Liviu-Alexandru MARINA

Laura FLOROIAN

ANALIZA ȘI SINTEZA CIRCUITELOR NUMERICE/ ELECTRONICĂ DIGITALĂ

Îndrumar de laborator

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 recunoscută CNCSIS, cod 81

ISBN 978-606-19-1817-1 (e-book)

Lucrarea a fost avizată de Consiliul Departamentului de Automatică și tehnologia informației, Facultatea de Inginerie electrică și știința calculatoarelor a Universității Transilvania din Brașov.

Cuprins

Laborator 1
Introducere în OrCAD Capture CIS – Lite

1.1. Scopul lucrării

În această lucrare de laborator se propune o introducere în pachetul software *OrCAD Capture CIS-Lite*, care va fi utilizat pentru proiectarea şi simularea schemelor logice.

1.2. Prezentare teoretică

OrCAD este un pachet de programe independente din punct de vedere al funcționării, destinat proiectării asistate de calculator a circuitelor electronice numerice (digitale) şi analogice, al cărui producător este *Cadence Design Systems*. Acesta permite proiectarea schemei şi verificarea, prin simulare, a funcționării corecte din punct de vedere logic, realizând în același timp cablajul de conexiune între componentele schemei. Pachetul *OrCAD* preia majoritatea sarcinilor de proiectare de rutină, permițând utilizatorului să se concentreze asupra activității de design.

OrCAD Capture este o aplicație utilizată pentru proiectarea circuitelor logice, fiind parte din pachetul de programe *OrCAD*. *Capture* include şi funcționalități de scripting (TCL-TK) care permit utilizatorilor automatizarea şi personalizarea schemelor logice. De asemenea, *Capture* permite exportul descrierii hardware a schemei logice către *Verilog* sau *VHL*, limbaje de descriere hardware [1].

OrCAD PSpice este o aplicație pentru simularea şi verificarea circuitelor logice analogice şi digitale. *PSpice* este un acronim pentru *Personal Simulation Program with Integrated Circuit Emphasis*.

Această aplicație rulează în mod tipic circuitele definite în *OrCAD Capture* şi, optional, poate integra *MATLAb Simulink*, utilizând interfața *Simulink to PSpice*.

Această primă lucrare de laborator cuprinde o introducere în funcționalitățile de bază ale programului, cum ar fi crearea unui proiect, încărcarea bibliotecilor necesare, căutarea şi inserarea componentelor.

1.2.1. Noțiuni de bază pentru crearea unei scheme logice

1.2.1.1. Crearea unui proiect

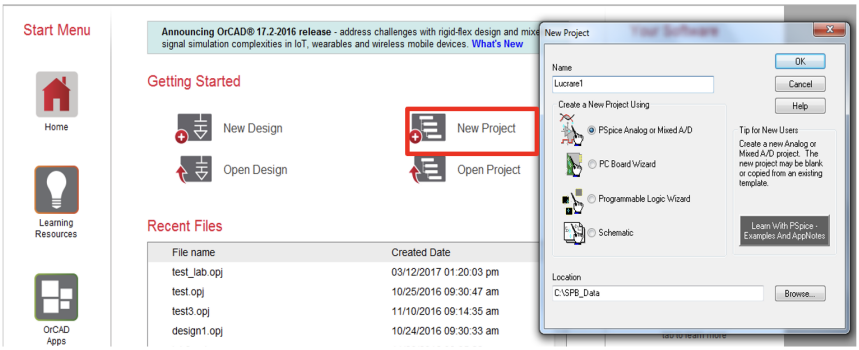


Fig. 1.1: Crearea unui nou proiect în OrCAD Capture.

Primul pas pentru proiectarea unei scheme logice este crearea unui nou proiect. Pentru acest lucru se vor realiza următoarele etape:

- Se alege opțiunea *New Project* din fereastra de start a programului. Pentru a crea un proiect cu scopul de a fi simulat, se va selecta opțiunea *PSpice Analog or Mixed A/D* şi se va adăuga un nume, așa cum se poate observa grafic şi în Figura 1.1. Proiectele vor fi salvate într-un folder propriu fiecărui student, creat pe partiția ”D:” a computerului și denumit după dorință. Folosindu-se opțiunea *Browse*, se va plasa fiecare proiect nou în acest folder;
- Se alege tipul de proiect. Se poate crea un nou proiect utilizând-se ca șablon unul deja existent (selectând *Create based upon an existing project*) sau se poate crea unul gol (*Create a blank project*), aşa cum se poate observa în Figura 1.2. Pe parcursul acestui laborator se vor crea, în general proiecte goale, fără a se utiliza un template deja existent. Structura

unui proiect este prezentată în Figura 1.3, unde se pot vizualiza paginile proiectului sau componentele stocate în memoria cache.

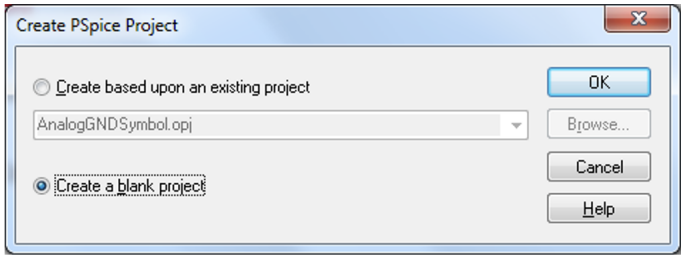


Fig. 1.2: Selectare tip proiect în OrCAD Capture.

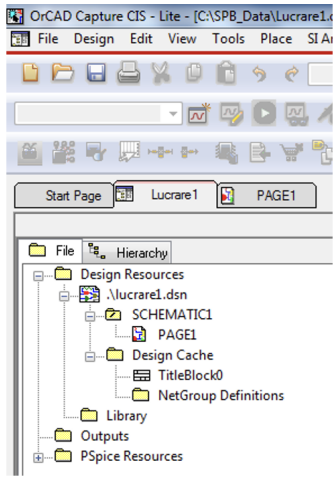


Fig. 1.3: Structura unui proiect în OrCAD Capture.

1.2.1.2. Selectarea şi adăugarea componentelor într-o schemă logică

După ce proiectul a fost creat, se poate trece la editarea schemei logice. Pentru a plasa componentele în fereastra de editare, se poate utiliza meniul *Place – Part*. În urma acestei comenzi se va deschide o fereastră unde se pot căuta componentele necesare, așa cum este prezentat în Figura 1.4. Deoarece, inițial, lista de componente poate fi goală, trebuie adaugate bibliotecile necesare.

Pentru ca schema să poată fi simulată, trebuie adăugate bibliotecile *PSpice*. Se recomandă ştergerea bibliotecilor existente şi adăugarea celor compatibile cu mediul de simulare *PSpice*.

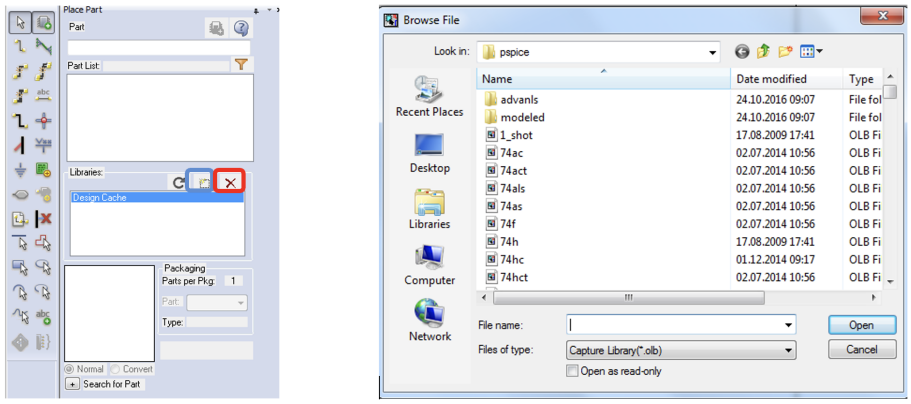


Fig. 1.4: Paşii necesari pentru adăugarea bibliotecilor necesare proiectării.

Pentru a se adăuga bibliotecile, se va utiliza butonul *Add Library*, încercuit cu albastru în Figura 1.4. În urma acționării acestui buton, se va deschide o fereastră de unde se pot selecta bibliotecile dorite. Pentru simplitate se vor selecta toate fișierele afișate, utilizându-se combinația de taste *Ctrl* + *A*. În Figura 1.4 poate fi observată și pictograma pentru butonul *Remove library*, evidențiată prin chenarul de culoare roșie. Calea relativă către directorul *pspice* este: ".../Cadence/SPB_17.2/tools/capture/library/pspice".

Pentru utilizarea porților logice de bază, cea mai simplă cale ar fi folosirea meniului *Place – PspiceComponent –Digital – Gates*, unde se oferă opțiunea selectării porților *ȘI*, *SAU*, *ȘI-NU*, *SAU-NU*, *SAU EXCLUSIV* cu două intrări și *INVERSOR*, prezentate în Figura 1.5.

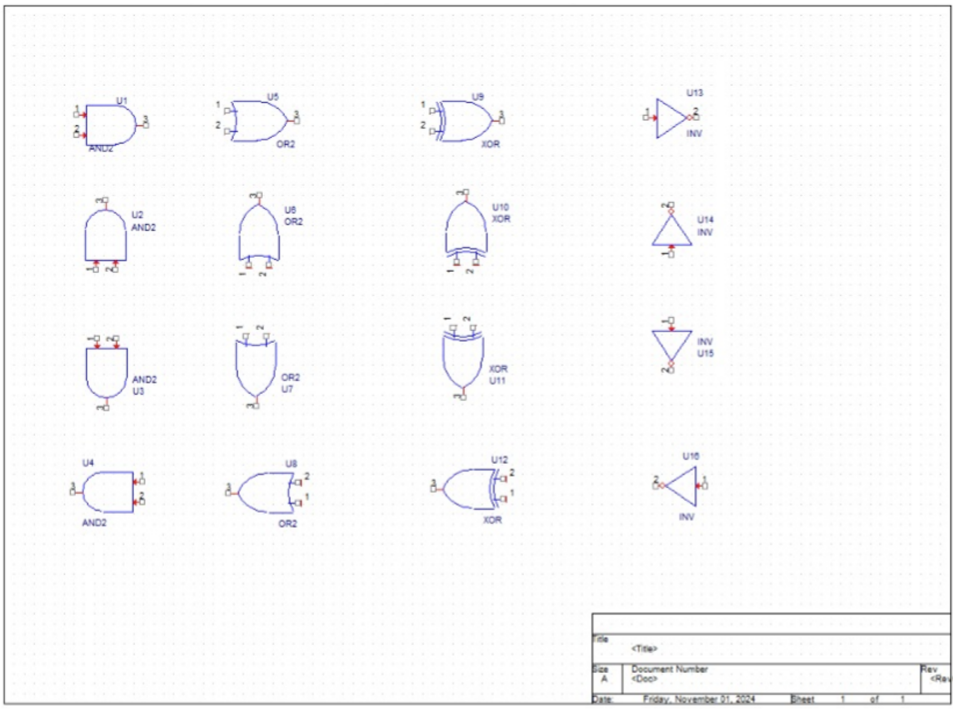


Fig. 1.5: Porți *ȘI*, *SAU*, *ȘI-NU*, *SAU-NU*, *SAU EXCLUSIV* cu două intrări și *INVERSOR*.

Pentru identificarea unei diversități mai mari de componente și pentru evitarea eventualelor probleme generate de adăugarea bibliotecilor necorespunzătoare, se poate utiliza şi fereastra destinată adăugării componentelor acceptate pentru simularea circuitului, folosindu-se meniul *Place – PSpice-Component – Search*. Prin selectarea opțiunii *Search*, în partea dreaptă a ecranului se deschide o fereastră ce conține 2 tabele: cel de sus cuprinde toate categoriile de elemente de circuit analogice și digitale din bibliotecile *PSpice*, iar cel de jos cuprinde toate elementele din categoria selectată în tabelul de sus (vezi Figura 1.6).

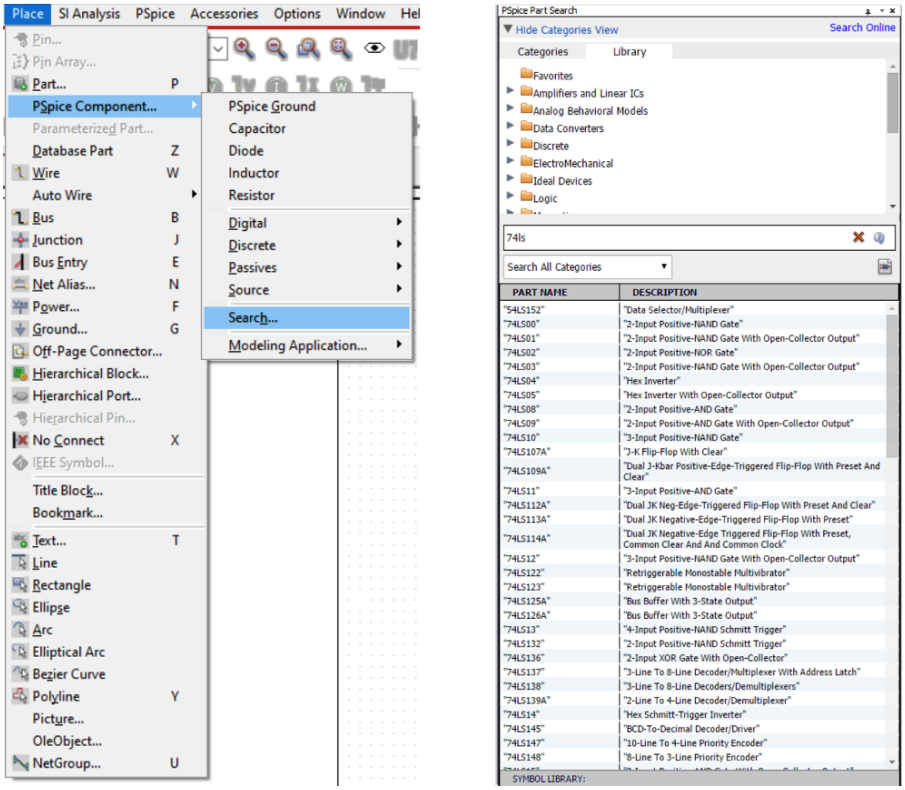


Fig. 1.6: Pașii necesari utilizării *PSpice Component Search*.

Pentru găsirea componentei necesare, se va scrie codul/numărul acesteia în câmpul *Part*. Pentru adăugarea în schema logică se va efectua dublu clic pe componenta dorită. Opțional, în lipsa codului specific componentei, se poate folosi şi numele acesteia. De exemplu, pentru o poartă *AND* cu 2 intrări, se poate căuta numele *AND2*, aşa cum se poate observa în Figura 1.7.

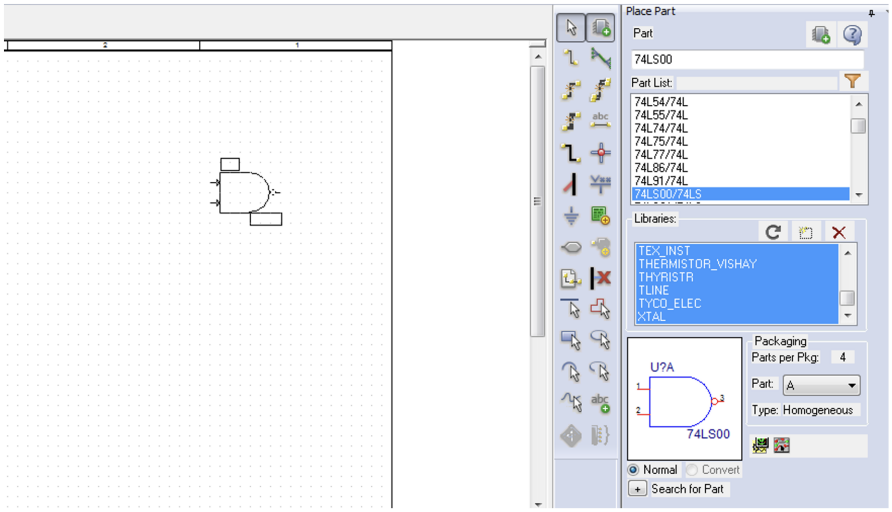


Fig. 1.7: Căutarea şi adăugarea unei componente în schema logică.

Pentru a roti componenta selectată se poate utiliza opțiunea *clic dreapta – Rotate* sau tasta *R*. De asemenea, se pot face reflexii orizontale sau verticale ale componentelor, cu instrucțiunile *Mirror Horizontally*, respectiv *Mirror Vertically*, existând și opțiunea de *Mirror Both*.

1.2.1.3. Adăugare porturi intrare/ieşire

Adăugarea porturilor de intrare, respectiv ieşire reprezintă un pas important în proiectarea unei scheme logice şi de aceea este importantă întelegerea rolului acestora. În *OrCAD*, aceste porturi sunt utilizate cu precădere în momentul proiectării unei scheme ierarhice. Au denumirea de *Hierarchical Ports* şi se pot adăuga utilizând meniul *Place – Hierarchical Port*. Se pot adăuga porturi de intrare (*PORTRIGHT-R*) pentru transmiterea semnalelor între scheme şi porturi de ieşire (*PORTLEFT-L*) pentru a reține semnalul de ieşire. Mai multe detalii se pot observa în Figura 1.8.

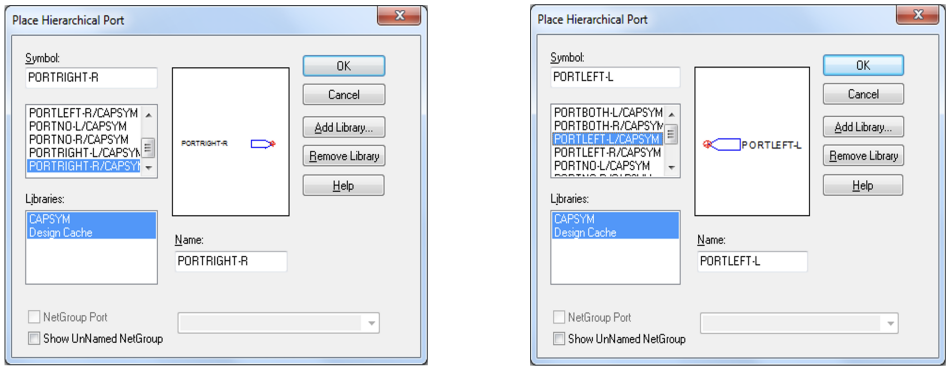


Fig. 1.8: Adăugare porturi de intrare/ieşire.

1.2.1.4. Adăugare fire interconectare

Pentru plasarea firelor de interconectare se poate utiliza comanda *Place – Wire* sau se poate utiliza direct tasta *W*. Atenție la plasarea corectă a firelor. Se va face exact din pin în pin. Firele nu trebuie să se interconecteze, în caz contrar în schema logică se va produce un scurt circuit, care duce la imposibilitatea simulării schemei. Desigur că există și situații în care firele sunt în contact electric, caz în care trebuie utilizată instrucțiunea *Junction (J)* pentru a semnala acest lucru.

În Figura 1.9 este exemplificat modul de selectare şi adăugare a firelor, cu și fără contact electric.

Când mai multe semnale de același tip au același traseu geometric, se utilizează magistrale, adică mănunchiuri de fire. De exemplu, magistrala de conectare a unui mouse conține 4 fire trecute prin același cablu și conectate printr-o singură mufă, la fel fiind în cazul tastaturii. Monitorul are o magistrală de 15 fire, iar în calculator există și magistrale foarte mari, de 128 și chiar de 256 fire.

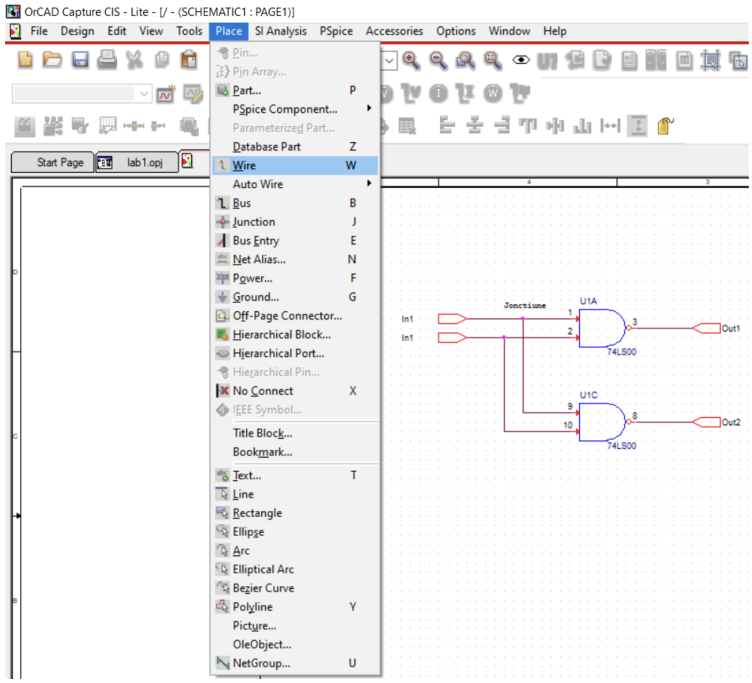


Fig. 1.9: Adăugare fire de interconectare, cu și fără contact electric.

Pentru a reprezenta o magistrală se folosește instrucțiunea *BUS*, iar pentru a prelua semnalele din ea *BUS ENTRY*. Un exemplu pentru utilizarea magistrale este expus în Figura 1.10.

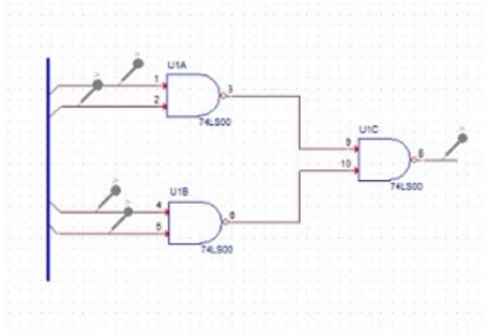


Fig. 1.10: Exemplificare utilizare magistrală de conectare.

1.2.1.5. Adăugare surse de semnal

Sursele de semnal definesc intrările în circuitul logic, având rolul de a furniza semnalul electric necesar.

În acest laborator se vor folosi surse de semnal digitale. Sursele de semnal se găsesc în fereastra *Place – Part* şi pot fi căutate folosind numele *DIGSTIM*, aşa cum se poate observa în Figura 1.11. Acestea se găsesc în librăria *sourcstm.olb*. După adăugare, sursele de semnal se pot edita, folosindu-se comanda *clic dreapta – Edit PSpice Stimulus*. În momentul plasării unei surse, se va modifica doar numele. Câmpul *Implementation* se va completa doar după editare. Mai multe detalii despre editarea surselor de semnal vor fi prezentate în laboratorul viitor, când se va simula o schemă logică.

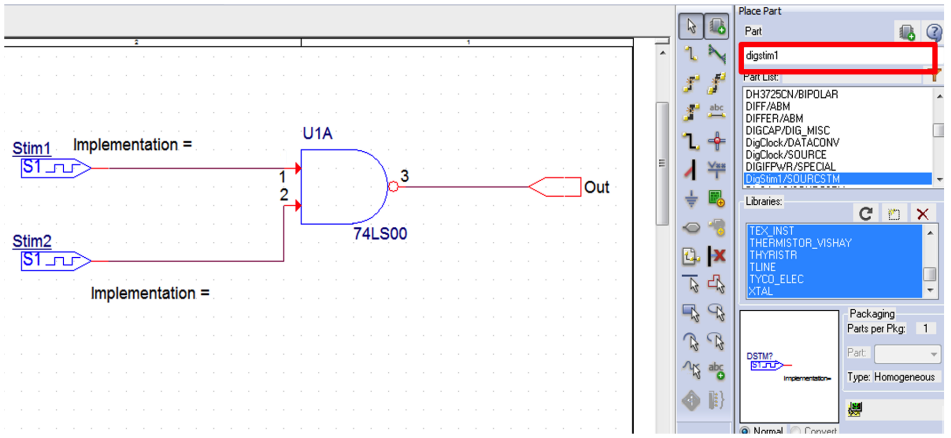


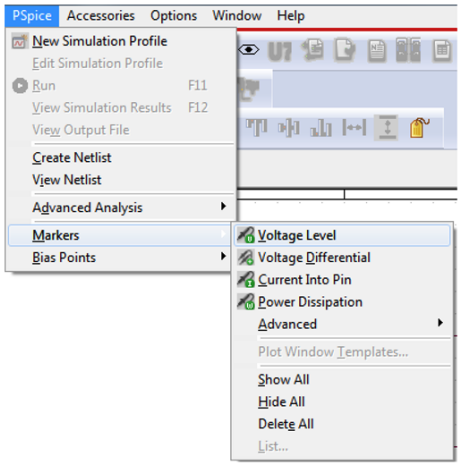
Fig. 1.11: Plasare surse de semnal.

1.2.1.6. Adăugare masă şi elemente de verificare ale semnalelor

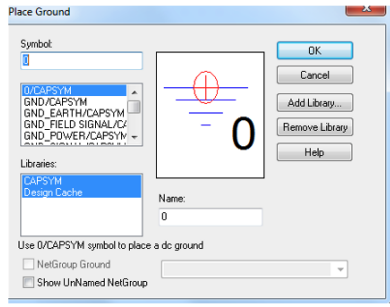
Pentru a vizualiza semnalele de ieşire în urma unei simulări, sau pentru a verifica valoarea semnalelor de intrare se pot adăuga elemente pentru măsurarea tensiunii (voltmetre). Acestea se pot plasa folosind meniul *PSpice – Markers - Voltage Level*, aşa cum se poate obseva în Figura 1.12 a.

De asemenea, pe parcursul proiectării, poate fi necesară și adăugarea masei digitale în schemele logice. Acest lucru se poate realiza utilizându-se comanda *Place – Ground*. Detalii se pot observa în Figura 1.12 b.

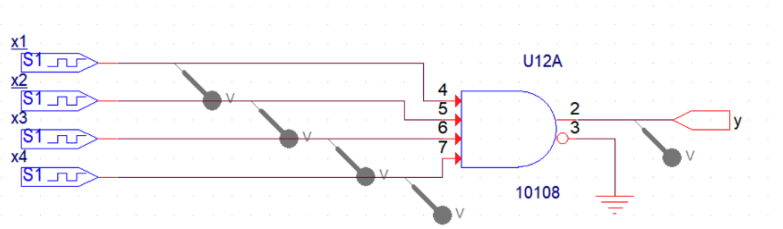
O schemă simplă care ilustrează modul de utilizare a masei și a elementelor de verificare a semnalelor poate fi vizualizată în Figura 1.12 c.



a. Adăugare elemente măsurare semnal.



b. Adăugare masă.



c. Schemă exemplificare utilizare voltmetru şi masă

Fig. 1.12: Adăugare masă şi elemente de verificare ale semnalelor.

1.3. Desfășurarea lucrării

În cadrul acestei lucrări de laborator se vor proiecta în *OrCAD Capture* schemele logice descrise mai jos, fără a simula funcționarea acestora. Se vor aplica noțiunile teoretice descrise anterior și se vor urmări etapele necesare proiectării unei scheme logice.

1.3.1. Proiectarea schemei logice pentru o funcție cu 2 variabile

Se cere proiectarea schemei logice a unei funcții cu două variabile utilizându-se porți logice *ȘI-NU* (componenta $74LS00$), pornindu-se de la forma acesteia:

$$F = \overline{\overline{x_1 \cdot x_2} \cdot \overline{x_1 \cdot x_2}} \tag{1}$$

Implementarea acestei scheme logice se poate observa în Figura 1.13.

