Proiectarea cu dispozitive programabile

Introducere

Clasificarea circuitelor integrate

Dispozitivele numerice se realizează în baza circuitelor integrate (CI).

După numărul de tranzistoare pe capsulă CI se clasifică în: SSI, MSI, LSI, VLSI şi ULSI.

Denumire	Notaţie	Anul apariţiei	Tehnologie	Dimensiune tranzistor	Număr de tranzistoare	Exemple
Small Scale Integration	SSI	1960	bipolară	10 μm	≤ 50	porţi logice
Medium Scale Integration	MSI	1965	bipolară MOS	8 μm	≤ 500	MUX, DMUX, registre
Large Scale Integration	LSI	1970	bipolară MOS	5 μm	≤ 10000	memorii, μP Z80
Very Large Scale Integration	VLSI	1980	bipolară MOS	1,5 - 3 μm	≤ 1000000	memorii, μcontrol, μP 486
Ultra Large Scale Integration	ULSI	1990	bipolară MOS	0,13 - 1 μm	> 1000000	calculator, transputer Pentium II, III, 4

După gradul de specificare al circuitelor folosite într-o aplicaţie, CI se clasifică în:

1. Circuite standard sau de uz general (de la SSI până la ULSI),
2. Circuite semidedicate (circuite logice programabile – PLD Programmable Logic Devices
3. Circuite dedicate –ASIC Application Specific Integration Circuit

Proiectarea cu circuite standard permite utilizarea eficientă a ariei de siliciu, dar are şi un şir de desavantaje:

- consum mare de energie;
- disipare mare de căldură;
- număr foarte mare de interconexiuni;
- viteza de lucru relativ scăzută, datorită conexiunilor externe şi a punctelor de sudură;
- fiabilitatea scade proporţional cu creşterea complexităţii;
- testarea şi depanarea dificilă.

Proiectarea cu circuite dedicate ASIC se face pentru un număr foarte mic de aplicaţii, sau chiar pentru una singură.

Pentru o aplicaţie dată, ASIC-urile obţin performanţe foarte bune, ocupă un spaţiu mult mai mic şi consumă puţină energie. În cazul unor producţii de serie mari, preţul de cost unitar devine foarte bun.

Dezavantajele:

- Durata mare a ciclului de dezvoltare a circuitului.
- Preţul ridicat al proiectării circuitului integrat (30 000$ - 2 500 000$)
- Sunt excluse optimizări post-design.
- Sunt puţin flexibile.

Circuitele programabile PLD au o structură foarte generală, configurabilă de utilizator. Termenul programare presupune configurarea circuitului la nivel fizic.

Avantajele proiectării cu PLD:

- Permit implementarea circuitelor specializate direct în hardware.
- Risc scăzut în faza de proiectare.
- Cost iniţial redus.
- Timp de proiectare redus.
- Permit optimizări post-design.

<u>Clasificarea PLD</u>

Termenul de circuit logic programabil sau ***PLD (Programmable Logic Devices)*** este un termen general care se referă la orice tip de circuit integrat care poate fi configurat la nivel fizic (programat) de către utilizator pentru implementarea unui proiect. Majoritatea acestor dispozitive folosesc tehnologii ce permit reprogramarea funcţiilor, ceea ce înseamnă că erorile de proiectare pot fi corectate fără a înlocui dispozitivul sau a modifica fizic conexiunile.

Unul dintre cele mai folosite circuite logice programabile a fost memoria de tip ROM programabilă o singură dată ***(PROM).***

Plecând de la această arhitectură s-au dezvoltat ***ariile logice programabile*** de tip ***PLA (Programmable Logic Array)*** dedicate implementării funcţiilor logice. În circuitele PLA atât aria de porţi ŞI cât şi cea de porţi SAU sunt programabile. Logica programată a devenit mai populară la mijlocul anilor `70 odată cu apariţia ariilor logice programabile de tip ***PAL-uri (Programmable Array Logic).*** Acest tip de arhitectură combină o arie programabilă de porţi ŞI cu o arie de porţi SAU fixă. Aceste circuite constituie varianta MSI (Medium Scale Integration) a producţiei PLD.

Capacitatea în continuă creştere a circuitelor integrate a oferit producătorilor ocazia de a realiza PLD tot mai mari (Variantele LSI şi VLSI) . Astfel au apărut PLD complexe (CPLD). Un CPLD este un ansamblu format din mai multe circuite PAL şi o structură de interconectare, toate realizate pe acelaşi chip.

Cam în aceeaşi perioadă de timp, alţi producători de CI au abordat în mod diferit problema extinderii dimensiunilor chipurilor. Astfel au apărut circuitele ***FPGA (Field Programmable Gate Array.*** Circuitele FPGA conţin un număr mare de blocuri logice amplasate în formă de matrice bidimensională şi o structură de interconectare amplasată în jurul fiecărui bloc.

Astfel circuitele PLD pot fi clasificate în:

SPLD (Simple Programmable Logic Device) **PROM, PLA, PAL**
CPLD (Complex Programmable Logic Device)
FPGA (Field Programmable Gate Array)

Circuitele PLD permit reducerea etapelor de trecere din faza de proiect la cea de prototip şi apoi în producţie.

Proiectarea circuitelor PLD se efectuează prin intermediul mijloacelor CAD (Computer Aided Design), care permit proiectarea fie în mod schematic, fie cu limbaje de descriere hardware HDL (Hardware Description Language): VHDL, Verilog, ABEL (Advanced Booleean Expression Language), AHDL.

Circuite logice programabile simple (SPLD)

Memorii ROM

Memoria ROM este un circuit combinaţional care stochează permanent (memorie nevolatilă) informaţia binară, iar această informaţie poate fi numai citită.

Memoria PROM (Programmable ROM) poate fi programată o singură data.

Metode de programare: prin măşti (la etapa de fabricare) sau fuzibile (de către utilizator).

