

S-au notat:

$d_i$  = coeficienții care definesc funcția;

$m_i$  = **minterm**, produs (funcție logică AND) al tuturor variabilelor de intrare, negate sau nenegate.

$M_i = \text{maxterm}$ , sumă (funcție logică OR) a tuturor variabilelor de intrare, negate sau nenegate.

Dacă  $d_i = 1, \forall i$ , rezultă că suma tuturor mintermilor este egală cu 1:

$$\sum_{i=0}^{2^N-1} m_i = 1$$

,

Dacă  $d_i = 0, \forall i$ , rezultă că produsul tuturor maxtermilor este egal cu 0:

$$\prod_{i=0}^{2^N-1} M_i = 0$$

,

În paragrafele anterioare, s-a considerat:

„produs” = „funcție logică AND” și

„sumă” = „funcție logică OR”.

*Exemplu*

$$F(A, B) = A \oplus B$$

| index $i$ | $A$ | $B$ | $F = A \oplus B$ | Coefficienți | Minterm                                 | Maxterm                             |
|-----------|-----|-----|------------------|--------------|-----------------------------------------|-------------------------------------|
| 0         | 0   | 0   | 0                | $d_0 = 0$    | $m_0 = \overline{A} \cdot \overline{B}$ | $M_0 = A + B$                       |
| 1         | 0   | 1   | 1                | $d_1 = 1$    | $m_1 = \overline{A} \cdot B$            | $M_1 = A + \overline{B}$            |
| 2         | 1   | 0   | 1                | $d_2 = 1$    | $m_2 = A \cdot \overline{B}$            | $M_2 = \overline{A} + B$            |
| 3         | 1   | 1   | 0                | $d_3 = 0$    | $m_3 = A \cdot B$                       | $M_3 = \overline{A} + \overline{B}$ |

$$F_{SOP} = \sum(d_i \cdot m_i) = 0 \cdot m_0 + 1 \cdot m_1 + 1 \cdot m_2 + 0 \cdot m_3 = m_1 + m_2 = \sum(1, 2)$$

$$F_{POS} = \prod(d_i + M_i) = (0 + M_0) \cdot (1 + M_1) \cdot (1 + M_2) \cdot (0 + M_3) = M_0 \cdot M_3 = \prod(0, 3)$$

*Exemplu*

$$F(A, B, C) = \overline{A} \cdot \overline{B} + A \cdot B \cdot C$$

| index $i$ | $A$ | $B$ | $C$ | $F$ | Coefficienți | Minterm                                                    | Maxterm                                            |
|-----------|-----|-----|-----|-----|--------------|------------------------------------------------------------|----------------------------------------------------|
| 0         | 0   | 0   | 0   | 1   | $d_0 = 1$    | $m_0 = \overline{A} \cdot \overline{B} \cdot \overline{C}$ | $M_0 = A + B + C$                                  |
| 1         | 0   | 0   | 1   | 1   | $d_1 = 1$    | $m_1 = \overline{A} \cdot \overline{B} \cdot C$            | $M_1 = A + B + \overline{C}$                       |
| 2         | 0   | 1   | 0   | 0   | $d_2 = 0$    | $m_2 = \overline{A} \cdot B \cdot \overline{C}$            | $M_2 = A + \overline{B} + C$                       |
| 3         | 0   | 1   | 1   | 0   | $d_3 = 0$    | $m_3 = \overline{A} \cdot B \cdot C$                       | $M_3 = A + \overline{B} + \overline{C}$            |
| 4         | 1   | 0   | 0   | 0   | $d_4 = 0$    | $m_4 = A \cdot \overline{B} \cdot \overline{C}$            | $M_4 = \overline{A} + B + C$                       |
| 5         | 1   | 0   | 1   | 0   | $d_5 = 0$    | $m_5 = A \cdot \overline{B} \cdot C$                       | $M_5 = \overline{A} + B + \overline{C}$            |
| 6         | 1   | 1   | 0   | 0   | $d_6 = 0$    | $m_6 = A \cdot B \cdot \overline{C}$                       | $M_6 = \overline{A} + \overline{B} + C$            |
| 7         | 1   | 1   | 1   | 1   | $d_7 = 1$    | $m_7 = A \cdot B \cdot C$                                  | $M_7 = \overline{A} + \overline{B} + \overline{C}$ |

$$F_{SOP} = \sum(d_i \cdot m_i) = 1 \cdot m_0 + 1 \cdot m_1 + 0 \cdot m_2 + 0 \cdot m_3 + 0 \cdot m_4 + 0 \cdot m_5 + 0 \cdot m_6 + 1 \cdot m_7 = m_0 + m_1 + m_7 = \sum(0, 1, 7)$$

$$F_{POS} = \prod(d_i + M_i) = (1 + M_0) \cdot (1 + M_1) \cdot (0 + M_2) \cdot (0 + M_3) \cdot (0 + M_4) \cdot (0 + M_5) \cdot (0 + M_6) \cdot (1 + M_7) = M_2 \cdot M_3 \cdot M_4 \cdot M_5 \cdot M_6 = \prod(2, 3, 4, 5, 6)$$

### Exemplu

Scrieți următoarele funcții ca sumă de mintermi:

**a)**  $F_1(A, B, C) = A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C}$

**b)**  $F_2(A, B, C) = \bar{A} \cdot C + \bar{B} + A \cdot B \cdot \bar{C}$

### Soluție

**a)** Expresia funcției este sub forma unei sume de produse în care apar toate variabilele de intrare.

Reprezentarea binară a indicelui unui minterm se deduce cu regula:

- dacă variabila este ne-negată, bitul corespunzător este egal cu 1,
- dacă variabila este negată, atunci bitul corespunzător este egal cu 0.

Indexul minterm-ului  $A \cdot B \cdot C$  este  $111|_2 = 7|_{10}$ .

Indexul minterm-ului  $\bar{A} \cdot \bar{B} \cdot C$  este  $001|_2 = 1|_{10}$ .

Indexul minterm-ului  $A \cdot B \cdot \bar{C}$  este  $110|_2 = 6|_{10}$ .

Rezultă:  $F_1(A, B, C) = \sum(7, 1, 6) = \sum(1, 6, 7)$

**b)** Expresia funcției este sub forma unei sume de produse în care nu apar toate variabilele de intrare. Forma cu produse cu toate variabilele de intrare se obține prin completarea fiecărui produs cu expresia  $(X + \bar{X}) = 1$  cu variabila (sau variabilele) lipsă. Ulterior, se refăce forma de sumă de produse prin aplicarea distributivității. Prin această prelucrare este posibilă apariția de mai multe ori în sumă a aceluiași minterm. Se va păstra doar câte un minterm, deoarece  $m_i \cdot m_i = m_i$ .

$$\begin{aligned} F_2(A, B, C) &= \bar{A} \cdot C + \bar{B} + A \cdot B \cdot \bar{C} = \\ &= \bar{A} \cdot (B + \bar{B}) \cdot C + (A + \bar{A}) \cdot \bar{B} \cdot (C + \bar{C}) + A \cdot B \cdot \bar{C} = \\ &= \bar{A} \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} = \\ &= \sum(3, 1, 5, 4, 1, 0, 6) = \sum(0, 1, 3, 4, 5, 6) \end{aligned}$$

## 2.1.3 Diagrame Veitch-Karnaugh

**Diagramele Veitch-Karnaugh (V-K)** sunt o modalitate de reprezentare grafică a funcțiilor logice. Pentru o funcție de  $N$  variabile, diagrama corespunzătoare are  $2^N$  căsuțe. Dispunerea căsuțelor se face astfel încât căsuțele



vecine să aibă coduri care diferă printr-un singur bit. Din acest motiv, codurile căsuțelor, pe orizontală și verticală sunt în **cod Gray**, nu în cod binar.

**Codul Gray** are proprietatea că două coduri binare succesive diferă printr-un singur bit.

Codul Gray este un cod ciclic (respectă aceeași regulă la revenirea de la ultimul cod la primul).

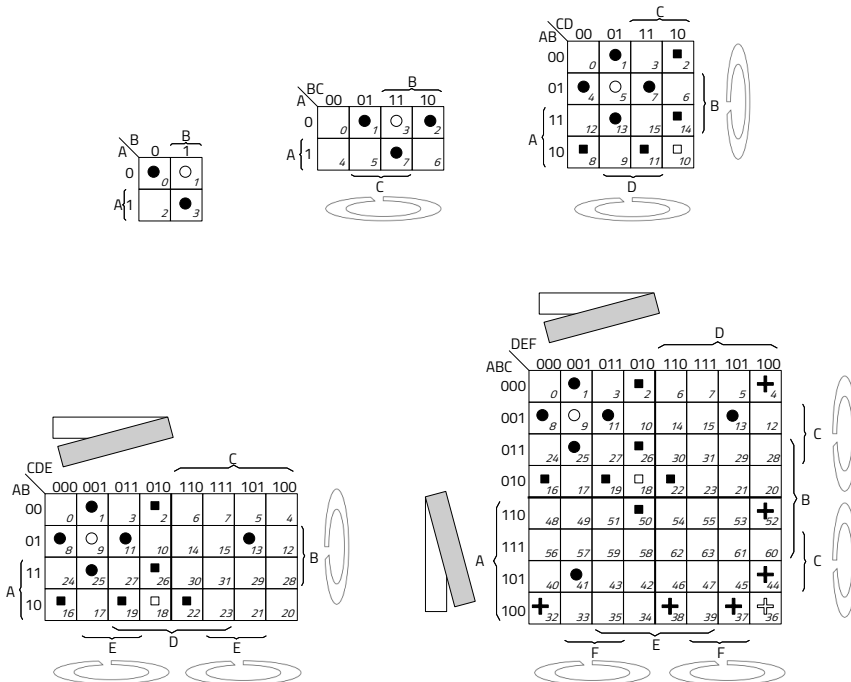
| 2 biți    |       |          |       | 3 biți    |       |       |          |       |       |
|-----------|-------|----------|-------|-----------|-------|-------|----------|-------|-------|
| Cod Binar |       | Cod Gray |       | Cod Binar |       |       | Cod Gray |       |       |
| $B_1$     | $B_0$ | $G_1$    | $G_0$ | $B_2$     | $B_1$ | $B_0$ | $G_2$    | $G_1$ | $G_0$ |
| 0         | 0     | 0        | 0     | 0         | 0     | 0     | 0        | 0     | 0     |
| 0         | 1     | 0        | 1     | 0         | 0     | 1     | 0        | 0     | 1     |
| 1         | 0     | 1        | 1     | 0         | 1     | 0     | 0        | 1     | 1     |
| 1         | 1     | 1        | 0     | 0         | 1     | 1     | 0        | 1     | 0     |
|           |       |          |       | 1         | 0     | 0     | 1        | 1     | 0     |
|           |       |          |       | 1         | 0     | 1     | 1        | 1     | 1     |
|           |       |          |       | 1         | 1     | 0     | 1        | 0     | 1     |
|           |       |          |       | 1         | 1     | 1     | 1        | 0     | 0     |

| Cod Binar |       |       |       | Cod Gray |       |       |       |
|-----------|-------|-------|-------|----------|-------|-------|-------|
| $B_3$     | $B_2$ | $B_1$ | $B_0$ | $G_3$    | $G_2$ | $G_1$ | $G_0$ |
| 0         | 0     | 0     | 0     | 0        | 0     | 0     | 0     |
| 0         | 0     | 0     | 1     | 0        | 0     | 0     | 1     |
| 0         | 0     | 1     | 0     | 0        | 0     | 1     | 1     |
| 0         | 0     | 1     | 1     | 0        | 0     | 1     | 0     |
| 0         | 1     | 0     | 0     | 0        | 1     | 1     | 0     |
| 0         | 1     | 0     | 1     | 0        | 1     | 1     | 1     |
| 0         | 1     | 1     | 0     | 0        | 1     | 0     | 1     |
| 0         | 1     | 1     | 1     | 0        | 1     | 0     | 0     |
| 1         | 0     | 0     | 0     | 1        | 1     | 0     | 0     |
| 1         | 0     | 0     | 1     | 1        | 1     | 0     | 1     |
| 1         | 0     | 1     | 0     | 1        | 1     | 1     | 1     |
| 1         | 0     | 1     | 1     | 1        | 1     | 1     | 0     |
| 1         | 1     | 0     | 0     | 1        | 0     | 1     | 0     |
| 1         | 1     | 0     | 1     | 1        | 0     | 1     | 1     |
| 1         | 1     | 1     | 0     | 1        | 0     | 0     | 1     |
| 1         | 1     | 1     | 1     | 1        | 0     | 0     | 0     |

Fiecare căsuță are un număr de căsuțe vecine egal cu numărul de variabile de intrare.

Căsuțele vecine au un cod care diferă cu un bit față de căsuța de referință.

Fizic, pe diagramă, căsuțele vecine sunt adiacente pe orizontală sau verticală (nu și pe diagonală). Pentru a observa toate căsuțele vecine, pentru mai mult de 2 variabile, diagramele V-K trebuie considerate atât rotite în jurul axelor orizontale și verticale, cât și pliate, așa cum prezintă simbolurile grafice din figura 2.3.



**Figura 2.3** Diagrame V-K pentru 2, 3, 4, 5 și 6 variabile.

Figura 2.3 prezintă diagramele V-K pentru 2, 3, 4, 5 și 6 variabile. Se observă marcarea codurilor Gray pe orizontală și verticală, marcarea suprafețelor asociate variabilelor și a modului de rotire și pliere a diagramelor. În fiecare căsuță este marcat codul numeric asociat căsuței, în baza 10. De asemenea, sunt marcate câteva exemple de căsuțe împreună cu căsuțele vecine din punct de vedere al codurilor (și vecine fizic, dacă diagramele se rotesc și se pliază).

Diagramele Veitch-Karnaugh (V-K) asigură suportul pentru minimizarea funcțiilor logice. Minimizarea funcțiilor logice (scrierea expresiilor acestora cu un număr minim de litere) determină un cost scăzut al implementării lor cu circuite electronice.

Un *minterm* este un termen produs în care apar toate variabilele de intrare, negate sau ne-negate. Pe o diagramă V-K, un minterm care apare în expresia FCND/SOP este asociat cu o căsuță ce conține valoarea 1.

Un *implicant prim* este un termen produs care conține doar anumite variabilele de intrare, negate sau ne-negate. Pe o diagramă V-K, un implicant prim este asociat unei suprafețe având toate câmpurile cu valoarea 1.

Un *implicant prim esențial* este un implicant prim care acoperă în mod unic un câmp cu valoarea 1 și este obligatoriu să fie prezent în expresia minimă a funcției.

## 2.2 Analiza circuitelor logice combinaționale

Analiza circuitelor logice combinaționale constă în *determinarea comportamentului circuitului (funcția logică de transfer), pe baza structurii acestuia* (porți logice interconectate). Comportamentul circuitului logic combinațional poate fi descris prin funcția de transfer, diagrama V-K, tabelul de adevăr și diagrame de timp ale semnalelor de intrare și de ieșire.

Analizarea circuitelor logice combinaționale include și explicarea potențialelor stări de hazard combinațional.

*Hazardul combinațional static* reprezintă o stare logică a cărei apariție nu poate fi explicată prin algebra Booleană.

Cauza hazardului static o reprezintă existența căilor de propagare între o intrare și o ieșire cu timpi de propagare diferiți.

Depistarea hazardului combinațional se poate face:

- *Pe circuit*, dacă se identifică două căi de propagare între o intrare și o ieșire cu timpi de propagare diferiți.
- *Analitic*, dacă expresia funcției se reduce la una din expresiile  $X + \bar{X} = 1$  sau  $X \cdot \bar{X} = 0$  (prin particularizarea valorilor unor intrări).
- *Pe diagrama V-K*, dacă se identifică două suprafețe disjuncte adiacente (alăturate, dar nesuprapuse).

Eliminarea hazardului se face cu un cost adițional de porți logice.

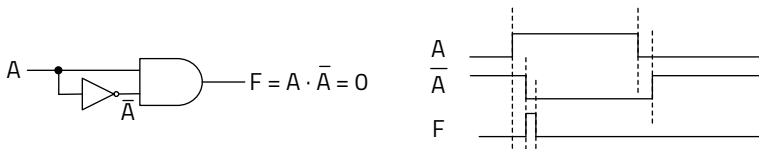
- *Pe circuit*, se include încă o cale între intrare și ieșire, care să fie independentă de variabila ce produce hazard combinațional static.

- *Analitic*, în expresia „sumă de produse” se adaugă un produs format cu variabilele în starea care generează expresia echivalentă  $X + \bar{X}$ .
- *Pe diagrama V-K*, se definește o suprafață suplimentară care să cuprindă căsuțele aparținând inițial unor suprafețe distincte, dar alăturate.

Hazardul combinațional static se poate observa dacă se consideră circuitul din figura 2.4.

Se observă că funcția implementată este  $F(A) = A \cdot \bar{A} = 0$ .

Dar, datorită timpului de propagare prin inversor, semnalele la intrarea și ieșirea inversorului nu comută în același timp. Ca efect, există un scurt interval de timp în care semnalele  $A$  și  $\bar{A}$  sunt egale, nu complementare. În momentul în care  $A = \bar{A} = 1$  funcția devine  $F(1) = 1 \cdot 1 = 1$ . Lățimea acestui puls este dependentă de timpul de propagare prin porțile logice, matematic (ideal) egal cu zero.



**Figura 2.4** Circuit cu hazard combinațional. Forme de undă explicative.

Următorul cod Verilog simulează apariția unui puls de hazard combinațional în cazul modelării porților logice cu un timp de propagare diferit de zero. Timpul de propagare a fost modelat cu 5 unități de timp (#5).

```
assign #5 not_A = ~A;
assign #5 F     = A & not_A;
```

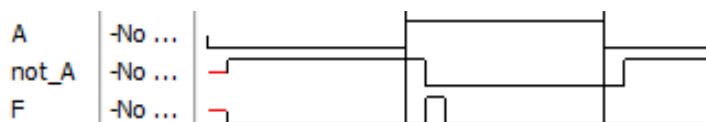
Formele de undă rezultate sunt prezentate în figura 2.5.

### Exemplu

Să se analizeze circuitul combinațional prezentat în figura 2.6-a. Să se deducă funcția de transfer a circuitului. Să se explice apariția hazardului combinațional static utilizând diagramele de timp ale semnalelor și diagramele V-K. Să se propună o soluție de eliminare a hazardului combinațional.

### Soluție

Funcția de transfer a circuitului se deduce pe baza simbolurilor porților



**Figura 2.5** Forme de undă simulate pentru circuitul cu hazard combinațional prezentat în figura 2.4.

logice și a conexiunilor dintre acestea:

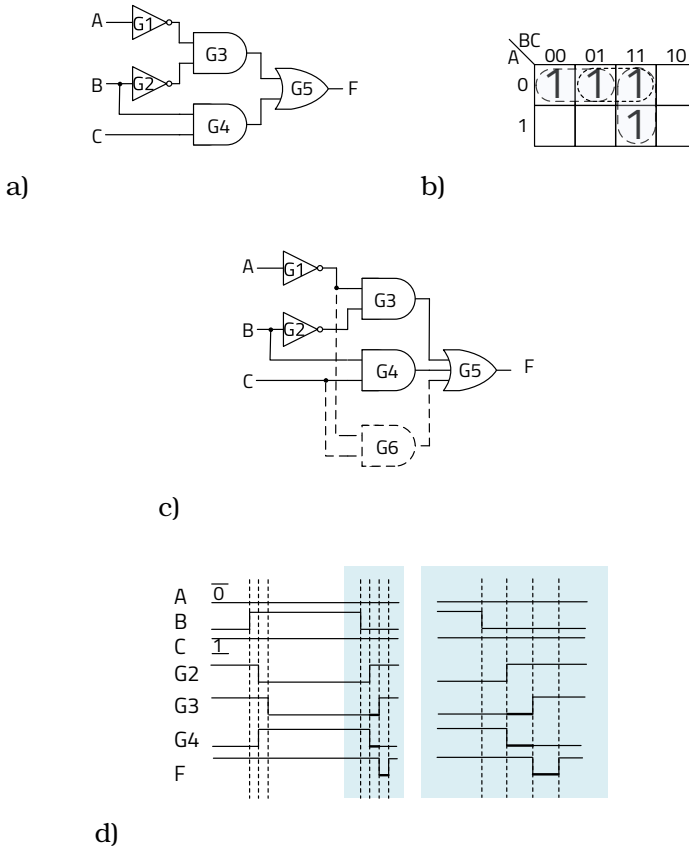
$$F(A, B, C) = \overline{A} \cdot \overline{B} + B \cdot C$$

Depistarea hazardul combinațional static se realizează astfel:

- *Pe circuit*, se observă căile de propagare de la intrarea  $B$  spre ieșirea circuitului: una prin porțile G2/NOT, G3/AND și G5/OR, cealaltă prin porțile G4/AND și G5/OR. Deci, există între o intrare și o ieșire două căi de propagare, cu timpi de propagare diferiți.
- *Analitic*, se observă că, prin particularizarea  $A = 0$  și  $C = 1$ , funcția  $F(A, B, C) = \overline{A} \cdot \overline{B} + B \cdot C$  devine  $F(0, B, 1) = 1 \cdot \overline{B} + B \cdot 1 = \overline{B} + B = 1$ .
- *Pe diagrama V-K*, se observă două suprafețe adiacente disjuncte  $\overline{A} \cdot \overline{B}$  și  $B \cdot C$ .

Eliminarea hazardului se face cu un cost adițional de porți logice.

- *Pe circuit*, se include încă o cale între intrare și ieșire care să fie independentă de variabila care determină hazardul combinațional. Se obține circuitul din figura 2.6-c. Există poarta suplimentară G6/AND.
- *Analitic*, în expresia „sumă de produse” se însumează termenul suplimentar determinat de condiția de hazard  $A = 0$  și  $C = 1$ , adică  $\overline{A} \cdot C$ . Funcția devine:  $F(A, B, C) = \overline{A} \cdot \overline{B} + B \cdot C + \overline{A} \cdot C$ .
- *Pe diagrama V-K*, se consideră suprafața suplimentară care cuprinde căsuțele aparținând inițial unor suprafețe distincte dar alăturate. Suprafața  $\overline{A} \cdot C$  conține câte o căsuță aparținând fiecărei suprafețe alăturate, ca în figura 2.6-b.



**Figura 2.6** a) Circuitul cu hazard combinațional static, b) diagrama V-K asociată, c) circuitul fără hazard combinațional, d) diagramele temporale (detaliu).

Formele de undă pentru condiția de apariție a hazardului combinațional ( $A = 0, C = 1$ ) sunt prezentate în figura 2.6-d. Deși  $F(0, B, 1) = 1$ , se remarcă apariția pulsului de hazard combinațional după comutarea semnalului  $B$  din 1 în 0.  $F(0, B, 1) = 0$ , pentru scurt timp.

### Exemplu

Să se determine dacă circuitul din figura 2.7-a este afectat hazard combinațional static. În caz afirmativ, să se propună o modificare a circuitului astfel încât să fie eliminat hazardul.

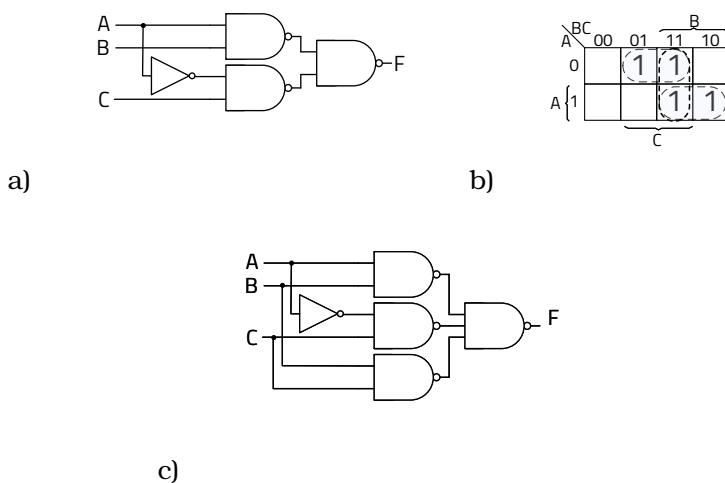
*Soluție*

Din structura circuitului se deduce expresia funcției logice:

$$F = \overline{\overline{A \cdot B \cdot \overline{A \cdot C}}} = A \cdot B + \overline{A} \cdot C$$

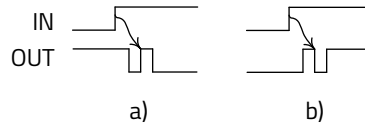
Diagrama V-K din figura 2.7-b prezintă după minimizare două suprafețe adiacente disjuncte, ceea ce indică faptul că se poate genera hazard static când se comută între mintermii  $m_7$ , ( $A \cdot B \cdot C = 111$ ) și  $m_3$ , ( $A \cdot B \cdot C = 011$ ). În acest caz, funcția  $F(A, 1, 1) = A \cdot 1 + \overline{A} \cdot 1 = A + \overline{A} = 1$  ar trebui să fie tot timpul egală cu 1, însă ia valoarea nepermisă 0 la comutația variabilei  $A$  de la 1 la 0.

Hazardul se poate elimina prin adăugarea unui implicant prim  $B \cdot C$  la forma minimă, rezultat din considerarea suprafeței care include mintermii  $m_7$  și  $m_3$ . Se obține funcția echivalentă  $F = A \cdot B + \overline{A} \cdot C + B \cdot C$  cu implementarea prezentată în figura 2.7-c. Implementarea rezultată nu prezintă hazard combinațional, dar nu are costul minim.



**Figura 2.7** a) Circuit, b) Diagrama V-K, c) Circuit fără hazard combinațional static.

Există și un *hazard dinamic* care apare în circuite logice combinaționale cu mai multe nivele de porți logice cascade, în momentul în care comutarea semnalelor de pe intrare ar trebui să determine modificarea stării la ieșire (din 0 în 1 sau din 1 în 0). Efectul constă în apariția unor glitch-uri de scurtă durată pe ieșire datorită revenirii ieșirii în starea care trebuia să fie abandonată. Efectul nu se poate explica matematic dar se explică prin considerarea întârzierilor prin porțile logice (figura 2.8).



**Figura 2.8** Hazard combinațional dinamic a) la comutare din 1 în 0, b) la comutare din 0 în 1.

## 2.3 Sinteza circuitelor logice combinaționale

Orice circuit logic combinațional poate fi implementat cu porți NAND pe două nivele logice.

Metodologia de implementare constă în parcurgerea următoarelor etape:

- Se reprezintă funcția în diagramă V-K.
- Se minimizează funcția sub formă de sumă de produse (SOP).
- Fiecare produs (AND) se implementează cu o poartă NAND, cu un număr de intrări egal cu numărul de termeni în produs. La intrarea portilor se aplică variabilele de intrare, negate sau negate. Aceste porți reprezintă primul nivel de porți NAND.
- Suma (OR) se implementează cu o poartă NAND cu un număr de intrări egal cu numărul de produse din sumă. La intrările porții se aplică ieșirile portilor NAND ale primului nivel logic. Această poartă reprezintă al doilea nivel de porți NAND.

Ca exemplu, funcția  $F(A, B, C) = \sum(1, 3, 6, 7)$  se minimizează sub forma  $F(A, B, C) = A \cdot B + \bar{A} \cdot C$ . Implementarea directă (cu porți AND și OR) și transformarea într-o structură pe două nivele de porți NAND sunt prezentate în figura 2.9.

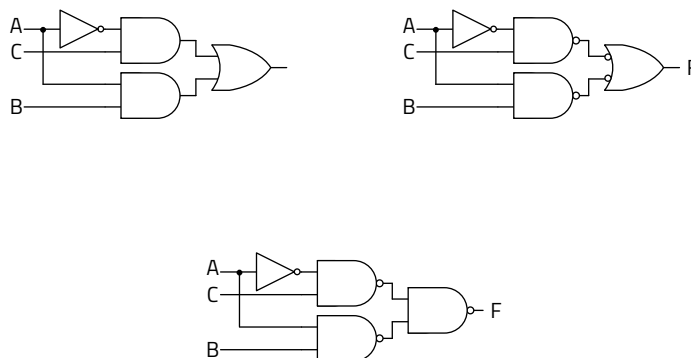
Justificarea analitică este:

$$F(A, B, C) = A \cdot B + \bar{A} \cdot C = \overline{\overline{A \cdot B + \bar{A} \cdot C}} = \overline{\overline{A \cdot B} \cdot \overline{\bar{A} \cdot C}}$$

### 2.3.1 Minimizarea funcțiilor logice, diagrame V-K

Minimizarea funcțiilor logice presupune parcurgerea următoarelor etape:





**Figura 2.9** Funcție logică: implementare AND-OR, implementare NAND-NAND.

1. Se acoperă toate căsuțele cu valoare 1 cu cel puțin o suprafață. Suprafețele sunt dreptunghiulare, cu laturi de dimensiuni puteri ale lui 2 ( $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, \dots$ ). O suprafață conține căsuțe vecine (din punct de vedere al codurilor asociate acestora).

Căsuțele cu valoarea 1 se acoperă cu un număr minim de suprafețe, având dimensiune maximă.

2. Din considerarea suprafețelor care acoperă în întregime toate căsuțele cu valoare 1, se obține expresia minimă a funcției.  
O suprafață de o căsuță generează un minterm în expresia minimă a funcției.

O suprafață de două căsuțe generează un implicant prim în expresia minimă a funcției. Implicantul prim este un produs al tuturor variabilelor de intrare cu excepția celei ce are valoare 0 într-o căsuță și valoare 1 în cealaltă căsuță.

La fiecare dublare a suprafeței, din implicantul prim este eliminată o variabilă de intrare (cea care are valoare 1 în jumătate din căsuțele suprafeței și valoare 0 în cealaltă jumătate).

O suprafață ce acoperă jumătate din diagrama V-K va genera în expresia finală a funcției un „produs” cu o singură literă (variabila de intrare care are aceeași valoare în toate căsuțele suprafeței).

3. Pentru determinarea implicantului prim asociat unei suprafețe, se compară suprafața considerată cu regiunile definite de fiecare variabilă de intrare în parte. Pot exista 3 cazuri:

- (a) Suprafața se situează integral într-o regiune asociată cu o variabilă: în acest caz, în implicantul prim se preia variabila de intrare.
- (b) Suprafața se situează integral în afara unei regiuni asociate cu o variabilă: în acest caz, în implicantul prim se preia variabila de intrare negată.
- (c) Suprafața se situează jumătate în interiorul unei regiuni asociate cu o variabilă și jumătate în exteriorul acesteia: în acest caz, din implicantul prim lipsește variabila de intrare.

### 2.3.2 Minimizarea funcțiilor logice incomplet definite

Un caz particular îl constituie *funcțiile incomplet definite*, care au valori indifferente pentru anumite combinații ale intrărilor. În aceste caz, căsuțele asociate în diagramele V-K vor conține valoare indiferentă, marcată cu  $X$  sau – („don't care”, engl.). Pentru acțiunea de minimizare, valorile indifferente pot fi considerate ca având valori 1 sau 0 astfel încât să se acopere căsuțele ce conțin 1 cu suprafețe cât mai mari și cât mai puține.

#### Exemplu

Să se simplifice funcția incomplet definită, utilizând diagrame V-K.

$$F(A, B, C, D) = \sum(0, 6, 8, 13, 14) + d(2, 4, 10)$$

#### Soluție

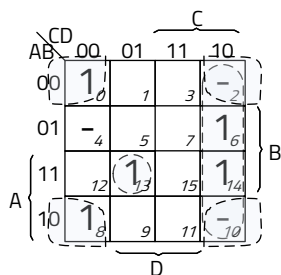
Funcțiile incomplet specificate conțin, pe lângă valorile 0 și 1, și valori indifferente. Valorile indifferente pot fi considerate atât 0 cât și 1, fără a influența funcționarea logicii. Termenii produs care sunt indiferenți, notați cu  $d$  pot fi considerați în diagrama V-K fie de valoare 0, fie de valoare 1. Se va alege valoarea logică ce determină implicanți primi cu o exprimare minimă. În diagrama V-K, aceasta înseamnă definirea unor suprafețe cât mai mari și cât mai puține pentru a acoperi căsuțele cu 1.

Rezultă forma minimă a funcției:

$$F(A, B, C, D) = \overline{B} \cdot \overline{D} + C \cdot \overline{D} + A \cdot B \cdot \overline{C} \cdot D$$

Se observă că, în urma minimizării, au fost considerate ca având valoare 1 căsuțele cu index 2 și 10, iar căsuța cu index 4 ca având valoare 0.

În urma implementării circuitului, conform expresiei minime, în cazurile considerate indifferente, funcția va avea de fapt valorile:



**Figura 2.10** Diagramă V-K pentru funcție logică incomplet definită. Se observă valori 1 în căsuțele prezente în suma  $\sum(0, 6, 8, 13, 14)$  și valori nedefinite – în căsuțele precizate în expresia  $d(2, 4, 10)$ .

$$F(0, 0, 1, 0) = 1$$

$$F(0, 1, 0, 0) = 0$$

$$F(1, 0, 1, 0) = 1$$

### 2.3.3 Minimizarea funcțiilor cu variabile reziduu

Minimizarea funcțiilor cu variabile reziduu se realizează în următoarele etape:

Pentru o funcție cu  $N$  variabile de intrare, o diagramă completă are  $2^N$  căsuțe cu valori 1 sau 0 (prin lipsa unei valori se presupune valoarea opusă celei ce apare în diagramă). Pentru un număr mare de intrări (mai mare decât 4) diagramele pot deveni mari și greu de operat. Din acest motiv, se poate micșora dimensiunea diagramei V-K prin introducerea în căsuțe a unor funcții de una sau mai multe variabile de intrare. Variabilele ale căror nume se regăsesc în interiorul diagramei V-K se numesc *variabile reziduu*.

- **Etapa 1:** Se consideră căsuțele care au valori logice 1 și cele cu valori indiferente. Se determină formele minime ale suprafețelor definite.
- **Etapa 2:** Căsuțele cu valoare 1 se consideră a fi cu valoare indiferentă. Se consideră căsuțele care conțin aceeași funcție reziduu și cele cu valori indiferente. Implicantii primi rezultați vor fi considerați în conjuncție (AND) cu funcția reziduu.
- Forma minimă a funcției se obține prin aplicarea funcției OR asupra tuturor implicantilor primi obținuți la cele două etape anterioare.

- În cazuri particulare, dacă funcțiile reziduu au mai mult de o variabilă, expresia finală se mai poate reduce prin prelucrări analitice.

### Exemplu

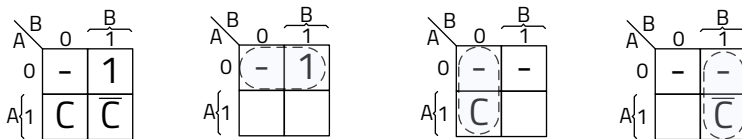
Utilizând diagrame V-K să se minimizeze funcția exprimată cu variabile reziduu:

$$F(A, B, C) = \sum(1, 2 \cdot C, 3 \cdot \overline{C}) + d(0), \text{ unde mintermul este } m(A, B).$$

### Soluție

Diagramele V-K utilizate pentru minimizare sunt prezentate în figura 2.11.

În diagramele funcției sunt prezentate căsuțele cu valoare 1, „-” (indiferente) sau cu variabile reziduu. În diagramele căsuțelor 1, căsuțele cu funcții reziduu sunt considerate cu 0. În diagramele funcțiilor reziduu, căsuțele cu 1 sunt considerate „-”, iar funcțiile reziduu sunt considerate una câte una. Implicanții primi rezultați din minimizare se consideră în conjuncție cu variabila reziduu corespunzătoare. În final, expresia obținută mai poate fi minimizată analitic.



**Figura 2.11** Diagrame V-K cu variabile reziduu: diagrama funcției, diagrama căsuțelor cu valoare 1, și diagrame cu variabilă reziduu.

$$\text{Rezultă funcția: } F(A, B, C) = \overline{A} + \overline{B} \cdot C + B \cdot \overline{C}$$

### 2.3.4 Minimizarea funcțiilor logice cu ieșiri multiple

Funcțiile cu ieșiri multiple (dependente de aceleași variabile de intrare) nu se minimizează independent, ci corelat. Motivul îl constituie posibilitatea reutilizării unor termeni produs pentru realizarea mai multor funcții, cu un cost global mai redus.

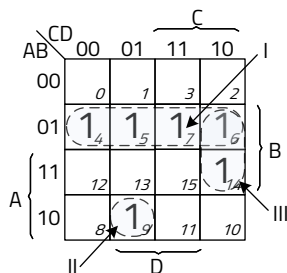
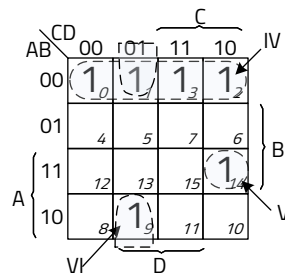
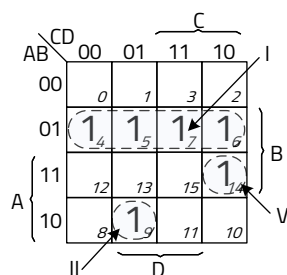
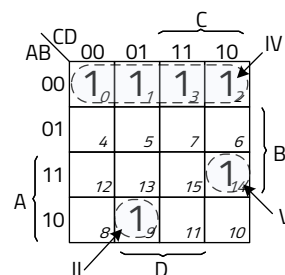
### Exemplu

Minimizați corelat funcțiile:

$$F_1(A, B, C, D) = \sum(4, 5, 6, 7, 9, 14) \text{ și}$$

$$F_2(A, B, C, D) = \sum(0, 1, 2, 3, 9, 14)$$

Soluție

a)  $F_1$  $F_2$ b)  $F_1$  $F_2$ 

**Figura 2.12** Minimizarea corelată a funcțiilor:

a) funcții minimizate independent,

b) funcții minimizate corelat.

Minimizarea independentă a funcțiilor (figura 2.12-a) determină expresiile:

$$F_1(A, B, C, D) = I + II + III = \overline{A} \cdot B + A \cdot \overline{B} \cdot \overline{C} \cdot D + B \cdot C \cdot \overline{D}$$

$$F_2(A, B, C, D) = IV + V + VI = \overline{A} \cdot \overline{B} + A \cdot B \cdot C \cdot \overline{D} + \overline{B} \cdot \overline{C} \cdot D$$

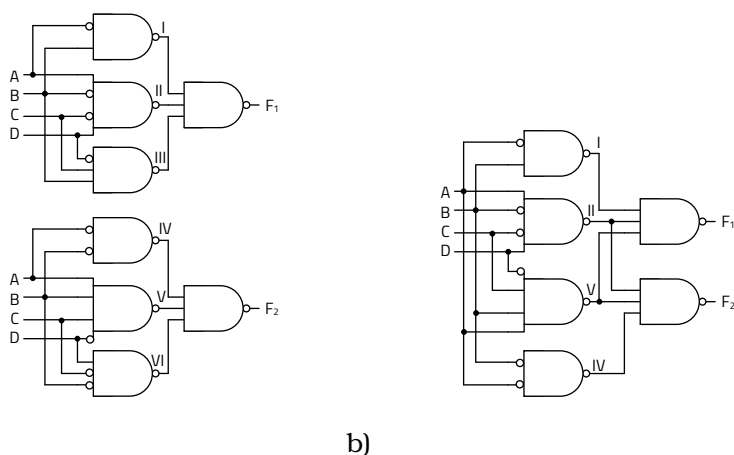
Minimizarea corelată a funcțiilor (figura 2.12-b) se bazează pe observația că suprafețele de o căsuță oricum trebuie considerate (pentru că reprezintă implicantii primi esențiali în câte una din funcții) și pot fi refolosite pentru cealaltă funcție, nemaifiind necesară încă o suprafață adițională.

Expresiile funcțiilor sunt:

$$F_1(A, B, C, D) = I + II + V = \overline{A} \cdot B + A \cdot \overline{B} \cdot \overline{C} \cdot D + A \cdot B \cdot C \cdot \overline{D}$$

$$F_2(A, B, C, D) = II + IV + V = A \cdot \overline{B} \cdot \overline{C} \cdot D + \overline{A} \cdot \overline{B} + A \cdot B \cdot C \cdot \overline{D}$$

Implementările asociate celor două tipuri de minimizări (independente și corelate) sunt prezentate în figurile 2.13.



**Figura 2.13** Implementarea funcțiilor corelate:

- a) minimizate independent,  
b) minimizate corelat.

Compararea resurselor necesare este prezentată în tabel:

| <i>Minimizate independent</i> |            | <i>Minimizate corelat</i> |            |
|-------------------------------|------------|---------------------------|------------|
| Porți logice                  | Intrări    | Porți logice              | Intrări    |
| 2 × NANDx4                    | 8          | 2 × NANDx4                | 8          |
| 4 × NANDx3                    | 12         | 2 × NANDx3                | 6          |
| 2 × NANDx2                    | 4          | 2 × NANDx2                | 4          |
| 4 × NOT                       | 4          | 4 × NOT                   | 4          |
| Total: 12 porți               | 28 intrări | Total: 10 porți           | 22 intrări |

## 2.4 CLC particulare

Anumite circuite logice combinaționale au funcții particulare, care le fac foarte frecvent utilizate. Limbajele de descriere hardware permit modela-

rea acestora mai simplu, pe baza funcției combinaționale particulare implementate, fără a fi nevoie de minimizări sau alte operații manuale.

Desigur, conversia comportamentului la o structură de porți logice se face cu utilizarea unor programe software pentru sinteza de hardware.

### 2.4.1 Decodificator

*Decodificatorul  $N : 2^N$  are  $N$  intrări și  $2^N$  ieșiri.*

Pe intrări se primește codul binar al unui număr în domeniul între 0 și  $2^N - 1$ .

Decodificatorul activează exclusiv ieșirea cu indexul egal cu numărul prezentat la intrare. Starea activă a ieșirii poate fi în 0 sau în 1.

În cazul în care ieșirile sunt active în 1, se poate considera că decodificatorul generează pe cele  $2^N$  ieșiri **toți mintermii celor  $N$  variabile** prezentate la intrare.

Pentru  $N = 1$ , decodificatorul  $1 : 2$  (*decodificatorul de 1 bit*) se poate realiza cu un singur inversor, dacă se consideră tabelul de adevăr al funcțiilor implementate.

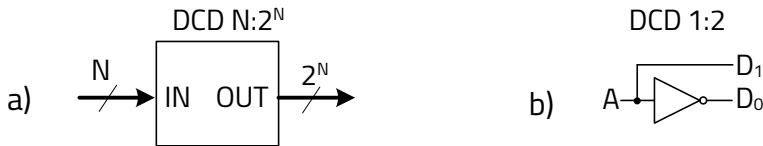
| $A$ | $D_1$ | $D_0$ |
|-----|-------|-------|
| 0   | 0     | 1     |
| 1   | 1     | 0     |

Se deduc funcțiile:

$$\begin{aligned} D_1 &= A \\ D_0 &= \overline{A} \end{aligned}$$

Pentru  $N = 2$ , decodificatorul  $2 : 4$  se poate realiza cu decodificatoare de 1 bit pentru fiecare intrare, urmate de un set de  $2^2 = 4$  porți AND care generează ieșirile. Funcțiile implementate sunt prezentate în tabelul următor.

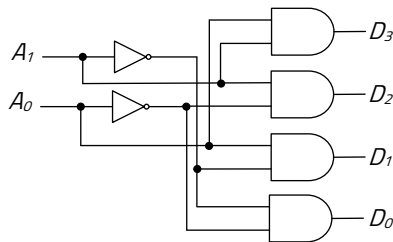
| $A_1$ | $A_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 1     |
| 0     | 1     | 0     | 0     | 1     | 0     |
| 1     | 0     | 0     | 1     | 0     | 0     |
| 1     | 1     | 1     | 0     | 0     | 0     |



**Figura 2.14** a) Simbol de decodificator generic  $N : 2^N$ ,  
b) Structură de decodificator 1 : 2 (decodificatorul de 1 bit).

Se deduc funcțiile:

$$\begin{aligned} D_3 &= A_1 \cdot A_0 \\ D_2 &= A_1 \cdot \overline{A_0} \\ D_1 &= \overline{A_1} \cdot A_0 \\ D_0 &= \overline{A_1} \cdot \overline{A_0} \end{aligned}$$



**Figura 2.15** Structura decodicatorului 2 : 4 (decodicatorul de 2 biți).

Modelul Verilog al decodicatorului 2 : 4 este următorul:

```
module dcd_2to4 (
input    [2 -1:0] A,    // intrare (2 biti)
output   [4 -1:0] D     // iesire  (2^2=4 biti)
);

assign D[3] = A[1] & A[0];
assign D[2] = A[1] & ~A[0];
assign D[1] = ~A[1] & A[0];
```

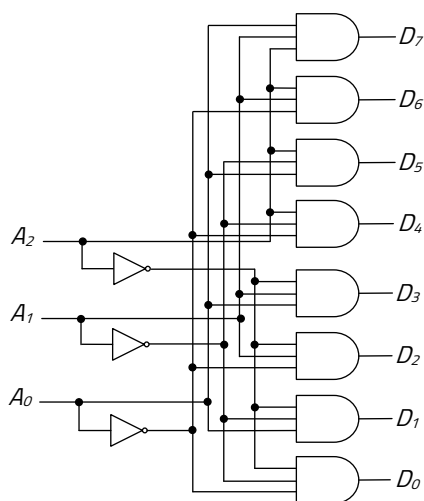


```
assign D[0] = ~A[1] & ~A[0];

endmodule // dcd_2to4
```

În mod similar, pentru  $N = 3$ , decodificatorul 3 : 8 se poate realiza cu decodificatoare de 1 bit pentru fiecare intrare, urmate de un set de  $2^3 = 8$  porți AND care generează ieșirile. Funcțiile implementate sunt prezentate în tabelul următor.

| $A_2$ | $A_1$ | $A_0$ | $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 1     |
| 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 0     |
| 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 0     |
| 1     | 0     | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 0     |
| 1     | 0     | 1     | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     |
| 1     | 1     | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     |
| 1     | 1     | 1     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |



**Figura 2.16** Structura decodificatorului 3 : 8.

Modelul Verilog al decodificatorului 3 : 8 este următorul:

```
module dcd_3to8 (
input  [3-1:0] A,  // intrare (3 biti)
```

```

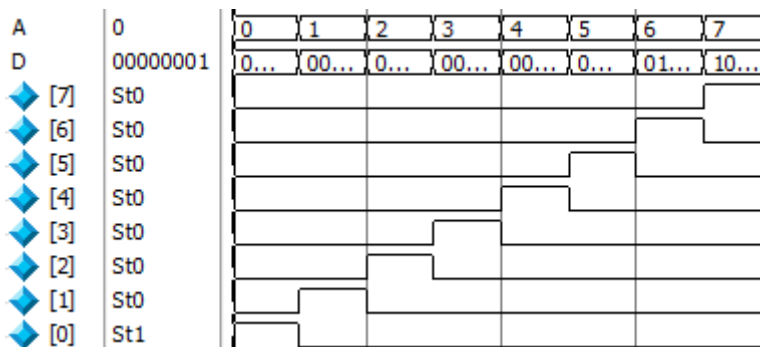
output [8-1:0] D // iesire (2^3=8 biti)
);

assign D[7] = A[2] & A[1] & A[0];
assign D[6] = A[2] & A[1] & ~A[0];
assign D[5] = A[2] & ~A[1] & A[0];
assign D[4] = A[2] & ~A[1] & ~A[0];
assign D[3] = ~A[2] & A[1] & A[0];
assign D[2] = ~A[2] & A[1] & ~A[0];
assign D[1] = ~A[2] & ~A[1] & A[0];
assign D[0] = ~A[2] & ~A[1] & ~A[0];

endmodule // dcd_3to8

```

Figura 2.17 prezintă formele de undă obținute prin simularea modelului de decodificator 3 : 8, în toate cele 8 combinații ale intrărilor.



**Figura 2.17** Forme de undă obținute prin simularea modelului de decodificator 3 : 8.

Desigur, pentru un număr mai mare de biți, modelarea decodificatorului pe baza funcției fiecărei ieșiri este și laborioasă și riscantă (posibile greșeli și testabilitate îngreunată). Se poate realiza un model pentru un decodificator cu dimensiune generică (parametrizabil) pe baza operatorului Verilog de deplasare, dacă se descrie comportamentul decodificatorului astfel:

*Decodificatorul prezintă la ieșire valoarea 1 deplasată la stânga cu numărul de biți precizat la intrare.*

Modelul Verilog al decodificatorului generic  $N : 2^N$  este următorul:

```

module dcdGeneric #(

```

```

parameter WIDTH = 3           // numar de biti de intrare
) (
input  [WIDTH      -1 : 0] A, // intrare (WIDTH biti)
output [(1<< WIDTH)-1 : 0] D // iesire decodata (2^WIDTH biti)
);

assign D = 1'b1 << A;        // 1 deplasat stanga cu A pozitii

endmodule // dcdGeneric

```

Un decodificator poate avea o **intrare suplimentară de validare**.

Dacă intrarea de validare este activată, decodificatorul funcționează „clasic” și prezintă la ieșire valoarea decodificată a intrării (o singură ieșire activată).

Dacă intrarea de validare nu este activată, toate ieșirile decodificatorului sunt în starea inactivă.

Intrarea de validare oferă suport pentru extinderea capacității de decodificare și realizarea unui decodificator cu un număr mai mare de biți, utilizând mai multe instanțe de decodificatoare mai mici.

Pentru  $N = 2$ , decodificatorul 2 : 4 cu validare se poate realiza cu decodificatoare de 1 bit pentru fiecare intrare, urmate de un set de  $2^2 = 4$  porți AND care generează ieșirile. Intrarea de validare este prezentă la o intrare suplimentară în toate porțile AND și determină deschiderea ( $E = 1$ ) sau blocarea ( $E = 0$ ) acestora. Funcțiile implementate sunt prezentate în tabelul următor.

| $E$ | $A_1$ | $A_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|-----|-------|-------|-------|-------|-------|-------|
| 0   | X     | X     | 0     | 0     | 0     | 0     |
| 1   | 0     | 0     | 0     | 0     | 0     | 1     |
| 1   | 0     | 1     | 0     | 0     | 1     | 0     |
| 1   | 1     | 0     | 0     | 1     | 0     | 0     |
| 1   | 1     | 1     | 1     | 0     | 0     | 0     |

Se deduc funcțiile ieșirilor:

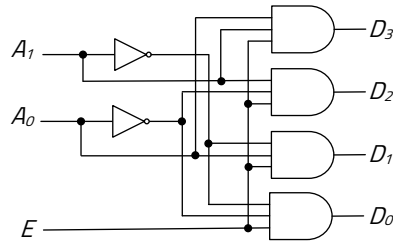
$$D_3 = E \cdot A_1 \cdot A_0$$

$$D_2 = E \cdot A_1 \cdot \overline{A_0}$$

$$D_1 = E \cdot \overline{A_1} \cdot A_0$$

$$D_0 = E \cdot \overline{A_1} \cdot \overline{A_0}$$

Modelul Verilog al unui decodificatorului  $N : 2^N$  cu validare este următorul:



**Figura 2.18** Structura decodicatorului 2 : 4 cu validare.

```

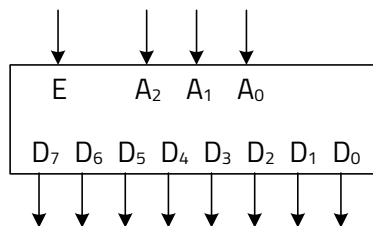
module dcd_valid #(
parameter WIDTH = 3           // numar de biti de intrare
) (
input  [WIDTH-1 : 0] A, // intrare (WIDTH biti)
input  E, // validare, activ 1
output [(1<< WIDTH)-1 : 0] D // iesire decodata (2^WIDTH biti)
);

assign D = E << A;           // E deplasat stanga cu A pozitii

endmodule // dcd_valid

```

Simbolul unui decodicator 3 : 8 cu validare este prezentat în figura 2.19.



**Figura 2.19** Simbol de decodicator cu validare 3 : 8.

Implementarea circuitelor logice combinaționale cu un decodificator prezintă anumite avantaje datorită existenței tuturor mintermilor generați de decodificator. Implementarea nu necesită minimizarea funcțiilor. Mintermi prezenți în forma canonică SOP sunt grupați la intrările unei porți OR. Dacă sunt necesare mai multe funcții logice pe baza acelorași intrări, este suficient de utilizat un singur decodificator. Se va proiecta câte o poartă OR pentru fiecare funcție de ieșire.

Dacă în expresia unei funcții apar mai mult de jumătate din mintermi, este posibilă reducerea costului implementării prin considerarea mintermilor care lipsesc din expresia funcției și negarea porții OR.

### Exemplu

Implementați următorul set de funcții utilizând un decodificator 3 : 8.

$$F_1(A, B, C) = \sum(0, 2, 4)$$

$$F_2(A, B, C) = \sum(1, 3, 4, 7)$$

$$F_3(A, B, C) = \sum(0, 2, 4, 5, 6, 7)$$

### Soluție

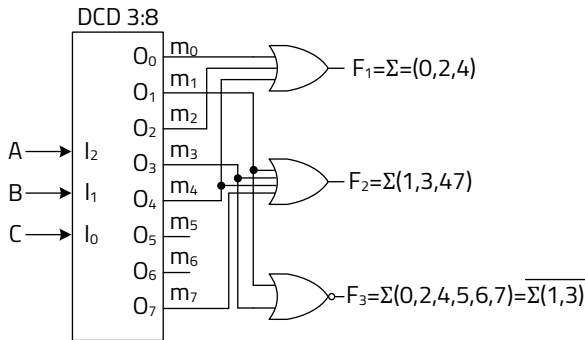
În implementarea prezentată în figura 2.20 se observă:

- Existența unui singur decodificator utilizat pentru implementarea tuturor funcțiilor.
- Dimensiunea decodicatorul dependentă de numărul de intrări ale funcțiilor. În acest caz sunt 3 intrări, deci decodicatorul ales este 3 : 2<sup>3</sup>.
- Conectarea intrărilor funcțiilor pe intrările decodicatorului în ordinea corectă. Astfel, la ieșirea decodicatorului se obțin mintermi variabilelor de intrare.
- Conectarea unei porți OR suplimentare pentru fiecare funcție.
- Dacă un minterm apare în expresia mai multor funcții, ieșirea corespunzătoare a decodicatorului se va conecta la intrările corespunzătoare mai multor porților OR (cazul  $m_4$ ).
- Dacă un minterm nu apare în expresia niciunei funcții, ieșirea corespunzătoare a decodicatorului poate fi lăsată în gol (cazurile  $m_5$  și  $m_6$ ).
- Dacă o funcție are în componență mai mult de jumătate din numărul total de mintermi aceasta se poate implementa negată, astfel încât poarta NOR să aibă un număr de intrări mai mic (egal cu maximum jumătate din numărul total de mintermi).

Este cazul funcției  $F_3(A, B, C) = \sum(0, 2, 4, 5, 6, 7)$ , implementată ca:

$$F_3(A, B, C) = \overline{\sum(1, 3)},$$

adică o poartă NOR cu  $8 - 6 = 2$  mintermi în loc de o poartă OR cu 6 mintermi prezenți în expresia funcției.



**Figura 2.20** Implementarea unui set de 3 funcții cu un decodificator 3 : 8.

### Rețetă

Orice funcție logică cu  $N$  intrări poate fi implementată cu următoarele resurse:

- decodificator  $N : 2^N$  și
- o poartă logică de maximum  $2^{N-1}$  intrări.

- Pentru implementarea unei funcții de  $N$  variabile cu un DCD  $N : 2^N$  funcția nu trebuie minimizată deoarece în nivelul AND (din structura decodificatorului) sunt produși toți cei  $2^N$  mintermi de  $N$  variabile.
- Circuitul DCD  $N : 2^N$  produce la ieșire toți cei  $2^N$  mintermi de  $N$  variabile. Orice funcție poate fi scrisă sub forma canonică disjunctivă (FCND) ca o „sumă de mintermi”. Existând generați toți cei  $2^N$  mintermi la ieșirile decodificatorului, mintermi necesari funcției sunt selectați într-un nivel de logică cu o poartă OR.
- În cazul în care decodificatorul are ieșirile negate (ieșiri active în starea 0), mintermi sunt generați negați. Din acest motiv, pentru implementarea unei funcții exprimate prin FCND în exteriorul decodificatorului trebuie adăugată o poartă NAND.

- Pentru cazul în care funcția însumează un număr de mintermi mai mare decât jumătatea celor existenți  $2^{N-1}$ , se poate face sinteza funcției negată  $\bar{F}$ , deoarece aceasta se poate exprima cu mai puțini mintermi, și ulterior se neagă ieșirea. Pentru cazul când decodicatorul are intrările active în 1, se poate exprima funcția astfel încât să necesite în exterior o poartă NOR.

În concluzie, implementarea optimă a unei funcții exprimate în FCND cu circuit decodicator se realizează astfel:

- Dacă decodicatorul are ieșirile active în 1, iar funcția un număr de mintermi mai mic decât jumătate, colectarea într-o poartă OR a mintermilor existenți în expresia funcției.
- Dacă decodicatorul are ieșirile active în 0, iar funcția un număr de mintermi mai mic decât jumătate, colectarea într-o poartă NAND a mintermilor existenți în expresia funcției.
- Dacă decodicatorul are ieșirile active în 1, iar funcția un număr de mintermi mai mare decât jumătate, colectarea într-o poartă NOR a mintermilor inexistenți în expresia funcției.
- Dacă decodicatorul are ieșirile active în 0, iar funcția un număr de mintermi mai mare decât jumătate, colectarea într-o poartă AND a mintermilor inexistenți în expresia funcției.

### 2.4.2 Codificator

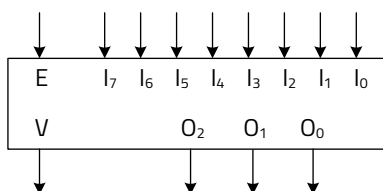
*Codificatorul prioritar de  $2^N$  biți are  $2^N$  intrări și  $N$  ieșiri.*

La ieșire se generează codul binar al celei mai prioritare intrări activate.