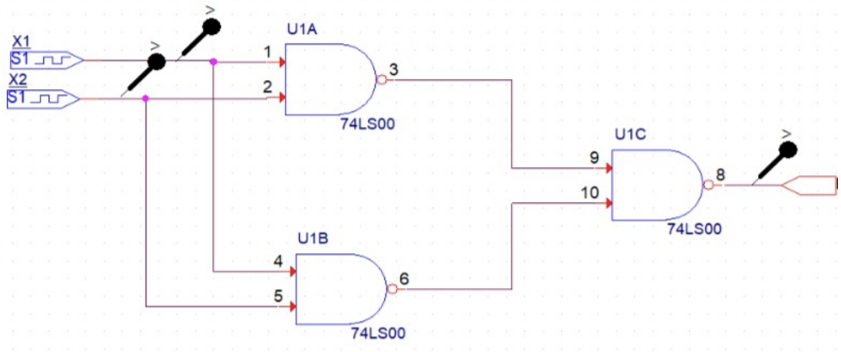


Fig. 1.13: Schemă logică funcție cu două variabile.

1.3.2. Proiectarea unui multiplexor 2:1

Pentru a exersa cunoştinţele dobândite în acest laborator, se propune proiectarea unei scheme logice a unui multiplexor 2:1 (vezi Figura 1.14).

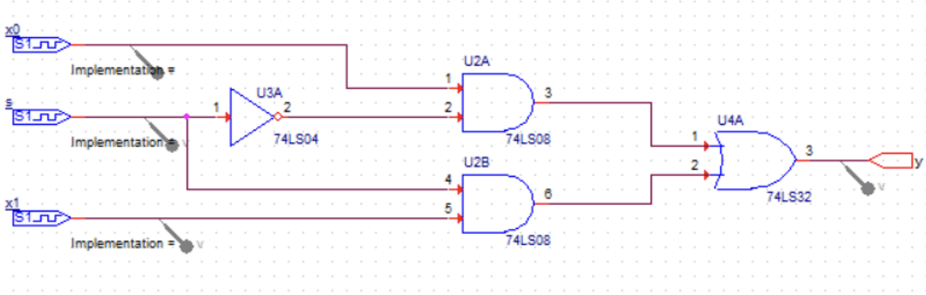


Fig. 1.14: Schemă logică multiplexor 2:1.

Ieşirea y este dată de ecuația rezultată din tabela de adevăr a circuitului:

$$y = x_0 \cdot \overline{s} + x_1 \cdot s \tag{2}$$

Pentru implementare se vor utiliza următoarele componente:

- 3 surse de semnal (x_0 , x_1, s);
- 1 *INVERSOR* pentru negarea semnalului s ($74LS04$);
- 2 porți logice *AND* ($74LS08$);
- 1 poartă *OR* ($74LS32$);
- 1 port de ieşire y (*PORT-LEFT*);
- voltmetre.

1.4. Conținutul referatului

1. Însuşirea cunoștințelor de bază ale pachetului de programe *OrCAD*;
2. Proiectarea schemelor logice.

1.5. Verificarea cunoştințelor

1. Care sunt paşii necesari creării unui proiect în *OrCAD Capture*?
2. Ce biblioteci trebuie adăugate pentru a permite simularea schemei logice proiectate?
3. Care este scopul surselor de semnal dintr-o schemă logică?
4. Care este scopul porturilor de intrare/ieşire?
5. Ce rol au elementele de verificare ale semnalelor?

Laborator 2
Simularea unei scheme logice în pachetul de programe OrCAD

2.1. Scopul lucrării

Schemele logice proiectate utilizându-se *OrCAD Capture* pot fi simulate pentru verificarea funcționării corecte, prin intermediul aplicației *OrCAD PSpice*. În acest laborator, se vor proiecta schemele logice ale unei funcții cu trei variabile și ale unui multiplexor 4:1, urmând să fie prezentați pașii necesari pentru simularea acestora.

2.2. Prezentare teoretică

Prima parte a acestei lucrări de laborator conține paşii de urmat în mediul de proiectare *OrCAD* pentru a se putea rula simularea unei scheme logice. De asemenea, se va introduce noțiunea de multiplexor, deoarece o parte a aplicației practice este reprezentată de simularea unui circuit de multiplexare.

2.2.1. Setări necesare rulării unei simulări

2.2.1.1. Crearea unui profil de simulare

În momentul creării unui nou proiect în *OrCAD Cadence*, este recomandat să se creeze şi un nou profil de simulare.

Pentru realizarea acestui lucru se va utiliza comanda *Pspice – New Simulation Profile*. În urma acestei comenzi, se va deschide o fereastră unde se va seta numele profilului de simulare. În secțiunea *Inherit from* se va alege opțiunea *None*, în cazul în care dorim să creăm un profil cu noi setări, sau se poate selecta un profil deja existent. După crearea profilului de simulare, o fereastră numită *Simulation Setting* va apărea pe ecran.

Trebuie setat timpul dorit pentru rularea simulării, în câmpul *Run to time*. Detalii se pot observa în Figura 2.1.

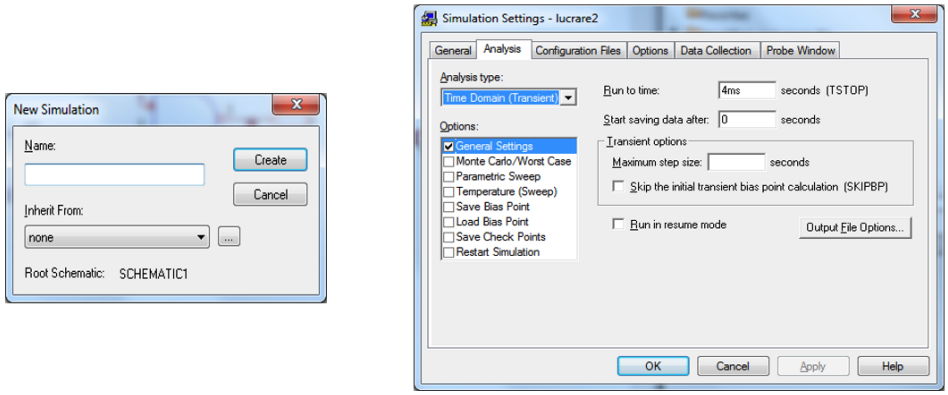
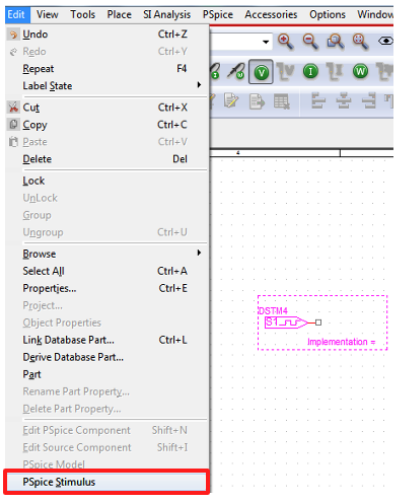


Fig. 2.1: Crearea şi setarea unui profil de simulare.

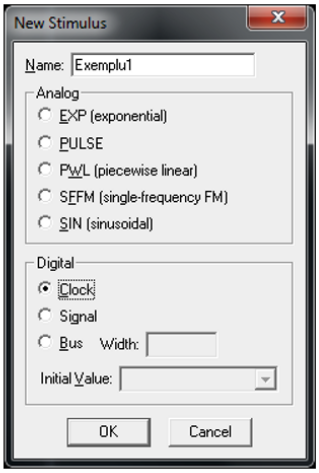
2.2.1.2. Editare surse semnal digitale (DigStim)

Pentru ca schema logică să poată fi simulată, trebuie adăugate şi editate sursele de semnal (secțiunea 1.2.1.5.). Pentru editare, se va selecta una dintre sursele adăugate şi se va utiliza opțiunea *Edit – Pspice Stimulus*, sau *clic dreapta – Edit PSpice Stimulus*. În urma acestei comenzi, se va deschide o nouă fereastră numită *Stimulus Editor*, unde se va efectua editarea (vezi Figura 2.2.a).

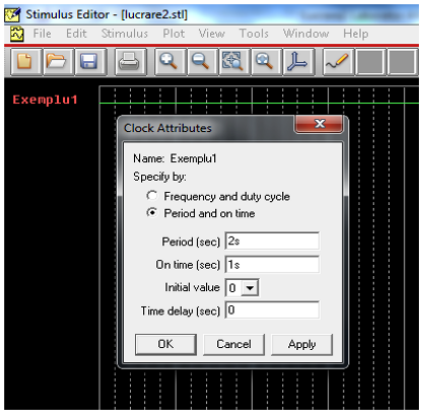
Prima etapă a editării constă în alocarea unui nume pentru implementarea sursei de semnal selectate şi alegerea tipului de semnal, digital sau analog. În cadrul acestui laborator vom lucra cu semnale digitale. În urma selectării tipului de semnal *Digital*, se poate alege una dintre opțiunile *Clock*, *Signal* sau *Bus*. Deoarece schemele logice care vor fi simulate, vor avea semnale de intrare ce comută în funcție de timp, vom alege opțiunea *Clock*. Dacă semnalul de intrare nu se schimba pe parcursul timpului de simulare, se poate alege şi opțiunea *Signal*, a cărei valoare poate fi setată la 1 sau 0. În Figura 2.2.b se pot observa paşii descrişi mai sus.



a. Editare surse de semnal.



b. Selectare tip de semnal.



c. Exemplu editare parametrii semnal intrare.

Fig. 2.2: Editare surse de semnal.

A doua etapă constă în editarea tipului de semnal selectat, în cazul de față semnalul digital, de tip *Clock*. Aşa cum s-a specificat şi mai sus, în electronica digitală se folosesc surse de semnal ce comută în timp între valorile 0 și 1. Aşadar, în fereastra ce se va deschide, se va alege opțiunea *Period and on time*. În urma acestei acțiuni vor apărea câmpurile pentru

perioda semnalului (*Period (sec)*, timpul în care semnalul e activ pe 1 logic (*On time (sec)*), valoarea inițială (*Initial Value*) şi întârzierea semnalului *(Time Delay (sec)*). Acesta câmpuri se vor completa în funcție de forma dorită a semnalului de intrare. În 2.2.c se pot observa aceste setări.

Pentru a edita toate surse de semnal din cadrul unei scheme, se poate proceda aşa cum a fost descris mai sus, şi anume prin selectarea fiecărei surse de semnal, editarea acesteia, salvarea proprietăților şi închiderea ferestrei *Stimulus Editor*.

O altă variantă este ca în fereastra *Stimulus Editor* să se utilizeze comanda *Stimulus – New* pentru a declara şi edita sursele de intrare. După ce toate sursele au fost adăugate, se vor salva modificările şi în pagina unde schema logică a fost proiectată, în câmpul *Implementation* al fiecărei surse de intrare, se va adăuga stimulul dorit, ce a fost editat în prealabil.

2.2.1.3. Adăugarea şi editarea generatoarelor de puls (Digclock)

Sursele de semnal într-o schemă logică pot fi de tipul generatoarelor de puls (*DigClock*), când simularea se realizează în domeniul timpului. Generatoarele de puls se pot adauga din ferestrele de dialog *Place – Part* sau *Place – Pspice Component – Search*, prin căutarea după textul *DigClock*. După adaugarea unui generator de puls, se pot edita proprietățile acestuia:

- se va adăuga un nume specific;
- *ONTIME* - timpul în care semnalul are valoarea setată ca *OPPVAL*;
- *OFFTIME* - timpul în care semnalul are valoarea setată ca *STARTVAL*;
- *DELAY* - întărzierea;
- *STARTVAL* - tensiunea existentă ca ieşire la timpul $t = 0$;
- *OPPVAL* - valoare de operare a generatorului, mai exact, tensiunea care va fi folosită ca ieşire, după întârzierea setată sau când primul *OFFTIME* a fost finalizat (când începe *ONTIME*).

Valoarea setată pentru *OPPVAL* trebuie să fie negata valorii *STARTVAL*. Pentru editarea acestor valori se va da dublu clic pe proprietatea ce se doreşte a fi modificată. În Figura 2.3 se poate observa un generator de semnal şi proprietățile ce pot fi modificate.

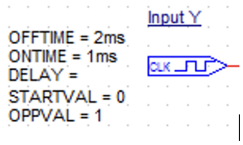


Fig. 2.3: Generator de puls.

2.2.1.4. Adăugarea conectorilor Off – Page

Pentru a reduce cablarea unei scheme, se pot utiliza conectorii *Off – Page*. Acest tip de conector se poate ataşa surselor de intrare şi apoi poate fi utilizat ca intrare pentru componentele schemei. Pentru a adăuga un astfel de conector, se va utiliza comanda *Place Off-Page Connector*. Un exemplu de utilizare se poate observa în Figura 2.4.

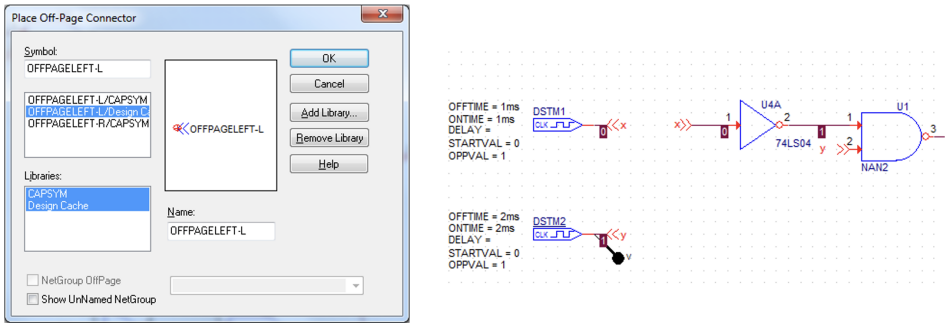


Fig. 2.4: Adăugare conector Off-Page.

2.2.1.4. Rularea simulării

Dacă toate setările necesare unei simulări au fost realizate, se poate rula simularea schemei logice realizate. Pentru acest lucru se va executa comanda *PSpice – Run*. Dacă se doreşte modificarea profilului de simulare creat, se poate utiliza comanda *PSpice – Edit Simulation Profile*.

Pentru a vizualiza rezultatul simulării se vor utiliza elemente de verificare ale semnalelor (voltmetre) (secțiunea 1.2.1.6.). Dacă se doreşte adăugarea unui nou semnal, direct în fereastra cu rezultatul simulării (*Pspice A/D*), se va utiliza comanda *Trace – Add Trace* şi se va selecta semnalul dorit. Un exemplu de ieşire al unei simulări se poate observa în Figura 2.5.

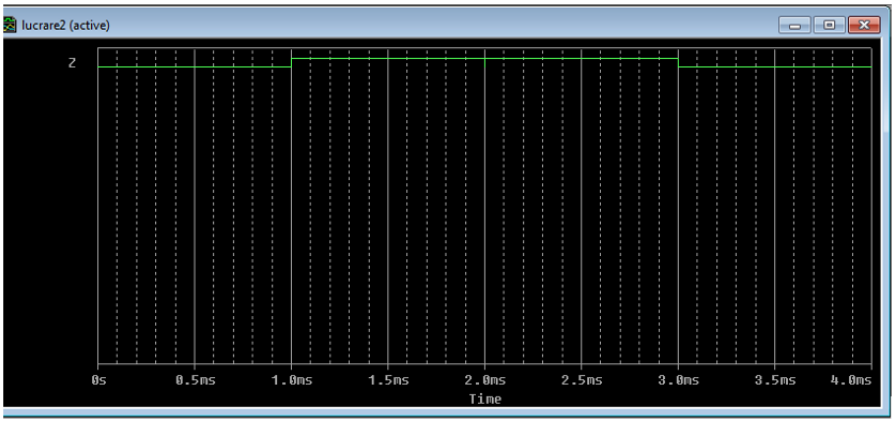


Fig. 2.5: Exemplu rezultat simulare.

2.2.2. Circuite multiplexoare

În cadrul acestui laborator, se va proiecta şi simula funcționarea unui multiplexor cu patru intrări şi o ieşire 4:1. În cele ce urmează se vor prezenta câteva detalii teoretice legate de funcționarea acestui circuit integrat.

Multiplexorul (MUX) reprezintă un circuit esențial în categoria integrării la scară medie (*MSI – Medium Scale Integration*). Acesta este adesea denumit *selector*, deoarece poate fi utilizat și ca un comutator pentru alegerea anumitor căi de semnal [2].

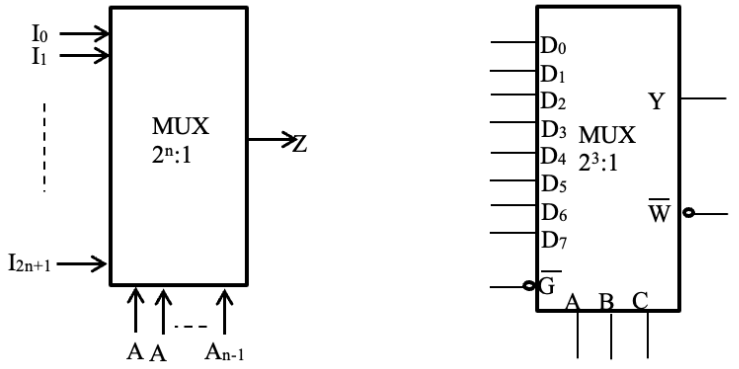
Multiplexoarele (MUX) sunt circuite logice combinaționale cu m intrări și o singură ieșire, utilizate pentru a permite transferul datelor de la una dintre intrări către ieșirea unică. Multiplexorul/selectorul are, în cazul general, 2^n intrări de date, $I_0, I_1 \ldots I_{2n-1}$, n intrări de selecție (adresă), $S_0, S_1 \ldots S_{n-1}$, şi o ieşire Z. Schema multiplexorului poate fi observată în Figura 2.6.

La un moment dat, ieșirea circuitului reflectă starea uneia dintre intrări I_k, unde indicele k este echivalentul zecimal al numărului binar reprezentat de stările 0 și 1 ale intrărilor de selecție::

$$k = S_{n-1}S_{n-2}...S_1S_0 \tag{3}$$

Ieşirea unui multiplexor selectează intrarea specificată de variabilele de adresă. Pentru a se garanta că la ieşire este activă întotdeauna doar intrarea

selectată, selecţia trebuie realizată numai după ce variabilele de adresă s-au stabilizat.



a. Circuit multiplexor. b. Simbolul de reprezentare al unui MUX 2*n* :1.

Fig. 2.6: Circuite de multiplexare.

În acest scop, multiplexoarele dispun de o intrare adițională, denumită intrare de autorizare, validare sau strobare, notată cu G. Această intrare are rolul de a controla funcţionarea circuitului, permițând:

- Inhibarea circuitului (oprirea selecţiei), în cazul în care semnalul de intrare este 1 logic;
- Dezinhibarea circuitului (permisiunea selecţiei), în cazul în care semnalul de intrare este 0 logic.

Intrarea de autorizare poate fi utilizată și pentru extinderea numărului de intrări. Acest lucru se realizează prin conectarea mai multor circuite de multiplexare în configurație ierarhică, unde semnalul $\overline{G}$ controlează activarea fiecărui multiplexor. Intrarea a fost desemnată cu $\overline{G}$ pentru că permite selecția când semnalul de intrare este 0 logic

2.3. Desfășurarea lucrării

În cadrul lucrării de laborator se va proiecta simula o funcție cu 3 variabil și un circut multiplexor 4:1.

2.3.1. Implementarea unei funcții cu trei variabile

Să se implementeze doar cu porți ȘI-NU, următoarea funcție booleeană, și să se simuleze circuitul digital, pornind de la forma minimă disjunctivă (*FMD*):

$$f^{FMD}(x_1, x_2, x_3) = x_1\overline{x}_2 + \overline{x}_1x_2 + x_2\overline{x}_3 + \overline{x}_2x_3 \tag{4}$$

Proiectarea unui circuit logic are ca scop principal reducerea numărului de circuite integrate utilizate, obținând astfel o soluție cu un cost total cât mai scăzut. În acest context, implementarea funcțiilor logice utilizând circuite integrate de tip SSI necesită o abordare în două etape:

- Minimizarea matematică a funcțiilor logice: Funcțiile sunt simplificate pentru a reduce complexitatea schemelor logice;
- Conversia în scheme logice echivalente: Expresiile minimizate sunt transformate în configurații fizice de porți logice, ceea ce constituie o problemă de optimizare influențată de experiența proiectantului și de restricțiile specifice. Aceste restricţi includ tipurile de porți logice disponibile, caracteristicile semnalelor (polaritate, amplitudine etc.), alegerea porților care minimizează numărul de capsule utilizate, numărul intrărilor disponibile pentru fiecare poartă, numărul de porți logice incluse într-o capsulă, numărul total de niveluri logice ale circuitului, întârzierile de propagare admisibile sau costul total al circuitelor implicate.

Implementarea funcțiilor logice utilizând doar porți de tip *ȘI-NU (NAND)* constituie, de cele mai multe ori, alternativa cu cea mai mare rentabilitate economică. Această abordare simplifică procesul de proiectare prin utilizarea unui număr limitat de tipuri de circuite integrate, dar menține un grad ridicat de eficiență a utilizării acestora.

Pentru a facilita proiectarea circuitului utilizând porți *ȘI-NU*, funcția logică f este rescrisă prin aplicarea teoremelor lui De Morgan (funcția 5) construindu-se un circuit echivalent.

Această abordare permite reducerea costurilor și respectarea restricțiilor impuse în proiectare, oferind totodată un circuit logic funcțional, optimizat și simplificat.

$$f^{FMD}(x_1, x_2, x_3) = \overline{\overline{x_1\overline{x}_2 + \overline{x}_1x_2 + x_2\overline{x}_3 + \overline{x}_2x_3}} = \overline{\overline{x_1\overline{x}_2} \cdot \overline{\overline{x}_1x_2} \cdot \overline{x_2\overline{x}_3} \cdot \overline{\overline{x}_2x_3}} \tag{5}$$

Se utilizează două tipuri de circuite integrate: porți logice *ŞI-NU* cu 2 intrări 7400 şi o poartă logică *ŞI-NU* cu 4 intrări 7420.

Pentru a efectua operația de complementare, la porțile logice *ŞI-NU* o intrare neutilizată trebuie conectată fie împreună cu o altă intrare utilizată, fie la un potențial corespunzător constantei logice 1. Detaliile de implementare sunt ilustrate în Figura 2.7.

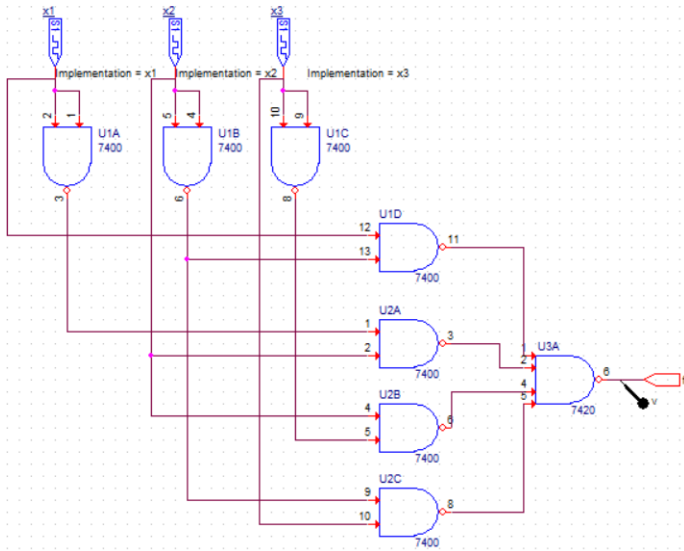


Fig. 2.7: Schema logică pentru funcția minimă disjunctivă definită de relația 5.

Următorul pas este construirea tabelului de adevăr și a diagramei de semnale pentru funcția dată. Pentru aceasta, se va ține seama de două reguli foarte importante, pentru evitarea apariției fenomenului de hazard, și anume:

- La o comutare este permisă schimbarea unei singure variabile. Tabelul de adevăr 1 prezintă și o variantă corectă de efectuare a comutărilor (există mai multe variante corecte);
- Intervalul de timp dintre două comutări trebuie să fie cel puțin egal cu timpul în care semnalul parcurge schema. Pentru schemele simple, cu maxim 3 niveluri logice și porți de tip *TTL* (*Tranzistor Tranzistor Logic*), acest timp este de circa 30 ms, și ca urmare, între două comutări trebuie să treacă un timp $t > 30$ ms.

Tabel 1: Tabel adevăr funcție cu 3 variabile

x_1	x_2	x_3	Ordinea comutărilor
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	3
1	0	0	6
1	0	1	7
1	1	0	5
1	1	1	8

Diagrama de semnale va arăta, așadar, ca în Figura 2.8. Informațiile din această diagramă trebuiesc introduse în *OrCAD* urmând pașii pentru editarea surselor de semnal descriși în secțiunea 2.2.1.2. Semnalul de intrare x_1 comută în valoarea 1 după $t = 200ms$, perioda acestuia fiind de $400ms$. Pentru setarea acestor valori, în fereastra descrisă în Figura 2.2c., se va completa câmpul *Period* cu valoarea $400ms$, iar câmpul *On time* cu valoarea $200ms$. În cazul semnalului x_2, acesta comută în valoarea 1 după $t = 100ms$, perioda acestuia fiind de $200ms$. Semnalul x_3 comută în valoarea 1 după $t = 50ms$, perioda acestuia fiind de $100ms$. Rezultatul simulării se poate observa în Figura 2.9.

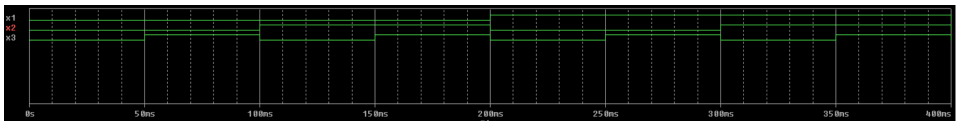


Fig. 2.8: Diagramă de semnale pentru funcția minimă disjunctivă definită de relația 5.

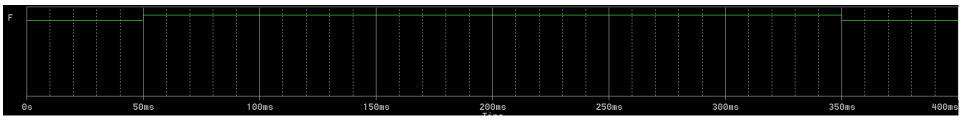


Fig. 2.9: Rezultatul simulării funcției definită de relația 5.

Vom edita apoi profilul simulării, unde la secțiunea *Analysis Type* vom selecta *Time domaine*, vom introduce durata simulării de $400ms$ (la secțiunea *Run to time*), iar la *Maximum step size* vom nota pasul simulării, adică intervalul de timp dintre două comutări, $50ms$. În final, în urma comenzii *Pspice - Run simulation*, va rula simularea.

2.3.2. Implementarea unui circuit de multiplexare 4:1

În continuare, se va proiecta și simula un circut multiplexor 4:1, pornind de la forma funcției acestuia :

$$Y = \overline{E}(\overline{A_1} \cdot \overline{A_0} \cdot I_0 + \overline{A_1} \cdot A_0 \cdot I_1 + A_1 \cdot \overline{A_0} \cdot I_2 + A_1 \cdot A_0 \cdot I_3) \qquad (6)$$

Tabel 2: Tabel de adevăr circuit multiplexor.