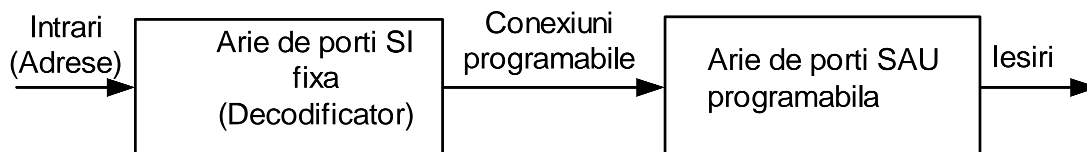
Fuzibilul este o peliculă subţire de CrNi, care se vaporizează la trecerea unui curent suficient prin el. Programarea constă în selecţia adresei şi a liniei de date şi aplicarea unui impuls de tensiune (10-30 V) pe un pin special de programare.

Tehnologia de fabricare: bipolară sau MOS.

Memoria EPROM (Erasable PROM) poate fi programată de mai multe ori, pentru că memoria poate fi „ştersă” prin expunere ala raze ultraviolete. Matricea de memorie conţine tranzistoare MOS cu poartă flotatntă FAMOS (Floating Avalanche MOS).

Memoria EEPROM (Electrically EPROM) poate fi ştearsă pe cale electrică.

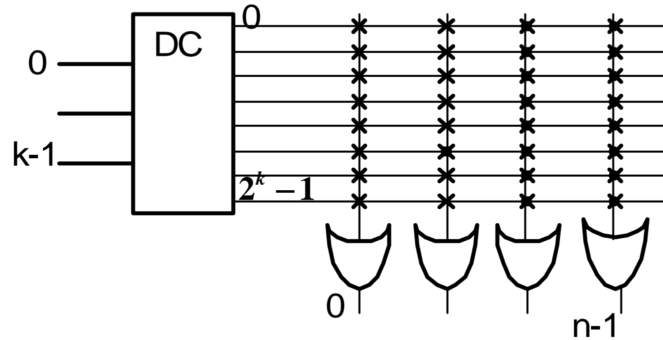
Memoria FLASH este o memorie EEPROM de capacitate mare



Memorie PROM

O memorie ROM $k \times n$ conţine:
k linii de intrare (adrese)
Decodificator (porţi ŞI) $k \to 2^k$
n elemente SAU cu 2^k intrări fiecare

Decodificatorul este conectat la toate cele n porţi SAU prin fuzibile. Astfel sunt $2^k \times n$ conexiuni programabile.

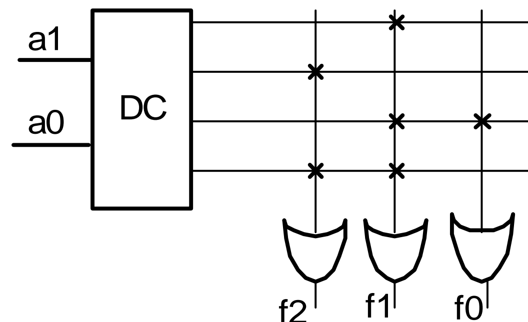


Programarea ROM se face conform sumei canonice. Nu este necesară minimizarea.
Exemplu. Combinația de intrare din tabelul de adevăr se aplică la intrările decodificatorului, Valorile funcțiilor constituie conținutul memoriei.

0 – fuzibilul se arde

1 – fuzibilul rămîne intact.

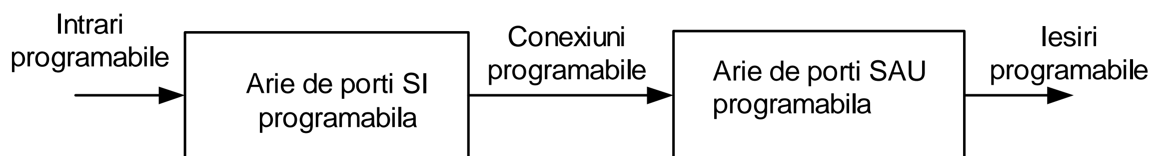
a1	a0	f2	f1	f0
0	0	0	1	0
0	1	1	0	0
1	0	0	1	1
1	1	1	1	0



Dezavantajul implementării funcțiilor logice cu memorii ROM este creşterea foarte mare a capacității memoriei odată cu creşterea numărului de intrări în circuit datorită rigidității posibilităților de adresare.

Circuite PLA

Circuitele PLA conțin 2 nivele de logică, o arie de porți ŞI şi o arie de porți SAU, ambele programabile.



Circuit PLA

Dimensiunea unui dispozitiv PLA este dată de: numărul de intrări n, numărul de ieşiri m şi numărul de termeni produs p. Numărul termenilor produs este mult mai mic decât numărul de mintermeni de n variabile (2^n).

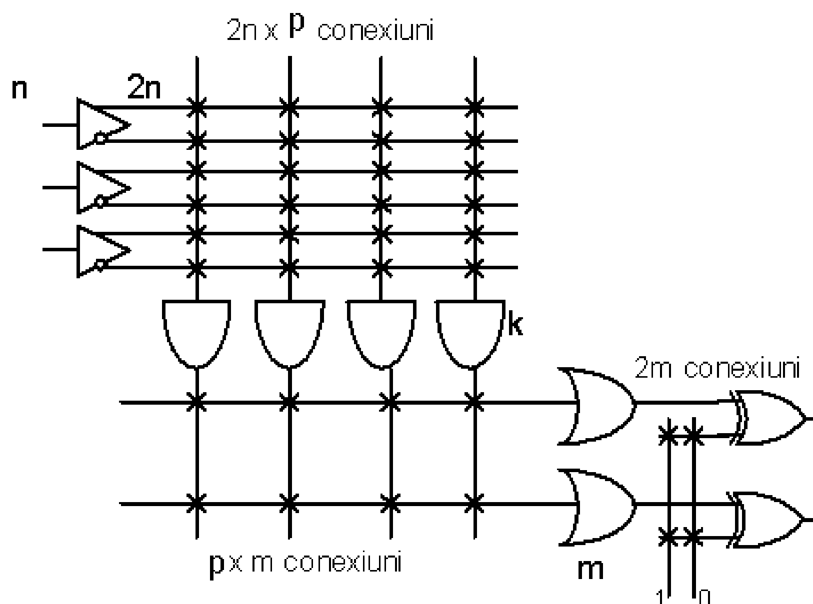
Porțile ŞI au câte $2n$ intrări care pot fi conectate prin programare la oricare din cele n variabile de intrare, sub formă directă sau complementată.