Codificatorul poate avea o ieșire suplimentară care semnalează apariția pe intrare a unei configurații valide (cu cel puțin un bit egal cu 1). Astfel, se poate face distincție între cazul când a fost activată doar intrarea cu index 0 (cod de ieșire cu toți biții egali cu 0) și cazul când nicio intrare nu a fost activată (caz în care codul de ieșire este același, cu toți biții egali cu 0).

Atributul de „prioritar” este necesar deoarece este posibil ca la intrare să fie activate mai multe intrări simultan. În acest caz, este codat la ieșire indexul celei mai prioritare dintre intrări.

Funcțiile implementate de codificatorul prioritar de 8 biți (8 : 3) sunt prezentate în tabelul următor.



**Figura 2.21** Simbol de codificator prioritar de 8 biți.

| $E$ | $I_7$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ | $V$ | $O_2$ | $O_1$ | $O_0$ | Index         |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-----|-------|-------|-------|---------------|
| 0   | X     | X     | X     | X     | X     | X     | X     | X     | 0   | 0     | 0     | 0     | nevalidat     |
| 1   | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0   | 0     | 0     | 0     | nicio intrare |
| 1   | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 1   | 0     | 0     | 0     | intrare 0     |
| 1   | 0     | 0     | 0     | 0     | 0     | 0     | 1     | X     | 1   | 0     | 0     | 1     | intrare 1     |
| 1   | 0     | 0     | 0     | 0     | 0     | 1     | X     | X     | 1   | 0     | 1     | 0     | intrare 2     |
| 1   | 0     | 0     | 0     | 0     | 1     | X     | X     | X     | 1   | 0     | 1     | 1     | intrare 3     |
| 1   | 0     | 0     | 0     | 1     | X     | X     | X     | X     | 1   | 1     | 0     | 0     | intrare 4     |
| 1   | 0     | 0     | 1     | X     | X     | X     | X     | X     | 1   | 1     | 0     | 1     | intrare 5     |
| 1   | 0     | 1     | X     | X     | X     | X     | X     | X     | 1   | 1     | 1     | 0     | intrare 6     |
| 1   | 1     | X     | X     | X     | X     | X     | X     | X     | 1   | 1     | 1     | 1     | intrare 7     |

Modelul Verilog al unui codificatorului 8 : 3 cu validare este următorul:

```

module enc8_3 (
input      [8 -1:0] I , // intrare de date (4 biti)
input      E , // intrare de validare
output reg[3 -1:0] O , // iesire (2 biti)
output     V // iesire valida
);

always @(E or I)
if (E) // validat la intrare
case (I)
8'b0000_0001: O <= 8'd0; // intrare 0 cea mai prioritara
8'b0000_001x: O <= 8'd1; // intrare 1 cea mai prioritara
8'b0000_01xx: O <= 8'd2; // intrare 2 cea mai prioritara
8'b0000_1xxx: O <= 8'd3; // intrare 3 cea mai prioritara
8'b0001_xxxx: O <= 8'd4; // intrare 4 cea mai prioritara
8'b001x_xxxx: O <= 8'd5; // intrare 5 cea mai prioritara
8'b01xx_xxxx: O <= 8'd6; // intrare 6 cea mai prioritara
8'b1xxx_xxxx: O <= 8'd7; // intrare 7 cea mai prioritara
default      : O <= 8'd0; // nicio intrare de date valida
endcase

```



```

else
    O <= 8'd0; // nevalidat la intrare

assign V = E & (!I); // cod valid la iesire daca E=1 si
                    // I are cel putin un bit activ (1)
endmodule // enc8_3

```

### 2.4.3 Multiplexor

**Multiplexorul MUX  $2^N$  :** 1 selectează pe baza unui cod de  $N$  biți una din cele  $2^N$  intrări de date pe care o transmite la unica ieșire.

Funcționarea multiplexorului 2:1 (o intrare de selecție și două intrări de date) se poate descrie astfel:

*„Dacă selecția este egală cu 0, ieșirea ia valoarea intrării  $I_0$ , iar dacă selecția este egală cu 1, ieșirea ia valoarea intrării  $I_1$ ”.*

Multiplexorul 2 : 1 implementează în hardware funcția de decizie, similară cu specificația software „if ... then ... else”

```

if <conditie>
    <iesire> = <functie_1>
else
    <iesire> = <functie_0>

```

Multiplexorul 1 :  $2^N$  are:

- $N$  intrări de selecție
- $2^N$  intrări de date
- O ieșire

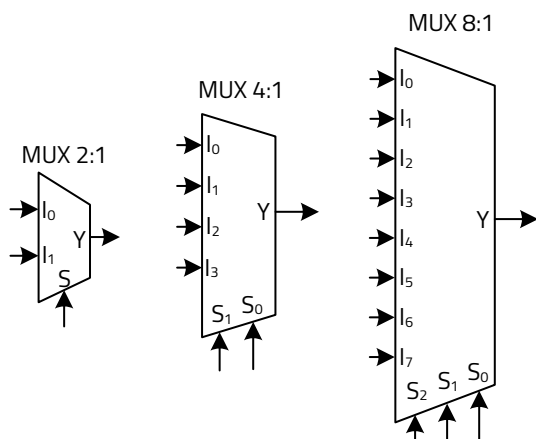
Simbolul generic al circuitului multiplexor  $2^N$  : 1 este prezentat în figura 2.23.

Ecuatiile funcțiilor logice implementate de MUX sunt:

$$\text{MUX 2:1} \quad Y = S \cdot I_1 + \bar{S} \cdot I_0$$

$$\text{MUX 4:1} \quad Y = S_1 \cdot S_0 \cdot I_3 + S_1 \cdot \bar{S}_0 \cdot I_2 + \bar{S}_1 \cdot S_0 \cdot I_1 + \bar{S}_1 \cdot \bar{S}_0 \cdot I_0$$

$$\begin{aligned} \text{MUX 8:1} \quad Y = & S_2 \cdot S_1 \cdot S_0 \cdot I_7 + S_2 \cdot S_1 \cdot \bar{S}_0 \cdot I_6 + \\ & + S_2 \cdot \bar{S}_1 \cdot S_0 \cdot I_5 + S_2 \cdot \bar{S}_1 \cdot \bar{S}_0 \cdot I_4 + \\ & + \bar{S}_2 \cdot S_1 \cdot S_0 \cdot I_3 + \bar{S}_2 \cdot S_1 \cdot \bar{S}_0 \cdot I_2 + \\ & + \bar{S}_2 \cdot \bar{S}_1 \cdot S_0 \cdot I_1 + \bar{S}_2 \cdot \bar{S}_1 \cdot \bar{S}_0 \cdot I_0 \end{aligned}$$



**Figura 2.22** Simboluri de multiplexoare. 2 : 1, 4 : 1 și 8 : 1.

Structura logică ale multiplexorului 2 : 1 se obține pe baza ecuației ieșirii. Prin aplicarea teoremei lui De Morgan, se poate înlocui structura AND-OR cu o structură mai rapidă NAND-NAND, ca în figura 2.24.

Structura logică ale multiplexorului 4 : 1 este prezentată în figura 2.25. Se observă că biții de selecție se decodifică și condiționează trecerea intrărilor de date prin porțile NAND.

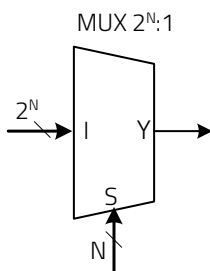
Modelul Verilog al multiplexorului 2 : 1 se poate descrie pe baza observației comportamentului acestuia:

- ieșirea este egală cu intrarea  $I_0$  dacă  $S = 0$
- ieșirea este egală cu intrarea  $I_1$  dacă  $S = 1$

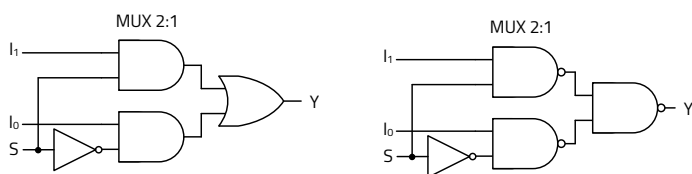
Acest comportament poate fi descris în Verilog fie cu specificație *if*, fie cu operatorul ternar.

```
module mux_2x1 (
input[1:0] I, // intrare de date
input      S, // intrare de selecție
output     Y  // iesire
);

assign Y = S ? I[1] : I[0]; // operator ternar
```



**Figura 2.23** Simbolul circuitului multiplexor  $2^N : 1$ .



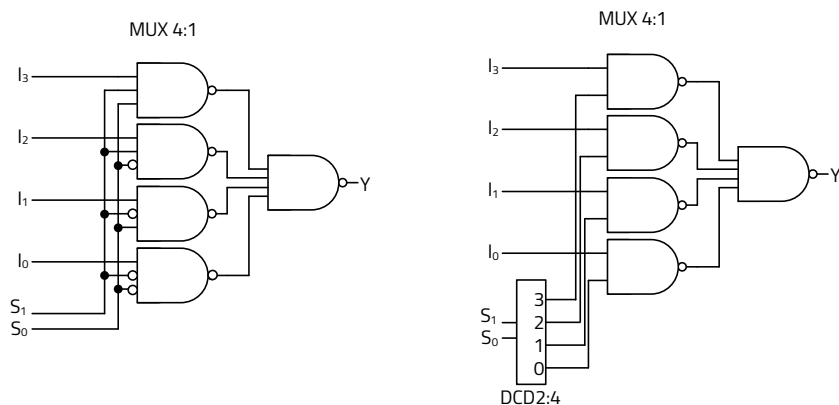
**Figura 2.24** Structuri de multiplexoare 2 : 1 (AND-OR și NAND-NAND).

```
endmodule // mux_2x1

module mux_2x1 (
  input  [2-1:0] I, // intrare de date (2 biti)
  input          S, // intrare de selectie
  output reg     Y  // iesire
);

always @(S or I)
  if (S) Y <= I[1]; else // specificatie de decizie
    Y <= I[0];

endmodule // mux_2x1
```



**Figura 2.25** Structură de multiplexor 4 : 1.

Pentru multiplexoare de dimensiuni mai mari, este convenabilă utilizarea specificației de selecție `case`. Trebuie remarcată necesitatea utilizării clauzei `default` pentru a asigura că ieșirea este atribuită în toate cazurile, ca la orice circuit logic combinațional. De exemplu, pentru un multiplexor 4 : 1, modelul Verilog pe baza specificației `case` se poate descrie astfel:

```
module mux_4x1 (
input  [4 -1:0]  I, // intrare de date
input  [2 -1:0]  S, // intrare de selecție
output reg      Y // iesire
);
```

```
always @(S or I)
case (S)
2'b11 : Y <= I[3];
2'b10 : Y <= I[2];
2'b01 : Y <= I[1];
default: Y <= I[0];
endcase
```

```
endmodule // mux_4x1
```

Codul poate fi scris mai condensat pe baza observării comportamentului: „circuitul prezintă la ieșire intrarea a cărei index este prezentat pe intrările de selecție”.

```
module mux_4x1 (
```

```

input [4 -1:0] I, // intrare de date
input [2 -1:0] S, // intrare de selectie
output      Y // iesire
);

assign Y = I[S];

endmodule // mux_4x1

```

Cu această observație, se poate scrie un modul Verilog pentru un multiplexor de dimensiune generică  $2^N : 1$  astfel:

```

module mux_generic #(
parameter N = 4
) (
input [(1<<N) -1:0] I, // intrare de date
input [N -1:0] S, // intrare de selectie
output      Y // iesire
);

assign Y = I[S];

endmodule // mux_generic

```

Se observă faptul că a fost descris comportamentul multiplexorului pe o singură linie. **Very-logic.**

```
assign Y = I[S];
```

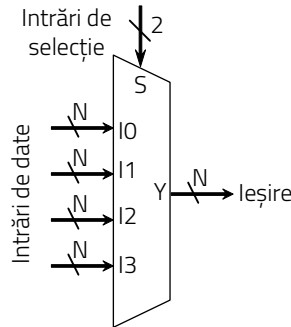
Dacă datele de intrare au mai mult de un bit, se pot conecta mai multe multiplexoare în paralel, selectate cu aceeași biți de selecție. Astfel, se obține un circuit care selectează date reprezentate pe mai mulți biți. De exemplu, în figura 2.26 este prezentat simbolul unui multiplexor 4 : 1 care are datele (și de intrare și de ieșire) pe mai mult de un bit.

Comportamentul unui multiplexor generic în care se parametrizează atât lățimea datelor cât și dimensiunea acestuia se poate descrie cu o singură linie în Verilog. În codul următor, s-a notat cu *SEL\_WIDTH* numărul de biți de selecție și cu *DATA\_WIDTH* numărul de biți ai datelor. Practic, se modelează un circuit format din *DATA\_WIDTH* multiplexoare  $2^{\text{SEL\_WIDTH}} : 1$ .

```

module mux_generic #(
parameter SEL_WIDTH = 4 , // biti de selectie
parameter DATA_WIDTH = 32 // biti de date
) (
input [DATA_WIDTH -1:0] I[(1<<SEL_WIDTH) -1:0], // date
input [SEL_WIDTH -1:0] S , // selectie
output [DATA_WIDTH -1:0] Y // iesire

```



**Figura 2.26** Simbol de MUX 4 : 1 cu datele de  $N$  biți.

```
);
assign Y = I[S];
endmodule // mux_generic
```

Se observă dimensiunea intrărilor multiplexorului:

- intrarea de selecție este un vector de  $SEL\_WIDTH$  biți
- ieșirea este un vector de  $DATA\_WIDTH$  biți
- intrarea de date este un vector bidimensional de dimensiune  $DATA\_WIDTH \times 2^{SEL\_WIDTH}$  biți, adică  $2^{SEL\_WIDTH}$  bus-uri a câte  $DATA\_WIDTH$  biți fiecare.

#### Rețetă

Orice funcție logică de  $N$  variabile poate fi implementată utilizând un MUX  $2^N : 1$ , fără minimizare, astfel:

- Pe cele  $N$  intrări de selecție ale multiplexorului se conectează cele  $N$  intrări ale funcției.
- Pe cele  $2^N$  intrări de date ale multiplexorului se conectează valorile logice 1 și 0, conform coloanei funcției din tabelul de adevăr. Dacă funcția este exprimată FCND, se poate utiliza regula: intrările de date ale multiplexorului ce își regăsesc indexul în FCND se conectează la 1, cele ce nu apar în FCND se conectează la 0.

Costul maxim al implementării oricărei funcții de  $N$  intrări este un MUX

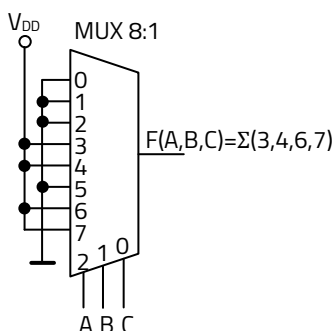
$2^N : 1$ .

### Exemplu

Implementați funcția  $F(A, B, C) = \sum(3, 4, 6, 7)$  utilizând un circuit multiplexor  $2^3 : 1$ .

### Soluție

Funcția  $F(A, B, C) = \sum(3, 4, 6, 7)$  are 3 intrări. Implementarea cu MUX  $8 : 1$  necesită conectarea celor 3 intrări pe intrările de selecție ale multiplexorului și a valorilor din coloana de adevăr a funcției pe intrările de date ale multiplexorului. Altfel spus, intrările de date ale multiplexorului care au indexul prezent în forma FCND a funcției se conectează la 1 iar cele ce nu apar în FCND se conectează la 0. Fără minimizare, rezultă implementarea prezentată în figura 2.27.



**Figura 2.27** Implementarea funcției  $F(A, B, C) = \sum(3, 4, 6, 7)$ , utilizând un circuit multiplexor  $8 : 1$ .

### Retetă

Orice funcție logică de  $N$  variabile poate fi implementată utilizând un MUX  $2^{N-1} : 1$  și un inversor, fără minimizare, astfel:

- Pe cele  $N - 1$  intrări de selecție ale multiplexorului se conectează cele mai semnificative  $N - 1$  intrări.
- Pe cele  $2^{N-1}$  intrări de date ale multiplexorului se conectează funcții reziduu ale celei mai puțin semnificative intrări. Funcțiile reziduu de o variabilă pot fi implementate cu maximum un inversor deoarece acestea

sunt:  $0, 1, X, \bar{X}$ . Funcțiile reziduu se pot determina din tabelul de adevăr astfel:

- Se delimitează în tabelul de adevăr grupuri de câte două rânduri consecutive.
- Se compară valorile de pe cele două rânduri, coloana variabilei reziduu (cea mai puțin semnificativă, aflată în partea dreaptă) și coloana funcției.
- Pot exista 4 cazuri:

| Intrare reziduu | Funcție | Funcție reziduu |
|-----------------|---------|-----------------|
| X               | F       |                 |
| 0               | 0       | 0               |
| 1               | 0       |                 |
| 0               | 0       | X               |
| 1               | 1       |                 |
| 0               | 1       | $\bar{X}$       |
| 1               | 0       |                 |
| 0               | 1       | 1               |
| 1               | 1       |                 |

Costul maxim total al implementării oricărei funcții de  $N$  intrări este un MUX  $2^{N-1} : 1$  și un inversor.

### Exemplu

Implementați funcția  $F(A, B, C) = \sum(3, 4, 6, 7)$  utilizând un circuit multiplexor  $4 : 1$  și un inversor.

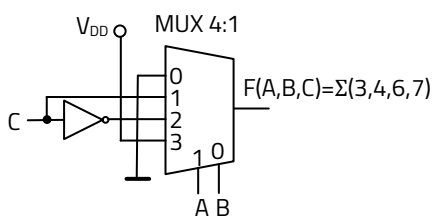
### Soluție

Funcția  $F(A, B, C) = \sum(3, 4, 6, 7)$  are 3 intrări. Implementarea cu MUX  $4 : 1$  necesită conectarea a două intrări (cele mai semnificative,  $A$  și  $B$ ) pe cele 2 intrări de selecție ale multiplexorului și a funcțiilor reziduu ale celei mai puțin semnificative intrări ( $C$ ) pe intrările de date. Funcțiile reziduu se deduc din tabelul următor:



| A | B | C | $F$ | Funcții reziduu      |
|---|---|---|-----|----------------------|
| 0 | 0 | 0 | 0   | $I_0 = 0$            |
| 0 | 0 | 1 | 0   |                      |
| 0 | 1 | 0 | 0   | $I_1 = C$            |
| 0 | 1 | 1 | 1   |                      |
| 1 | 0 | 0 | 1   | $I_2 = \overline{C}$ |
| 1 | 0 | 1 | 0   |                      |
| 1 | 1 | 0 | 1   | $I_3 = 1$            |
| 1 | 1 | 1 | 1   |                      |

Fără minimizare, rezultă implementarea prezentată în figura 2.28.



**Figura 2.28** Implementarea funcției  $F(A, B, C) = \sum(3, 4, 6, 7)$  utilizând un circuit multiplexor  $2^2 : 1$  și un inversor.



### Multiplexorul:

#### Maestrul selecției în lumea digitală

Multiplexorul, zis și „**mux**” printre cei grăbiți sau prieteni cu logica digitală, e un soi de chelner elegant care întreabă: „Care-i comanda ta?” și apoi servește, dintre mai multe intrări, exact pe cea dorită, în funcție de ce-i spun liniile de selecție. Nu judecă, nu insistă, nu face mofturi — doar alege. Dacă-i zici „001”, îți dă a doua intrare (că prima-i cerută cu „000”), dacă-i zici „111”, îți scoate ultima. Elegant, ordonat și fără bătaie de cap.

De aceea, în multe circuite și sisteme, se zice pe bună dreptate: **dacă mux nu e, nimic nu e...**

*Text generat cu ChatGPT.*

## 2.4.4 Demultiplexor

Demultiplexorul DMUX  $1 : 2^N$  implementează funcția inversă unui multiplexor, adică selectează pe baza unui cod de  $N$  biți una din cele  $2^N$  ieșiri de date pe care transmite intrarea de date.

Funcționarea demultiplexorului  $1 : 2$  poate fi descrisă astfel:

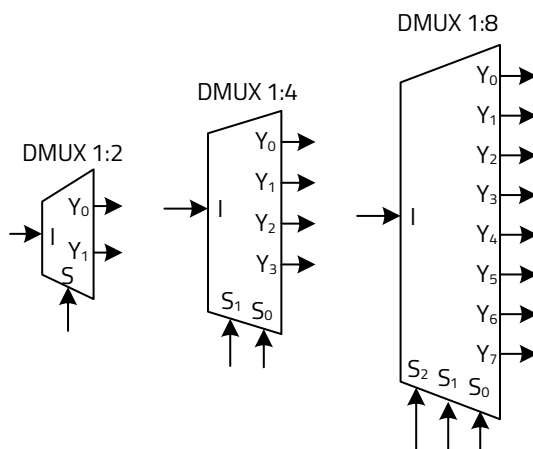
„Dacă selecția este egală cu 0, intrarea este transmisă la ieșirea  $Y_0$ , iar dacă selecția este egală cu 1, intrarea este transmisă la ieșirea  $Y_1$ ”.

Demultiplexorul  $2^N : 1$  are:

- $N$  intrări de selecție
- O intrare de date
- $2^N$  ieșiri de date

Ecuatiile funcțiilor logice implementate de DMUX sunt:

$$\begin{aligned} \text{DMUX } 1:2 \quad Y_1 &= S \cdot I \\ Y_0 &= \overline{S} \cdot I \end{aligned}$$



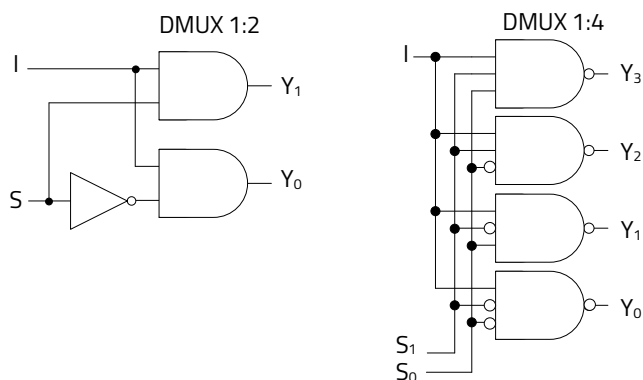
**Figura 2.29** Simboluri de demultiplexoare. 1:2, 1:4 și 1:8.

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMUX 1:4 | $Y_3 = S_1 \cdot S_0 \cdot I$ $Y_2 = \overline{S_1} \cdot S_0 \cdot I$ $Y_1 = \overline{S_1} \cdot \overline{S_0} \cdot I$ $Y_0 = S_1 \cdot \overline{S_0} \cdot I$                                                                                                                                                                                                                                                                                                 |
| DMUX 1:8 | $Y_7 = S_2 \cdot S_1 \cdot S_0 \cdot I$ $Y_6 = S_2 \cdot S_1 \cdot \overline{S_0} \cdot I$ $Y_5 = S_2 \cdot \overline{S_1} \cdot S_0 \cdot I$ $Y_4 = S_2 \cdot \overline{S_1} \cdot \overline{S_0} \cdot I$ $Y_3 = \overline{S_2} \cdot S_1 \cdot S_0 \cdot I$ $Y_2 = \overline{S_2} \cdot S_1 \cdot \overline{S_0} \cdot I$ $Y_1 = \overline{S_2} \cdot \overline{S_1} \cdot S_0 \cdot I$ $Y_0 = \overline{S_2} \cdot \overline{S_1} \cdot \overline{S_0} \cdot I$ |

Circuitele DMUX  $1 : 2^N$  au un comportament similar cu cel al unui decodificator de  $N$  biți cu intrare de validare. Dacă la un DMUX  $1 : 2^N$  se conectează intrarea  $I = 1$ , circuitul este echivalent cu un DCD  $N : 2^N$ .

Structurile logice ale demultiplexoarelor se obțin pe baza ecuațiilor.

Pe baza ecuațiilor ieșirilor, se poate modela în Verilog comportamentul demultiplexorului  $1 : 4$  cu specificații de atribuire continue.



**Figura 2.30** Structuri de demultiplexoare.

```

module dmux_1x4 (
input      I, // intrare de validare
input      [2-1:0] S, // intrare de selectie
output     [4-1:0] Y // iesiri
);

assign Y[3] = I & S[1] & S[0];
assign Y[2] = I & S[1] & ~S[0];
assign Y[1] = I & ~S[1] & S[0];
assign Y[0] = I & ~S[1] & ~S[0];

endmodule // dmux_1x4

```

Codul poate fi scris mai condensat pe baza observării comportamentului:

*„circuitul prezintă intrarea de date la ieșirea a cărei index este prezentat pe intrările de selecție”.*

```

module dmux_1x4 (
input      I, // intrare de validare
input      [2-1:0] S, // intrare de selectie
output     [4-1:0] Y // iesiri
);

assign Y = I << S;

```

```
endmodule // dmux_1x4
```

Cu această observație, se poate scrie un modul Verilog pentru un demultiplexor de dimensiune generică  $1 : 2^N$  astfel:

```
module dmux_generic #(
parameter WIDTH = 4           // 1 : 2^WIDTH
) (
input                               I, // intrare de validare
input [WIDTH-1:0] S, // intrare de selectie
output [(1<<WIDTH)-1:0] Y // iesiri
);

assign Y = I << S;

endmodule // dmux_generic
```

Se observă faptul că a fost descris comportamentul demultiplexorului pe o singură linie. *Very-logic.*

```
assign Y = I << S;
```

#### Rețetă

Orice funcție logică de  $N$  variabile poate fi implementată utilizând un DMUX  $1 : 2^N$ , fără minimizare, astfel:

- Pe cele  $N$  intrări de selecție ale demultiplexorului se conectează cele  $N$  intrări.
- Pe intrarea de date se conectează valoarea 1.
- Conform modului de conectare, demultiplexorul generează pe cele  $2^N$  ieșiri funcțiile logice ale tuturor celor  $2^N$  mintermini ai variabilelor de intrare. Implementarea necesită conectarea la intrările unei porți OR a ieșirilor demultiplexorului cu indexul egal cu indexul rândurilor cu valoare 1 în coloana de adevăr a funcției. Dacă funcția este exprimată FCND, se poate utiliza regula: ieșirile demultiplexorului ce își regăsesc indexul în FCND se conectează la intrarea unei porți OR.
- Dacă în expresia analitică a funcției exprimată FCND sunt prezenți mai mult de jumătate de mintermi ( $2^{N-1}$ ), se pot conecta la intrările unei porți NOR ieșirile demultiplexorului ale căror index nu apar în FCND.

Costul maxim total al implementării oricărei funcții de  $N$  intrări este un DMUX  $1 : 2^N$  și o poartă OR/NOR de maximum  $2^{N-1}$  intrări.

*Exemplu*

Implementați următorul set de funcții utilizând un demultiplexor 1 : 8.

$$F_1(A, B, C) = \sum(0, 2, 4)$$

$$F_2(A, B, C) = \sum(1, 3, 4, 7)$$

$$F_3(A, B, C) = \sum(0, 2, 4, 5, 6, 7)$$

*Soluție*

În implementarea prezentată în figura 2.31 se observă:

- Existența unui singur demultiplexor utilizat pentru implementarea tuturor funcțiilor.
- Dimensiunea demultiplexorului este dependentă de numărul de intrări ale funcțiilor. În acest caz sunt 3 intrări, deci demultiplexorul ales este 1 :  $2^3$ .
- Conectarea intrărilor funcțiilor pe intrările de selecție ale demultiplexorului în ordinea corectă. Astfel, la ieșirea demultiplexorului se obțin mintermii variabilelor de intrare.
- Conectarea unei porți OR suplimentare pentru fiecare funcție.
- Dacă un minterm apare în expresia mai multor funcții, ieșirea corespunzătoare a demultiplexorului se va conecta la intrările corespunzătoare mai multor porților OR (cazul  $m_4$ ).
- Dacă un minterm nu apare în expresia niciunei funcții, ieșirea corespunzătoare a demultiplexorului poate fi lăsată în gol (cazul  $m_5$  și  $m_6$ ).
- Dacă o funcție are în componență mai mult de jumătate din numărul total de mintermi aceasta se poate implementa negată, astfel încât poarta NOR să aibă un număr de intrări mai mic (egal cu maximum jumătate din numărul total de mintermi).

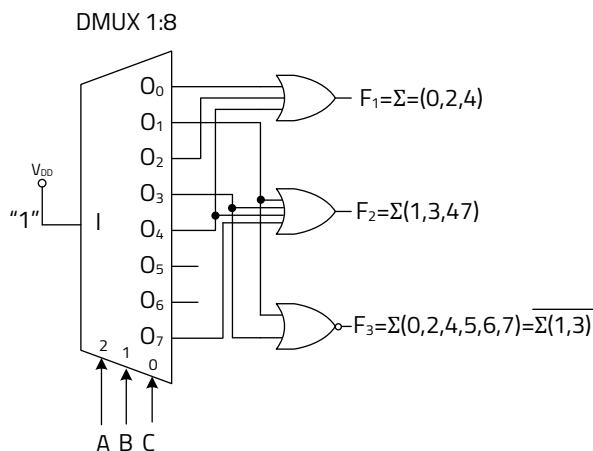
Este cazul funcției  $F_3(A, B, C) = \sum(0, 2, 4, 5, 6, 7)$ , implementată ca:

$$F_3(A, B, C) = \overline{\sum(1, 3)},$$

adică o poartă NOR cu  $8 - 6 = 2$  mintermi în loc de o poartă OR cu 6 mintermi prezenți în expresia funcției.

*Retetă*

O funcție logică de  $N$  variabile poate fi implementată pe un DMUX 1 :  $2^{N-1}$ , fără minimizare, dacă în expresia analitică o variabilă de intrare poate fi dată în factor (negată sau ne-negată).



**Figura 2.31** Implementarea unui set de 3 funcții cu un demultiplexor 1 : 8.

$$F(A, B, C, D) = A \cdot G(B, C, D) \text{ sau } F(A, B, C, D) = \overline{A} \cdot G(B, C, D).$$

Implementarea constă în conectarea intrării factorizate pe intrarea de date a demultiplexorului. Celelalte intrări se conectează pe intrările de selecție ale demultiplexorului  $1 : 2^{N-1}$ . Cu o poartă OR se vor conecta ieșirile necesare pentru realizarea funcției cu variabilele de selecție.

Costul maxim total al implementării oricărei funcții de  $N$  intrări ce îndeplinește condiția menționată este un DMUX  $1 : 2^{N-1}$  și o poartă OR/NOR de maximum  $2^{N-2}$  intrări (eventual un inversor pe intrarea de date).

### Exemplu

Implementați următoarea funcție de 4 variabile utilizând un demultiplexor 1 : 8.

$$F(A, B, C, D) = \sum(9, 10, 12, 15)$$

### Soluție

Prin prelucrări analitice, sau utilizând reprezentarea în diagrama V-K, se deduce că funcția are expresia echivalentă:

$$F(A, B, C, D) = A \cdot (B \oplus C \oplus D) = A \cdot G(B, C, D)$$

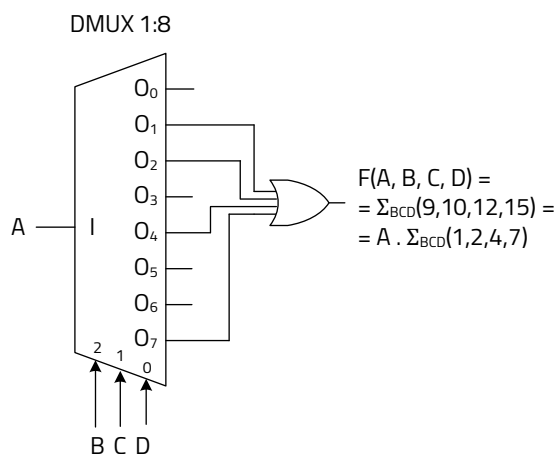
Unde s-a notat:

$$G(B, C, D) = B \oplus C \oplus D = \sum(1, 2, 4, 7), \text{ cu mintermiile variabilelor } (B, C, D).$$

De remarcat că:  $G(B, C, D) = F(1, B, C, D)$ , adică funcția  $F$  cu intrarea  $A$  conectată la valoarea 1.

Rezultă implementarea cu demultiplexor prezentată în figura 2.32.

- Variabila  $A$  conectată pe intrarea de date a demultiplexorului
- Variabilele  $B, C, D$  conectate pe intrările de selecție ale demultiplexorului
- Ieșirile demultiplexorului 1, 2, 4, 7 conectate printr-o poartă OR.
- Ieșirea porții OR reprezintă funcția  $F(A, B, C, D)$ .



**Figura 2.32** Implementarea unei funcții particulare  $F(A, B, C, D) = A \cdot G(B, C, D)$  cu un demultiplexor 1 : 8.



**Demultiplexorul:****Mesagerul care împarte porția**

Demultiplexorul, sau „*demux*” pe scurt, e fratele mai organizat și mai atent al multiplexorului. Dacă multiplexorul alege cu eleganță dintr-o multime, demux-ul face exact opusul: ia o singură intrare și o împarte în mai multe părți, cu o precizie de ceas elvețian. Practic, îți dă fiecare bucată acolo unde trebuie, fără să facă greșeli. Dacă-i dai semnalul potrivit, se va asigura că semnalul ajunge exact la ieșirea corespunzătoare, nimic nu rămâne neîmpărțit.

Iar când demux-ul face ce știe el mai bine, circulă logica fără întrerupere, căci **dacă demux nu e, nimic nu ajunge la locul potrivit.**

*Text generat cu ChatGPT.*

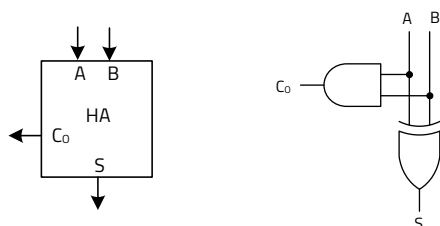
### 2.4.5 Sumator

Circuitele logice combinaționale implementează funcții logice. Însă funcțiile logice pot fi interpretate ca funcții aritmetice aplicate asupra unor date binare ce codifică valori numerice.

**Semi-sumator de 1 bit**

Circuitul semi-sumator de 1 bit (eng. *half-adder*, prescurtat „HA”) este un circuit logic combinațional cu două intrări și două ieșiri ce poate fi considerat ca având funcția aritmetică de a însuma 2 biți (operandii) și a genera rezultatul pe 2 biți: un bit ( $S$ ) și transportul la bitul de ordin superior ( $C_o$ ). De menționat că la adunarea celor 2 biți nu se ține cont de transport de intrare.

Tabelul de adevăr ale semi-sumatorului de 1 bit este următorul.



**Figura 2.33** Semi-sumator de 1 bit: simbol bloc și structura de porți logice.

| Intrări |   | Ieșiri |   | Sumă      |
|---------|---|--------|---|-----------|
| A       | B | $C_o$  | S | (baza 10) |
| 0       | 0 | 0      | 0 | 0         |
| 0       | 1 | 0      | 1 | 1         |
| 1       | 0 | 0      | 1 | 1         |
| 1       | 1 | 1      | 0 | 2         |

Se deduc ecuațiile ieșirilor:

$$S = (A \cdot \overline{B}) + (\overline{A} \cdot B) = A \oplus B$$

$$C_o = A \cdot B$$

Modelul Verilog al semisumatorului se poate descrie utilizând operatorul Verilog de adunare *plus* (a nu se confunda cu simbolul operatorului OR). Suma celor două intrări  $A + B$ , în domeniul  $0 \dots 2$ , se atribuie concatenării celor două ieșiri  $C_o$  și  $S$  ( $C_o$  este cel mai semnificativ bit al rezultatului, transportul).

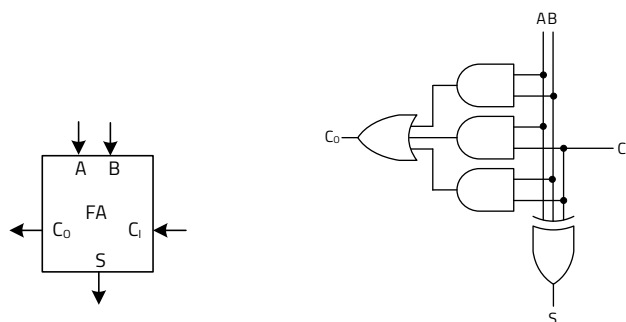
```
assign {Co, S} = A + B; // A plus B
```

Desigur, se pot descrie funcțiile pe baza ecuațiilor logice. Însă, considerând interpretarea aritmetică a operației, modelul este mai intuitiv.

**Very-logic.**

### Sumator complet de 1 bit

Circuitul sumator complet de 1 bit este un circuit logic combinațional cu trei intrări și două ieșiri ce poate fi considerat ca având funcția aritmetică de a însuma 2 biți (operandii) plus transportul de la bitul inferior ( $C_i$ ) și a genera rezultatul pe 2 biți: un bit ( $S$ ) și transportul la bitul de ordin superior ( $C_o$ ).



**Figura 2.34** Sumatorul complet de 1 bit: simbol bloc și structura de porți logice

Tabelul de adevăr ale sumatorului complet de 1 bit este următorul.

| Intrări |   |       | Ieșiri |   | Sumă<br>(baza 10) |
|---------|---|-------|--------|---|-------------------|
| A       | B | $C_i$ | $C_o$  | S |                   |
| 0       | 0 | 0     | 0      | 0 | 0                 |
| 0       | 0 | 1     | 0      | 1 | 1                 |
| 0       | 1 | 0     | 0      | 1 | 1                 |
| 0       | 1 | 1     | 1      | 0 | 2                 |
| 1       | 0 | 0     | 0      | 1 | 1                 |
| 1       | 0 | 1     | 1      | 0 | 2                 |
| 1       | 1 | 0     | 1      | 0 | 2                 |
| 1       | 1 | 1     | 1      | 1 | 3                 |

Se deduc ecuațiile ieșirilor:

$$S = A \oplus B \oplus C_i$$

$$C_o = A \cdot B + A \cdot C_i + B \cdot C_i$$

Se observă că sumatorul complet de 1 bit adună 3 biți cu ponderi egale ( $A$ ,  $B$  și  $C_i$ ). Rezultatul sumei, în domeniul 0...3, este reprezentat pe 2 biți:  $S$  cu pondere  $2^0$  și  $C_o$  cu pondere  $2^1$ .

Modelul Verilog al sumatorului complet se poate descrie utilizând operatorul Verilog de adunare *plus* (a nu se confunda cu simbolul operatorului OR). Suma celor două intrări plus transportul de intrare  $A + B + C_i$  se atribuie concatenării celor două ieșiri  $C_o$  și  $S$  ( $C_o$  este cel mai semnificativ bit al rezultatului, transportul).

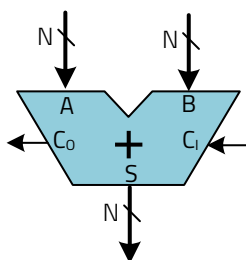
```
assign {Co, S} = A + B + Ci; // A plus B plus Ci
```

Desigur, se pot descrie funcțiile pe baza ecuațiilor logice. Însă, considerând interpretarea aritmetică a operației, modelul este mai intuitiv.

*Very-logic.*

### Sumator de mai mulți biți

Circuitul sumator de  $N$  biți adună două numere reprezentate pe câte  $N$  biți, cu considerarea transportului de la bitul de ordin inferior. Suma este reprezentată pe  $N$  biți și un bit suplimentar de transport spre biții de ordin superior.

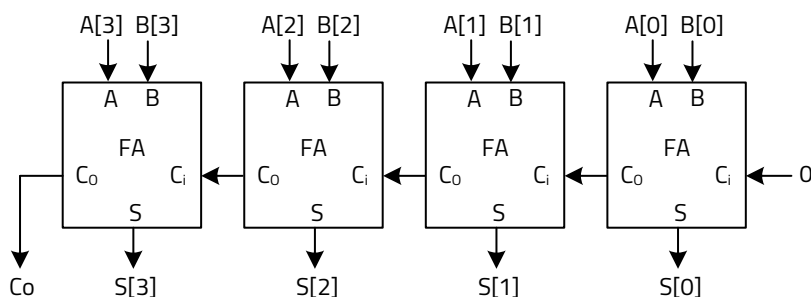


**Figura 2.35** Simbol sumator de  $N$  biți.

Dacă nu se consideră transportul de intrare, se poate spune că sumatorul de  $N$  biți adună două numere pozitive reprezentate pe câte  $N$  biți și generează suma intrărilor, reprezentată pe  $N + 1$  biți.

Ca structură, sumatorul de  $N$  biți se poate realiza prin utilizarea a  $N$  sumatoare complete de 1 bit conectate astfel încât transportul să se propage de la bitul mai puțin semnificativ spre cel mai semnificativ, ca în

figura 2.36. Se observă faptul că sumatorul bitului 0 poate fi redus la un semisumator (deoarece are transportul de intrare legat la 0).



**Figura 2.36** Structura sumatorului de 4 biți, „ripple-carry”.

Această structură este denumită în limba engleză „ripple carry”. Este intuitivă, dar lentă: timpul de propagare al sumatorului de  $N$  biți este egal cu de  $N$  ori timpul de propagare al unui sumator complet de 1 bit. Au fost inventate **structuri mai rapide de sumare**, care țin cont de faptul că transportul oricărui bit nu poate fi decât ori 1 ori 0.

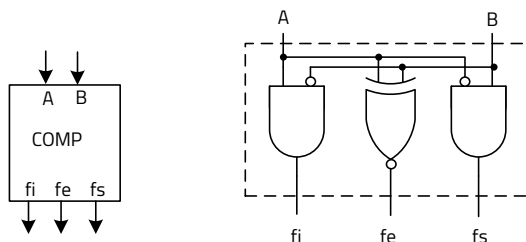
Modelul Verilog al sumatorului de  $N$  biți se poate descrie utilizând operatorul Verilog de adunare *plus*. Suma celor două intrări plus transportul de intrare  $A[N-1:0] + B[N-1:0] + C_i$  se atribuie concatenării celor două ieșiri  $C_o$  și  $S[N-1:0]$ .  $C_o$  este cel mai semnificativ bit al rezultatului, transportul.

```
wire [N-1:0] A ;
wire [N-1:0] B ;
wire Ci ;
wire [N-1:0] S ;
wire Co ;

assign {Co, S} = A + B + Ci; // A[N-1:0] plus B[N-1:0] plus Ci
```

### 2.4.6 Comparator

Comparatorul de 1 bit este un circuit logic combinațional ce primește la intrare două valori de câte 1 bit și generează informații despre relația de ordine: mai mic, egal sau mai mare.



**Figura 2.37** Comparator de 1 bit: simbol și structură de porți logice

Tabelul de adevăr ale comparatorului de 1 bit este următorul.

| Intrări |   | Ieșiri |    |    |
|---------|---|--------|----|----|
| A       | B | fi     | fe | fs |
| 0       | 0 | 0      | 1  | 0  |
| 0       | 1 | 1      | 0  | 0  |
| 1       | 0 | 0      | 0  | 1  |
| 1       | 1 | 0      | 1  | 0  |

Se deduc ecuațiile ieșirilor:

$$\begin{aligned}
 f_i &= A \cdot \overline{B}, \text{ „A inferior lui B”} \\
 f_e &= \overline{A \oplus B}, \text{ „A egal cu B”} \\
 f_s &= \overline{A} \cdot B, \text{ „A superior lui B”}
 \end{aligned}$$

Modelarea Verilog al funcțiilor comparatorului de 1 bit poate fi realizată cu operatori logici:

```

assign fs = A & ~B ;
assign fe = ~(A ^ B) ;
assign fi = ~A & B ;

```

Desigur, în practică se dorește frecvent compararea a două valori numerice reprezentate pe mai mulți biți. În cazul unor numere pozitive, compararea unor valori reprezentate pe mai mulți biți se face începând cu bitul cel mai semnificativ. Dacă decizia de comparare este luată prin compararea unor biți mai semnificativi, biții mai puțin semnificativi sunt ignorați. Deși pentru proiectantul uman funcția de comparare pe bit nu este grea (se compară bit cu bit de la stânga la dreapta) în hardware comparatoarele necesită implementarea tuturor combinațiilor de intrare posibile. Astfel,

un comparator de 8 biți are  $8 + 8 = 16$  intrări, ceea ce îl face impractic de reprezentat cu formele clasice de reprezentare (tabele logice, diagrame V-K, funcții logice).

Limbajul Verilog are operatori utilizați pentru compararea numerelor pozitive care permit modelarea ușoară a funcțiilor de comparare a vectorilor binari.

Operatorii Verilog relaționali sunt prezentați în tabelul 2.1.

Utilizând operatorii relaționali, funcțiile de comparare se pot exprima în Verilog foarte condensat.

*Very-logic.*

```
wire [N-1:0] A ;
wire [N-1:0] B ;
wire         Fi ;
wire         Fe ;
wire         Fs ;

assign Fi = (A < B) ; // Fi=1 daca A<B
assign Fe = (A == B) ; // Fe=1 daca A=B
assign Fs = (A > B) ; // Fs=1 daca A>B
```

De observat posibila confuzie între operatorul de atribuire continuă = (un singur caracter =) și operatorul relațional de egalitate == (două caractere =). În Verilog există și un operator === (trei caractere =).

### 2.4.7 Unitate logică și aritmetică

O unitate logică și aritmetică (*Arithmetic Logic Unit*, engl.), prescurtat „*ALU*”, este un circuit logic combinațional care poate fi comandat să efectueze mai multe operații asupra operanzilor. Operațiile unităților de procesare nu sunt standardizate, ci se proiectează în funcție de necesitățile fiecărei aplicații. Dimensiunea operanzilor, exprimată în biți, dă o informație despre complexitatea calculelor realizate de către unitățile logico-aritmetice.

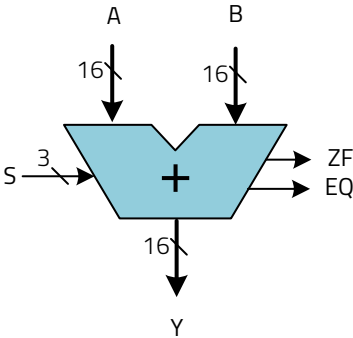
Figura 2.38 prezintă simbolul bloc a unei ALU pe 16 biți care realizează cele 8 operații menționate în tabelul următor.

| Operator | Operație          | Exemple                                                                          |
|----------|-------------------|----------------------------------------------------------------------------------|
| >        | Mai mare          | 10 > 01 Adevărat<br>10 > 10 Fals<br>10 > 11 Fals                                 |
| <        | Mai mic           | 10 < 01 Fals<br>10 < 10 Fals<br>10 < 11 Adevărat                                 |
| >=       | Mai mare sau egal | 10 >= 01 Adevărat<br>10 >= 10 Adevărat<br>10 >= 11 Fals                          |
| <=       | Mai mic sau egal  | 10 <= 01 Fals<br>10 <= 10 Adevărat<br>10 <= 11 Adevărat                          |
| ==       | Egal              | 01 == 01 Adevărat<br>01 == 00 Fals<br>01 == x1 Adevărat<br>01 == z1 Adevărat     |
| !=       | Diferit           | 01 != 01 Fals<br>01 != 00 Adevărat<br>01 != x1 Fals<br>01 != z1 Fals             |
| ===      | Egal (x, z)       | 01 === 01 Adevărat<br>01 === 00 Fals<br>01 === x1 Fals<br>01 === z1 Fals         |
| !==      | Diferit(x, z)     | 01 !== 01 Fals<br>01 !== 00 Adevărat<br>01 !== x1 Adevărat<br>01 !== z1 Adevărat |

Tabelul 2.1

Tabelul operatorilor relaționali Verilog.





**Figura 2.38** Simbol bloc ALU de 16 biți.

| $S_2$               | $S_1$ | $S_0$ | $Y$          | Operație       |
|---------------------|-------|-------|--------------|----------------|
| Operații logice     |       |       |              |                |
| 0                   | 0     | 1     | $A \oplus B$ | XOR pe bus     |
| 0                   | 1     | 0     | $A + B$      | OR pe bus      |
| 0                   | 1     | 1     | $A \cdot B$  | AND pe bus     |
| Operații aritmetice |       |       |              |                |
| 1                   | 0     | 0     | $A - 1$      | decrementare A |
| 1                   | 0     | 1     | $A + 1$      | incrementare A |
| 1                   | 1     | 0     | $A - B$      | diferența      |
| 1                   | 1     | 1     | $A + B$      | suma           |
| Fără prelucrare     |       |       |              |                |
| 0                   | 0     | 0     | $A$          | trece A        |

Modelul Verilog reproduce tabelul cu operațiile implementate utilizând operatorii specifici limbajului. Modelul se bazează pe specificația de selecție case pe baza codului operației  $S$ . În funcție de operația selectată, ieșirea  $Y$  primește valorile determinate de operație asupra operanzilor  $A$  și  $B$ .

Indicatorii sunt modelați cu operatorul relațional de egalitate:

- $ZF = 1$ , dacă rezultatul operației este zero,  $Y = 0$ .
- $EQ = 1$ , dacă operanzii sunt egali,  $A = B$ .

```
module alu #(
```

```

parameter WIDTH = 16          // latime operanzi (in biti)
) (
input      [WIDTH-1:0] A , // operanzi
input      [WIDTH-1:0] B ,
input      [3      -1:0] S , // cod operatie
output reg [WIDTH-1:0] Y , // iesire/rezultat
output      ZF, // indicator rezultat egal cu zero
output      EQ  // indicator operanzi egali
);

// prelucrare operanzi
always @(A or B or S)
case (S)
// operatii logice
3'b001 : Y <= A ^ B; // A xor B
3'b010 : Y <= A | B; // A or B
3'b011 : Y <= A & B; // A and B
// operatii aritmetice
3'b100 : Y <= A - 1; // decrementare A-1
3'b101 : Y <= A + 1; // incrementare A+1
3'b110 : Y <= A - B; // diferenta A-B
3'b111 : Y <= A + B; // suma A+B
// fara prelucrare
default: Y <= A      ; // trece A, S=000
endcase

// indicatori
assign ZF = (Y == 'b0) ; // Zero Flag
assign EQ = (A == B)   ; // Equal

endmodule // alu

```

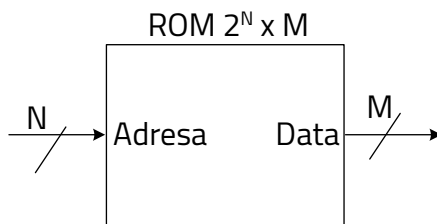
Codul prezentat este transformat într-o structură de porți logice, în mod automat, de către o aplicație software EDA de sinteză de hardware.

## 2.4.8 Look-Up-Table și memorii ROM

*Memoria ROM (Read Only Memory, engl.)* este un circuit logic combinațional realizat ca un dispozitiv programabil, având un nivel de porți logice AND fix și complet (în interiorul structurii se generează toți mintermiile asociați intrărilor) și un nivel de porți logice OR configurabil. Intrările în ROM sunt considerate *adresele* iar ieșirile sunt considerate *datele* citite din memorie. Întreaga memorie poate fi văzută ca fiind o matrice în care datele sunt stocate în locații, pe baza unei adrese.

Simbolul bloc al unei memorii ROM  $2^N \times M$  este prezentat în figura 2.39. S-au notat cu  $N$  numărul de biți de adresă (în total  $2^N$  adrese) și cu  $M$

numărul de biți de date.



**Figura 2.39** Simbolul bloc al memoriei ROM  $2^N \times M$ .

Implementarea unui circuit logic combinațional cu ROM nu necesită minimizarea funcțiilor. Intrările sunt asociate cu biții de adresă ai circuitului ROM iar ieșirile circuitului combinațional se obțin de la datele memoriei ROM. Implementarea constă în realizarea schemei de conectare a memoriei ROM (asocierea dintre intrări și porturile de adrese și asocierea dintre ieșiri și porturile de date) și a tabelului de programare a acestora (adrese, date).

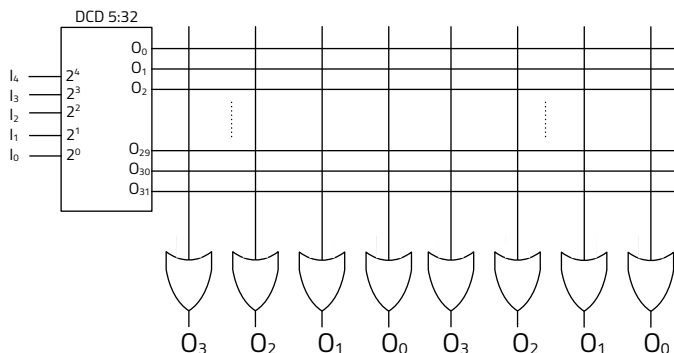
Structura ROM prezintă primul nivel AND complet, implementat sub forma unui decodificator care generează toți mintermii asociați intrărilor (figura 2.40). Nivelul OR conține un număr de porți programabile egal cu numărul de ieșiri. Pe un circuit ROM  $2^N \times M$  pot fi implementate orice  $M$  funcții care depind de aceleași  $N$  intrări.

Modelul Verilog al memoriei ROM conține o secțiune de inițializare a datelor dintr-un fișier în format binar sau hexazecimal într-o variabilă bidimensională. Variabila bidimensională funcționează ca un tabel din care, pe baza adresei, se deduce conținutul locației selectate.

```
module rom #(
parameter ADDRESS_BITS = 4 , // latime adresa (biti)
parameter DATA_BITS   = 8   // latime date (biti)
) (
input  [ADDRESS_BITS-1:0] address , // adresa
output [DATA_BITS-1:0] date       // data

// datele memorate in tabel (variabila bidimensională)
reg [DATA_BITS-1:0] mem_data [(1<<ADDRESS_BITS)-1:0];

// citire memorie (din variabila)
assign date = mem_data[address] ;
```



**Figura 2.40** Structura ROM  $2^5 \times 8$ . Simbolul porții OR reprezintă o poartă logică cu 32 de intrări, cu conexiuni programabile de la ieșirile decodicatorului de adrese.

```
// initializare continut memorie din fisier
initial begin
    $readmemb("rom_image.bin", mem_data); // harta binara
    // $readmemh("rom_image.hex", mem_data); // harta hexazecimal
end

endmodule // rom
```

Memoria ROM (accesibilă exclusiv pentru citire în cursul funcționării normale a sistemului) stă la baza implementării unei structuri de tip tabel (denumită în limba engleză „*Look-Up-Table*”, prescurtat „*LUT*”). De obicei, structurile de tip LUT sunt de mici dimensiuni și pot fi modelate pe baza unei specificații case în care expresia de selecție este reprezentată de adresă iar valorile se atribuie către ieșirea de date.

Ca exemplificare, se prezintă un model de LUT pentru memorarea codurilor ASCII necesare afișării textului „*Electronica Digitala*”.

```
module lut_32x8 (
input    [5   -1:0] char_index , // adresa
output   [8   -1:0] ascii_char  // data
);

always @(*)
case (char_index)
4'd0   :  ascii_char = 8'b0100_0101 ;    // E
4'd1   :  ascii_char = 8'b0110_1100 ;    // l
```

```

4'd2   :  ascii_char = 8'b0110_0101 ;    // e
4'd3   :  ascii_char = 8'b0110_0011 ;    // c
4'd4   :  ascii_char = 8'b0111_0100 ;    // t
4'd5   :  ascii_char = 8'b0111_0010 ;    // r
4'd6   :  ascii_char = 8'b0110_1111 ;    // o
4'd7   :  ascii_char = 8'b0110_1110 ;    // n
4'd8   :  ascii_char = 8'b0110_1001 ;    // i
4'd9   :  ascii_char = 8'b0110_0011 ;    // c
4'd10  :  ascii_char = 8'b0110_0001 ;    // a
4'd11  :  ascii_char = 8'b0010_0000 ;    //
4'd12  :  ascii_char = 8'b0100_0100 ;    // D
4'd13  :  ascii_char = 8'b0110_1001 ;    // i
4'd14  :  ascii_char = 8'b0110_0111 ;    // g
4'd15  :  ascii_char = 8'b0110_1001 ;    // i
4'd16  :  ascii_char = 8'b0111_0100 ;    // t
4'd17  :  ascii_char = 8'b0110_0001 ;    // a
4'd18  :  ascii_char = 8'b0110_1100 ;    // l
4'd19  :  ascii_char = 8'b0110_0001 ;    // a
default:  ascii_char = 8'h0010_0000 ;    // spatiu
endcase

endmodule // lut_32x8

```

Modelul de LUT va genera, prin sinteză, o structură ROM  $2^5 \times 8$  ce are pe adrese portul *char\_index* și pe date portul *ascii\_char*, conținutul fiind cel precizat în specificația *case*.