E	A0	A1	I0	I1	I2	I3	Y	Detalii
0	0	0	1	X	X	X	1	$Y = I0$
0	0	0	0	X	X	X	0	
0	0	1	X	1	X	X	1	$Y = I1$
0	0	1	X	0	X	X	0	
0	1	0	X	X	1	X	1	$Y = I2$
0	1	0	X	X	0	X	0	
0	1	1	X	X	X	1	1	$Y = I3$
0	1	1	X	X	X	0	0	

Componentele necesare proiectării circuitului sunt:

- 1 poartă *SAU* cu 4 intrări ;
- 4 porți *ŞI* cu 4 intrări;
- 3 porți de negare ($74LS04$).

Pentru a simula circuitul se vor utiliza tabelul de adevăr 2 şi diagrama de semnale din Figura 2.10. Intrarea de date I_0, conform diagramei de semnal, are perioada de $2s$ și comută în valoarea 1 după $t = 1s$. Intrarea de date I_1 are perioada de $2s$ și comută în valoarea 1 după $t = 2s$. Se poate observa că următoarele semnale de intare își dublează perioada și timpul de comutare. Semnalul de activare E, se va seta ca semnal continuu (de tip $Signal$), având valoarea logică 0 pe tot parcursul simulării. Pentru a verifica dacă simularea a avut rezultatul așteptat, se va utiliza un multiplexor $4 : 1$, componenta deja existentă în *OrCAD*, a cărei serie este 74153. Rezultatele trebuie sa fie identice.

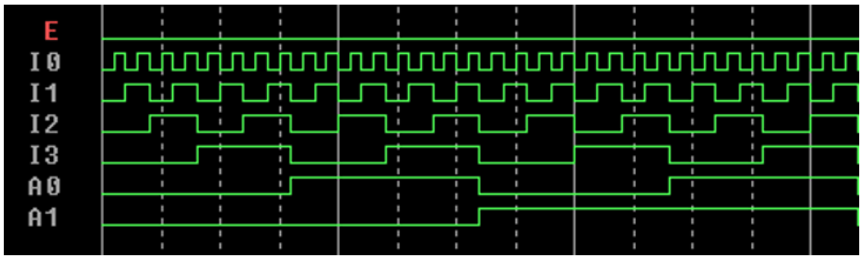


Fig. 2.10: Diagrama de semnal pentru semnalele de intrare.

Rezultatul obținut în urma simulării trebuie să coincidă cu cel din Figura 2.11.

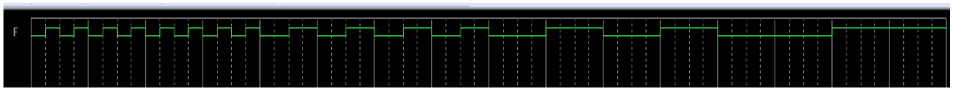


Fig. 2.11: Rezultatul simulării multiplexorului 4:1. În imagine se poate observa forma semnalului de ieșire.

2.4. Conținutul referatului

1. Prezentarea informațiilor tehnice legate de circuitele multiplexoare;
2. Parcurgerea paşilor pentru setările necesare simulării circuitul;
3. Proiectarea şi simularea schemeloe logice ale unui funcții cu trei variabile și ale unui multiplexor 4:1 conform cerințelor expuse.

2.5. Verificarea cunoştințelor

1. Cum funcționează un circuit multiplexor?
2. Ce tipuri de intrări are un circuit multiplexor?
3. Ce rol au intrările de selecție?
4. Ce rol are intrarea de strobare?
5. Care este procedura de conectare a mai multor circuite mulplexor într-o schemă logică?
6. Care este utilitatea conectorilor Off-Page?

Laborator 3
Realizarea schemelor logice utilizându-se blocurile ierarhice
Implementarea unui comparator pe 4 biți

3.1. Scopul lucrării

În această lucrare de laborator se propune proiectarea şi simularea unui comparator pe 4 biți utilizându-se blocuri ierarhice.

3.2. Prezentare teoretică

În cadrul acestui laborator, vor fi prezentate conceptele teoretice necesare pentru proiectarea schemelor utilizând *OrCAD Capture*. De asemenea, vor fi introduse noțiuni teoretice referitoare la funcționarea comparatoarelor numerice.

3.2.1. Proiectarea unui bloc ierarhic în *OrCAD Capture*

Utilizarea blocurilor ierarhice (*Hierarchical Blocks*) reprezintă un mod util de structurare a proiectelor mari, în mod special a celor care încep de la o diagramă bloc şi a celor în cadrul cărora se observă repetarea unui circuit comun. Pinii ierarhici din cadrul unui bloc ierarhic acționează ca punct de ataşare a conexiunilor electrice dintre mai multe astfel de blocuri sau cu alte elemente din schema bloc.

3.2.1.1. Adăugarea şi editarea unui bloc ierarhic

Pentru adăugarea unui bloc ierarhic se va utiliza comanada *Place - Hierarchical Block*. În urma acestei comenzi se va deschide o fereastră denumită *Place Hierarchical Block* unde se vor edita numele blocului, numele implementării şi tipul implementării, aşa cum se poate observa în Figura 3.1.

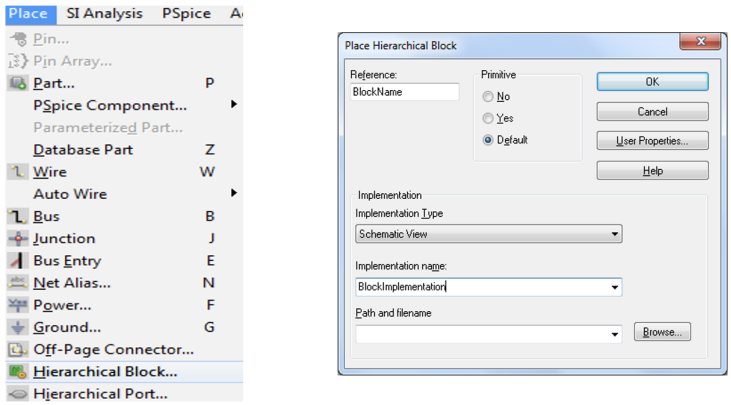


Fig. 3.1: Adăugarea şi editarea unui bloc ierarhic.

În momentul adăugării unui bloc ierarhic, trebuie modificate/adăugate proprietățile acestuia:

- În câmpul *Reference* se va adăuga numele blocului. Acesta trebuie să fie unic şi să nu conțină caractere speciale sau spații;
- În câmpul *Implementation name* se va adăuga numele implementării. Mai multe blocuri pot avea aceeaşi implementare;
- Se va alege opțiunea *Schematic View* ca *Implementation Type*. *Schematic view* indică faptul că implementarea ataşată este un director *schematic* (*schematic folder*).

După setarea acestor proprietăți, se va acționa butonul *OK* şi se va desena un chenar ce reprezintă blocul ierarhic. În acest moment blocul nu are pini de conectare (vezi Figura 3.2).

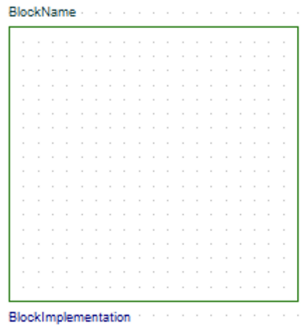


Fig. 3.2: Bloc ierarhic fără pini.

Pentru a realiza sau a accesa implementarea acestui bloc, se va da dublu clic pe blocul proiectat. O căsuță de dialog o să apară, aşa cum se poate observa în Figura 3.3, denumită *New Page in Schematic* care va permite setarea unui nume pentru o nouă pagină în proiectul creat. După setarea numelui se va da clic pe *OK* şi o să apară o nouă pagină pentru proiectarea circuitului în interiorul blocului. Această pagină o să conțină, practic, implementarea blocului ierarhic.

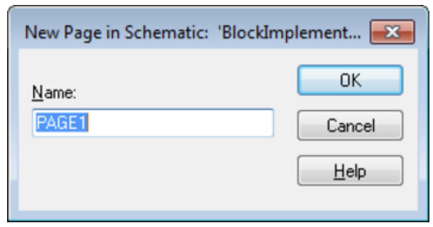


Fig. 3.3: Creare unei noi pagini pentru implementarea blocului ierarhic.

3.2.1.2. Adăugarea porturilor ierarhice

Intrările şi ieşirile blocurilor ierarhice sunt reprezentate de aşa numitele porturi ierarhice. Pentru a adăuga astfel de porturi se va alege opțiunea *Place – Hierarchical Ports*. În urma acestei comenzi se va deschide fereastra *Place Hierarchical Port*, din care se vor selecta porturi pentru ieşire (*PORTLEFT-L*) sau pentru intrare (*PORTRIGHT-R*), aşa cum se poate deduce din Figura 3.4. Porturile ierarhice vor fi utilizate în cadrul blocurilor ierarhice pentru transportul semnalelor. Sursele de semnal sau generatoarele de puls vor fi folosite, în continuare, ca semnale de intrare într-o schemă logică.

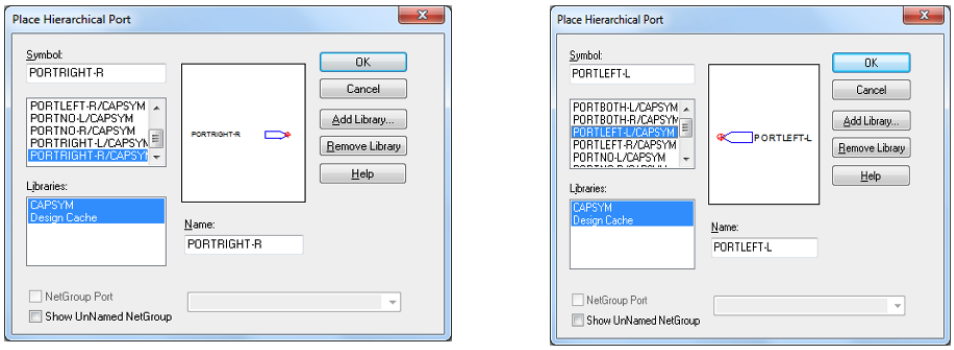


Fig. 3.4: Adăugare porturi ierarhice de intrare/ieşire.

3.2.1.3. Sincronizarea blocurilor ierarhice

În urma implementării blocului ierarhic se doreşte afişarea pinilor de intrare/ieşire pentru a furniza semnale de intrare, respectiv pentru a citi semnalul de ieşire. Pentru a realiza acest lucru, după ce implementarea a fost realizată si salvată, se va selecta blocul ierarhic, şi se va sincroniza cu implementarea sa apăsând clic dreapta şi alegându-se opțiunea *Syncronize Up*, la fel ca în Figura 3.5.

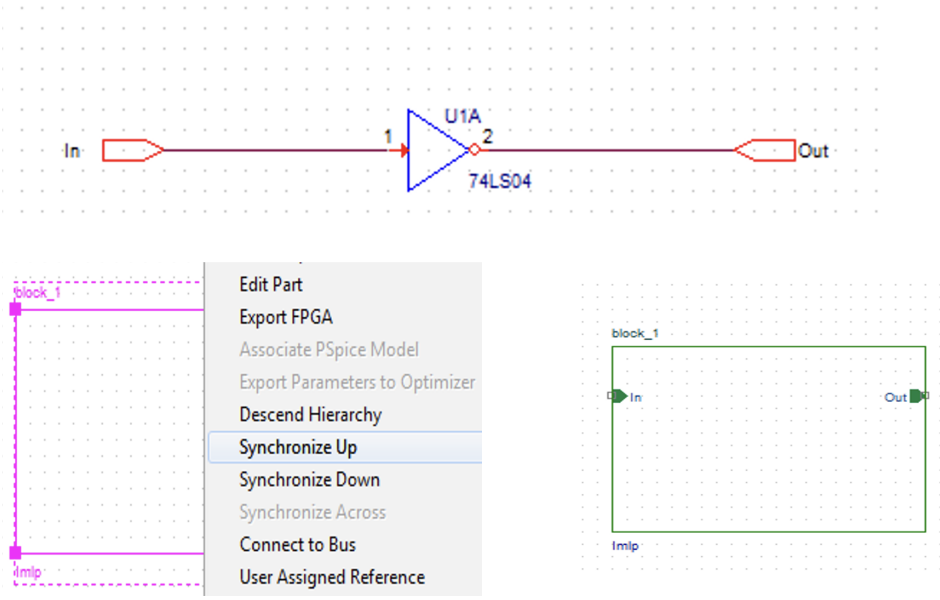


Fig. 3.5: Sincronizarea unui bloc ierarhic cu implementarea acestuia.

3.2.1.4. Generarea unei componente

Blocurile ierarhice pot fi utilizate pentru generarea componentelor ce pot fi folosite și în alte proiecte. Acest lucru poate fi foarte util atunci când dorim să reutilizăm un circuit proiectat într-un alt proiect. Pentru acest lucru se vor respecta următorii paşi:

1. În fereastra *Project Manager* se alege folder-ul corespunzător implementării blocului;
2. Din meniul *Tools* se va alege opțiunea *Generate Part*;
3. În fereastra *Generate Part* se urmează următorii pași (Figura 3.6):
   - În câmpul *Netlist/Source file* se alege locația proiectului (*design*) din care se va genera noua componentă. În exemplul din Figura 3.6 locația este ”C:/SPB_Data/lab3.dsn”;
   - În câmpul *Part Name* se va completa denumirea componentei;
   - Din lista *Netlist/source file type* se va selecta opțiunea *Capture Schematic Design*;
   - Se va specifica locația unde se va salva biblioteca ce conține noua componentă (de exemplu ”C:/SPB_Data/lab3.olb”).
4. După apasarea butonul *OK* va apărea o nouă fereastră. Din această nouă fereastră se va apăsa butonul *Save*.

În urma acestor paşi se va crea o nouă bibliotecă, ale cărei componente se vor putea utiliza în alte proiecte. Detalii pot fi observate şi în Figura 3.6.

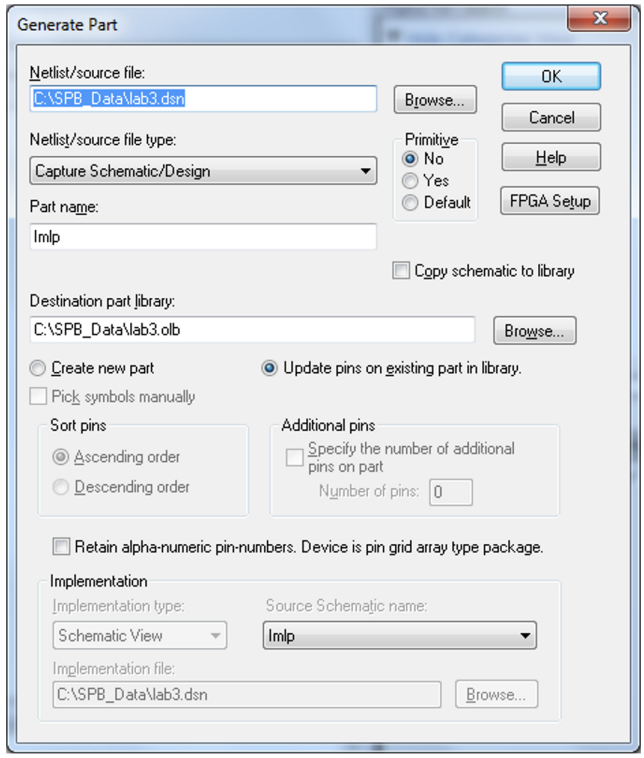


Fig. 3.6: Generarea unei noi componente.

3.2.2. Comparatoare numerice

Pentru a demonstra utilizarea blocurilor ierarhice, se va implementa un comparator numeric, circuit logic combinațional (CLC) ce va fi descris în această secțiune.

Comparatoarele numerice sunt CLC-uri utilizate pentru a determina relația dintre două numere binare A și B, fiecare format din n biți. Aceste circuite permit compararea paralelă a fiecărei perechi de biți corespondenți din cele două numere, stabilind relațiile posibile: $A < B$, $A = B$, sau $A > B$. Numerele binare A și B sunt aplicate la intrările circuitului, fiecare având câte n biți (A_n și B_n). Relația dintre cele două numere este indicată de stările logice ale celor trei ieșiri ale comparatorului, astfel: $Z_1 = 1$ indică $A < B$, $Z_2 = 1$ indică $A = B$, iar $Z_3 = 1$ arată relația $A > B$.

Un comparator cu 4 biți compară toate cele patru poziții ale numerelor $A(A_3, A_2, A_1, A_0)$ şi $B(B_3, B_2, B_1, B_0)$ simultan. Cifrele cele mai puțin semnificative sunt A_0 şi B_0. Schema bloc a comparatorului numeric, precum şi schema logică a comparatorului de un bit se pot observa în Figura 3.7, respectiv Figura 3.8. Schema logică prezentată în Figura 3.7 este celula de bază a comparatorului pentru numere cu mai mulți biți.

De asemenea, pentru a întelege modul de funcționare, se poate analiza tabelul de adevăr 3 al unui comparator de un bit.

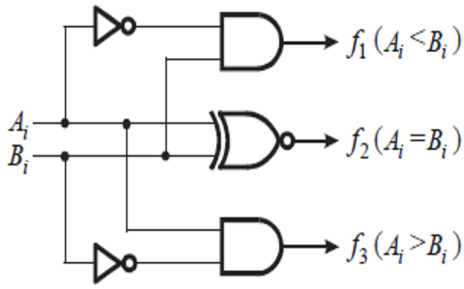


Fig. 3.7: Schema-bloc a unui comparator pe un bit.

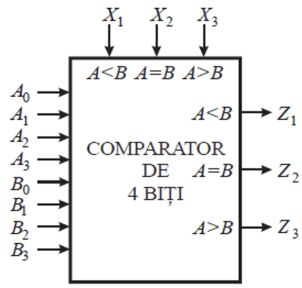


Fig. 3.8: Schema logică a comparatorului pe un bit.

3.3. Desfășurarea lucrării

În cadrul lucrării de laborator se va proiecta un comparator numeric şi apoi se va simula funcționării acestuia.

Tabel 3: Tabelul de adevăr al unui comparator pe un bit

A_i	B_i	$f_1(A_i < B_i)$	$f_2(A_i = B_i)$	$f_3(A_i > B_i)$
0	0	1	1	0
1	0	1	0	1
0	1	0	1	0
1	0	0	0	1
0	1	0	1	0
1	0	0	0	1

Se cere implementarea unui comparator iterativ pe 4 biti, utilizând-se blocuri ierarhice, pentru verificarea **condiției de egalitate** ($A = B$). Acest comparator va fi compus din 4 blocuri de comparare pe 1 bit interconectare. Comparatoarele de 1 bit, vor compara fiecare bit al numărului dorit (de exemplu A_0 cu B_0).

Ieşirea fiecărui bloc, denumită *EqOut* va reprezenta intrarea *EqIn* a blocului ce urmează. Ieşirea *EqOut* din ultimul bloc, reprezintă rezultatul comparației. Fiecare comparator de 1 bit va avea ecuația 7:

$$q = \overline{(\overline{x}y + x\overline{y})}EqIn \tag{7}$$

La intrarea primului bloc *EqIn* va avea valoarea 1.

Se vor compara numerele $A = 1100$ şi $B = 1100$. Apoi valoarea lui A se va modifica în $A = 1110$. După realizarea comparatorului cu blocuri ierarhice, se va compara rezultatul cu cel al circuitului integrat $74LS85$.

În Figura 3.9 este afișat un de bloc de comparație pentru un bit. Schema completă presupune legarea în serie a patru astfel de blocuri de comparație, deoarece A și B sunt reprezentate pe 4 biți.

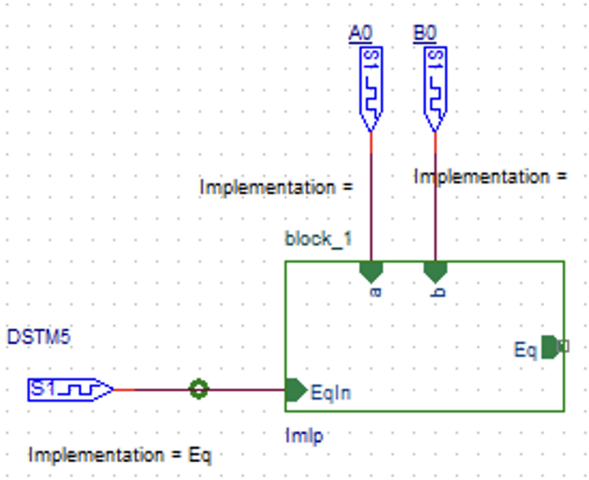


Fig. 3.9: Schema logică a comparatorului de un bit.

3.4. Conținutul referatului

1. Însuşirea cunoştințelor legate de proiectarea blocurilor ierarhice în *OrCAD Capture*;
2. Proiectarea unui comparator numeric utilizându-se blocuri ierarhice;
3. Rularea simulării pentru comparatorul pe 4 biți.

3.5. Verificarea cunoştințelor

1. Ce rol au comparatoarele numerice?
2. Ce operații realizează un comparator numeric?
3. Cum se realizează compararea a două numere?

Laborator 4
Simularea logică a fenomenelor de hazard static şi dinamic

4.1. Scopul lucrării

Această lucrare de laborator are ca obiectiv simularea logică a fenomenelor de hazard static și dinamic, aspecte fundamentale în analiza și proiectarea circuitelor digitale. Hazardul reprezintă un factor critic, în special în funcționarea circuitelor de comutare combinaționale, care constituie elemente componente ale circuitelor secvențiale asincrone. În acest context, este esențială înțelegerea profundă a cauzelor generatoare de hazard și a strategiilor eficiente pentru eliminarea acestuia, având în vedere impactul semnificativ asupra stabilității și performanței sistemelor digitale.

4.2. Prezentare teoretică

În analiza circuitelor logice combinaționale realizată în lucrările de laborator anterioare, s-a presupus că elementele logice utilizate pentru realizarea schemelor sunt ideale, iar comutarea lor se desfășoară fără întârziere. Totuși, în practică, din cauza timpului finit de comutare, fiecare modul logic generează întârzieri inevitabile, iar funcționarea în condiții reale poate genera probleme subtile care pot conduce la erori operaționale. Aceste întârzieri, manifestate de-a lungul lanțului dintre intrări și ieșiri, pot determina, la un moment dat, o discrepanță între starea ieșirii circuitului și starea efectivă a intrărilor. În astfel de situații, circuitul poate prezenta, pentru intervale foarte scurte de timp, un comportament incorect, fenomen ce conduce la apariția unor fluctuații nedorite ale semnalului sub formă de impulsuri tranzitorii (cunoscute sub denumirea de *glich-uri*). Acest comportament este definit în literatură ca fiind *hazard* [2]..

CLC-urile pot prezenta două tipuri diferite de hazard: *hazardul static* şi *hazardul dinamic*.

Hazardul static se manifestă atunci când o schimbare a stării intrărilor determină, pentru un interval scurt de timp, o modificare nejustificată a stării ieșirii. Această anomalie apare chiar dacă, logic, starea ieșirii ar trebui să rămână neschimbată în raport cu noua configurație a intrărilor.

Hazardul dinamic se produce atunci când, în urma modificării stării intrărilor, ieșirea trebuie să treacă într-o nouă stare, conform logicii circuitului. Cu toate acestea, tranziția nu este directă, ci implică o serie de oscilații între starea anterioară și cea nouă înainte de a se stabiliza.

4.2.1. Hazardul static

Hazardul static se manifestă printr-o modificare tranzitorie neașteptată a stării uneia sau mai multor ieșiri ale circuitului, pentru un interval de timp foarte scurt. Această modificare este contrară comportamentului logic așteptat și revine ulterior la starea inițială. Se disting două tipuri de hazard static:

- *hazard static în unități*: apare atunci când ieșirea, care ar trebui să rămână în starea logică 1, trece temporar în 0;
- *hazard static în zerouri*: apare atunci când ieșirea, care ar trebui să rămână în starea logică 0, trece temporar în 1.

Se poate arăta că hazardul static apare dacă expresia logică a unui semnal într-un punct al circuitului poate fi redusă la forma $A + A$ sau $A \cdot A$. Spre exemplu, pentru funcția logică $f(A, B, C) = AB + AC$ există potențial de hazard deoarece, în cazul $B = C = 1$ expresia se reduce la $A + A$, ceea ce favorizează apariția fenomenului.

Se analizează astfel un circuit logic combinațional (CLC) a cărui funcționare evidențiază apariția hazardului static. În cazul funcției 8, implementată de circuitul din Figura 4.1, se observă că, dacă elementele componente ale circuitului prezintă întârzieri diferite, schimbarea valorii intrărilor x_1, x_2 şi x_3 din 110 în 111 poate determina apariția hazardului static. Această situație apare atunci când $t_{p1} > t_{p2}$, unde:

- t_{p1} timpul de propagare al porții *ȘI-NU* notată cu 1;

- t_{p2} timpul de propagare al porții *ȘI-NU* notată cu 2.

$$f^{FMD}(x_1, x_2, x_3) = x_1\overline{x_3} + x_2x_3 \tag{8}$$

Diagramele de semnal din Figura 4.1 evidențiază existența hazardului static în circuitul analizat, ilustrând variațiile temporale ale semnalelor la ieșirile porților circuitului. Aceste diagrame confirmă impactul întârzierilor diferențiale în determinarea comportamentului tranzitoriu al circuitului.

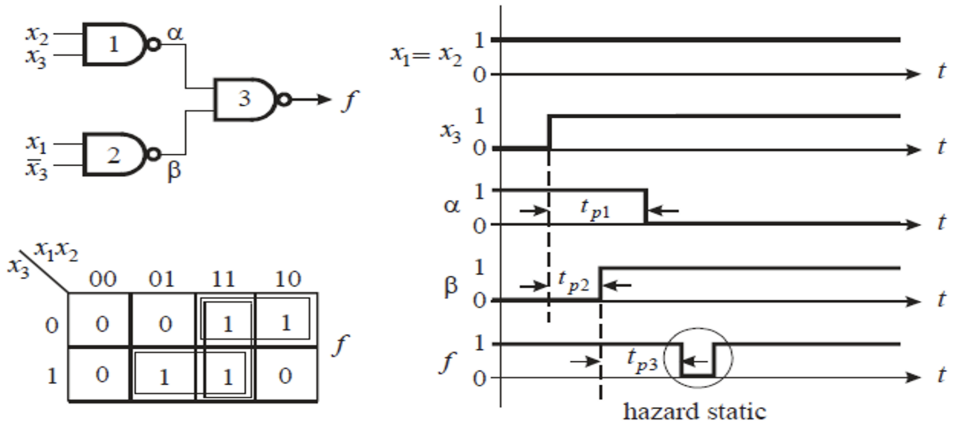


Fig. 4.1: Evidențierea hazardului static al circuitului logic combinațional din exemplu.

4.2.2. Hazardul dinamic

Hazardul dinamic apare atunci când modificarea stării intrărilor determină, conform logicii circuitului, o schimbare a stării ieșirii, însă această tranziție nu se realizează direct, ci este însoțită de un număr de oscilații tranzitorii între starea veche și cea nouă. Acest tip de hazard este frecvent întâlnit în circuitele de comutare cu mai multe niveluri logice, în special în porțiunile de circuit susceptibile la hazard static [2].

Se poate arăta că un circuit care implementează o funcție în formă canonică disjunctivă și care nu prezintă hazard static în unități este, de asemenea, lipsit de alte forme de hazard static sau dinamic.