Porțile SAU au câte p intrări care pot fi conectate prin programare la ieşirile oricărei porți ŞI.

Ieşirile porților SAU sunt conectate la câte o poartă XOR cu o intrare programabilă, la care se aplică 0 pentru a obține funcția în formă directă, şi 1 – pentru a obține funcția complementată.

Circuitul permite implementarea unui număr de $\boldsymbol{m}$ funcții logice, fiecare de câte $\boldsymbol{n}$ variabile, cu condiția că numărul termenilor produs să nu-l depăşească pe p.

Astfel circuitul conține $\boldsymbol{2n \times p + p \times m + 2m}$ conexiuni programabile.

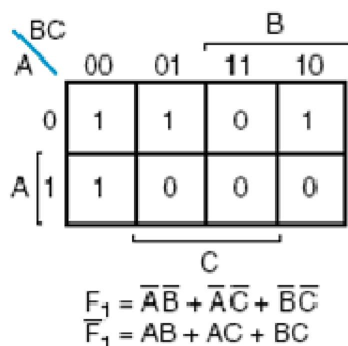


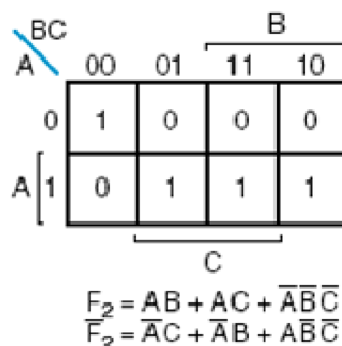
Pentru implementarea funcțiilor logice în PLA este necesară minimizarea lor şi elaborarea tabelului de programare. Tabelul de programare specifică termenii produs şi termenii sumă pentru funcțiile care urmează a fi implementate.

Exemplu. Fie dat tabelul de adevăr pentru funcțiile F1 şi F2

A	*B*	*C*	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Se efectuează minimizarea funcțiilor.





Analizînd expresiile minimizate se aleg acelea care permit utilizarea termenilor comuni. Acestea sunt:

$F_1' = AB + AC + BC$ or $F_1 = (AB + AC + BC)'$

$F_2 = AB + AC + A'B'C'$

Astfel vom folosi doar 4 termeni produs :

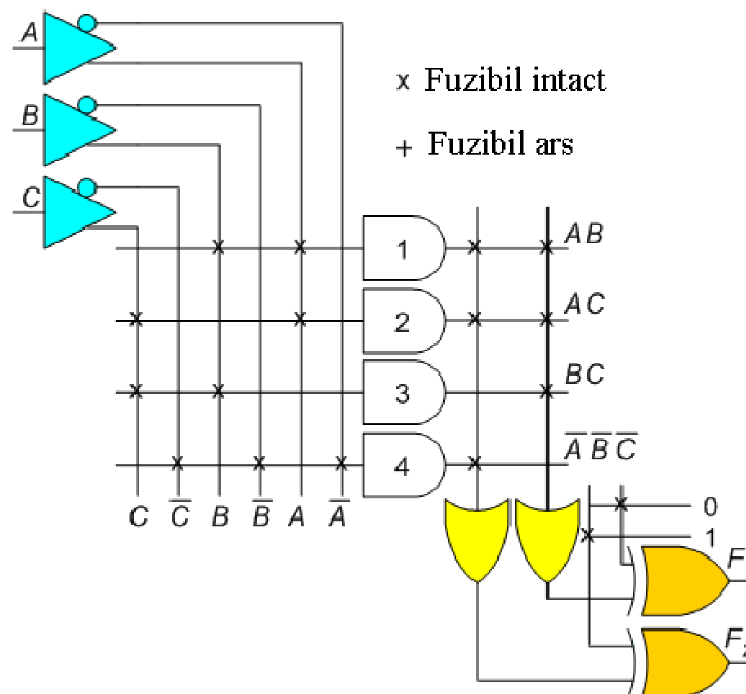
AB, AC, BC, and $A'B'C'$.

Tabelul de programare a PLA va arăta astfel:

Tabelul de programare

	Termeni produs	Intrări A B C	Ieșiri (C) F_1	(T) F_2
AB	1	1 1 –	1	1
AC	2	1 – 1	1	1
BC	3	– 1 1	1	–
$\bar{A}\bar{B}\bar{C}$	4	0 0 0	–	1

Circuitul PLA programat:

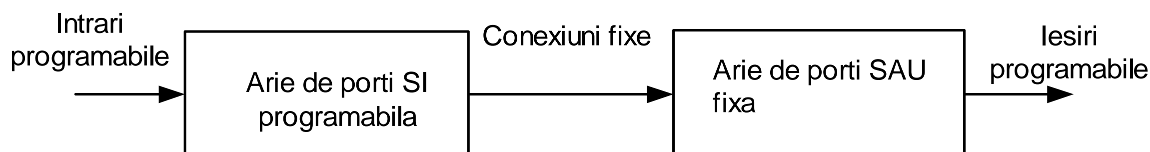


Exemplu de dispozitiv PLA este Signetics 82S100, apărut la jumătatea anilor 70. Dispozitivul are 16 intrări, 48 porți ŞI și 8 ieșiri. Astfel el are 2x16x48=1536 conexiuni fuzibile în matricea de porți ŞI și 8x48=384 în matricea de porți SAU.

Cu toate că circuitele PLA sunt foarte flexibile, ele nu şi-au găsit o aplicare practică mare, fiindcă este necesară programarea pe două nivele – ŞI şi SAU.