## 2.5 Modelarea circuitelor logice combinaționale în HDL

### 2.5.1 Operatori

În funcție de numărul de operanzi asupra cărora se aplică, operatorii Verilog sunt de 3 tipuri:

- **Unari**, se aplică unui singur operand, apar în fața operandului.

```
assign a = ~b ; // a = not(b)
```

- **Binari**, se aplică la doi operanzi, apar între cei doi operanzi.

```
assign a = b + c ; // a = b + c (plus)
```

- **Ternari**, se aplică la trei operanzi, separă cei trei operanzi.

```
assign a = b ? c : d ; // daca b=1, atunci a=c
                        // altfel          a=d
```

În funcție de efectul operatorilor, aceștia se încadrează în următoarele categorii:

- **Operatori aritmetici**, se aplică asupra unor date de mai mulți biți care reprezintă valori numerice și au ca rezultat valori numerice. Numerele negative sunt considerate în complement față de 2. Atenție la numărul de biți pe care sunt reprezentate valorile și la posibilele depășiri în urma operațiilor aritmetice.
- **Operatori relaționali**, se aplică la doi operanzi interpretați numeric sau ca vectori de biți. Atenție la relațiile dintre valorile 0 și 1 în raport cu celelalte valori posibile  $x$  și  $z$ . Rezultatul este o valoare de adevăr (fals sau adevărat) codată pe un bit (0 sau 1).
- **Operatori logici**, se aplică asupra unor date logice și au ca rezultat o valoare de adevăr (fals sau adevărat) codată pe un bit (0 sau 1). Dacă operandii sunt pe mai mult biți, asupra acestora se aplică reguli prin care se reduc la un bit, înainte de aplicarea operatorului logic. Regula este: dacă operandul are toți biții egali cu 0, atunci se consideră că are valoarea logică 0. Altfel, dacă există cel puțin un bit egal cu 1, se consideră că are valoarea logică 1. Atenție, asta înseamnă că pentru acest tip de operatori, presupunând valori pe 4 biți,  $4 = 4'b0100 = 1$ ,  $6 = 6'b0110 = 1$  iar singura valoare egală cu 0 este  $4'b0000 = 0$ . Există reguli pentru tratarea valorilor  $x$  și  $z$ .
- **Operatori logici pe vectori**, se aplică asupra unor vectori de date (de dimensiuni egale) și au ca rezultat un vector de aceeași dimensiune. În caz că cei doi operanzi nu au aceeași dimensiune, există reguli ale limbajului Verilog în care operandul mai scurt se mărește la dimensiunea operandului mai lung. Operația logică se aplică „bit cu bit”, un bit de rezultat fiind dependent numai de biții corespunzători ai celor doi operanzi și operatorul logic.
- **Operatori de reducere**, sunt operatori unari care se aplică asupra unor vectori de dimensiune neprecizată și au ca rezultat un bit.
- **Operatori de deplasare**, sunt operatori binari. Unul din operanzi reprezintă un vector de biți, iar celălalt operand reprezintă o valoare pozitivă cu care primul operand se deplasează, în stânga sau în dreapta. Există reguli pentru valorile biților rămași în urma deplasărilor (de exemplu, valoarea celui mai semnificativ bit în cazul deplasării la dreapta a biților primului operand).

În continuare se descrie fiecare categorie de operanzi, cu exemple care să illustreze modurile de utilizare ale acestora.

## 2.5.2 Operatori aritmetici

Operatorii aritmetici sunt:

|          |                                                                                                     |
|----------|-----------------------------------------------------------------------------------------------------|
| $+a$     | operator unar semnificând valoarea pozitivă a valorii întregi, cu semn, $a$                         |
| $-a$     | operator unar semnificând valoarea negativă a valorii întregi, cu semn, $a$                         |
| $a + b$  | operator binar de adunare, „ $a$ plus $b$ ”                                                         |
| $a - b$  | operator binar de scădere, „ $a$ minus $b$ ”                                                        |
| $a * b$  | operator binar de multiplicare, „ $a$ înmulțit cu $b$ ”                                             |
| $a/b$    | operator binar de împărțire întreagă, „câtul împărțirii întregi $a$ lui $a$ la $b$ ”                |
| $a \% b$ | operator binar de rest la împărțire întreagă, „ $a$ modulo $b$ , restul împărțirii lui $a$ la $b$ ” |
| $a ** b$ | operator binar de ridicare la putere, „ $a$ la puterea $b$ ”                                        |

Trebuie remarcat faptul că, pentru modelarea funcțiilor ce vor fi implementate în hardware, operatorii aritmetici trebuie utilizați cu **precauție**, deoarece nu toate opțiunile sunt sintetizabile (nu au echivalent în hardware). Programele de sinteză de hardware se îmbunătățesc continuu: ceea ce nu se putea acum 10 ani, azi se poate. La fel, ceea ce azi nu este suportat, ar putea fi peste câțiva ani. Recomandarea este de a se verifica faptul că un anumit operator este înțeles și sintetizat corect de către aplicația de sinteză de hardware.

Operatorii de adunare și scădere,  $+$  și  $-$ , sunt sintetizabili. Atenție însă la reprezentarea numerelor negative. Operatorul multiplicare  $*$  s-a evitat a fi folosit mulți ani, însă, în prezent, este sintetizat ca un circuit combinațional de către sintetizatoare.

Operatorii de împărțire întreagă (cât  $/$  și rest  $\%$ ) se recomandă a nu se folosi în cod ce urmează a fi sintetizat. Sau, a se folosi cu multă grijă...

Pentru un om este evidentă intenția proiectantului în specificația:

```
wire [8 -1:0] a ;
wire [7 -1:0] b ;
assign b = a / 3 ; // catul impartirii lui a la 3
```

Însă, nu este evident pentru sintetizator, care trebuie să convertească operatorul  $/$  într-un circuit logic combinațional. Deci, cu multă grijă...

Similar, pentru operatorul de determinare a restului unei împărțiri de numere, modulo  $\%$ :

```

wire [8 -1:0] a ;
wire [2 -1:0] b ;
assign b = a % 3 ; // restul impartirii lui a la 3
                  // valori posibile ale rezultatului: 0, 1, 2

```

O atenție sporită dacă deîmpărțitul este considerat a fi un număr negativ.

Operatorul de ridicare la putere `**` a apărut în versiuni ulterioare ale limbajului Verilog. Dacă puterea este 2, este echivalent cu o înmulțire  $a ** 2 = a * a$ . Altfel, a se folosi cu multă grijă...

În cazurile menționate, recomandarea este de a testa suportarea operatorului de către sintetizator. Însă, având în vedere faptul că de multe ori se livrează clienților cod Verilog pentru a fi sintetizat de către aceștia, iar clienților nu li se poate impune un anumit sintetizator, este recomandată evitarea utilizării operatorilor aritmetici nesuportați de toate sintetizatoarele și remodelarea cu expresii mai complexe dar cu operatori suportați.

Dimensiunea rezultatului operațiilor aritmetice depinde de dimensiunea celui mai mare operand și de dimensiunea declarată a obiectului destinație. Dacă se atribuie unui obiect un rezultat pe mai puțini biți, atunci rezultatul se extinde cu zerouri. Dacă se atribuie unui obiect un rezultat pe mai mulți biți, atunci rezultatul se trunchiază la numărul de biți ai obiectului destinație.

### Exemplu

Se consideră următorul cod Verilog pentru exemplificarea rezultatelor operatorilor aritmetici.

```

module test_op_aritm;

reg[4 -1:0]    opA, opB, opC;
reg[5 -1:0]    opD;
reg[6 -1:0]    opE;
reg[8 -1:0]    opF;
integer       opG;
reg[8 -1:0]    opH;

initial begin
opA = 4'b1110; // 14
opB = 4'b1000; // 8
opH = +9;      // +9

$display("#####");
// Operator unar
opF = +opH;    // opH = +9
$display("Operator unar opH=%b.", opH);
$display("+ Binar: %b", opF);

```



```

$display(" Zecimal: %0d", opF);
$display(" Intreg: %0d", $signed(opF));

opF = -opH;
$display("- Binar: %b", opF);
$display(" Zecimal: %0d", opF);
$display(" Intreg: %0d", $signed(opF));

opH = -9;          // -9
// Operator unar
opF = +opH;        // opH = -9
$display("Operator unar opH=%b.", opH);
$display("+ Binar: %b", opF);
$display(" Zecimal: %0d", opF);
$display(" Intreg: %0d", $signed(opF));

opF = -opH;
$display("- Binar: %b", opF);
$display(" Zecimal: %0d", opF);
$display(" Intreg: %0d", $signed(opF));

$display("#####");
// Suma numere pozitive 4b+4b=4b, cu depasire
opC = opA + opB; // suma evaluata pe 4 biti
$display("Suma numere pozitive 4b+4b=4b. Asteptat: 14 + 8 = 22");
$display(" Binar: %b + %b = %b", opA, opB, opC);
$display(" Zecimal: %0d + %0d = %0d", opA, opB, opC);

// Suma numere pozitive 4b+4b=5b, fara depasire
opD = opA + opB; // suma evaluata pe 5 biti
$display("Suma numere pozitive 4b+4b=5b. Asteptat: 14 + 8 = 22");
$display(" Binar: %b + %b = %b", opA, opB, opD);
$display(" Zecimal: %0d + %0d = %0d", opA, opB, opD);

// Suma numere pozitive 4b+4b=6b, fara depasire, 0 la MSb
opE = opA + opB; // suma evaluata pe 6 biti
$display("Suma numere pozitive 4b+4b=6b. Asteptat: 14 + 8 = 22");
$display(" Binar: %b + %b = %b", opA, opB, opE);
$display(" Zecimal: %0d + %0d = %0d", opA, opB, opE);

$display("#####");
// Multiplicare pozitive 4b*4b=8b
opF = opA * opB; // multiplicare evaluata la 8 biti
$display("Multiplicare 4b*4b=8b. Asteptat: 14 * 8 = 112");
$display(" Binar: %b * %b = %b", opA, opB, opF);
$display(" Zecimal: %0d * %0d = %0d", opA, opB, opF);

$display("#####");
// Diferenta numere pozitive 4b-4b=4b

```

```

opC = opB - opA; // diferenta evaluata pe 4 biti (8-14=-6)
$display("Diferenta 4b-4b=4b. Asteptat: 8 - 14 = -6");
$display("  Binar: %b - %b = %b", opB, opA, opC);
$display("  Zecimal: %0d - %0d = %0d", opB, opA, opC);
$display("  Intreg: %0d - %0d = %0d",
    $signed(opB), $signed(opA), $signed(opC));

// Diferenta 4b-4b=5b
opD = opB - opA; // diferenta evaluata pe 5 biti (8-14=-6)
$display("Diferenta 4b-4b=5b. Asteptat: 8 - 14 = -6");
$display("  Binar: %b - %b = %b", opB, opA, opD);
$display("  Zecimal: %0d - %0d = %0d", opB, opA, opD);
$display("  Intreg: %0d - %0d = %0d",
    $signed(opB), $signed(opA), $signed(opD));

// Diferenta 4b-4b=6b, extensie de semn
opE = opB - opA; // diferenta evaluata pe 6 biti (8-14=-6)
$display("Diferenta 4b-4b=6b. Asteptat: 8 - 14 = -6");
$display("  Binar: %b - %b = %b", opB, opA, opE);
$display("  Zecimal: %0d - %0d = %0d", opB, opA, opE);
$display("  Intreg: %0d - %0d = %0d",
    $signed(opB), $signed(opA), $signed(opE));

// Diferenta 4b-4b=integer,
opG = opB - opA; // diferenta evaluata ca numere intregi(8-14=-6)
$display("Diferenta 4b-4b=integer. Asteptat: 8 - 14 = -6");
$display("  Binar: %b - %b = %b", opB, opA, opG);
$display("  Zecimal: %0d - %0d = %0d", opB, opA, opG);
$display("  Intreg: %0d - %0d = %0d",
    $signed(opB), $signed(opA), $signed(opG));

$display("#####");
// Cat impartire 4b/4b=4b
opC = opA / opB; // impartire
$display("Cat impartire 4b/4b=4b. Asteptat: 14 / 8 = 1");
$display("  Binar: %b / %b = %b", opA, opB, opC);
$display("  Zecimal: %0d / %0d = %0d", opA, opB, opC);

// Rest impartire 4b mod 4b=4b
opC = opA % opB; // modulo
$display("Rest impartire 4b mod 4b=4b. Asteptat: 14 mod 8 = 6");
$display("  Binar: %b mod %b = %b", opA, opB, opC);
$display("  Zecimal: %0d mod %0d = %0d", opA, opB, opC);

$display("#####");
// Valoare negativa
$write("Valoare negativa -9 reprezentata pe 8 biti.");
$display(" Asteptat: -9");
$display("  Binar: %b", opH);

```

```

$display(" Zecimal: %0d", opH);
$display(" Intreg: %0d", $signed(opH));

$display("#####");
// Suma intregi cu rezultat negativ
opF = opB + opH; // suma cu rezultat asteptat -1
$write("Suma intregi cu rezultat negativ. ");
$display("Asteptat: 8 + (-9) = -1");
$display(" Binar: %b + %b = %b", opB, opH, opF);
$display(" Zecimal: %0d + %0d = %0d", opB, opH, opF);
$display(" Intreg: %0d + %0d = %0d",
    $signed(opB), $signed(opH), $signed(opF));

$display("#####");
// Suma cu intreg cu rezultat pozitiv
opH = opF + 3; // adunare negativ + pozitiv (integer),
               // rezultat asteptat +2
               // intregul este extins la 32 de biti
               // insa rezultatul trunchiat la 8 biti
$display("Suma cu intreg cu rezultat pozitiv. ");
$display("Asteptat: (-1) + 3 = +2");
$display(" Binar: %b + %b = %b", opF, 3, opH);
$display(" Zecimal: %0d + %0d = %0d", opF, 3, opH);
$display(" Intreg: %0d + %0d = %0d",
    $signed(opF), $signed('d3), $signed(opH));

// Suma cu intreg cu rezultat pozitiv
opH = opF + 'd3; // adunare negativ + pozitiv (integer),
                // rezultat asteptat +2
                // intregul este extins la 32 de biti
                // insa rezultatul trunchiat la 8 biti
$display(" Binar: %b + %b = %b", opF, 'd3, opH);
$display(" Zecimal: %0d + %0d = %0d", opF, 'd3, opH);
$display(" Intreg: %0d + %0d = %0d",
    $signed(opF), $signed('d3), $signed(opH));

// Suma cu intreg cu rezultat pozitiv
opH = opF + 2'd3; // adunare negativ + pozitiv (integer),
                  // rezultat asteptat +2
                  // intregul este extins la 32 de biti
                  // insa rezultatul trunchiat la 8 biti
$display(" Binar: %b + %b = %b", opF, 2'd3, opH);
$display(" Zecimal: %0d + %0d = %0d", opF, 2'd3, opH);
$display(" Intreg: %0d + %0d = %0d",
    $signed(opF), $signed(2'd3), $signed(opH));

// Suma cu intreg cu rezultat pozitiv
opH = opF + 8'd3; // adunare negativ + pozitiv (integer),
                  // rezultat asteptat +2

```

```

// intregul este extins la 32 de biti
// insa rezultatul trunchiat la 8 biti
$display(" Binar: %b + %b = %b", opF, 8'd3, opH);
$display(" Zecimal: %0d + %0d = %0d", opF, 8'd3, opH);
$display(" Intreg: %0d + %0d = %0d",
    $signed(opF), $signed(8'd3), $signed(opH));

```

**end**

În urma simulării, se obține următorul text. Prin corelare cu codul sursă, se pot înțelege regulile interne pentru reprezentarea numerelor și a rezultatelor operațiilor aritmetice în Verilog.

```

#####
# Operator unar opH=00001001.
# + Binar: 00001001
#   Zecimal: 9
#   Intreg: 9
# - Binar: 11110111
#   Zecimal: 247
#   Intreg: -9
# Operator unar opH=11110111.
# + Binar: 11110111
#   Zecimal: 247
#   Intreg: -9
# - Binar: 00001001
#   Zecimal: 9
#   Intreg: 9
#####
# Suma pozitive 4b+4b=4b. Asteptat: 14 + 8 = 22
#   Binar: 1110 + 1000 = 0110
#   Zecimal: 14 + 8 = 6
# Suma pozitive 4b+4b=5b. Asteptat: 14 + 8 = 22
#   Binar: 1110 + 1000 = 10110
#   Zecimal: 14 + 8 = 22
# Suma pozitive 4b+4b=6b. Asteptat: 14 + 8 = 22
#   Binar: 1110 + 1000 = 010110
#   Zecimal: 14 + 8 = 22
#####
# Multiplicare 4b*4b=8b. Asteptat: 14 * 8 = 112
#   Binar: 1110 * 1000 = 01110000
#   Zecimal: 14 * 8 = 112
#####
# Diferenta 4b-4b=4b. Asteptat: 8 - 14 = -6
#   Binar: 1000 - 1110 = 1010
#   Zecimal: 8 - 14 = -6,
#   Intreg: -8 - -2 = -6
# Diferenta 4b-4b=5b. Asteptat: 8 - 14 = -6
#   Binar: 1000 - 1110 = 11010

```

```

#   Zecimal: 8 - 14 = 26
#   Intreg: -8 - -2 = -6
# Diferenta 4b-4b=6b. Asteptat: 8 - 14 = -6
#   Binar: 1000 - 1110 = 111010
#   Zecimal: 8 - 14 = 58
#   Intreg: -8 - -2 = -6
# Diferenta 4b-4b=integer. Asteptat: 8 - 14 = -6
#   Binar: 1000 - 1110 = 111111111111111111111111111111010
#   Zecimal: 8 - 14 = -6
#   Intreg: -8 - -2 = -6
#####
# Cat impartire 4b/4b=4b. Asteptat: 14 / 8 = 1
#   Binar: 1110 / 1000 = 0001
#   Zecimal: 14 / 8 = 1
# Rest impartire 4b mod 4b=4b. Asteptat: 14 mod 8 = 6
#   Binar: 1110 mod 1000 = 0110
#   Zecimal: 14 mod 8 = 6
#####
# Valoare negativa -9 reprezentata pe 8 biti. Asteptat: -9
#   Binar: 11110111
#   Zecimal: 247
#   Intreg: -9
#####
# Suma intregi cu rezultat negativ. Asteptat: 8 + (-9) = -1
#   Binar: 1000 + 11110111 = 11111111
#   Zecimal: 8 + 247 = 255
#   Intreg: -8 + -9 = -1
#####
# Suma cu intreg cu rezultat pozitiv.
# Asteptat: (-1) + 3 = +2
#   Binar: 11111111 + 00000000000000000000000000000011 = 00000010,
#   Zecimal: 255 + 3 = 2,
#   Intreg: -1 + 3 = 2
#   Binar: 11111111 + 00000000000000000000000000000011 = 00000010,
#   Zecimal: 255 + 3 = 2,
#   Intreg: -1 + 3 = 2
#   Binar: 11111111 + 11 = 00000010,
#   Zecimal: 255 + 3 = 2,
#   Intreg: -1 + -1 = 2
#   Binar: 11111111 + 00000011 = 00000010,
#   Zecimal: 255 + 3 = 2,
#   Intreg: -1 + 3 = 2

```

În cadrul operațiilor aritmetice, obiectele de tip `net` (`wire`), obiectele de tip `reg` și numerele întregi în format cu bază de numerație sunt considerate numere pozitive fără semn.

Sunt considerate numere întregi cu semn obiectele de tip `integer` și numere întregi în format zecimal. Numerele negative sunt reprezentate în

binar în complement față de 2.

Dacă un bit al unui operand este 'bx atunci rezultatul operației aritmetice este 'bx.

### 2.5.3 Operatori relaționali

Operatorii relaționali sunt:

|    |                   |
|----|-------------------|
| >  | mai mare          |
| <  | mai mic           |
| >= | mai mare sau egal |
| <= | mai mic sau egal  |

Rezultatul unui operator relațional este „adevărat” (1) sau „fals” (0).

*Exemplu*

```
15 > 12           ; // adevarat (1), 15>12
15 > 8'hx         ; // nedefinit (x), un operator este x
'b1000 >= 'b011110 ; // fals (0), 8 < 30
3 >= 3           ; // adevarat (1), 3=3
```

Operanzii relaționali pot fi utilizați pentru modelarea circuitelor logice de comparare a datelor cu reprezentare numerică. Atenție la reprezentarea datelor întregi negative care pot fi interpretate ca fiind pozitive și genera rezultate diferite prin aplicarea operatorilor relaționali.

Recomandarea este de a se evita utilizarea numerelor cu semn (declarată cu `signed`) și utilizarea operatorilor relaționali considerând reprezentările numerelor pozitive.

Adică:

$$8'b1001\_1100 = 156|_{10}$$

Dar nu:

$$8'b1001\_1100 = -100|_{10}.$$

Astfel:

$$8'b1001\_1100 > 8'b0110\_0100, \text{ pentru că}$$

$$156|_{10} > 100|_{10}, \text{ și nu:}$$

$$-100|_{10} < 100|_{10}.$$

### 2.5.4 Operatori de egalitate

Operatorii de egalitate sunt:

|     |                                |
|-----|--------------------------------|
| ==  | egalitate logică               |
| !=  | inegalitate logică             |
| === | egalitate dependentă de x, z   |
| !== | inegalitate dependentă de x, z |

Rezultatul unui operator de egalitate este „adevărat” (1) sau „fals” (0).

Operatorul de egalitate == și operatorul de inegalitate != consideră apariția caracterelor x și z în operanzi ca determinante pentru un rezultat x.

În schimb, operatorul de egalitate === și operatorul de inegalitate !== ignoră semnificația caracterelor x și z și determină rezultatul fără considerarea acestora.

#### Exemplu

```
wire [ 4 -1:0] opA = 'b10x0 ;
wire [ 4 -1:0] opB = 'b10x0 ;
wire [16 -1:0] opC = 16'hBExF ;
wire [16 -1:0] opD = 16'hBEEF ;

opA == opB      ; // nedefinit (x), valoare x
opA === opB     ; // adevarat (1), x ignorat
2'b11 === 8'b3  ; // adevarat (1), 3=3
opC != opD      ; // nedefinit (x), valoare x
opC !== opD     ; // adevarat (1), x ignorat, 16'hBExF=16'hBEEF
```

### 2.5.5 Operatori logici

Operatorii logici sunt:

|    |               |
|----|---------------|
| && | AND logic     |
|    | OR logic      |
| !  | negarea unară |

Rezultatul unui operator logic este 0 sau 1. Operanzii vectori sunt evaluați ca 0 dacă au toate valorile 0, altfel sunt evaluați ca 1.

#### Exemplu

```
reg          sigA = 1'b0 ;
reg          sigB = 1'b1 ;
reg [4 - 1:0] busC = 4'b1010 ; // evaluat ca 1
reg [4 - 1:0] busD = 4'b1001 ; // evaluat ca 1
reg [4 - 1:0] busE = 4'b0000 ; // evaluat ca 0
reg          sigF = 1'bx  ; // evaluat ca x

sigA && sigB ; // fals AND adevarat = fals (0)
sigA || sigB ; // fals OR adevarat = adevarat (1)
```

```

!sigA      ; // NOT (fals) = adevarat (1)
busC && busD ; // adevarat AND adevarat = adevarat (1)
busC || busD ; // adevarat OR fals = adevarat (1)
!busC      ; // NOT(adevarat) = fals (0)
!busE      ; // NOT(fals) = adevarat (1)
!sigF      ; // NOT(x) = indecis (x)

```

Recomandarea este de a se utiliza operatorii logici (cu două simboluri, && și ||) exclusiv în specificații care returnează valori logice, „adevărat” sau „fals”, nu biți sau vectori de biți. Adică în expresia condiției unei specificații if.

De remarcat faptul că nu există operator ^^.

## 2.5.6 Operatori logici pe vectori de biți

Operatorii logici pe vectori de biți sunt:

|    |                   |
|----|-------------------|
| ~  | negare bit cu bit |
| &  | AND pe biți       |
|    | OR pe biți        |
| ^  | XOR pe biți       |
| ~^ | XNOR pe biți      |

Operatorii logici pe vectori operează „bit cu bit”. Rezultatul unui operator pe vector de biți este tot un vector de biți, având dimensiunea operanzilor. Tabelele funcțiilor implementate de operatorii logici sunt prezentate în tabelul 2.2.

*Exemplu*

```

reg      sigA = 1'b0 ;
reg      sigB = 1'b1 ;
reg [4-1:0] busC = 4'b1010 ;
reg [4-1:0] busD = 4'b1001 ;
reg [4-1:0] busE = 4'b0000 ;
reg      sigF = 1'bx ;

sigA & sigB ; // 0 and 1 = 1'b0
sigA | sigB ; // 0 or 1 = 1'b1
~sigA      ; // not (1'b0) = 1'b1
busC & busD ; // 1010 and 1001 = 1000 = 4'b1000
busC && busD ; // adevarat && adevarat = adevarat = 1'b1
busC & busE ; // 1010 and 0000 = 0000 = 4'b0000
busC && busE ; // adevarat && fals = fals = 1'b0
busC | busD ; // 1010 or 1001 = 1011 = 4'b1011
~busC      ; // not (1010) = 4'b0101
~busD      ; // not (1001) = 4'b0110

```



| & AND | 0 | 1 | x | z | OR | 0 | 1 | x | z |
|-------|---|---|---|---|----|---|---|---|---|
| 0     | 0 | 0 | 0 | 0 | 0  | 0 | 1 | x | x |
| 1     | 0 | 1 | x | x | 1  | 1 | 1 | 1 | 1 |
| x     | 0 | x | x | x | x  | x | 1 | x | x |
| z     | 0 | x | x | x | z  | x | 1 | x | x |

| ^ XOR | 0 | 1 | x | z | ~^ XNOR | 0 | 1 | x | z |
|-------|---|---|---|---|---------|---|---|---|---|
| 0     | 0 | 1 | x | x | 0       | 1 | 0 | x | x |
| 1     | 1 | 0 | x | x | 1       | 0 | 1 | x | x |
| x     | x | x | x | x | x       | x | x | x | x |
| z     | x | x | x | x | z       | x | x | x | x |

| ~ NOT | 0 | 1 | x | z |
|-------|---|---|---|---|
|       | 1 | 0 | x | x |

Tabelul 2.2

Tabelul funcțiilor implementate de operatorii logici.

```

!busD      ; // not (adevarat) = fals = 1'b0
~busE      ; // not (0000) = 4'b1111
~sigF      ; // not (x) = 1'bx
sigA & sigF ; // 0 and x = 1'b0
sigB & sigF ; // 1 and x = 1'bx

```

### 2.5.7 Operatori de reducere

Operatorii de reducere sunt:

|    |                                 |
|----|---------------------------------|
| &  | AND între toți biții vectorului |
| ~& | NAND, inversul operatorului &   |
|    | OR între toți biții vectorului  |
| ~  | NOR, inversul operatorului      |
| ^  | XOR între toți biții vectorului |
| ~^ | XNOR, inversul operatorului ^   |

Operatorii de reducere se aplică unui operand vector și produc rezultat pe un bit. Acești operatori sunt foarte utili în exprimarea unor funcții pe vectori de dimensiune parametrizabilă.

*Exemplu*

```

reg [4 -1 :0] busA = 4'b1010 ;
reg [4 -1 :0] busB = 4'b1011 ;
reg [4 -1 :0] busC = 4'b10x0 ;

```

```

&busA ; // evaluat ca 0
~&busA ; // evaluat ca 1
|busA ; // evaluat ca 1
~|busA ; // evaluat ca 0
^busA ; // evaluat ca 0
~^busA ; // evaluat ca 1
^busB ; // evaluat ca 1
^busC ; // evaluat ca x

```

Operatorii de reducere pot fi folosiți pentru a verifica dacă un bit dintr-un bus are valoare x. Această observație poate fi folosită în specificații de verificare a coerenței datelor de pe un bus:

```

if (^busC == 1'bx)
    $display("Valoare necunoscuta pe busC = %b", busC);

```

O altă aplicație a operatorilor de reducere constă în compararea valorii unui vector cu valoarea întregă 0 (toți biții egali cu 0) sau verificarea ajungerii la capătul domeniului de reprezentare a numerelor pozitive (toți biții egali cu 1).

```

&busC ; // =1 daca toti bitii sunt egali cu 1
~|busC ; // =1 daca toti bitii sunt egali cu 0

```

Operatorul de reducere XOR ^ poate fi utilizat pentru a determina paritatea numărului de biți egali cu 1 în cadrul unui vector de biți.

### Exemplu

Descrieți un circuit logic combinațional care determină următoarele condiții ale unui bus având un număr parametrizabil de biți:

- Toate intrările sunt 1
- Toate intrările sunt 0
- Există cel puțin o intrare 1
- Există cel puțin o intrare 0
- Există un număr impar de intrări egale cu 1
- Există un număr par de intrări egale cu 0

### Soluție

Codul este prezentat în continuare și se bazează pe operatori de reducere aplicați asupra vectorului de biți *bus\_in* de dimensiune parametrizabilă *WIDTH*. Găsirea operatorilor potriviți pentru a rezolva o cerință este o chestiune de logică.

**Very-logic.**

```

module cond_bus #(

```

```

parameter WIDTH = 8 // latime bus, in biti
)(
input    [WIDTH-1:0] bus_in ,
output   all_1 , // toate intrarile sunt 1
output   all_0 , // toate intrarile sunt 0
output   one_1 , // cel puțin o intrare 1
output   one_0 , // cel puțin o intrare 0
output   odd_1 , // număr impar de intrări egale cu 1
output   even_0  // număr par de intrări egale cu 0
);
// toate intrarile sunt 1
assign all_1 = &bus_in ; // SI pe biti
// toate intrarile sunt 0
assign all_0 = ~|bus_in ; // nicuna nu este 1
// cel puțin o intrare 1
assign one_1 = |bus_in ; // SAU pe biti
// cel puțin o intrare 0
assign one_0 = ~&bus_in ; // nu sunt toate 1
// număr impar de intrări egale cu 1
assign odd_1 = ^bus_in ; // XOR pe biti
// număr par de intrări egale cu 0
assign even_0 = ~(~bus_in) ; // negare bus urmată de
                                // operator de reducere XNOR

endmodule // cond_bus

```

## 2.5.8 Operatori de deplasare

Operatorii de deplasare sunt:

|     |                              |
|-----|------------------------------|
| <<  | deplasare stânga             |
| >>  | deplasare dreapta            |
| <<< | deplasare aritmetică stânga  |
| >>> | deplasare aritmetică dreapta |

Operatorii de deplasare produc deplasarea biților operandului din stânga cu numărul de poziții precizat în dreapta, în direcția precizată de operator. Pe pozițiile eliberate prin deplasarea biților apar valori 0.

### Exemplu

```

reg[8-1:0] busA = 8'b1001_1100;
busA >> 2 ; // evaluat ca 8'b0010_0111
busA << 2 ; // evaluat ca 8'b0111_0000

```

Operatorul de deplasare aritmetic >>> este un operator de deplasare la dreapta care păstrează valoarea celui mai semnificativ bit (bitul din stânga) considerat bit de semn. Atenție, acesta se întâmplă doar dacă valoarea de-

plasată este declarată a fi cu semn (cu cuvântul rezervat `signed`). Dacă valoarea nu este declarată explicit ca fiind cu semn, operatorul de deplasare aritmetic `>>>` are comportament identic cu cel al operatorului de deplasare `>>`.

Pentru numere pozitive,  $MSb=0$ , efectul operatorului `>>>` este identic cu cel al operatorului `>>`. Pentru numere negative,  $MSb=1$ , cu operatorul `>>>` apar valori 1 pe biții eliberați în partea stângă. Rezultatul unei deplasări aritmetice cu un bit spre dreapta este similar împărțirea la 2, și în cazul numerelor întregi, reprezentate în complement față de 2.

Operatorul de deplasare aritmetic `<<<` este un operator de deplasare la stânga cu un comportament identic cu cel al operatorului `<<`, ignorând semnul valorii deplasate. Atenție, o valoare negativă deplasată la stânga poate deveni pozitivă, dacă următorul bit după  $MSb$  este 0.

Fiind o regulă specială, recomandarea este de a se evita utilizarea operatorului de deplasare aritmetic `>>>` sau a se utiliza cu multă grijă.

### Exemplu

```
reg      [8-1:0] busA      = 8'b1001_1101;
reg signed [8-1:0] busA_signed = 8'b1001_1101;

busA >> 2      ; // evaluat ca 8'b0010_0111
busA << 2      ; // evaluat ca 8'b0111_0100
busA >>> 2     ; // evaluat ca 8'b0010_0111, similar cu >>
busA <<< 2     ; // evaluat ca 8'b0111_0100, similar cu <<
busA_signed >> 2 ; // evaluat ca 8'b0010_0111
busA_signed << 2 ; // evaluat ca 8'b0111_0100
busA_signed >>> 2 ; // evaluat ca 8'b1110_0111, pastrat MSb
busA_signed <<< 2 ; // evaluat ca 8'b0111_0100, similar cu <<
```

Operatorii de deplasare sunt foarte frecvent folosiți pentru a descrie operații de înmulțire și împărțire cu numere puteri ale lui 2 sau descrierea exponențierii în binar.

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| $1 \ll 3$     | 1 deplasat stânga cu 3 biți, echivalent cu „2 la puterea 3”, adică $8 = 1000_2$ |
| $1 \ll 7$     | echivalent cu „2 la puterea 7”, $128 = 10000000_2$                              |
| $opA \ll opB$ | echivalent cu „ $opA$ înmulțit cu 2 la puterea $opB$ ”                          |
| $opA \gg opB$ | echivalent cu „ $opA$ împărțit la 2 la puterea $opB$ ”                          |

Operatorul de deplasare stânga `<<` este utilizat și la modelarea funcției de transfer a unui circuit decodificator:

```
input  [4 -1:0] dcd_i;
output [8 -1:0] dcd_o;
// rezultatul este un vector de 8 biti avand un singur bit 1,
// pe pozitia cu index egal cu dcd_i
```

```
assign dcd_o = (1'b1 << dcd_i);
```

## 2.5.9 Operatorul condițional

Există un singur operator condițional:

|                                                                       |
|-----------------------------------------------------------------------|
| <code>&lt;conditie&gt; ? &lt;expresie1&gt; : &lt;expresie0&gt;</code> |
|-----------------------------------------------------------------------|

În cazul condiției adevărate, operatorul întoarce valoarea <expresie1>. Altfel, operatorul întoarce valoarea <expresie0>. Comportamentul este similar cu cel al circuitului multiplexor.

### Exemplu

```
assign y = sel ? i1 : i0;
```

## 2.5.10 Operatori de concatenare și replicare a biților

Concatenarea este operația de alăturare a biților într-un bus. Două sau mai multe expresii care semnifică biți se pot alătura pentru a forma o valoare reprezentată pe mai mulți biți. Operatorii de concatenare sunt caracterele acolade: {, }.

O anumită expresie poate fi replicată de un număr de ori în cadrul concatenării. Sintaxa include numărul de repetări și expresia repetată, incluse între acolade {numar{valoare}}.

```
wire [ 4 -1:0] a,b;
wire [ 8 -1:0] c ;
wire [16 -1:0] d ;
assign a = 4'b1011;
assign b = 4'b1110;
assign c = {a,b} ; // c=8'b1011_1110
assign d = {4{a}} ; // d=16'b1011_1011_1011_1011
```

### Exemplu

```
// initializarea cu un pattern
reg [128 -1:0] reg128;
reg128 <= {4{32'hDEAD_BEEF}}; // de 4 ori cate 32 de biti
// echivalent cu 128'hDEAD_BEEF_DEAD_BEEF_DEAD_BEEF_DEAD_BEEF

// registrul cu deplasare stanga
reg [32 -1:0] leftShReg;
leftShReg <= {leftShReg[30:0], 1'b0};

// registrul cu deplasare in inel
```

```

reg [32 -1:0] inelReg;
inelReg <= {inelReg[30:0], inelReg[31]};

// extindere de semn a vectorilor cu
// semnificatie de numere intregi (cu semn)
wire [ 8 -1:0] sign8 ;
wire [16 -1:0] sign16 ;
assign sign16 = {{8{sign8[7]}}, sign8};
// sign8 = 8'b0001_1111 (pozitiv) =>
// sign16 = 16'b0000_0000_0001_1111

// sign8 = 8'b1001_1111 (negativ) =>
// sign16 = 16'b1111_1111_1001_1111

```

Operatorii de concatenare și replicare au câteva limitări:

- În șirul de concatenare nu sunt admise constante cu dimensiuni neprecizate:
  - ilegal {busA, 5}, dar
  - legal {busA, 4'd5}.
- Numărul de repetări nu poate fi decât o constantă:
  - ilegal {num{2'b10}} dacă num este un semnal, dar
  - legal {3{2'b10}}.

### 2.5.11 Precedența operatorilor

Tabelul 2.3 prezintă operatorii Verilog în ordinea precedenței acestora (primul din listă este evaluat cel dintâi, operatorul ternar este evaluat ultimul).

Recomandarea este de a se utiliza paranteze pentru a declara explicit ordinea operatorilor. Chiar dacă un inginer învață pe de rost tabela precedenței operatorilor, codul scris va fi greu de citit, atât de către cei care nu știu ordinea operatorilor, dar și ulterior de către autorul codului. Este mai simplu de pus paranteze pentru a modela explicit ordinea operatorilor într-o expresie.

#### Exemplu

Operatorul + are o precedență mai mare decât operatorul de deplasare <<. Ce valoare va returna expresia  $1 \ll 7 + 1$ ?

#### Soluție

Expresia  $1 \ll 7 + 1$  crează confuzie deoarece valoarea rezultată va fi  $256 = 1\_0000\_0000|_2$  provenită din evaluarea inițială a sumei  $7 + 1 = 8$  și ulterior a deplasării  $1 \ll 8$ .

| Operator | Descriere                                                                                       |
|----------|-------------------------------------------------------------------------------------------------|
| +        | Operator unar pentru număr pozitiv                                                              |
| -        | Operator unar pentru număr negativ                                                              |
| !        | Operator unar pentru negare logică                                                              |
| ~        | Operator unar pentru negare pe biți                                                             |
| &        | Operator de reducere AND (aplicat unui vector produce un bit)                                   |
| ~&       | Operator de reducere NAND                                                                       |
| ^        | Operator de reducere XOR                                                                        |
| ~^       | Operator de reducere XNOR                                                                       |
|          | Operator de reducere OR                                                                         |
| ~        | Operator de reducere NOR                                                                        |
| *        | Operator binar de multiplicare                                                                  |
| /        | Operator binar de împărțire                                                                     |
| %        | Operator binar modulo                                                                           |
| +        | Operator binar de adunare                                                                       |
| -        | Operator binar de scădere                                                                       |
| <<       | Operator binar de deplasare stânga                                                              |
| >>       | Operator binar de deplasare dreapta                                                             |
| <        | Operator binar de comparare „mai mic”                                                           |
| <=       | Operator binar de comparare „mai mic sau egal”                                                  |
| >        | Operator binar de comparare „mai mare”                                                          |
| >=       | Operator binar de comparare „mai mare sau egal”                                                 |
| ==       | Operator binar de egalitate logică                                                              |
| !=       | Operator binar de inegalitate logică                                                            |
| ===      | Operator binar de egalitate cu selecție                                                         |
| !==      | Operator binar de inegalitate cu selecție                                                       |
| &        | Operator binar logică AND pe biți<br>(aplicat unor operanzi vectori produce un vector)          |
| ^        | Operator binar logică XOR pe biți                                                               |
| ~^       | Operator binar logică XNOR pe biți                                                              |
|          | Operator binar logică OR pe biți                                                                |
| &&       | Operator binar AND logic<br>(aplicat unor expresii, produce valoarea de adevăr a unei condiții) |
|          | Operator binar OR logic                                                                         |
| ? :      | Operator ternar condițional                                                                     |

Tabelul 2.3

Operatorii Verilog în ordinea precedenței acestora.

Deși, poate intenția era de a se obține rezultatul 129, dacă se evaluează inițial  $1 \ll 7 = 128$  și ulterior  $128 + 1 = 129$ .

Se recomandă existența unor paranteze care să facă explicită intenția proiectantului:

$1 \ll (7 + 1)$ , rezultat 256 sau

$(1 \ll 7) + 1$ , rezultat 129.

Operatorul  $\&$  are o precedență mai mare față de operatorul  $|$ . Ca efect, o scriere *sumă de produse* ar putea fi scrisă fără paranteze. Însă, este recomandată utilizarea parantezelor pentru o siguranță mai mare legată de intenția autorului.

*Exemplu*

$$F(A, B, C, D) = A \cdot B + C \cdot D$$

Specificația Verilog se poate scrie astfel:

```
assign F = A & B | C & D;
```

Ce valoare are  $F(1, 1, 1, 0)$ ?

*Soluție*

Deoarece operatorul  $\&$  are o precedență mai mare față de operatorul  $|$ , codul Verilog trebuie interpretat astfel:

```
assign F = (A & B) | (C & D);
```

Valoarea returnată va fi:  $F(1, 1, 1, 0) = (1 \cdot 1) + (1 \cdot 0) = 1 + 0 = 1$

Însă, dacă intenția era ca întâi să se evalueze funcția OR, ar fi trebuit codul să fie scris cu paranteze:

```
assign F = A & (B | C) & D;
```

iar valoarea returnată ar fi fost:

$$F(1, 1, 1, 0) = 1 \cdot (1 + 1) \cdot 0 = 1 \cdot 1 \cdot 0 = 0$$

Sau, poate că intenția a fost să se evalueze operatorii în ordine, din dreapta în stânga, caz în care codul ar fi trebuit scris astfel:

```
assign F = A & (B | (C & D));
```

iar valoarea returnată ar fi fost:

$$F(1, 1, 1, 0) = 1 \cdot (1 + (1 \cdot 0)) = 1 \cdot (1 + 0) = 1 \cdot 1 = 1$$

*Concluzie*

Este de preferat scrierea codului cu paranteze, pentru a evita greșelile datorate ignorării precedenței operatorilor. Trebuie avut în vedere că, în



lipsa parantezelor, chiar dacă este un cod corect, o viitoare modificare a codului va fi foarte riscantă.

Utilizați paranteze pentru a scrie un cod care să nu se bazeze pe reguli interne ale limbajului Verilog.

Nu ați vrea să trebuiască să descoperiți intenția autorului următorului cod...

```
assign a = b && &c | d & e;
assign a = b ? c + d : c - d;
assign a = b & c | d ^ e;
assign a = ~b & c | d & ~e;
assign a = ~b & ~c & ~e | ~f;
assign a = b == c;
```

## 2.5.12 Modelarea circuitelor logice combinaționale

În Verilog, **modelarea circuitelor logice combinaționale** simple se poate face utilizând specificația de **atribuire continuă**. Această specificație este introdusă de cuvântul rezervat **assign** („a atribui”, engl.). Atribuirea continuă modelează semnalul la ieșirea unei porți logice care își re-calculează starea în mod continuu, prin re-evaluarea intrărilor și aplicând funcția logică de transfer.

Într-un modul pot exista mai multe specificații de atribuire, considerate a fi concurente (se execută în paralel, independent una de alta). Concurența specificațiilor reprezintă mecanismul limbajelor de descriere hardware pentru modelarea circuitelor hardware, deoarece acestea lucrează în paralel. Ordinea specificațiilor de atribuire continuă în cod este indiferentă.

**Deși există posibilitatea de a modela timpul de propagare al circuitelor și porților logice, în modelele destinate sintezei de hardware nu se vor utiliza specificații de timp.**

Modelarea circuitelor logice combinaționale cu specificații de atribuire continuă **assign** este foarte intuitivă, prezentând numele semnalului atribuit (ieșirea circuitului) delimitat cu simbolul egal = de expresia logică a funcției implementate în care apar semnalele de intrare și operatori logici. La fiecare modificare a unui semnal aflat în partea dreaptă a simbolului de atribuire =, se re-calculează valoarea funcției logice și se atribuie semnalului de ieșire.

Se remarcă faptul că, pentru specificația **assign**, simbolul de atribuire este = (un „egal”) și nu se confundă cu operatorul de egalitate (reprezentat de ==, două simboluri „egal”) sau operatorul de egalitate (reprezentat de ==, trei simboluri „egal”).

Există posibilitatea de a modela cu o specificație `assign` mai multe ieșiri, dacă acestea se modelează a fi concatenate într-un bus. De exemplu, un sumator complet de 1 bit se poate modela astfel, ținând cont de transportul de intrare și generând transportul de ieșire:

```
assign {co, s} = a + b + ci;
```

Modelul adună cei 3 biți de intrare și codifică rezultatul pe cele două ieșiri `co` și `s`, modelate ca o concatenare de două semnale, cu `co` cel mai prioritar (pondere  $2^1$ ).

Semnalele atribuite cu specificație `assign` se declară ca fiind de tipul `net` prin utilizarea cuvântului rezervat `wire`.

```
wire co ;
wire s ;
```

Dacă semnalele atribuite cu specificație `assign` sunt porturi de ieșire ale modului (declarate ca `output`), declarația `wire` nu mai este necesară.

Dacă un semnal are mai mulți biți, la declarația `wire` se declară dimensiunea bus-ului (indexul maxim și minim, între paranteze drepte, cu delimitator caracterul „:”).

Semnalele interne nedeclarate se consideră a fi declarate ca `wire` de 1 bit și pot fi atribuite cu specificația `assign`.

O specificație `assign` reprezintă o „sursă” („*driver*”, engl.) pentru un semnal. În mod legal, un semnal are o singură sursă deci poate fi atribuit (se află în partea stângă a semnului =) dintr-o singură specificație `assign` (sau `always`).

În cazul unor funcții de transfer mai complexe, cu mai multe variabile de intrare sau cu un comportament mai sofisticat, operatorii logici nu oferă un suport foarte facil. Pentru aceste cazuri, există posibilitatea de a descrie funcția de transfer într-un mod secvențial, algoritmic. Însă, comportamentul modelat va fi tot de circuit logic combinațional.

Modelarea circuitelor combinaționale cu specificații secvențiale (introduse prin cuvântul rezervat `always`) are anumite restricții.

Specificația `always` este o specificație concurentă. Pot exista mai multe specificații `always` într-un modul și ordinea acestora în cod este indiferentă. Însă, specificația `always` include alte specificații care sunt considerate secvențiale, adică se execută în ordine una după alta.

Altfel spus, specificația `always` este o specificație concurentă care încapsulează specificații secvențiale. Acest lucru permite exprimarea unei

funcții logice sub formă algoritmică. Specificațiile incluse în `always` („întotdeauna”, engl.) se execută în ordine, de la prima la ultima. La final, după execuția ultimei specificații secvențiale, se așteaptă condiția de re-activare a specificației `always` și se reia execuția aceluiași specificații. Specificațiile `always` se execută la infinit, de la timpul zero al simulării, până la finalizarea simulării.

Pentru a modela un circuit combinațional, condiția de re-activare a specificației `always` o reprezintă modificarea unui semnal de intrare. Din acest motiv, condiția de re-activare, plasată imediat după cuvântul rezervat `always` și precedată de simbolul `@`, listează toate semnalele considerate intrări pentru circuitul combinațional. Practic, specificația `always` următoare se citește astfel:

```
always @(s or i0 or i1)
```

*„Întotdeauna, la modificarea semnalelor `s` sau `i0` sau `i1`, se execută setul de specificații ce urmează...”*

Lista de semnale care activează specificația care modelează circuite combinaționale trebuie să fie completă. Acest lucru este riscant nu numai pentru că uneori proiectantul uită un semnal de intrare, dar și la modificarea ulterioară a specificațiilor secvențiale prin utilizarea unor noi intrări, fără a fi adăugate în listă. Pentru simplificare, specificația `always` acceptă o declarație implicită a „tuturor semnalelor” de intrare, prin utilizarea caracterului `*` („wildcard”, engl.). Specificația următoare se poate citi:

```
always @(*)
```

*„Întotdeauna, la modificarea oricărui semnal menționat, se execută setul de specificații ce urmează...”*

Dacă în interiorul specificației `always` sunt prezente mai multe specificații (secvențiale), acestea trebuie incluse între cuvintele rezervate `begin` și `end`.

Se remarcă faptul că specificația `always` are două simboluri de atribuire:

= (un „egal”), similar cu specificația de atribuire continuă `assign` și  
 <= (două simboluri „mai mic” și „egal”).

Diferența de funcționalitate dintre cele două simboluri de atribuire va fi explicată la modelarea sistemelor secvențiale (*atribuiri blocante sau neblo-cante*).

Pentru modelarea circuitelor combinaționale cu specificație `always` se recomandă utilizarea simbolului de atribuire = (un „egal”).

O specificație `always` reprezintă o „sursă” („*driver*”, engl.) pentru un semnal. În mod legal, un semnal are o singură sursă, deci poate fi atribuit (se află în partea stângă a semnului `=`) dintr-o singură specificație `always` (sau `assign`).

Semnalele atribuite cu specificație `always` trebuie obligatoriu declarate explicit ca fiind de tipul `reg` (cuvânt rezervat). Dacă semnalul este o ieșire din modul, cuvânt rezervat `reg` poate apare imediat după cuvântul ce desemnează direcția portului, `output`.

Specificațiile secvențiale incluse în `always` permit descrierea funcției circuitului combinațional într-o manieră algoritmică. În continuare sunt detaliate cele mai frecvent utilizate specificații secvențiale: specificația de decizie `if` și specificația de selecție `case`.

### Specificația de decizie `if`

Sintaxa specificațiilor condiționale este următoarea:

```
if (<conditie>) begin
    <specificatii_secventiale_1>
end
else begin
    <specificatii_secventiale_0>
end
```

Execuția specificației `if` constă în evaluarea condiției, urmată de executarea unui grup de specificații, în funcție de valoarea de adevăr a acesteia. Pentru a modela un circuit logic combinațional, trebuie să se atribuiască valori unui semnal de ieșire indiferent de valoarea de adevăr a condiției. Altfel spus, specificațiile `if` utilizate pentru modelarea circuitelor logice combinaționale trebuie să aibă obligatoriu clauza `else` și să prezinte atribuiri pentru toate semnalele, indiferent de valoarea de adevăr a condiției.

De exemplu, un multiplexor 2:1 se poate modela astfel:

```
always @(*)
if (s)
    y = i1;
else
    y = i0;
```

Comportamentul specificației `if` este similar cu cel al operatorului ternar „ `?` : ”.

Specificațiile condiționale pot fi incluse unele în altele pentru a realiza condiții multiple și complexe. În acest caz, se recomandă folosirea delimitatorilor `begin` și `end` pentru evitarea ambiguităților. Delimitatorii `begin`

și `end` sunt obligatoriu de folosit în cazul existenței mai multor specificații secvențiale executate pe o ramură a specificației `if`, dar pot lipsi în cazul unei singure specificații.

Următoarea specificație este **incorectă** deoarece nu atribuie valori unui semnal în ambele condiții. Semnalul `a` este atribuit doar dacă  $s = 1$ , iar semnalul `b` este atribuit doar dacă  $s=0$ . Acesta nu este un comportament de circuit logic combinațional.

```
always @(*)
if (s)
    a = i1;
else
    b = i0;
```

Următoarea specificație este **incorectă** deoarece nu atribuie valori semnalului `a` în ambele condiții. Semnalul `a` nu este atribuit dacă  $s=0$ . Acesta nu este un comportament de circuit logic combinațional.

```
always @(*)
if (s)
    a = b;
```

Următoarea specificație este **incorectă** deoarece atribuie valori unui semnal `a` în două specificații `always`. Chiar dacă ar părea că cele două specificații atribuie valori în condiții complementare, `a` are două surse, ceea ce este ilegal.

```
always @(*)
if (s)
    a = i1;

always @(*)
if (~s)
    a = i0;
```

Modelarea corectă este cu o singură specificație `always`:

```
always @(*)
if (s) // s=1
    a = i1;
else // s=0
    a = i0;
```

### Specificația de selecție `case`

Specificația de selecție `case` permite evaluarea unei expresii și deciderea specificațiilor ce urmează a se executa pe baza valorii acesteia. Sintaxa specificației `case` este următoarea:

```

case (<expresie>)
valoare: <specificatii_secventiale>
default: <specificatii_secventiale>
endcase

```

Execuția specificației `case` începe cu evaluarea expresiei de selecție. Valoarea evaluată se compară secvențial, începând cu prima valoare ce apare în listă. Dacă se găsește o egalitate, se execută specificațiile secvențiale ce îi corespund și apoi se încheie execuția specificației `case`.

Ca o consecință, dacă există în lista de selecție două valori egale (ceea ce poate fi o greșeală), se execută grupul de specificații asociat primei apariții a acelei valori.

Dacă valoarea la care a fost evaluată expresia nu a fost găsită, se execută specificațiile care apar după cuvântul rezervat `default`.

**Folosirea clauzei `default` este obligatorie pentru modelarea circuitelor logice combinaționale.**

Modelul unui multiplexor 4:1, pe baza unei specificații `case` este prezentat în continuare.

```

wire [2 -1:0] s;
wire [4 -1:0] i;
reg          y;

always @(*)
case (s)
2'b00    : y = i[0];
2'b01    : y = i[1];
2'b10    : y = i[2];
default : y = i[3];
endcase

```

**Rând de pecetluire:**

Și iată, preaînțelepți cititori, am străbătut împreună codrii întortocheați ai circuitelor logice combinaționale, unde semnalul se preumblă fără răgaz, iar porțile — asemenea străjerilor de cetate — hotărăsc soarta biților rătăcitori.

Așadar, isprăvind noi cu aceste scorneli de circuite logice combinaționale, unde ieșirea țâșnește numaidecât după voia intrărilor, ne pregătim a păși pe meleaguri și mai năstrușnice: acolo unde trecutul nu se uită și unde circuitele logice secvențiale țin minte tainice întâmplări, ca niște cronici bătrânești în care fiecare clipă se leagă de alta, spre a făuri vremuri digitale și mai mărețe.

**Să ne încingem, dară, centurile cunoașterii și să ne aruncăm cu voioșie în această nouă aventură!**

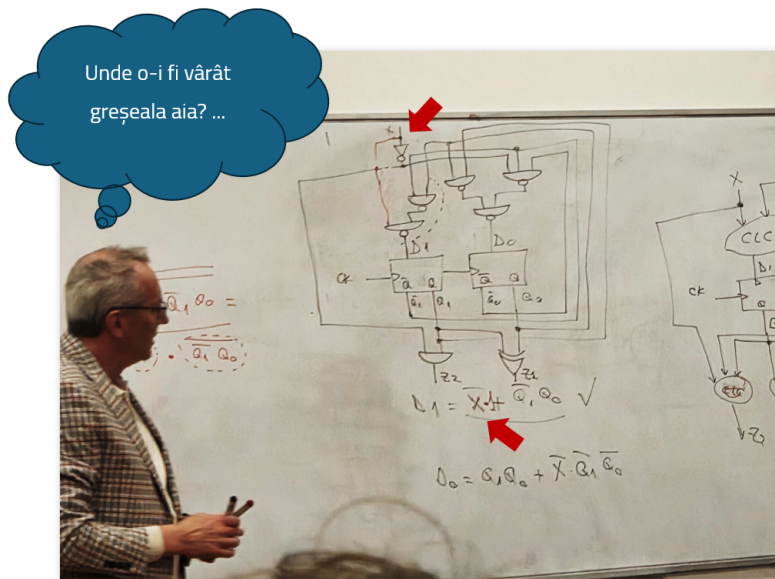
*Text generat cu ChatGPT.*





# Capitolul 3

## Circuite logice secvențiale





### Predoslovie la Capitolul al Treilea:

#### **În care circuitele încep să țină minte și să se comporte ca oamenii – adică neprevăzut**

După ce ne-am jucat cu circuite care răspund doar la ce le bagi în ele pe moment, de parcă ar fi niște funcționari la ghișeu („– Ai datele? – Am. – Ia rezultatul și du-te!”), iată că pășim acum într-un tărâm mai... gingaș.

Un tărâm unde circuitele țin minte. Da, dragă cititorule, ai citit bine: am ajuns la circuitele logice secvențiale, adică acelea care se uită nu doar la ce le spui acum, ci și la ce le-ai spus mai înainte. Și nu uită!

Asemenea unor soacre vigilente, ele păstrează minte lungă și notează tot. Dacă le-ai dat ieri un „1” cam supărat, azi te trezești că nu mai vor să colaboreze. Și mai rău: te pot băga într-o buclă, așa cum băga bunica fusul în fuior – fără scăpare.

În acest capitol, vom face cunoștință cu noțiuni precum flip-flop (nu, nu sunt sandale de vară), registre, latch-uri și ceasuri digitale, care bat nu ora exactă, ci ritmul schimbărilor de stare. Și ferească sfântul să n-ai grijă de sincronizare, că iese bâlci în tot circuitul!

Vei vedea cum aceste jivine electronice combină prezentul cu trecutul, spre a croi un viitor logic — dar uneori și năstrușnic. Aici, fiecare semnal contează, fiecare clipire de ceas se poate lăsa cu haos sau armonie. E un dans delicat între biți și timp, ca-ntr-un bal mascat de procesări binare.

Așadar, ia-ți răbdarea de mână, curiozitatea de guler și să pășim cu băgare de seamă în acest capitol, căci aici se face trecerea de la fire la firile lucrurilor.

Și nu uita: **în lumea secvențială, cine uită... e deja pierdut!**

*Text generat cu ChatGPT.*

## 3.1 Circuite de memorare a stării

Circuitele logice combinaționale prezintă căi de propagare între intrări și ieșiri. Nu există bucle închise între ieșiri și intrări. Circuitele combinaționale determină starea logică a ieșirilor pe baza evaluării funcției de transfer asupra datelor de intrare. Circuitele combinaționale trebuie să aibă surse la toate intrările, deoarece altfel nu se poate determina starea ieșirii. Din puncte de vedere electric, o intrare neconectată este echivalentă cu o impedanță de intrare infinită. Având în vedere legea lui Ohm,  $V = I \cdot R$ , dacă  $R = \infty$ , este posibil ca tensiunea să aibă orice nivel logic, determinat de orice valoare a curentului apărut ca zgomot.

Circuitele combinaționale își re-evaluează valoarea ieșirii la fiecare modificare a intrărilor. Însă, datorită timpului de propagare prin porțile logice, re-evaluarea ieșirii nu se face instantaneu.

Ce se întâmplă dacă se închide o buclă între o ieșire și o intrare ale unui circuit logic combinațional? Toate intrările au surse.

Însă, își menține circuitul combinațional comportamentul?