Pentru a ilustra acest fenomen, se consideră un circuit logic combinațional a cărui schemă este prezentată în Figura 4.2. Circuitul realizează următoarea funcție:

$$f^{FMD}(x_1, x_2, x_3, x_4, x_5) = (x_2\overline{x_4} + x_3x_4)(x_1 + x_5) \tag{9}$$

Se consideră că întârzierile induse de porțile logice sunt diferite, astfel încât $t_{p1} < t_{p2}$, și $t_{p4} > t_{p1} + t_{p3}$, unde cu t_{pi} reprezintă timpul de propagare prin poarta logică i.

Astfel, hazardul dinamic apare la ieșirea circuitului f atunci când intrările x_1, x_2, x_3, x_4, x_5 își schimbă valoarea logică din 01101 în 01110. Manifestarea hazardului dinamic la ieșirea finală f este influențată și de hazardul static prezent la ieșirea intermediară γ, amplificând efectele tranzitorii.

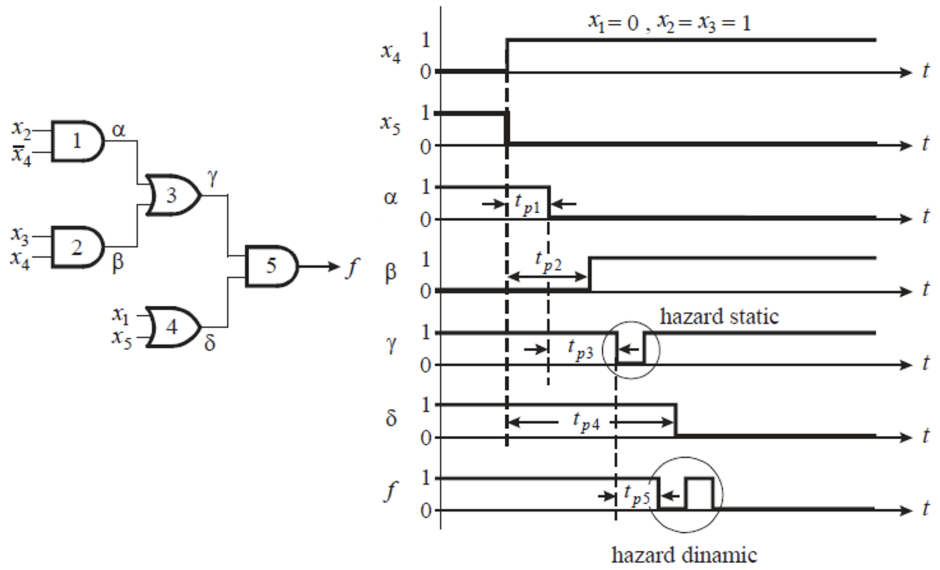


Fig. 4.2: Evidențierea hazardului dinamic al circuitului logic combinațional din exemplu.

4.2.3. Eliminarea hazardului

Hazardul reprezintă o sursă potențială de erori în funcționarea circuitelor de comutare, motiv pentru care identificarea și abordarea sa încă din etapa de

proiectare este esențială. Prevenirea acestui fenomen asigură o funcționare corectă și stabilă a circuitului.

Pentru simplificarea analizei, se presupune că, într-un anumit moment, doar o singură variabilă de intrare a circuitului poate suferi modificări. În aceste condiții, combinația curentă de valori binare ale intrărilor se transformă întotdeauna într-o combinație binară adiacentă.

Se poate dovedi că un circuit logic combinațional, implementat cu o schemă pe două niveluri de porți *ŞI* şi *SAU*, respectiv *ŞI-NU* nu are hazard dacă îndeplinește următoarea condiție: orice pereche de combinații adiacente ale valorilor variabilelor de intrare, pentru care funcția circuitului este egală cu 1, trebuie să fie acoperită de cel puțin un termen al expresiei minimizate disjunctive a funcției respective.

Funcția logică reprezentată pe diagrama Karnaugh din Figura 4.1, are ca variabile de intrare x_1, x_2 şi x_3 iar perechile de valori binare adiacente sunt: (110 şi 100), (110 şi 111), (111 şi 011). Analizând expresia logică 8, rezultă că:

- perechea (110 şi 100) este acoperită de termenul $x_1\overline{x_3}$;
- perechea (111 şi 011) este acoperită de termenul x_2x_3 ;
- perechea (110 şi 111) nu este acoperită, ceea ce poate duce la apariția hazardului.

Pentru a elimina acest risc, se introduce termenul redundant x_1x_2, care acoperă perechea (110 şi 111). Noua expresie a funcției devine:

$$f^{FMD}(x_1, x_2, x_3) = x_1\overline{x_3} + x_2x_3 + x_1x_2 \tag{10}$$

Schema circuitului care implementează funcția 10 este ilustrată în Figura 4.3.

Adăugarea termenilor redundanți în expresia unei funcții f, îngăduie stabilizarea ieșirilor circuitului pe toată durata tranziției semnalelor de intrare între starea inițială și cea finală.

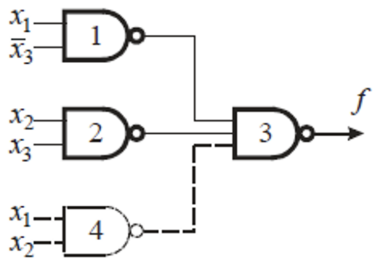


Fig. 4.3: Circuit logic fără hazard static.

În general, această metodă nu afectează semnificativ timpul total de propagare al circuitului. Cu toate acestea, adăugarea termenilor redundanți sporește complexitatea schemei logice și poate introduce alte forme de hazard. Prin urmare, această abordare trebuie utilizată doar atunci când hazardul are potențialul de a genera defecțiuni critice.

Pe lângă utilizarea termenilor redundanți, hazardul în circuitele logice combinaționale poate fi minimizat prin:

- adaugarea unor elemente de întârziere în schemă: acestea sincronizează apariția semnalelor la intrările porților logice, eliminând discrepanțele cauzate de timpii de propagare inegali. Totuși, această metodă crește timpul total de propagare al circuitului și implică calcule complexe, deoarece timpii de propagare ai componentelor variază;
- adaugarea unor elemente de întârziere pasive: elementele de întârziere pasive plasate la ieșirea circuitului blochează impulsurile false generate de comutările neintenționate. Deși această soluție este relativ simplă, poate afecta viteza globală de funcționare a circuitului.

4.3. Desfășurarea lucrării

În cadrul lucrării de laborator se va simula un circuit logic combinațional pentru a observa fenomenul de hazard. Pentru a proiecta şi simula acest circuit se va utiliza o diagrama Karnaugh pentru a determina forma minimă disjunctivă a funcției.

4.3.1. Determinarea formei minime disjunctive

Primul pas pentru realizarea schemei în *Orcad Capture* este determinarea formei minime disjunctive a funcției ce urmează a fi implementată. Pentru a realiza acest lucru se va porni de la diagrama Karnaugh, diagrama ce se regăseşte în Tabelul 4.

Tabel 4: Diagrama Karnaugh pentru determinarea formei minime disjunctive

$x_4x_5/x_1x_2x_3$	000	001	011	010	110	111	101	100
00	0	0	0	0	0	0	0	0
01	0	0	1	0	1	1	0	1
11	0	0	1	0	1	1	0	1
10	0	0	1	0	1	1	0	1

Reamintim algoritmul de aflare a formei minime disjunctive a unei funcții booleene:

1. Se caută implicanții primi ai funcției. Pentru asta, compartimentele notate cu 1 pe diagrama Karnaugh se grupează astfel încât să se obțină subcuburi cu dimensiunea cea mai mare posibilă. O parte, dar nu toate dintre compartimentele unui subcub, poate face parte din mai multe subcuburi, de diverse dimensiuni;
2. Se scriu termenii normali ce corespund implicanților primi și conțin partea lor comună;
3. Se caută implicanții primi esențiali. Pentru aceasta, se găsesc pe diagramă acele compartimente incluse într-un singur implicant prim. Acel implicant devine implicant prim esențial;
4. Celulele marcate cu 1 rămase neacoperite de implicanții primi esențiali se caută să se acopere folosindu-se un număr cât mai mic din implicanții funcției care au mai rămas;
5. Se scrie forma minimă a funcției, ca suma implicanților primi esențiali și implicanților primi aleși la punctul 4.

După obținerea formei minime disjunctive se va reduce aritmetic forma acesteia, pentru a reduce numărul de componente necesare circuitului logic combinațional.

4.3.2. Simularea schemei în OrCAD PSpice

Utilizându-se pachetul de programe *OrCAD*, se va proiecta şi simula schema ce implementează funcția determinată la punctul 4.3.1. Schema rezultată trebuie să fie similară cu cea din Figura 4.4.

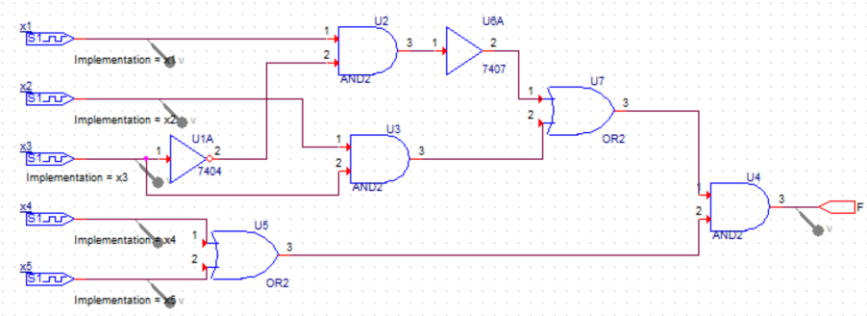


Fig. 4.4: Schema logică simulare hazard.

Pentru a putea introduce fenomenul de hazard este necesară setarea unei întârzieri în profilul de simulare. Pentru acest lucru, la crearea profilului de simulare, din tab-ul *Options* se va selecta categoria *Gate-level Simulation*, iar proprietatea *Timing Mode*, se va seta ca *Worst-Case Scenario*, conform Figurii 4.5. Acest lucru se realizează pentru a se încerca simularea unui circuit real, în dauna unui circuit optim, cum se întâlneşte în mod normal în acest mediu de simulare.

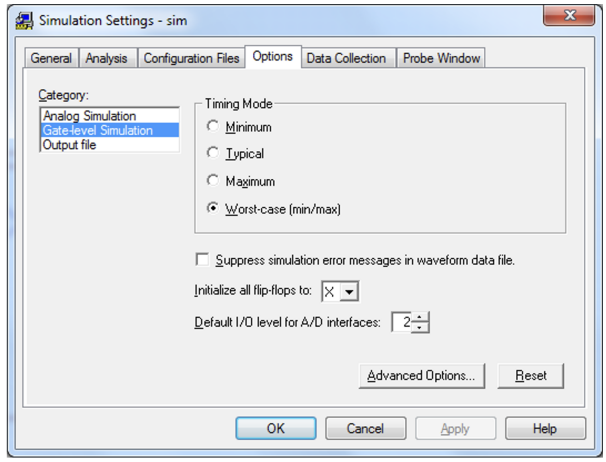


Fig. 4.5: Setare întârziere într-un profil de simulare.

De asemenea, în circuit se va introduce un buffer, pentru adăugarea unei întârzieri suplimentare. Acest buffer se introduce pe nivelul *ŞI* logic al schemei. Această componentă se găseşte în *OrCAD* cu seria 7407. Pentru a seta nivelul de întârziere a buffer-ului, se va modifica proprietatea *MNTYMXDLY*, cu una din cifrele cuprinse între 1 şi 4, unde 1 reprezintă întârziere minimă, iar 4,"cel mai rău caz probabi". În acest laborator vom seta valoarea 4. Accesarea proprietăților buffer-ului se realizează prin dublu-clic asupra componentei.

Pentru a seta semnalele de intrare necesare simulării, se va utiliza diagrama de semnale din Figura 4.6. Intrările x_1, x_2, x_5, sunt de semnale continue, cu valori logice $x_1 = 1, x_2 = 1$, repectiv $x_5 = 0$. Semnalele x_3 și x_4 vor fi setate de tip *clock*, valoare logică modificându-se dupa $10ns$, perioda semnalului fiind de $100ns$.

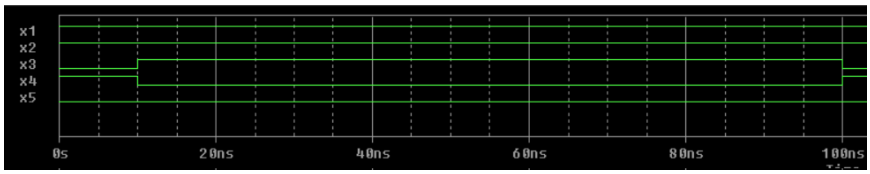


Fig. 4.6: Diagrama semnale de intrare.

Rezultatul simulării, unde hazardul este prezent la început, se poate observa în Figura 4.7.

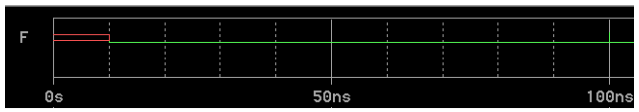


Fig. 4.7: Diagrama semnale de intrare.

După realizarea acestei simulări, unde hazardul ar trebui să fie prezent, se va încerca eliminarea acestuia. Pentru acest lucru se va adauga un termen redundant, conform teoriei descrise în secțiunea 4.2.3.

4.4. Conținutul referatului

1. Deducerea formei minime disjunctive utilizându-se diagrama Karnaugh furnizată;
2. Proiectarea şi simularea schemei logice utilizându-se forma minimă obținută;
3. Eliminarea hazardului schemei logice.

4.5. Verificarea cunoştințelor

1. Cum poate fi definit fenomenul de hazard?
2. Care sunt motivele apariției fenomenului de hazard?
3. Ce tipuri de hazard cunoaşteți?
4. Cum poate fi definit hazardul static?
5. Cum poate fi definit hazardul dinamic?
6. Care sunt metodele de eliminare ale hazardului dintr-o schemă logică?
7. Care sunt dezavantajele adăugării unui termen redundant într-un circuit?

Laborator 5
Sumator binar pe 2 biți

5.1. Scopul lucrării

În acestă lucrare de laborator se va proiecta şi simula un sumator pe doi biți. Lucrarea cuprinde o parte teoretică, ce va prezenta funcționarea unui sumator binar, urmată de implementare versiunii pe doi biți în *OrCAD Capture CIS Lite*.

5.2. Prezentare teoretică

În domeniul electronicii, un sumator reprezintă un circuit digital (circuit logic integrat) utilizat pentru efectuarea operațiilor de adunare asupra a două sau mai multe valori numerice. În arhitectura majorității calculatoarelor, sumatoarele joacă un rol esențial, fiind integrate nu doar în unitățile aritmetico-logice (UAL, eng. ALU), ci și în alte componente ale unității centrale de procesare (CPU), unde sunt utilizate pentru operații precum calcularea adreselor, determinarea indicilor tabelari și altele.

Un sumator digital este capabil să efectueze atât operații de adunare, cât și de scădere, prin utilizarea reprezentărilor numerice în cod invers sau cod complement. Această versatilitate se bazează pe folosirea aceleiași funcții de adunare, dar aplicată unor valori exprimate cu exponent negativ. Deși pot fi proiectate sumatoare pentru diverse sisteme de reprezentare numerică, cele mai frecvente implementări sunt optimizate pentru operarea cu numere binare, datorită prevalenței acestora în calculul digital.

Un sumator complet adună două numere binare și poate prelua o valoare de transport dintr-o adunare anterioară, generând o sumă (S) și, eventual, un transport de ieșire. Acest tip de sumator adună trei numere binare de 1-bit și produce rezultatul prin două ieșiri binare. Intrările sunt notate frecvent cu A, B (numerele de adunat) și T_{in} (eng. C_{in}), iar ieșirile cu S și $T_{ieș}$ (eng. C_{out}). Tabela de adevăr a circuitului este prezentată în Tabelul 5.

Intrări		Ieșiri		
A	B	T_{in}	$T_{ieș}$	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

Tabel 5: Tabel de adevăr sumator pe 2 biți

5.2.1. Adunarea a trei biți

La adunarea a două numere binare trebuie realizată suma a trei biți, doi care provin de la operanzi şi al treilea reprezentând bitul de depăşire sau transport (*carry*). Rezultatul sumei pe trei biți poate fi urmărit în tabela de adevăr 5. Rezultatul adunării a două numere pe doi biți, $A = 11$ (numărul 3 în zecimal) şi $B = 10$ (numărul 2 în zecimal) este:

$$A + B = 11 + 10 = 101 \tag{11}$$

, unde 101 este 5 in zecimal. Bitul de transport (*carry*), va fi 0 la început.

Cu alte cuvinte, pe un rang k, trebuie efectuată operația de adunare, utilizându-se relația 12:

$$A_k + B_k + T_{k-1} = T_k S_k \tag{12}$$

, unde:

- A_k şi B_k sunt biți de rang k ai operanzilor A şi B;

- T_{k-1} este bitul de transport rezultat din adunarea de pe rangul precedent $k-1$;
- S_k este bitul sumă ce se scrie la rangul k;
- T_k este bitul de trasport ce rezultă din adunarea de pe rangul k şi se va aduna la rangul următor.

5.2.2. Ecuațiile sumatorului complet

Un sumator complet are la intrare 3 biți, iar la ieşire 2 biți (suma şi bitul de trasport). Ecuațiile sumatorului complet sunt:

$$\begin{aligned} S &= \overline{A} \cdot \overline{B} \cdot T_{in} + A \cdot \overline{B} \cdot \overline{T_{in}} + \overline{A} \cdot B \cdot \overline{T_{in}} + A \cdot B \cdot T_{in} \\ &= \overline{A}(\overline{B} \cdot T_{in} + B \cdot \overline{T_{in}}) + A(\overline{B} \cdot \overline{T_{in}} + B \cdot T_{in}) \\ &= \overline{A}(B \oplus T_{in}) + A(B \oplus \overline{T_{in}}) \\ &= A \oplus B \oplus T_{in}. \end{aligned} \tag{13}$$

generalizând se deduce:

$$S_k = A_k \oplus B_k \oplus T_{k-1}. \tag{14}$$

Pentru bitul de transport, plecând de la ecuația:

$$\begin{aligned} T_{out} &= \overline{A} \cdot B \cdot T_{in} + A \cdot \overline{B} \cdot T_{in} + A \cdot B \cdot \overline{T_{in}} + A \cdot B \cdot T_{in} \\ &= (\overline{A} \cdot B + A \cdot \overline{B})T_{in} + AB(\overline{T_{in}} \cdot T_{in}) \\ &= (A \oplus B)T_{in} + AB. \end{aligned} \tag{15}$$

se deduce:

$$T_{out(k)} = (A_k \oplus B_k)T_{in(k-1)} + AB. \tag{16}$$

5.3. Desfășurarea lucrării

În cadrul lucrării de laborator, pe baza relațiilor 14 şi 16 se va proiecta şi simula schema sumatorului elementar ce realizează adunarea pe un rang.

Pentru implementare se vor utiliza blocurile ierarhice, prezentate în cadrul laboratorului 3.

Aşadar, se va proiecta şi simula un sumator pe 2 biți, care adună două numere. Se pot utiliza numerele $A = 10$ (valoarea 2 în zecimal) şi $B = 11$ (valoarea 3 în zecimal). Rezultatul adunării va fi $S = 101$ (valoarea 5 în zecimal). Implementarea schemei bloc se va realiza utilizându-se blocurile ierarhice.

Schema sumatorului elementar poate fi construită din două scheme identice, denumite semisumatoare. Semisumatorul realizează adunarea biților pe un rang fără a lua în considerare şi bitul de transport rezultat din adunarea de pe rangul precedent. Se permite astfel o reprezentare a schemei pe două niveluri ierarhice diferite, cum se poate observa în Figura 5.1.

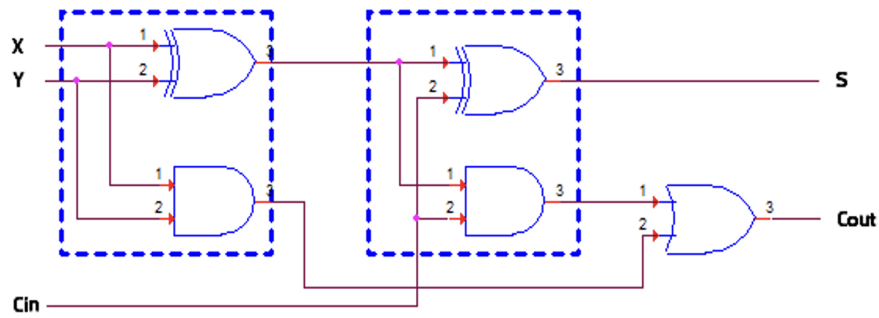


Fig. 5.1: Sumator elementar compus din două semisumatoare.

Implementarea în *Orcad Capture CIS Lite* a unui semisumator se poate observa în Figura 5.2, unde *CARRY* este bitul de transport. Această schemă este considerată ca fiind nivelul ierarhic cel mai de jos.

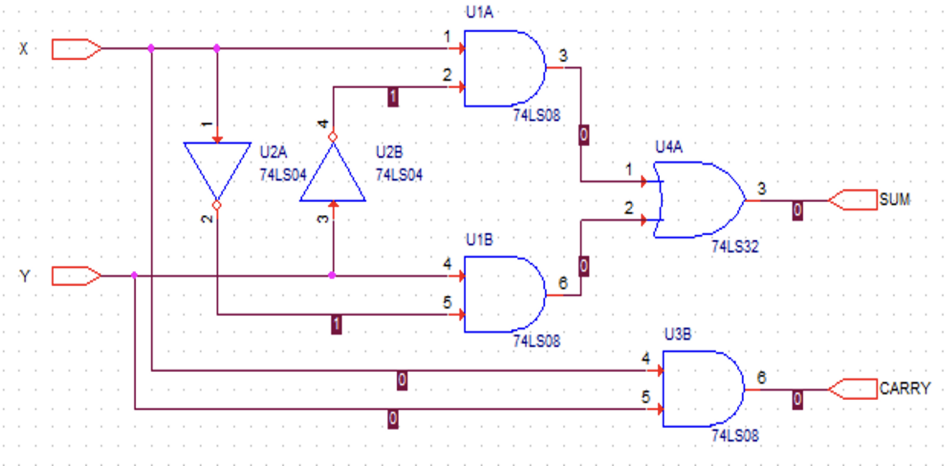


Fig. 5.2: Schema logică a semisumatorului.

Două scheme identice de semisumator, ca cea din Figura 5.2 se interconectează realizând o schemă de sumator complet, cum se poate observa în Figura 5.3. Semisumatorul va fi implementat ca bloc ierarhic, şi poate fi utilizat de câte ori este necesar.

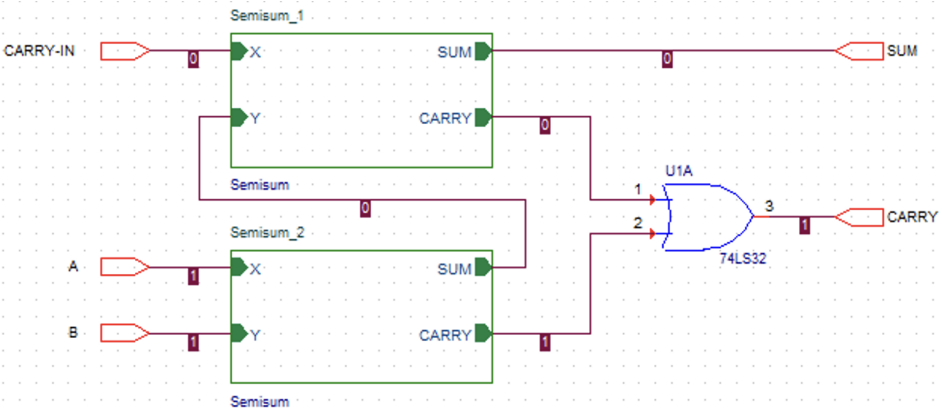


Fig. 5.3: Schema bloc a sumatorului complet.

Pentru a proiecta şi simula un sumator pe doi biți, două sumatoare complete trebuie interconectate. Bitul de transport rezultat din adunarea de pe primul rang va fi folosit ca intrare în cel de-al doilea sumator. Blocul ce conține sumatorul pe un bit poate fi vizualizat în Figura 5.4. Pentru a edita

semnalele de intrare se va utiliza tabelul de adevăr 5 al unui sumator binar pe doi biți.

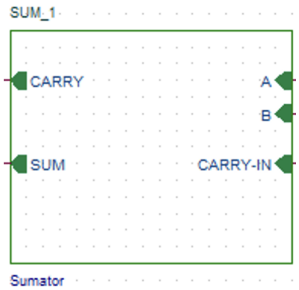


Fig. 5.4: Bloc ierarhic ce conține schema unui sumator complet.

5.4. Conținutul referatului

1. Însuşirea cunoştințelor legate de funcționarea sumatorul binar;
2. Proiectarea şi simularea schemei logice a unui sumator pe 2 biți;
3. Verificarea corectitudinii schemei prin validarea simulării.

5.5. Verificarea cunoştințelor

1. Cum se poate defini un sumator binar?
2. Ce operații aritmetice poate să execute un sumator binar?
3. Ce este un sumator complet?
4. Câti biți are la intrare un sumator binar pe doi biți şi ce reprezintă aceştia?
5. Câți biți are la ieşire un sumator pe 2 biți şi ce reprezintă aceştia?
6. Descrieți operația de adunare a trei biți.

Laborator 6
Multiplicator binar pe 2 biți

6.1. Scopul lucrării

În această lucrare de laborator se vor prezenta noțiuni introductive legate de funcționarea multiplicatoarelor binare. De asemenea, se propune proiectarea și simularea unui multiplicator pe 2 biți utilizând-se pachetul de programe *OrCAD*.

6.2. Prezentare teoretică

Un multiplicator binar este un circuit logic combinațional, utilizat în sistemele digitale, pentru a efectua multiplicarea (înmulțirea) a două numere binare. Aceste circuite sunt folosite în diverse aplicații, în special în domeniul procesării semnalelor.

În comparație cu operațiile de adunare şi scădere, înmulțirea este un proces complex ce include calcularea produselor parțiale și însumarea acestora.

Circuitele specializate în efectuarea rapidă a operației de înmulțire a numerelor binare sunt, de regulă, concepute utilizând principiul arhitecturii iterative.

6.2.1. Operația de multiplicare

Prin înmulțirea a două numere binare, fiecare având patru poziții, notate A_0, A_1, A_2, A_3 și B_0, B_1, B_2, B_3, rezultă un produs reprezentat pe opt biți, notați P_0, P_1, P_2, P_3, P_4, P_5, P_6 și P_7. Simbolic, operația de înmulțire se desfășoară conform regulii prezentate în Tabelul 6.

Realizarea celor opt funcții $P_i(i = 0, .., 7)$, direct, ca funcții dependente de cele opt variabile $(A0 \div A3)$ şi $(B0 \div B3)$ prezintă un grad ridicat de complexitate. În consecință, procesul de înmulțire se bazează pe

				A_3	A_2	A_1	A_0	$\times$
				B_3	B_2	B_1	B_0	
				A_3B_0	A_2B_0	A_1B_0	A_0B_0	
			A_3B_1	A_2B_1	A_1B_1	A_0B_1		
		A_3B_2	A_2B_2	A_1B_2	A_0B_2			
	A_3B_3	A_2B_3	A_1B_3	A_0B_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

Tabel 6: Regula înmulțirii binare [2]

adăugarea succesivă a deînmulțitului la produsul parțial, utilizând fie o abordare combinațională, sub forma unui circuit iterativ alcătuit din sumatoare paralele, fie o abordare secvențială, bazată pe un singur sumator paralel.