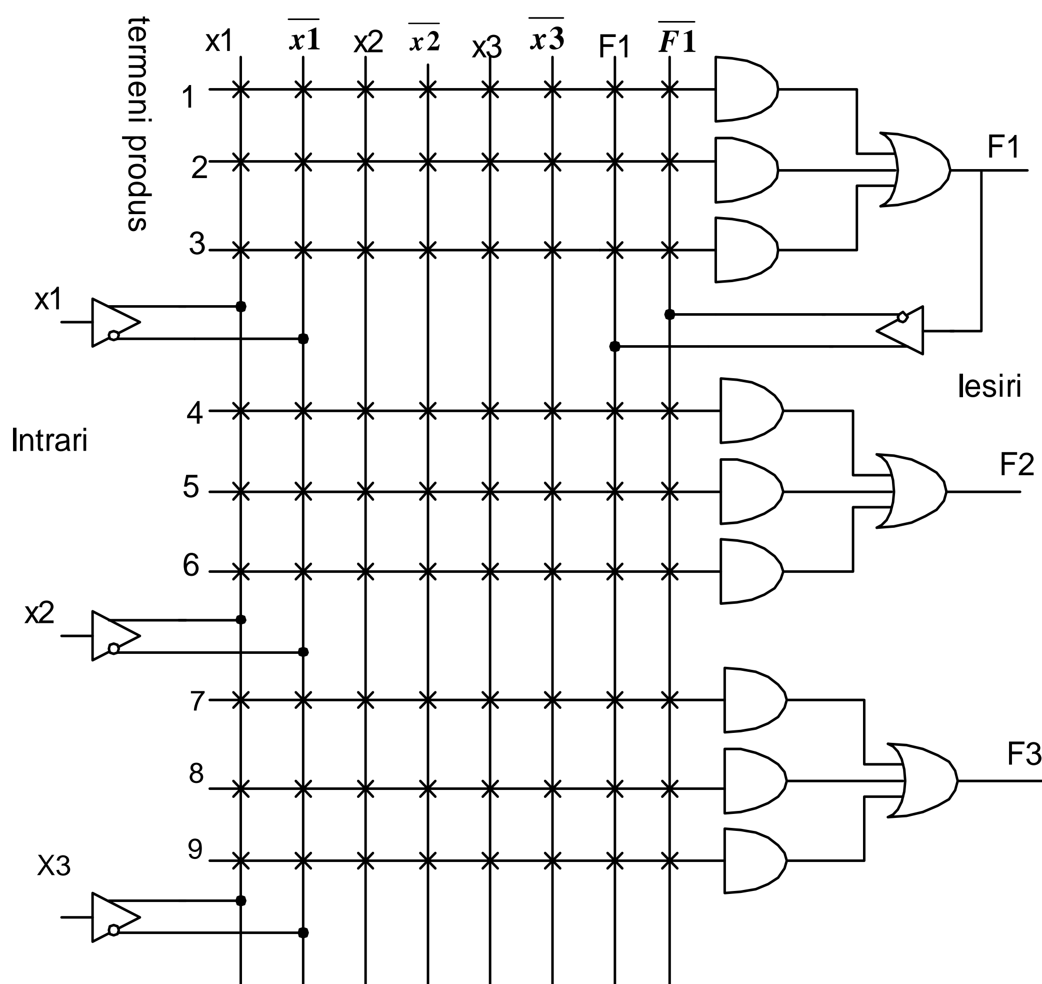
Circuite PAL

Circuitele PAL conțin 2 nivele de logică, o arie de porți ŞI programabilă şi o arie de porți SAU fixă.



Circuit PAL

Circuitele PAL sunt mai uşor programabile dar nu sunt atât de flexibile, ca circuitele PLA.



Circuitul prezentat conține 3 intrări şi 3 ieşiri. Fiecare intrare e conectată la o poartă buffer (tampon)/inversor. Fiecare ieşire este generată de o poartă SAU fixă. Fiecare secțiune conține 3 porți ŞI programabile cu 6 intrări. Ieşirea F1 poate fi programată ca intrare la porțile ŞI.

Numărul porților ŞI în fiecare secțiune este fix, dar dacă funcția conține mai mulți termeni produs, există posibilitatea de a utiliza mai multe secțiuni.

Pentru implementarea funcțiilor logice în PAL este necesară minimizarea lor şi elaborarea tabelului de programare. Tabelul de programare specifică doar termenii produs pentru funcțiile care urmează a fi implementate.

Exemplu.

$$F_1 = \Sigma(1,3,4,5,7)$$

$$F_2 = \Sigma(1,3,4,6,7)$$

$$F_3 = \Sigma(1,2,3,4,5,7)$$

Se efectuează minimizarea funcțiilor:

F1

x3 \ x1 x2	00	01	11	10
0				1
1	1	1	1	1

F2

x3 \ x1 x2	00	01	11	10
0			1	1
1	1	1	1	

F3

x3 \ x1 x2	00	01	11	10
0		1		1
1	1	1	1	1

$$F_1 = x_1\overline{x}_2 + x_3$$

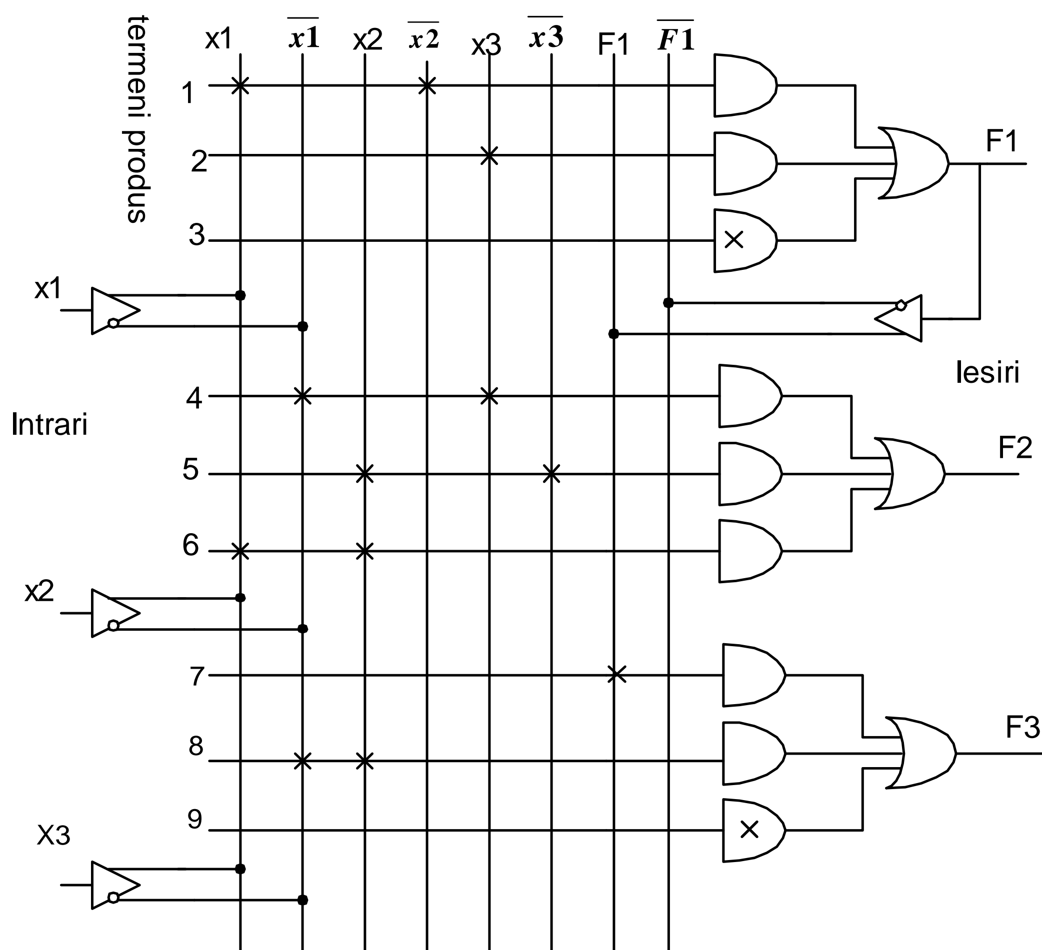
$$F_2 = \overline{x}_1 x_3 + x_2\overline{x}_3 + x_1 x_2$$

$$F_3 = x_1\overline{x}_2 + x_3 + \overline{x}_1 x_2 = F_1 + \overline{x}_1 x_2$$

Tabelul de programare a PLA va arăta astfel:

Termeni produs	Intrari pentru porti SI				Iesiri
	x1	x2	x3	F1	
1	1	0	-	-	
2	-	-	1	-	$F_1 = x_1\bar{x}_2 + x_3$
3	-	-	-	-	
4	0	-	1	-	
5	-	1	0	-	$F_2 = \bar{x}_1 x_3 + x_2\bar{x}_3 + x_1 x_2$
6	1	1	-	-	
7	-	-	-	1	
8	0	1	-	-	$F_3 = F_1 + \bar{x}_1 x_2$
9	-	-	-	-	

Circuitul PAL programat:



În prezent una din cele mai utilizate structuri de PLD combinaţionale este PAL16L8, compania AMD (Advanced Micro Devices). Dispozitivul are 16 intrări, 64 de porţi ŞI, divizate în 8 secţiuni, 8 porţi SAU pentru generarea celor 8 ieşiri. Fiecare poartă ŞI are 32 intrări pentru variabilele de intrare în formă directă şi inversă.

Circuitele PAL secvenţiale conţin bistabile de tip D (ex. PAL16R8) sau macrocelule (ex. GAL16V8 cu ştergere electrică – Generic Array Logic, firma

Lattice Semiconductor). O macrocelulă este formată din bistabile care pot funcţiona în regimurile D şi T şi câteva multiplexoare pentru alegerea regimului (combinaţional, secvenţial, ieşire directă sau inversă).

Avantajele oferite de circuitele PLD sunt: consumul redus de putere, performanţe mai bune datorită lungimi mult reduse a interconexiunilor şi o fiabilitate mai ridicată.

În prezent circuitele SPLD se utilizează ca componente de reţea care cer performanţe ridicate per ansamblu: hub-uri de reţea bridge-uri, routere, produse din zona telefoniei mobile, video game-urilor şi a web browserelor.

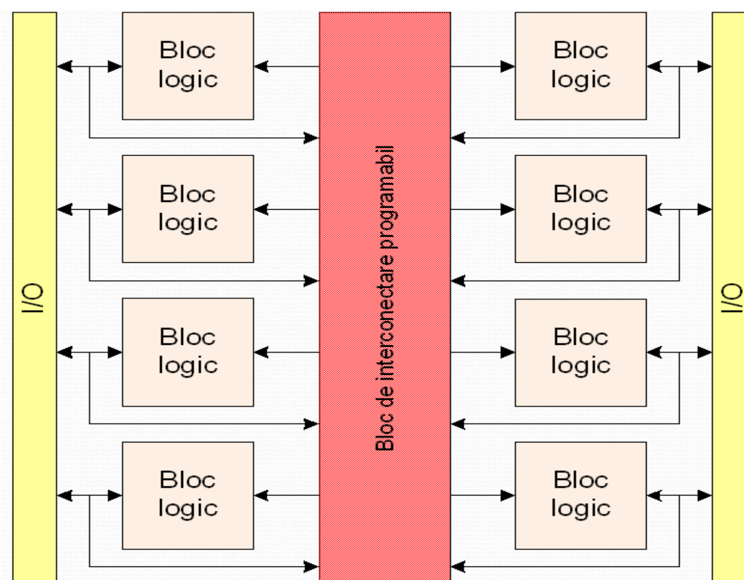
Producători: AMD, Philips Semiconductors , Lattice Semiconductor.