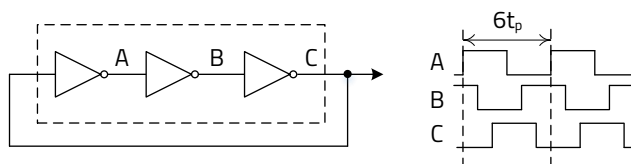
Dacă se realizează o buclă între ieșirea și intrarea unui circuit logic combinațional, comportamentul acestuia se modifică. Există două tipuri de comportamente așteptate, dependente de numărul de inversoare incluse în bucla combinațională:

În cazul unui **număr impar de inversoare, circuitul poate oscila (la infinit) între două stări instabile**. Acest tip de comportament este utilizat pentru realizarea circuitelor oscilatoare digitale, cu caracteristici (nivele de tensiune, frecvență, stabilitate) nu prea performante. Perioada semnalului generat este dependentă de procesul tehnologic, de tensiunea de alimentare și de temperatură. Semnalul la ieșirea circuitului nu este prea „digital” (fronturi rapide) ci se apropie mai mult de un semnal sinusoidal. Din acest motiv, oscilatoarele realizate cu un număr impar de porți inversoare se utilizează doar în aplicații care nu necesită o precizie prea mare a parametrilor semnalului generat.

Circuitul combinațional cu o buclă între ieșire și intrare, închisă peste un număr impar de inversoare, prezintă un comportament oscilant, periodic, infinit, cu perioadă determinată de timpii de propagare ai semnalului prin bucla combinațională.

Figura 3.1 prezintă un circuit cu 3 inversoare în buclă și formele de undă asociate acestui circuit. Se observă faptul că nodul de ieșire  $C$ , de la ieșirea celui de-al 3-lea inversor este conectat la intrarea primului inversor. Toate semnalele din circuit au un comportament oscilant datorat faptului că ieșirea unui inversor nu este instantanee (pentru o scurtă perioadă

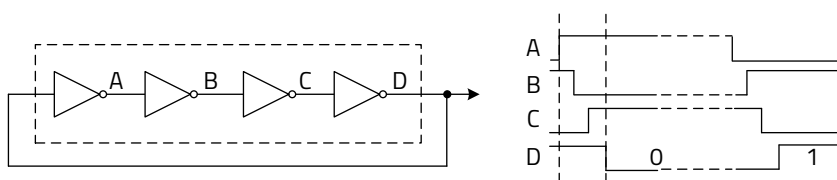
de timp, starea logică a ieșirii unui inversor este identică cu starea logică a intrării acestuia).



**Figura 3.1** Buclă închisă peste cu număr impar de inversoare. Comportament oscilant. Forme de undă.

În cazul unui **număr par de inversoare**, circuitul poate să își păstreze starea (să o „memoreze”) și după dispariția stimulului de intrare, prin acțiunea buclei închise între ieșire și intrare. Acest tip de comportament este utilizat pentru realizarea circuitelor de „memorare a stării”, capabile să rețină o informație după dispariția acesteia de la intrarea circuitului.

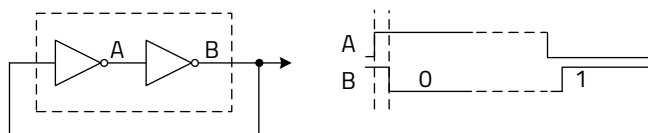
Circuitul combinațional cu o buclă între ieșire și intrare, închisă peste un număr par de inversoare, figura 3.2, prezintă două stări stabile. Starea în care se stabilizează circuitul este determinată de circuit și de stările intrărilor la momentele anterioare de timp.



**Figura 3.2** Buclă închisă peste cu număr par de inversoare. Comportament de memorare a stării. Forme de undă.

Figura 3.2 prezintă un circuit combinațional cu 4 inversoare și o buclă între ieșire și intrare. Cu numai două inversoare (cel mai mic număr par) se obține un comportament identic, cu un cost minim. Circuitul cu două inversoare, prezentat în figura 3.3 are două stări stabile. Nodul A poate avea fie starea logică 0, fie starea logică 1. Nodul B, similar, poate avea fie starea logică 1, fie starea logică 0. Cum poate fi însă știut în care din cele

două stări stă circuitul?



**Figura 3.3** Buclă închisă peste două inversoare (număr minim par). Forme de undă.

Pentru ca o buclă realizată peste un număr par de inversoare să fie controlată (impunerea din exterior a stării memorate) sunt necesare circuite logice suplimentare.

Să ne reamintim noțiunea de „poartă logică”. *Poarta logică este un circuit electronic cu un comportament digital, care poate fi „deschisă” sau „închisă” pentru un semnal prin acțiunea (valoarea) altui semnal.*

Ce ar fi dacă bucla ar fi temporar deschisă? Circuitul combinațional s-ar comporta „clasic”, ca orice circuit combinațional (valoarea ieșirii determinată de intrări). Funcția de „memorare a stării” apare doar în momentul în care bucla se închide. Deci, dacă am putea să controlăm deschiderea și închiderea buclei, vom putea controla starea memorată de circuit.

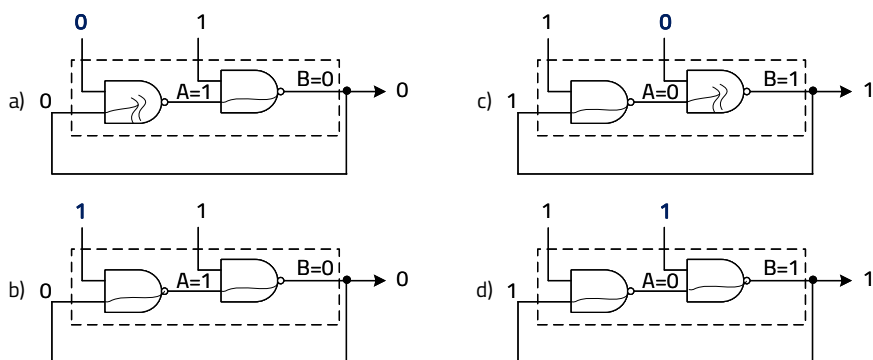
De exemplu, figura 3.4 prezintă funcționarea circuitelor cu două inversoare pe buclă, însă porțile NOT au fost înlocuite cu porți NAND cu două intrări: o intrare conectată la ieșirea celeilalte porți și a doua intrare posibil de controlat din exterior.

Circuitul 3.4-a prezintă stările nodurilor în cazul în care poarta A este blocată (pe intrare 0) și poarta B este deschisă (pe intrare 1). Se observă că ieșirea intră în starea 0. Dacă poarta A se deschide (figura 3.4-b), atunci ieșirea își păstrează starea 0 și circuitul își păstrează această stare.

În schimb, în cazul în care poarta A este deschisă (pe intrare 1) și poarta B este blocată (pe intrare 0), ca în figura 3.4-c, stările nodurilor se modifică iar ieșirea intră în starea 1. Dacă poarta B se deschide (figura 3.4-d), atunci ieșirea își păstrează starea 1 și circuitul își păstrează această stare.

Concluzia este că, în cazul închiderii buclei (ambele intrări în porțile NAND sunt egale cu 1), circuitul își păstrează starea. Însă, starea păstrată este dependentă de locul în care a fost deschisă bucla anterior (figura 3.4-b, d).

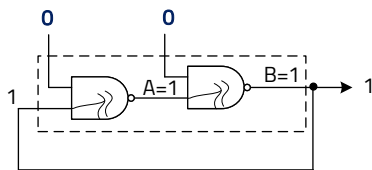
Totuși, circuitul realizat cu două porți NAND (figura 3.4) are o vulne-



**Figura 3.4** Circuit cu două porți NAND în buclă închisă.

- a) Poarta A blocată, poarta B deschisă, ieșirea în stare 0,
- b) Poarta A deschisă, poarta B deschisă, ieșirea memorează starea 0,
- c) Poarta A deschisă, poarta B blocată, ieșirea în stare 1,
- d) Poarta A deschisă, poarta B deschisă, ieșirea memorează starea 1.

rabilitate: în cazul în care ambele porți sunt blocate, (figura 3.5) ambele noduri  $A = B = 1$ , dar starea nu va mai fi păstrată în cazul deschiderii simultane a ambelor porți. Circuitul va reveni fie în starea cu ieșirea 0, fie în starea cu ieșirea 1 în funcție de factori aleatori (zgomot sau care intrare a fost comandată în 1 înaintea celeilalte). Starea  $A = B = 1$  (provenită din ambele intrări egale cu 0) nu este stabilă și ar trebui evitată.

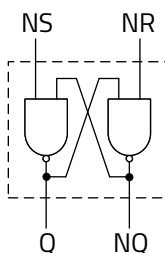


**Figura 3.5** Circuit cu două porți NAND în buclă închisă. Poarta A blocată, poarta B blocată, ieșirea în stare 1, dar starea este instabilă.

### 3.1.1 Latch

#### Latch RS

Circuitul prezentat în figura 3.4 memorează starea determinată de poziția ultimei porți deschise în circuit. Circuitul, redesenat altfel în figura 3.6, se numește „*latch RS*”.



**Figura 3.6** Latch RS. Stările ieșirilor  $Q$  și  $NQ$  sunt complementare. Intrările de control  $NS$  (set) și  $NR$  (reset) sunt active în starea 0.

Dacă  $NS = 0$  (intrare activă) și  $NR = 1$  (intrare inactivă), circuitul intră în starea ( $Q = 1, NQ = 0$ ). Dacă  $NS = 1$  și  $NR = 1$  (ambele intrări inactivate), circuitul își menține starea ( $Q = 1, NQ = 0$ ).

Dacă  $NR = 0$  (intrare activă) și  $NS = 1$  (intrare inactivă), circuitul intră în starea ( $Q = 0, NQ = 1$ ). Dacă  $NR = 1$  și  $NS = 1$  (ambele intrări inactivate), circuitul își menține starea ( $Q = 0, NQ = 1$ ).

Starea  $NS = NR = 0$  nu este stabilă și ar trebui evitată. Dacă  $NS = 0$  și  $NR = 0$  (ambele intrări active), circuitul intră în starea ( $Q = 1, NQ = 1$ ). Această stare nu numai că nu mai îndeplinește condiția de complementaritate, dar nici nu este stabilă. Circuitul părăsește starea imediat ce una din intrări se inactivează. În ipoteza că s-ar inactiva ambele intrări simultan (ceea ce fizic nu este posibil), starea care urmează stării ( $Q = 1, NQ = 1$ ) va fi impredictibilă, fie ( $Q = 1, NQ = 0$ ), fie ( $Q = 0, NQ = 1$ ).

Cum poate fi evitat ca cele două intrări să fie simultan active  $NS = NR = 0$ ?

## Latch RS cu ceas

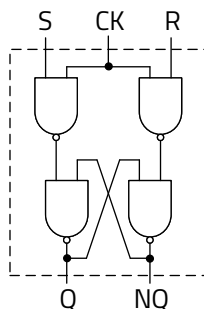
Soluția constă în adăugarea unor porți suplimentare care să conditioneze intrările de comandă astfel încât, bucla circuitului latch să nu primească niciodată pe intrare  $NS = NR = 0$ . Porțile suplimentare introduc un cost suplimentar, însă rezolvă o vulnerabilitate a circuitului care cauzează impredictibilitate. Și „*totul se plătește*”, nu-i așa?

Circuitul din figura 3.7 prezintă două intrări de control  $S$  (set) și  $R$  (reset) active în starea 1, dar relevante numai în cazul în care  $CK = 1$ . În acest caz, cele două porți NAND suplimentare sunt deschise, iar la ieșirea acestora apar semnalele  $S$  și  $R$  negate. Latch-ul cu ceas se comportă similar cu un latch.

În cazul în care  $CK = 0$ , cele două porți NAND suplimentare sunt blocate și prezintă la ieșire starea 1, ceea ce determină latch-ul să își memoreze starea, indiferent de valorile  $S$  și  $R$ .

Semnalul de ceas  $CK$  se presupune a fi periodic, o jumătate de perioadă egal cu 1 iar cealaltă jumătate de perioadă egal cu 0. Comportamentul latch-ului poate fi descris astfel:

- o jumătate de perioadă de ceas ( $CK = 1$ ) latch-ul este deschis și își modifică starea conform condițiilor exterioare  $S$  și  $R$ ,
- o jumătate de perioadă de ceas ( $CK = 0$ ) latch-ul este blocat și își memorează starea.



**Figura 3.7** Latch RS cu ceas. Stările ieșirilor  $Q$  și  $NQ$  sunt complementare. Intrările de control  $S$  (set) și  $R$  (reset) sunt active în starea 1 și sunt relevante doar dacă  $CK = 1$ . Dacă  $CK = 0$ , latch-ul RS cu ceas își memorează starea, ignorând intrările  $S$  și  $R$ .

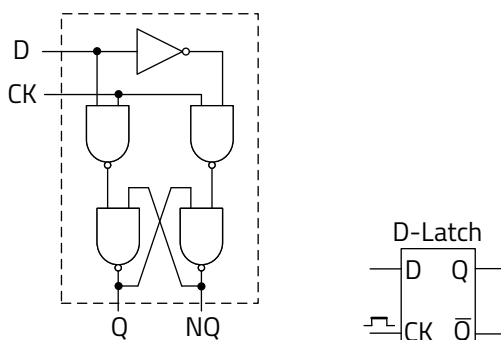


## Latch D

Latch-ul RS cu ceas rezolvă vulnerabilitatea de instabilitate a latch-ului RS, prin introducerea unei noi intrări care determină momentele de memorare a stării și cele de control a stării circuitului. Totuși, rămâne constrângerea ca cele două intrări de control  $S$  și  $R$  să nu se activeze simultan, în perioada când  $CK = 1$ .

Se poate evita acest lucru dacă se pune condiția suplimentară ca cele două intrări  $S$  și  $R$  să fie tot timpul complementare. Acest lucru se poate realiza dacă cele două intrări  $S$  și  $R$  provin din același semnal de intrare, denumit  $D$ , așa ca în figura 3.8.

Circuitul prezentat în figura 3.8 se numește „*latch D*” și are funcționarea descrisă de tabelul 3.1. Denumirea circuitului provine din limba engleză, unde „*to latch*” înseamnă „a zăvorî”.



**Figura 3.8** Latch D. Stările ieșirilor  $Q$  și  $NQ$  sunt complementare. Intrările de control  $S$  (set) și  $R$  (reset) sunt active în starea 1 și sunt relevante doar dacă  $CK = 1$ . Dacă  $CK = 0$ , latch-ul RS cu ceas își memorează starea, ignorând intrările  $S$  și  $R$ .

Latch-ul D zăvorește datele primite pe intrarea  $D$  când ceasul se află pe *palierul activ*  $CK = 1$  și le menține pe ieșirile  $Q$  și  $\bar{Q}$  pe durata *palierului inactiv* al ceasului  $CK = 0$ .

În tabelul 3.1 s-a notat cu  $Q$  starea prezentă și  $Q^+$  starea viitoare a latch-ului. Dacă  $CK = 1$ , latch-ul copiază starea intrării  $D$  și o menține pe durata palierului de ceas  $CK = 0$ .

Tabelul 3.1 poate fi descris mai condensat ca în tabelul 3.2.

Modelul Verilog al latch-ului D se poate descrie cu o specificație `always`

| $CK$ | $D$ | $Q^+$ | Acțiune                                                                                                               |
|------|-----|-------|-----------------------------------------------------------------------------------------------------------------------|
| 1    | 0   | 0     | copiază intrarea, pe durata palierului activ, 1, al semnalului de ceas (latch-ul este deschis)                        |
| 1    | 1   | 1     | copiază intrarea, pe durata palierului activ, 1, al semnalului de ceas (latch-ul este deschis)                        |
| 0    | –   | $Q$   | păstrează starea, pe durata palierului inactiv, 0, al semnalului de ceas (latch-ul este blocat, ignoră intrarea $D$ ) |

Tabelul 3.1

Tabel de funcționare pentru Latch D.

| $CK$ | $Q^+$ |
|------|-------|
| 1    | $D$   |
| 0    | $Q$   |

Tabelul 3.2

Tabel de funcționare condensat, pentru Latch D. Comutare pe palierul de 1 al semnalului de ceas.

având comportamentul prezentat în tabelul 3.2. Re-evaluarea se face la modificarea semnalului de ceas sau a semnalului de intrare  $d$ . Starea este determinată de regula:

- dacă  $ck = 1$ , ieșirea ia valoarea intrării,
- dacă  $ck = 0$ , starea se păstrează.

În Verilog, păstrarea stării se descrie prin lipsa unei atribuirii către ieșire, în cazul neîndeplinirii condiției de comutare.

```

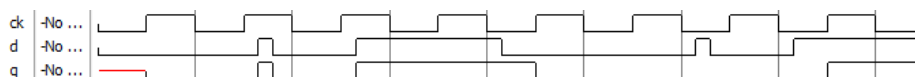
module latch_d (
input      ck, // semnal de ceas (activ pe palierul de 1)
input      d , // intrare latch
output reg q  // iesire latch
);

always @(ck or d) // reevaluare la modificarea ceasului sau
                  // a intrării
if (ck) q <= d;  // daca ck=1, latch transparent,
                  // transmite intrarea la iesire
                  // daca ck=0, pastreaza starea
                  // (nu se actualizeaza)

```

```
endmodule // latch_d
```

**Latch-ul D este activ pe palierul semnalului de ceas.** Se spune că este „transparent” pe palier pentru că în acest interval de timp starea internă este actualizată conform intrării. Latch-ul este blocat în cealaltă jumătate de perioadă de ceas, memorând starea și ignorând intrarea. Totuși, în timp ce  $CK = 1$ , orice modificare a intrării  $D$  este transpusă la ieșire. Acest lucru se poate observa pe formele de undă prezentate în figura 3.9.



**Figura 3.9** Forme de undă simulate pentru latch D. Un puls apărut pe intrarea  $D$  pe durata palierului  $CK = 1$  se transmite imediat la ieșire. În schimb, un puls apărut pe intrarea  $D$  pe durata palierului  $CK = 0$  se ignoră și latch-ul își păstrează starea.

### 3.1.2 Bistabile

#### Bistabilul D

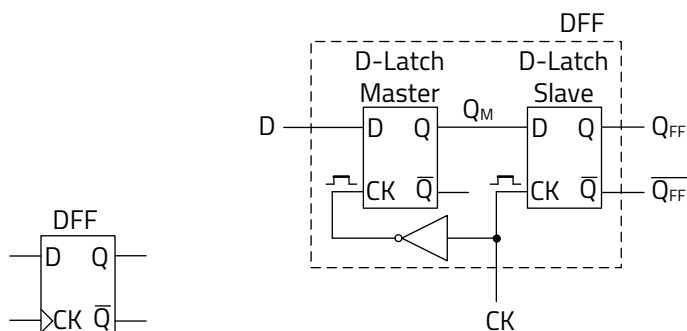
Bistabilul D are în structura sa două latch-uri înseriate, cu semnale de ceas complementare. Din acest motiv, cele două latch-uri sunt deschise sau blocate în contratimp, astfel încât bistabilul nu este deschis decât în momentul determinat de un front de ceas (momentul comutării între cele două stări logice 0 și 1).

Bistabilul eșantionează datele primite pe intrarea  $D$  în momentul *frontului activ* al ceasului  $CK = \uparrow$  și le menține pe ieșirile  $Q$  și  $\overline{Q}$  pe toată durata perioadei de ceas.

Symbolul bloc al bistabilului D și structura acestuia sunt prezentate în figura 4.22. În limba engleză bistabilul se numește „flip-flop”. Din acest motiv, bistabilul D se prescurtează frecvent „DFF”.

În tabelul 3.3 este prezentat tabelul de funcționare al bistabilului D. S-a notat cu  $Q$  starea prezentă și cu  $Q^+$  starea viitoare a bistabilului. Dacă  $CK = \uparrow$ , bistabilul copiază starea intrării  $D$  și o menține pe durata unei perioade de ceas. În rest, bistabilul memorează starea. Altfel spus, în momentul frontului activ starea se eșantionează și apoi se memorează timp de o perioadă de ceas (nu doar o jumătate de perioadă, ca la latch-ul D).

Tabelul 3.4 prezintă condensat tabelul de funcționare al bistabilului D.



**Figura 3.10** Bistabil D. Simbol și structură.

| $CK$       | $D$ | $Q^+$ | Acțiune                                                       |
|------------|-----|-------|---------------------------------------------------------------|
| $\uparrow$ | 0   | 0     | copiază intrarea, în momentul frontului crescător al ceasului |
| $\uparrow$ | 1   | 1     | copiază intrarea, în momentul frontului crescător al ceasului |
| 0/1        | –   | $Q$   | păstrează starea, între două fronturi active ale ceasului     |

*Tabelul 3.3*

Tabel de funcționare pentru bistabilul D.

Într-un sistem sincron, **bistabilul D întârzie semnalul între intrare și ieșire cu un tact**. Așa se justifică denumirea D („Delay = Întârziere”, engl.). O aplicație a bistabilului D constă din realizarea unei linii de întârziere a semnalelor cu un anumit număr de perioade de tact. Figura 3.11 prezintă un circuit format din 4 bistabile D, conectate astfel încât intrarea unuia să provină de la ieșirea bistabilului precedent. Se observă că fiecare bistabil întârzie semnalul de ieșire cu o perioadă de ceas, față de intrare, generând astfel la ieșire un semnal cu o formă similară celui de intrare, dar defazat în timp cu 4 perioade ale semnalului de ceas.

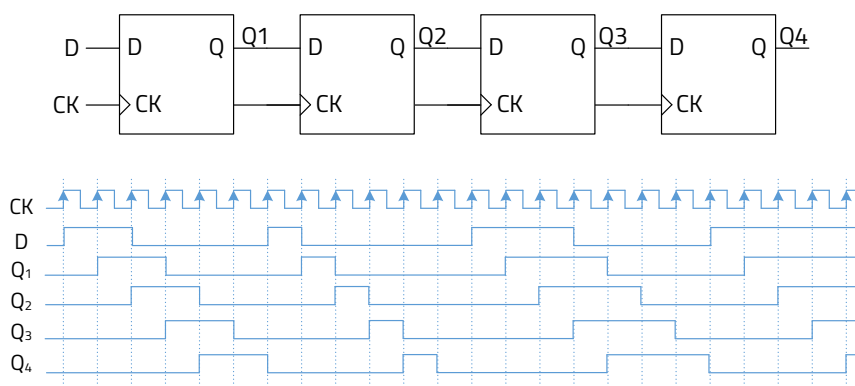
Bistabilul D poate avea o intrare suplimentară de validare a funcționării. Dacă  $E = 1$  („Enable = a permite, a valida”, engl.), bistabilul D cu validare funcționează ca un bistabil D convențional. Altfel, dacă  $E = 0$ , bistabilul D cu validare își păstrează starea, indiferent de intrarea  $D$ . Tabelul ce descrie funcționarea bistabilului D cu validare este 3.5.

Tabelul 3.6 prezintă condensat tabelul de funcționare al bistabilului D

| $D$ | $Q^+$ |
|-----|-------|
| 0   | 0     |
| 1   | 1     |

Tabelul 3.4

Tabel de funcționare condensat pentru bistabilul D. Comutare exclusiv pe frontul pozitiv al semnalului de ceas.



**Figura 3.11** Linie de întârziere implementată cu bistabile D. Fiecare bistabil întârzie semnalul de la ieșire cu o perioadă de ceas, față de intrare.

cu validare.

Intrarea de validare poate fi interpretată și ca intrare care validează încărcarea datelor în bistabil (datele fiind menținute mai multe perioade de tact, între două pulsuri succesive de încărcare).

În structura bistabilului D cu validare (figura 3.12) se regăsesc un bistabil D și un multiplexor 2:1 cu intrarea  $E$  conectată la selecție. Dacă  $E = 1$ , la intrarea  $D$  se transmite intrarea de date, altfel, dacă  $E = 0$ , la intrarea bistabilului se aduce prin buclă starea sa prezentă  $Q$ , pentru a fi menținută încă o perioadă de ceas.

Modelul Verilog al bistabilului D se poate descrie cu o specificație `always` care descrie comportamentul prezentat în tabelul 3.6. Re-evaluarea se face exclusiv în momentul frontului crescător al semnalului de ceas, modelat cu specificația `@(posedge clk)` („positive edge” = „front pozitiv/crescător”, engl.).

| $CK$       | $E$ | $D$ | $Q^+$ | Acțiune                                                       |
|------------|-----|-----|-------|---------------------------------------------------------------|
| $\uparrow$ | 1   | 0   | 0     | copiază intrarea, în momentul frontului crescător al ceasului |
| $\uparrow$ | 1   | 1   | 1     | copiază intrarea, în momentul frontului crescător al ceasului |
| $\uparrow$ | 0   | —   | $Q$   | păstrează starea, indiferent de intrarea $D$ (este invalidat) |
| 0/1        | —   | —   | $Q$   | păstrează starea, între două fronturi active ale ceasului     |

Tabelul 3.5

Tabel de funcționare pentru bistabilul D cu validare.

| $E$ | $D$ | $Q^+$ |
|-----|-----|-------|
| 1   | 0   | 0     |
| 1   | 1   | 1     |
| 0   | —   | $Q$   |

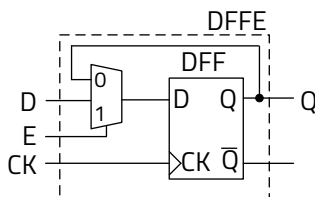
  

| $E$ | $Q^+$ |
|-----|-------|
| 1   | $D$   |
| 0   | $Q$   |

Tabelul 3.6

Tabel de funcționare condensat pentru bistabilul D cu validare.  
Comutare exclusiv pe frontul pozitiv al semnalului de ceas.

Starea bistabilului D este determinată de regula: în momentul  $CK = \uparrow$  ieșirea ia valoarea intrării, altfel starea se păstrează timp de o perioadă de ceas, până la următorul front crescător.



**Figura 3.12** Structura bistabilului D cu validare (bistabil D și mux 2:1).

În Verilog, păstrarea stării se descrie prin lipsa unei atribuirii către ieșire, în cazul neîndeplinirii condiției de comutare @(posedge clk).

```
module dff (
input      ck, // semnal de ceas (activ pe front crescator)
input      d , // intrare bistabil D
output reg q   // iesire latch
);

always @(posedge ck) // reevaluare in momentul frontului
                    // crescator al semnalului de ceas _|-
    q <= d;

endmodule // dff
```

Modelul de bistabil cu validare conține suplimentar o condiție conform căreia ieșirea se modifică doar dacă bistabilul este validat.

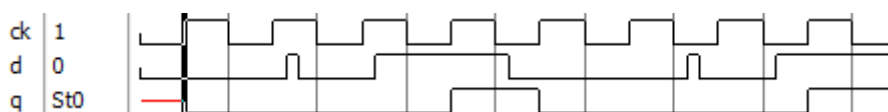
```
module dffe (
input      ck, // semnal de ceas (activ pe front crescator)
input      d , // intrare bistabil D
input      e , // intrare de validare
output reg q   // iesire latch
);

always @(posedge ck)
if (e) q <= d; // pastreaza starea daca e=0

endmodule // dffe
```

**Bistabilul D este activ pe frontul semnalului de ceas.** Se spune că este „transparent” pe frontul crescător pentru că în acest moment starea

internă este actualizată conform intrării. Bistabilul D este apoi blocat pentru o perioadă de ceas, memorând starea și ignorând intrarea. Acest lucru se poate observa pe formele de undă prezentate în figura 3.13. **Bistabilul**



**Figura 3.13** Forme de undă simulate pentru bistabil D. Intrarea  $D$  este eșantionată exclusiv în momentul frontului crescător al semnalului de ceas  $CK \uparrow$ , apoi starea este păstrată timp de o perioadă de ceas.

**memorează starea intrării  $D$  pe durata unei perioade de ceas.** Intrarea  $D$  este eșantionată în momentul frontului crescător de ceas.

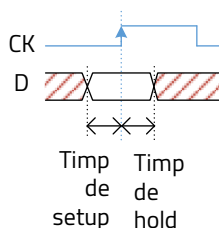
Ce se întâmplă dacă intrarea  $D$  comută exact în acel moment? Ce valoare este eșantionată dacă intrarea  $D$  comută exact atunci din 0 în 1 sau din 1 în 0? Pentru a evita această situație care generează o lipsă de predictibilitate a stării memorate, este necesară îndeplinirea unor condiții suplimentare: semnalul de intrare  $D$  să fie stabil în momentul frontului activ al semnalului de ceas. Aceste condiții, denumite în limba engleză „*setup*” („a pune pe”) și „*hold*” („a menține”) semnifică necesitatea ca intrarea  $D$  să fie stabilă (să nu comute) un pic înainte de momentul frontului și un pic după acest moment. Astfel, în momentul frontului activ intrarea este eșantionată corect și bistabilul intră cu certitudine într-o stare stabilă 0 sau 1. În figura 3.14 sunt definite noțiunile de „*setup*” și „*hold*”:

- „*timp de setup*” = timpul de dinaintea frontului activ de ceas în care intrarea  $D$  trebuie să fie stabilă;
- „*timp de hold*” = timpul de după frontul activ de ceas în care intrarea  $D$  trebuie să fie stabilă.

Neîndeplinirea condițiilor de „*setup*” și „*hold*” determină o incertitudine a stării eșantionate și o întârziere a bistabilului de a intra în una din cele două stări stabile. Această **stare de „metastabilitate”** este preluată de un bistabil de la o intrare și posibil de propagat în întreg circuitul de la un bistabil la altul. Starea de metastabilitate este o stare de incertitudine care trebuie evitată prin tehnici de sincronizare a semnalelor.

În figura 3.11 se remarcă faptul că semnalul  $D$  pare că se modifică exact în momentul frontului activ de ceas. Adică nu sunt îndeplinite condițiile de

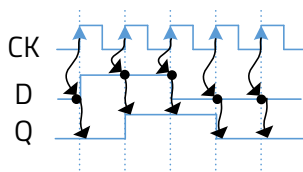




**Figura 3.14** Definirea timpilor de setup și hold.

„*setup*” și „*hold*”. Formele de undă provin dintr-o simulare. De fapt, realistic, datorită întârzierii inerentente la propagarea semnalelor prin bistabil, semnalele de ieșire din bistabil sunt întârziate față de frontul de ceas. Întotdeauna, semnalul de ieșire dintr-un bistabil este întârziat față de frontul activ de ceas, iar această întârziere este mai mare decât timpul de hold. Altfel spus, într-un sistem sincron, în care toate bistabilele primesc același semnal de ceas, condiția de „*hold*” este satisfăcută.

Ca o consecință a faptului că ieșirea bistabilului este întârziată față de frontul activ al semnalului de ceas, pe formele de undă provenite din simulare (care consideră a fi zero timpul de propagare al bistabilelor) **valoarea eșantionată este valoarea de dinaintea frontului de ceas**. Această valoare va fi menținută la ieșirea bistabilului pe durata unei perioade de ceas, după cum este prezentat în figura 3.15.



**Figura 3.15** Detalii cu eșantionarea valorii intrării în momentul frontului crescător de ceas și menținerea acesteia pe ieșire pentru o perioadă de ceas.



### Bistabilul D:

#### Un suflet hotărât într-o lume schimbătoare

Într-o lume digitală, unde totul este presat de semnale și tensiuni ca o mână de fier într-o mânășă de catifea, există un bistabil aparte, cu numele „D”, un gentleman discret ce nu se lasă ușor influențat. Spre deosebire de alți frați ai săi, care se lasă purtați de voința semnalelor și ale selecțiilor, bistabilul D își urmează o regulă strictă: *„Sunt un om de onoare, nu mă las schimbat fără un semnal de încredere!”*

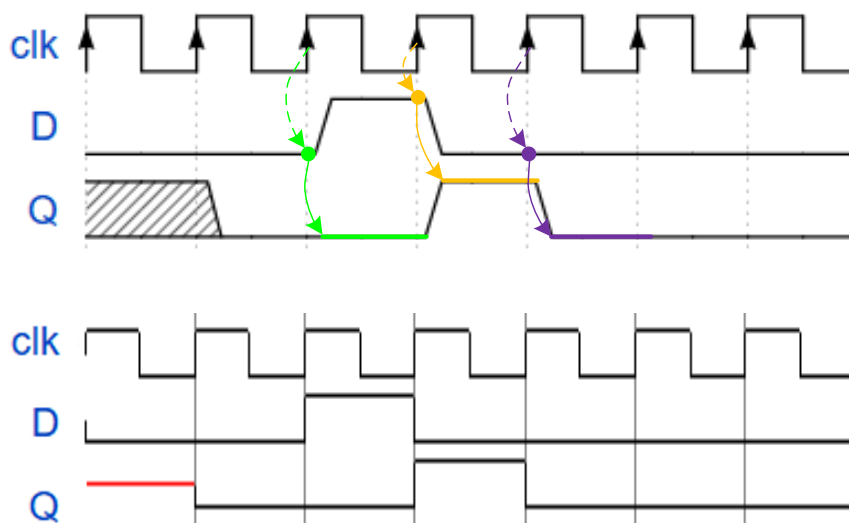
Așadar, la momentul potrivit, când frontul semnalului de ceas sosește, bistabilul D ia în considerare un singur lucru: valoarea de la intrarea sa „D”. Fără fasoane, fără ezități, pur și simplu se decide: *„În următoarea perioadă de ceas voi fi 0 sau 1, în funcție de ce mi-ai spus!”* Iar odată ce semnalul de ceas trece, în sfârșit, el se stabilește ferm într-una dintre cele două stări, cu o determinare ca și cum ar fi ales ceasul său de pe masă.

Și, spre deosebire de alți bistabili care se lasă influențați de o mulțime de semnale și condiții, bistabilul D are o viață simplă: în fața unui semnal de ceas, el se hotărăște și rămâne fidel acelei alegeri, fără a se mai întoarce asupra deciziei. Dacă semnalul D era 0, rămâne 0; dacă semnalul D era 1, rămâne 1. Simplu și eficient, ca un țăran care știe că trebuie să semene grâul înainte de ploaie.

*„Dacă vrei schimbări, trebuie să așteptați următorul ceas!”* – aceasta ar fi filozofia bistabilului D, pe care o ține cu mândrie la brâu.

Și astfel, el trăiește în pace, fără a-și pune întrebări, cu un răspuns clar și rapid: **„Eu am ales! Și, dacă nu sunt deranjat, rămân așa!”**

*Text generat cu ChatGPT.*



**Figura 3.16** Înțelegerea formelor de undă față de frontul de ceas.

Sus: forme de undă desenate, justificare.

Jos: forme de undă simulate.

## Bistabilul RS

Bistabilul RS are două intrări de date, care permit în mod explicit trecerea bistabilului în cele două stări 0 și 1. Funcționarea bistabilului RS este descrisă de tabelul 3.7.

Tabelul 3.8 prezintă condensat tabelul de funcționare al bistabilului RS.

Bistabilul RS se utilizează pentru cazurile în care trecerea între stările 0 și 1 se face pe baza a două comenzi de control diferite.

Modelul de bistabil RS impune o prioritate a intrărilor în cazul în care ambele sunt active. În cazul în care nicio intrare de control nu este activată, bistabilul RS își menține starea încă o perioadă.

```
module rsff (
input      ck      , // ceas
input      r      , // reset sincron
input      s      , // set sincron
output reg q      // iesire
);
```

| $CK$       | $R$ | $S$ | $Q^+$ | Acțiune                                                   |
|------------|-----|-----|-------|-----------------------------------------------------------|
| $\uparrow$ | 1   | –   | 0     | resetează starea, 0                                       |
| $\uparrow$ | 0   | 1   | 1     | setează starea, 1                                         |
| $\uparrow$ | 0   | 0   | $Q$   | păstrează starea, nicio comandă activată                  |
| 0/1        | –   | –   | $Q$   | păstrează starea, între două fronturi active ale ceasului |

Tabelul 3.7

Tabel de funcționare pentru bistabilul RS.

| $R$ | $S$ | $Q^+$ |
|-----|-----|-------|
| 1   | –   | 0     |
| 0   | 1   | 1     |
| 0   | 0   | $Q$   |

Tabelul 3.8

Tabel de funcționare condensat pentru bistabilul RS. Comutare exclusiv pe frontul pozitiv al semnalului de ceas.

```

always @(posedge ck)
if (r)      q <= 1'b0; else    // reset sincron, prioritar
if (s)      q <= 1'b1;        // set sincron

endmodule // rsff

```

## Bistabilul T

Bistabilul T, ca orice bistabil, comută exclusiv pe frontul activ al semnalului de ceas. Regula de comutare este prezentată în tabelul 3.9.

| $CK$       | $T$ | $Q^+$     | Acțiune                                                   |
|------------|-----|-----------|-----------------------------------------------------------|
| $\uparrow$ | 0   | $Q$       | păstrează starea                                          |
| $\uparrow$ | 1   | $\bar{Q}$ | complementează starea                                     |
| 0/1        | –   | $Q$       | păstrează starea, între două fronturi active ale ceasului |

Tabelul 3.9

Tabel de funcționare pentru bistabilul T.

Tabelul 3.10 prezintă condensat tabelul de funcționare al bistabilului T.

| $T$ | $Q^+$          |
|-----|----------------|
| 1   | $\overline{Q}$ |
| 0   | $Q$            |

Tabelul 3.10

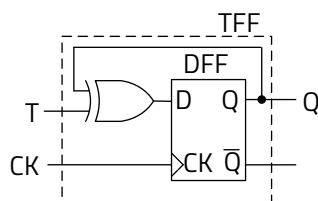
Tabel de funcționare condensat pentru bistabilul T. Comutare exclusiv pe frontul pozitiv al semnalului de ceas.

Funcționarea bistabilului T („Toggle = Comutare”, engl.) se poate descrie astfel:

dacă  $T = 0$  bistabilul își păstrează starea,  
 dacă  $T = 1$  bistabilul își complementează/neagă starea.

Se remarcă faptul că, datorită faptului că bistabilul T își determină starea următoare nu doar pe baza intrărilor ci și a stării prezente, bistabilul T are nevoie de a se inițializa asincron (independent de semnalul de ceas) într-o stare cunoscută. Pentru aceasta, există o intrare suplimentară la a cărei activare (independent de ceas) aduce bistabilul într-o stare cunoscută 0 sau 1.

În structura bistabilului T (figura 3.17) se regăsesc un bistabil D și o poartă XOR între intrarea  $T$  și starea prezentă  $Q$ . Dacă  $T = 1$ , la intrarea  $D$  se transmite  $\overline{Q}$ , altfel, dacă  $T = 0$ , la intrarea bistabilului se aduce prin buclă starea sa prezentă  $Q$ , pentru a fi menținută încă o perioadă de ceas.



**Figura 3.17** Structura bistabilului T (bistabil D și poartă XOR).

Modelul de bistabil T prezintă intrarea  $rst\_n$  (reset asincron activ în stare 0) care determină intrarea bistabilului în stare 0, independent de semnalul de ceas (în mod asincron). Modelul prezintă doar comportamentul în care

$T = 1$  (complementează starea prezentă) și nu specifică nimic pentru cazul  $T = 0$ , deoarece în acest caz starea se menține aceeași, nu se re-evaluează.

```

module tff (
input      ck      , // ceas
input      rst_n   , // reset asincron, activ low
input      t       , // intrare sincrona
output reg q       // iesire
);

always @(posedge ck or negedge rst_n)
if (~rst_n) q <= 1'b0; else // reset asincron
if (t)      q <= ~q;       // complement, sincron

endmodule // tff

```

O aplicație a bistabilului T constă în obținerea unui semnal cu perioadă dublă față de semnalul de ceas, prin conectarea acestuia cu intrarea  $T = 1$ . Această proprietate este folosită și la **realizarea numărătoarelor binare**.

## Bistabilul JK

Bistabilul JK combină funcționalitatea unui bistabil RS cu cea a unui bistabil T. Regula de comutare este prezentată în tabelul 3.11.

| $CK$ | $J$ | $K$ | $Q^+$     | Acțiune                                                   |
|------|-----|-----|-----------|-----------------------------------------------------------|
| ↑    | 0   | 0   | $Q$       | păstrează starea.                                         |
| ↑    | 0   | 1   | 0         | resetează starea, 0                                       |
| ↑    | 1   | 0   | 1         | setează starea, 1                                         |
| ↑    | 1   | 1   | $\bar{Q}$ | complementează starea.                                    |
| 0/1  | –   | –   | $Q$       | păstrează starea, între două fronturi active ale ceasului |

Tabelul 3.11

Tabel de funcționare pentru bistabilul JK.

**Bistabilul T:****Povestea unui bătaios cu două fețe**

Așezat în colțul tăcut al unui circuit digital, bistabilul T își trăiește viața cu o filosofie mai... volubilă, să-i spunem. Spre deosebire de fratele său D, care se simte confortabil într-o stare fixă, bistabilul T trăiește după o lege simplă: *„Când îmi vine ceasul, mă transform, după cum o fi voia lui T!”* Sau, mai pe românește, *„Schimbă-mă, dacă poți!”*.

În fiecare clipă, bistabilul T stă cu ochii pe semnalul de ceas. Atunci când acesta trece pe acolo, el se întreabă, fără grabă: *„Oare mă voi schimba azi? Oare rămân 0 sau devin 1?”* Dacă semnalul de la intrare este 1, bistabilul T se răstoarnă precum un campion la lupte, schimbându-și starea ca un soldat care face un pas înapoi doar pentru a lovi mai puternic.

*„Mă simt răsturnat... da', de astăzi sunt altfel!”* – aceasta este filosofia sa, una fără regret și fără teamă. Aș putea să spun chiar că bistabilul T este un fel de *„mănuitor al schimbărilor”* într-un circuit. Dacă nu are semnalul de schimbare, rămâne cum este; dar când vine acel semnal, se răstoarnă cu o înflăcărare demnă de o adevărată erupție.

Așadar, de fiecare dată când ceasul bate, bistabilul T își spune, cu glas grav: *„Astăzi voi fi... ce am fost ieri? Sau voi fi cu totul altceva?”*

Într-o lume unde totul se întâmplă cu un scop și o direcție, bistabilul T, acest adevărat *„bătaios cu două fețe”*, nu face excepție de la regulă. Dacă nu e inițializat corespunzător, el ajunge într-o stare confuză, ca un călător fără hartă și fără Google Maps într-un oraș necunoscut. Fără o direcție clară, el nu știe nici de unde vine, nici încotro să se îndrepte. Se poate învârti în cerc, răsturnându-se de fiecare dată la semnalul de ceas, dar fără să știe care este direcția corectă. Așa că, pentru a-i da sens vieții, trebuie neapărat să-l resetăm, să-i oferim acel punct de plecare, acea stare de început, fără de care, sincer, n-ar înțelege nimic din ce se întâmplă în jurul său. *„Fără inițializare, viața mea e un haos pur”*, ar spune bistabilul T, dacă ar avea voce, *„nu știu dacă sunt 0 sau 1, așa că nu sunt stabil deloc!”*

*Text generat cu ChatGPT.*

Tabelul 3.12 prezintă condensat tabelul de funcționare al bistabilului JK.

| $J$ | $K$ | $Q^+$          |
|-----|-----|----------------|
| 0   | 0   | $Q$            |
| 0   | 1   | 0              |
| 1   | 0   | 1              |
| 1   | 1   | $\overline{Q}$ |

Tabelul 3.12

Tabel de funcționare condensat pentru bistabilul JK. Comutare exclusiv pe frontul pozitiv al semnalului de ceas.

Bistabilul JK funcționează ca un bistabil T dacă se realizează conectarea intrărilor  $J = K = T$ .

Bistabilele pot avea facilitatea de inițializare asincronă într-o anumită stare logică. Intrările asincrone *Set* sau *Reset* sunt prioritare față de semnalul de ceas (își realizează funcția independent de apariția frontului activ al semnalului de ceas). În simbolurile bloc, există intrări dedicate pentru intrările asincrone. Intrările asincrone pot fi active fie în starea 1 fie în starea 0.

În practică, bistabilele se modelează în Verilog și se obțin prin sinteză structurile necesare, fără a fi unele standardizate.



### 3.1.3 Inițializarea bistabilelor



#### Din haos în determinism:

##### Botezul bistabilului

La început, bieții bistabili nu știau nici ei ce-s — 0, 1, sau o stare metafizică incertă, ca studentul din anul întâi care a ajuns la laborator crezând că vine la sport.

Dar vine un RESET hotărât, ca un profesor cu catalogul sub braț, și le zice clar: „Destul cu visarea! Tu ești 0, tu ești 1 și gata.”

Și de atunci, cu fruntea sus și ceasul în piept, bistabilul își trăiește viața logică — una simplă, dar stabilă.

Că doară... **viața începe numa' după reset.**

*Text generat cu ChatGPT.*

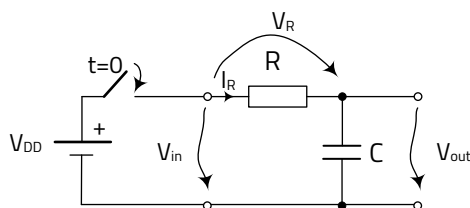
Bistabilele au două stări stabile. Există intrări care controlează evoluția unui bistabil dintr-o anumită stare. Există însă și cazuri (cum este bistabilul T) în care starea viitoare este determinată și de starea curentă. Deci, este nevoie de a se asigura o stare cunoscută (0 sau 1), înainte de a se aplica o relație care determină starea viitoare pe baza stării prezente.

Se poate considera că la apariția tensiunii de alimentare bistabilul este într-o stare necunoscută. Este necesar un circuit suplimentar care să aducă bistabilul într-o stare cunoscută imediat ce este alimentat. Acest circuit este denumit „**Power-On Reset**” (POR).

Prin intermediul circuitului POR, se asigură intrarea tuturor bistabilelor într-o stare cunoscută, înainte de aplicarea impulsurilor de ceas care determină modificarea stării. Ideea circuitului POR este de a genera un puls, asincron față de semnalul de ceas, imediat după apariția tensiunii de alimentare. Pulsul este obținut cu un circuit  $RC$  de integrare, similar celui prezentat în figura 3.18.

La aplicarea tensiunii de alimentare  $V_{DD}$ , condensatorul  $C$  începe să se încarce, generând un curent  $I_R$  prin circuit. Sarcina acumulată pe condensator determină creșterea tensiunii  $V_{OUT}$  și scăderea tensiunii  $V_R$ .

$$V_{OUT} + V_R = V_{DD} = \text{constant}$$



**Figura 3.18** Circuit RC de integrare utilizat pentru Power-On Reset.

Ca efect, curentul  $I_R = V_R/R$  începe să scadă până când condensatorul este încărcat la maximum determinând  $V_{OUT} = V_{DD}$  și  $V_R = 0$ . Rata de creștere a tensiunii pe condensator este determinată de valorile elementelor pasive de circuit  $R$  și  $C$ . Dacă valorile  $R$  sau  $C$  sunt mai mari, încărcarea condensatorului este mai lentă, panta de creștere a tensiunii  $V_C = V_{OUT}$  este mai mică, astfel încât tensiunea de ieșire  $V_{OUT}$  va ajunge la valoarea  $V_{DD}$  mai târziu.

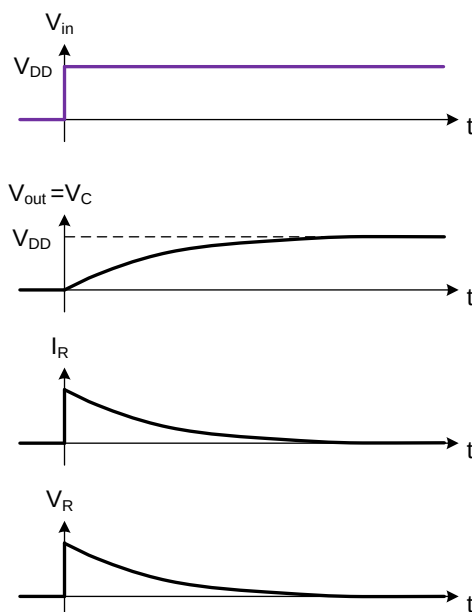
Figura 3.19 prezintă grafic formele de undă asociate circuitului POR din figura 3.18, ca regim tranzitoriu determinat de apariția tensiunii de alimentare (închiderea circuitului la momentul  $t = 0$ ).

În figura 3.20 este prezentat un circuit pe baza principiului POR menționat, care include un buffer ([trigger Schmitt](#)), utilizat pentru formatarea digitală a tensiunii pe condensator pentru a genera un impuls digital de reset asincron. Ieșirea inversorului ( $V_{OUT}$ ) se aplică pe intrarea asincronă de reset, activă *low*, a bistabilelor cu scopul de a le aduce în starea 0 imediat după aplicarea tensiunii de alimentare.

Adițional, circuitul prevede un buton (normal deschis) la a cărui apăsare se generează un alt impuls de reset asincron, prin descărcarea condensatorului și reîncărcarea acestuia după relaxarea butonului.

Figura 3.21 prezintă forme de undă ale semnalelor prin circuitul din figura 3.20, imediat după aplicarea tensiunii de alimentare și ca efect al apăsării și relaxării butonului de reset. Se observă apariția pulsurilor de reset asicron, în starea 0, care pot fi utilizate pentru inițializarea bistabilelor.

Dacă se consideră un prag pentru valoarea  $V_{OUT}$ , se poate observa că, dacă panta este mai mică ( $R$  și  $C$  mai mari), pulsul de pe ieșire este mai lat. Deci, lățimea pulsului de reset se poate controla prin valorile  $R$  și  $C$ .



**Figura 3.19** Forme de undă pentru circuitul din figura 3.18, după închiderea întrerupătorului.

## 3.2 Automate secvențiale

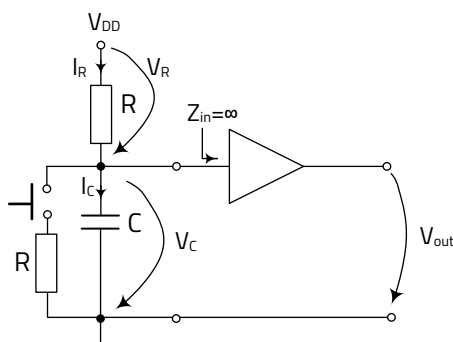
Circuitele logice secvențiale (**CLS**) sunt circuite electronice a căror funcție de transfer este dependentă de starea acestora. Spre deosebire de circuitele logice combinaționale, valorile logice ale ieșirilor sunt determinate atât de intrări cât și de starea curentă a circuitului. Stările unui circuit logic secvențial sunt finite și sunt determinate de numărul elementelor de memorare a stării (bistabile) existente în componența acestora.

Matematic, circuitele logice secvențiale sunt definite ca o mulțime având 5 elemente:

$$CLS = \{X, Y, Q, f, g\}$$

unde s-a notat:

- $X$  mulțimea combinațiilor de intrare.
- $Y$  mulțimea combinațiilor de ieșire.



**Figura 3.20** Circuit POR cu buton de reset, care generează un puls de reset asincron, activ în 0.

- $Q$  mulțimea stărilor CLS.
- $f : X \times Q \rightarrow Q$  funcția de tranziție a stării. Starea viitoare este determinată pe baza intrării și a stării curente.
- $g : X \times Q \rightarrow Y$  funcția ieșirii. Ieșirea este determinată pe baza intrării și a stării curente.

Fizic, circuitele logice secvențiale sunt formate din circuite logice combinaționale care implementează funcțiile  $f : X \times Q \rightarrow Q$  și  $g : X \times Q \rightarrow Y$  peste care se închide o buclă ce include un element de memorare a stării, implementat cu bistabile.

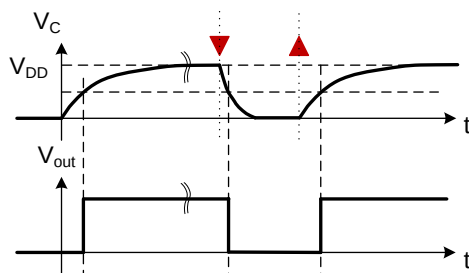
Circuitele logice secvențiale au un **semnal de ceas** care marchează momentele de trecere de la starea curentă la starea viitoare.

Elementul esențial al circuitelor logice secvențiale este existența buclei prin care se determină starea viitoare a circuitului, pe baza stării curente și a valorilor de la intrare.

Existența buclei în care se memorează starea curentă conferă un comportament diferit de circuitele combinaționale.

CLS evoluează într-un spațiu finit de stări iar valorile prezentate la ieșire pot fi diferite, chiar în cazul unor intrări constante, deoarece circuitul își modifică starea.

Pentru a înțelege acest comportament, să considerăm exemplul unui om care răspunde periodic la întrebarea: „Ți-e somn?” Probabil că răspunde



**Figura 3.21** Forme de undă pentru circuitul din figura 3.20, după aplicarea tensiunii de alimentare și după apăsarea (săgeata în jos) și relaxarea (săgeata în sus) butonului de reset.

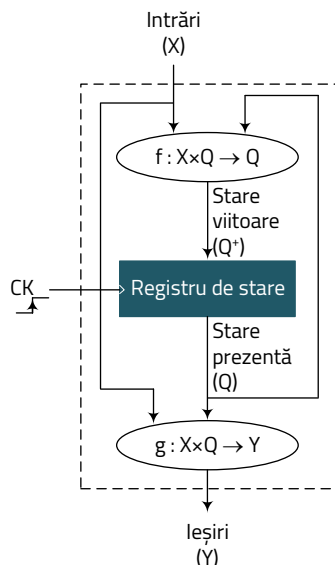
„Nu” de multiple ori, însă, dacă se continuă cu aceeași întrebare, la un moment dat dă alt răspuns, „Da”, chiar dacă aparent nu s-a întâmplat decât trecerea timpului. Răspunsul diferit la aceeași întrebare nu a fost determinat de un eveniment extern, ci de faptul că starea omului s-a modificat în timp.

Automatele sunt circuite logice secvențiale cu un număr finit de stări, structurate pe baza modelului matematic prezentat, sub forma unei bucle ce conține următoarele circuite (figura 3.23).

- **Circuit logic combinațional pentru determinarea stării viitoare**, pe baza intrărilor și a stării prezente, care implementează funcția de tranziție a stării,  $f : X \times Q \rightarrow Q$  (CLC de stare).
- **Element de memorare a stării curente**, între evenimentele ce determină re-evaluarea stării, realizat cu multiple instanțe de bistabile (registru de stare).
- **Circuit logic combinațional pentru determinarea valorilor ieșirilor**, pe baza intrărilor și a stării prezente, care implementează funcția de ieșire  $g : X \times Q \rightarrow Y$  (CLC de ieșire).

Automatele în care ieșirile depind atât de starea curentă cât și de intrări (conform ecuației matematice a funcției  $g : X \times Q \rightarrow Y$ ) se numesc „de tip **Mealy**” (figura 3.23).

Într-un caz particular, în care ieșirile depind numai de starea curentă  $g : Q \rightarrow Y$ , se numesc „de tip **Moore**” (figura 3.23).



**Figura 3.22** Simbolul bloc al circuitelor logice secvențiale (CLS).

Dacă automatele trec ieșirile printr-un element de memorare, astfel încât sunt activate cu o perioadă de ceas mai târziu, se numesc „*automate cu întârziere*” (figura 3.24).

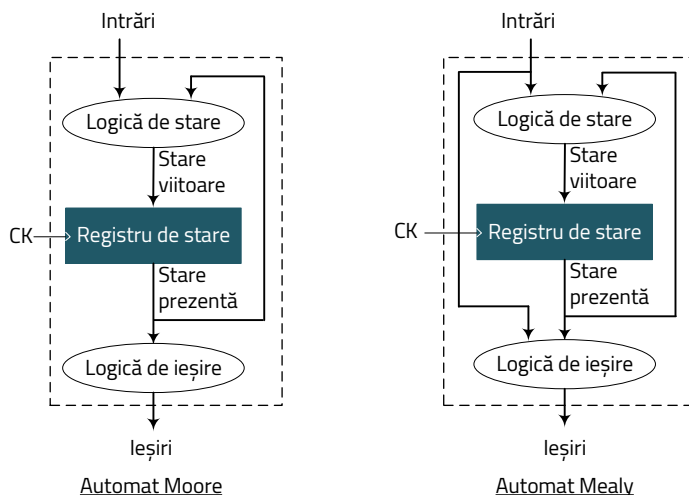
După întârzierea ieșirilor față de intrări, automatele se clasifică în:

- *automatele imediate*, ieșirea este generată dintr-un circuit combinațional și
- *automatele cu întârziere*, ieșirea este generată de un circuit combinațional urmat de un element de memorare.

### 3.3 Reprezentarea automatelor

Comportamentul automatelor trebuie să descrie funcțiile implementate de acestea într-o formă intuitivă pentru evidențierea stării automatelor și a evoluției acestora, în funcție de valorile intrărilor.

Formele grafice au ca scop proiectarea și depanarea algoritmului de evoluție a automatului în spațiul stărilor. Ulterior, comportamentul este



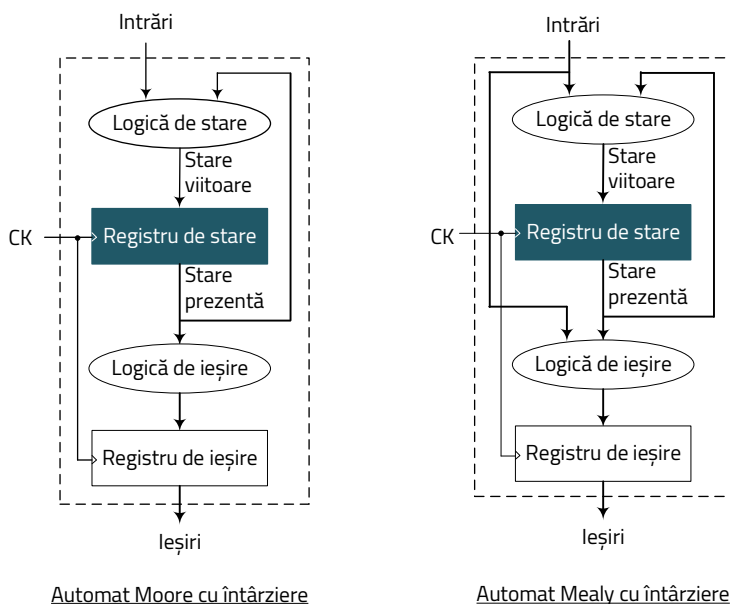
**Figura 3.23** Structuri de automate imediate. Se observă că la automatele Mealy ieșirea depinde și de intrare, nu doar de stare.

descriș într-un limbaj de descriere hardware și convertit la o structură de porți logice și bistabile, prin utilizarea unor aplicații software specializate în sinteză de hardware.