În analiza de față se consideră prima variantă, iar schema logică a circuitului recursiv destinat înmulțirii binare este ilustrată în Figura 6.1.

În cadrul acestui circuit, primul rând de celule efectuează însumarea produsului dintre deînmulțitul ($A_3A_2A_1A_0$) și cea mai puțin semnificativă cifră a înmulțitorului, B_0, cu produsul parțial inițial (zero), generând astfel un nou produs parțial. Fiecare rând ulterior de celule adaugă la produsul parțial obținut anterior deînmulțitul înmulțit cu următoarea cifră a înmulțitorului (B_1, B_2, ...), continuând procesul iterativ până la obținerea rezultatului final [2].

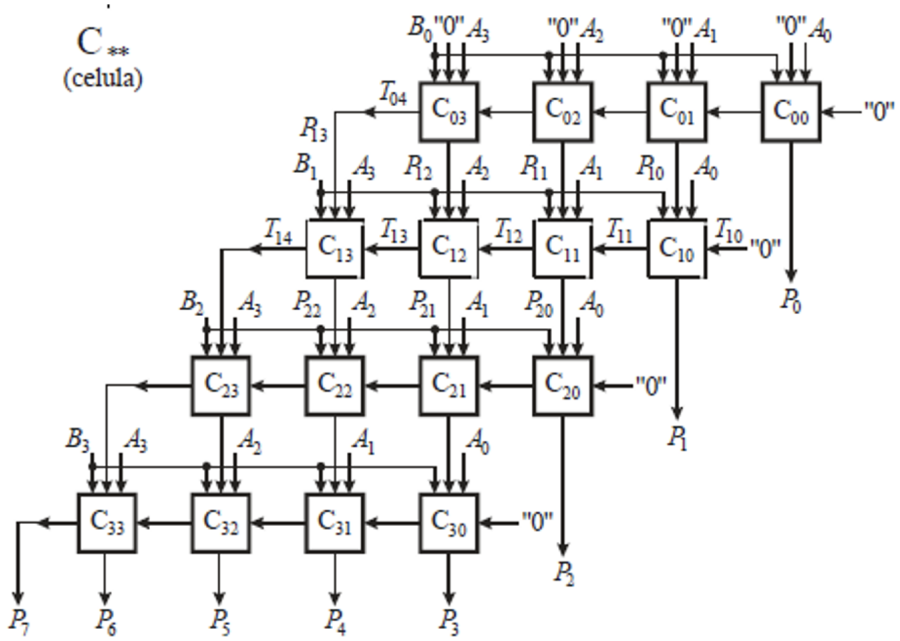


Fig. 6.1: Circuit iterativ pentru înmulțirea a două numere reprezentate pe 4 biți.

În scopul proiectării unui multiplicator combinațional destinat operării asupra cuvintelor binare de lungime n biți, este necesară interconectarea a n^2 celule într-o arhitectură matricială bidimensională, structură ce reflectă principiul algoritmic al înmulțirii binare a două cuvinte de n biți. Fiecare celulă a circuitului recursiv ilustrat în Figura 6.1, funcționează ca un modul fundamental, având rolul de a realiza înmulțirea elementară între doi biți ai operanzilor.

Pentru a asigura o implementare eficientă a procesului de înmulțire, acest circuit trebuie să respecte următoarele cerințe esențiale:

- să execute operația elementară de înmulțire între doi biți;
- să integreze mecanismele necesare pentru efectuarea operației de adunare, esențială în acumularea produselor parțiale specifice algoritmului de înmulțire a numerelor pe mai mulți biți;
- să faciliteze integrarea într-o rețea bidimensională, optimizată pentru propagarea eficientă a operanzilor și a rezultatelor intermediare, facilitând astfel realizarea combinațională a operației de înmulțire.

Fiecare celulă C_{ij} a circuitului iterativ din Figura 6.1, are schema logică bloc prezentată în Figura 6.2, cu ieşirile şi intrările definite în expresiile 17, respectiv 18 ale funcțiilor de ieşire.

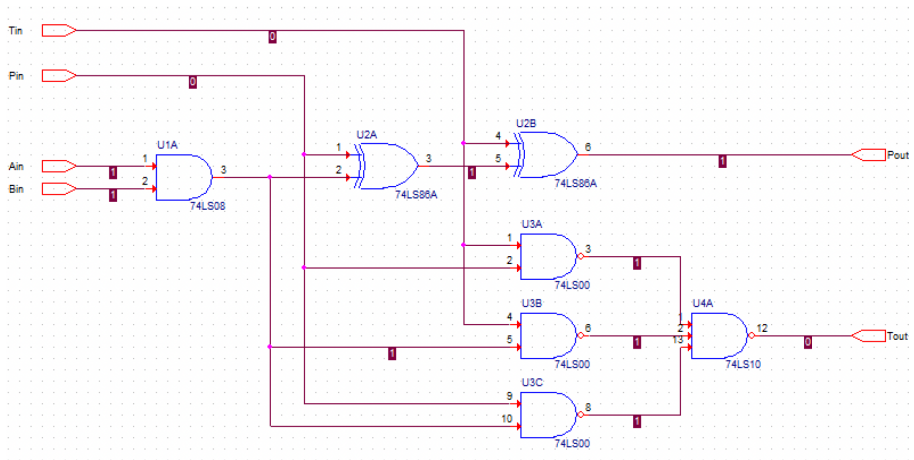


Fig. 6.2: Schema logică a unei celule C_{ij} din multiplicatorul iterativ.

Din Figura 6.1 se deduce că:

- $P_{0,i} = 0$, unde $i = 0 \div 3$ (produsul parțial inițial este nul);
- $T_{j,0} = 0$, unde $j = 0 \div 3$ (transportul parțial inițial este nul);
- $T_{j,i_{max}+1} = P_{j+1,j_{max}}$ (produsul parțial transmis către celula cu cea mai mare pondere a unui rând este determinat de transportul generat de celula cu cea mai mare pondere a rândului anterior).

6.2.2. Ecuațiile de ieşire ale celulelor multiplicatorului binar

În acestă secțiune se prezintă ecuațiile de ieşire ale multiplicatorului binar, ecuații reprezentative pentru produsul parțial P şi pentru bitul de transport T.

Ecuațiile produsului parțial P bitului de transport T sunt definte ca:

$$P_{j+1,i-1} = A_i \cdot B_j \oplus P_{ij} \oplus T_{ij}. \tag{17}$$

, respectiv

$$T_{j,i+1} = \overline{\overline{A_i \cdot B_j \cdot P_{ij}} \cdot \overline{A_i \cdot B_j \cdot T_{ij}} \cdot \overline{P_j \cdot T_{ij}}}. \quad (18)$$

, unde:

- i reprezintă rangurile deînmulțitorului;
- j reprezintă rangurile înmulțitorului;
- $P_{j,i}$ reprezintă produsul parțial de intrare în celula C_{ij}, provenit de la celula $C_{j-1,i+1}$;
- $P_{j+1,i-1}$ reprezintă produsul parțial provenit de la celula C_{ij}, şi aplicat celulei $C_{j+1,i-1}$;
- $T_{j,i}$ reprezintă transportul de intrare în celula C_{ij}, provenit de la celula precedentă $C_{j,i-1}$;
- $T_{j,i+1}$ reprezintă transportul provenit de la celula C_{ij}, şi aplicat celulei următoare $C_{j,i+1}$.

6.3. Desfășurarea lucrării

În cadrul lucrării de laborator se va proiecta şi simula un multiplicator binar pe doi biți. Pentru proiectarea circuitul se va utiliza programul *OrCAD Capture*, iar pentru simulare *OrCAD PSpice*. Pe baza relațiilor 17, respectiv 18 se va realiza schema unui multiplicator binar, capabil să înmulțească două numere, reprezentate pe doi biți.

Aşadar, se vor înmulți numerele $A = 11$ (3 în zecimal) cu $B = 11$ (3 în zecimal). Rezultatul acestui produs va fi un număr pe 4 biți.

Pentru implementarea în *Orcad Capture* a multiplicatorului binar, se vor utiliza blocurile ierarhice. Figura 6.2 este considerată ca fiind nivelul ierarhic cel mai de jos, reprezentând implementarea unei celule C_{ij}. Multiplicatorul de 2 biți va utiliza două celule pe fiecare nivel. Aceste celule vor fi conectate conform schemei din Figura 6.3. Se vor utiliza semnale de intrare continue cu valoare 0 sau 1, în funcție de necesitate.

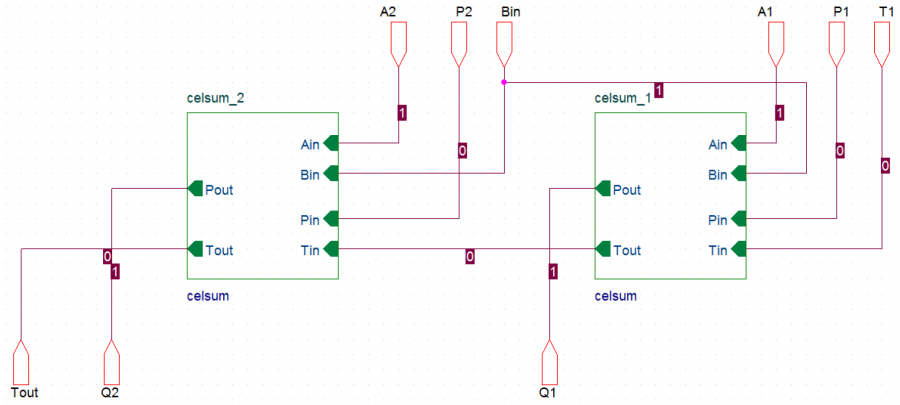


Fig. 6.3: Multiplicarea cu bitul de rang j al înmulțitorului.

În Figura 6.4, este prezentată schema completă a multiplicatorului, necesară pentru a înmulți două numere, pe 2 biți. Fiecare dintre blocurile din acestă schemă reprezintă un rând/nivel din Tabelul 6. Aşadar, fiecare bloc însumează, pe fiecare rang i al deînmulțitului, produsul parțial P_{ij} cu transportul inițial T_{ij} şi cu produsul dintre bitul A_i al deînmulțitului cu valoarea bitului B_j al înmulțitorului.

Produsul parțial P_{ij} aplicat celulei C_{ij}, provine de la celula $C_{j-1,i+1}$, situată pe rândul precedent pe aceeaşi coloană, dar de un rang mai mare cu o unitate. Transportul T_{ij} aplicat celulei C_{ij} provine de la celula precedentă $C_{j,i-1}$ ca rezultat al însumărilor de pe rangul precedent $(i-1)$ al aceluiaşi rând. Drept rezultat, celula generează $P_{j+1,i-1}$ (produsul parțial ce se va aduna pe aceeaşi coloană dar pe rândul j următor) şi $T_{j,i+1}$ (transportul ce se va aduna la celula de pe acelaşi rând j dar pe rangul următor $i+1$).

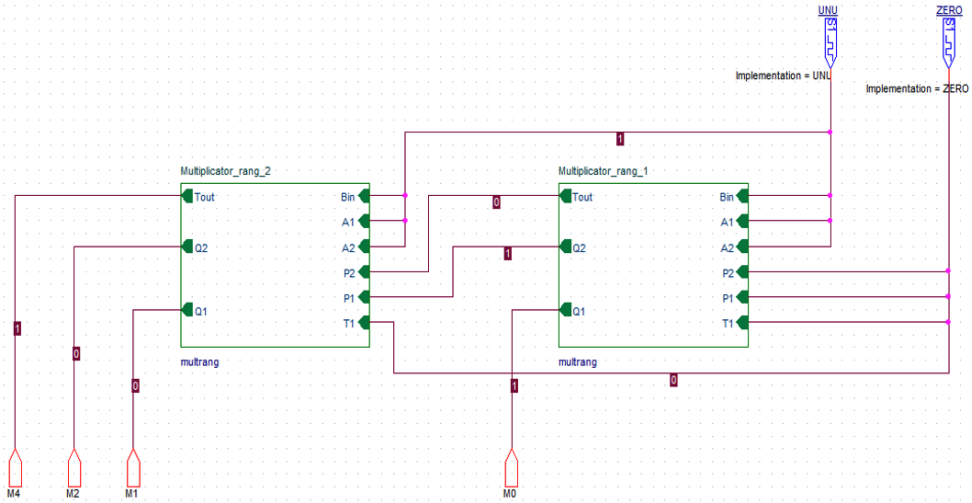


Fig. 6.4: Schema ierarhizată a multiplicatorului pe 2 biți.

6.4. Conținutul referatului

1. Însuşirea conceptelor teoretice cu privire la funcționarea multiplicatoarelor binare;
2. Proiectarea şi simularea schemei logice reprezentând un multiplicator binar pe 2 biți;
3. Verificarea corectitudinii schemei prin validarea simulării.

6.5. Verificarea cunoştințelor

1. Cum se poate defini un multiplicator binar?
2. Care sunt metodele de realizare a operației de înmulțire?
3. Care sunt cerințele pe care trebuie să le satisfacă un circuit de multiplicare?
4. Descrieți operația de înmulțire.
5. Reprezentați grafic operația de înmulțire.

Laborator 7
Circuite basculante bistabile

7.1. Scopul lucrării

În această lucrare de laborator se propune proiectarea și simularea circuitelor basculante bistabile.

7.2. Prezentare teoretică

Circuitele basculante bistabile (CBB) reprezintă o categorie de circuite logice secvențiale caracterizate prin două stări distincte, între care tranzițiile sunt realizate prin aplicarea unor semnale externe de comandă. Aceste circuite dispun de memorie, ceea ce permite determinarea stării anterioare a circuitului prin analiza ieșirilor curente, oferind astfel informații despre ultima comandă primită la intrare.

Datorită acestei proprietăți, circuitele basculante bistabile sunt fundamentale pentru proiectarea circuitelor logice secvențiale, fiind utilizate într-o gamă largă de aplicații, precum numărătoare, registre, memorii RAM și altele. În funcție de caracteristicile și modul de operare, există patru tipuri principale de circuite basculante bistabile: RS, JK, D și T. Aceste tipuri diferite permit adaptarea lor în funcție de cerințele specifice ale aplicațiilor electronice [2, 3].

7.2.1. Circuitele basculante bistabile de tip RS

Circuitele basculante bistabile de tip RS (Reset-Set) dispun de două intrări de comandă, specificate convențional cu *S (Set)* și *R (Reset)*, precum și de două ieșiri complementare Q și $\overline{Q}$. Funcția intrării S este de a înregistra informația în circuit (unde, prin convenție, informația reprezintă starea logică 1), iar intrarea R are rolul de a șterge informația (resetare).

Prin aplicarea unui semnal logic 1 la una dintre cele două intrări (S sau R, circuitul reacționează conform funcționalității sale: $S = 1$ determină

setarea stării circuitului ($Q = 1, \overline{Q} = 0$), iar $R = 1$ resetează circuitul ($Q = 0, \overline{Q} = 1$). Acest comportament oferă posibilitatea de a controla și memora stările circuitului în mod precis.

Modul de funcționare al unui circuit basculat RS poate fi interpretat urmărind tabelul de adevăr 7. Putem deduce următoarele aspecte din tabelul 7:

t_n	t_{n+1}			
R_n	S_n	Q_n	Q_{n+1}	Q_{n+1}
0	0	0	0	$Q_{n+1} = Q_n$
0	0	1	1	$Q_{n+1} = Q_n$
0	1	0	1	$Q_{n+1} = 1$
0	1	1	1	$Q_{n+1} = 1$
1	0	0	0	$Q_{n+1} = 0$
1	0	1	0	$Q_{n+1} = 0$
1	1	0	?	Nedefinit
1	1	1	?	Nedefinit

Tabel 7: Tabel de adevăr circuitele basculante bistabile de tip RS

- Intrări de comandă inactive: Când ambele intrări de comandă sunt inactive ($R_n = S_n = 0$), circuitul își menține starea curentă ($Q_{n+1} = Q_n$), ceea ce înseamnă că starea rămâne neschimbată;
- Operația de setare (*Set*): Când doar intrarea de setare este activă ($S_n = 1$, $R_n = 0$), circuitul înscrie informația ($Q_{n+1} = 1$), indiferent de starea sa anterioară;
- Operația de resetare (*Reset*): Când doar intrarea de resetare este activă ($S_n = 0$, $R_n = 1$), circuitul șterge informația ($Q_{n+1} = 0$), indiferent de starea sa anterioară;
- Condiție invalidă: Cazul în care ambele intrări sunt active simultan ($R_n = S_n = 1$) este definit ca invalid, deoarece nu este logic să înscrii și să ștergi informația în același timp. Pentru funcționarea corectă a circuitului, trebuie respectată condiția $R_n \cdot S_n = 0$.

Se menționează faptul că t_n reprezintă momentul curent, cu intrările și ieșirie R_n, S_n, Q_n. De asemenea, t_{n+1} reprezintă momentul de timp ce urmează cu ieșirea specifică Q_{n+1}.

În continuare se prezintă diagramele Karnaugh 8, 9 pentru ieșirile Q_{n+1} și $\overline{Q_{n+1}}$, cu formele minime disjunctive aferente 19, 20.

Tabel 8: Diagramele Karnaugh pentru ieșirea Q_{n+1}

Q_n/S_nR_n	00	01	11	10
0	0	0	X	1
1	1	0	X	1

$$Q_{n+1} = S_n + Q_n \cdot \overline{R_n}. \tag{19}$$

Tabel 9: Diagramele Karnaugh pentru ieșirea $\overline{Q_{n+1}}$

Q_n/S_nR_n	00	01	11	10
0	1	1	X	0
1	0	1	X	0

$$\overline{Q_{n+1}} = R_n + \overline{Q_n} \cdot \overline{S_n}. \tag{20}$$

În Figura 7.1 se regăseşte schema circuitului basculat de tip RS, cu porți *ȘI-NU*, după aplicarea legilor DeMorgan.

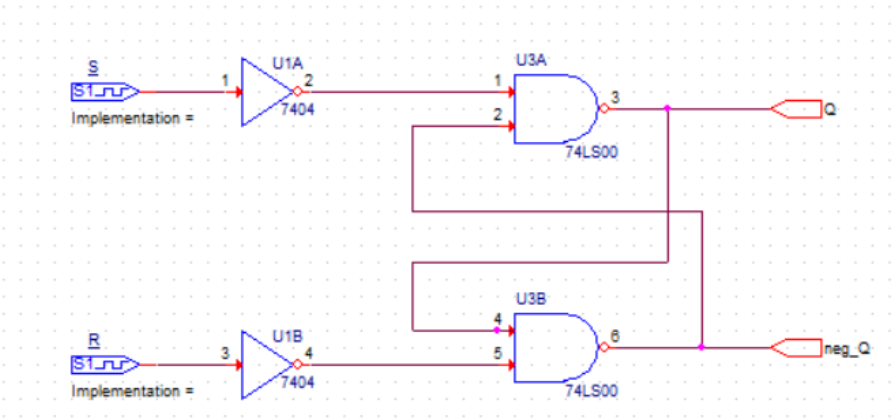


Fig. 7.1: Circuit basculat bistabil de tip RS asincron.

Circuitul prezentat anterior este asincron. O variantă foarte utilizată este cea sincronă, unde comenzile sunt activate de un semnal de tact (CLK) (vezi Figura 7.2). Atâta timp cât semnalul de tact este în 0 logic, ieșirile cele două porți *ȘI* au valoarea 0 logic, fără a fi influențate de valorile intrărilor R și S. Când semnalul de tact este 1, atunci rezultă o funcționare asincronă, deoarece $\overline{R} = R$ și $\overline{S} = S$.

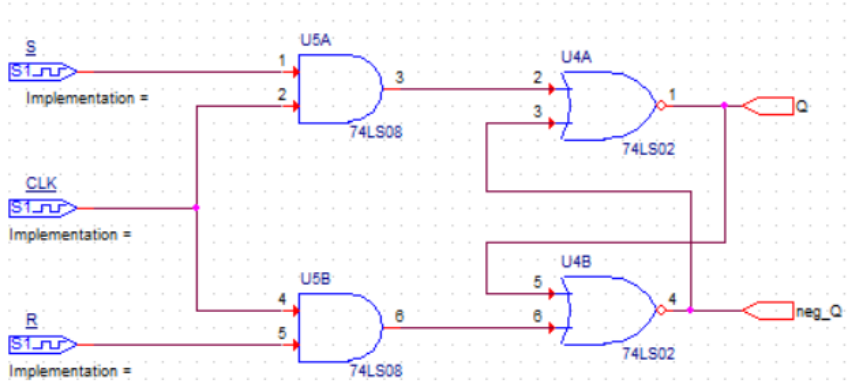


Fig. 7.2: Circuit basculat bistabil de tip RS sincron.

7.2.2. Circuitele basculante bistabile de tip JK

Problema circuitelor basculante RS legată de starea nedeterminată a ieșirilor atunci când ambele intrări sunt aduse simultan la nivel logic 1, este rezolvată prin utilizarea a două porți logice *ȘI* cu trei intrări și a circuitelor de reacție [4], conform schemei din Figura 7.3.

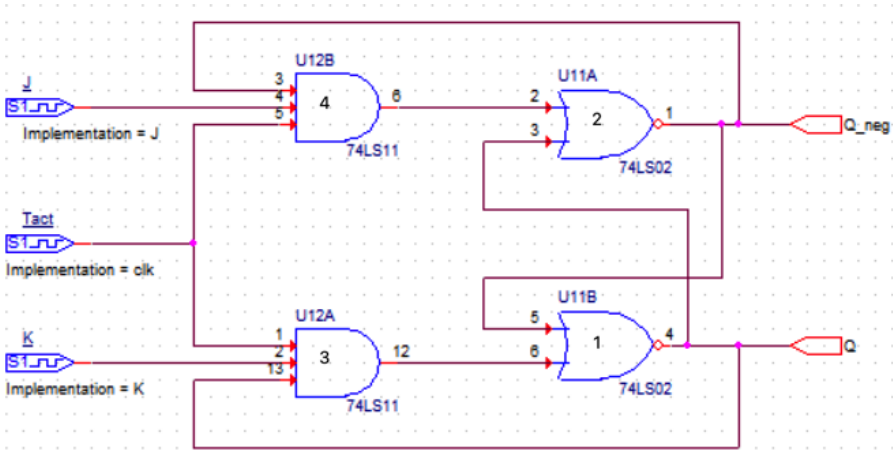


Fig. 7.3: Circuit basculat bistabil de tip JK sincron.

Din conexiunile interne rezultă că ieșirile porților *SAU-NU* influențează direct intrările porților *ȘI*, creând o buclă de reacție. Pentru a evita instabilitatea (auto-oscilarea), durata semnalului de ceas trebuie să fie foarte scurtă, astfel încât să revină la nivel logic 0 înainte ca modificările la nivelul ieșirilor să fie complet propagate.

Un comportament distinct al acestui circuit este observat atunci când intrările sunt simultan trecute în starea logică 1. În acest caz:

- Circuitul inversează starea ieșirilor: dacă starea inițială este $Q = 0$ și $\overline{Q} = 1$ traseul intern (poarta 4 | poarta 2), schimbă starea în $Q = 1$ și $\overline{Q} = 0$;
- Similar, dacă starea inițială este $Q = 1$ și $\overline{Q} = 0$ semnalul urmează traseul (poarta 3 | poarta 1), determinând o comutare în $Q = 0$ și $\overline{Q} = 1$.

Această funcționalitate evidențiază importanța temporizării corecte și rolul semnalului de ceas în determinarea comportamentului CBB de tip JK, aspect sintetizat vizual în Tabelul 10.

Tabel 10: Tabel de adevăr CBB de tip JK

CLK	J	K	Q_{n+1}
1	0	0	Q_n
1	1	0	1
1	0	1	0
1	1	1	$\overline{Q}_n$

7.2.3. Circuitele basculante bistabile de tip D

Un circuit basculant bistabil de tip D, se poate obţine plecând de la un circut basculant bistabil de tip JK și adaugând un inversor la intrarea K şi conectarea ieșirii lui la intrarea J. Figura 7.4 prezintă un CBB de tip D, iar tabelul de adevăr este prezentat prin 11.

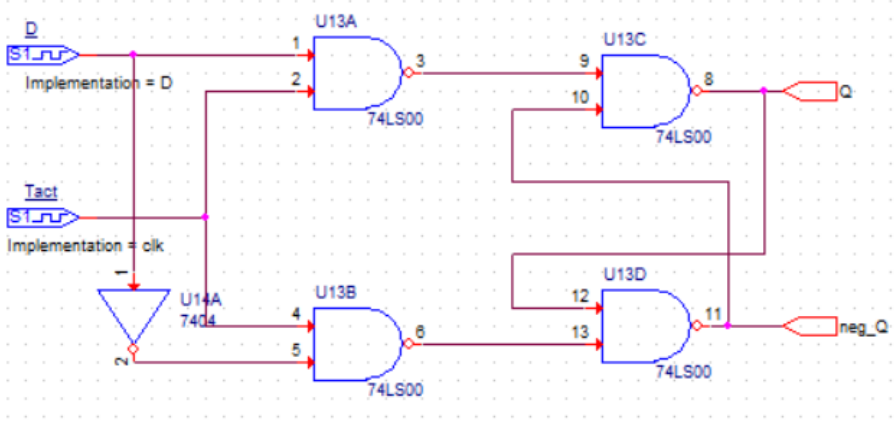


Fig. 7.4: Circuit basculat bistabil de tip D sincron.

O reprezentare grafică a obținerii unui bistabil de tip D pornind de unul de tip JK se poate observa și în Figura 7.5, unde se utilizează reprezentările

Tabel 11: Tabel de adevăr
CBB de tip D

CLK	D	Q_{n+1}
1	1	1
1	0	0

grafice ale circuitelor integrate pentru circuitele basculante bistabile de tip JK, respectiv D.

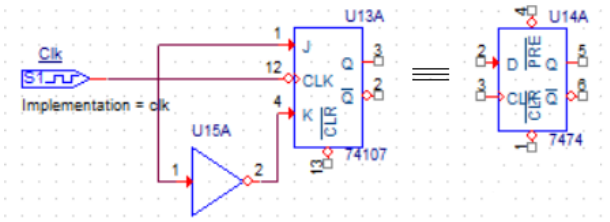


Fig. 7.5: Conversie CBB de tip JK în CBB de tip D.

Prin introducerea inversorului în configurația circuitului, comportamentul bistabilului JK se restrânge la cazurile în care intrările sunt complementare. În esență, acest lucru face ca bistabilul D să funcționeze ca un element sincron cu semnalul de tact. La fiecare front activ al semnalului de ceas, nivelul logic aplicat pe intrarea D este copiat pe ieșirea Q, iar circuitul își menține această stare până la apariția următorului front activ.

Totuși, o analiză mai detaliată (precum cea din Figura 7.6) demonstrează că ieșirea Q nu este identică în mod continuu cu intrarea D. Copierea valorii logice are loc doar în momentele sincronizate cu frontul activ al semnalului de tact. Astfel, semnalul de la ieșire prezintă o formă de undă distinctă, ce constă în nivelurile logice discrete preluate la fiecare impuls de tact, și nu o replică exactă a formei semnalului de la intrare.

Această observație subliniază natura sincronă a bistabilului D și rolul semnalului de tact în determinarea comportamentului său secvențial.