Dispozitive logice programabile complexe (CPLD)

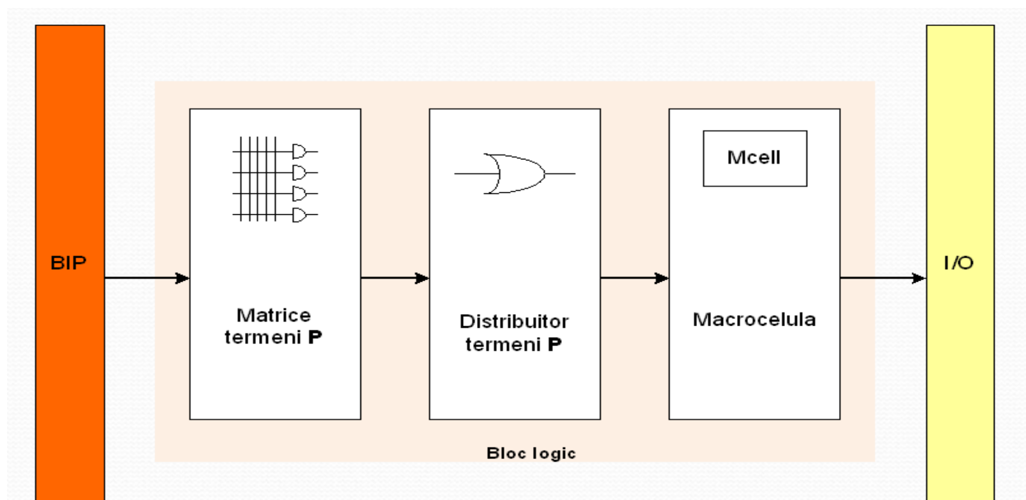
Un CPLD este un ansamblu de PLD separate, realizate pe acelaşi cip şi însoţite de o structură de interconectare programabilă.

Circuitele CPLD sunt circuite VLSI ale căror părţi componente sunt:

- PLD (PAL, GAL), care formează blocurile logice (funcţionale);
- Bloc de interconectare programabil;
- Blocuri de intrare/ieşire



Structura blocului logic diferă la diverşi fabricanţi, dar în general include matricea termenilor produs (ŞI), distribuitorul de termeni produs (SAU) şi macrocelule.



Matricea SAU (distribuitorul de termini produs) nu este complet fixă şi permite de a utiliza aceiaşi termeni pentru diferite funcții.

Macrocelula reprezintă un bistabil care poate fi programat sa lucreze în regim D sau T, multiplexoare pentru alegerea modului de lucru (combinațional sau secvențial), elemente XOR pentru a obține funcția în forma directă sau inversă.

Funcțiile logice simple pot fi implementate în cadrul uni singur bloc. Funcțiile mai complexe pot necesita mai multe blocuri, care vor fi interconectate prin matricea de rutare.

Circuitele din seria XC9500 de la Xilinx reprezintă o familie de CPLD cu arhitectură similară. Un bloc funcțional a acestor circuite conține până la 36 intrări şi 18 ieşiri, 18 macrocelule cu câte un bistabil. Funcția logică poate conține până la 90 termeni produs.

Circuitele CPLD folosesc tehnologia EECMOS (Electrically Erasable Complementary metal–oxide–semiconductor) şi UVEPROM. Programarea se face prin încărcarea unui cod binar (bitstream) în circuit prin cablul JTAG.

Avantajele CPLD

1. Preț de cost redus; Dimensiuni mici; Consum de energie redus; Fiabilitate înaltă a schemelor; Viteză de lucru mare; Ciclul de viață a proiectelor redus;
2. Sunt reprogramabile;
3. Nonvolatile (conținutul nu se pierde odată cu deconectarea de la sursă);
4. Securitate mare a informației (Biții de securitate nu pot fi resetați decât prin ştergerea circuitului ceea ce duce implicit la pierderea proiectului.)
5. Timpul de reținere a schemei este previzibil, deoarece matricea de interconexiune are o mărime fixă.

Producători de circuite CPLD

Firma **Xilinx**: familia de circuite XC9500, CoolRunner II.

Firma **Vantis** (firmă subsidiară a firmei **AMD**): familia de circuite **PAL**, **MACH** (Macro Array CMOS High).

Firma **Altera,** familia de circuite **MAX** (Multiple Array Matrix)**: MAX 5000, MAX 7000, MAX 9000.** O altă familie de circuite logice programabile care îmbină caracteristicile circuitelor FPGA (număr mare de regiştri) cu cele ale

circuitelor EPLD (viteză mare şi întârzieri previzibile) este familia **FLEX** (Flexible Logic Element Matrix) cu membrii **FLEX 6000, FLEX 8000 şi FLEX 10k**.
Cypress Semiconductor cu familiile de circuite CY7C34x, FLASH 370i;
Lattice Semiconductor cu familiile ispLSIxxxx şi GAL (ex. GAL 16V8, GAL 20V10 etc.);