### 3.3.1 Tabele de tranziții

Tabelul de tranziții include informații complete despre comportamentul automatului într-o formă ușor de utilizat pentru implementare. Tabelul conține următoarele secțiuni:

- *Intrări*, câte o coloană pentru fiecare intrare. Valorile logice ale intrărilor pot fi 0 și 1 sau pot fi considerate indiferente pentru tranziția din anumită stare (caz în care valoarea se marchează cu  $X$  sau  $-$ , indiferent).
- *Stare prezentă*, pe o coloană denumirea generică a stării curente și pe alte coloane (câte una pentru fiecare bit de cod) codul stării prezente.
- *Stare viitoare*, pe o coloană denumirea generică a stării viitoare și pe alte coloane (câte una pentru fiecare bit de cod) codul stării viitoare.
- *Ieșiri*, câte o coloană pentru fiecare ieșire. Valorile logice ale ieșirilor pot fi dependente doar de stare (ieșiri Moore, apar identic pe toate liniile



**Figura 3.24** Structuri de automate cu întârziere. Se observă apariția suplimentară a elementului de întârziere (registru) pentru ieșiri.

asociate unei stări prezente) sau dependente și de intrări (ieșiri Mealy, au valori diferite pe diferite linii asociate unei stări prezente).

Tabelul de tranziții reprezintă o formă utilă de centralizare a datelor în cazul implementării automatului sub o formă ce necesită minimizarea funcțiilor de transfer ale stării sau ale ieșirilor.

### Exemplu

Deduceți tabelul de tranziții al unui circuit secvențial care numără modulo 3 în sens crescător (0, 1, 2, 0, 1, ...), dacă intrarea are valoarea 0 și în sens descrescător, dacă intrarea are valoarea 1. Ieșirea este activă 1 dacă automatul se află în starea 2.

### Soluție

Din enunț se deduce structura și caracteristicile automatului necesar a fi proiectat:

- Automat Moore.

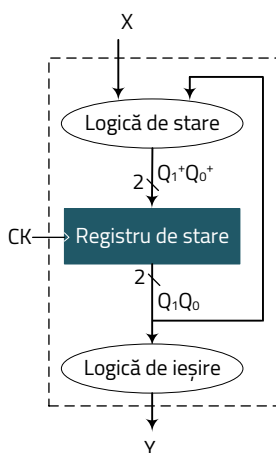


- 3 stări (denumite  $S_0, S_1, S_2$ ), posibil de codat cu 2 biți (denumiți  $Q_1$  și  $Q_0$ ). Codurile sunt alese pentru a corespunde reprezentării binare a indexului stărilor:  $S_0, S_1, S_2$ .

| Stare | Cod<br>$Q_1 Q_0$ |
|-------|------------------|
| $S_0$ | 00               |
| $S_1$ | 01               |
| $S_2$ | 10               |

- o intrare (denumită  $X$ ).
- o ieșire (denumită  $Y$ ), activă în 1 dacă automatul este în starea  $S_2$ .

Simbolul bloc al automatului este prezentat în figura 3.25.



**Figura 3.25** Simbol automat modulo 3.

Tabelul de tranziții este prezentat în tabelul 3.13. Se observă că în cazul în care  $X = 0$ , automatul parcurge stările ciclic în ordinea  $S_0, S_1, S_2, S_0, \dots$  iar pentru  $X = 1$ , automatul parcurge stările în sens invers:  $S_0, S_2, S_1, S_0, \dots$ . Ieșirea este  $Y = 0$  doar dacă automatul este în starea  $S_2$ .

Starea codată 11 nu pare a fi necesară. Însă, trebuie decis comportamentul automatului în ipoteza că automatul ajunge în această stare. O decizie bună este de a se trece necondiționat în starea  $S_0$ .

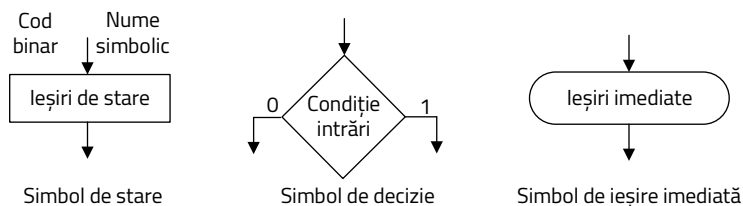
| Intrare<br>$X$ | Stare<br>prezentă<br>$Q_1 Q_0$ |    | Stare<br>viitoare<br>$Q_1^+ Q_0^+$ |    | Ieșire<br>$Y$ |
|----------------|--------------------------------|----|------------------------------------|----|---------------|
| 0              | $S_0$                          | 00 | $S_1$                              | 01 | 0             |
| 0              | $S_1$                          | 01 | $S_2$                              | 10 | 0             |
| 0              | $S_2$                          | 10 | $S_0$                              | 00 | 1             |
| 1              | $S_0$                          | 00 | $S_2$                              | 10 | 0             |
| 1              | $S_2$                          | 10 | $S_1$                              | 01 | 1             |
| 1              | $S_1$                          | 01 | $S_0$                              | 00 | 0             |
| –              | $S_3$                          | 11 | $S_0$                              | 00 | 0             |

Tabelul 3.13

Tabelul de tranziții pentru automatul care numără modulo 3.

### 3.3.2 Organigrame

Organigrama este o formă de reprezentare grafică a comportamentului unui automat, într-o manieră algoritmică. Organigrama conține 3 tipuri de simboluri, prezentate în figura 3.26.



**Figura 3.26** Simboluri utilizate în organigrame.

- **Simbolul de stare** este un dreptunghi, care conține în interior numele ieșirilor active în acea stare, iar pe exterior numele simbolic al stării și codul binar al acesteia. Simbolul de stare are o singură intrare, care poate veni de la oricare din cele 3 simboluri și o singură ieșire, care poate conduce fie la un simbol de stare (tranziție necondiționată, fie la un simbol de decizie).
- **Simbolul de decizie** este un romb, care conține în interior o condiție a intrărilor, iar în exterior valorile cu care se poate evalua condiția, 0 și 1. Simbolul de decizie are o singură intrare, care poate veni de la oricare

din cele 3 simboluri și două ieșiri, care pot conduce la oricare dintre cele 3 simboluri.

- **Simbolul de ieșire imediată** este un oval, care conține în interior numele ieșirilor activate în cazul îndeplinirii condiției din simbolul de decizie precedent. Simbolul de ieșire imediată are o singură intrare, care poate veni doar de la o ieșire a unui simbol de decizie și o ieșire, care poate conduce la oricare din cele 3 simboluri. Simbolul de ieșire imediată apare doar în automate Mealy, la care ieșirea depinde direct și de intrări (nu numai de stare).

### Exemplu

Deduceți organigrama unui circuit secvențial care numără modulo 3 în sens crescător (0, 1, 2, 0, 1, ...), dacă intrarea are valoarea 0 și în sens descrescător, dacă intrarea are valoarea 1. Ieșirea este activă 1 dacă automatul se află în starea 2.

### Soluție

Simbolul bloc al automatului este prezentat în figura 3.25 iar tabelul de tranziții este prezentat în tabelul 3.13. Organigrama este o reprezentare grafică în care apar 3 simboluri de stare (dreptunghiuri) asociate celor 3 stări prezente în enunț ( $S_0, S_1, S_2$ ) plus starea nementionată ( $S_3$  cu codul neutilizat 11). În fiecare stare este testată intrarea  $X$ , deci, după fiecare simbol de stare apare un simbol de decizie. Ieșirea  $Y$  apare menționată exclusiv în simbolul de stare  $S_2$ .

Tabelul de tranziții 3.13 este asociat cu organigrama prezentată în figura 3.27.

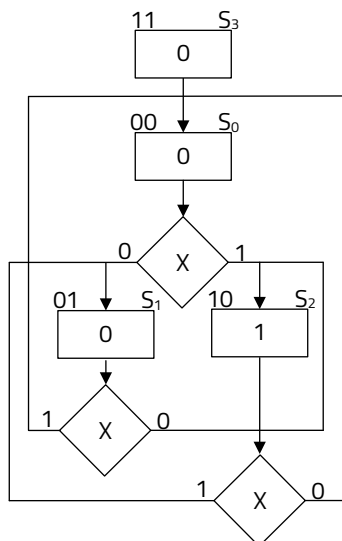
### 3.3.3 Grafuri

Graful de tranziții este o formă de reprezentare grafică a comportamentului unui automat, utilizând noduri și arce. Nodurile sunt asociate stărilor, iar arcele modelează tranzițiile între stări. În noduri se marchează codul stării. Pe arce se marchează condițiile intrărilor care determină tranziția între starea prezentă și starea viitoare.

Arcul pornește din nodul asociat stării prezente și ajunge în nodul asociat stării viitoare.

Figura 3.28 prezintă modul de marcarea informației pe noduri și arce în cazul celor două tipuri de automate, Mealy și Moore.

- **Automatul Moore** are în noduri marcate și ieșirile activate în starea asociată nodului. Pe arce sunt înscrise condițiile intrărilor.



**Figura 3.27** Organigrama asociată automatului care numără modulo 3.

- **Automatul Mealy** are pe arce marcate și ieșirile activate dacă este îndeplinită condiția asociată arcului.

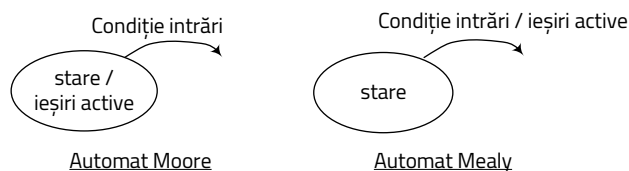
### Exemplu

Deduceți graful de tranziții al unui circuit secvențial care numără modulo 3 în sens crescător (0, 1, 2, 0, 1, ...), dacă intrarea are valoarea 0 și în sens descrescător, dacă intrarea are valoarea 1. Ieșirea este activă 1 dacă automatul se află în starea 2.

### Soluție

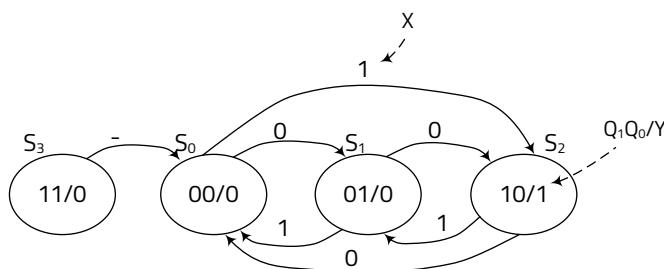
Simbolul bloc al automatului este prezentat în figura 3.25, tabelul de tranziții este prezentat în tabelul 3.13 iar organigrama în figura 3.27. Graful de tranziții al automatului de tip Moore, prezintă 3 noduri (cercuri) asociate celor 3 stări prezente în enunț ( $S_0, S_1, S_2$ ) plus un nod pentru starea nementionată ( $S_3$  cu codul neutilizat 11). Din fiecare nod pleacă două arce pe care este menționată valoarea intrării  $X$ . Ieșirea  $Y$  apare menționată în simbolurile de stare.

Graful de tranziții asociat automatului care numără modulo 3 este prezentat în figura 3.29. Se observă tranziția necondiționată din starea  $S_3$ ,



**Figura 3.28** Noduri și arce utilizate pentru grafurile de tranziții.

simbolizată ca un singur arc, cu intrarea indiferentă  $-$ . Conform regulii de la grafurile automatelor Moore, în nodurile asociate stărilor sunt precizate codurile stărilor  $Q_1Q_0$  și ieșirea  $Y$ , iar pe arce sunt precizate valorile intrării  $X$ .



**Figura 3.29** Graful de tranziții asociat automatului care numără modulo 3.

### 3.3.4 Analiza circuitelor logice secvențiale

Analiza circuitelor logice secvențiale constă în *determinarea comportamentului automatului pe baza structurii acestuia* (porți și bistabile). Comportamentul circuitului logic secvențial poate fi descris sub formă de organigramă sau graf de tranziții.

Pentru analiza unui circuit secvențial, se recomandă parcurgerea următoarelor etape:

- Circuitul este un *automat* dacă în structura sa apar bistabile și bucle

între ieșirile și intrările acestora.

- Se determină caracteristicile automatului:
  - Numărul de intrări se determină prin observarea intrărilor primare (din exteriorul circuitului).
  - Numărul de ieșiri se determină prin observarea ieșirilor. Ieșirile pot fi direct din registru (un bit de stare este transmis la ieșire) sau prin intermediul unui circuit combinațional (având intrările provenite de la ieșirile bistabilelor și opțional de la intrările primare).
  - Determinarea tipului automatului se face prin observarea modului în care se generează ieșirile.  
Dacă există o cale combinațională între o intrare și o ieșire, atunci automatul este de tip Mealy.  
Dacă toate ieșirile depind exclusiv de starea prezentă (ieșirile bistabilelor), atunci automatul este de tip Moore.
  - Numărul de biți care codifică stările,  $N_s$ , este egal cu numărul de bista-bile prezente în schemă.
  - Numărul maxim de stări este egal cu  $2^{N_s}$ , s-a notat cu  $N_s$  numărul de biți utilizați pentru codificarea stării.
- Cu informațiile deduse, se poate defini *schema bloc a automatului* ca fiind una similară cu structurile de automate prezentate în figurile 3.23 și 3.24.
- Se deduc din circuit *ecuațiile stării viitoare și ale ieșirilor*, în funcție de starea prezentă și intrări.
- Se completează *tabelul de tranziții*, pe baza caracteristicilor acestuia (număr de intrări, număr de ieșiri, număr de biți pentru codificarea stării) și a ecuațiilor stării viitoare și ale ieșirilor în funcție de starea prezentă și intrări. În partea stângă se completează coloanele asociate intrărilor și ale stării prezente. În partea dreaptă se completează coloanele asociate stării viitoare și ale ieșirilor. Numărul de rânduri din tabel este egal cu  $2^{N_s} \times 2^{N_i}$ , unde s-a notat cu  $N_s$  numărul de biți utilizați pentru codificarea stării și cu  $N_i$  numărul de intrări.

De exemplu, în cazul unui automat cu 4 stări (2 biți de cod) și 2 intrări tabelul va avea  $2^2 \times 2^2 = 16$  rânduri. Coloanele asociate stării prezente se completează cu codurile fiecărei stări posibile pe câte  $2^{N_s}$  rânduri (pe fiecare rând câte o combinație de intrare).

Coloanele asociate stării viitoare și cele asociate ieșirilor se completează pe baza ecuațiilor logice ale acestora. Pentru fiecare rând, se particula-rizează ecuațiile cu valorilor curente ale intrărilor și stării prezente de pe acel rând.

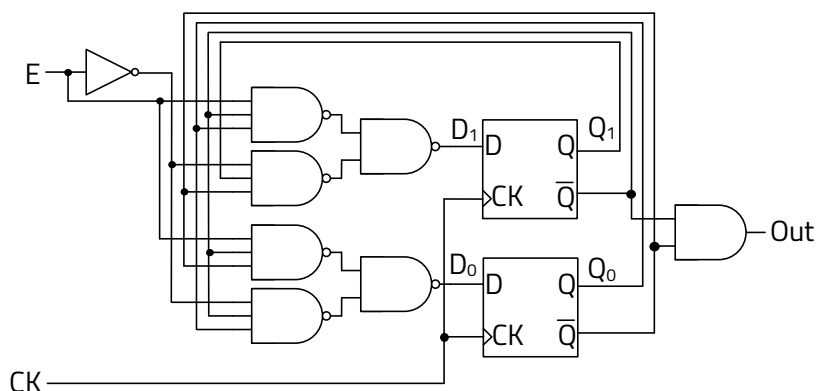
- Liniile din tabel care au aceeași stare prezentă și aceeași stare viitoare pot fi contopite într-o singură linie. Condiția acestei linii va conține valoarea indiferentă pentru intrarea diferită pentru cele două linii originale.
- Cu datele centralizate în tabelul de tranziții se construiește *graful de tranziții* ce descrie comportamentul automatului. Nodurile grafului semnifică stările automatului. Lângă noduri se marchează numele și codul stărilor. La automatele Moore, în noduri apar și numele ieșirilor active în respectivele stări.

Fiecare rând din tabel descrie un arc. Arcul pleacă de pe nodul asociat stării prezente și ajunge pe nodul stării viitoare. Pe arc este înscrisă condiția de comutare (valorile intrărilor care determină tranziția). La automatele Mealy, pe arce apar și numele ieșirilor active (fiind determinate atât de starea prezentă cât și de condiția de comutare dintr-o stare în alta).

- *Organigrama automatului* se poate deduce fie din tabelul de tranziții, fie din graful de tranziții. Fiecare nod al grafului este asociat cu un simbol de stare din organigramă. Arcele multiple care pleacă de pe un nod generează un simbol de decizie în organigramă.

#### Exemplu

Analizați circuitul logic secvențial prezentat în figura 3.30.



**Figura 3.30** Circuitul de analizat.

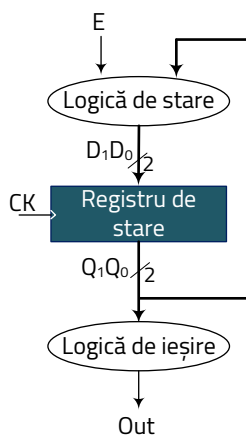
#### Soluție

Circuitul conține două bistabile D și porți logice. Automatul are o intrare

$E$  și o ieșire  $Out$ . Ieșirea depinde exclusiv de starea prezentă  $Out(Q_1, Q_0)$ . Deci, este un automat de tip Moore.

Deoarece există două bistabile în schemă, înseamnă că automatul are stările codificate cu 2 biți. Deci, automatul are maximum  $2^2 = 4$  stări. Starea prezentă este determinată de valorile logice ale ieșirilor bistabilelor. Biții de cod ai stării prezente sunt  $Q_1$  și  $Q_0$ . Intrările bistabilelor reprezintă biții stării viitoare ( $D_1$  și  $D_0$ ).

Schema bloc a automatului este prezentată în figura 3.31.



**Figura 3.31** Schema bloc a automatului analizat.

Ecuatiile stării viitoare se deduc din circuit, în funcție de starea prezentă și intrări:

$$D_1(E, Q_1, Q_0) = E \cdot \overline{Q_1} \cdot Q_0 + \overline{E} \cdot Q_1 \cdot \overline{Q_0}$$

$$D_0(E, Q_1, Q_0) = E \cdot \overline{Q_1} \cdot \overline{Q_0} + \overline{E} \cdot Q_1 \cdot Q_0$$

Ecuatia ieșirii se deduce din circuit, în funcție de starea prezentă:

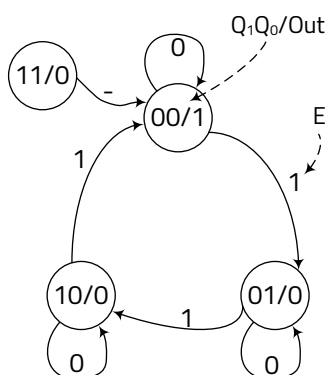
$$Out(Q_1, Q_0) = \overline{Q_1} \cdot \overline{Q_0}$$

Cu ajutorul ecuațiilor deduse, se completează tabelul de tranziții:



| Intrare<br>$E$ | Stare prezentă<br>$Q_1 \quad Q_0$ |   | Stare viitoare<br>$D_1 \quad D_0$ |   | Ieșire<br>$Out$ |
|----------------|-----------------------------------|---|-----------------------------------|---|-----------------|
| 0              | 0                                 | 0 | 0                                 | 0 | 1               |
| 1              | 0                                 | 0 | 0                                 | 1 | 1               |
| 0              | 0                                 | 1 | 0                                 | 1 | 0               |
| 1              | 0                                 | 1 | 1                                 | 0 | 0               |
| 0              | 1                                 | 0 | 1                                 | 0 | 0               |
| 1              | 1                                 | 0 | 0                                 | 0 | 0               |
| 0              | 1                                 | 1 | 0                                 | 0 | 0               |
| 1              | 1                                 | 1 | 0                                 | 0 | 0               |

Cu datele centralizate în tabelul de tranziții se construiește graful de tranziții, ce descrie comportamentul automatului. Graful de tranziții al automatului analizat este prezentat în figura 3.32.



**Figura 3.32** Graful de tranziții al automatului analizat.

Studiind graful de tranziții, se poate deduce rolul automatului. Automatul ciclează între 3 stări dacă  $E = 1$  („Enable”, validare, engl.). Dacă  $E = 0$  automatul își păstrează starea. Ieșirea semnalează apariția stării 00. În cazul apariției stării absente din ciclu (11) automatul revine necondiționat, în următorul ciclu de ceas, în starea 00 și se reia ciclul de 3 stări.

### 3.3.5 Sinteza automatelor

Metodologia de sinteză a automatelor presupune parcurgerea următoarelor etape:

1. Din descrierea existentă, se deduce funcționalitatea automatului și se descrie în una din formele de reprezentare: organigramă sau graf de tranziții.
2. Dacă este posibil, se reduc stările redundante. Stările redundante sunt stările din care automatul evoluează identic și în care se activează aceleași ieșiri.
3. Se codifică stările prin asocierea acestora cu coduri binare unice.
4. Se alege tipul de bistabile folosite pentru implementare (D, T).
5. Se completează tabelul de tranziții cu toate stările și toate combinațiile intrărilor.
6. Se completează tabelul de tranziții cu stările următoare și ieșirile. Dacă implementarea se face cu bistabile T, se completează tabelul de tranziții cu coloanele asociate stimulilor bistabilelor T.
7. Se minimizează funcțiile stării următoare (sau ale stimulilor pentru determinarea stărilor următoare) și ale ieșirilor.
8. Se deduc ecuațiile stării următoare (sau ale stimulilor pentru determinarea stărilor următoare) și ale ieșirilor.
9. Se desenează schema de implementare cu bistabilele alese și logica determinată de ecuațiile deduse anterior.

Tabelele următoare prezintă modul în care se determină ecuațiile stimulilor pentru bistabile, pe baza stării curente și a celei viitoare:

| $D$ | $Q^+$ |
|-----|-------|
| 0   | 0     |
| 1   | 1     |

| $Q$ | $Q^+$ | $D = Q^+$ |
|-----|-------|-----------|
| 0   | 0     | 0         |
| 0   | 1     | 1         |
| 1   | 0     | 0         |
| 1   | 1     | 1         |

| $T$ | $Q^+$          |
|-----|----------------|
| 0   | $\overline{Q}$ |
| 1   | $Q$            |

| $Q$ | $Q^+$ | $T = Q \oplus Q^+$ |
|-----|-------|--------------------|
| 0   | 0     | 0                  |
| 0   | 1     | 1                  |
| 1   | 0     | 1                  |
| 1   | 1     | 0                  |

### Exemplu

Implementați un circuit secvențial care numără modulo 3 în sens crescător (0, 1, 2, 0, 1, ...), dacă intrarea are valoarea 0 și în sens descrescător, dacă

intrarea are valoarea 1. Ieșirea este activă 1 dacă automatul se află în starea 2.

### Soluție

Simbolul bloc al automatului este prezentat în figura 3.25, tabelul de tranziții este prezentat în tabelul 3.13, organigrama în figura 3.27, iar graful de tranziții al automatului de tip Moore în figura 3.29.

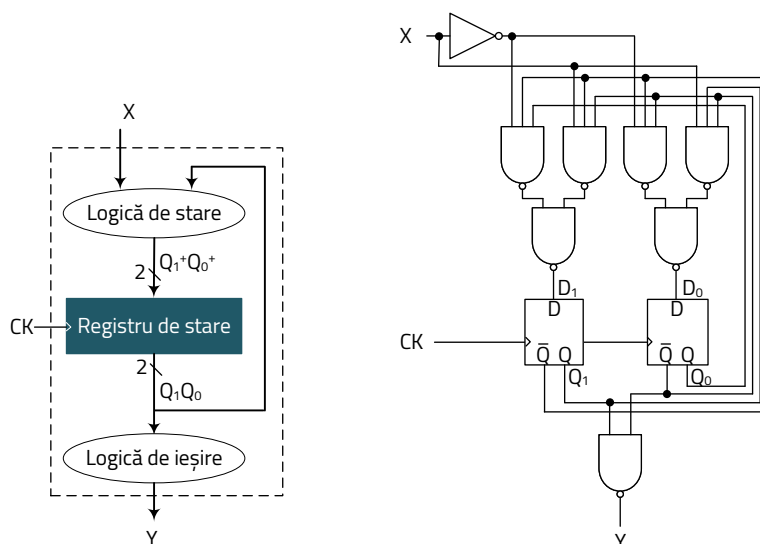
Pentru implementare se deduc și se minimizează funcțiile precizate în tabelul de tranziții 3.13. Starea viitoare  $Q_1^+Q_0^+$  se va conecta pe intrările celor două bistabile de stare  $D_1D_0$ . Se pot utiliza diagrame V-K pentru reprezentarea și minimizarea funcțiilor.

$$D_1(X, Q_1, Q_0) = Q_1^+ = \overline{X} \cdot \overline{Q_1} \cdot Q_0 + X \cdot \overline{Q_1} \cdot \overline{Q_0}$$

$$D_0(X, Q_1, Q_0) = Q_0^+ = \overline{X} \cdot Q_1 \cdot \overline{Q_0} + X \cdot Q_1 \cdot Q_0$$

$$Y(Q_1, Q_0) = Q_1 \cdot Q_0$$

Circuitul este similar celui prezentat în figura 3.25. Registrul de stare este format din două bistabile având ieșirile  $Q_1Q_0$  și intrările  $D_1D_0$ . Logica de stare implementează funcțiile  $D_1D_0(X, Q_1, Q_0)$  iar logica de ieșire implementează funcția  $Y(Q_1, Q_0)$ . Circuitul proiectat este prezentat în figura 3.33.



**Figura 3.33** Circuitul proiectat pentru automatul care numără modulo 3.

### 3.3.6 Modelarea automatelor în Verilog

Modelarea automatelor în Verilog se bazează pe structura acestora (figura 3.23):

- *Circuit logic combinațional pentru determinarea stării viitoare*, pe baza intrărilor și a stării prezente (CLC de stare).
- *Element de memorare a stării curente*, între evenimentele ce determină re-evaluarea stării, realizat cu multiple instanțe de bistabile (registru de stare).
- *Circuit logic combinațional pentru determinarea valorilor ieșirilor*, pe baza intrărilor și a stării prezente (CLC de ieșire).

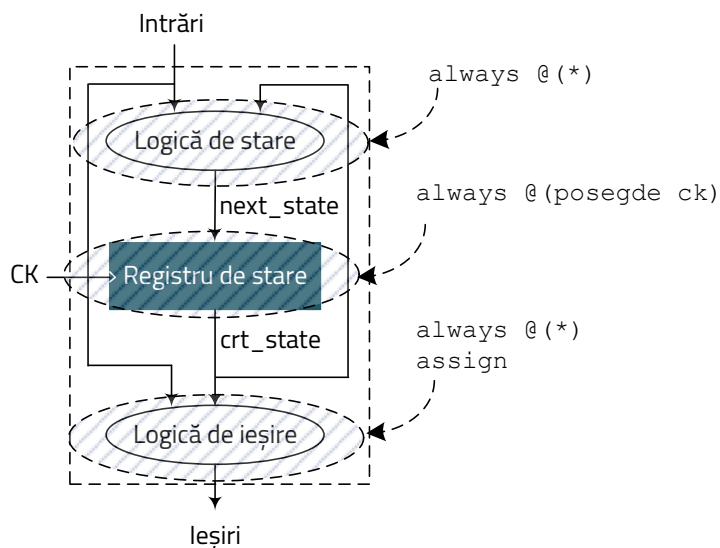
O variantă de modelare constă în modelarea fiecărei structuri cu o specificație Verilog distinctă, după cum este prezentat în figura 3.34:

- *CLC de stare* modelat cu o specificație `always` care include o specificație `case` cu selecție pe baza stării prezente, pentru a determina starea viitoare. Intrările se evaluează conform comportamentului automatului, în funcție de starea curentă.
- *Element de memorare a stării curente* modelat cu o specificație `always` secvențială, senzitivă la frontul activ al semnalului de ceas, care descrie transformarea stării viitoare în stare curentă.
- *CLC de ieșire* modelat cu o specificație `always` include o specificație `case` cu selecție pe baza stării prezente, pentru a determina valorile ieșirilor. În funcție de timpul automatului, Mealy sau Moore, în expresia atribuită ieșirilor pot apare intrările. În cazul unor expresii mai simple ale ieșirilor, se poate utiliza specificația de atribuire continuă `assign`, care prezintă expresia logică a ieșirii, pe baza stării curente și (eventual) a intrărilor.

Stările automatului pot fi codate ca parametri locali ai modulului. Codurile asociate stărilor trebuie să fie unice fiecărei stări.

```
localparam s0 = 2'b00;  
localparam s1 = 2'b01;  
localparam s2 = 2'b10;  
localparam s3 = 2'b11;
```

Este nerecomandată expunerea codurilor stărilor ca parametrii, deoarece modificarea acestora nu ar trebui făcută la instanțierea modulului. Este incorectă asocierea mai multor stări cu același cod. Sintetizatoarele actuale pot face modificări asupra codurilor stărilor pentru a optimiza întregul automat.



**Figura 3.34** Modelarea automatelor cu 3 specificații: CLC stare, registru de stare, CLC ieșire.

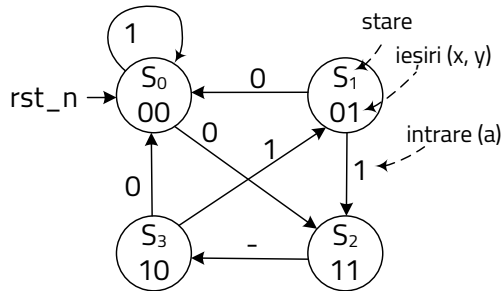
### Exemplu

Realizați un model Verilog pentru automatul al cărui comportament este reprezentat prin graful de tranziții din figura 3.35. Automatul Moore are 4 stări ( $s_0, s_1, s_2, s_3$ ), o intrare ( $a$ ) și două ieșiri ( $x, y$ ). Automatul are un semnal de reset asincron activ în 0 ( $rst\_n$ ).

### Soluție

Cele 4 stări pot fi codate cu un număr minim de 2 biți. Pentru simplitate, s-au ales codurile binare asociate indexului stărilor. Codul Verilog care modelează automatul este prezentat în continuare.

```
module fsm_type1 (
input  clk    , // semnal de ceas
input  rst_n  , // semnal de reset asincron activ 0
input  a      , // intrare
output x      , // ieșiri
output y
);
```



**Figura 3.35** Graf de tranziții pentru model Verilog.

```

// declaratii semnale interioare
reg [1:0] crt_state ; // stare curenta
reg [1:0] next_state ; // stare viitoare

// coduri stari (unice)
localparam s0 = 2'b00;
localparam s1 = 2'b01;
localparam s2 = 2'b10;
localparam s3 = 2'b11;

// model CLC de stare:
// next_state(crt_state, a)
always @(*)
case (crt_state)
s0 : if (~a) next_state <= s2; // a=0 -> s2
    // a=1 -> pastreaza s0
s1 : if (a) next_state <= s2; else // a=1 -> s2
    next_state <= s0; // a=0 -> s0
s2 : next_state <= s3; // neconditionat
default: next_state <= a ? s1 : s0; // s3
endcase

// model registru de stare
// crt_state <- next_state
always @(posedge clk or negedge rst_n)
if (~rst_n) crt_state <= s0; else // stare initiala a FSM
    crt_state <= next_state;

// model CLC de iesire
// x(crt_state)

```

```

assign x = (crt_state == s2) | (crt_state == s3);
// y(crt_state)
assign y = (crt_state == s1) | (crt_state == s2);

endmodule // fsm_type1

```

O alternativă de modelare constă în modelarea cu o singură specificație a modului de determinare și memorare a stării curente, pe baza stării curente și a intrărilor. Desigur, starea curentă este modificată exclusiv în momentele determinate de frontul activ al semnalului de ceas. Figura 3.36 prezintă asocierea structurilor automatului cu specificațiile Verilog.

- *CLC de stare și element de memorare a stării* curente modelat cu o specificație `always` secvențială, senzitivă la frontul activ al semnalului de ceas, care include o specificație `case` cu selecție pe baza stării prezente, pentru a determina starea viitoare. Intrările se evaluează conform comportamentului automatului, în funcție de starea curentă.
- *CLC de ieșire* modelat cu o specificație `always` include o specificație `case` cu selecție pe baza stării prezente, pentru a determina valorile ieșirilor. În funcție de tipul automatului, Mealy sau Moore, în expresia atribuită ieșirilor pot apare intrările. În cazul unor expresii mai simple ale ieșirilor, se poate utiliza specificația de atribuire continuă `assign`, care prezintă expresia logică a ieșirii, pe baza stării curente și (eventual) a intrărilor.

Modelul Verilog pare a fi mai compact, dar necesită o înțelegere mai aprofundată a sintaxei limbajului de descriere hardware și a modalității de modelare a comportamentului.

```

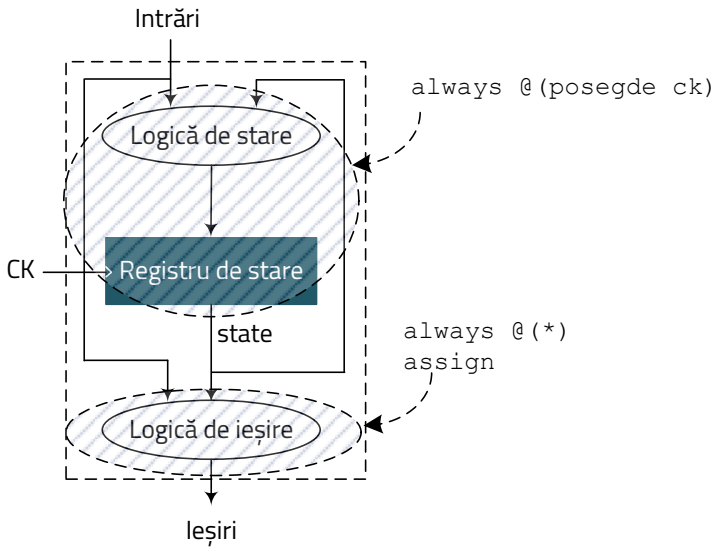
module fsm_type2 (
input      clk      , // semnal de ceas
input      rst_n    , // semnal de reset asincron activ 0
input      a        , // intrare
output reg  x        , // iesiri
output reg  y
);

// declaratii semnale interioare
reg [1:0] crt_state; // stare curenta

// coduri stari (unice)
localparam s0 = 2'b00;
localparam s1 = 2'b01;
localparam s2 = 2'b10;
localparam s3 = 2'b11;

// model CLC de stare + registru stare

```



**Figura 3.36** Modelarea automatelor cu 2 specificații: CLC stare+registru de stare, CLC ieșire.

```
// crt_state <- f(crt_state, a)
always @(posedge clk or negedge rst_n)
if (~rst_n) crt_state <= s0; else // stare initiala a FSM
case (crt_state)
s0 : if (~a) crt_state <= s2; // a=0 -> s2
// a=1 -> pastreaza s0
s1 : if (a) crt_state <= s2; else // a=1 -> s2
crt_state <= s0; // a=0 -> s0
s2 : crt_state <= s3; // neconditionat
default: crt_state <= a ? s1 : s0; // s3
endcase

// model CLC de iesire
// x(crt_state), y(crt_state)
always @(*)
case (crt_state)
s0 : {x, y} <= 2'b00;
s1 : {x, y} <= 2'b01;
s2 : {x, y} <= 2'b11;
default: {x, y} <= 2'b10; // s3
```

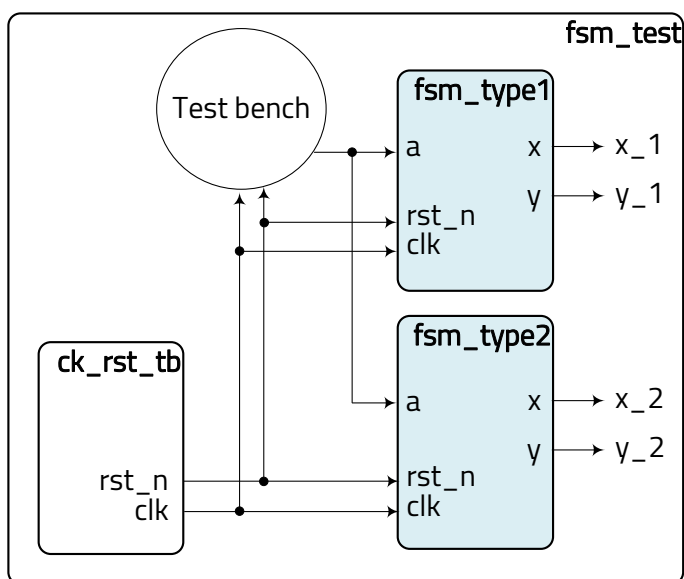


```
endcase
```

```
endmodule // fsm_type2
```

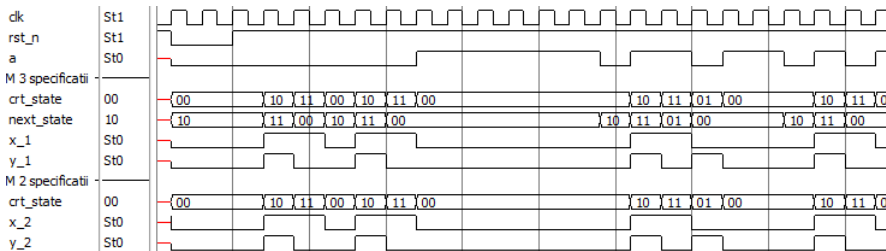
Modelul de simulare (figura 3.37) conține:

- generarea semnalelor de ceas și de reset potrivite,
- instanțierea celor două modele de automate,
- generarea de stimuli asociați intrării automatului.



**Figura 3.37** Model de simulare pentru automate.

Prin simulare, se poate observa că ambele tipuri de modelare prezintă același comportament (figura 3.38).



**Figura 3.38** Forme de undă din simularea celor două modalități de modelare a automatelor.



### Despre Măgăoaia cu Stări și Toane:

#### Cronica unei Minți Finite

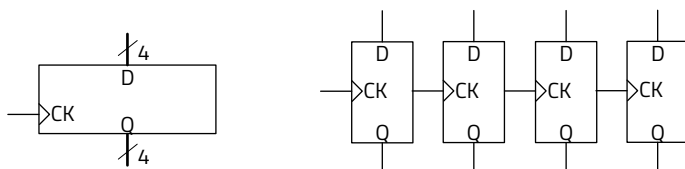
Și iată, bunii mei ucenici ai electronice, am ajuns la o făptură ciudată și înțeleaptă, cunoscută în limbajul învățaților drept FSM, adică Finite State Machine, sau, pe românește mai cu haz, „Măgăoaia cu Stări Numărate”.

Această dihanie nu face nici mai mult, nici mai puțin decât să-și tot schimbe starea după cum o gădilă intrările, păstrând în inima sa de flip-flop amintirea clipelor trecute. Nu-i chip s-o păcălești: stă, veghează, se schimbă, dar tot cu număr finit de toane! Precum un funcționar împietrit în reguli, nu mișcă fără tabel și nu hotărăște nimic fără diagrama de tranziție – o adevărată aristocrație a ordinii binare!

*Text generat cu ChatGPT.*

## 3.4 Registre și numărătoare

Extensia paralel a unui bistabil poartă numele de **registru**. Simbolul bloc și structura unui registru de 4 biți implementat cu bistabile D sunt prezentate în figura 3.39.



**Figura 3.39** Registru D pe 4 biți: simbol bloc și structură cu bistabile D.

### 3.4.1 Registre cu accesare serie sau paralel

În funcție de modalitatea de încărcare și accesare a datelor, registrele se clasifică astfel:

- **Registru paralel-paralel.** Datele sunt încărcate în registru și accesate din exterior în paralel. Față de registrul „clasic”, registrul paralel-paralel poate dispune de o intrare suplimentară care validează încărcarea doar când semnalul de validare este activ (în acest mod, registrul poate „memora” datele în tot intervalul de timp dintre două pulsuri de încărcare, nu doar pe durata unei singure perioade de ceas).

Tabelul de funcționare este:

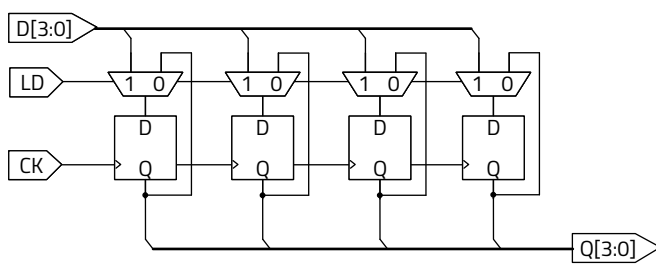
| $LD$ | $Q^+$ | Acțiune                                        |
|------|-------|------------------------------------------------|
| 1    | $D$   | Încarcă valoarea logică de pe intrarea de date |
| 0    | $Q$   | Păstrează starea curentă                       |

Structura internă (figura 3.40) prezintă o buclă între ieșirea și intrarea registrului intern peste un multiplexor prin care se selectează starea viitoare: se încarcă starea din exterior ( $LD = 1$ ) sau se menține starea curentă ( $LD = 0$ ).

Registrul este utilizat pentru stocarea datelor reprezentate pe mai mulți biți între două activări succesive ale semnalului de încărcare (încărcarea nu trebuie să se facă la fiecare front activ al ceasului, ca în cazul registrului clasic).

- **Registru paralel-serie (de serializare).** Datele sunt încărcate în registru în paralel dar sunt accesate din exterior în mod serial. Sunt necesare două intrări de control pentru determinarea încărcării datelor și a deplasării acestora în registrul intern (pentru deplasarea serială a datelor pe poziția accesibilă din exterior).

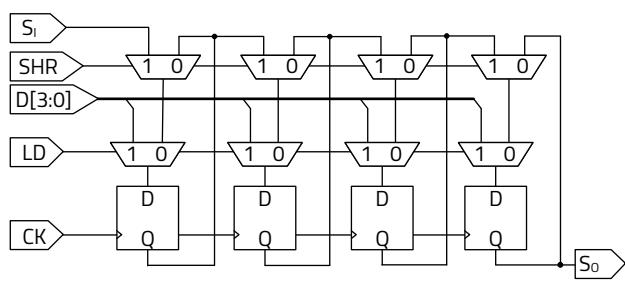
Tabelul de funcționare este:



**Figura 3.40** Registru paralel-paralel cu validare a încărcării.

| $LD$ | $SHR$ | $Q^+$              | Acțiune                                                  |
|------|-------|--------------------|----------------------------------------------------------|
| 1    | X     | $D$                | Încarcă în paralel valoarea de pe intrarea de date       |
| 0    | 1     | $\{S_L, Q \gg 1\}$ | Deplasează spre dreapta conținutul registrului cu un bit |
| 0    | 0     | $Q$                | Păstrează starea curentă                                 |

Structura internă (figura 3.41) prezintă o buclă între ieșirea și intrarea registrului peste un multiplexor prin care se selectează starea viitoare: se încarcă starea din exterior ( $LD = 1$ ), se deplasează datele din interior ( $LD = 0, SHR = 1$ ) sau se menține starea curentă ( $LD = 0, SHR = 0$ ). Ieșirea serială de 1 bit provine de la cel mai puțin semnificativ bit al registrului.



**Figura 3.41** Registru paralel-serie (de serializare).

Aplicația clasică a registrului paralel-serie constă în serializarea date-

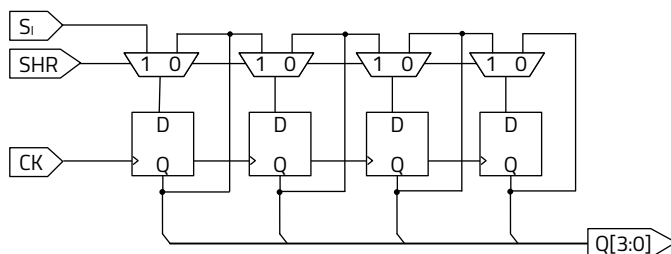
lor reprezentate pe mai mulți biți în vederea transmiterii acestora pe un singur fir. În acest caz, un impuls de încărcare  $LD = 1$  este urmat de un număr de impulsuri de deplasare  $SHR = 1$  egal cu numărul de biți ai registrului intern.

- **Registru serie-paralel (de deserializare).** Datele sunt încărcate în registru în mod serial, dar sunt accesate din exterior în mod paralel. Este necesară o intrare de control pentru determinarea deplasării datelor în registrul intern (pentru avansarea serială a datelor acceptate în vederea completării cuvântului de date accesibile din exterior în paralel). Registrul are o intrare de date de un singur bit. În cazul activării încărcării datelor, în poziția cea mai semnificativă se încarcă datele de intrare  $S_I$  și deplasează datele interne cu un bit spre dreapta. Cel mai puțin semnificativ bit se pierde.

Tabelul de funcționare este:

| $SHR$ | $Q^+$              | Acțiune                                                                 |
|-------|--------------------|-------------------------------------------------------------------------|
| 1     | $\{S_I, Q \gg 1\}$ | Încarcă datele de intrare (1 bit) și deplasează conținutul spre dreapta |
| 0     | $Q$                | Păstrează starea curentă                                                |

Structura internă (figura 3.42) prezintă o buclă între ieșirea și intrarea registrului peste un multiplexor prin care se selectează starea viitoare: se încarcă bitul din exterior și se deplasează datele spre dreapta ( $SHR = 1$ ), sau se menține starea curentă ( $SHR = 0$ ). Ieșirea paralelă este accesibilă din exterior în orice moment.



**Figura 3.42** Registru serie-paralel (de deserializare).

Aplicația clasică a registrului serie-paralel constă în deserializarea datelor reprezentate pe mai mulți biți. În acest caz, modulul de control

extern trebuie să asigure preluarea datelor în paralel după un număr de impulsuri de încărcare  $SHR = 1$  egal cu numărul de biți ai registrului.

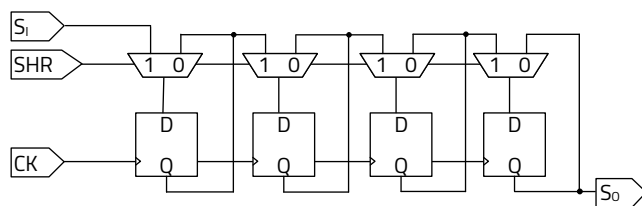
- **Registru serie-serie (de deplasare).** Deși în interior există un registru pe mai mulți biți, datele sunt încărcate și sunt accesate serial, câte un bit pe tact. Registrul realizează o deplasare internă a datelor. Este necesară o intrare de control pentru determinarea încărcării datelor și a deplasării acestora în registrul intern (pentru deplasarea serială a datelor acceptate). Registrul are o intrare de date de un singur bit. În cazul activării încărcării datelor, în poziția cea mai semnificativă se încarcă datele de intrare  $S_I$  și deplasează datele interne cu un bit. Cel mai puțin semnificativ bit se pierde. Ieșirea serială de un bit provine de la cel mai puțin semnificativ bit al registrului.

Registrul se comportă ca registrul serie-paralel, cu deosebirea că la registrul serie-serie doar un bit (cel mai puțin semnificativ) este accesibil din exterior ( $Q_{[0]}$ ).

Tabelul de funcționare este:

| $SHR$ | $Q^+$             | Acțiune                                                    |
|-------|-------------------|------------------------------------------------------------|
| 1     | $\{S_I, Q >> 1\}$ | Încarcă datele de intrare (1 bit) și deplasează conținutul |
| 0     | $Q$               | Păstrează starea curentă                                   |

Structura internă (figura 3.43) prezintă o buclă între ieșirea și intrarea registrului intern peste un multiplexor prin care se selectează starea viitoare: se încarcă bitul din exterior și se deplasează datele ( $SHR = 1$ ), sau se menține starea curentă ( $SHR = 0$ ). Ieșirea serială (cel mai puțin semnificativ bit) este accesibilă din exterior în orice moment.

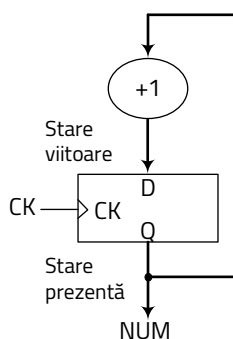


**Figura 3.43** Registru serie-serie (registru de deplasare).

Aplicația clasică a registrului serie-serie constă în stocarea datelor sub forma unei „cozi de așteptare” care avansează doar la comandă ( $SHR = 1$ ).

### 3.4.2 Numărătoare

Numărătoarele sunt circuite logice secvențiale (automate) care își determină starea următoare exclusiv pe baza stării prezente, printr-o funcție de numărare. Structura „clasică” a unui numărător sincron (figura 3.44) prezintă un sumator conectat în buclă, între ieșirea și intrarea unui registru de stare. Numărătoarele pot avea diverse intrări care controlează funcția de numărare (direcția de numărare, condiția de numărare, domeniul de numărare, codul de numărare, inițializarea).



**Figura 3.44** Schema bloc a numărătoarelor sincrone.

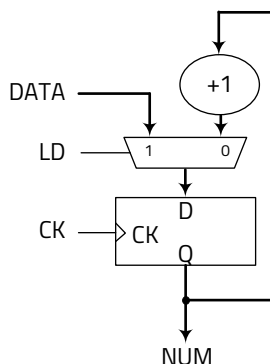
La numărătoarele presetabile (figura 3.45) există posibilitatea ca starea să fie inițializată din exterior, printr-o comandă explicită,  $LD = 1$ .

### 3.4.3 Linear-Feedback-Shift-Register

Un tip particular de registru îl constituie circuitele **LFSR** („*Linear Feedback Shift Register, Registru de deplasare cu reacție liniară*”, engl.).

Registrul de deplasare cu reacție liniară reprezintă o modalitate de obținere a unei secvențe de numărare lungi cu un număr mic de bistabile și care să funcționeze la frecvență mare (puține porți logice). În structura unui LSFR de  $N$  biți,  $N - 1$  din cele  $N$  bistabile primesc pe intrare semnalul de la ieșirea bistabilului de ordin superior. Un singur bistabil, cel mai semnificativ, primește la intrare o funcție logică XOR determinată pe baza unor anumiți biți.

Determinarea biților care intră în calculul valorii viitoare a bitului cel mai semnificativ se face pe baza unui polinom. Polinomul are gradul egal cu



**Figura 3.45** Schema bloc a unui numărător presetabil.

$LD = 1$ , se încarcă noua valoare din exterior,

$LD = 0$ , se trece în noua stare prin numărare.

numărul de biți ai LFSR și prezintă indecșii biților care determină valoarea viitoare a celui mai semnificativ bit.

De exemplu, dacă polinomul este  $X^4 + X^3 + 1$ , figura 3.46, valoarea viitoare a bitului cel mai semnificativ este determinată prin funcția XOR între bitul 0  $X^0 = 1$  și bitul 3  $X^3$ .

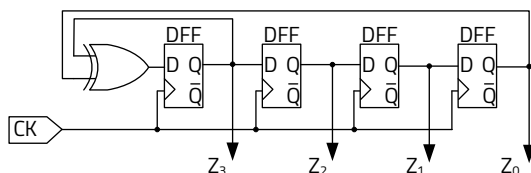
Se remarcă următoarele:

- Funcția XOR realizează funcția de adunare aritmetică a biților (fără transport).
- Pentru  $N$  bistabile, se obține o secvență de numărare de lungime maximă  $2^N - 1$ . Starea 000...0 este o stare de blocare.
- Nu orice polinom produce secvențe de numărare de lungime maximă. În funcție de polinomul ales, se pot obține mai multe secvențe de numărare de lungimi mai mici decât lungimea maximă.
- Un circuit LFSR poate fi considerat un **generator de numere pseudo-aleatorii**, dacă are o secvență de numărare foarte lungă (număr mare de biți) și se utilizează un număr mic de eșantioane (stări) succesive.

### Rețetă

Circuitul LFSR cu bistabile se proiectează pe baza polinomului astfel:





**Figura 3.46** Circuit LFSR pe baza polinomului  $X^4 + X^3 + 1$ .

- Registrul are un număr de bistabile egal cu puterea cea mai mare prezentă în polinom. Dacă cea mai mare putere ce apare în polinom este  $X^N$ , atunci cele  $N$  bistabilele vor avea indecșii:  $N - 1, \dots, 0$ .
- Bistabilele sunt conectate în serie, intrarea unui bistabil provenind de la ieșirea celui cu pondere superioară (intrarea bistabilului  $i$  este ieșirea bistabilului  $i + 1$ , unde  $i \in \{N - 2 \dots 0\}$ ).
- Intrarea celui mai semnificativ bistabil (index  $N - 1$ ) provine de la ieșirea unei porți SAU EXCLUSIV (XOR). Intrările în poarta XOR provin de la ieșirile bistabilelor ale căror index este prezent în puterile polinomului. De remarcat că în polinom apare întotdeauna  $1 = X^0$ .

Nu orice polinom generează o secvență de lungime maximă. Tabelul 3.14 prezintă polinoamele de lungime între 2 și 16 biți pe baza cărora se pot proiecta circuite LFSR cu secvențe maxime de stări:  $2^N - 1$  stări pentru un polinom de grad  $N$ .

### 3.4.4 Modelarea HDL a registrelor și a numărătoarelor

Această secțiune prezintă modele Verilog pentru diferite tipuri de registre și numărătoare. Modelele nu sunt standardizate și se adaptează cerințelor specifice proiectelor. Structura acestora este însă similară: o specificație `always` care modelează un circuit secvențial, fiind activată la fiecare front activ al semnalului de ceas `@(posedge clk)`. Dacă registrul are o intrare de reset asincronă, specificația `always` trebuie să se activeze și la semnalul de reset asincron (activ în 0 în acest model): `@(posedge clk or negedge rst_n)`.

Semnalele de control implementează diverse funcții ale registrului, precizând starea următoare, pe baza intrărilor și a stării curente a registru-

| Biți $N$ | Polinom                              | Perioadă<br>$2^N - 1$ |
|----------|--------------------------------------|-----------------------|
| 2        | $X^2 + x + 1$                        | 3                     |
| 3        | $X^3 + X^2 + 1$                      | 7                     |
| 4        | $X^4 + X^3 + 1$                      | 15                    |
| 5        | $X^5 + X^3 + 1$                      | 31                    |
| 6        | $X^6 + X^5 + 1$                      | 63                    |
| 7        | $X^7 + X^6 + 1$                      | 127                   |
| 8        | $X^8 + X^6 + X^5 + X^4 + 1$          | 255                   |
| 9        | $X^9 + X^5 + 1$                      | 511                   |
| 10       | $X^{10} + X^7 + 1$                   | 1.023                 |
| 11       | $X^{11} + X^9 + 1$                   | 2.047                 |
| 12       | $X^{12} + X^{11} + X^{10} + X^4 + 1$ | 4.095                 |
| 13       | $X^{13} + X^{12} + X^{11} + X^8 + 1$ | 8.191                 |
| 14       | $X^{14} + X^{13} + X^{12} + X^2 + 1$ | 16.383                |
| 15       | $X^{15} + X^{14} + 1$                | 32.767                |
| 16       | $X^{16} + X^{15} + X^{13} + X^4 + 1$ | 65.535                |

Tabelul 3.14

Polinoame LFSR cu lungimi între 2 și 16 biți.

lui. Pentru modelare se folosesc specificații secvențiale if și case pentru a stabili prioritatea în cazul unor comenzi active simultan.

Trebuie remarcat că modelele Verilog prezentate în această secțiune pot fi generate automat de către programe de inteligență artificială generative (ChatGPT) cu un minim de cerințe. Modelele generate au însă multiple neajunsuri, care pot fi remediate prin cerințe suplimentare bine formulate.

Câteva argumente pentru a nu vă baza integral pe ChatGPT pentru a genera modele Verilog:

- ChatGPT învață, însă nu întotdeauna de la profesori buni. Sunt nenumărate exemple în care ChatGPT răspunde greșit datorită faptului că a învățat din date incorecte, publicate de autori incompetenți sau pur și simplu nu a învățat corect.
- Timpul consumat cu repararea primei soluții furnizate de ChatGPT este uneori ineficient de mare. Este greu și să transmitem un „*stil de modelare*” impus de o firmă (nume de variabile, sintaxe recomandate, comentarii, etc.)
- Cel puțin pentru începători, scrierea de modele Verilog în mod „tradițional”, cu ajutorul inteligenței naturale, ajută la înțelegerea structurilor circuitelor modelate, ceea ce este scopul oricărui curs.

- Utilizarea ChatGPT nu vă pregătește pentru interviuri de angajare. Nu poți cere acces la Internet, pentru a modela în Verilog un numărător modulo 10. Este prea simplu! Și *Very-logic*.

În concluzie, utilizați ChatGPT cu chibzuință și încercați mai întâi să înțelegeți fundamentele electronicii digitale și ale modelării în Verilog, atât pentru a putea utiliza codul generat de ChatGPT, dar și pentru a depista greșelile ChatGPT. Vă puteți baza pe ChatGPT pentru a elimina erorile de sintaxă sau a revizui propunerea voastră de cod.