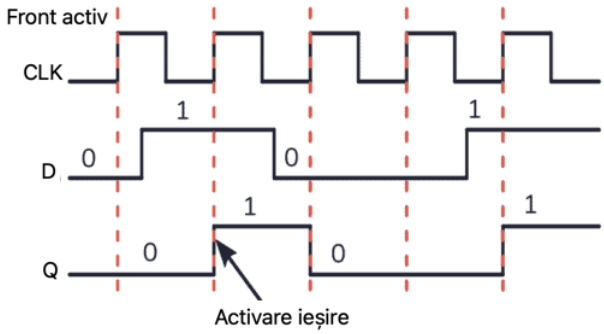


Fig. 7.6: Analiză circuit CBB de tip D [4].

subsubsection7.2.3.4. Circuitele basculante bistabile de tip T

Există o mulțime de aplicații în care circuitul basculant bistabil tip JK este utilizat cu intrările $J = K = 1$. Rezultă astfel că, din tabelul de adevăr 10 al circuitului JK, se păstrează doar intrările ale căror valoare au acelaşi nivel logic. Rezultă astfel tabelul de adevăr 12 al circuitului bistabil de tip T, CBB denumit și celulă de numărare.

Tabel 12: Tabel de adevăr CBB de tip T

CLK	$J = K$	Q_{n+1}
1	0	Q_n
1	1	$\overline{Q_n}$

Conform observațiilor din Figura 7.7, dacă ambele intrări sincronizate ale bistabilului sunt activate (nivel logic 1) pe frontul activ al semnalului de tact, circuitul își schimbă starea curentă, trecând în cea complementară. Această abilitate de a alterna între stări poate fi utilizată eficient în dezvoltarea numărătoarelor, unde schimbarea succesivă a stărilor este esențială pentru procesul de numărare.

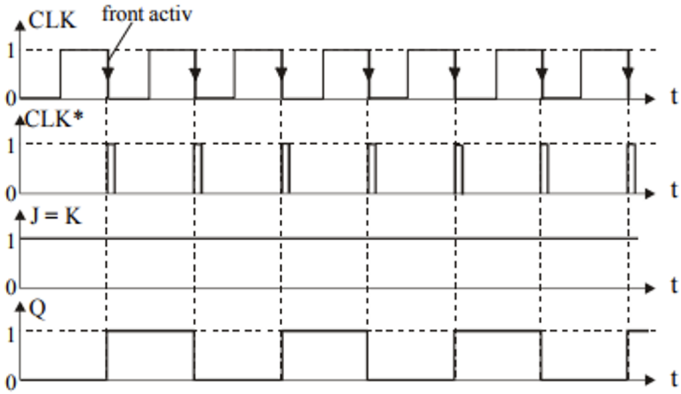


Fig. 7.7: Analiză circuit CBB de tip T [4].

7.3. Desfășurarea lucrării

În cadrul lucrării de laborator se vor simula în *OrCAD Capture* circuitele basculante bistabile de tip JK şi D.

Pentru CBB de tip JK se va alege componenta 74107 şi se vor utiliza semnalele din diagrama prezentată în Figura 7.8. Pentru semnalele de intrare CLK, J și K se va utiliza componenta *DigStim1*. Componenta 74107 prezintă, în afară de intrare CLK, J și K, și o intrare $\overline{CLR}$, ce are rolul de a reseta CBB-ul de tip JK. Deoarece această intrare este negată, este necesar să se conecteze un semnal continuu de valoare 1 logic, pentru a permite funcționarea circuitului.

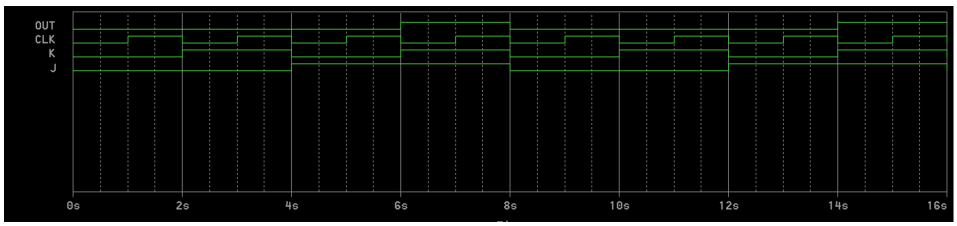


Fig. 7.8: Diagramă semnale CBB de tip JK.

Pentru circuitul basculant bistabil de tip D se va alege componenta 7474 şi se vor utiliza semnalele din diagrama expusă prin Figura 7.9. La fel ca în cazul circuitului basculant bistabil de tip JK, se va utiliza componenta

DigStim1 pentru semnalele de intrare. Componenta 7474 are, de asemenea o intrare pentru resetare, $\overline{CLR}$, și, în plus, oferă și posibilitatea de a preseta circutul cu o anumită valoare, prin intrarea $\overline{PRE}$. În cadrul acestei lucrări de laborator intrarea $\overline{PRE}$ se va conecta la un semnal continuu de valoare 1 logic. De asemenea, pentru CBB de tip D se cere și implementarea utilizându-se **porţi logice**, conform figurii 7.4, pentru a compara semnalele de ieşire obţinute.

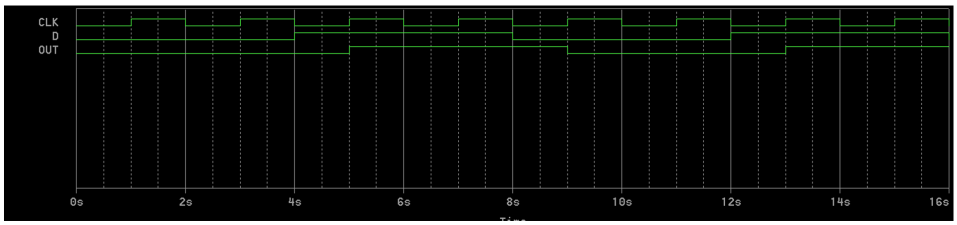


Fig. 7.9: Diagramă semnale CBB de tip D.

Pentru a evita starea de incertitudine de la începutul simulării, circuitele basculante bistabile vor fi inițializate cu valoarea 0 logic, prin accesarea ferestrei de dialog *Gate Level Simulation* (*Edit Simulation Profile - Options - Gate Level Simulation*) și urmând pașii din Figura 7.10.

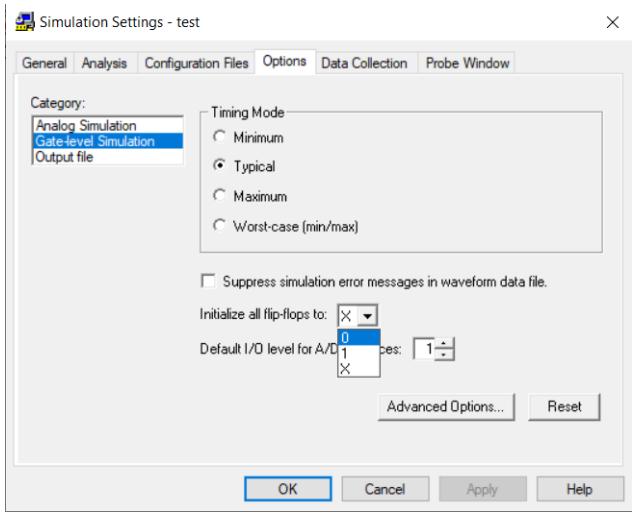


Fig. 7.10: Initializare circuite basculante bistabile.

Componente necesare proiectării și simulării schemelor sunt următoarele:

- 1 CBB de tip JK: 74107;
- 1 CBB de tip D: 7474;
- 4 porţi *NAND*: 74*LS*00;
- 1 poartă inversor *INV*: 7404.

7.4. Conținutul referatului

1. Însuşirea cunoștințelor despre funcționare circuitelor basculante bistabile;
2. Proiectarea și simularea unor exemple de CBB.

7.5. Verificarea cunoştințelor

1. Enumerați tipurile de bistabile descrise în acest laborator.
2. Care sunt condiţiile de basculare ale unui circuit basculant de tip D?
3. Care este dezavantajul circuitelor de tip RS?
4. Ce inconvenient al circuitelor RS rezolvă circuitul basculat JK?
5. Cum se obţine un bistabil de tip D, pornind de la un bistabil de tip JK?

Laborator 8
Numărătoare sincrone

8.1. Scopul lucrării

În această lucrare de laborator se introduc numărătoarele binare și se propune implementarea și simularea unui numărător binar sincron tip serie, utilizându-se pachetul de programe *OrCAD*.

8.2. Prezentare teoretică

Numărătoarele sunt esențiale în circuitele secvențiale pentru contorizarea impulsurilor, operând fără a necesita intrări de date. Schimbările de stare ale acestora sunt guvernate de starea curentă și de reguli prestabilite, fiecare stare fiind asociată unui număr specific din intervalul de contorizare. Astfel, ele constituie un mecanism ordonat de tranziție între stări distincte.

Structura unui numărător constă dintr-o combinație de bistabile și porți logice, care dirijează evoluția stărilor. Numărul maxim de stări posibile este determinat de numărul de bistabile utilizate, 2^n, ceea ce definește numărătorul ca fiind modulo 2^n. Fiecare stare este reprezentată printr-un cuvânt de cod binar format din n biți, corespunzător ieșirilor bistabilelor. Codul de numărare, descris prin succesiunea acestor cuvinte de cod, ilustrează modul în care circuitul avansează în procesul de contorizare.

Numărătoarele pot fi clasificate în funcție de modul în care impulsurile de tact sunt distribuite în circuit:

- **Numărătoarele asincrone** utilizează o arhitectură în care doar bistabilul cel mai puțin semnificativ primește direct semnalul de tact. Celelalte bistabile își preiau semnalul de tact din ieșirea bistabilului precedent (Q sau $\overline{Q}$). Această configurație determină comutări succesive între bistabile, ducând la întârzieri cumulative;

- **Numărătoarele sincrone** în schimb, aplică semnalul de tact simultan tuturor bistabilelor. Această simultaneitate elimină întârzierile propagate între bistabile, rezultând o comutare sincronizată a tuturor celulelor binare.

Frecvența de operare a numărătoarelor sincrone este semnificativ îmbunătățită, fiind limitată doar de întârzierea unui singur bistabil și de porțile logice suplimentare. Această caracteristică face ca numărătoarele sincrone să fie preferate în aplicațiile care necesită viteze mari de procesare și o sincronizare precisă a tranzițiilor.

8.2.1. Numărător binar sincron tip serie

Pentru a se deduce modul de funcționare al unui numărător binar sincron tip serie, se poate analiza tabelul de adevăr 13 furnizat.

Tabel 13: Tabel de adevăr numărător sincron de tip serie

Stare	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
0	0	0	0	0

De asemenea, schema logică a unui numărător binar sincron tip serie, compus din 4 celule JK, este afişată în Figura 8.1.

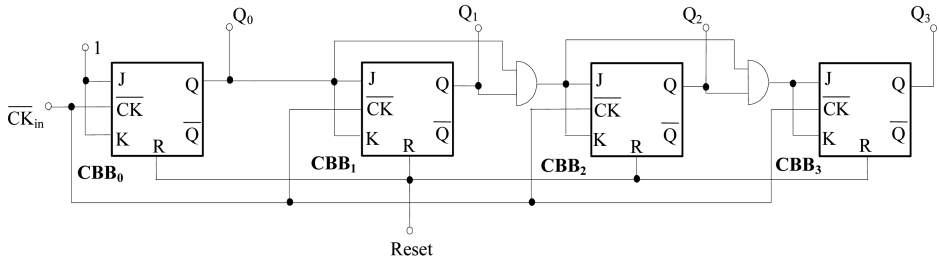


Fig. 8.1: Schema logică a unui numărător binar sincron tip serie [5].

Celulele de tip JK sunt configurate să comute între stări în funcție de condițiile definite de intrările J și K. Atunci când $J = K = 1$ celula comută în starea complementară, iar porțile logice *ȘI* din circuit determină momentele exacte ale tranziției. Într-un numărător binar de tip serie, fiecare celulă are un comportament specific:

- CBB_0: Primește impulsuri de tact direct și comută la fiecare ciclu. Intrările J_0 și K_0 sunt setate la 1 pentru a permite comutarea la fiecare impuls;
- CBB_1: Comută la fiecare al doilea impuls de tact, când Q_0 este 1. Astfel, intrările J_1 și K_1 sunt conectate la Q_0;
- CBB_2: Comută la fiecare al patrulea impuls, condiționat de stările Q_0 și Q_1 ambele fiind 1. Intrările sunt configurate drept $J_2 = K_2 = Q_0 \cdot Q_1$;
- CBB_3: Comută la fiecare al optulea impuls, atunci când Q_0, Q_1 și Q_2 sunt simultan în starea logică 1. Intrările devin $J_3 = K_3 = Q_0 \cdot Q_1 \cdot Q_2$.

Această logică permite realizarea unui numărător binar eficient, a cărui frecvență maximă de lucru este determinată de timpul necesar pentru comutarea celulelor și de întârzierea cumulată în porțile logice. Diagrama de comutare detaliată este prezentată în Figura 8.2.

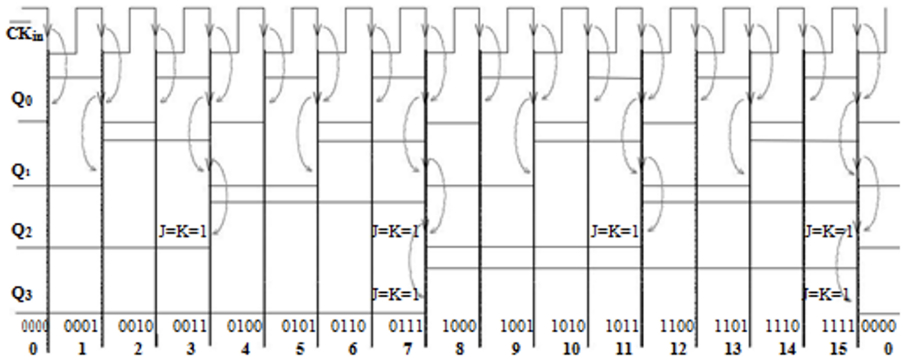


Fig. 8.2: Diagrama de comutare a unui numărător binar de tip serie [5].

8.2.2. Numărător binar sincron tip paralel

Pentru a crește suplimentar viteza de operare a numărătorului binar de tip serie, o soluție constă în evitarea legării în cascadă a porților logice *ȘI*. În schimb, fiecare poartă poate fi conectată direct la ieșirile celulelor bistabile (CBB), conform diagramei din Figura 8.3.

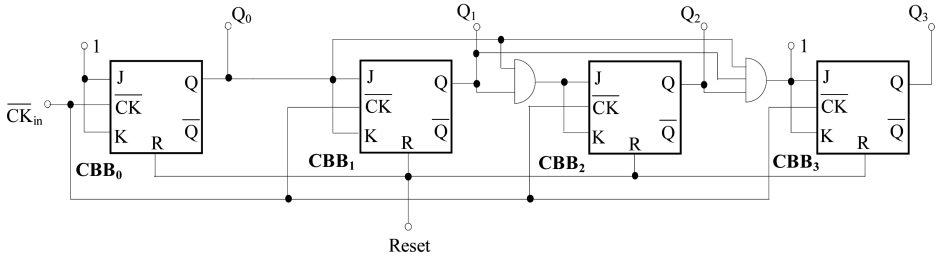


Fig. 8.3: Schema logică a unui numărător binar sincron tip paralel [5].

Prin această abordare, frecvența maximă de lucru este limitată exclusiv de timpul de comutare al fiecărei celule bistabile și de timpul de propagare asociat unei singure porți logice. Această metodă generează cel mai rapid tip de numărător binar, fiind ideală pentru aplicații care necesită timpi de răspuns foarte mici.

Însă, această configurație prezintă un dezavantaj semnificativ: fiecare intrare suplimentară conectată la o ieșire a celulelor bistabile crește sarcina electrică a acestora. Creșterea sarcinii electrice, la rândul său, mărește

timpul de comutare al celulelor și reduce frecvența maximă de operare a numărătorului. Astfel, compromisul între viteză și încărcare devine un factor critic în proiectarea acestui tip de circuit.

8.2.3. Numărător binar sincron reversibil

Cele mai întâlnite numărătoare în industrie sunt cele de tip sincron reversibil, a căror schemă logică se poate observa în Figura 8.4.

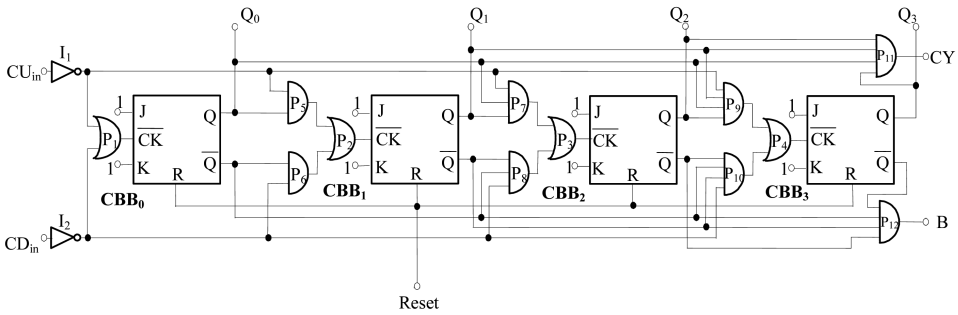


Fig. 8.4: Schema logică a unui numărător binar sincron tip reversibil [5].

Față de numărătoarele în serie și în paralel, noua schemă introduce o modificare importantă: intrările J și K ale celulelor bistabile sunt permanent conectate la nivel logic 1, iar impulsurile de tact sunt direcționate către intrările de tact ale CBB prin intermediul unor porți logice adiționale.

Comutarea circuitelor bistabile se produce pe frontul descendent al unuia dintre semnalele de comandă: CU (COUNT UP) pentru numărare crescătoare sau CD (COUNT DOWN) pentru numărare descrescătoare. Sensul de numărare este determinat de activarea unuia dintre aceste semnale, în timp ce celălalt este menținut la nivel logic 1. Dacă ambele semnale CU și CD se află la nivel logic 1, ieșirile porților logice I_1 și I_2 devin 0 logic, ceea ce blochează funcționarea porților $P_5, P_6, P_7, P_8, P_9, P_{10}, , P_{11}, P_{12}$. Când apare un front descendent pe unul dintre semnalele CU sau CD, porțile aferente vor deveni active: P_5, P_7, P_9, P_{11} pentru numărarea crescătoare sau P_6, P_7, P_{10}, P_{12} pentru numărarea descrescătoare. Tranzițiile sunt apoi transmise către circuitele basculante bistabile, iar condițiile de basculare

depind de starea logică a ieșirilor conectate la respectivele porți: 1 logic pentru numărarea crescătoare și 0 logic pentru numărarea descrescătoare.

Acest tip de numărător este dotat cu ieșiri suplimentare pentru extinderea funcționalităților:

- **CY** (carry sau transport): Această ieșire generează un impuls pozitiv (tranziție 0 – 1 – 0) atunci când numărătorul atinge starea maximă (toate ieșirile Q_0, Q_1, Q_2, Q_3 sunt 1). Ieșirea **CY** este utilizată pentru a extinde numărătoarele în funcție de numărarea crescătoare, conectându-se la intrarea **CU** a numărătorului următor.
- **BW** (borrow sau împrumut): Similar, această ieșire generează un impuls pozitiv (tranziție 0 – 1 – 0) atunci când numărătorul ajunge la starea minimă (toate ieșirile Q_0, Q_1, Q_2, Q_3 sunt 0). Aceasta este folosită pentru extinderea numărătorului în sens descrescător, conectându-se la intrarea **CD** a numărătorului următor.

8.3. Desfășurarea lucrării

Se cere implementarea şi simularea unui numărător binar sincron tip serie, utilizându-se pachetul de programe *OrCAD*. Pentru îndeplinirea acestei sarcini, se va folosi schema bloc prezentată în Figura 8.1.

Pentru a simula numărătorul se va utiliza diagrama de semnale prezentată în Figura 8.5, unde primul semnal este cel de Clear, iar al doilea este impulsul de tact (CLK). Pentru intrările *CLR* si *CLK* se vor utiliza intrări de tip *DigClock*, ce pot fi configurate conform cu Figura 8.6.

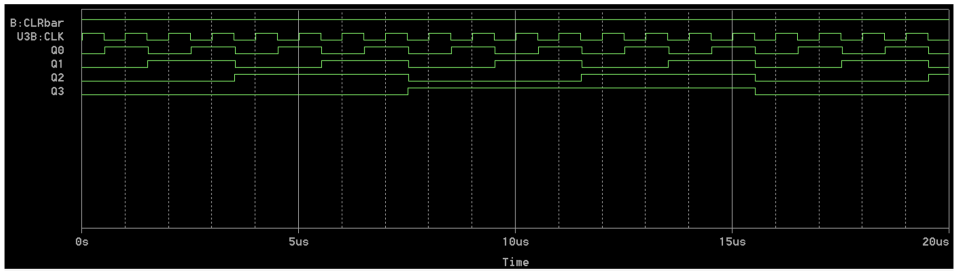


Fig. 8.5: Diagrama de semnal pentru simularea numărătorului.

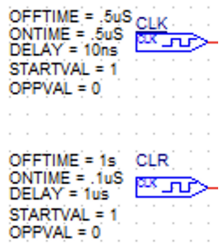


Fig. 8.6: Configurarea intrărilor CLR şi CLK.

Intrările J si K ale primului bistabil vor fi setate cu 1 logic. Pentru acest lucru se poate utiliza o sursă ce va furniza un semnal permanent. Această sursă poartă se găseşte sub denumirea de $D - HI$.

În cazul în care sursa continuă de semnal nu se regăseşte în meniul *Place - Power*, se va adăga biblioteca corespunzătoare urmând paşii din Figura 8.7.

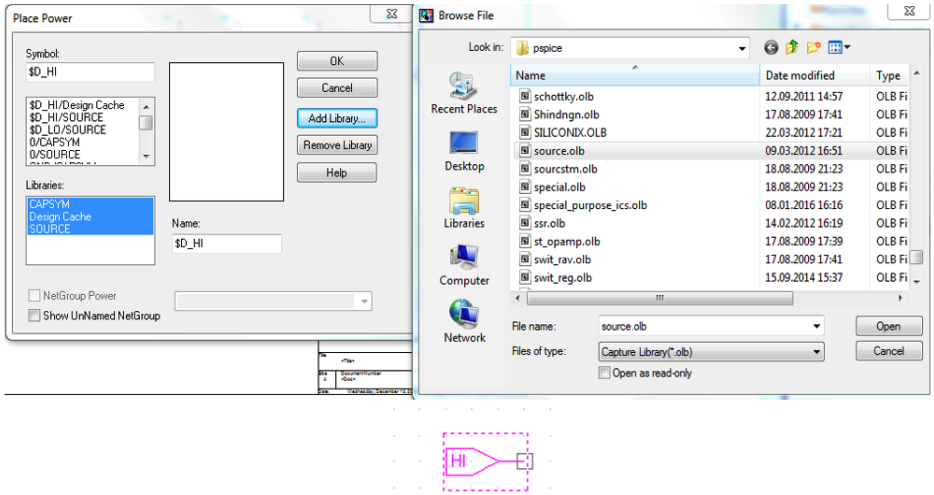


Fig. 8.7: Simbol sursă de semnal şi paşii necesari adaugării bibliotecii sursei.

Componente necesare proiectării și simulării schemei numărătorului binar sincron tip serie sunt următoarele:

- 4 CBB de tip JK: 74107;
- 2 porţi *AND*: 74*LS*08.

În urma rulării simulării, se vor afişa toate cele 4 semnale de ieşire ale CBB-urilor, precum şi semnalele de intrare. De asemenea, pentru a evita starea de incertitudine de la începutul simulării, circuitele basculante bistabile vor fi inițializate cu valoarea 0 logic, prin accesarea ferestrei de dialog *Gate Level Simulation* (*Edit Simulation Profile - Options - Gate Level Simulation*) și urmând pașii din Figura 7.10.

8.4. Conținutul referatului

1. Însuşirea cunoștințelor despre funcționarea numărătoarelor binare;
2. Proiectarea și simularea unui numărător binar sincron tip serie.

8.5. Verificarea cunoştinţelor

1. Care dintre numărătoarele prezentate în laborator este mai rapid, şi de ce?
2. Care sunt particularităţile unui numărător reversibil?
3. Ce determină capacitatea de numărare a unui numărător?

Laborator 9
Registre

9.1. Scopul lucrării

În această lucrare de laborator se vor prezenta noțiuni introductive legate de funcționarea registrelor și se propune proiectarea și simularea registrelor de memorie și deplasare utilizând pachetul de programe *OrCAD*.

9.2. Prezentare teoretică

Un bistabil extins pentru funcționarea în paralel este denumit registru. Acesta reprezintă un circuit logic secvențial conceput pentru a stoca sau transfera secvențe (numere) binare [3]. În funcție de scopul lor specific, registrele pot fi împărțite în următoarele categorii:

- Registre de memorare (cu încărcare paralelă) - utilizate pentru păstrarea datelor, având o structură de tip latch.
- Registre de deplasare (cu încărcare serială) - folosite pentru mutarea datelor binare în mod secvențial.
- Registre mixte (cu încărcare paralelă și serială) - capabile să combine atât memorarea, cât și deplasarea datelor.
- Registre universale - dispozitive versatile care pot îndeplini multiple funcții, inclusiv stocare și deplasare.

9.2.1. Registre de memorare (RM)

Registrele de memorare sunt utilizate pentru stocarea temporară a valorilor binare în cadrul sistemelor numerice. Acestea sunt construite cu circuite basculante bistabile de tip D, care funcționează sincronizat sub controlul unui semnal comun de tact. Procesul de memorare are loc simultan în toate celulele bistabile, fie pe frontul activ, fie pe nivelul activ al semnalului de tact.

Un aspect distinct al acestui tip de registru este lipsa conexiunilor între bistabilele individuale, deoarece nu este necesar un transfer serial al datelor. Schema logică a unui astfel de registru, configurat pentru 4 biți, este ilustrată în Figura 9.1.

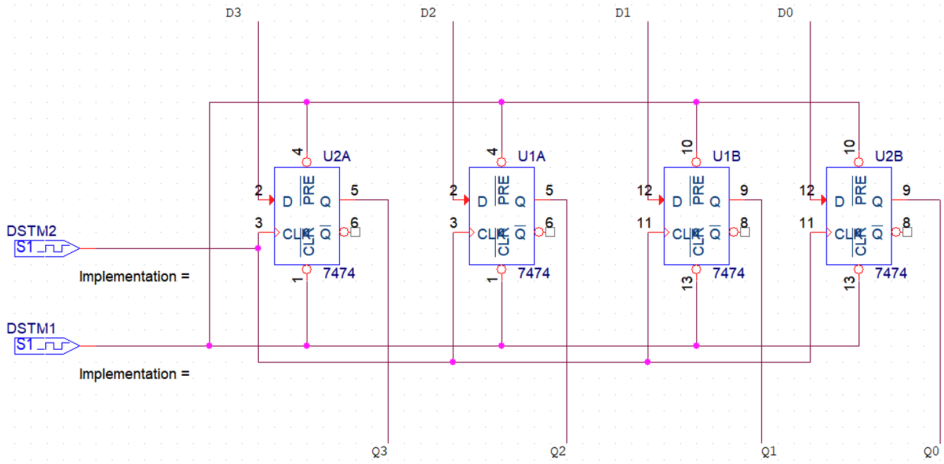


Fig. 9.1: Schemă bloc registru memorare.

Procesul de memorare al unui număr binar $A_b = a_{N-1}a_{N-2}\ldots a_0a_1$, prezent la momentul t_n la intrările D_k ale registrului, are loc pe frontul descendent al semnalului de tact. Astfel, la momentul t_{n+1}, același număr binar va apărea și la ieșirile registrului.