Structura macrocelulei

Circuite FPGA

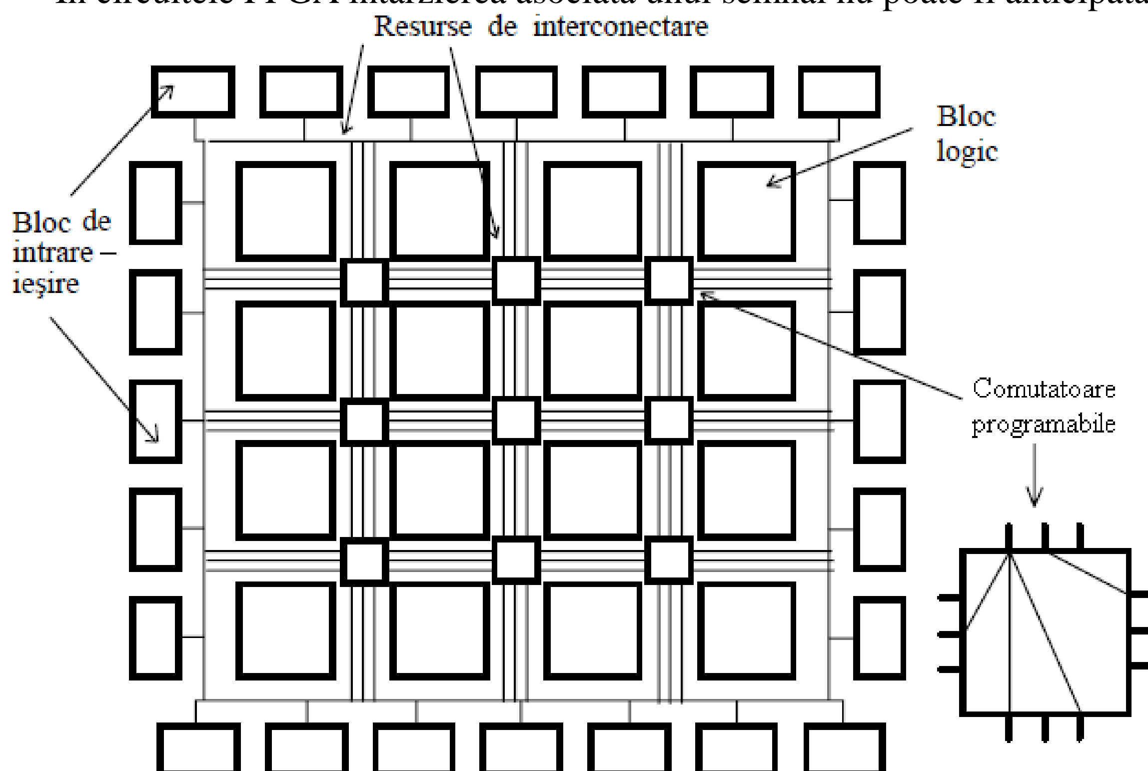
FPGA sunt circuite VLSI cu următoarele componente principale:

- Blocuri logice configurabile CLB (Congigurable Logic Blocks), amplasate in formă de matrice bidimensională;
- matrice de comutatoare programabile, amplasate în jurul fiecărui CLB;
- blocuri de intrare/ieşire.

Toate componentele FPGA sunt programabile (reconfigurabile) de către utilizator.

Canalele şi blocurile de comutare dintre blocuri conţin resurse de interconectare. Aceste resurse conţin de obicei segmente de interconectare de diferite lungimi. Interconexiunile conţin comutatoare programabile cu rolul de a conecta blocurile logice la segmentele de interconectare, sau un segment de interconectare la altul. În plus, există celule de I/E la periferia reţelei, care pot fi programate ca intrări sau ieşiri.

În circuitele FPGA întârzierea asociată unui semnal nu poate fi anticipată.



În prezent sunt utilizate trei tehnologii de programare a circuitelor FPGA:

- **Antifuzibile.** (Un antifuzibil este un dispozitiv cu două terminale care în mod normal se află în starea de înaltă impedanţă, iar atunci când este expus la o

tensiune ridicată, trece în starea cu rezistență redusă (300-500 Ω).) **Avantaje:** nevolatile, consum mic de putere. **Dezavantaje:** pot fi programate o singură dată. Din categoria circuitelor FPGA cu antifuzibile fac parte circuitele firmelor Actel, Quicklogic, Cypress.

- **SRAM.** Programarea acestor circuite se realizează prin celule de memorie statică. Din această categorie de circuite FPGA fac parte cele ale firmelor Xilinx, Altera, AT&T. **Avantaje:** sunt reprogramabile, procesul de fabricare este standard. **Dezavantaje:** Sunt volatile, consum mai mare de putere, densitatea de integrare este mai mică, au o securitate redusă a informației, deoarece codul de configurare se păstrează pe o memorie ROM externă.

Totuşi avantajele tehnologiei SRAM sunt mai importante decât neajunsurile şi ele, de fapt, domină piața.

Din această categorie de circuite FPGA fac parte cele ale firmelor Xilinx, Altera, AT&T.

- **EPROM, EEPROM/FLASH** Tehnologia EEPROM/FLASH combină avantajele tehnologiilor precedente: Sunt nevolatile, reprogramabile, folosesc un proces de fabricație standard, consum redus de putere, securitatea sporită a informației.

Din această categorie de circuite FPGA fac parte cele ale firmelor, Altera, Actel, Lattice.

Blocuri logice configurabile

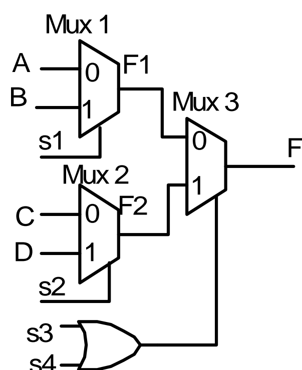
CLB furnizează resursele logice necesare implementării elementelor combinaționale şi secvențiale, blocurilor de calcul aritmetic, blocurilor de memorie de capacitate redusă ale unui sistem digital.

Părțile componente ale unui CLB sunt generatoarele de funcții logice, bistabile, de obicei de tip D şi multiplexoare pentru selectarea funcției dorite la ieşire.

Generatoarele de funcții logice pot fi de 2 tipuri:

1. În bază de multiplexor (ex.: Actel, QuickLogic). Tehnologia – antifuzibil.
2. În bază de blocuri LUT – look-up table (ex.:**Xilinx** - famiile XC4000, Spartan, Virtex; **Altera** - famiile Flex, Cyclone, Arria, Stratix; **Atmel** – AT6000, AT40K). Tehnologia: memorie SRAM

CLB realizate pe bază de multiplexoare



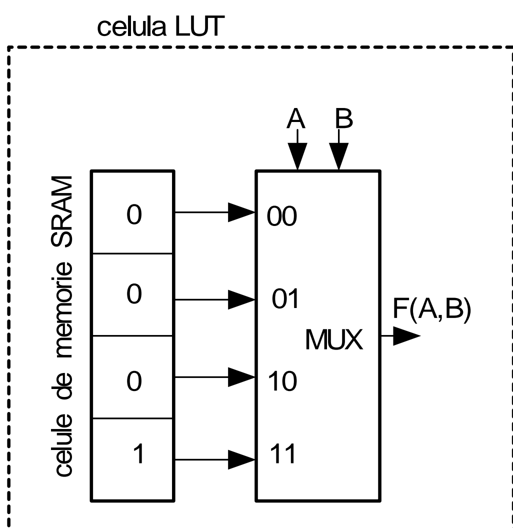
Intrările multiplexoarelor Mux 1 şi Mux 2 sunt conectate la semnale cu valori constante A,B şi C, D. Se pot implementa diferite funcţii logice ale căror variabile sunt intrările de selecţie s1, s2, s3 şi s4.