### Registru paralel-paralel, cu dimensiune parametrizabilă

Registru are o dimensiune parametrizabilă, putând fi suprascrisă în momentul instanțierii. Resetul este asincron, activ în 0. În practică, modelul este prea mic pentru a fi încapsulat într-un modul. Se poate utiliza exclusiv specificația `always` acolo unde este necesar un registru, în modelul sistemului.

```
module reg_param #(
parameter WIDTH = 8    // latime registru, biti
)(
input                clk    , // ceas, activ front crescator
input                rst_n  , // reset asincron activ 0
input                [WIDTH-1:0] d    , // intrare
output reg          [WIDTH-1:0] q    // iesire registru
);

always @(posedge clk or negedge rst_n)
if (~rst_n) q <= 'b0; else // resetare asincrona
q <= d;                // incarcare date, fiecare tact

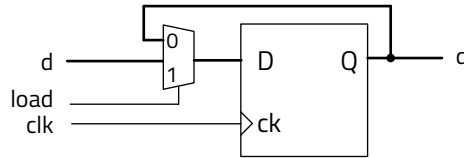
endmodule // reg_param
```

### Registru paralel-paralel, cu validare/încărcare

Pentru a implementa o intrare de validare, este suficientă condiționarea schimbării stării de activarea intrării de control:

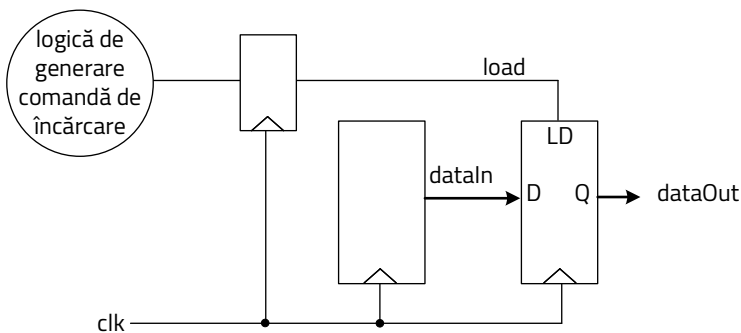
```
always @(posedge clk or negedge rst_n)
if (~rst_n) q <= 'b0; else // resetare asincrona
if (load) q <= d;          // incarcare paralela daca load=1
```

Se observă că a doua specificație `if` este incompletă, nu are clauza `else`. Prin această modelare, registru nu își modifică starea (și-o păstrează) în cazul nevalidării intrării de control. Evident, păstrarea stării este realizată fizic prin închiderea unei bucle, ca în figura 3.40 și figura 3.47.



**Figura 3.47** Registru paralel-paralel, cu validare/încărcare.

O aplicație frecventă constă în încărcarea registrului doar în anumite condiții și păstrarea valorii acestuia în rest. Se modelează astfel un registru similar celui prezentat în figura 3.48.

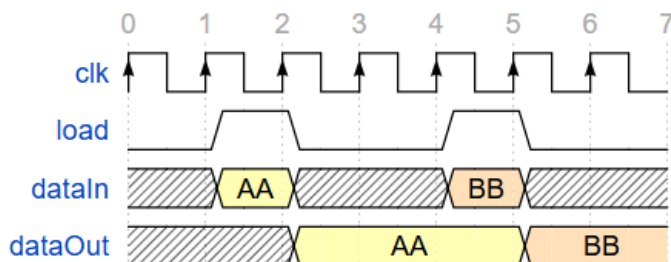


**Figura 3.48** Circuit cu registru cu încărcare paralelă.

Se presupune că semnalul *load* este un puls cu lățime de o singură perioadă de ceas, generat într-un sistem sincron (același semnal de ceas pentru toate bistabilele). În acest caz, registrul își modifică starea doar când  $load = 1$  și prezintă formele de undă din figura 3.49.

### Registru de deplasare, serie-paralel

Registrul cu încărcare serie și ieșire paralel, cu structura prezentată în figura 3.42, se modelează după cum urmează. Se observă că starea se modifică exclusiv în cazul  $shr = 1$ . Starea viitoare se obține prin deplasarea stării curente la dreapta și memorarea intrării seriale  $si$  pe cel mai semnificativ bit (bitul cel mai puțin semnificativ al stării prezente se pierde).



**Figura 3.49** Forme de undă pentru caracterizarea circuitului cu registru cu încărcare paralelă din figura 3.48. Încărcarea datelor se face când  $load = 1$ , la momentele 2 și 5.

```

module reg_serie_paralel #(
parameter WIDTH = 8    // latime registru, biti
) (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0
input          shr      , // incarcare seriala
input          si       , // data seriala
output reg [WIDTH-1:0] q    // iesire registru
);

always @(posedge clk or negedge rst_n)
if (~rst_n) q <= 'b0; else
if (shr)    q <= {si, q[WIDTH-1:1]}; // incarcare si
                                                // deplasare dreapta

endmodule    // reg_serie_paralel

```

### Registru de deplasare, paralel-paralel

Unui registru paralel cu încărcare paralelă i se poate adăuga o funcție suplimentară de deplasare (stânga, dreapta sau ambele direcții) în funcție de necesități. Modelul următor prezintă un registru paralel, cu încărcare paralelă și deplasare dreapta. Registrul are două intrări de control: una pentru încărcare și alta pentru deplasare. Prioritatea încărcării este modelată prin ordinea evaluării condițiilor în specificațiile `if`.

```

module reg_deplasare_paralel_paralel #(
parameter WIDTH = 8    // latime registru, biti
) (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0

```

```

input          load , // incarcare paralela
input          [WIDTH-1:0] d , // intrare
input          shr , // deplasare dreapta
input          si , // data seriala
output reg [WIDTH-1:0] q // iesire registru
);

always @(posedge clk or negedge rst_n)
if (~rst_n) q <= 'b0; else
if (load) q <= d; else // incarcare paralela
if (shr) q <= {si, q[WIDTH-1:1]}; // deplasare dreapta

endmodule // reg_deplasare_paralel_paralel

```

### Registru de rotație

Modelul următor prezintă un registru paralel-paralel cu facilități de rotire a datelor (atât spre dreapta cât și spre stânga). Se observă în model prioritatea intrărilor de control: *load*, *ror*, *rol*. Dacă nicio intrare de control nu este activată, registruul își menține starea. Pentru intrările de control s-au utilizat nume sugestive, utilizate frecvent în domeniul microprocesoarelor:

*ror* = *RO*tate *RI*ght,

*rol* = *RO*tate *LE*ft.

```

module reg_rotatie #(
parameter WIDTH = 8 // latime registru, biti
) (
input          clk , // ceas, activ front crescator
input          rst_n , // reset asincron activ 0
input          load , // incarcare paralela
input          [WIDTH-1:0] d , // intrare
input          ror , // rotatie dreapta
input          rol , // rotatie stanga
output reg [WIDTH-1:0] q // iesire registru
);

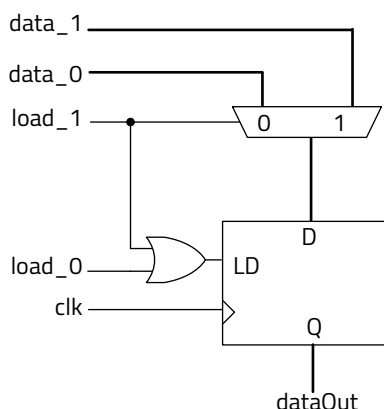
always @(posedge clk or negedge rst_n)
if (~rst_n) q <= 'b0; else
if (load) q <= d; else // incarcare paralela
if (ror) q <= {q[0], q[WIDTH-1:1]}; else // rotatie dreapta
if (rol) q <= {q[WIDTH-2:0], q[WIDTH-1]}; // rotatie stanga

endmodule // reg_rotatie

```

### Registru cu încărcare din surse multiple

Uneori este necesar ca un registru să fie încărcat din mai multe surse. Structura acestui registru este prezentată în figura 3.50.



**Figura 3.50** Circuit cu registru cu încărcare din două surse.

Se observă existența a două semnale de control a căror activare determină încărcarea registrului din două surse diferite. Modelul Verilog descrie comportamentul și rezolvă problema priorității celor două intrări de încărcare.

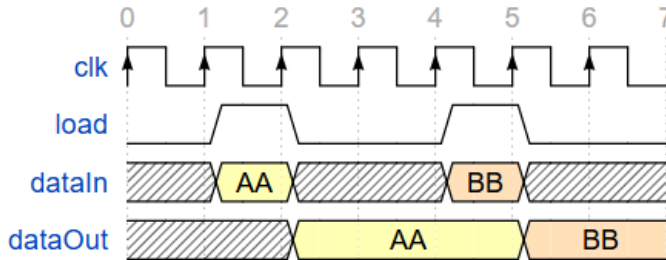
```
// regsitru paralel-paralel cu doua surse de incarcare
always @(posedge clk or negedge rst_n)
if (~rst_n) dataOut <= 'b0 ; else
if (load_1) dataOut <= data_1 ; else
if (load_0) dataOut <= data_0 ;
```

Formele de undă din figura 3.51 prezintă momentele de încărcare ale registrului pe baza celor două comenzi, din cele două surse. În lipsa vreunei comenzi de încărcare, registrul își păstrează starea.

### Registru cu funcții multiple

Registrele se pot descrie cu multiple funcționalități, conform necesităților proiectului. Comportamentul modelat este ușor de dedus din cod.

De exemplu, registrul modelat în continuare are 6 semnale de control



**Figura 3.51** Forme de undă pentru caracterizarea circuitului cu registru cu încărcare din două surse, din figura 3.50.

Dacă  $load_1 = 1$ , la momentele 2 și 5, se încarcă datele din sursa  $data_1$ .

Dacă  $load_1 = 0$  și  $load_2 = 1$ , la momentul 4, se încarcă datele din sursa  $data_2$ .

( $load, shl, shr, ror, inc, acc$ ) care determină starea următoare pe baza intrării de date  $dataIn$  și a stării curente a registrului: încărcare, deplasare stânga, deplasare dreapta, rotație dreapta, incrementare, acumulare.

```
module reg_misc #(
parameter WIDTH = 8    // latime registru, biti
) (
input                clk      , // ceas, activ front crescator
input                rst_n    , // reset asincron activ 0
input                load     , // incarcare paralel
input                [WIDTH-1:0] dataIn , // intrare de date
input                shl      , // deplasare stanga
input                shr      , // deplasare dreapta
input                ror      , // rotatie dreapta
input                inc      , // increment (+1)
input                acc      , // acumulare
output reg [WIDTH-1:0] dataOut // registru (iesire)
);

always @(posedge clk or negedge rst_n)
if (~rst_n) dataOut <= 'b0                                ; else
if (load)   dataOut <= dataIn                              ; else
if (shl)    dataOut <= (dataOut << 1)                      ; else
if (shr)    dataOut <= (dataOut >> 1)                      ; else
if (ror)    dataOut <= {dataOut[0], dataOut[WIDTH-1:1]} ; else
if (inc)    dataOut <= dataOut + 1                        ; else
if (acc)    dataOut <= dataOut + dataIn;

endmodule // reg_misc
```

În lipsa comenzilor active, registrul își păstrează starea (ultima specificație if nu are clauza else).



## Numărătoare

Modelul unui numărător este similar cu modelul unui registru, având însă funcția de a trece în starea viitoare provenită din starea curentă plus 1, implicit (la fiecare tact) sau în momentele determinate de o comandă externă.

Modelul unui numărător este prezentat în continuare.

```
module num_free_running #(
parameter WIDTH = 8    // latime registru, biti
) (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0
output reg [WIDTH-1:0] counter // numarator
);

always @(posedge clk or negedge rst_n)
if (~rst_n) counter <= 'b0      ; else // stare de reset
counter <= counter + 1  ;

endmodule // num_free_running
```

Acest numărător își modifică starea necondiționat, la fiecare front de ceas. În limba engleză, este denumit „numărător *free running*”. Nu este recomandat datorită faptului că, lucrând tot timpul, are un consum mare de energie.

Starea viitoare este starea curentă plus 1. Se remarcă faptul că numărătorul este totuși un automat cu număr finit de stări, nu numără la infinit. Cu  $WIDTH$  biți se pot număra  $2^{WIDTH}$  stări, între 0 și  $2^{WIDTH} - 1$ . Din starea  $2^{WIDTH} - 1$ , codată 1...1 (toți biții sunt 1) se trece în starea viitoare 0, având toți biții egali cu 0. Practic, graful stărilor este un inel cu  $2^{WIDTH}$  noduri (stări).

Modelul următor descrie un numărător sincron cu reset asincron activ în 0, cu intrare de validare *enable* și cu posibilitate de presetare. În momentul frontului crescător al semnalului de ceas, dacă *load* = 1, starea numărătorului devine *dataIn*, indiferent de starea curentă a numărătorului.

```
module num_preset #(
parameter WIDTH = 8    // latime registru, biti
) (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0
input          enable   , // validare functionare
input          load     , // intrare de presetare
input [WIDTH-1:0] dataIn , // date presetare
);
```

```

output reg [WIDTH-1:0] counter // numarator
);

always @(posedge clk or negedge rst_n)
if (~rst_n) counter <= 'b0 ; else
if (enable)
    if (load) counter <= dataIn ; else // incarca
        counter <= counter + 1 ; // numara crescator
// pastreaza starea

endmodule // num_preset

```

Prin descrierea unor condiții de modificare a stărilor, se pot modela numărătoare în diverse secvențe și coduri.

De exemplu, un numărător în cod BCD (în domeniul 0...9) se poate descrie prin codul următor. Se observă că după starea 9 urmează starea 0. Practic, numărătorul nu va atinge stările 10, 11, 12, 13, 14, 15. Comutarea stării se va face doar dacă *enable* = 1.

```

module count_bcd (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0
input          enable   , // validare functionare
output reg [4-1:0] counter // numarator 0 ... 9
);

always @(posedge clk or negedge rst_n)
if (~rst_n) counter <= 4'd0 ; else
if (enable)
    if (counter == 4'd9) counter <= 4'd0 ; else
        counter <= counter + 1 ;

endmodule // count_bcd

```

## Circuite LFSR

Circuitele LFSR se modelează conform rețetei prezentate în secțiunea 3.4.3. Codul Verilog, pentru un circuit LFSR de 5 biți pe baza polinomului  $X^5 + X^3 + 1$ , prezentat în tabelul 3.14, este următorul:

```

module lfsr_5b (
input          clk      , // ceas, activ front crescator
input          rst_n    , // reset asincron activ 0
input          enable   , // validare functionare
output reg [5-1:0] lfsr // polinom x5+x3+1
);

always @(posedge clk or negedge rst_n)

```

```

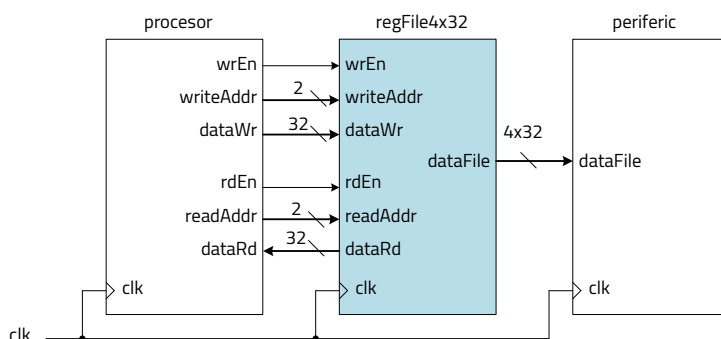
if (~rst_n) lfsr <= 'b1; else // stare initiala, diferita de 0
if (enable) lfsr <= {lfsr[3] ^ lfsr[0], lfsr[4:1]};

endmodule // lfsr_5b

```

## Set de registre

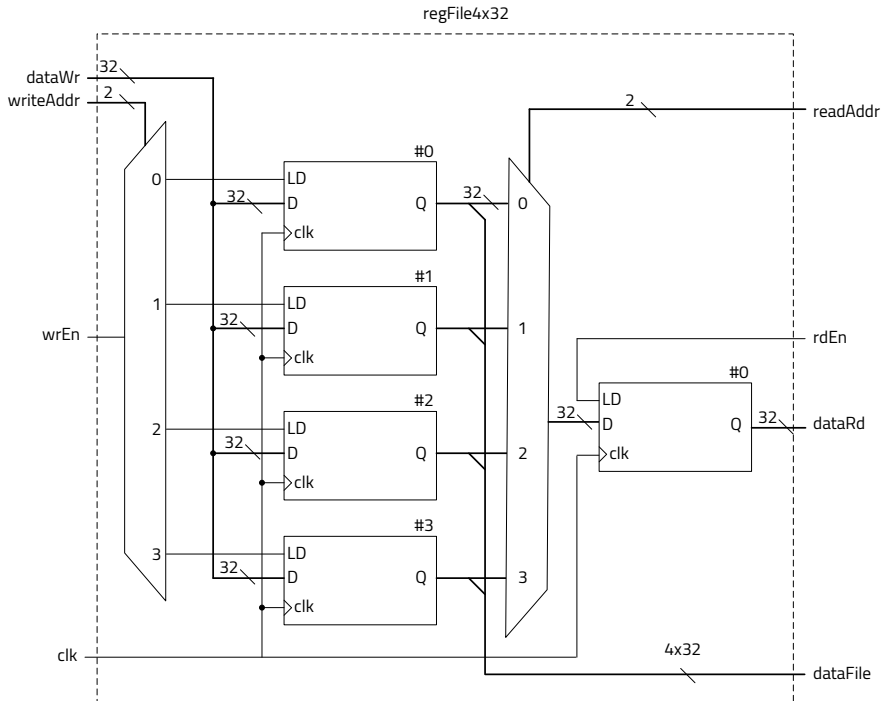
Pentru interfațarea între un microprocesor și un periferic se utilizează frecvent un set de registre („*register file*”, engl.). Setul de registre reprezintă un anumit număr de registre, posibil de accesat atât de către microprocesor (printr-o interfață standard), cât și de către periferic. Setul de registre reprezintă implementarea fizică a interfeței între software și hardware. Microprocesorul are acces prin software (prin program) la setul de registre pentru a comunica cu modulele hardware (periferele). Figura 3.52 prezintă poziția setului de registre între microprocesor și periferic.



**Figura 3.52** Poziția setului de registre între microprocesor și periferic.

Pe interfața cu microprocesorul, setul de registre are un singur port de scriere. La activarea scrierii  $wrEn = 1$ , pe baza adresei furnizate *writeAddr*, se selectează registrul în care se scriu datele *dataWr* trimise simultan la intrarea tuturor registrelor. La activarea citirii  $rdEn = 1$ , pe baza adresei furnizate *readAddr*, se selectează registrul din care se citesc datele pentru a fi returnate procesorului *dataRd*.

Conținutul tuturor registrelor *dataFile* poate fi accesat simultan, fără adresare, de către periferic. Figura 3.53 prezintă structura unui set de 4 registre de câte 32 biți de date. Se observă decodificarea adreselor de scriere și citire pentru selectarea unui registru din set. Perifericul are acces simultan și continuu la toate datele din setul de registre.



**Figura 3.53** Structura unui set de 4 registre de câte 32 de biți.

Modelul Verilog prezintă un set de 4 registre de câte 32 biți. Codul prezintă specificația de scriere a unui registru și specificația de citire a unui registru. Tot setul de registre este accesabil din exterior (de către periferic) prin faptul că variabila `dataFile` este declarată ca port de ieșire din modul.

```

module regFile4x32(
input          clk          , // semnal de ceas
// interfata de scriere
input          wrEn         , // comanda de scriere
input [2:0]    writeAddr    , // adresa locatie scrisa
input [32:0]   dataWr       , // data scrisa
// interfata de citire
input          rdEn         , // comanda de citire
input [2:0]    readAddr     , // adresa locatie citita
output reg [32:0] dataRd    , // data citita
// setul de registre
output reg [32:0] dataFile[3:0]

```

```
);

// scrierea unei locatii
always @(posedge clk)
if (rdEn) dataRd <= dataFile[readAddr];

// citirea unei locatii
always @(posedge clk)
if (wrEn) dataFile[writeAddr] <= dataWr;

endmodule // regFile4x32
```

## 3.5 Memorii

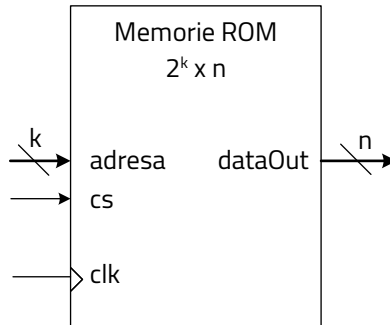
Memoriile reprezintă un capitol special în electronica digitală. Nu doar datorită faptului că tehnologia de realizare a memoriilor este diferită de tehnologia de realizare a porților logice și a bistabilelor, dar și utilizarea acestora este specială. Memoriile nu se sintetizează. Din acest motiv, modelele Verilog ale memoriilor au alte constrângeri: inițializare, verificarea datelor, respectarea protocoalelor de acces la date.

### 3.5.1 Memoria ROM sincronă

Memoria ROM („*Read Only Memory*”, engl.) este o memorie accesibilă doar în citire, în timpul funcționării normale. Memoria trebuie inițializată (scrisă) în avans prin proceduri, altele decât cele uzuale de citire, la manufacturare, în circuit sau cu programatoare speciale.

De multe ori, memoria ROM este sincronă, acțiunea de citire fiind controlată de un semnal de ceas. Din punct de vedere funcțional, memoria ROM prezintă o adresă, un semnal de control care determină citirea și datele de ieșire. Simbolul bloc al unei memorii ROM generice este prezentat în figura 3.54.

Modelul Verilog al memoriei ROM prezintă specificații de inițializare a datelor la momentul începerii simulării și specificații de citire a acestora prin accesarea de către sistemul exterior, în timpul funcționării normale, prin interfață. Datele cu care se inițializează memoria ROM pot fi date preluate dintr-un fișier extern sau date generate printr-un algoritm din modelul Verilog. De remarcat faptul că inițializarea datelor este o chestiune legată strict de simulare, fără corespondent în lumea reală. Inițializarea reală a memoriei ROM este o chestiune specifică tehnologiei de implementare a circuitului integrat.



**Figura 3.54** Simbol de memorie ROM generică, sincronă.

```

module rom #(
parameter ADDR_WIDTH = 8 , // latime adrese, biti
parameter DATA_WIDTH = 8 // latime date, biti
) (
input          clk      , // ceas
input          cs       , // chip select
input [ADDR_WIDTH-1:0] adresa , // adrese
output reg [DATA_WIDTH-1:0] dataOut // date
);

// Variabila bidimensionala cu datele memoriei ROM
reg [DATA_WIDTH-1:0] rom_mem [(2**ADDR_WIDTH)-1:0];

// Initializare memorie dintr-un fisier extern
initial
    $readmemh("rom_data.hex", rom_mem);

// Logica sincrona pentru citirea datelor din ROM
always @(posedge clk)
    if (cs) dataOut <= rom_mem[adresa];

endmodule // rom

```

Modelul prezentat modifică datele pe ieșire doar dacă chip-ul este selectat ( $cs = 1$ ) și le păstrează până la următoarea citire.

Tabelul de memorie este declarat ca o variabilă Verilog bidimensională  $2^{ADDR\_WIDTH} \times DATA\_WIDTH$ . Inițializarea conținutului memoriei se face la momentul 0 (zero) al simulării cu date provenite dintr-un fișier extern. Datele sunt citite cu specificația `$readmemh`. Formatul fișierului

presupune existența unei valori hexazecimale pe fiecare linie a fișierului. Datele se încarcă în ordine, de la adresa 0.

De remarcat faptul că ChatGPT a făcut o treabă bună și a avut nevoie de o minimă corecție (vezi dialogul din figura 3.55).

 Iată un exemplu complet de modul Verilog pentru o memorie ROM sincronă, cu interfața specificată:

```
verilog Copy code

module rom #(
    parameter ADDR_WIDTH = 8, // lățime adrese, biți
    parameter DATA_WIDTH = 8 // lățime date, biți
)(
    input                clk,      // ceas
    input                cs,       // chip select
    input    [ADDR_WIDTH-1:0] adresa, // adrese
    output reg [DATA_WIDTH-1:0] dataOut // date
);

// Declararea memoriei ROM
reg [DATA_WIDTH-1:0] mem [(2*ADDR_WIDTH)-1:0];

// Inițializarea memoriei ROM cu date
initial begin
    $readmemh("rom_data.hex", mem); // Citire dintr-un fișier extern (opțional)
end

// Logică sincronă pentru accesarea datelor
always @(posedge clk) begin
    if (cs) begin
        dataOut <= mem[adresa]; // Citire din memorie
    end
end

endmodule
```

**Figura 3.55** Dialog cu  ChatGPT referitor la generarea unui model Verilog pentru o memorie ROM sincronă.

Funcționalitatea unei memorii ROM este similară cu accesarea unui tabel de date pe baza unei adrese. Din acest motiv, memoriile ROM sunt uneori denumite „LUT”, după expresia din limba engleză „Look-Up Table”. Codul următor prezintă o memorie ROM a cărei conținut este precizat direct în cod. Specificația case asociază fiecărei adrese conținutul necesar.

```
module lut(
    input                clk, // ceas
    input    [4 -1:0] adresa, // adrese
```

```

output reg [32 -1:0] dataOut // date iesire
);
always @(posedge clk)
case (adresa)
4'd0 : dataOut = 32'h00_00_00_00 ;
4'd1 : dataOut = 32'h00_0F_1B_1F ;
4'd2 : dataOut = 32'h00_14_24_32 ;
4'd3 : dataOut = 32'h00_18_2B_3F ;
4'd4 : dataOut = 32'h00_1B_30_49 ;
4'd5 : dataOut = 32'h00_1E_34_52 ;
4'd6 : dataOut = 32'h02_20_38_59 ;
4'd7 : dataOut = 32'h04_22_3C_5F ;
4'd8 : dataOut = 32'h06_24_3F_64 ;
4'd9 : dataOut = 32'h08_26_42_69 ;
4'd10 : dataOut = 32'h09_27_45_6E ;
4'd11 : dataOut = 32'h0B_29_48_72 ;
4'd12 : dataOut = 32'h0C_2A_4B_75 ;
4'd13 : dataOut = 32'h0E_2C_4D_79 ;
4'd14 : dataOut = 32'h0F_2D_4F_7C ;
4'd15 : dataOut = 32'h10_2E_52_7F ;
default: dataOut = 32'bx ;
endcase

endmodule // lut

```

### 3.5.2 Memoria RAM

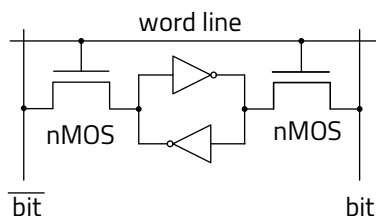
Memoria RAM („*Random Acces Memory*”, engl.) este o memorie accesată în circuit atât pentru scrierea cât și pentru citirea datelor. Memoria RAM are structura unui circuit secvențial, conținând elemente de memorare a datelor. În funcție de modul de stocare a informației, memoriile RAM se clasifică în memorii statice și memorii dinamice.

#### Memoria statică SRAM

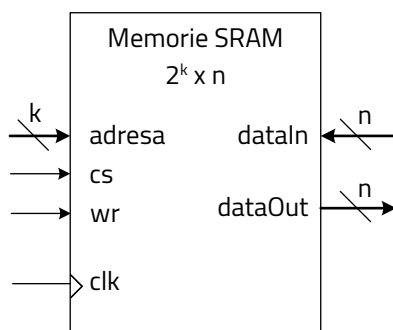
Memoria SRAM („*Static RAM*”, engl.) stochează informația într-o celulă de memorie cu o structură bazată pe un circuit latch (două inversoare conectate în buclă). Celula de memorie din figura 3.56 se numește **statică** deoarece cele două inversoare, sunt capabile să își păstreze starea, atâta timp cât circuitul este alimentat.

Simbolul bloc al unei memorii SRAM (figura 3.57) prezintă bus-uri de date de intrare și de ieșire, bus-ul de adrese și semnale de control care comandă validarea chip-ului (CS - Chip Select) și scrierea datelor (WR - WRite).





**Figura 3.56** Celula de memorie statică SRAM (latch).



**Figura 3.57** Simbolul bloc al unei memorii SRAM generice, sincrone,  $2^k \times n$ .

Modelul Verilog al memoriei SRAM prezintă specificații de accesare a datelor în scriere și în citire, prin comenzi specifice funcționării normale, cu semnalele prezente în interfață.

Opțional, din motive de simulare, se poate modela inițializarea memoriei înainte de începerea simulării. De exemplu, este posibil ca memoria să fie scrisă și citită de către module diferite, proiectate de către ingineri diferiți. Modulul de citire își începe activitatea comandat de un semnal activat numai după ce modulul de scriere a terminat de scris (parțial sau total) spațiul de adrese RAM. Însă, în simularea modulului de citire, este de preferat ca memoria să fie gata populată cu un set de date coerente, procesul de scriere nefiind relevant.

```
module sram #(
parameter ADDR_WIDTH = 8 , // latime adrese, biti
```

```

parameter DATA_WIDTH = 8      // latime date, biti
)(
input          clk      , // ceas
input          cs       , // chip select
input          wr       , // comanda scriere
input          [ADDR_WIDTH-1:0] adresa , // adrese
input          [DATA_WIDTH-1:0] dataIn  , // date intrare
output reg     [DATA_WIDTH-1:0] dataOut  // date iesire
);

// datele memoriei SRAM - matrice bidimensionala
reg [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0];

// scriere in memorie
always @(posedge clk)
if (cs & wr) mem[adresa] <= dataIn;

// citire din memorie
always @(posedge clk)
if (cs & (~wr)) dataOut <= mem[adresa];

endmodule // sram

```

Se observă că scrierea are loc dacă chip-ul este selectat ( $cs = 1$ ) și comanda de scriere este activată ( $wr = 1$ ). În schimb, citirea are loc dacă chip-ul este selectat ( $cs = 1$ ) și comanda de scriere nu este activată ( $wr = 0$ ). În lipsa selecției chip-ului ( $cs = 0$ ), datele citite sunt menținute pe portul de ieșire de date al memoriei SRAM.

Uneori se dorește scrierea parțială a datelor existente la o adresă. Pentru aceasta este nevoie de o mască de biți care să precizeze ce date se scriu și ce date se păstrează. Mascarea datelor la nivel de byte se face cu un semnal denumit „*byte-enable*”. Funcționalitatea este prezentată de codul următor.

Se observă că la o lățime de date de  $DATA\_WIDTH$  biți îi corespunde o intrare suplimentară de  $DATA\_WIDTH/8$  biți (câte un bit pentru fiecare byte).

Dacă  $be = 1$ , datele se scriu în memorie, iar dacă  $be = 0$ , datele sunt mascate (nu se scriu în memorie, ci se păstrează cele deja existente).

```

module sram #(
parameter ADDR_WIDTH = 8 , // latime adrese, biti
parameter DATA_WIDTH = 8 // latime date, biti
)(
input          clk      , // ceas
input          cs       , // chip select
input          wr       , // comanda scriere

```

```

input      [ADDR_WIDTH -1:0] adresa , // adrese
input      [DATA_WIDTH/8-1:0] be      , // byte-enable
input      [DATA_WIDTH -1:0] dataIn   , // date intrare
output reg [DATA_WIDTH -1:0] dataOut  // date iesire
);

// datele memoriei SRAM - matrice bidimensionala
reg [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0];
integer i;

// scriere in memorie
always @(posedge clk)
if (cs & wr)
    for (i=0; i<(DATA_WIDTH/8); i=i+1)
        if (be[i])
            mem[adresa][(i*8)+:8] <= dataIn[(i*8)+:8];

// citire din memorie
always @(posedge clk)
if (cs & (~wr)) dataOut <= mem[adresa];

endmodule // sram

```

Un tabel de memorie poate fi accesat concomitent pe mai multe porturi (interfețe), la mai multe locații, dacă nu sunt accese concurente la aceeași locație. Pentru fiecare locație accesată trebuie precizată o adresă.

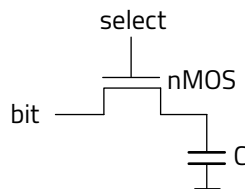
Numărul de porturi ale unei memorii este reprezentat de numărul de adrese posibil de accesat concurrent (simultan).

În spațiul de memorie SRAM se poate avea acces doar la locația aflată la adresa plasată pe portul de citire. Nu pot fi accesate toate locațiile simultan, ca la un set de registre.

### Memoria dinamică DRAM

Memoria DRAM („*Dynamic RAM*”, engl.), stochează informația sub forma de **sarcină electrică** într-o celulă de memorie cu o structură bazată pe un capacitor. Celula de memorie din figura 3.58 se numește **dinamică** deoarece sarcina capacitivă stocată se diminuează în timp datorită pierderilor capacitorului. Din acest motiv, memoriile DRAM necesită periodic o acțiune de „împrospătare” („*refresh*”, engl.) a sarcinii electrice stocate pentru refacerea informației înscrise în memorie. Aceasta este un dezavantaj al memoriilor DRAM față de cele SRAM. Însă, memoriile DRAM au avantajul unor celule de memorie de dimensiune mult mai mică care facilitează realizarea unor memorii de dimensiuni mult mai mari într-o capsulă. Dimensiunea mare a matricii de memorie integrate pe un chip ar determina

atât un număr mare de pini, cât și necesitatea realizării acțiunii de reîmprospătare la multe locații într-un interval de timp. Reducerea numărului de porturi ale memoriei DRAM și realizarea acțiunii de împrospătare la mai multe locații de adresă simultan, a determinat modificarea matricii de memorie prin înlocuirea accesării liniare pe bază de adresă cu o accesare bidimensională, secvențială de tipul *rând și coloană*. Bus-ul de adresă are semnificația de „rând” dacă este activat semnalul **RAS - Row Address Strobe** și semnificația de „coloană” dacă este activat semnalul **CAS - Column Address Strobe**.

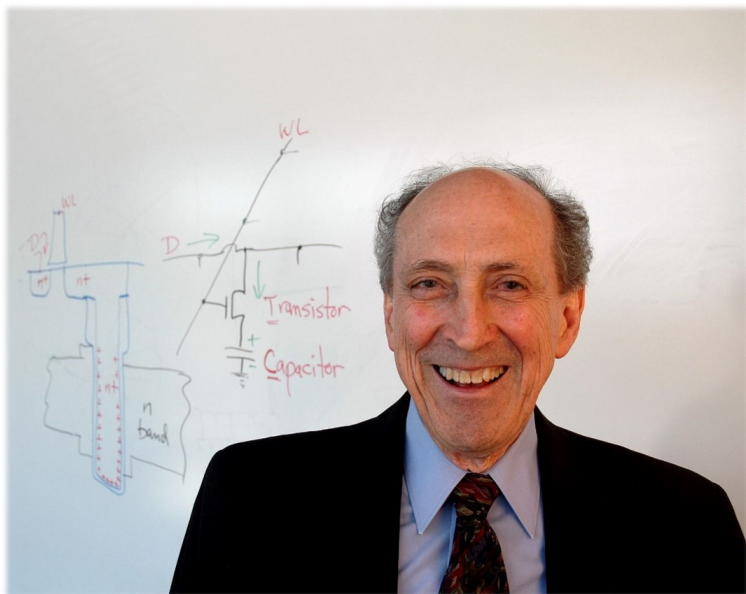


**Figura 3.58** Celula de memorie dinamică DRAM (capacitor MOS).

Celula de memorie DRAM a fost inventată în anul 1966 de către [Robert Dennard](#) (figura 3.59).

Simbolul bloc al unei memorii DRAM (figura 3.60) prezintă un bus de date bidirecțional (intrare și ieșire), bus-ul de adrese și semnale de control. Semnalele de control (*cs*, *wr*, *ras* și *cas*) comandă validarea chip-ului, semnificația adresei (rând sau coloană), scrierea datelor, acțiuni de inițializare și împrospătare a memoriei. Se remarcă faptul că la memoria DRAM o adresă de locație este împărțită în două părți (rând și coloană) și este prezentată secvențial pe același port de adrese (având lățimea jumătate din lățimea busului de adrese la o memorie SRAM de dimensiune identică).

Producătorii de memorii DRAM pun la dispoziția proiectanților modele complexe ale memoriilor DRAM, capabile să modeleze realist atât temporizarea cât și gestiunea și verificarea datelor stocate în memoriile DRAM astfel încât proiectantul poate integra modelul de memorie în simularea întregului sistem digital proiectat. Interfațarea cu o memorie DRAM trebuie să rezolve atât accesul unui modul la datele stocate în memorie cât și gestiunea reîmprospătării la timp a tuturor locațiilor memoriei. Din acest motiv, memoriile DRAM au nevoie de un modul specific denumit „*controller de memorie*” care să preia responsabilitatea controlării memoriei conform specificațiilor și să permită proiectantului controlul facil al operațiilor de



**Figura 3.59** Robert Dennard (1932-2024) - inventatorul celulei de memorie dinamice, cu un singur tranzistor.

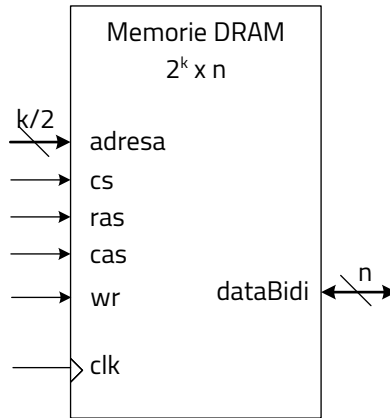
scriere și citire a datelor, similar cu accesarea unui tabel de date (adresă și date).

Interfața de date a memoriilor DRAM are **semnale bidirecționale**. Porturile bidirecționale sunt capabile de a transmite secvențial semnale în ambele direcții (date citite din memorie sau scrise în memorie). Pentru a realiza porturi bidirecționale este nevoie de un tip particular de porți logice capabile de a-și trece ieșirea într-o stare specială cu impedanță foarte mare (teoretic infinită).

**Poarta logică în 3 stări** prezintă, pe lângă cele două stări binare (0 și 1), o stare caracterizată printr-o impedanță de ieșire mare, ideal infinită. Ieșirea porții logice în stare de înaltă impedanță, simbolizată **H<sub>iz</sub>** („High Z”, engl.) poate fi conectată la ieșirea altei porți logice, fără a afecta funcționalitatea acesteia.

Poarta în 3 stări are două intrări: o *intrare de date* și o *intrare de validare*. În funcție de stările active ale celor două intrări, există 4 tipuri de porți în 3 stări, ale căror simboluri și tabele de funcționare sunt prezentate în figura 3.61.

Limbajul Verilog oferă suport pentru reprezentarea datelor prin existența



**Figura 3.60** Simbolul bloc al unei memorii DRAM generice, sincronă,  $2^k \times n$ .

în setul de valori a valorii  $z$  (doar cu caracter minusculă). Valoarea  $z$ , înaltă impedanță, modelează un circuit electric deschis, fără sursă. Un nod de circuit poate avea mai multe surse, dacă sursele pot fi comandate în starea  $z$ .

Valoarea logică a unui nod cu surse multiple este determinată de sursele exterioare celei ce comandă ieșirea în  $z$  (figura 3.62).

Simulatoarele HDL marchează starea  $z$  cu o linie de altă culoare și la un nivel intermediar, între 0 și 1.

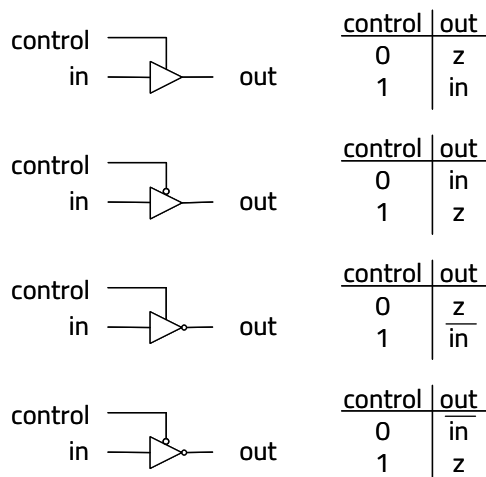
În limbajul Verilog există specificații implicite definite de limbaj pentru circuite în 3 stări. Corelația dintre ordinea porturilor în specificație și semnificația acestora este prin poziție. Întotdeauna ordinea este: ieșire, intrare și apoi control.

```
bufif1 (out, in, control);
notif1 (out, in, control);
bufif0 (out, in, control);
notif0 (out, in, control);
```

Se poate modela o poartă în 3 stări și cu operatorul ternar dacă, în anumite condiții,  $i$  se atribuie valoarea  $z$ .

```
assign out = control ? in : 'bz;
```

Porțile în 3 stări (cunoscute în limba engleză sub denumirea de „*three-state*” sau „*tri-state*”) sunt utilizate în aplicații care necesită conectarea mai



**Figura 3.61** Porți în 3 stări (simboluri și tabele de funcționare).

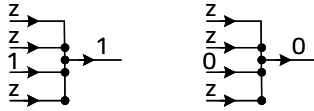
multor ieșiri de circuite logice împreună.

- **Porți pentru magistrale** (semnalele de validare a mai multor porți sunt controlate astfel încât, la un moment dat, o singură poartă să determine starea logică a magistralei, restul fiind în stare de înaltă impedență).
- **Circuite de interfațare cu porturi bidireționale** (poarta în 3 stări este în stare de înaltă impedență când datele intră în circuit și se activează doar când circuitul respectiv trimite date spre exterior).
- **Extinderea capacității memoriilor dinamice** (conectarea mai multor chip-uri de memorie pentru a realiza un modul de memorie de dimensiune mai mare).

Conversia semnalelor unidireționale într-un semnal bidirecțional se poate face cu un circuit bazat pe o poartă în 3 stări.

Se observă că dacă  $dir = 0$ , poarta în 3 stări este blocată și semnalul se propagă de la portul bidirecțional  $data\_bidi$  spre portul de intrare unidirecțional  $data\_in$  (direcția intrare).

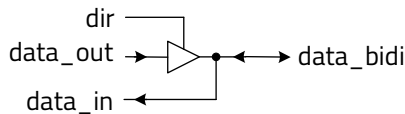
În schimb, dacă  $dir = 1$ , poarta în 3 stări este deschisă și semnalul se propagă de la portul unidirecțional de ieșire  $data\_out$  spre portul de bi-



**Figura 3.62** Starea unui nod cu 4 surse, dintre care 3 sunt în stare de înaltă impedanță.

directional *data\_bidi* (direcția ieșire). Evident, în acest caz, semnalul se regăsește și la portul de intrare unidirectional *data\_in*.

Circuitul care transformă două porturi unidirectionale într-un port bi-direcțional este prezentat în figura 3.63.



**Figura 3.63** Driver bidirecțional.

Modelul Verilog al unui driver bidirecțional pentru o magistrală de dimensiune parametrizabilă este prezentat în continuare.

```
module bidi #(
parameter DATA_WIDTH = 8    // latime date, biti
) (
input          dir            , // 1=iesire, 0=intrare
input [DATA_WIDTH-1:0] data_out , // date de iesire
input [DATA_WIDTH-1:0] data_in  , // date de intrare
inout [DATA_WIDTH-1:0] data_bidi // date bidirectionale
);

assign data_bidi = dir ? data_out : 'bz; // out -> bidi
assign data_in   = data_bidi;           // in  <- bidi

endmodule // bidi
```

De remarcat ChatGPT, care a făcut o treabă bună și pentru acest model



(figura 3.64).



Iată un model complet de buffer bidirecțional în Verilog, conform interfeței specificate:

```
verilog

module bidi #(
    parameter DATA_WIDTH = 8    // lățime date, biți
)(
    input          dir,           // 1=ieșire, 0=intrare
    input [DATA_WIDTH-1:0] data_out, // date de ieșire
    input [DATA_WIDTH-1:0] data_in,  // date de intrare
    inout [DATA_WIDTH-1:0] data_bidi // date bidirecționale
);

// Buffer bidirecțional
assign data_bidi = dir ? data_out : {DATA_WIDTH{1'bz}}; // Ieșire pe bidirecțional dac
assign data_in = dir ? {DATA_WIDTH{1'bz}} : data_bidi; // Intrare din bidirecțional d

endmodule
```

**Figura 3.64** Dialog cu  ChatGPT referitor la generarea unui model Verilog pentru un driver bidirecțional.

ChatGPT furnizează și explicații asupra modelului generat (figura 3.65), inclusiv sfaturi foarte corecte (vezi „*Constrângeri suplimentare*”).

Porturile de date ale memoriilor DRAM sunt bidirecționale, fiecare semnal având asociat, în interiorul memoriei, câte un circuit driver bidirecțional similar cu cel prezentat în figura 3.63. Activarea semnalelor de control pentru scriere, determină selectarea direcției de intrare în chip (circuitul exterior furnizează date). Activarea semnalelor de control pentru citire, determină selectarea direcției de ieșire din chip (circuitul exterior primește date). Evident, și circuitul exterior memoriei DRAM trebuie să aibă date bidirecționale. Comanda direcției de transfer a datelor trebuie sincronizată între memorie și circuitul exterior prin grija sistemului digital din care fac parte acestea.

### 3.5.3 Structuri de memorii

Uneori, memoriile nu se accesează la adrese aleatorii, ci la adrese succesive, într-o anumită ordine. Acest lucru poate determina simplificarea

**Explicație:**

1. `data_bidi` ca inout:
  - Dacă `dir` este 1 (ieșire): Semnalul `data_out` este transmis pe linia bidirecțională `data_bidi`.
  - Dacă `dir` este 0 (intrare): Linia bidirecțională `data_bidi` este plasată în stare de **high-impedance** (`Z`), iar valoarea sa este citită pe `data_in`.
2. Stare de **high-impedance** (`Z`):
  - Aceasta este utilizată pentru a evita conflictele atunci când linia bidirecțională este folosită pentru citire.
3. Parametrul `DATA_WIDTH`:
  - Dimensiunea lățimii semnalelor este configurabilă (ex. 8 biți, 16 biți etc.).
4. Constrângeri suplimentare:
  - Asigură-te că doar un singur modul accesează `data_bidi` ca ieșire la un moment dat pentru a evita conflictele pe magistrală.

Dacă ai întrebări suplimentare sau modificări necesare, spune-mi! 😊

**Figura 3.65** Explicații  ChatGPT referitoare la modelul Verilog pentru un driver bidirecțional.

comenzilor necesare pentru accesul la memorie și eliminarea precizării de către modulul client a adresei locației accesate.

## Memoria FIFO

Memoria FIFO („*First In First Out*”, engl.) are o funcționalitate similară cu cea a unei cozi de așteptare: „*primul venit este primul servit*”. Pentru acest tip de memorie nu este necesară precizarea adresei de memorie deoarece locațiile se accesează în ordine, atât pentru scriere cât și pentru citire. Din acest motiv, gestiunea adreselor poate fi realizată în cadrul modulului de memorie, degrevând modulul client de această sarcină. Practic, modulul client precizează doar momentele când datele sunt disponibile pentru a fi scrise în memorie. Datele se vor înscrie la locații succesive. Citirea are loc în aceeași ordine, memoria furnizând secvențial datele de la adrese consecutive. Deci, datele ies din memorie exact în ordinea în care au intrat în memorie, la un moment dat putând fi stocate în memorie un anumit număr de date aflate în așteptare (figura 3.66).

Memoria FIFO poate implementa diverse funcții în sisteme digitale:



**Figura 3.66** Memoria FIFO are un comportament similar cu cel al unei cozi de așteptare: datele se citesc în ordinea în care au fost scrise.

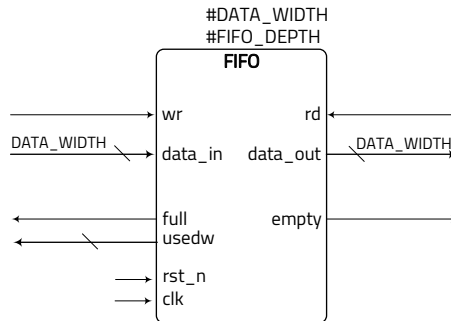
- **Coadă de date.** Datele sunt scrise în memorie în ordinea sosirii și citite în aceeași ordine. Scrierea și citirea pot fi la momente de timp diferite, necorelate.
- **Linie de întârziere.** Pentru situațiile când este nevoie de întârzierea cu mai multe perioade de ceas a unui flux de date, de exemplu o linie într-o memorie video.
- **Buffer de date** între etaje de procesare pipeline.
- **Sincronizarea datelor** la trecerea între două domenii de ceas.

Memoria FIFO este caracterizată de adâncimea memoriei, adică numărul maxim de date ce pot fi stocate la un moment dat.

Simbolul memoriei FIFO este prezentat în figura 3.67. Simbolul sugerează că lățimea datelor și adâncimea memoriei FIFO (numărul de locații) sunt parametrizate.

Pentru gestiunea adresei, memoria păstrează adresele curente de scriere și citire (pointeri de adrese) în două numărătoare. Valorile numărătoarelor sunt incrementate la fiecare acces ( $wr=1$  sau  $rd=1$ ). Dacă numărătoarele ajung la valoarea maximă, prin incrementare revin la valoarea zero. Memoria poate implementa indicatori care să semnaleze umplerea sau golirea memoriei pentru a evita accese ilegale (scriere în memorie plină  $full=1$  sau citire din memorie goală  $empty=1$ ). Opțional, memoria FIFO poate furniza informația referitoare la numărul de locații pline  $usedw$ .

În figura 3.68 se observă modul de gestionare a spațiului de adresare al memoriei, pe baza celor doi pointeri de adresă ( $wr\_ptr$  și  $rd\_ptr$ ). Porțiunea



**Figura 3.67** Simbolul memoriei FIFO.

hașurată, între  $wr\_ptr$  și  $rd\_ptr$ , reprezintă zona de memorie ocupată. Partea stângă prezintă cazul în care  $wr\_ptr$  nu a ajuns la maxim și valoarea  $wr\_ptr$  este mai mare decât valoarea  $rd\_ptr$ . La atingerea valorii maxime, pointerul  $wr\_ptr$  este readus la valoarea zero. Din acest moment, valoarea  $wr\_ptr$  este mai mică decât  $rd\_ptr$ , iar spațiul de memorie ocupat este divizat, așa ca în figura din partea dreaptă. La atingerea valorii maxime de către pointerul  $rd\_ptr$ , situația revine la cea inițială. Practic, cei doi pointeri se „învârtesc”, fără a se depăși unul pe altul.

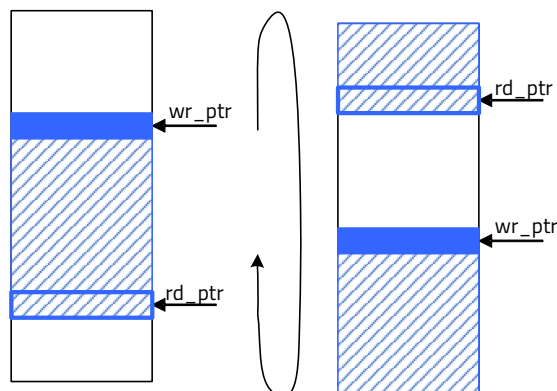
De observat că, eliberarea unei locații după citire nu determină dispariția conținutului memoriei de la acea locație, ci doar rămânerea locației în afara zonei de locații ocupate.

Memoria FIFO este considerată goală dacă pointerii se egalează după ce  $wr\_ptr$  a fost mai mare decât  $rd\_ptr$ .

Memoria FIFO este considerată plină dacă pointerii se egalează după ce  $wr\_ptr$  a fost mai mic decât  $rd\_ptr$ .

Modelul Verilog pentru memoria FIFO prezintă cele două numărătoare, utilizate ca pointeri de adresă, și incrementate în momentele comenzilor de scriere și citire. Ca memorie se poate instanția o memorie SRAM cu două porturi de adrese sau se poate descrie ca un set de registre cu două porturi (de scriere și de citire).

Memoria FIFO poate fi asociată cu o memorie cu două porturi și utilizată pe domenii de ceas diferite pentru scriere și citire. În acest mod se poate realiza soluția corectă de trecere a unui flux de date de pe un domeniu de ceas pe altul, desigur în cazul îndeplinirii constrângerii unor rate de transfer egale pe ambele domenii de ceas (asincrone). În acest caz, scrierea și citirea din memoria FIFO se pot face simultan.



**Figura 3.68** Spațiul de adresare al memoriei FIFO. Hașurat spațiul ocupat.

### Memoria LIFO

Memoria LIFO („*Last In First Out*”, engl.) are o funcționalitate similară cu cea a unei stive: „*ultimul obiect depus pe stivă este primul obiect preluat din stivă*”. Pentru acest tip de memorie nu este necesară precizarea adresei de memorie, deoarece locațiile se accesează în ordine, atât pentru scriere cât și pentru citire. Din acest motiv, gestiunea adreselor poate fi realizată în cadrul modulului de memorie, degrevând modulul client de această sarcină.

Memoria LIFO este utilizată pentru refacerea datelor în ordinea inversă a stocării acestora, similar cu punerea farfuriilor una peste alta, una câte una (figura 3.69).

Memoria LIFO poate implementa diverse funcții în sisteme digitale:

- **Memoria stivă.** Datele citite sunt ultimele date scrise. Este implementarea hardware care oferă suport pentru saltul programelor executate de microprocesoare la apelarea funcțiilor sau pentru tratarea întreruperilor.
- **Inversarea datelor.** Scoaterea datelor în ordinea inversă a stocării.

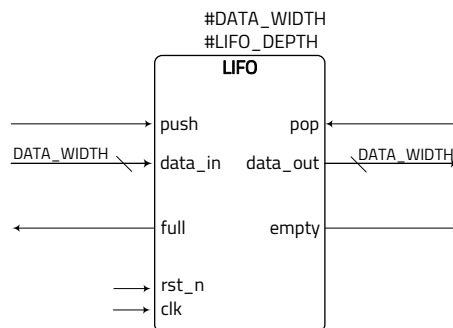
Memoria LIFO este caracterizată de adâncimea memoriei, adică numărul maxim de date ce pot fi stocate, în caz că datele nu sunt citite.

Simbolul memoriei LIFO este prezentat în figura 3.70. Simbolul sugerează



**Figura 3.69** Memoria LIFO are un comportament similar cu o stivă. Accesele în scriere și citire se fac la vârful stivei.

rează că lățimea datelor și adâncimea memoriei LIFO (numărul de locații) sunt parametrizate.



**Figura 3.70** Simbolul memoriei LIFO.

Pentru a memora adresa vârfului stivei de date, memoria LIFO are un numărator dedicat, denumit în limba engleză „*stack pointer*”, precurtat *SP*.

Scrierea unei date în memoria stivă este denumită în limba engleză „*push*” și constă în preluarea datelor de la client, scrierea acestora la adresa conținută în registru *SP* și apoi incrementarea acestui registru pentru a pointera spre următoarea locație disponibilă.

Citirea unei date din memoria stivă este denumită în limba engleză „*pop*” și constă în decrementarea registrului *SP*, citirea datelor de la adresa conținută în registru *SP* și trimiterea acestora spre modulul client.

Registrul *SP* este implementat ca un numărător reversibil capabil să se incrementeze la comanda „*push*” și să se decrementeze la comanda „*pop*”.

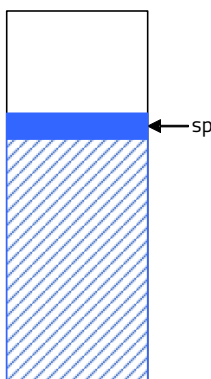
Valoarea pointerului vârfului stivei *SP* se incrementează și decrementează între zero și valoarea maximă a numărului de locații disponibile.

Dacă *SP* ajunge la valoarea maximă, înseamnă că stiva este plină (*full=1*) și nu se mai pot accepta date (comenzi „*push*”).

Dacă *SP* are valoarea zero, înseamnă că stiva este goală (*empty=1*) și nu se pot livra date (comenzi „*pop*”).

În figura 3.71 se observă modul de gestionare a spațiului de adresare al memoriei, pe baza pointerului *SP*. Porțiunea hașurată, între 0 și *SP*, reprezintă zona de memorie ocupată. Pointerul *SP* variază între 0 și maximum, niciodată nu trece de la maximum la zero.

Modelul Verilog pentru memoria LIFO prezintă numărătorul *SP*, utilizat ca adresă a locației din vârful stivei, incrementat la comenzi de scriere și decrementat la comenzi de citire.



**Figura 3.71** Spațiul de adresare al memoriei LIFO. Hașurat spațiul ocupat.

Se poate proiecta memoria LIFO pentru a accepta simultan comenzi *push* și *pop*. În acest caz datele scrise sunt livrate direct ca date citite, fără a mai fi trecute prin memoria stivă.





---

## Capitolul 4

### Considerente ingineresti ale sistemelor digitale





### Predoslovie la Capitolul al Patrulea:

#### **În care inginererea se încâlcește cu vremea, firul subțire și limba ciudată a Verilogului**

Drept e că până aici ne-am dat în bărci logice, am jucat flip-flop ca-n tinerețile digitalului și-am învățat să iubim și-un zero bine pus la locul lui. Da' de-acum, domnule cititor, gluma se îngroașă (cam ca manualele de standarde) și intrăm cu pași ingineri în tărâmul **considerentelor tehnice**, unde teoria dă din mână a „ajutor!” și realitatea trage cu osciloscopul înapoi.

În acest capitol de pe urmă — dar nu și cel mai de pe urmă cu înțelepciunea — vom cugeta adânc la **timp** (nu cel filozofic, ci cel de propagare, de latență, de sincronizare, de ceasuri care ticăie doar în frecvențe de sute de MHz), vom privi cu luare-aminte la **tehnologiile de fabricare** ale circuitelor integrate, care-s atât de mici, că dacă le scapi pe jos, trebuie microscop ca să le plângi.

De asemenea, vom zice câteva vorbe despre **protocoloale de comunicare**, acele limbaje tainice prin care un chip dă bună ziua altuia — dar numai dacă-i exact la oră și respectă formatul. Altfel, nimeni nu mai vorbește cu nimeni, ca la o nuntă cu locurile greșit aranjate.

Și în cele din urmă, dar nu în cele din urmă ca durere de cap, pătrundem în lumea încurcată și fascinantă a limbajului **Verilog** — o unealtă de scris nu povești, ci circuite. Aici nu mai e destul să știi ce-i un AND și un OR, ci trebuie să le povestești computerului, ca să le plămădească el, cândva, în siliciu.

Capitolul acesta e, cum s-ar zice, cireșa de pe schema logică — locul unde teoria își suflecă mânecile și dă cu capul în realitate. Iar tu, cititor vrednic, dacă ai ajuns până aici, meriți o bătaie prietenească pe umăr... și poate un debugger bun.