Acest proces poate fi sintetizat astfel:

$$t_n : D_k = a_k \implies t_{n+1} : Q_k = D_k = a_k \tag{21}$$

, unde x_k poate avea valoarea 0 sau 1, iar $k = 0, 1, \ldots, N-1$. În acest mod, cei N biți sunt încărcați simultan în registru, ceea ce reprezintă o încărcare paralelă.

Acest tip de registru este cunoscut și sub denumirea de *registru cu încărcare paralelă* sau *memorie tampon* (latch).

9.2.2. Registre de deplasare (RD)

Registrele de deplasare sunt circuite logice secvențiale care, cu fiecare impuls al semnalului de ceas, își deplasează informația către stânga sau dreapta cu o poziție. Practic, ele transferă datele dintr-o celulă în cea anterioară sau următoare. Prima celulă va prelua valoarea de la intrarea serială, iar ultima celulă își pierde conținutul. Aceste registre pot fi implementate cu bistabile de tip D, configurate în cascadă.

Schema logică a unui astfel de registru este ilustrată în Figura 9.2. Configurația, denumită Serial In Serial Out (SISO), este una dintre cele mai simple, având doar trei conexiuni: intrarea serială (SI), ieșirea serială (SO) și semnalul de tact (CLK). Ieșirea circuitului de rang k este conectată la intrarea celui de rang $k+1$ (ex. $Q_k = D_k$). Toate celulele folosesc același semnal de tact.

La momentul t_n, dacă ieșirile celulelor sunt:

$$Q_0(n) = a_0, Q_1(n) = a_1, \ldots, Q_{N-1}(n) = a_{N-1},$$

iar intrarea serială este $SI(n) = a_{in}$, după frontul negativ al semnalului de tact, ieșirile registrului devin:

$$Q_0(n+1) = SI(n), Q_1(n+1) = Q_0(n), \ldots, Q_{N-1}(n+1) = Q_{N-2}(n).$$

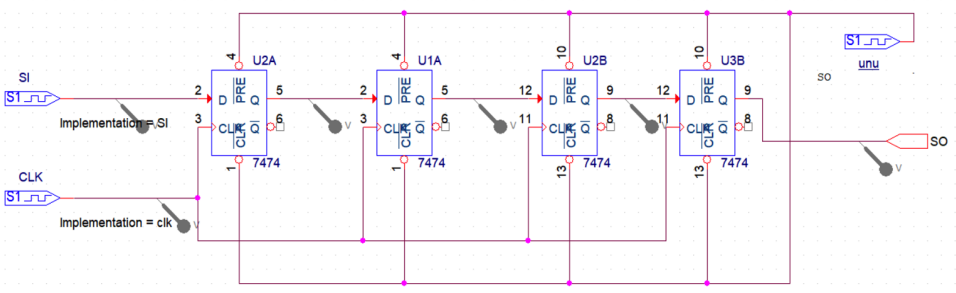


Fig. 9.2: Schemă bloc registru de deplasare.

Cu fiecare semnal de tact, registrul de deplasare își transferă informația către dreapta, mutând-o cu o poziție (de la cel mai puțin semnificativ bit la cel mai semnificativ bit). În mod similar, se poate crea un registru pentru deplasare la stânga, conectând intrarea CBB-ului k la ieșirea CBB-ului $k+1$. Direcția de deplasare devine relevantă în cazul utilizării bidirecționale, deoarece un registru pentru deplasare stânga-dreapta poate funcționa ca un registru dreapta-stânga, prin inversarea indicilor ieșirilor Q_k. În practică, se construiesc registre integrate care pot realiza ambele direcții de deplasare: acestea sunt numite registre bidirecționale sau reversibile. Direcția de deplasare este determinată de semnalul de control ($\overline{\text{Sens}}$). Configurația unui astfel de registru este prezentată în Figura 9.3.

După cum se poate observa, între celulele registrului au fost adăugate multiplexoare 2:1. Acestea stabilesc conexiunile $Q_k = D_{k+1}$ pentru deplasarea stânga-dreapta sau $D_k = Q_{k+1}$ pentru deplasarea dreapta-stânga. Dacă semnalul de control este $\overline{\text{Sens}} = 0$, multiplexorul MUX_k va selecta $Y = I_0$, ceea ce corespunde conexiunii $D_{k+1} = Q_k$, iar registrul va deplasa datele de la stânga la dreapta. În schimb, pentru $\overline{\text{Sens}} = 1$, multiplexorul MUX_i va selecta $Y = I_1$, realizând conexiunea $D_k = Q_{k+1}$, ceea ce determină deplasarea datelor de la dreapta la stânga.

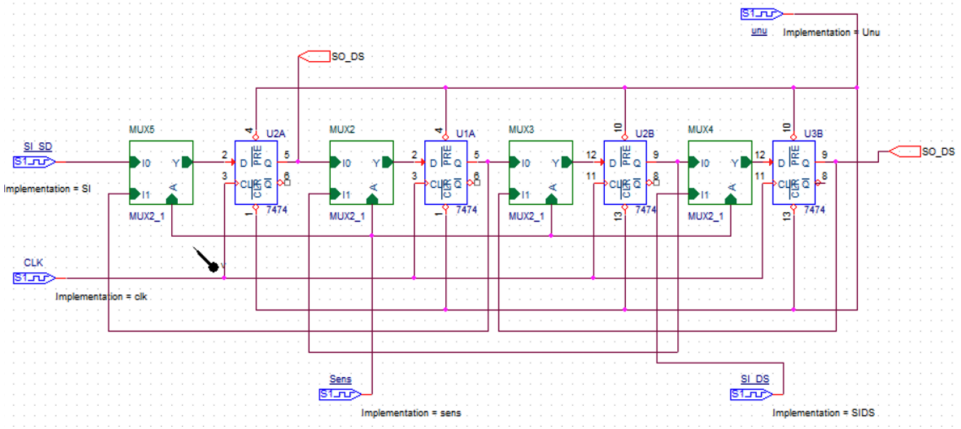


Fig. 9.3: Schemă bloc registru de deplasare bidirecțional.

9.2.3. Registre de mixte sau combinate (RC)

În numeroase aplicații, este avantajoasă utilizarea unor registre capabile să includă atât intrări și ieșiri paralele, cât și seriale, astfel încât să permită conversia între serie-paralel și paralel-serie. Un exemplu de astfel de registru, care dispune de intrări seriale și paralele, precum și ieșiri seriale și paralele, este ilustrat în Figura 9.4.

Funcționarea acestui registru este determinată de starea semnalului de control $\overline{CM}$:

- Dacă $\overline{CM} = 0$, registrul acționează ca un registru de deplasare stânga-dreapta.
- Dacă $\overline{CM} = 1$, registrul funcționează ca un registru de memorare.

Primul mod de funcționare utilizează tactul $\overline{CK_S}$, în timp ce al doilea folosește tactul $\overline{CK_P}$. Există două semnale de tact separate, deoarece, în unele aplicații, fiecare mod de operare necesită un semnal de tact distinct.

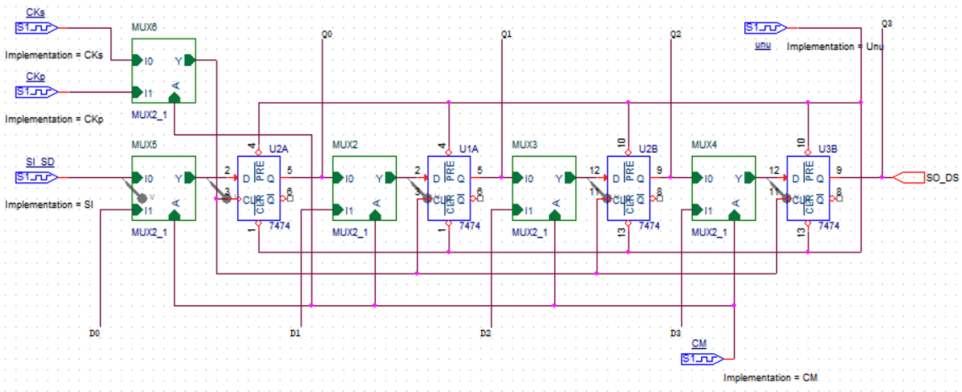


Fig. 9.4: Schemă bloc registru de combinat.

9.2.4. Registre de universal (RU)

Similar cu registrul mixt, se poate construi și un registru universal, care combină toate funcțiile menționate anterior: deplasarea stânga-dreapta și dreapta-stânga, memorarea, precum și intrările și ieșirile seriale și paralele. Pentru a realiza acest tip de registru sunt necesare multiplexoare 4:1, iar structura sa este asemănătoare celei ilustrate în Figura 9.4.

Funcționarea unui registru universal de acest tip este prezentată în Tabelul 14.

Tabel 14: Funcționarea registrului universal

$\overline{CM_0}$	$\overline{CM_1}$	Funcție
0	0	*Registru de deplasare stânga-dreapta*
1	0	*Registru de deplasare dreapta-stânga*
2	0	*Registru de memorare*
3	0	*Funcție neutilizată*

9.3. Desfășurarea lucrării

În cadrul lucrării de laborator se vor proiecta şi simula un registru de memorare și un registru de deplasare conform schemelor figurilor 9.1 și 9.2. Pentru proiectarea circuitelor se va utiliza programul *OrCAD Capture*, iar pentru simulare *OrCAD PSpice*. Intrările circutelor se vor seta conform figurilor 9.5, respectiv 9.6.

Pentru a se evita starea de incertitudine de la începutul simulării, circuitele basculante bistabile vor fi inițializate cu valoarea 0 logic, prin accesareaferestrei de dialog *Gate Level Simulation* (*Edit Simulation Profile - Options - Gate Level Simulation*) și urmând pașii din Figura 7.10.

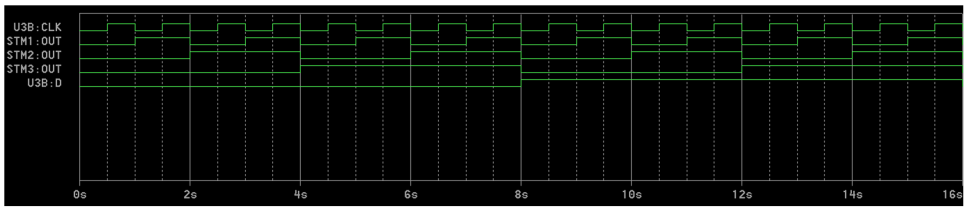


Fig. 9.5: Intrări circuit registru memorare.

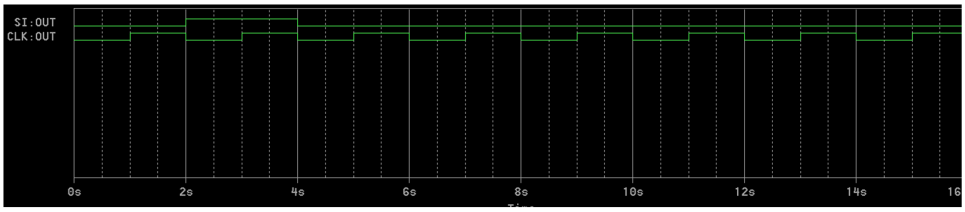


Fig. 9.6: Intrări circuit registru deplasare.

9.4. Conținutul referatului

1. Însuşirea conceptelor teoretice cu privire la funcționarea registrelor;
2. Proiectarea şi simularea schemei logice a unui registru;
3. Verificarea corectitudinii schemei prin validarea simulării.

9.5. Verificarea cunoştințelor

1. Pentru ce se poate utiliza un registru de memorare?
2. Ce este un registru de deplasare?
3. Ce este un registru universal?
4. Este posibil ca un registru să aibă intrări serie și ieșiri în paralel?

Laborator 10
Introducere în limbajul Verilog

10.1. Scopul lucrării

În această lucrare de laborator se va prezenta Verilog, un limbaj de descriere hardware (HDL - Hardware Description Language) ce poate fi utilizat pentru pentru proiectarea și implementarea circuitelor digitale [6].

10.2. Prezentare teoretică

Limbajele de descriere hardware constituie un element esențial în industria electronică, facilitând procesul de proiectare și implementare a circuitelor digitale. Printre cele mai cunoscute astfel de limbaje se numără *Verilog* și *VHDL*. Deși sintaxa lor are asemănări cu C, C++ sau Java, este esențial să subliniem că instrucțiunile în aceste limbaje nu sunt executate secvențial, ca în cazul unui procesor. Codul realizat în Verilog are ca scop fie implementarea pe un *FPGA* (Field-Programmable Gate Array), fie conceperea unui *ASIC* (Application-Specific Integrated Circuit).

Verilog oferă o gamă largă de abstractizări puternice, facilitând generarea automată a porților logice direct din cod. Spre deosebire de metodele tradiționale, precum desenarea manuală a schemelor logice în programe precum *OrCAD*, aceste abstractizări permit o accelerare semnificativă a procesului de proiectare. Ele susțin avansul rapid al tehnologiilor de fabricație, oferind posibilitatea de a controla și modela structuri complexe în mod eficient, reducându-le la componente fundamentale esențiale [7].

Limbajul Verilog permite descrierea sistemelor numerice utilizând diferite niveluri de abstractizare. Circuitul poate fi descris prin următoarele metode:

- descriere structurală: Se bazează pe compunerea ierarhică, utilizând primitive precum porți logice (*ȘI*, *SAU* etc.) și module, pentru a implementa diverse funcționalități.
- descriere comportamentală: Specifică comportamentul unui sistem numeric. Aceasta se poate detalia la nivel de transfer între registre (*flux de date*) sau la nivel procedural.

Circuitele logice combinaționale sunt concepute pentru a aplica funcții logice asupra seturilor de intrări, generând ieșiri care depind exclusiv de valorile actuale ale acestor intrări. Ori de câte ori o valoare de intrare se modifică, ieșirea este actualizată instantaneu, fără a ține cont de stările anterioare. Logica combinațională poate fi analizată și descrisă folosind mai multe metode, inclusiv:

- diagrame structurale care ilustrează interconectarea porților logice;
- tabele de adevăr ce prezintă toate combinațiile posibile de intrări și rezultatele lor corespunzătoare;
- expresii booleene ce definesc funcțiile logice în mod matematic.

În contrast cu logica combinațională, circuitele secvențiale iau în considerare atât valorile curente ale intrărilor, cât și stările precedente ale circuitului. În funcție de abordare, sistemele secvențiale pot fi clasificate în două categorii: sincrone și asincrone.

În cadrul laboratorului actual, vom explora bazele acestui limbaj nou, focalizându-ne asupra descrierii structurale a unui circuit combinațional.

10.2.1. Organizarea structurală a limbajului Verilog

Realizarea unui circuit digital utilizând un limbaj de descriere hardware (HDL) implică redactarea unei secțiuni de cod care să definească funcționarea acestuia. Odată compilat, acest cod generează o reprezentare digitală a circuitului, ce fi simulată în cadrul unui mediu controlat pentru a-i verifica funcționalitatea. De asemenea, se pot folosi instrumente care permit sintetizarea codului și conversia acestuia în fișiere de configurare, utilizate ulterior pentru programarea unor medii precum FPGA.

Datorită complexității crescute a proiectării unor astfel de circuite, se preferă adesea o abordare de tip top-down. Această metodă implică împărțirea unui sistem complex în sub-blocuri funcționale mai mici, continuând acest proces până se ajunge la nivelul cel mai elementar (de exemplu, o poartă logică). Astfel, implementarea și testarea devin mai ușor de gestionat. În acest mod, procesul de proiectare stabilește, în același timp, arhitectura sistemului.

10.2.1.1. Module

Modulul este componenta fundamentală a limbajului Verilog, reprezentând o unitate atomică ce descrie atât comportamentul, cât și interfața unui circuit. Acesta oferă utilizatorului o interfață de intrare/ieșire, care encapsulează funcționalitatea și implementarea internă.

Pentru declararea unui modul, se utilizează perechea de cuvinte cheie `module` și `endmodule`. Fiecare modul conține următoarele elemente:

- un nume unic pentru identificare;
- o listă de porturi care definește interfața externă: porturi de intrare (`input`), porturi de ieșire (`output`) sau porturi bidirecționale (`inout`), fiecare putând avea unul sau mai mulți biți;
- implementarea logicii funcționale corespunzătoare.

Exemplu de structură a unui modul Verilog:

```
module example_module (
    output iesire,       // port de ieșire
    input [2:0] in1,    // port de intrare pe 3 biți
    input in2           // port de intrare pe 1 bit
);
/* implementarea funcționalității interne */
endmodule
```

10.2.1.2. Primitive

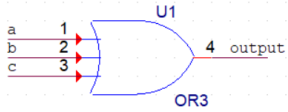
Porțile fundamentale reprezintă elementele de bază, denumite și *primitive*, în limbajul Verilog. Aceste primitive sunt predefinite și includ următoarele categorii:

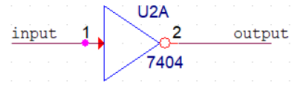
- porți logice: `and`, `or`, `nand`, `nor`, `xor`, `xnor`;
- porți pentru transmisie: `not`, `buf` și altele;
- tranzistori: `pmos`, `tranif` etc.

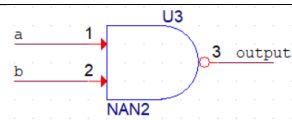
Fiecare primitivă dispune de porturi care permit conectarea acesteia la exterior. Primitivele predefinite permit conectarea unui număr variabil de intrări (precum `or`, `and`, `xor` etc.) sau ieșiri multiple (de exemplu, `buf`, `not`). Pentru a utiliza o primitivă, aceasta trebuie instanțiată împreună cu lista de semnale aferente, care vor fi asociate porturilor sale. Este important de subliniat că, în cazul acestor primitive, porturile de ieșire sunt specificate înaintea celor de intrare.

Tabelul 15 prezintă demonstrații practice de declarare a porților logice în Verilog. În plus, pentru primitivele predefinite, atribuirea unui nume de instanță este opțională.

Simbol	Implementare

	`or(output, a, b, c);`
	sau
	`or o1(output, a, b, c);`

	`not(output, input);`
	sau
	`not not_gate(output, input);`

	`nand(output, a, b);`
	sau
	`nand n(output, a, b);`

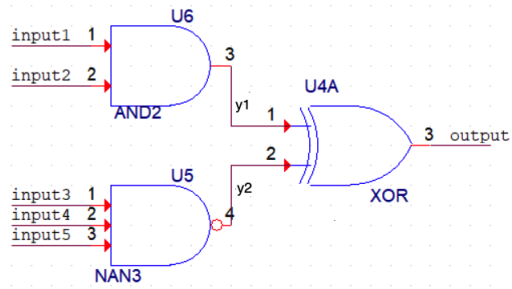
Tabel 15: Exemple de cod pentru utilizarea porților logice în Verilog.

10.2.1.3. Wires

O singură primitivă sau un modul individual nu poate realiza, de unul singur, funcționalitatea cerută. Din acest motiv, devine necesară interconectarea modulelor sau a primitivelor. Semnalele dintr-o diagramă sunt specificate folosind elemente de tip *wire*, declarate prin intermediul cuvântului cheie `wire`.

În Tabelul 16, semnalele `y1` și `y2`, fiecare având un singur bit, sunt utilizate pentru a interconecta ieșirile porților logice `and` (`y1`) și `nand` (`y2`) cu intrările porții `xor`.

Schemă	Secțiune Cod

	`wire y1, y2;`
	`and(y1, input1, input2);`
	`nand(y2, input3`
	`,input4, input5);`
	`xor(output, y1, y2);`

Tabel 16: Exemple de cod pentru utilizarea `wire` în Verilog.

Pentru semnalele ce constau din mai mulți biți, se utilizează vectori. În exemplul de mai jos semnalul `m` cuprinde 8 biți, iar `n` este reprezentat pe 5 biți. În acest context, bitul cel mai semnificativ (*most significant bit* - MSB) este localizat în partea stângă a vectorului, iar bitul cel mai puțin semnificativ (*least significant bit* - LSB) este situat în partea dreaptă.

```
wire [7:0] m; // semnal pe 8 biți
wire [4:0] n; // semnal pe 5 biți
```

10.3. Desfășurarea lucrării

În cadrul lucrării de laborator se cere implementarea și simularea:

- unei porți *XOR* folosind operații *AND*, *OR*, *INV*. Se va utiliza scheletul de cod furnizat în cadrul laboratorului, în secțiunea 10.3.1., unde se vor completa zonele marcate cu `TODO`;
- unui sumator elementar complet, utilizând sumatoare elementare parțiale. Se va utiliza, de asemenea, scheletul de cod furnizat în cadrul laboratorului, în secțiunea 10.3.2., unde se vor completa zonele marcate cu `TODO`.

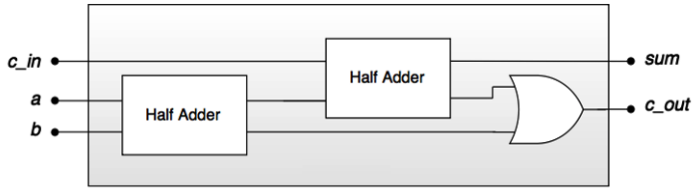


Fig. 10.1: Schemă logică sumator elementar complet.

Sumatorul binar complet prezentat în laboratorul 5, are schema prezentată în Figura 10.1 fiind compus din două sumatoare parțiale, fiecare dintre acestea având schema logică din Figura 10.2.

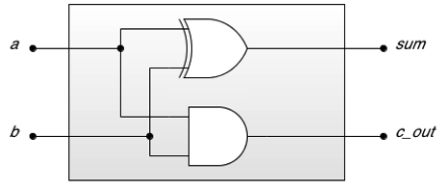


Fig. 10.2: Schemă logică sumator parțial.

Metodele necesare, din limbajul Verilog, pentru proiectarea circuitelor menționate sunt:

- `and(out,in1,in2)` pentru poarta logică *AND* cu două intrări;

- `or(out,in1,in2)` pentru poarta logică *OR* cu două intrări;
- `not(out,in1)` pentru poarta logică *INV* de inversare/negare.

Pentru implementarea codului se poate utiliza un mediu de dezvoltare online, urmând adresa:

`https://www.tutorialspoint.com/compilers/online-verilog-compiler.htm`.

10.3.1. Schelet cod de completat pentru implementare poartă *XOR*

```
    module xor_gate (
    output xor_out,
    input a,
    input b
);
    wire not_a, not_b;
    wire and1, and2;

    // TODO : se completează secvența de cod pentru poarta XOR
    // NOT gates
    //not (out, in);

    // AND gates
    //and (out, in1, in2);

    // OR gate
    // or ();

endmodule

// testare implementare
module xor_test;

        // intrări
        reg a;
        reg b;
        // ieșirea
        wire xor_out;
        reg [1:0] err_cnt = 0;

        // Instanțiere Unit Under Test (UUT)
        xor_gate uut (
                .xor_out(xor_out),
```

```
                .a(a),
                .b(b)
        );
    // In Verilog, blocul inițial block este utilizat pentru executarea
    // unui set de instrucțiuni o singură dată la începutul simularii.
    // Este folosit pentru condițiile inițiale, cum ar fi inițializarea
    // variabilelelor sau generarea stimulilor utilizați în test bench.
        initial begin
                // Initializare intrari
                a = 0;
                b = 0;

                // Așteaptă 100 ns
                #100;

                // Adaugare stimuli
               a = 1;
               b = 0;

                #5;
                $display("xor_out = %b", xor_out); // Afișează rezultat
                if(xor_out != 1) begin
                        $display ("Sum error for a = %d b = %d",a,b);
                        err_cnt = err_cnt + 1;
                end

                //TODO: se testează pentru celelalte 3 combinații posibile
        //ale intrărilor.

                #10;
                if (!err_cnt)
                        $display ("XOR test completed successfully");
        end
endmodule
```

10.3.2. Schelet cod de completat pentru implementarea unui sumator elementar complet

```
module xor_gate (
    output xor_out,
    input a,
    input b
```

```
);

// TODO: se copiază implementarea porții XOR din exercițiul anterior.
endmodule

module half_adder(
    output sum,
    output c_out,   // bit transport rezultat sau carry out
    input  a,
    input  b);

   // TODO: implementare sumator parțial conform figurii

endmodule

module full_adder(
    output sum,
    output c_out,   // bit transport rezultat sau carry out
    input  a,
    input  b,
    input  c_in);    // bit transport folosit ca intrare sau carry in

    // TODO: implementare sumator elementar complet conform figurii

endmodule

// testare implementare
module full_adder_test;

        // Intrări
        reg a;
        reg b;
        reg c_in;

        // Ieșiri
        wire sum;
        wire c_out;
        reg [1:0] err_cnt = 0;

        // Instantiere Unit Under Test (UUT)
        // TODO: instanțiere modul full_adder pentru testare
```

```
initial begin
        // Initialize Inputs
        a = 0;
        b = 0;
        c_in = 0;

        // Așteptare 100 ns
        #100;

        // Adaugare stimuli
a = 1;
b = 0;
c_in = 1;

        #5;
        $display("sum = %b, c_out = %b", sum, c_out);
        if(sum != 0 || c_out != 1) begin
                $display (
        "Sum error for a=%d b=%d c_in=%d",a,b,c_in
        );
                err_cnt = err_cnt + 1;
        end

        #10;
        a = 1;
        b = 1;
        c_in = 0;

        #5;
        $display("sum = %b, c_out = %b", sum, c_out);
        if(sum != 0 || c_out != 1) begin
                $display (
        "Sum error for a=%d b=%d c_in=%d",a,b,c_in
        );
                err_cnt = err_cnt + 1;
        end

        // TODO: testare altă combinație de intrări

        #10;
        if (!err_cnt)
                $display ("Adder test completed successfully");
```

```
        end
endmodule
```

10.4. Conținutul referatului

1. Însuşirea conceptelor de bază ale limbajelor de descriere hardware;
2. Implementarea și simularea unor exemple de circuite electronice.

10.5. Verificarea cunoştințelor

1. Ce este un limbaj de descriere hardware?
2. Ce limbaje de descriere hardware cunoaşteți?
3. Care este modul de execuție al codului într-un limbaj HDL?
4. La ce se referă cuvântul cheie *wire*?
5. Ce este o primitivă?

Laborator 11
Analiza tipurilor de descriere a modulelor în Verilog

11.1. Scopul lucrării

În cadrul acestei lucrări de laborator se vor prezenta diferitele tipuri de descriere a modulelor în limbajul Verilog, precum și caracteristicile fiecărei metode. Alături de această componentă teoretică, se vor implementa și exemple practice pentru fiecare tip de descriere, utilizându-se un mediu de dezvoltare online (https://www.tutorialspoint.com/compilers/online-verilog-compiler.htm). De asemenea, vor fi introduse și conceptele de atribuiri blocante și non-blocante, alături de elmente specifice limbajului Verilog, cum ar fi blocurile `initial` și `always`.

11.2. Prezentare teoretică

Descrierea modulelor în Verilog poate fi realizată prin mai multe metode, fiecare având propriile sale caracteristici și aplicații. Cele mai comune tipuri de descriere sunt:

- **Descriere structurală**
- **Descrierea la nivel de flux de date**
- **Descrierea comportamentală sau procedurală**

11.2.1. Descriere structurală

Această metodă este cea mai aproape de implementarea fizică și se concentrează pe conexiunile dintre componente. Un modul este descris ca o colecție de instanțe ale altor module sau a unor porți logice primitive (*AND*, *OR*, *NOT*, etc.). Este similară cu desenarea unui circuit pe o placă de circuit imprimat. Avantajul acestei metode este că oferă o reprezentare clară a structurii hardware, dar poate deveni complexă pentru circuite mari. Această metodă a fost utilizată în cadrul laboratorului 10, pentru implementarea porții *XOR* și a sumatorului elementar complet.