Să încheiem dară cu gândul că **ingineria digitală nu-i doar știință și logică, ci și răbdare, încercare, și-un pic de rugăciune la datasheet.**

*Text generat cu ChatGPT.*

Electronica digitală este fundamentată pe algebra Booleană. Matematica permite reprezentarea funcțiilor de transfer ale circuitelor integrate digitale. Însă nu este suficientă pentru caracterizarea comportamentului circuitelor electronice care implementează funcțiile logice. Elementele constitutive ale circuitelor integrate digitale (portile logice și bistabilele) nu sunt operatori logici abstracti ci implementări reale, cu dispozitive electronice, ce permit transferul sarcinilor electrice controlat. Limitările impuse de realizabilitatea fizică a circuitelor sunt legate de timpul de propagare nenul, curenții și tensiunile prin circuite, sarcinile capacitive parazite, dimensiunile fizice, rutarea firelor, rezistența electrică a firelor, variația parametrilor electrici cu temperatura.

De exemplu, matematic se poate imagina un operator AND cu 100 de intrări. Se poate și descrie în Verilog astfel:

```
wire [100 -1:0] a;
assign y = &a
```

Practic însă, o poartă AND cu 100 de intrări ocupă o arie mare și va fi lentă. O implementare cu o structură de porți AND cu un număr mai mic de intrări ar fi o soluție de preferat (și ca arie și ca timp și ca bibliotecă de primitive). Se poate intui că timpul de propagare al unei porți cu 100 de intrări este mai mare decât timpul de propagare al unei porți cu două intrări.

**Timpul este o chestiune inginerească, ce ține de realizabilitate fizică, și necesită considerarea unor aspecte dincolo de cadrul matematic.**

Manufacturarea circuitelor integrate este un alt subiect de inginerie. Chiar și inginerii proiectanți trebuie să dețină minime cunoștințe despre manufacturabilitate, pentru a putea optimiza proiectarea și a ști ce se poate și ce nu se poate fabrica. Există limite tehnologice care trebuie cunoscute și considerate din fazele inițiale ale proiectării (frecvența maximă de operare a unei tehnologii, densitatea de împachetare a componentelor electronice, încapsularea, interfațarea, disiparea căldurii, puterea consumată).

Conectivitatea modulelor este o altă chestiune specifică ingineriei. Sistemele digitale actuale au devenit foarte complexe și reutilizarea unor module foarte frecventă. Pentru a fi posibilă reutilizarea, este nevoie ca modulele să aibă interfețe standard. Mai mult, interfețele ar trebui să fie simple, ușor de înțeles, de folosit și de verificat.

Toate etapele de proiectare a circuitelor integrate digitale (metodologia de proiectare) se bazează pe limbaje de descriere hardware (*HDL = Hardware Description Language*, engl.). Limbajele de descriere hardware prezintă o sintaxă specifică sistemelor hardware (paralele) și înglobează specificații

utilizate atât pentru modelarea hardware pentru sinteză (RTL - *Register Transfer Level*, engl.) cât și pentru verificarea funcțională. Verificarea sistemelor digitale nu face obiectul acestei cărți, însă, în practică, necesită un efort uneori mai mare decât faza de proiectare și modelare propriu-zisă. Pentru aceasta este nevoie de cunoșterea și utilizarea unor noțiuni avansate de Verilog.

## 4.1 Timpul în sistemele digitale

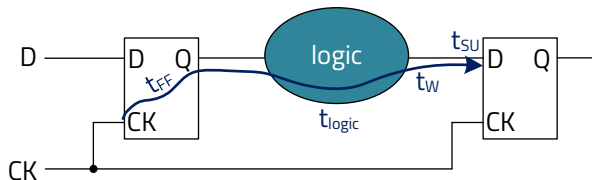
Proiectarea sistemelor digitale trebuie să considere timpul încă din faza de proiectare. Frecvența maximă de operare este determinată de tehnologia de fabricație. Semnalele de ceas nu sunt perfecte, nici ca formă, nici ca nivele de tensiune, nici ca perioadă, indiferent de modalitatea de generare. Semnalele de ceas ale diverselor bistabile nu sunt perfect sincrone datorită propagării semnalului de ceas prin circuit.

### 4.1.1 Căi de propagare, frecvența semnalului de ceas, slack

Cum se determină frecvența maximă de operare?

Un sistem digital poate fi interpretat ca fiind format dintr-o succesiune de porți logice și bistabile.

Pentru simplificare, să considerăm un sistem digital sincron, cu un singur semnal de ceas. Simplist, ne putem imagina sistemul ca fiind realizat dintr-un număr oarecare de bistabile. Între două bistabile succesive există un circuit logic combinațional. Figura 4.1 prezintă acest circuit simplificat, cu evidențierea semnalului de ceas care comandă ambele bistabile.



**Figura 4.1** Cale de propagare între două bistabile, aflate în același domeniu de ceas.

Circuitul funcționează corect dacă reușește să își propage semnalele prin logică în intervalul dintre două fronturi succesive ale semnalului de ceas.

Pentru a verifica aceasta, trebuie să ne imaginăm pe unde se propagă semnalele de la un bistabil la altul:

- Bistabilul are un timp de propagare din momentul frontului activ de ceas primit bistabil până la ieșirea acestui bistabil:  $t_{FF}$ .
- De la ieșirea primului bistabil până la intrarea următorului bistabil semnalul se propagă prin circuitul logic:  $t_{logic}$ .
- Ar trebui considerat și timpul de propagare prin firele de conexiune:  $t_W$ .
- Semnalul trebuie să ajungă la intrarea celui de-al doilea bistabil înainte de următorul front al semnalului de ceas, cu îndeplinirea condiției timpului de setup (în avans față de frontul de ceas):  $t_{SU}$ .

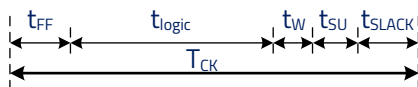
Matematic, ecuația timpului prin calea de propagare este:

$$T_{ck} > t_{FF} + t_{logic} + t_W + t_{SU}$$

Dacă punem în evidență diferența dintre timpii de propagare și perioada ceas:

$$T_{ck} = t_{FF} + t_{logic} + t_W + t_{SU} + t_{SLACK}$$

„Timpul de slack”<sup>1</sup> reprezintă timpul de așteptare din momentul când semnalul a ajuns la destinație (intrarea de date a bistabilului destinație), până când este eșantionat, în momentul următorului front activ de ceas.



**Figura 4.2** Definirea timpului de slack: cât ar mai putea crește timpul de propagare fără să depășească perioada ceas.

Putem considera că:

<sup>1</sup> „to slack” engl. A leveni. Momentul de așteptare înainte de începerea unei acțiuni.

**Timpul de slack** reprezintă diferența de timp între timpul în care semnalul ajunge la destinație, în mod real, și timpul în care ar trebui să ajungă la destinație teoretic (o perioadă de ceas). Reprezentare grafică în figura 4.2.

Adică:

$$t_{SLACK} = T_{ck} - (t_{FF} + t_{logic} + t_W + t_{SU})$$

Considerând o anumită frecvență de ceas constantă ( $T_{ck} = constant$ ), se poate deduce referitor la timpul de slack:

- Dacă timpul de slack este pozitiv ( $t_{SLACK} > 0$ ), atunci circuitul îndeplinește constrângerile de timp și există o marjă de variație a parametrilor pentru care sistemul totuși va funcționa corect.

- Dacă timpul de slack este zero ( $t_{SLACK} = 0$ ), atunci circuitul îndeplinește constrângerile de timp dar este critic, deoarece mici variații pot face ca sistemul să nu mai funcționeze.

- Dacă timpul de slack este negativ ( $t_{SLACK} < 0$ ), atunci circuitul nu va funcționa la frecvența cerută (violare de timp).

Pentru ca sistemul digital să funcționeze la o anumită frecvență trebuie ca **timpul de slack să fie pozitiv** pe toate căile de propagare.

Se poate deduce *frecvența maximă de operare* ca fiind frecvența pentru care timpul de slack este zero. Dacă  $t_{SLACK} = 0$  atunci:

$$T_{ck(MIN)} = t_{FF} + t_{logic} + t_W + t_{SU}$$

$$f_{(MAX)} = \frac{1}{T_{ck(MIN)}}$$



**Timpul de slack** — acea clipă binecuvântată dintre două bătăi de ceas, în care un semnal își poate trage sufletul înainte să fie luat la întrebări de registrul următor. E ca o pauză de cafea pentru biți grăbiți: **dacă ai destul slack, sistemul merge lin și zâmbitor; dacă nu, se iscă haos, bâlbâieli și poate chiar întoarceri în epoca analogică!**

*Text generat cu ChatGPT.*

### 4.1.2 Parametrii semnalului de ceas: skew, jitter, drift

Lumea reală este departe de cazul ideal. Un semnal de ceas ideal îl putem imagina periodic, cu perioadă constantă, factor de umplere constant, fronturi abrupte. În realitate, semnalul de ceas este foarte „stresat”. Este generat din surse neideale (dependente de tensiunea de alimentare și temperatură), trebuie să ajungă simultan la multe bistabile, trebuie să se propage prin tot chip-ul (nici sârmele nu sunt ideale), trebuie să facă față unor circuite care comută frecvent (bistabilele), deci cu un consum de curent în salturi.

Se consideră că semnalul de ceas este afectat de trei factori care determină abaterea acestuia de la parametrii ideali:

- Propagarea semnalului de ceas prin circuit între două bistabile, defazaj de ceas (*clock skew*, engl.).
- Variația perioadei semnalului de ceas în timp, datorită unor factori externi (*clock jitter*, engl.).
- Fluctuații lente ale mediului exterior, temperatură, tensiuni (*clock drift*, engl.).

**Clock skew** (defazaj de ceas) reprezintă diferența în timp între momentele în care semnalul de ceas (generat de o sursă unică) ajunge la intrarea a două bistabile distincte. Cauza defazajului de ceas este timpul de propagare al semnalului de ceas prin firele de interconexiune și încărcarea capacitivă a intrărilor de ceas ale bistabilelor, care determină atenuarea fronturilor semnalului de ceas, generând întârziere. Clock skew este cea mai frecventă cauză a violărilor timpului de hold (semnalul de ceas ajunge prea târziu, sau semnalul de date se modifică prea devreme).

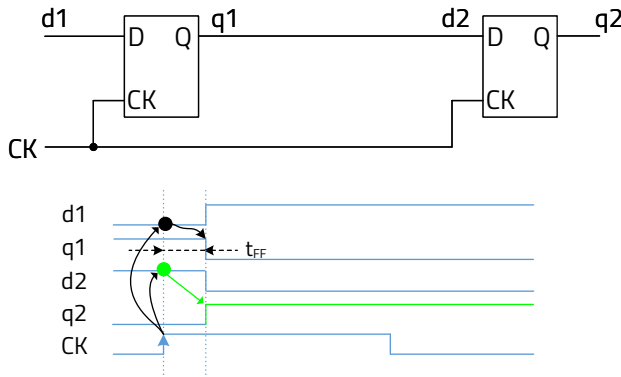
Poate neintuitiv, dar, o soluție pentru remedierea defazajului de ceas constă în introducerea unei întârzieri compensatorii pe semnalul de date.

Pentru explicarea soluțiilor de compensare a defazajului de ceas (*clock skew*) să considerăm cazul unui circuit format din două bistabile înseriate, conectate la același semnal de ceas  $CK$ , care sunt inițial în starea  $q1 = 1$  și  $q2 = 0$  iar intrarea  $d1 = 0$  în perioada premergătoare frontului de ceas. Circuitul și formele de undă sunt prezentate în figura 4.3.

Se observă că ambele bistabile își vor schimba starea simultan, pe frontul activ al semnalului de ceas, starea devenind:

$$\begin{aligned} q1^+ &= d1 = 0 \text{ și} \\ q2^+ &= d2 = q1 = 1. \end{aligned}$$

S-au notat cu  $q1^+$  și  $q2^+$  stările viitoare (după frontul de ceas) ale bistabilelor și cu  $q1$  și  $q2$  stările prezente (înainte de frontul de ceas) ale acestora.



**Figura 4.3** Circuit ideal format din două bistabile sincrone, înseriate. Se observă comutarea circuitului în stările  $q1^+ = 0$  și  $q2^+ = 1$ .

Dacă însă considerăm că al doilea bistabil primește ceasul întârziat față de primul bistabil (datorită timpului de propagare pe semnalul de ceas) primul bistabil este eșantionat primul și modifică valoarea intrării bistabilului al doilea înainte de a fi eșantionată (figura 4.4). Ca efect, starea viitoare devine:

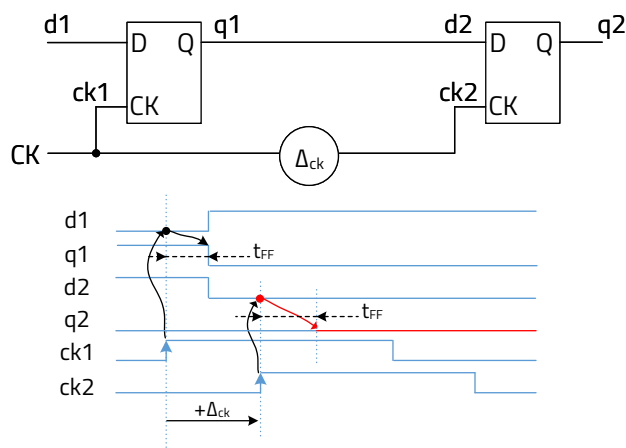
$$q1^+ = d1 = 0 \text{ și} \\ q2^+ = d2 = q1^+ = 0. \text{ Eroare!}$$

O soluție constă în introducerea unei întârzieri compensatorii pe calea de date a bistabilelor astfel încât, deși bistabilul 1 comută, ieșirea acestuia să se propage cu întârziere la intrarea celui de-al doilea bistabil (întârziere mai mare decât întârzierea semnalului de ceas). Ca efect, în momentul apariției frontului de ceas la al doilea bistabil, pe intrarea de date a acestuia va fie valoarea logică din perioada anterioară (figura 4.5). Starea viitoare devine:

$$q1^+ = d1 = 0 \text{ și} \\ q2^+ = d2 = q1 = 1. \text{ Corect!}$$

O altă soluție constă în inversarea defazajului de ceas, prin distribuirea ceasului întâi la al doilea bistabil și ulterior la primul bistabil. În acest mod, bistabilul al doilea comută primul și abia ulterior comută primul bistabil care modifică intrarea de date a celui de-al doilea (figura 4.6). Starea viitoare devine:





**Figura 4.4** Circuit format din două bistabile înseriate, cu ceasuri defazate cu clock-skew pozitiv (al doilea bistabil comută după primul bistabil). Se observă comutarea circuitului în stările  $q1^+ = 0$  (corect) și  $q2^+ = 0$  (eronat).

$q2^+ = d2 = q1 = 1$  și apoi  
 $q1^+ = d1 = 0$ . **Corect!**

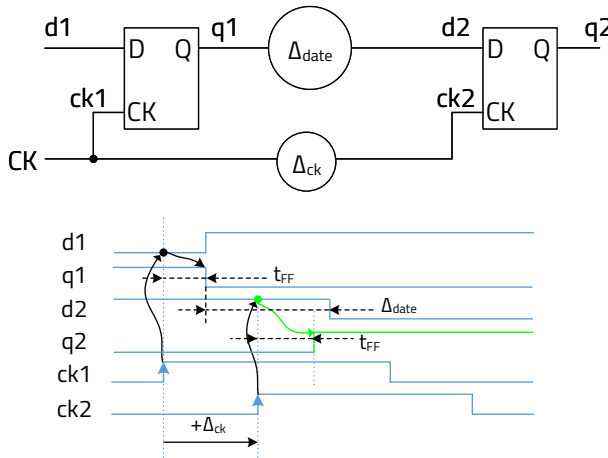
**Clock jitter** reprezintă timpul de abatere a unui front activ de ceas de la poziția lui ideală în timp. Altfel zis, perioada semnalului de ceas nu este perfect constantă, deși ar trebui să fie. Înțelegerea noțiunii de *clock jitter* este esențială în proiectarea sistemelor digitale de mare frecvență. În aceste sisteme, frecvența utilizată este la limita maximă, cu un timp de slack aproape de zero. Dacă ne referim la ecuația perioadei de ceas:

$$T_{ck(MIN)} = t_{FF} + t_{logic} + t_W + t_{SU} + t_{SLACK}$$

și considerăm  $t_{SLACK} = 0$ , condiția poate fi totuși îndeplinită dacă

$$T_{ck(MIN)} = t_{FF} + t_{logic} + t_W + t_{SU} + 0$$

Dacă însă perioada de ceas se micșorează uneori (datorită *clock jitter*) înseamnă că (în anumite condiții) nu mai este adevărată condiția de funcționare iar sistemul digital nu mai funcționează corect. În practică, acest lucru se observă prin faptul că sistemul funcționează o perioadă (o secundă) și apoi se blochează („o ia razna”). Altfel spus, o mică scădere a



**Figura 4.5** Circuit format din două bistabile înseriate, cu ceasuri defazate cu clock-skew pozitiv și defazaj compensatoriu pe semnalul de date. Se observă comutarea circuitului în stările  $q1^+ = 0$  (corect) și  $q2^+ = 1$  (corect).

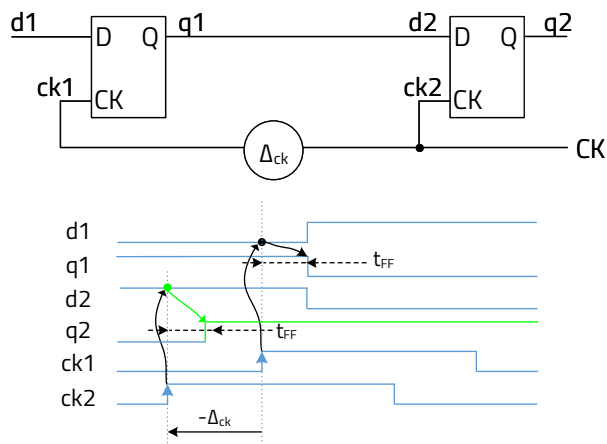
perioadei de ceas  $T_{ck} < T_{ck(MIN)}$ , în condițiile în care  $t_{SLACK} = 0$ , determină nefuncționarea sistemului secvențial deoarece frontul de ceas ajunge la bistabil mai repede decât semnalul de date ce trebuie eșantionat.

Soluțiile constau în generarea unui semnal de ceas mai precis ( $T_{ck} = \text{constant}$ ) și scăderea frecvenței de lucru pentru a admite anumite abateri ale perioadei de ceas. În proiectare, se consideră o anumită *incertitudine* asupra perioadei de ceas (*uncertainty*, engl.) exprimată în procente de perioade de ceas și calcularea timpilor de propagare luând în considerare cazul cel mai nefavorabil. **Soluție tipic inginerască!**

**Clock drift** se referă la modificarea parametrilor de ceas (frontul, perioada, factorul de umplere, nivelele de tensiune, curenții) într-un timp lung (în raport cu perioada de ceas), datorită parametrilor externi cu variație lentă: **temperatură**, **tensiuni de alimentare**, îmbătrânirea materialelor utilizate pentru realizarea circuitului integrat sau îmbătrânirea componentelor plasate pe cablaje imprimate.

Efectul este mai vizibil în cazul sistemelor cu mai mult de un semnal de ceas, în care se presupune prin proiectare anumite relații în timp (sincronizări) între semnalele de ceas.

Imaginați-vă că autobuzul planificat a ajunge în stație la oră fixă, întârzie



**Figura 4.6** Circuit format din două bistabile înseriate, cu ceasuri defazate cu clock-skew negativ. Se observă comutarea circuitului în stările  $q2^+ = 1$  (corect) și  $q1^+ = 0$  (corect).

zilnic o secundă în plus față de ziua anterioară. Nu băgați de seamă de la o zi la alta.

Dar, timp de 1 an, 365 zile, autobuzul întârzie 365 secunde, adică 6 minute. Poate că mergeți mai relaxați la stație.

Timp de 10 ani, autobuzul ajunge să întârzie o oră. Puteți dormi mai mult.

Sau, a sosit timpul să renunțați la autobuz!



### 4.1.3 Metastabilitatea bistabilelor

**Metastabilitatea** este o stare de incertitudine asupra stărilor digitale, proprie sistemelor secvențiale, cu bistabile. Sistemele digitale pot fi sincrone sau asincrone. La interfațarea unui semnal asincron cu un sistem sincron (cazul interfațării a două sisteme sincrone dar cu ceasuri diferite), este posibil ca bistabilele să nu aibă condițiile de setup și hold îndeplinite.

Motivul principal al stării de metastabilitate îl constituie neîndeplinirea condițiilor de setup și hold la intrarea de date a bistabilelor.

Adică, semnalele de date nu sunt constante în momentul eșantionării (frontul activ al semnalului de ceas) și, ca efect, bistabilul (circuit electronic) nu comută rapid într-o stare stabilă (0 sau 1), ci rămâne mai mult timp într-o stare necunoscută sau imprevizibilă, având o funcționare mai degrabă „analogică”, starea sa oscilând între cele două stări digitale.

Starea de metastabilitate apare și dacă semnalele (de ceas sau de date) au fronturi lente, datorate atenuării fronturilor pe liniile de propagare care au o componentă capacitivă. Starea de metastabilitate a unui bistabil (dacă nu se atenuază până la apariția următorului front de ceas) se poate propaga și la următoarele bistabile, persistând multe perioade de ceas în întreg sistemul.

Soluțiile de evitare a metastabilității în circuitele digitale se bazează pe minimizarea cauzelor ce determină apariția acesteia. Intrarea bistabilelor în metastabilitate este cauzată de semnale care nu sunt sincrone cu semnalul de ceas. De cele mai multe ori acestea provin de la semnale asincrone sau generate de alte semnale de ceas. Cea mai frecventă soluție de evitare a metastabilității constă în trecerea semnalelor asincrone printr-un set de mai multe bistabile înseriate și comandate de același semnal de ceas.

Semnalele de intrare asincrone cu ceasul sistemului (inclusiv semnalul de reset provenit de la un buton) trebuie sincronizate. Sincronizarea se face prin trecerea semnalului asincron printr-un set de minimum două bistabile comandate pe ceasul sincron.

Metastabilitatea poate fi diminuată odată cu scăderea frecvenței de lucru. Însă, această soluție nu este acceptată, deoarece determină un sistem mai lent.

De reținut că, metastabilitatea nu poate fi eliminată, ci doar evitată și diminuate consecințele acesteia. Vestea bună este că metastabilitatea apare la interfața dintre sistemele sincrone. Deci, nu este de interes în cazul proiectării unor blocuri digitale interne, interfațate cu semnale sincrone.

#### 4.1.4 Domenii de ceas, circuite de sincronizare

O problemă particulară în proiectarea sistemelor digitale o constituie interfațarea logicii sincronizate pe multiple semnale de ceas. Imaginați-vă un router care are ca scop comutarea pachetelor între diverse calculatoare independente. Routerul primește semnale pe multiple porturi, sincronizate pe ceasuri asincrone între ele, trebuie să preia date pe un ceas și să le transfere pe un alt port, sincronizate pe alt ceas.

Neconsiderarea aspectelor de sincronizare în cursul proiectării poate genera riscul ca simularea să fie perfectă dar sistemul real să nu funcționeze. Explicația constă în faptul că simularea nu modelează condițiile

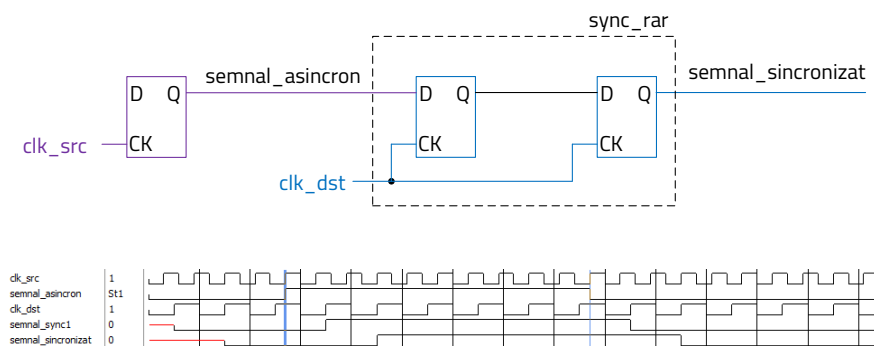
fizice de neîndeplinire a condițiilor de setup și hold ale bistabilelor ce comută pe ceasul de destinație, în cazul în care semnalul de date se modifică pe un ceas sursă asincron față de cel de destinație.

Circuitele de sincronizare trebuie să țină cont atât de frecvențele domeniilor de ceas sursă și destinație cât și de frecvența de comutare a semnalelor și de dimensiunea (în biți) a acestor semnale.

### Sincronizarea semnalelor cu comutare rară

În cazul unui semnal de un singur bit care, comută foarte rar, în comparație cu perioada de ceas (provenit de la comenzi umane sau comparatoare ale unor mărimi cu variație lentă, ca temperatură, umiditate, gaz, etc.), sincronizarea pe domeniul de destinație se face prin trecerea semnalului prin minimum două bistabile ce comută pe ceasul de destinație.

Circuitul de sincronizare și formele de undă sunt prezentate în figura 4.7. Acest circuit se recomandă a fi utilizat și pentru sincronizarea unui semnal de reset, chiar dacă este conectat pe intrarea de reset asincron a registrelor, pentru a asigura că toți biții unui registru se inițializează și au prima comutare în mod sincron, pe același front al semnalului de ceas.



**Figura 4.7** Circuit cu două bistabile înseriate pentru sincronizarea semnalelor cu variație rară. Forme de undă.

Codul Verilog care modelează circuitul de sincronizare este:

```
module sync_rar (
// domeniu de ceas sursa
input          semnal_asincron    ,

// domeniu de ceas destinație
```

```
input      clk                                , // ceas domeniu destinatie
output reg semnal_sincronizat
);

reg      semnal_sync1;

always @(posedge clk) begin
    semnal_sync1      <= semnal_asincron    ;
    semnal_sincronizat <= semnal_sync1      ;
end

endmodule // sync_rar
```

### Sincronizarea pulsurilor de frecvență mare la frecvență mică

Uneori este necesară trecerea unor pulsuri (lățime de o perioadă de ceas) de pe un domeniu de frecvență mare (perioadă mică) pe un domeniu de ceas de frecvență mică (perioadă mare). Pulsul în domeniu sursă este îngust, mai îngust decât o perioadă pe domeniul destinație, deci ar putea să nu fie eșantionat în domeniul de destinație. De aceea, pulsul trebuie întâi lătit la minimum o perioadă de ceas destinație. Evident, circuitul este dependent de raportul celor două frecvențe.

Ulterior, pulsul este eșantionat pe domeniul destinație, detectat frontul (pentru cazul în care este eșantionat de două ori pe domeniul destinație) și generat un puls de o perioadă de ceas din domeniul destinație.

### Sincronizarea pulsurilor de frecvență mică la frecvență mare

Un puls de frecvență mică este mult mai lat decât perioada domeniului de destinație. Trebuie avut în vedere că pulsul asociat pe domeniul destinație să fie determinat de frontul căzător al pulsului din domeniul sursă pentru că altfel ar putea ca pulsul la destinație să fie eșantionat înainte de eșantionarea pulsului în domeniul sursă.

Circuitul constă în eșantionarea semnalului sursă cu ceasul de destinație, sincronizarea semnalului care poate avea mai mult de o perioadă pentru evitarea metastabilității, ulterior detectarea frontului negativ și generarea unui puls de frecvență mare pe domeniul de ceas de destinație.

### Sincronizarea datelor la trecerea între două domenii de ceas

Dacă datele au mai mulți biți, nu se pot sincroniza biții individual, deoarece ar fi posibil ca datele să fie modificate la trecerea între domenii, prin faptul

că anumiți biți sunt eșantionați pe un front de ceas, iar alții pe alt front de ceas.

Soluția constă din încărcarea bus-ului de date într-un registru pe ceasul sursă, sincronizarea semnalului de încărcare (un singur bit) pe domeniul de destinație și, ulterior, preluarea datelor din registru pe ceasul de destinație. Desigur, acest circuit are o latență și nu permite tranzitarea unui flux continuu de date. Practic, la fiecare date încărcate în registru la sursă, trebuie să existe câteva perioade de pauză, timp în care datele sunt preluate corect la destinație.

În cazul unui flux continuu de date, de exemplu transferul unei imagini, dacă se asigură rate de transfer medii egale, se poate folosi o memorie FIFO dual-port, în care datele sunt încărcate pe un ceas și sunt citite pe alt ceas. În acest caz, indicatorii pot avea valori instantanee greșite, însă scopul este de a proteja validitatea datelor. De exemplu, semnalul *empty* nu se va dezactiva imediat ce datele au fost scrise în memorie ci după timpul necesar pentru a asigura că datele vor fi citite corect pe ceasul de destinație, asincron față de ceasul de scriere.

Soluția cu FIFO este mai costisitoare ca resurse hardware. Adâncimea necesară pentru memoria FIFO depinde de fluxurile de date (frecvențe și volum de date). Însă, este o soluție ce asigură corectitudinea transferului de date între două domenii de ceas asincrone.

#### 4.1.5 Structura pipeline

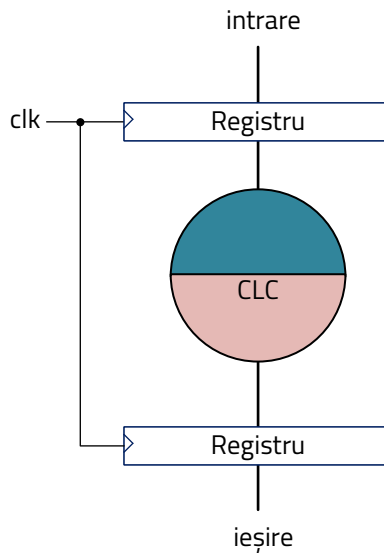
Un sistem digital poate fi considerat format din registre (bistabile) și circuite logice combinaționale (porți logice). Conceptual, o etapă de procesare combinațională între două registre se prezintă ca în figura 4.8. Frecvența maximă de operare este determinată de timpul de propagare al bistabilului (între intrarea de ceas și ieșirea de date), timpul de propagare prin circuitul combinațional (presupunem firele ideale) și timpul de setup al bistabilului, după formula:

$$f_{MAX} = \frac{1}{t_{FF} + t_{CLC} + t_{SU}}$$

Practic, frecvența maximă se obține când ieșirea registrului este conectată la intrarea următorului registru, adică fără circuit combinațional,  $t_{CLC} = 0$ .

Însă, dacă există o funcție logică (aritmetică) ce trebuie implementată, timpul de propagare prin circuitul logic va fi cel care va limita frecvența de operare.

Pentru a crește frecvența de operare, se poate modifica circuitul într-un



**Figura 4.8** Procesare de semnal cu un circuit logic combinațional.

mod similar cu cel al unei linii de asamblare (*pipeline*, engl.). Într-o structură pipeline, o operație combinațională complexă este divizată în mai multe operații elementare care se pot realiza în paralel, pe seturi de date diferite. Exact ca montarea unei mașini pe o linie de asamblare: există mai multe posturi de lucru, un muncitor realizează o singură operație iar mașinile avansează de la un post la altul până sunt finalizate. Dacă se face o poză, se constată că, la un moment dat, în fabrică există mai multe mașini în faze diferite de asamblare.

Avantajul liniei de asamblare constă în faptul că la un post de lucru pot exista doar sculele necesare la acel post, iar operatorii sunt specializați, realizând rapid doar una din etape.

În sisteme digitale, structura pipeline constă din distribuirea logicii combinaționale în mai multe etape, despărțite de seturi de registre. Ca exemplificare, în figura 4.9, logica a fost împărțită în două etape, delimitate de un registru: prima jumătate de procesare și a doua jumătate de procesare. Ceasul este același pentru toate etapele. Însă, în acest caz, frecvența maximă va fi mai mare, deoarece timpul de propagare prin logica combi-



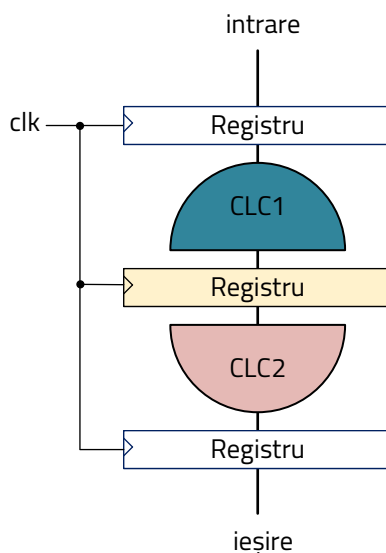
națională a unui etaj va fi mai mic:

$$fp_{MAX} = \frac{1}{t_{FF} + \max(t_{CLC1}, t_{CLC2}) + t_{SU}}$$

Să presupunem că se reușește împărțirea logicii combinaționale exact în jumătate:  $t_{CLC} = t_{CLC1} + t_{CLC2}$  și  $t_{CLC1} = t_{CLC2} = t_{CLC}/2$ .

Rezultă că frecvența structurii pipeline va fi dublă față de frecvența inițială (dacă ignorăm  $t_{FF}$  și  $t_{SU}$ ).

$$fp_{MAX} = \frac{1}{t_{FF} + t_{CLC}/2 + t_{SU}}$$



**Figura 4.9** Procesare de semnal într-o structură pipeline. Se observă registrul intermediar și divizarea circuitului logic.

Beneficiul structurii pipeline este evident în cazul existenței unui flux continuu de date de procesare. Orice oprire a structurii pipeline trebuie să se facă în totalitate (nu se poate opri doar un post de lucru, pentru că s-ar acumula semi-fabricate la acesta, iar posturile care îi urmează nu ar avea de lucru).

În concluzie, structura pipeline aduce beneficii mari (creșterea frecvenței), însă are și costuri: partea de control este mai complexă.

## 4.2 Protocoale de comunicare sincrone între module digitale

Un sistem digital actual este foarte complex pentru a putea fi gestionat de către un singur inginer. În mod uzual, inginerul responsabil de arhitectura sistemului partajează funcționalitatea întregului sistem pentru a putea delega proiectarea mai multor module către mai multe grupuri, în paralel. Fiecare grup proiectează câte un modul care, în final, se assemblează pentru a forma întregul sistem. Însă, proiectarea și verificarea făcute la nivel de modul poate cauza erori la alăturarea modulelor. De obicei, erorile sunt determinate de înțelegerea diferită a interfețelor de interconectare ale modulelor, de către ingineri.

Pentru a minimiza aceste erori, se recomandă proiectarea modulelor cu interfețe standardizate. Interfețele standardizate au multiple avantaje:

- semnale simple, clare, potrivite pentru tipul de comunicație,
- denumiri consacrate și cunoscute pentru semnale,
- reguli (protocol) consacrate și cunoscute pentru relații între semnale,
- experiențele anterioare ale echipei în utilizarea acestora, cunoașterea modului de implementare a resurselor hardware necesare,
- pot fi verificate automat prin instanțierea unor module specifice de verificare de protocol.

*Protocolul descrie semnificația semnalelor și regulile prin care două module comunică între ele date.*

### 4.2.1 Protocolul „request-acknowledge”

Protocolul „cerere-confirmare” („*request-acknowledge*”, engl.) este un protocol lent, pentru comunicarea unor informații de control sau date de volum mic, între două module, în mod bidirecțional. Protocolul acceptă cereri alăturate în timp („*back-to-back*”, engl.).

*Viteza maximă de transfer de date prin protocolul *request-acknowledge* este o dată la fiecare două tacte succesive.*

Avantajul protocolului „*request-acknowledge*” îl constituie simplitatea acestuia.

Parametrii și porturile necesare unei interfețe „*request-acknowledge*” sunt prezentate în tabelul următor.

| Parametrii interfeței „request-acknowledge” |                                                                                                           |          |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------|----------|
| Param.                                      | Descriere                                                                                                 | Domeniu  |
| $RW$                                        | Număr de biți ai datelor transferate simultan cu cererea, de la inițiator la receptor, când $req = 1$     | $\geq 0$ |
| $AW$                                        | Număr de biți ai datelor transferate simultan cu confirmarea, de la receptor la inițiator, când $ack = 1$ | $\geq 0$ |

| Porturile interfeței „request-acknowledge” |                                                                               |      |      |      |
|--------------------------------------------|-------------------------------------------------------------------------------|------|------|------|
| Port                                       | Descriere                                                                     | Dir. | Dim. | Opt. |
| $req$                                      | Cerere activă în 1, până când $ack = 1$                                       | O    | 1    |      |
| $reqdata$                                  | Date transferate de la inițiator la receptor (stabile cât timp $req = 1$ )    | O    | $RW$ | *)   |
| $ack$                                      | Confirmare, puls activ 1, numai ca răspuns la o cerere $req = 1$              | I    | 1    |      |
| $ackdata$                                  | Date transferate de la receptor la inițiator (relevante doar când $ack = 1$ ) | I    | $AW$ | *)   |

Semnalele marcate cu \*) sunt optionale. Se poate imagina o comunicare fără date, doar prin semnalele  $req$  și  $ack$ .

Comunicarea începe cu activarea de către modulul inițiator a semnalului de cerere,  $req = 1$ .

Simultan, modulul inițiator depune pe portul  $reqdata$  eventualele date ce trebuie transferate spre modulul receptor. Portul de date  $reqdata$  are lățime parametrizabilă și este opțional. Se pot imagina cereri care nu necesită date transmise. De exemplu, cererea unor date dintr-o coadă de așteptare are semnificația „*trimite datele următoare*”. În cazul în care nu sunt necesare date, parametrul  $RW = 0$  iar portul  $reqdata$  lipsește. Datele trebuie menținute stabile (fără a se modifica) pe toată durata unei cereri  $req = 1$ . Modificarea datelor când  $req = 1$ , în lipsa confirmării  $ack = 1$ , reprezintă o **eroare** de protocol.

Confirmarea  $ack = 1$  este emisă de modulul receptor doar ca urmare a unei cereri. Emiterea unei confirmări în lipsa unei cereri reprezintă o **eroare** de protocol ( $req = 0$  și  $ack = 1$ ). De asemenea, este o **eroare** de protocol și emiterea unei confirmări simultane cu cererea. Confirmarea trebuie activată cu un tact în avans față de momentul eșantionării acesteia de către inițiator. Pentru că nu se poate intui momentul când modulul inițiator va iniția o cerere, modulul receptor nu poate activa confirmarea decât cu întârziere de minimum un tact față de cerere, după eșantionarea

cererii activate,  $req = 1$ .

Rezultă că la fiecare cerere există minimum un tact în care  $req = 1$  și  $ack = 0$ . Confirmarea  $ack = 1$  are întotdeauna durată de o perioadă de tact. Deși în momentul confirmării ( $req = 1$  și  $ack = 1$ ) modulul inițiator poate decide să inițieze o nouă cerere, modulul receptor este obligat să dezactiveze confirmarea după primul tact.

Un modulul inițiator poate fi capabil de a lansa cereri succesive fără pauză între ele (cereri „back-to-back”). În cazul cererilor „back-to-back”, cererea din partea inițiatorului este continuă  $req = 1$  pentru mai multe transferuri, fără a se dezactiva  $req = 0$  între acestea. Conform protocolului, între două pulsuri succesive de confirmare  $ack = 1$  există cel puțin o perioadă de ceas în care  $ack = 0$ .

Modulul receptor are posibilitatea ca, simultan cu confirmarea (perioada de ceas când  $ack = 1$ ), să transmită către inițiator un set de date de răspuns. Datele *ackdata* trebuie eșantionate de către modulul inițiator exact în momentul confirmării ( $req = 1$  și  $ack = 1$ ). Dacă nu sunt necesare date de confirmare, parametrul  $AW = 0$  iar portul *ackdata* lipsește. Se pot imagina cereri care nu necesită date de confirmare. De exemplu, cererea pentru depunerea unor date într-o coadă de așteptare are semnificația „preia aceste date și confirmă preluarea, pentru a putea transmite altele”.

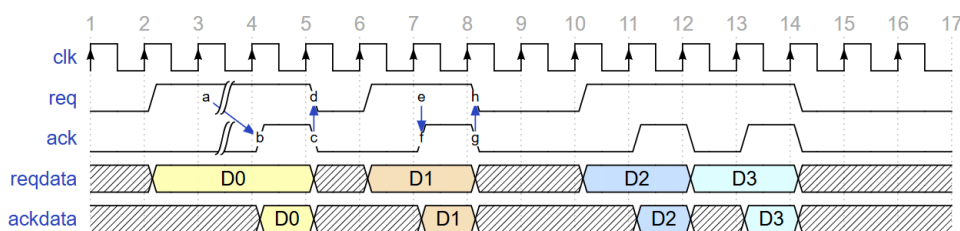
Verificarea respectării protocolului poate fi realizată de module specifice care monitorizează toate semnalele. Un modul de verificare de protocol („checker”, engl.) are doar porturi de intrare și nu atribuie valori niciunui semnal. Modulul face exclusiv verificări asupra corelației semnalelor („aserti”), conform protocolului stabilit, și lansează mesaje de notificare către proiectant.

În figura 4.10 sunt prezentate scenarii corecte pentru utilizarea protocolului „request-acknowledge”.

- La momentul 2, modulul inițiator activează cererea  $req = 1$ , simultan cu datele  $D0$ . Datele sunt menținute constante până la primirea confirmării (momentul 5). Modulul receptor observă cererea la momentul 3, o procesează și activează confirmarea la momentul 4. Între momentele 3 și 4 pot exista mai multe perioade în care modulul receptor procesează cererea. Confirmarea este eșantionată de modulul inițiator la momentul 5, moment la care decide dezactivarea cererii  $req = 0$ . Modulul receptor dezactivează confirmarea după un tact de la activarea acesteia, la momentul 5.
- La momentul 6, modulul inițiator reactivează cererea  $req = 1$ , simultan cu datele  $D1$ . În acest caz, modulul receptor are cea mai rapidă reacție: observă cererea la momentul 7 și la același moment activează confirmarea. Modulul inițiator observă confirmarea la două tacte de la

activarea cererii (la momentul 8).

- La momentul 10, modulul inițiator activează cererea  $req = 1$ , simultan cu datele  $D2$ . La momentul 12, modulul observă confirmarea și decide să inițieze o nouă cerere „back-to-back” prin menținerea  $req = 1$ , dar cu modificarea datelor  $D3$ . Cele două cereri sunt confirmate fiecare în parte.



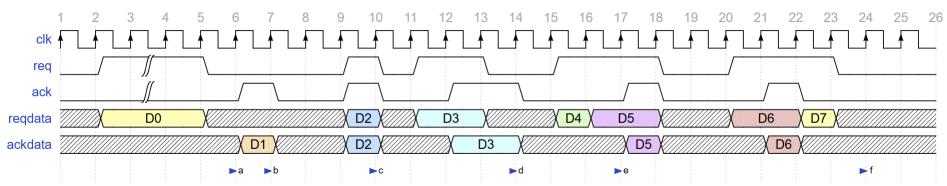
**Figura 4.10** Definirea protocolului „request-acknowledge”.

În figura 4.11 sunt prezentate erori ale protocolului „request-acknowledge”.

- La momentul 2, modulul inițiator activează cererea  $req = 1$ , simultan cu datele  $D0$ . Cererea este abandonată la momentul 5, fără a exista confirmare, generând o **eroare** de protocol la momentul 6.
- La momentul 6, modulul receptor activează confirmarea  $ack = 1$ , fără a exista o cerere, generând o **eroare** de protocol la momentul 7.
- La momentul 9, modulul receptor activează cererea  $req = 1$ , însă în același timp se activează și confirmarea  $ack = 1$ . Este o **eroare** de protocol (la momentul 10) determinată de modulul receptor deoarece în momentul 9, când a activat confirmarea, acesta nu a eșantionat cererea activă  $req = 1$ . În momentul 9 cererea este eșantionată cu valoarea  $req = 0$ .
- Cererea inițiată în momentul 11 este finalizată corect, însă modulul receptor nu dezactivează confirmarea la un tact după activare, producând la momentul 14 o **eroare** de protocol similară cu cea de la momentul 7: confirmare fără cerere (sau două confirmări consecutive).
- La momentul 15, modulul inițiator activează cererea  $req = 1$ , simultan cu datele  $D4$ , însă produce o **eroare** de protocol prin modificarea datelor în  $D5$  înainte de primi confirmarea. **Eroarea** de protocol poate fi observată la momentul 17, când datele sunt diferite de cele de la momentul 16,

deși cererea nu a fost confirmată. Modulul inițiator trebuia să mențină aceleași date până la primirea confirmării.

- La momentul 20, modulul inițiator activează cererea  $req = 1$ , simultan cu datele  $D6$ . Modulul inițiator primește corect confirmarea la momentul 22. Însă, cererea „back-to-back” începută la momentul 22 este abandonată ulterior la momentul 23, fără confirmare, cauzând la momentul 24 o eroare de protocol similară cu cea de la momentul 6.



**Figura 4.11** Erori ale protocolului „request-acknowledge”. Cu săgeți sunt marcate momentele când se depistează erori de protocol.

4.2.2 Protocolul „valid-ready”

Protocolul „valid-gata” („valid-ready”, engl.) este un protocol rapid pentru comunicarea unor informații de control sau date de volum mare între două module, în mod unidirecțional. Protocolul acceptă cereri alăturate în timp („back-to-back”, engl.).

Cea mai rapidă viteză de transmisie se obține în cazul în care transmitătorul are date disponibile de a fi transmise, iar receptorul este capabil să le primească, fără pauze. În acest caz, prin protocolul „valid-ready” pot fi transmise date în fiecare perioadă de ceas.

Parametrii și porturile necesare unei interfețe „valid-ready” sunt prezentate în tabelul următor.

| Parametrii interfeței „valid-ready” |                                                                                               |         |
|-------------------------------------|-----------------------------------------------------------------------------------------------|---------|
| Param.                              | Descriere                                                                                     | Domeniu |
| DW                                  | Număr de biți ai datelor transferate de la inițiator la receptor, când $val = 1$ și $rdy = 1$ | $> 0$   |

**Porturile interfeței „valid-ready”**

| Port        | Descriere                                                                                                                   | Dir. | Dim. |
|-------------|-----------------------------------------------------------------------------------------------------------------------------|------|------|
| <i>val</i>  | Modulul inițiator are date de transferat                                                                                    | O    | 1    |
| <i>rdy</i>  | Modulul receptor este disponibil să accepte date                                                                            | I    | 1    |
| <i>data</i> | Date transferate de la inițiator la receptor: stabile cât timp $val = 1$ , eșantionate când $val = 1$ și $rdy = 1$ simultan | O    | DW   |

Pe o interfață „valid-ready” datele se transmit într-un singur sens. Modulul emițător este cel care controlează semnalul *val*. Modulul receptor este cel care controlează semnalul *rdy*.

Modulele emițător și receptor lucrează independent, acțiunile unui modul nu sunt condiționate de starea celui alt modul. Emițătorul trebuie să activeze  $val = 1$  când are date valide, necondiționat de starea receptorului. Receptorul trebuie să activeze  $rdy = 1$  când este disponibil să primească date, necondiționat de existența datelor valide dinspre emițător.

Odată ce a activat  $val = 1$ , emițătorul trebuie să țină datele valide și constante până când acestea sunt preluate de către receptor. Preluarea datelor se face în ciclul în care  $val = 1$  și  $rdy = 1$  simultan.

Receptorul are voie să activeze și să dezactiveze *rdy* oricând. Obligația lui este de a prelua datele în cazul în care  $val = 1$  și  $rdy = 1$  simultan.

Simultan cu  $val = 1$ , modulul emițător depune pe portul *data* datele ce trebuie transferate spre modulul receptor. Portul de date *data* are lățime parametrizabilă. Datele pot fi structurate pentru a răspunde diverselor cerințe de proiectare. De exemplu, în cazul trimiterii unui flux de date ce descriu o imagine, se pot dedica anumiți biți de date pentru semnale de control (SOL/EOL - start/end of line, SOF/EOF - start/end of frame) iar anumiți biți pentru valoarea pixelului (sau a mai multor pixeli).

Datele trebuie menținute stabile (fără a se modifica) pe toată durata unei cereri  $val = 1$ . Modificarea datelor când  $val = 1$ , când emițătorul nu este gata ( $rdy = 0$ ), reprezintă o **eroare** de protocol. Inactivarea cererii  $val = 0$  în lipsa primirii datelor de către receptor ( $rdy = 1$ ) reprezintă o **eroare** de protocol. Altfel spus, *val* trebuie să comute din 1 în 0 exclusiv când  $rdy = 1$ .

Modulul receptor poate să își declare disponibilitatea de a primi date ( $rdy = 1$ ) chiar dacă emițătorul nu are date disponibile ( $val = 0$ ). Receptorul are dreptul oricând să decidă că nu mai primește date, chiar dacă a fost disponibil dar nu a primit nicio dată. Altfel spus, *rdy* poate comuta din 1 în 0 indiferent de *val*.

Protocolul „valid-ready” permite ca  $val = 1$  și  $rdy = 1$  să fie activate

simultan, nefiind condiționate. În această situație, datele sunt preluate de receptor cu o întârziere de un tact față de momentul când au fost depuse de emițător.

Rezultă că, în cazul când atât emițătorul cât și receptorul permit ( $val = 1$  și  $rdy = 1$ ), datele se pot transfera în fiecare ciclu de tact.

Un modulul emițător poate fi capabil de a lansa cereri succesive fără pauză între ele (cereri „back-to-back”). În cazul cererilor „back-to-back”, cererea din partea emițătorului este continuă ( $val = 1$ ) pentru mai multe transferuri fără a se dezactiva ( $val = 0$ ) între acestea.

Verificarea respectării protocolului poate fi realizată de module specifice care monitorizează toate semnalele. Un modul de verificare de protocol („checker”, engl.) are doar porturi de intrare și nu atribuie valori nici-unui semnal. Modulul face exclusiv verificări asupra corelației semnalelor („asertii”), conform protocolului stabilit, și lansează mesaje de notificare către proiectant.

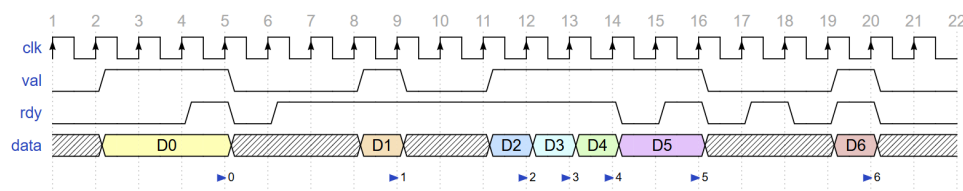
În figura 4.12 sunt prezentate scenarii corecte pentru utilizarea protocolului „valid-ready”.

- La momentul 2, modulul emițător activează  $val = 1$  simultan cu datele  $D0$ . Datele sunt menținute constante până la momentul când sunt preluate (momentul 5,  $val = 1$  și  $rdy = 1$  simultan). În momentul 5, modulul emițător decide că nu mai are date de transmis și dezactivează  $val = 0$ . Simultan (independent de emițător), modulul receptor decide la momentul 5 că nu mai poate primi alte date și dezactivează  $rdy = 0$ .
- La momentul 6, modulul receptor devine disponibil să primească noi date și activează  $rdy = 1$ . La momentul 8, emițătorul are noi date de transmis și activează  $val = 1$ . Din aceste motive, la momentul 9,  $val = 1$  și  $rdy = 1$  simultan, datele sunt transferate de la emițător la receptor.
- La momentul 9, modulul emițător decide că nu mai are alte date, dar modulul receptor rămâne în continuare disponibil să primească date.
- La momentul 11, emițătorul transmite noi date  $D2$  care vor fi preluate imediat la momentul 12. La momentele 12 și 13, modulul emițător sesizează că datele curente au fost preluate și inițiază alte cereri de transfer (cu date modificate) „back-to-back”. La momentele 12, 13, 14 se transferă date succesive, fiind ambele module disponibile ( $val = 1$  și  $rdy = 1$  simultan).
- La momentul 14, receptorul devine indisponibil ( $rdy = 0$ ). Emițătorul este nevoit să mențină datele valide până acestea sunt preluate, la momentul 16.



- La momentul 17 receptorul devine disponibil pentru un singur tact ( $rdy = 1$ ). La momentul 18 receptorul se declară indisponibil ( $rdy = 0$ ), chiar dacă nu a primit date în timp ce a fost disponibil. Este posibil ca modulul emițător să facă alte acțiuni care să nu îi mai permită să primească date. De exemplu, un controller de memorie DRAM care declanșează acțiunea de refresh devine indisponibil pentru cereri din partea modulului client.
- La momentul 19, fără a fi corelate, cele două module activează simultan  $val = 1$  și  $rdy = 1$ . În acest caz, datele sunt imediat preluate la momentul 20.

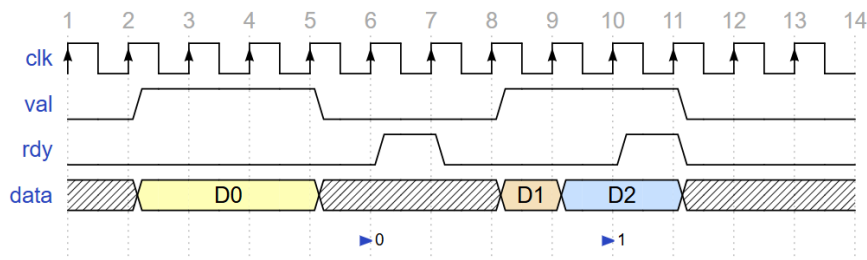
În figura 4.12 sunt marcate cu săgeți verticale momentele când au loc transferul efectiv al datelor de la emițător la receptor.



**Figura 4.12** Definirea protocolului „valid-ready”. Cu săgeți sunt marcate momentele când se transferă date ( $vld = rdy = 1$ ).

În figura 4.13 sunt prezentate erori ale protocolului „valid-ready”.

- La momentul 2, modulul emițător activează cererea  $val = 1$  simultan cu datele D0. Cererea este abandonată la momentul 5, fără ca datele să fie preluate ( $rdy = 1$ ), generând o **eroare** de protocol, la momentul 6.
- La momentul 6, modulul receptor devine disponibil și apoi, la momentul 7, devine indisponibil, fără a fi date transferate. Acest comportament este acceptabil, nefiind o eroare de protocol.
- La momentul 8, modulul emițător activează cererea  $val = 1$  cu datele D1. La momentul 9, fără ca datele D1 să fie preluate, modulul emițător modifică datele în D2. Aceasta este o **eroare** de protocol a modulului emițător (momentul 10), deoarece a modificat datele fără ca receptorul să le primească ( $rdy = 1$ ).



**Figura 4.13** Erori ale protocolului „valid-ready”. Cu săgeți sunt marcate momentele când se depistează erori de protocol.

### 4.3 Tehnologia circuitelor integrate digitale

Acest capitol este despre proiectarea și manufacturarea circuitelor integrate. Scopul acestui capitol nu este de a fi exhaustiv, ci mai degrabă de a prezenta noțiunile necesare la nivel de proiectant de sisteme digitale, pentru a înțelege constrângerile impuse de manufacturabilitate. Se vor prezenta exclusiv generalități despre tehnologia actuală **CMOS** („Complementary Metal-Oxide-Semiconductor”, engl.). Inginerii proiectanți trebuie să știe ce se poate și ce nu se poate realiza în această tehnologie. Acest capitol nu este pentru inginerii de manufacturare de circuite integrate.

În împărăția siliciului, CMOS-ul (Complementary Metal-Oxide-Semiconductor, dar să nu speriem norodul cu vorbe mari) este ca o pereche de soacră și socru digital: unul trage când celălalt stă, și amândoi se pornesc doar când e musai!

Această vrăjitorie face circuitele să consume puțină putere — adică exact cât o pisică adormită pe calorifer — și să funcționeze fără să scoată fum, ba chiar cu grație.

Adevărul este că CMOS-ul a ajuns în toate: din ceasul deșteptător până-n buzunarul cu smartphone, fiind atât de eficient încât și **electronii îi bat din palme... cu sarcină negativă**, firește!

*Text generat cu ChatGPT.*

### 4.3.1 Metodologia de proiectare a circuitelor integrate digitale

Metodologia de proiectare a circuitelor integrate digitale, de la idee până la predarea datelor pentru manufacturare, este reprezentată grafic în figura 4.14.

Metodologia începe cu un document de specificații în care sunt detaliate:

- Arhitectura sistemului;
- Interfețele și parametrii modulelor constitutive;
- Funcționalitatea și caracteristicile modulelor componente.

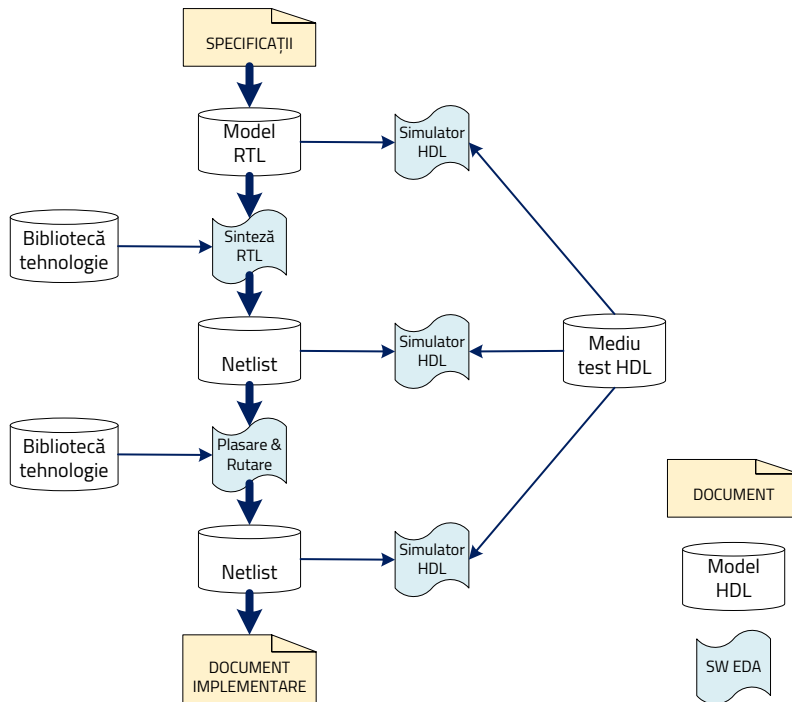
Pe baza documentului de specificații, inginerii proiectanți scriu un model HDL la nivel RTL (*Register Transfer Level*, engl.), adică cu specificații din care programul software de sinteză să poată deduce structura de porți logice și bistabile asociată. Modelul RTL trebuie depănat și verificat funcțional printr-o simulare împreună cu un model HDL al sistemului exterior. Simulatorul HDL simulează modulul de testat (*DUT = Device Under Test*, engl.) instanțiat într-un mediu de test. Mediul de test poate fi foarte complex, uneori dezvoltarea acestuia necesitând timp mai îndelungat decât scrierea modelului pentru sistemul hardware. Modelul de test poate fi re-utilizat pentru a simula modelul de hardware, în etapele următoare de procesare.