În cele ce urmează este prezentat un exemplu de implementare al unui multiplexor 4:1 folosindu-se descrierea structurală în Verilog, împreună cu un testbench pentru verificarea funcționalității acestuia. Multiplexorul 4:1 are patru intrări de date (*data0*, *data1*, *data2*, *data3*), două intrări de selecție (*select0*, *select1*) și o ieșire (*mux_out*). În funcție de valorile intrărilor de selecție, ieșirea va reflecta una dintre cele patru intrări de date. Circuitele digitale de tip multiplexor sunt utilizate în aplicații precum rutarea semnalelor, selectarea datelor din mai multe surse, sau implementarea funcțiilor logice complexe. Schema logică a multiplexorului 4:1 este prezentată în Figura 11.1.

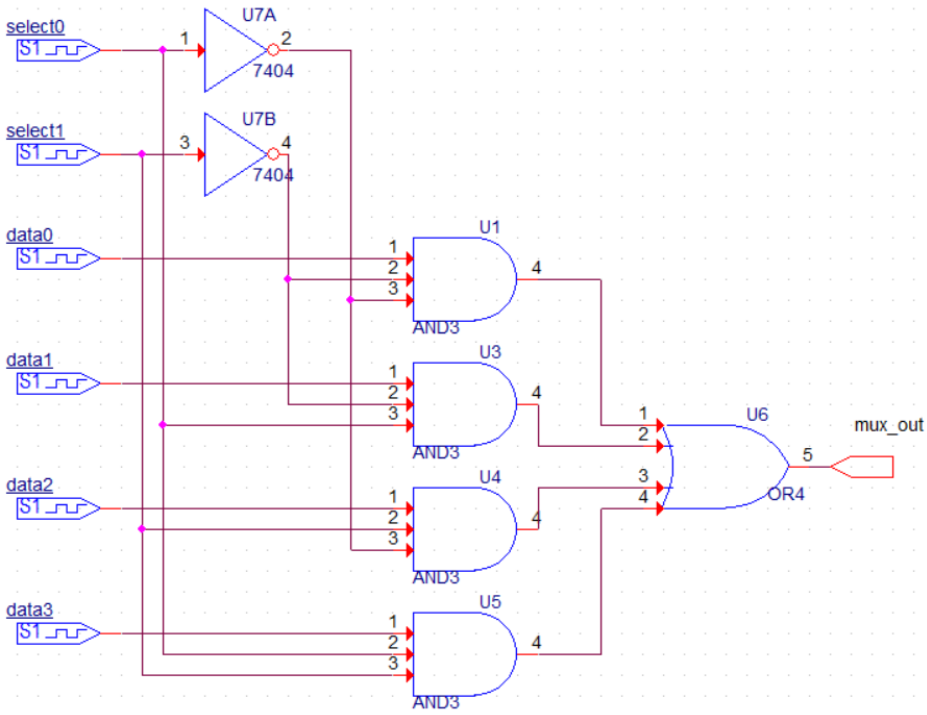


Fig. 11.1: Circuit multiplexor 4:1.

Codul Verilog pentru implementarea multiplexorului 4:1 și testbench-ul aferent este prezentat mai jos:

```
module mux4to1(
    output mux_out,
    input data0,
    input data1,
    input data2,
    input data3,
    input select0,
    input select1
);

    wire n_select0;
    wire n_select1;
    wire w0;
    wire w1;
    wire w2;
    wire w3;

    not(n_select0, select0);
    not(n_select1, select1);

    and(w0, data0, n_select1, n_select0);
    and(w1, data1, n_select1, select0);
    and(w2, data2, select1, n_select0);
    and(w3, data3, select1, select0);

    or(mux_out, w0, w1, w2, w3);

endmodule

// Testbench
module tb_mux4to1;
    reg data0, data1, data2, data3;
    reg select0, select1;
    wire mux_out;

    mux4to1 uut (
        .mux_out(mux_out),
        .data0(data0),
        .data1(data1),
        .data2(data2),
        .data3(data3),
        .select0(select0),
```

```
        .select1(select1)
    );

    initial begin
        // Test all combinations
        data0 = 0; data1 = 1; data2 = 0; data3 = 1;
        select0 = 0; select1 = 0; #10;
        $display("sel=00, mux_out=%b", mux_out);

        select0 = 1; select1 = 0; #10;
        $display("sel=01, mux_out=%b", mux_out);

        select0 = 0; select1 = 1; #10;
        $display("sel=10, mux_out=%b", mux_out);

        select0 = 1; select1 = 1; #10;
        $display("sel=11, mux_out=%b", mux_out);

        $finish;
    end
endmodule
```

11.2.2. Descrierea la nivel de flux de date

Descrierea la nivel de flux de date se concentrează pe transferul de date între registre și rețele. Aceasta este o abordare intermediară, mai aproape de hardware decât descrierea procedurală. Se bazează pe expresii logice și operatori de atribuire continuă.

Atribuirile continue sunt o caracteristică fundamentală a limbajului Verilog, utilizată pentru a descrie circuite logice combinaționale, adică circuite fără memorie, a căror ieșire depinde doar de valorile curente ale intrărilor.

Pentru realizarea unei atribuirii continue, se utilizează cuvântul cheie `assign`, urmat de o expresie logică care definește relația dintre intrări și ieșiri, precum în exemplul de mai jos:

```
assign <out> = <expression>;
```

Caracteristicile atribuirilor continue sunt următoarele:

- Valoarea atribuirii este actualizată instantaneu și continuu. De fiecare dată când oricare dintre semnalele din partea dreaptă a egalului își schimbă

valoarea, expresia este re-evaluată, iar rezultatul este atribuit imediat semnalului din partea stângă.

- Operanzii din stânga trebuie să fie de tip `wire`.
- Nu se poate utiliza în cadrul unui bloc `initial` sau `always`.
- Paralelism inerent. Atribuirile continue sunt concurente. Nu există o ordine de execuție, ele ”rulează” toate în paralel. Acest lucru este esențial, deoarece reflectă natura paralelă a hardware-ului real.

Mai jos sunt câteva exemple de utilizare a atribuirilor continue în Verilog:

- AND logic:
```
assign y = a & b;
```
- Multiplexor:
```
assign out = select ? in1 : in0;
```
- Circuit combinațional cu o complexitate mai mare:
```
assign a_or_b = a | b;
assign c_and_d = c & d;
assign final_out = a_or_b & c_and_d;
```

În ultimul exemplu, dacă valoarea lui `a` se schimbă, `a_or_b` se actualizează imediat, ceea ce declanșează actualizarea lui `final_out`. Aceasta se întâmplă fără a fi nevoie de un ceas sau de o procedură secvențială.

În Verilog se pot utiliza diferiți operatori pentru a construi expresii logice în cadrul atribuirilor continue. Acești operatori pot fi clasificați în mai multe categorii:

- Aritmetici - utilizați pentru calcule matematice:
  - Unari: - (schimbare semn)
  - Binari: +, -, *, /, %
- Logici - aplică o operație pe fiecare bit al unui număr:
  - Unari: ! (negație, exemplu: !1 e 0, iar !0 e 1)
  - Binari: &&, ||
- Relaționali - compară două valori: >, <, >=, <=, ==, ! =
- Pe biți - aplică o operație pe fiecare bit al unui număr:

  - Unari: ~ (inversarea fiecărui bit)
  - Binari: & (și pe biți), — (sau pe biți), ^ (xor pe biți), ~^ sau ^~ (xnor)
- Alți operatori speciali:
  - Concatenare: {var0, var1, ...} (unește mai multe variabile sau numere pentru a forma un șir de biți mai lung)
  - Deplasare: << (stânga), >> (dreapta) pentru a muta biții unui număr
  - Operator ternar: (condiție) ? valoare_adevarat : valoare_fals; (o scurtătură pentru if-else)

Mai jos este un exemplu complet de implementare al unui multiplexor 4:1 folosindu-se descrierea la nivel de flux de date în Verilog:

```
module mux4to1 (
    output mux_out,
    input data0,
    input data1,
    input data2,
    input data3,
    input select0,
    input select1
);

assign mux_out =
    (data0 & ~select1 & ~select0) |
    (data1 & ~select1 & select0)  |
    (data2 & select1 & ~select0)  |
    (data3 & select1 & select0);
endmodule
```

11.2.3. Descrierea comportamentală sau procedurală

Acest tip de descriere se referă la modul în care codul este executat în secvențe, adică pas cu pas, similar cu un limbaj de programare tradițional. În Verilog, această abordare este folosită în blocurile de tip `initial` și `always`, care descriu comportamentul circuitului în timp.

Acest stil de descriere se axează pe comportamentul sau funcționalitatea circuitului, fără a specifica direct cum este implementat hardware-ul.

Exemplu de implementare al unui multiplexor 4:1 folosindu-se descrierea comportamentală în Verilog:

```
module mux4to1(
    output reg mux_out,
    input data0,
    input data1,
    input data2,
    input data3,
    input select0,
    input select1
    );

    always @(*) begin
        case ({select1, select0})
            2'b00: mux_out = data0;
            2'b01: mux_out = data1;
            2'b10: mux_out = data2;
            2'b11: mux_out = data3;
            default: mux_out = 1'bx;
        endcase
    end

endmodule
```

11.2.3.1. Initial și Always

Blocurile `initial` și `always` delimitează secțiunile de cod procedural din modul, permițând utilizarea structurilor de control asemănătoare celor din limbajele de programare procedurală.

Blocul `initial` din Verilog este un bloc procedural care se execută o singură dată, la începutul simulării (la timpul t=0).

Are următoarele caracteristici principale:

- Este utilizat pentru a inițializa variabilele și semnalele la începutul simulării.
- Spre deosebire de blocul `always`, care rulează continuu sau de fiecare dată când se schimbă o intrare, un bloc `initial` se execută o singură dată.

- Dacă un modul conține mai multe blocuri `initial`, toate se vor executa în paralel la timpul t=0, ceea ce duce la o execuție concurentă.
- Se utilizează adesea în mediile de testare pentru a genera stimuli de testare sau pentru a seta condiții inițiale.

Exemplu cod Verilog cu bloc `initial`:

```
initial begin
    X = 16'b1010101010101010;        // Atribuim registrului X o valoare
                                     // binară pe 16 biți.
    Y = {X[8:15], 8'b11110000};      // Y primește concatenarea ultimilor 8
                                     // biți din X și 8 biți de 1 și 0.
    Z = 16'hBEEF;                    // Registrul Z primește o valoare
                                     // hexazecimală pe 16 biți.
end
```

Fiecare dintre aceste metode are avantajele și dezavantajele sale, iar alegerea uneia dintre ele depinde de cerințele specifice ale proiectului și de preferințele designerului.

Blocul `always` din Verilog este un bloc procedural care se execută în mod repetat, de fiecare dată când se schimbă o condiție specificată în lista de sensibilitate.

Sintaxa generală a unui bloc `always` este următoarea:

```
always @(<sensitivity list>)
begin
  // Instrucțiuni procedurale
end
```

Mai jos sunt câteva caracteristici și utilizări ale blocului *always*:

- *Sensitivity list* sau Lista de sensibilitate: Specifică semnalele sau evenimentele care declanșează execuția blocului. Poate conține o listă de semnale (ex: @(a,b,c)) sau evenimente la front (ex: @(posedge clk) pentru frontul crescător al ceasului sau @(negedge reset) pentru frontul descrescător al semnalului de reset).

  - Declarațiile `always @(a,b,c)`: Descriu circuite combinaționale. Ori de câte ori una dintre intrări (a, b sau c) își schimbă valoarea, blocul se re-evaluează.
  - Declarațiile `always @(posedge clk)`: Descriu circuite secvențiale, adică cele sincronizate cu un ceas. De exemplu, un flip-flop sau un registru.

- Blocurile `always` pot fi utilizate pentru a modela atât circuite secvențiale (sincronizate cu un ceas), cât și circuite combinaționale (fără memorie).
- Poate conține instrucțiuni procedurale, cum ar fi atribuiri, condiții (if-else), bucle (for, while) și altele.

Pentru circuite secvențiale (cu memorie) sintaxa utilizată este `always @(posedge clk or negedge reset_n)`, blocul fiind folosit pentru a modela registre, flip-flop-uri, mașini de stări finite, și alte elemente sincronizate. Se execută doar la frontul de ceas sau al semnalului de reset.

Pentru circuite combinaționale (fără memorie) sintaxa utilizată este `always @(*)`. `@(*)` este un scurtătură care include automat toate intrările din expresia blocului în lista de sensibilitate. Este modul preferat de a descrie circuite combinaționale (ex: multiplexoare, decodoare) pentru a evita erorile. Blocurile `always` pot fi sintetizate în hardware și sunt esențiale pentru a descrie un circuit real.

Mai jos este un exemplu de bloc `always` care descrie un registru sincronizat cu ceasul:

```
reg clk;
reg [7:0] D;
reg [7:0] Q;

always @(posedge clk) begin
    Q <= D; // La fiecare front pozitiv de clk,
            // Q primește valoarea lui D
end
```

11.2.3.2. Construcții de control

Construcțiile de control în Verilog sunt instrucțiuni procedurale care asigură fluxul de execuție al unui bloc de cod. Ele sunt similare cu cele dintr-un limbaj de programare tradițional, permițând modelarea de logici complexe, cum ar fi condiții și repetiții.

1. Instrucțiuni condiționale - Acestea permit selectarea logicii pe baza unei condiții.

   - **if-else** - Modelează o decizie binară. E util pentru a implementa un multiplexor sau un circuit care alege între două opțiuni.

```
// Exemplu: Multiplexor 2-la-1
module Mux2_to_1 (
    input a, b,
    input sel,
    output out
);
    reg out;

    always @(a, b, sel) begin
        if (sel == 1'b1) begin
            out = a;
        end
        else begin
            out = b;
        end
    end
endmodule
```

   - **case** - Oferă o structură mai clară pentru multiple decizii bazate pe o singură variabilă. E ideal pentru a modela un decodor sau o mașină de stări.

```
// Exemplu: Decodor 2-la-4
module Decoder2_to_4 (
    input [1:0] in,
    output reg [3:0] out
);
    always @(in) begin
        case (in)
            2'b00: out = 4'b0001;
```

```
            2'b01: out = 4'b0010;
            2'b10: out = 4'b0100;
            2'b11: out = 4'b1000;
            default: out = 4'b0000; // Poate fi ignorat
        endcase
    end
endmodule
```

2. Instrucțiuni de repetare - Folosite în principal în medii de testare pentru a genera stimuli sau a inițializa variabile.

   - **for** - Execută o buclă de un număr fix de ori.

```
// Exemplu: Inițializarea unui registru mare
initial begin
    reg [7:0] data_bus;
    integer i;
    for (i = 0; i < 8; i = i + 1) begin
        data_bus[i] = 1'b0; // Setează toți biții la 0
    end
    $display("Registrul a fost initializat la 0.");
end
```

   - **forever** - Rulează la nesfârșit, adesea pentru a genera un ceas.

```
// Exemplu: Generarea unui semnal de ceas
reg clk;

initial begin
    clk = 1'b0;
    forever #10 clk = ~clk; // Comută ceasul la fiecare 10 unități de timp
end
```

   - **while** - Execută o buclă atâta timp cât o condiție rămâne adevărată.

```
// Exemplu: Așteptarea unei condiții
initial begin
    reg [3:0] counter = 0;
    while (counter < 5) begin
        #5; // Așteaptă 5 unități de timp
        counter = counter + 1;
        $display("Counter este acum: %0d", counter);
    end
    $display("Bucla while s-a terminat.");
end
```

3. Instrucțiuni de control al timpului - Acestea modelează întârzieri și evenimente în simulare.

   - **întârziere (#)** - Introduce o pauză în execuția codului procedural.

```
// Exemplu: Stimuli temporizati
initial begin
    reg a, b;
    a = 1'b0;
    b = 1'b0;

    #15; // Așteaptă 15 unități de timp
    a = 1'b1;

    #5;  // Așteaptă încă 5 unități de timp
    b = 1'b1;
end
```

   - **așteptare (@)** - Așteaptă un eveniment specificat în lista de sensibilitate.

```
// Exemplu: așteptarea unui front crescător de ceas
always @(posedge clk) begin
    // Acest cod se execută doar pe frontul pozitiv al lui 'clk'
    data_out <= data_in; // Atribuire non-blocantă
end
```

11.2.3.3. Atribuiri blocante și non-blocante

Atribuirile în Verilog sunt de două tipuri: blocante (=) și non-blocante (<=). Acestea controlează modul în care instrucțiunile sunt executate în timp, fiind esențiale pentru a modela corect circuitele.

Atribuirile blocante (=) se execută imediat și blochează execuția oricărei alte instrucțiuni din același bloc procedural până când se finalizează. Ele sunt folosite în principal pentru a modela circuite combinaționale, unde rezultatul depinde doar de intrările curente. Se execută secvențial, în ordinea în care apar în cod, având următoarea sintaxă:

```
variabilă = expresie; // variabilă ia valoarea expresiei imediat
```

Un exemplu de utilizare a atribuirilor blocante în cadrul unui bloc `always` este prezentat mai jos:

```
always @(*) begin
    a = b + 1;
    c = a * 2; // Noua valoare a lui ‘a‘ este folosită
               // imediat pentru ‘c‘.
end
```

Atribuirile blocante sunt utilizate în cadrul blocurilor `always` care modelează circuite combinaționale.

Atribuirile non-blocante (<=) nu blochează execuția altor instrucțiuni și permit ca toate instrucțiunile dintr-un bloc procedural să fie evaluate în paralel. Ele sunt folosite în principal pentru a modela circuite secvențiale, cum ar fi registrele și flip-flop-urile, unde valorile sunt actualizate la un anumit eveniment (de exemplu, la un front de ceas).

Un exemplu de utilizare a atribuirilor non-blocante în cadrul unui bloc `always`, pentru a descrie logica secvențială a unui circuit sincronizat cu ceasul, este prezentat mai jos:

```
always @(posedge clk) begin
    a <= b + 1;
    c <= a * 2; // Valoarea **veche** a lui ‘a‘ (de la ciclul anterior)
                // este folosită pentru a calcula ‘c‘.
end
```

Așadar atribuirile blocante se utilizează în cadrul blocurilor `always` care modelează circuite combinaționale, iar cele non-blocante în cadrul blocurilor `always` care modelează circuite secvențiale.

11.3. Desfășurarea lucrării

Pentru fiecare tip de descriere a modulelor în Verilog, se vor rula exemplele practice fie în mediul de dezvoltare online EDA Playground (https://www.edaplayground.com/), fie în mediul https://www.tutorialspoint.com/compilers/online-verilog-compiler.htm.
Pentru fiecare exemplu, se va crea un fișier Verilog care să conțină implementarea și un testbench pentru verificarea funcționalității.

11.3.1. Implementarea unui demultiplexor 1:4 folosind descriere comportamentală

Se cere implementarea unui demultiplexor 1:4 folosind descriere comportamentală în Verilog. Demultiplexorul primește un semnal de intrare și, în funcție de valorile selectorilor, transmite acest semnal către una dintre cele patru ieșiri.

Un demultiplexor este un circuit logic care primește un singur semnal de intrare și îl direcționează către una dintre mai multe ieșiri, în funcție de valorile unor semnale de selecție. În cazul unui demultiplexor 1:4, există o singură intrare și patru ieșiri, iar doi biți de selecție determină care dintre cele patru ieșiri va primi semnalul de intrare. Aceste se utilizează în aplicații precum rutarea semnalelor, distribuția datelor și în sistemele de comunicații.

Se va completa codul de mai jos în cadrul blocului `always` folosind o construcție `case` pentru a direcționa semnalul de intrare către ieșirea corespunzătoare. După implementare, se va rula testbench-ul pentru a verifica funcționarea corectă a demultiplexorului.

```
% Exemplu Verilog: Demultiplexor 1-la-4
module demux1to4(
    input din,           // semnal de intrare
    input [1:0] sel,     // selectori
    output reg d0,       // ieșiri
    output reg d1,
    output reg d2,
    output reg d3
);

    always @(*) begin
        // Resetare ieșiri
        d0 = 0; d1 = 0; d2 = 0; d3 = 0;
        // TODO: adăugați logica demultiplexorului folosind case
    end
endmodule

// test bench pentru demultiplexorul 1-la-4
module tb_demux1to4;
```

```
    reg din;
    reg [1:0] sel;
    wire d0, d1, d2, d3;

    demux1to4 uut (
        .din(din),
        .sel(sel),
        .d0(d0),
        .d1(d1),
        .d2(d2),
        .d3(d3)
    );

    initial begin
        din = 1;
        sel = 2'b00; #10;
        $display("sel=00, d0=%b, d1=%b, d2=%b, d3=%b", d0, d1, d2, d3);

        sel = 2'b01; #10;
        $display("sel=01, d0=%b, d1=%b, d2=%b, d3=%b", d0, d1, d2, d3);

        sel = 2'b10; #10;
        $display("sel=10, d0=%b, d1=%b, d2=%b, d3=%b", d0, d1, d2, d3);

        sel = 2'b11; #10;
        $display("sel=11, d0=%b, d1=%b, d2=%b, d3=%b", d0, d1, d2, d3);

        din = 0; sel = 2'b00; #10;
        $display("din=0, sel=00, d0=%b, d1=%b, d2=%b, d3=%b", d0, d1, d2, d3);

        $finish;
    end
endmodule
```

Demultiplexorul 1:4 primește un semnal de intrare (din) și, în funcție de valorile selectorilor (sel), transmite acest semnal către una dintre cele patru ieșiri (d0-d3). Celelalte ieșiri rămân la 0. Testbench-ul verifică funcționarea pentru toate combinațiile de selectori și pentru ambele valori posibile ale semnalului de intrare.

11.3.2. Implementarea unui convertor BCD la afișaj de 7 segmente

Utilizându-se una dintre tipurile de descriere, se va implementa un convertor BCD (Binary-Coded Decimal) la afișaj de 7 segmente. Afișajul de 7 segmente este un dispozitiv electronic utilizat pentru a afișa cifrele zecimale și unele litere, fiind compus din șapte segmente luminoase care pot fi aprinse sau stinse pentru a forma diferite caractere.

Un convertor BCD este un circuit care primește o valoare BCD pe 4 biți (reprezentând cifrele de la 0 la 9) și generează semnalele necesare pentru a controla un afișaj de 7 segmente, astfel încât să afișeze cifra corespunzătoare. Acest convertor este esențial în multe aplicații digitale, cum ar fi ceasurile digitale, calculatoarele și alte dispozitive care afișează informații numerice.

Deoarece trăim în mod natural într-o lume zecimală (baza 10), avem nevoie de o modalitate de a converti aceste numere zecimale într-un mediu binar (baza 2), pe care calculatoarele și dispozitivele electronice digitale îl înțeleg, iar codul BCD ne permite să facem acest lucru.

Am văzut anterior că un cod binar pe n biți este un grup de n biți care poate avea până la 2^n combinații distincte de 1 și 0. Avantajul sistemului BCD este că fiecare cifră zecimală este reprezentată de un grup de patru cifre binare (biți), într-un mod similar cu sistemul hexazecimal. Astfel, pentru cele 10 cifre zecimale (09) avem nevoie de un cod binar pe patru biți.

Principalul avantaj al codului BCD este că permite o conversie ușoară între forma zecimală (baza 10) și cea binară (baza 2). Totuși, dezavantajul este că acest cod este ineficient, deoarece stările dintre 1010 (zecimal 10) și 1111 (zecimal 15) nu sunt folosite. Cu toate acestea, codul BCD are numeroase aplicații importante, în special pentru afișaje digitale.

În sistemul de numerotare BCD, un număr zecimal este separat în grupuri de câte patru biți pentru fiecare cifră zecimală din număr. Fiecare cifră zecimală este reprezentată prin valoarea sa binară ponderată, realizând o translație directă a numărului. Astfel, un grup de patru biți reprezintă fiecare cifră zecimală afișată, de la 0000 pentru zero până la 1001 pentru nouă.

De exemplu, 357_{10} (Trei sute cincizeci și șapte) în zecimal ar fi prezentat în cod BCD ca:

357_{10} = 0011 0101 0111 (BCD)

Putem observa că BCD folosește o codificare ponderată, deoarece fiecare bit binar din grupul de 4 biți reprezintă o anumită pondere a valorii finale. Cu alte cuvinte, BCD este un cod ponderat, iar ponderile folosite în codul BCD sunt 8, 4, 2, 1, denumit frecvent codul 8421, deoarece formează reprezentarea binară pe 4 biți a cifrei zecimale relevante.

Așadar, se cere implementarea unui modul Verilog care să convertească o valoare BCD pe 4 biți într-un semnal de control pentru un afișaj de 7 segmente. Se poate utiliza oricare dintre descrierile prezentate, specifice limbajului Verilog (structurală, la nivel de flux de date sau comportamentală).

Mai multe detalii despre implementare se vor furniza în timpul orelor de laborator.

Pentru simulare se recomandă utilizarea mediului online EDA Playground (https://www.edaplayground.com/).

11.4. Conținutul referatului

1. Prezentarea conceptelor fundamentale ale limbajelor de descriere hardware, cu accent pe structura, sintaxă și utilizare practică;
2. Exemplificarea procesului de implementare și simulare a circuitelor electronice folosind Verilog, incluzând etapele de scriere a codului și testare.
3. Analiza comparativă a diferitelor tipuri de descriere în Verilog (structurală, la nivel de flux de date, comportamentală), evidențiind avantajele și dezavantajele fiecăreia.
4. Descrierea modului de execuție și simulare al codului Verilog, incluzând rolul testbench-ului.
5. Explicarea modului de funcționare al blocurilor `initial` și `always` în Verilog.
6. Discutarea diferențelor dintre atribuiri blocante și non-blocante în Verilog și când se utilizează fiecare tip.

11.5. Verificarea cunoştinţelor

1. Enumerați tipurile de descriere utilizate în Verilog și explicați diferențele dintre acestea.
2. Dați exemple de cod pentru fiecare tip de descriere și comentați modul de funcționare.
3. Explicați modul de execuție și simulare al codului Verilog, incluzând rolul testbench-ului.
4. Descrieți modeul de funcționare al blocurilor `initial` și `always` în Verilog.
5. Care sunt diferențele dintre atribuiri blocante și non-blocante în Verilog și când se utilizează fiecare tip?

Bibliografie

[1] Cadence. Orcad capture (2024). URL https://www.cadence.com/en_US/home/tools/pcb-design-and-analysis/orcad/orcad-capture.html.

[2] F. Moldoveanu, D. F. *Circuite logice şi comenzi secvențiale; Circuite logice combinaționale* (Editura Universității Transilvania, Braşov, 2003).

[3] Nicula, D. Electronică digitală - carte de învățatură 2.0 (2015). URL https://www.dannicula.ro/ed_ci/.

[4] Anghel, S. D. *Bazele electronicii analogice si digitale* (Presa Universitară Clujeană, Cluj-Napoca, 2007).

[5] Zet, C. Circuite numerice (2022). URL http://iota.ee.tuiasi.ro. Resources at "http://iota.ee.tuiasi.ro/ cn/".

[6] Nicula, D. Verilog - carte de învățatură (2019). URL https://dannicula.ro/books/verilog/.

[7] Dicu, T. Arhitectura calculatoarelor (2023). URL https://ocw.cs.pub.ro/courses/ac-is.