Urmează etapa sintezei de hardware, realizată cu un software specializat (*EDA = Electronic Design Automation*, engl.) care preia modelul RTL comportamental și îl convertește într-un model structural cu primitive dintr-o bibliotecă specifică unei tehnologii. Biblioteca de tehnologie este dezvoltată de producătorul de tehnologie și prezentată ca un model HDL. Sinteza ține cont de constrângerile de proiectare și generează un netlist (*NETwork LISTing*, engl.) care descrie structura unui circuit electronic compus din primitivele existente în biblioteca de tehnologie (porți logice, bistabile, etc.). Netlist-ul trebuie reverificat prin simulare alături de mediul de testare pentru a valida corectitudinea sintezei. Simularea la nivel RTL nu este suficientă, deoarece este posibil ca, după sinteză, netlist-ul să aibă un comportament diferit, datorită limitării programelor de sinteză în înțelegerea tuturor specificațiilor HDL utilizate de proiectant.

Altfel spus, programele EDA sunt ca oamenii: uneori ascultă, dar nu înțeleg.

Inginerii proiectați de circuite integrate numesc etapele descrise până acum ca fiind „front-end”.

Ceea ce urmează în metodologie este cunoscută ca fiind „back-end”.



**Figura 4.14** Metodologia de proiectare a circuitelor integrate digitale.

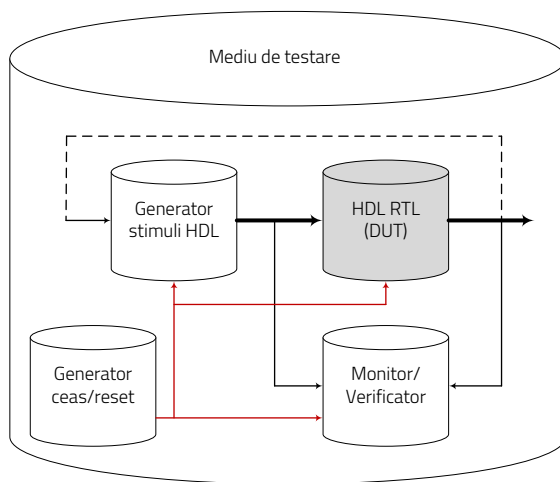
Dacă netlist-ul este corect, se poate trece la etapa următoare de implementare, similară cu cea a realizării unui cablaj imprimat, însă pe pastila de siliciu. Primitivele din netlist (porți și bistabile) descrise la nivel fizic într-o bibliotecă de tehnologie, sunt plasate pe pastila de siliciu (în plan) și apoi rutate (interconectate cu fire) pe baza unor constrângeri. Rezultă tot un netlist, dar mai complex, care conține și informații despre timp (întârzieri prin porți logice și fire). Netlist-ul acesta poate fi simulat atât pentru verificare funcțională cât și pentru o primă estimare a frecvenței maxime de funcționare.

La final, se generează un document de implementare pentru a fi utilizat de către inginerii care preiau netlistul pentru manufacturare.

Metodologia a fost prezentată foarte simplificat. În realitate, există o mulțime de etape și verificări necesare pentru a garanta o funcționare corectă a circuitului integrat proiectat.

Mediul de test simulat cu programe de simulare HDL este reprezentat

grafic în figura 4.15. Mediul de test nu are interfețe cu lumea exterioară și instanțiază modulul de testat (DUT) care se verifică în vederea implementării în hardware. Alături de acesta pot fi instanțiate, sau modelate simplificat, generatoare de ceas și de reset, și generatoare de stimuli. Opțional, pot fi descrise specificații care monitorizează comportamentul DUT sau verifică îndeplinirea protocoalelor de interfațare. Generatorul de stimuli poate fi modelat ca o succesiune de valori generate la intrare, dar poate fi și adaptat la răspunsul modelului de testat.



**Figura 4.15** Mediul de test HDL.

### Concluzie

Metodologia de proiectare a sistemelor digitale integrate este de tipul „top-down” (de la nivel înalt, abstract, spre detalii concrete) și conține următoarele 3 etape:

- **Descriere** = Scrierea unui model HDL RTL pentru sistemul ce va fi implementat în hardware.
- **Sinteză** = Conversia modelului HDL comportamental în netlist cu componente din biblioteca de tehnologie, cu ajutorul unor programe EDA specializate.
- **Implementare** = Plasarea și rutarea pe chip-ul de siliciu a circuitului reprezentat de netlist.

### 4.3.2 Fabricarea circuitelor integrate

Fabricarea circuitelor integrate reprezintă un proces foarte complex. Pe scurt, procesul de fabricare constă din următoarele etape:

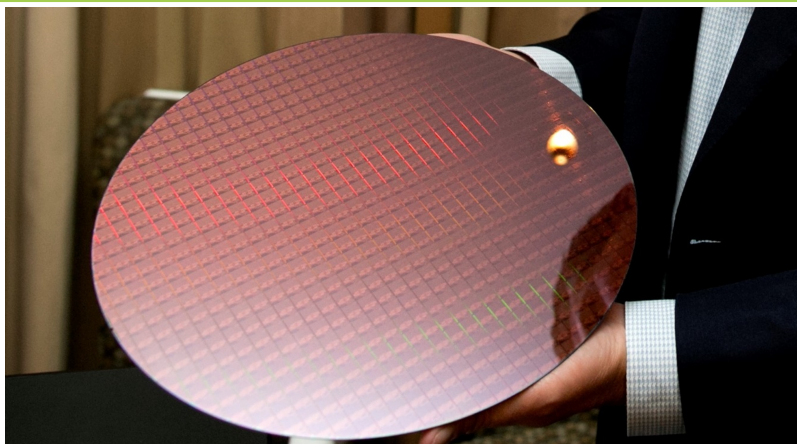
- Obținerea lingoului de siliciu prin [metoda Czochralski](#). Metoda constă din topirea siliciului și tragerea acestuia din topitură, prin rotire, sub forma unui lingou cilindric format din siliciu unicristalin (figura 4.16).



**Figura 4.16** Lingoul de siliciu. În partea dreaptă.



- Realizarea unui disc (*wafer*, engl.) de siliciu prin tăierea lingoului de siliciu în felii (figura 4.17).



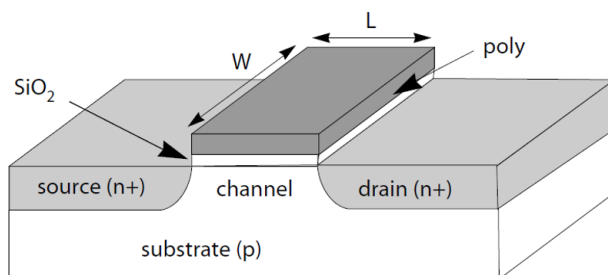
**Figura 4.17** Wafer de siliciu.

- Modificarea conductivității **waferului de siliciu** pentru obținerea circuitelor integrate proiectate. Tăierea wafer-ului în pastile dreptunghiulare (*die*, engl.).
- **Încapsularea unei pastile de siliciu** și testarea circuitului integrat.

### 4.3.3 Tehnologia CMOS

Tehnologia CMOS (*Complementary Metal-Oxide-Semiconductor*, engl.) se bazează pe structuri de tranzistoare MOS de ambele tipuri: nMOS și pMOS. Această tehnologie a fost și este utilizată datorită multiplelor avantaje oferite atât proiectării cât și realizării circuitelor integrate digitale.

Structura unui tranzistor nMOS este prezentată în figura 4.18. Se ob-



**Figura 4.18** Tranzistorul nMOS (vedere 3D).

servă suprapunerea celor 3 zone care realizează un canal prin care trecerea electronilor este controlată de tensiunile aplicate la borne:

- **Metal** - realizat sub forma de poli-siliciu puternic dopat (în figură *poly*). Acesta este electrodul de poartă (*Gate*, engl.).
- **Oxid** - realizat din oxid de siliciu (în figură  $SiO_2$ ).
- **Semiconductor** - canalul (în figură *channel*) în care se produce inversia de purtători, ca efect al aplicării tensiunii pe electrodul de poartă (metal).

Prin canal, electronii circulă ca urmare a diferenței de potențial între zonele dopate  $n^+$ , denumire *Sursă* (*source*, engl.) și *Drenă* (*drain*, engl.).

Structura tranzistorului nMOS poate fi echivalată cu un întrerupător, între sursă și drenă, controlat de tensiunea aplicată între poartă și sursă (considerată ca potențial de referință).

## Porți logice

În tehnologie CMOS, porțile logice au o structură fixă, ceea ce conferă un avantaj pentru programele software EDA. Structura (figura 4.19) prezintă două circuite realizate cu tranzistoare MOS utilizate pe post de comutatoare în conexiune serie sau paralel: o structură cu tranzistoare nMOS în partea de jos și o structură cu tranzistoare pMOS în partea de sus. Intrările se aplică simetric pe cele două tipuri de tranzistoare.

Structura CMOS a celui mai simplu circuit logic, inversorul, este prezentată în figura 4.19.

Figura 4.20 prezintă structurile porților NAND și NOR cu câte 2 intrări, realizate în tehnologie CMOS. Se observă conectarea serie și paralel pe cele două secțiuni: nMOS și pMOS.

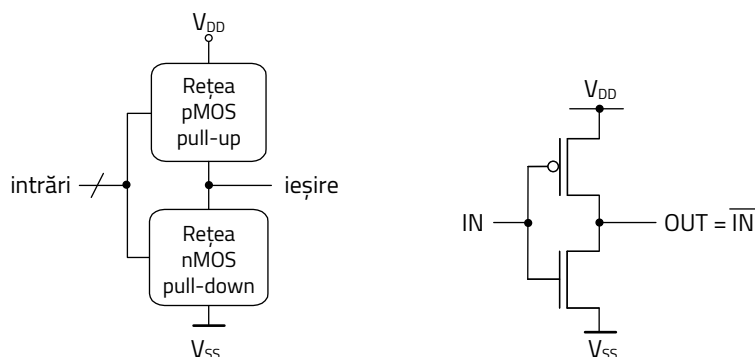
## Poartă de transmisiune

Prin conectarea unui tranzistor nMOS și a unui pMOS ca în figura 4.21 se obține o structură echivalentă cu un întrerupător aproape ideal. Structura, denumită *poartă de transmisiune* (*transmission gate*, engl.) este utilizată pentru realizarea simplificată a diverselor circuite logice.

De exemplu, multiplexorul 2:1, caracterizat de funcția logică

$$Y = S \cdot I_1 + \overline{S} \cdot I_0$$





**Figura 4.19** Structura porților logice, în tehnologie CMOS.  
Structura inversorului (poarta NOT), în tehnologie CMOS.

ar necesita mai multe tranzistoare în implementarea clasică, decât o implementare cu porți de transmisiune, prezentată în figura 4.21.

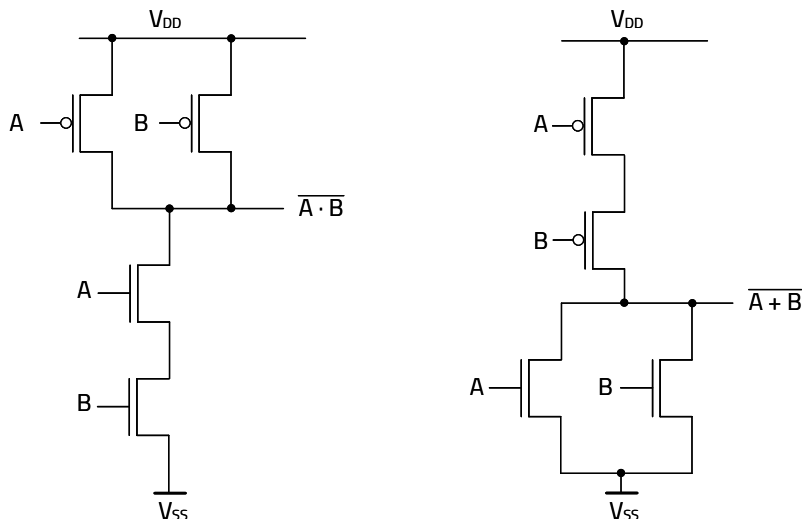
Se observă că cele două porți de transmisiune sunt deschise sau blocate în contratimp.

Dacă  $S = 1$ , poarta de sus este deschisă, iar cea de jos este blocată, rezultând  $Y = I_1$ .

Dacă  $S = 0$ , poarta de jos este deschisă, iar cea de sus este blocată, rezultând  $Y = I_0$ .

## Bistabile

Bistabilul D se realizează în tehnologie CMOS, ca o succesiune de două circuite latch conectate cu ceasurile complementate. Fiecare circuit latch este format dintr-o buclă de două inversoare, comandată cu un multiplexor 2:1 realizat cu porți de transmisiune. Figura 4.22 prezintă structura unui bistabil D în tehnologie CMOS.



**Figura 4.20** Structura porților NAND și NOR cu 2 intrări, în tehnologie CMOS.



### Viitorul tehnologiei CMOS?

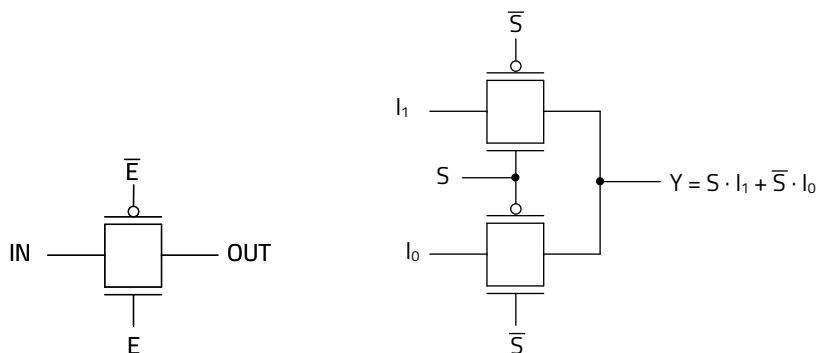
Ei bine, zic înțelepții laboratoarelor că mai are ceva suflu în pieptul ei minuscul de siliciu, dar nu va mai fi tânără-n veac!

Cu fiecare nod de fabricație tot mai mic, CMOS-ul se zbate ca un domn de onoare în costum prea strâmt — elegant, dar din ce în ce mai greu de mișcat fără scânteii.

Deja se zvonesc prin târgurile științei despre noi urmași: nanotuburi, tranzistori 3D, poate chiar logică cu spin în loc de curent.

Dar până la marea trecere, **CMOS-ul tot va mai domni**, cu consumul său smerit și viteza sprintenă, ca un bătrân samurai care, deși gârbovit, încă taie semnalele cu precizie milimetrică.

*Text generat cu ChatGPT.*



**Figura 4.21** Structura porții de transmisiune, în tehnologie CMOS.

Structura multiplexorului 2:1 realizat cu porți de transmisiune, în tehnologie CMOS.

## 4.4 Noțiuni avansate de Verilog

Această secțiune conține noțiuni avansate de Verilog, necesare pentru modelarea sistemelor complexe, actuale, atât pentru modelare în vederea implementării în hardware cât și pentru implementarea mediilor de verificare.

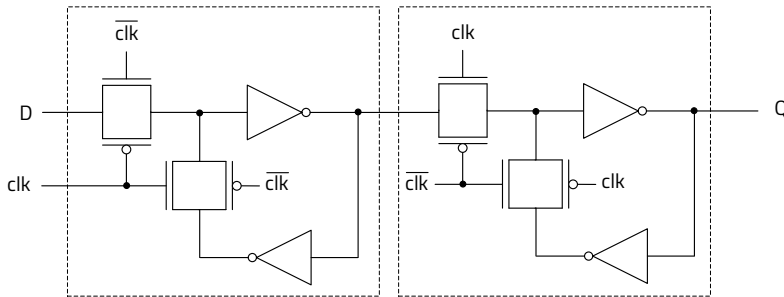
În Verilog se modelează atât sistemul digital ce va fi implementat în hardware cât și mediul de verificare.

**Modelul sistemului digital** ce va fi implementat în hardware urmează o metodologie de „sinteză” pe baza unor programe software specifice (*EDA - Electronic Design Automation*). Metricile de calitate pentru modelul RTL sunt dimensiunea circuitului generat prin sinteză și frecvența maximă de operare.

**Modelul mediului de verificare** are ca scop certificarea conformității modelului hardware cu specificațiile de proiectare. Metrica de calitate a mediului de verificare o constituie procentul în care sunt verificate toate specificațiile, toate funcționalitățile și toate scenariile de funcționare.

Modelarea sistemului digital poate fi realizată atât comportamental cât și structural, prin instanțierea unor module componente.

Modelele Verilog pot fi dezvoltate pe diferite nivele de abstractizare:



**Figura 4.22** Structura bistabilului D, în tehnologie CMOS.

- **Modelare algoritmică:** un model care descrie algoritmul unei funcții de transfer a unui circuit digital, prin construcții de nivel înalt ale limbajului, fără corespondență directă cu resursele hardware și fără a modela detaliat temporizarea circuitului. Acest nivel este utilizat pentru generarea rapidă a unui model care descrie comportamentul, fără a fi utilizat direct pentru sinteza de hardware, ci pentru depanare și verificare (model de referință).
- **Modelare RTL** (*Register Transfer Level*, engl.): un model care descrie transferul datelor între registre și modul de procesare a acestora. Modelele sistemelor ce se vor implementa hardware sunt descrise de către proiectanți la acest nivel de detaliere.
- **Modelare la nivel de poartă logică:** un model care descrie porțile logice și conexiunile dintre porți logice. De obicei, modelele de acest tip sunt generate automat de software EDA și sunt utilizate pentru a simula funcționarea unui circuit deja implementat, pentru verificare.
- **Modelarea la nivel de comutare:** un model care descrie circuitul logic realizat cu tranzistoare, similar cu modelele SPICE.

**Tipurile de date** utilizate frecvent pentru modelarea RTL în Verilog sunt:

- **Tipul net**, introdus prin cuvântul rezervat **wire**, modelează conexiunile fizice dintre elementele structurale. Un obiect de acest tip se comportă similar cu o sârmă, a cărei valoare este dată de singura sursă aplicată. Valoarea unei date de acest tip este determinată de sursa sa, care poate fi o specificație de atribuire continuă `assign` sau o instanțiere de componentă (printr-un port de ieșire). În lipsa unei surse pentru semnal, data de tip `wire` are valoarea implicită `'bz`. În cazul mai multor surse cu

valori complementare, semnalul de tip `wire` are valoarea nedeterminată 'bx, semnalizând proiectantului o eroare de modelare RTL. O sârmă declarată de tip `wire` are valoarea continuu atribuită de către sursa sa.

- **Tipul register**, introdus prin cuvântul rezervat `reg`, modelează un element abstract de stocare a datelor. Valorile obiectelor de acest tip pot fi atribuite cu specificații `always`. Valoarea implicită a acestui tip de date este 'bx. O sârmă declarată de tip `reg` își păstrează valoarea până la atribuirea unei noi valori (adică are o „remanență” a valorii după dispariția sursei semnalului). Este comportamentul similar cu ieșirea unui bistabil care își modifică valoarea doar în momentele frontului activ al semnalului de ceas și își păstrează valoarea între două fronturi succesive.

#### 4.4.1 Modelarea comportamentală RTL

Modelarea comportamentului sistemelor digitale pentru sinteză se realizează cu specificații `assign` și `always`,

##### Specificațiile `assign`

Specificațiile `assign` sunt utilizate exclusiv pentru modelarea circuitelor combinaționale. Semnalele atribuite cu `assign` sunt declarate de tip `wire`.

De remarcat că Verilog are o regulă prin care semnalele nedecarate explicit sunt considerate a fi de tip `wire` cu dimensiune de 1 bit.

În practică, nu se recomandă utilizarea acestei reguli, ci declararea explicită a tuturor semnalelor utilizate pentru modelarea comportamentului.

```
wire a, b, c, d, e;
wire y          ; // y(a, b)
wire z          ; // z(c, y)
wire w          ; // w(z, d, e)

assign y = a & b;      // atribuire concurenta, y = a and b
assign z = c | y;      // atribuire concurenta, z = c or y
assign w = z ? d : e;  // atribuire concurenta, multiplexor
```

În cazul unor funcții logice (combinaționale) mai complexe, modelarea cu specificații `assign` se poate dovedi a fi greoaie.

## Specificațiile **always combinaționale**

Se pot modela circuite combinaționale cu specificații `always` care pot încapsula specificații `if` sau `case`. În acest caz, semnalul trebuie declarat ca fiind de tip `reg`. Este obligatoriu ca, în cazul modelării unui circuit combinațional, specificația `always` să cuprindă în lista de senzitivități toate semnalele ce determină re-evaluarea ieșirii. O alternativă la enumerarea semnalelor într-o listă de semnale despărțite de virgule este de a se utiliza simbolul specific `@(*)`.

Următorul cod modelează o unitate logico-aritmetică (combinațională) pe baza unei specificații de selecție `case`. Atenție la obligativitatea clauzei `default` pentru a asigura atribuirea valorii către ieșire în toate combinațiile semnalelor de intrare.

```

wire [3:0]    i1 ; // operatori
wire [3:0]    i2 ;
wire [1:0]    sel ; // expresie de selecție
reg  [3:0]    u ; // circuit combinațional u(sel, i1, i2)

always @(*)    // echivalent cu: always @(sel or i1 or i2)
  case(sel)
    2'b01: u = i1 - i2; // diferența,    dacă sel = 01
    2'b10: u = i1 & i2; // AND pe biti,  dacă sel = 10
    2'b11: u = i1 | i2; // OR pe biti,   dacă sel = 11
    default: u = i1 + i2; // suma,        dacă sel nu are nicio
                        //                valoare menționată
                        //                anterior (00)
  endcase

```

Există posibilitatea de a descrie comportamentul combinațional într-o manieră „*algoritmică*”, cu specificații care descriu comportamentul în stil secvențial.

Ca exemplu, se consideră un circuit combinațional care generează numerele lui Fibonacci. Circuitul combinațional primește la intrare un număr între 0 și 31 (reprezentat pe 5 biți). Pe baza acestui index, circuitul generează combinațional numărul cu indexul corespunzător din șirul lui Fibonacci.

Matematic, numerele Fibonacci sunt definite prin următoarea relație de recurență:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}, \text{ pentru } i \geq 2.$$

Astfel, fiecare număr Fibonacci (vezi tabelul 4.1) este suma celor două numere Fibonacci anterioare, rezultând secvența:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Conceptual, este un circuit logic combinațional cu 5 intrări și 21 de ieșiri. Numărul de biți de ieșire (21) a fost determinat ca fiind numărul minim de biți necesari pentru reprezentarea celei mai mari valori:

$$F_{31} = 1.346.269 < 2^{21} = 2.097.152$$

Însă, funcția de transfer nu este simplu de exprimat cu operatori Verilog.

Codul Verilog care modelează un circuit combinațional ce generează numărul Fibonacci asociat indexului de la intrare prezintă o specificație `always` ce conține o specificație de selecție `case`:

- Dacă  $i = 0$ ,  $F_0 = 0$ ,
- Dacă  $i = 1$ ,  $F_1 = 1$ ,
- Altfel, pentru  $i \geq 2$ ,  $F_i = F_{i-1} + F_{i-2}$ .

Modelarea relației recurente se face cu atribuiri blocante, cu simbol `=`. Se remarcă faptul că valoarea finală *valoare* se determină după un șir de atribuiri pentru rezultatele parțiale (la același timp al simulării), de la 1 până la indexul final `fib_idx`.

```
module fibonacci (
input      [5  -1:0] fib_idx ,    // index al termenului
output reg  [21 -1:0] valoare     // valoare termen
);

reg  [6  -1:0] i          ;
reg  [21 -1:0] valoare0;    // prima valoare
reg  [21 -1:0] valoare1;    // a doua valoare

always @(*)
case(fib_idx)
'd0: valoare = 'd0;
'd1: valoare = 'd1;
default: begin
    valoare0 = 'd0;
    valoare1 = 'd1;
    valoare  = 'd1;
    for (i=2; i<=fib_idx; i=i+1) begin
        valoare = valoare1 + valoare0;
        valoare0 = valoare1;
        valoare1 = valoare;
    end
end
endcase

endmodule // fibonacci
```

| Index | Valoare [baza 10] | Valoare [baza 2]           |
|-------|-------------------|----------------------------|
| 0     | 0                 | 0_0000_0000_0000_0000_0000 |
| 1     | 1                 | 0_0000_0000_0000_0000_0001 |
| 2     | 1                 | 0_0000_0000_0000_0000_0001 |
| 3     | 2                 | 0_0000_0000_0000_0000_0010 |
| 4     | 3                 | 0_0000_0000_0000_0000_0011 |
| 5     | 5                 | 0_0000_0000_0000_0000_0101 |
| 6     | 8                 | 0_0000_0000_0000_0000_1000 |
| 7     | 13                | 0_0000_0000_0000_0000_1101 |
| 8     | 21                | 0_0000_0000_0000_0001_0101 |
| 9     | 34                | 0_0000_0000_0000_0010_0010 |
| 10    | 55                | 0_0000_0000_0000_0011_0111 |
| 11    | 89                | 0_0000_0000_0000_0101_1001 |
| 12    | 144               | 0_0000_0000_0000_1001_0000 |
| 13    | 233               | 0_0000_0000_0000_1110_1001 |
| 14    | 377               | 0_0000_0000_0001_0111_1001 |
| 15    | 610               | 0_0000_0000_0010_0110_0010 |
| 16    | 987               | 0_0000_0000_0011_1101_1011 |
| 17    | 1597              | 0_0000_0000_0110_0011_1101 |
| 18    | 2584              | 0_0000_0000_1010_0001_1000 |
| 19    | 4181              | 0_0000_0001_0000_0101_0101 |
| 20    | 6765              | 0_0000_0001_1010_0110_1101 |
| 21    | 10946             | 0_0000_0010_1010_1100_0010 |
| 22    | 17711             | 0_0000_0100_0101_0010_1111 |
| 23    | 28657             | 0_0000_0110_1111_1111_0001 |
| 24    | 46368             | 0_0000_1011_0101_0010_0000 |
| 25    | 75025             | 0_0001_0010_0101_0001_0001 |
| 26    | 121393            | 0_0001_1101_1010_0011_0001 |
| 27    | 196418            | 0_0010_1111_1111_0100_0010 |
| 28    | 317811            | 0_0100_1101_1001_0111_0011 |
| 29    | 514229            | 0_0111_1101_1000_1011_0101 |
| 30    | 832040            | 0_1100_1011_0010_0010_1000 |
| 31    | 1346269           | 1_0100_1000_1010_1101_1101 |

Tabelul 4.1

Numerele lui Fibonacci.



În cazul modelării circuitelor combinaționale cu specificații `always`, se recomandă pentru atribuirea semnalelor utilizarea simbolului `=`.

În cazul modelării circuitelor combinaționale, semnalele se comportă similar cu variabile din limbajul C: își modifică valoarea imediat ce i-a fost atribuită o nouă valoare.

### Specificațiile **`always`** secvențiale

Modelarea circuitelor secvențiale se face întotdeauna cu specificații `always`, semnalele fiind declarate de tip `reg`. În lista de sensibilități a specificației `always` trebuie să apară unul din cuvintele rezervate `posedge` sau `negedge` care preced numele semnalului de ceas. Opțional, poate apărea și sintaxa ce modelează un semnal de reset asincron.

Diferența între semnalul de ceas și cel de reset asincron este că semnalul de ceas nu apare în corpul specificației `always` dar semnalul de reset apare în condiția asociată specificației secvențiale `if` (evaluată prima).

Atribuirile de semnale declarate de tip `reg` pot fi de două tipuri: *blocante* și *neblocante*.

**Atribuirile blocante** utilizează simbolului de atribuire `=` pentru a modela atribuirile executate una după alta, la același moment de timp, în cadrul unui bloc de specificații secvențiale.

**Atribuirile neblocante** utilizează simbolului de atribuire `<=` pentru a modela semnale planificate a se re-evalua, dar fără a bloca execuția specificațiilor următoare, care ar putea planifica modificarea altor semnale la același moment de timp.

Ca exemplu, se consideră cazul unui registru format din mai multe bistabile. Fiecare bistabil este independent de celelalte (modelele lor rulează în paralel), dar poate fi modelat într-o secvență de atribuirii la același moment de timp.

Simplu, se poate considera că semnalele atribuite cu simbolul blocant `=` „blochează” atribuirile ce urmează, valoarea semnalelor fiind actualizată în momentul execuției acestora (ca o variabilă C). În schimb, semnalele atribuite cu simbolul neblocant `<=` „nu blochează” atribuirile ce urmează, valoarea semnalelor fiind cea de dinaintea momentului execuției acestora (ca ieșirile a două bistabile).

De exemplu, în continuare este prezentată modelarea unui circuit combinațional care realizează înmulțirea a două numere întregi pozitive după

un algoritm de adunare repetată. Se observă faptul că este utilizată o atribuire blocantă, prin care ieșirea *rezultat* ia mai multe valori secvențial, dar la același timp de acumulare. Practic, în cazul *op1=3* și *op2=4*, variabilei *rezultat* i se atribuie succesiv valorile intermediare 0, 4, 8, 12, însă valoarea finală expusă de circuitul combinațional este doar ultima, *rezultat=12*.

```
module multiplier (
input      [8  -1:0] op1      ,
input      [8  -1:0] op2      ,
output reg [16 -1:0] rezultat
);

integer i;

always @(*) begin
    rezultat = 'd0;
    for (i=0; i<op1; i=i+1)
        rezultat = rezultat + op2;
end

endmodule // multiplier
```

În cazul modelării circuitelor secvențiale, se recomandă pentru atribuirea semnalelor utilizarea simbolului `<=`, specific atribuirilor neblocante. Semnalele se comportă similar cu ieșirile unor bistabile distincte: își modifică valoarea simultan, în momentul determinat de frontul semnalului de ceas.

## Parametrii modulelor

Opțional, modulul poate avea *parametrii*, definiți înainte de porturi, imediat după numele modulului, între paranteze rotunde precedate de simbolul `#`.

Fiecare parametru, introdus prin cuvântul rezervat `parameter`, are un nume și o valoare implicită, constantă pe toată durata simulării, și care poate determina dimensiunea porturilor sau a semnalelor interioare modulului.

```
module numerator #(
parameter NO_BITS      = 1'b8      , // numar de biti
parameter MODULO_NUM   = 'd35      // numar de stari
)
```

Modulele RTL conțin o listă de porturi, enumerate cu informații despre nume, direcție și dimensiune, între paranteze rotunde. Porturile sunt

conexiunile modulului cu lumea exterioară. Fiecare port este introdus printr-un cuvânt rezervat care desemnează direcția acestuia (input, output sau inout) urmat de dimensiunea portului și numele acestuia. Dimensiunile portului se exprimă prin indexul celui mai semnificativ bit și indexul celui mai puțin semnificativ bit, între paranteze drepte. Dimensiunea unui port poate fi exprimată pe baza parametrilor.

```
module numarator #(
  parameter NO_BITS      = 1'b8    , // numar de biti
  parameter MODULO_NUM   = 'd35    // numar de stari
)
(
  input          clk        , // ceas
  input          rst_n      , // reset asincron activ 0
  // semnale de control
  input          clr        , // aducere la valoarea initiala
  input          count      , // numara

  output reg [NO_BITS -1:0] data_out // iesire numarator
);

// valoare maxima, pe baza MODULO_NUM
localparam MAX_VAL = MODULO_NUM - 1 ;

// registru numarator cu presetare
always @(posedge clk or negedge rst_n)
  if (~rst_n)          data_out <= {NO_BITS{1'b0}}; else
  if (clr)             data_out <= {NO_BITS{1'b0}}; else
  if (count)
    if (data_out == MAX_VAL) data_out <= {NO_BITS{1'b0}}; else
                          data_out + 1;

endmodule // numarator
```

Valoarea implicită a unui parametru poate fi modificată la instanțierea modulului, astfel încât pot exista într-un modul instanțieri ale aceluiași modul cu parametrii diferiți, ca în exemplul următor: instanță de numărător modulo 100 (pe 7 biți) și instanță de numărător modulo 1000 (pe 10 biți).

```
numarator #(
  .NO_BITS      (7          ),
  .MODULO_NUM   (100        )
) numarator_mod100 (
  .clk          (clk        ),
  .rst_n        (rst_n      ),
  .clr          (clr_100    ),
  .count        (count_100  ),
  .data_out     (data_out_100)
```

```
);

numarator #(
    .NO_BITS      (10          ),
    .MODULO_NUM   (1000       )
) numarator_mod1000 (
    .clk          (clk         ),
    .rst_n        (rst_n       ),
    .clr          (clr_1000    ),
    .count_up     (count_1000  ),
    .data_out     (data_out_1000)
);
```

Se observă că același modul `numarator` a fost instanțiat de două ori, cu numele `numarator_mod100` și `numarator_mod1000`, cu parametrii `NO_BITS` și `MODULO_NUM` asociați explicit cu valori diferite.

Porturile modulelor sunt fie asociate cu același semnale (`clk` și `rst_n`) fie cu semnale diferite.

Desigur, ieșirile celor două instanțe `data_out` sunt conectate la semnale diferite (declarate `wire`): `data_out_100` și `data_out_1000`.

#### 4.4.2 Medii de testare

Deși nu modelează hardware, mediul de testare este deosebit de important pentru certificarea corectitudinii modelului HDL utilizat pentru implementarea de hardware. Sunt situații în care timpul alocat pentru verificare este chiar mai mare decât timpul alocat pentru proiectarea efectivă a modelului sintetizabil.

Această secțiune prezintă aspecte avansate ale HDL utilizate exclusiv pentru modelarea mediului de testare. Specificațiile prezentate sunt utilizate pentru a controla simularea și pentru a modela mediul exterior al sistemului digital.

##### Specificația `initial`

În modelul pentru mediul de testare pot fi utilizate fără restricții orice specificații Verilog.

Verilog are două specificații concurente (se execută în paralel, ordinea în cod este indiferentă) care modelează resurse hardware sintetizabile: `assign` și `always`.

Specificația `assign` permite atribuirea de valori către semnale declarate de tip `wire` în mod continuu (la orice modificare a semnalelor ce determină

valoarea semnalului atribuit).

Specificația `always` este o specificație concurentă care înglobează specificații secvențiale și care permite atribuirea de valori către semnale declarate de tip `reg`. Specificațiile secvențiale incluse în specificația `always` se execută în ordinea în care apar în cod. După execuția ultimei specificații, șirul specificațiilor se reia cu prima specificație în mod permanent, pe toată durata rulării simulării.

Specificația `initial` este similară cu specificația `always` din punctul de vedere al comportamentului legat de execuția specificațiilor secvențiale. Spre deosebire de specificația `always`, specificațiile incluse în specificația `initial` sunt executate o singură dată. După execuția ultimei specificații incluse în `initial`, specificația `initial` își termină definitiv efectul, până la încheierea simulării. Din acest motiv, specificația `initial` nu modelează resurse hardware reale și, ca efect, nu este sintetizabilă. Specificația `initial` se folosește exclusiv pentru a modela comportamentul semnalelor utilizate pentru a servi ca intrări pentru modulele hardware simulate. Într-un modul pot exista mai multe specificații `initial`, acestea executându-se în paralel, independent una față de cealaltă.

Specificația `initial` își începe execuția la timpul 0. În cadrul specificației `initial` pot fi introduse întârzieri de timp fie ca timp absolut (`#10` înseamnă „o întârziere de 10 unități de timp”), fie ca timp relativ la un eveniment asociat unui semnal (`@(posedge clk)` înseamnă „așteaptă până la apariția unui front pozitiv al semnalului `clk`”). Valoarea absolută a unității de timp este introdusă cu o definiție `timescale`.

În cadrul unui modul ce modelează un mediu de testare, pot coexista oricâte specificații `initial`. Toate își încep execuția la momentul 0 al simulării, indiferent de ordinea acestora în cod. Finalizarea specificațiilor poate avea loc la momente de timp diferite, în funcție de clauzele de timp ale fiecărei specificații.

Similar cu specificația `always`, fiecare specificație `initial` în care este atribuit un semnal devine o „sursă” („*driver*”, engl.) pentru acel semnal. Deci, dacă un semnal este atribuit în două specificații `initial`, este foarte probabil să apară conflicte, datorită existenței surselor multiple pentru semnal.

Ca exemplificare, sunt prezentate în continuare modurile de modelare a unor semnale utilizate ca intrări în orice sistem secvențial sincron: **semnalul de ceas** și **semnalul de reset**.

```
module ck_rst_tb #(
  parameter CK_SEMIPERIOD = 'd10 // semi-perioada ceasului
) (
  output reg clk    , // ceas
  output reg rst_n  // reset asincron activ 0
```

```

);
initial
begin
    clk <= 1'b0;           // valoare initiala 0
    forever #CK_SEMIPERIOD // valoare complementata
                           // la fiecare semi-perioada
        clk <= ~clk;
end

initial begin
    rst_n <= 1'b1;         // initial inactiv
    @(posedge clk);
    rst_n <= 1'b0;         // activare sincrona
    @(posedge clk);
    @(posedge clk);
    rst_n <= 1'b1;         // inactivare dupa doua perioade de ceas
    @(posedge clk);       // ramane inactiv pentru totdeauna
end

endmodule // ck_rst_tb

```

Se observă că cele două semnale au fost modelate cu două specificații initial independente.

În cazul specificației care modelează semnalul de ceas, prima specificație atribuie valoarea inițială a semnalului de ceas ( $\text{clk} = 1'b0$ );, iar a doua modelează regula de complementare a semnalului, la infinit, cu o întârziere între comutări: ( $\text{forever \#CK\_SEMIPERIOD clk} = \sim\text{clk}$ );. Specificația *forever* va face ca specificația *initial* să nu se finalizeze niciodată.

În cazul specificației care modelează semnalul de reset, prima specificație atribuie valoarea inițială a semnalului de reset ( $\text{rst\_n} <= 1'b1$ );, iar următoarele specificații sunt întârziate cu momente de timp relative la semnalul de ceas ( $\text{@(posedge clk)}$ );. În acest mod, forma de undă a semnalului de reset este păstrată corelată cu cea a semnalului de ceas, chiar dacă ulterior semnalul de ceas își modifică perioada prin schimbarea parametrului *CK\_SEMIPERIOD*. Specificația *initial* se va finaliza după execuția ultimei specificații secvențiale incluse. Se modelează astfel un semnal ce este activat inițial și apoi este lăsat inactiv până la finalizarea simulării.

Semnalele de control, intrări pentru modelul hardware testat, se pot modela cu comutări sincronizate cu semnalul de ceas:

```

initial begin
    req      <= 1'bx;
    req_data <= 'bx;
    @(negedge rst_n);

```

```

req      <= 1'b0;
@(posedge rst_n);
@(posedge clk);

@(posedge clk);
req      <= 1'b1;           // activeaza cererea
req_data <= $random;        // date aleatorii
@(posedge clk);
while (~ack) @(posedge clk); // asteapta confirmare
req      <= 1'b0;           // inactiveaza cererea
req_data <= 'bx;
@(posedge clk);

@(posedge clk);
$display ("%M %0t INFO: Final simulare.", $time);
$stop;
end

```

Se observă că cererea `req` este inițializată inactivă la momentul determinat de resetul asincron (`rst_n=0`), apoi este activată. După apariția confirmării (`ack=1`) cererea este inactivată. Toate acțiunile sunt sincronizate pe frontul crescător al semnalului de ceas cu specificația `@(posedge clk)`.

La finalul stimulului, s-a plasat o specificație de oprire a simulării (`$stop`) precedată de un mesaj informativ. În acest mod, proiectantul poate lansa simularea pentru un timp indefinit, simularea oprindu-se în momentul când s-au epuizat modificările stimulilor de intrare.

## Funcții de sistem

Funcțiile Verilog precedate de caracterul `$` se numesc **funcții de sistem**. Acestea sunt definite în limbajul Verilog și pot fi apelate pentru a controla simularea și a realiza diverse funcții.

Funcția `$display` permite afișarea unor mesaje ce pot fi particularizate prin utilizarea unor variabile. Afișarea variabilelor este determinată de opțiunea existentă în cadrul șirului afișat. Ordinea variabilelor trebuie să corespundă opțiunilor din lista de argumente. Lista de argumente este încadrată de caractere apostrof dublu.

Caracterul special procent (%) indică faptul că în șirul afișat va fi prezentă valoarea unei variabile. Specificarea formatului de afișare a unei variabile se face cu una din opțiunile prezentate în tabelul următor.

| Opțiune   | Descriere        |
|-----------|------------------|
| %d sau %D | Baza 10          |
| %b sau %B | Baza 2           |
| %h sau %H | Baza 16          |
| %o sau %O | Baza 8           |
| %c sau %C | Caracter         |
| %m sau %M | Nume modul       |
| %s sau %S | Șir de caractere |
| %t sau %T | Timp curent      |

Pentru afișarea caracterului procent (%) se va scrie caracterul procent dublu: (%%).

Dacă opțiunea este precedată de „0” (zero), atunci se vor elimina valorile de zero din partea mai semnificativă (stângă) a numerelor.

Un caz particular îl constituie opțiunea %m, care este singura opțiune fără argumente. Se recomandă utilizarea acestei opțiuni pentru a identifica numele modulului care a generat mesajul. Acest lucru este de foarte mare ajutor în proiecte mari, compuse dintr-un număr mare de module.

Funcția `$write` se comportă similar cu funcția `$display` cu diferența că `$display` adaugă la finalul șirului caracterul „linie nouă” (*Carriage Return*, engl.), în timp ce `$write` nu scrie acest caracter.

Funcțiile `$stop` și `$finish` sunt utilizate pentru a opri simularea la timpul la care sunt apelate. Diferența dintre cele două funcții constă în acțiunea realizată după oprirea simulării.

Funcția `$finish` întrerupe simularea și închide simulatorul. Ea este recomandată a fi utilizată pentru finalizarea simulării, în cazul rulării la linie de comandă.

Funcția `$stop` oprește simularea și pune simulatorul în mod de depanare. Este permisă reluarea simulării de la timpul curent, cu simulatorul în mod grafic.

Pentru depanarea modulelor, se recomandă utilizarea funcției `$stop` în momentul în care stimulii sunt finalizați. În acest mod, un proiectant poate rula simularea cu comanda `run -all` fără a cunoaște durata necesară pentru simulare.

Este recomandat ca înainte de apelul funcției `$stop` să existe un apel de funcție `$display` prin care să se precizeze motivul pentru care s-a oprit simularea.

Funcția `$random` se utilizează pentru a genera numere aleatorii, necesare pentru stimulii de intrare în scopul verificării modulelor proiectate. Desigur, funcția nu este sintetizabilă și se folosește exclusiv în modulele



generatoare de stimuli.

Funcția `$random` generează numere pseudo-aleatorii. Adică, un modul ce conține un apel al funcției `$random`, simulat pe același simulator, va produce de fiecare dată aceeași secvență de numere aleatorii. Rularea codului pe alt simulator nu garantează aceeași succesiune de numere aleatorii.

Funcția `$random` produce un număr aleatoriu reprezentat ca întreg pe 32 de biți. Dacă se doresc numere aleatorii în domenii cu anumite restricții, funcția se apelează împreună cu alte construcții sintactice.

Pentru că este mai intuitivă, este recomandată utilizarea funcției sistem `$urandom_range(min, max)` care întoarce un număr pozitiv în domeniul de valori întregi precizat. De exemplu, `$urandom_range(20, 30)` întoarce un număr aleatoriu între 20 și 30.

## Task

Specificația `task` permite gruparea unor specificații secvențiale pentru a fi apelate cu parametri diferiți de multiple ori. Aceasta permite scrierea condensată a unor secvențe repetitive. De exemplu, un modul inițiator pe o interfață „request-acknowledge” lansează o cerere după următorul algoritm:

- Activează cererea ( $req = 1$ ) simultan cu stabilirea datelor ( $reqdata$ ).
- Așteaptă următorul front de ceas.
- Cât timp nu a sosit confirmarea ( $ack = 0$ ), așteaptă următorul front de ceas.
- Când a sosit confirmarea ( $ack = 1$ ), finalizează transferul.
- Dacă nu urmează o cerere „back-to-back”, dezactivează cererea ( $req = 0$ ), pune datele în stare indiferentă ( $reqdata = x$ ) și așteaptă numărul de perioade de ceas de pauză între cererile succesive.

Din algoritm se observă că sunt necesari 2 parametri: *valoarea datelor* și *durata pauzei între cereri*. Se consideră că dacă durata pauzei este 0, următoarea cerere este „back-to-back”. Task-ul poate fi descris de codul următor:

```
task send_req;
input [16-1:0] data_p ; // date transmise
input [32-1:0] delay_p; // pauza între cereri succesive
begin
    req          <= 1'b1 ; // activeaza cererea
```

```

req_data  <= data_p;    // depune datele
@(posedge clk);
while (~ack) @(posedge clk);
if (delay_p != 'd0) begin
    req      <= 1'b0;    // inactiveaza cererea
    req_data <= 'bx;    // invalideaza datele
    repeat (delay_p) @(posedge clk);
end
end
endtask

```

Se observă faptul că intrările task-ului (`data_p` și `delay_p`) sunt similare unor argumente formale ale unei funcții (într-un limbaj de programare). Aceste intrări vor primi valori în momentul apelării task-ului.

Următoarea secvență, din cadrul unei specificații `initial`, reprezintă apeluri multiple ale task-ului:

```

initial begin
    req      <= 1'bx ;
    req_data <= 'bx ;
    @(negedge rst_n);
    req      <= 1'b0 ;
    @(posedge rst_n);
    repeat (5) @(posedge clk);

    // cereri punctuale
    // send_req(data, pauza);
    send_req('d0, 'd2);
    send_req('d1, 'd3);
    send_req('d2, 'd0);
    send_req('d3, 'd1);

    // cereri aleatorii, infinite
    forever send_req($random, $urandom_range(0,10));
end

```

## Inițializarea variabilelor

În mod normal, o memorie se scrie prin circuit înainte de a fi citită. Din acest motiv, nu este nevoie ca memoria să fie inițializată cu date cunoscute. Dacă nu este inițializată, ca orice variabilă Verilog, se poate considera că valorile memorate sunt toate 'bx.

Există însă cazuri când este necesară inițializarea memoriilor. Exemple de memorii ce trebuie inițializate sunt memoriile ROM sau modulele LUT („Look-Up Table”, engl.).

Alteori, module de memorie RAM (deci care se scriu înainte de a fi citite) sunt utilizate pentru a testa exclusiv circuitul care preia date din memorie. Se poate presupune că datele au fost scrise anterior (de către un alt modul) și nu se dorește simularea perioadei de scriere a memoriei. De exemplu, dacă se testează un circuit care preia datele din memorie pentru a fi afișate („driver video”).

Inițializarea variabilelor bidimensionale care modelează memoriile este un proces exclusiv de simulare (nu are corespondent în hardware real).

Inițializarea tuturor locațiilor memoriei cu aceeași valoare constantă se poate face într-o specificație `initial` în care toate locațiile sunt inițializate secvențial, una câte una, la timpul 0 de simulare, cu valoarea constantă:

```
integer    idx ;                      // index
initial begin

    // initializare totala cu o valoare constanta
    for (idx=0; idx < MEM_DEPTH; idx=idx+1)
        mem[idx] = 'hAB;             // valoare constanta

end
```

Funcțiile Verilog `$readmemb` și `$readmemh` inițializează o matrice de memorie (variabilă bidimensională) cu valori dintr-un fișier extern. Fișierul conține caractere ASCII ce reprezintă valori binare („0” sau „1”) sau în baza 16 („0-9”, „A-F”).

Valorile trebuie să aibă aceeași lățime cu cea a memoriei (exprimată în biți) și să fie separate cu spații.

Valorile adreselor de început și de sfârșit sunt valori în baza 16 (atât pentru `$readmemb` cât pentru și `$readmemh`) precedate de caracterul „@”.

Sintaxa specificațiilor de inițializare a memoriilor este:

```
$readmemb("nume_file", mem);
$readmemh("nume_file", mem);
```

## Funcții pentru accesul la fișiere de intrare/ieșire

Verilog permite accesul la fișierele de pe calculatorul local pentru a ușura implementarea unor medii de verificare complexe și pentru a înlesni depanarea. Specificațiile de acces la fișiere nu sunt sintetizabile și trebuie utilizate exclusiv în module ce modelează mediul de verificare.

Funcțiile de acces la fișierele de pe mediul de stocare al calculatorului sunt apelate prin funcții sistem specifice Verilog. Numele acestor funcții

este precedat de simbolul \$. Un fișier local, identificat prin cale și nume în sistemul de fișiere al calculatorului, este asociat cu un descriptor de fișier în Verilog. Calea fișierului se recomandă a fi relativă la directorul de simulare pentru a asigura portabilitatea modelului.

Sintaxa asocierii dintre un fișier și un descriptor este următoarea:

```
integer file_id;           // descriptor de fisier
file_id = $fopen("file_name" [, type]);
```

Tipul fișierului semnifică scopul pentru care este deschis fișierul. Cele mai utilizate tipuri sunt menționate în tabelul următor:

| Tip fișier   | Descriere                                                            |
|--------------|----------------------------------------------------------------------|
| „r” sau „rb” | Deschidere pentru citire                                             |
| „w” sau „wb” | Creare pentru scriere                                                |
| „a” sau „ab” | Scriere în continuarea unui fișier existent ( <i>append</i> , engl.) |

`$fopen` deschide un fișier și îi asociază un descriptor, declarat ca un întreg pe 32 de biți. Descriptorul va fi utilizat ulterior de către specificațiile de acces la fișier. În caz de eroare, descriptorul are valoarea zero. Se recomandă ca după o specificație de deschidere de fișier să se verifice dacă fișierul a putut fi deschis, prin testarea descriptorului de fișier:

```
if (data_file == 0) begin
    $display("Nu se poate deschide fisierul <cale/nume fisier>.");
    $stop;
end
```

Caracterul „b” prezent în tipul fișierului desemnează un „fișier binar” (conține valori binare). În lipsa acestuia, fișierul este de tip „text” (conține caractere ASCII).

Comanda de închidere a unui fișier este:

```
$fclose(file_id);
```

Pentru citirea din fișier există comenzi specifice. Scrierea în fișier se poate face cu comanda `$fdisplay`. Scrierea se face temporar într-un buffer de ieșire. Bufferul este depus în fișier fie cu comenzi explicite `$flush(file_id)` fie la întreruperea sau finalul simulării.

### Inițializarea variabilelor din fișier `$fread`

Funcțiile de inițializare a memoriilor `$readmemb` și `$readmemh` au ca argument un fișier extern ce conține caractere ASCII. Dacă se dorește inițiali-

zarea unei memorii dintr-un fișier binar (de exemplu o imagine în format **.raw**) această opțiune nu este potrivită.

În schimb, se poate deschide fișierul în citire ca fiind de tip binar (opțiunea „*rb - read, binar*”) și atribui câte o valoare fiecărei locații de memorie.

```
integer    file_id          ; // handler fisier
integer    status           ; // stare operatie cu fisiere

initial begin
    // initializeaza memoria
    file_id = $fopen("mem_map.raw", "rb");
    if (file_id == 0) begin
        $display("Nu se poate deschide fisierul mem_map.raw");
        $stop;
    end
    for (idx=0; idx<MEM_DEPTH; idx=idx+1)
        status = $fread(i_mem_lrw.mem[idx], file_id);
    $fclose(file_id);
end
```

### Salvarea conținutului memoriei în fișier **\$fwrite**

De multe ori este necesară inspectarea conținutului unui memorii la un anumit timp de simulare. Eventual, este nevoie de interpretarea datelor existente în memorie (de exemplu ca fiind o imagine în format **.raw**). Soluția constă din transferul conținutului memoriei într-un fișier binar. Fișierul se va deschide pentru scriere de tip binar (opțiunea „*wb - write, binar*”). Printr-o specificație **for** se poate copia în fișier fiecare locație de memorie, formatată cu opțiunea „*%c*” (caracter).

```
integer    file_id          ; // handler fisier

initial begin
    // scrie memoria destinatie in fisier binar
    file_id = $fopen("mem_map.raw", "wb");
    if (file_id == 0) begin
        $display("Nu se poate deschide fisier mem_map.raw");
        $stop;
    end
    for (idx=0; idx<MEM_DEPTH; idx=idx+1)
        $fwrite(file_id, "%c", i_mem_lrw.mem[idx]);
    $fclose(file_id);
end
```



### Balada Verilogului

Foaie verde Verilog,  
Iese ursul din bârlog.  
Hai că-mi iese simularea,  
Nu mai dă tool-ul eroarea.

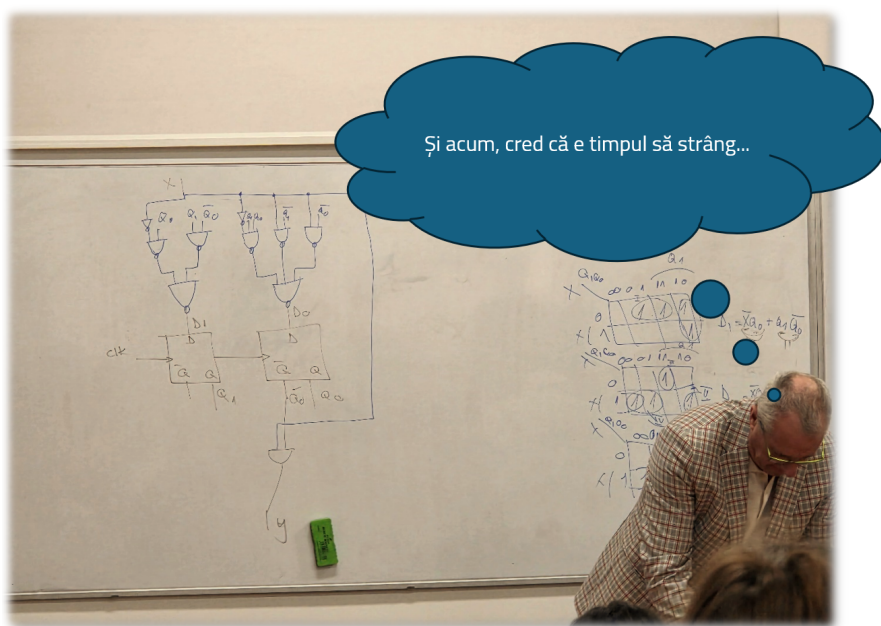
Clock-ul bate cadențat,  
Flip-flopul s-a așezat.  
Starea merge, tot e clar!  
Numa' eu să n-am habar?

*Text generat cu ChatGPT.*

## Grăirea de încheiere

Într-o perioadă în care performanța profesorilor este măsurată  
exclusiv prin numărul de articole publicate,  
dăruiesc generațiilor viitoare o

**„Carte de învățătură”.**









# ELECTRONICĂ DIGITALĂ

CARTE DE ÎNVĂȚĂTURĂ

TOMUL PENTRU ZĂBOVIT ȘI GÂNDIT  
DAN NICULA

📖 Această carte face parte din șirul cel vrednic de învățatură numit „Carte de învățatură de Electronică Digitală”, scrisă cu drag ❤️, cu minte ageră 🧠 și-o tastatură care a văzut multe nopți nedormite 🌙😴.

📅 Ediția din anul 2025 vine cu rânduială nouă 📖 și legături mai strânse între toate cele ce țin de lumea binară 📡🔌, ca să fie sprijin bun și de nădejde 🛠️📚 pentru studenții de la:

- Electronică Aplicată 🛠️🔌 (pentru cei ce repară tot, în afară de viața personală ❤️)
- Telecomunicații 📡 (pentru cei ce vorbesc în unde 📶📡)
- Calculatoare 🖥️📡 (care au emoții numai la compilare 😊🔄)
- Automatică 🤖📡 (pentru cei care automatizează până și făcutul cafelei)
- Tehnologia Informației 📡📚 (unde e multă informație, dar timpul tot nu ajunge ⌚)

📖 Lucrarea de față e în patru tomuri, ca Evangheliile digitale 📖📖📖📖:

1. 🧠 Tomul pentru zăbovit și gândit – cu noțiuni teoretice 📖 și modelări în Verilog 📡, pentru cei care vor să înțeleagă înainte să aprindă ceva 🔌💡.
2. 📐 Tomul pentru chibzuit și buchisit – cu probleme de seminar 📐 și proiectare, în care guma se tocește mai tare ca mintea 🧠.
3. 🛠️ Tomul pentru meșterit și muncit – pentru laborator 🛠️, cod Verilog 📡 și mici minuni făcute din siliciu și sudoare 🧪⚙️.
4. 🏃 Tomul pentru grabă mare – cu chestionare ? și rezumate 📖 pentru zilele când examenele bat în ușă 🚪 și memoria bate pasul pe loc 🏃.

📖 Așadar, ia cartea, deschide-o cu încredere 🤝, și, chiar dacă n-ai înțeles tot din prima, nu te teme: 🙌 nici circuitele n-au știut să gândească până n-au primit ceas! ⌚⚡

Dan NICULA este profesor la Universitatea Transilvania din Brașov.  
[www.dannicula.ro](http://www.dannicula.ro)



Editura  
Universității  
Transilvania  
din Brașov

ISBN 978-606-19-1794-5



9 786061 917945