$$F = \overline{(s_3 + s_4)}\ F_1 + (s_3 + s_4)\ F_2$$

$$F_1 = \overline{s}_1 A + s_1 B$$

$$F_2 = \overline{s}_2 C + s_2 D$$

Celule logice realizate pe bază de blocuri LUT



Exemplu de celulă LUT de 2 biţi

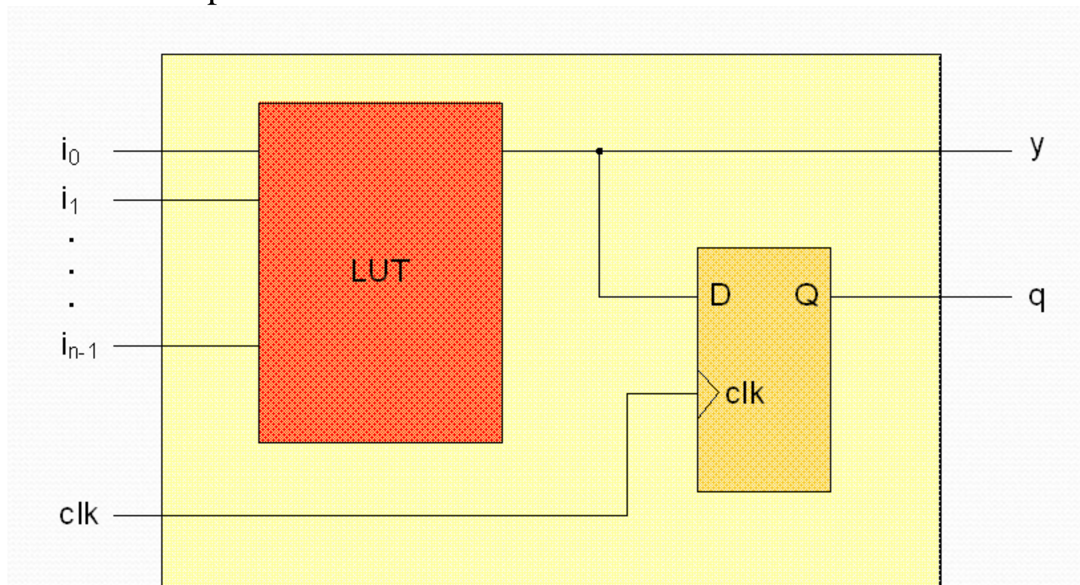
Numărul celulelor de memorie este 2^N

N – numărul de variabile ale funcţiei (2)

A	B	F(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

Programarea acestor circuite se realizează prin celule de memorie statică. Logica este implementată cu ajutorul unor tabele (*lookup table*) realizate din celulele de memorie, intrările funcţiilor controlând liniile de adresă. Fiecare tabelă de 2*n* celule de memorie implementează orice funcţie cu *n* intrări. Una sau mai multe tabele, combinate cu bistabile, formează un bloc logic configurabil.

Structura simplificată a unui CLB:



Se poate observa că ieşirea tabelei asociative poate fi transmisă direct la ieşire sau stocată, în prealabil, într-un bistabil.

Utilizarea circuitelor FPGA:

1. În calitate de circuite integrate, pentru accelerarea algoritmilor, în medii de calcul reconfigurabil;
2. În cadrul procesorului gazdă ca unitate funcțională pentru implementarea de instrucțiuni specializate;
3. Pe magistrala procesor-memorie cache, în calitate de coprocessor;
4. Pe magistrala memorie-subsistem de I/E, având rol de procesor ataşat;
5. Accesat prin interfața de I/E sau prin rețea, în calitate de unitate independentă de prelucrare.

Deosebirile dintre CPLD şi FPGA

1. Circuitele FPGA au o granularitate fină, ceea ce înseamnă că ele conțin o mulțime (până la 100 000) de blocuri mici cu bistabile. Circuitele CPLD au o granularitate grosieră, deoarece conțin un număr relativ mic (~ 100) de blocuri logice mari cu bistabile.
2. Majoritatea circuitelor FPGA sunt bazate pe memorii SRAM. Ele necesită o configurare de la o memorie ROM externă la fiecare racordare la rețea.
3. Circuitele FPGA au resurse speciale de rutare pentru a implementa eficient funcții aritmetice (numărărtoare, sumatoare, comparatoare), pe când circuitele CPLD nu le au.
4. În circuitele CPLD întârzierile pot fi calculate chiar în faza de proiectare, deoarece aceste circuite au o rețea de interconexiune fixă. În contrast cu CPLD-urile, FPGA-urile au o structură de interconexiuni formată din segmente de diferite lungimi. Numărul de segmente necesare pentru a conecta două celule logice nu este nici fix şi nici predictibil, deci întârzierile nu se pot cunoaşte decât după ce se face asignarea şi plasarea celulelor (după implementare).

Sisteme pe circuite integrate programabile - SOPC (System on programmable Chip)

Industria dispozitivelor FPGA este, în prezent, cea mai profitabilă dintre ramurile industriei electronice (conform unui studiu publicat de Forbes). Este, de asemenea, cea mai inovativă din punct de vedere tehnologic, actualmente cele mai noi procese de producție a semiconductorilor fiind testate pe dispozitive FPGA.

Odată cu creşterea nivelului de integrare a circuitelor au apărut arhitecturi care combină avantajele CPLD şi FPGA. Un exemplu este familia de circuite FLEX (Flexible Logic Element matriX), firma ALTERA. Aceste circuite, pe lângă blocurile logice, blocurile de intrare/ieşire şi rețeaua de interconectare programabilă, conțin şi blocuri SRAM încorporate (EAB –Embedded Array Block)

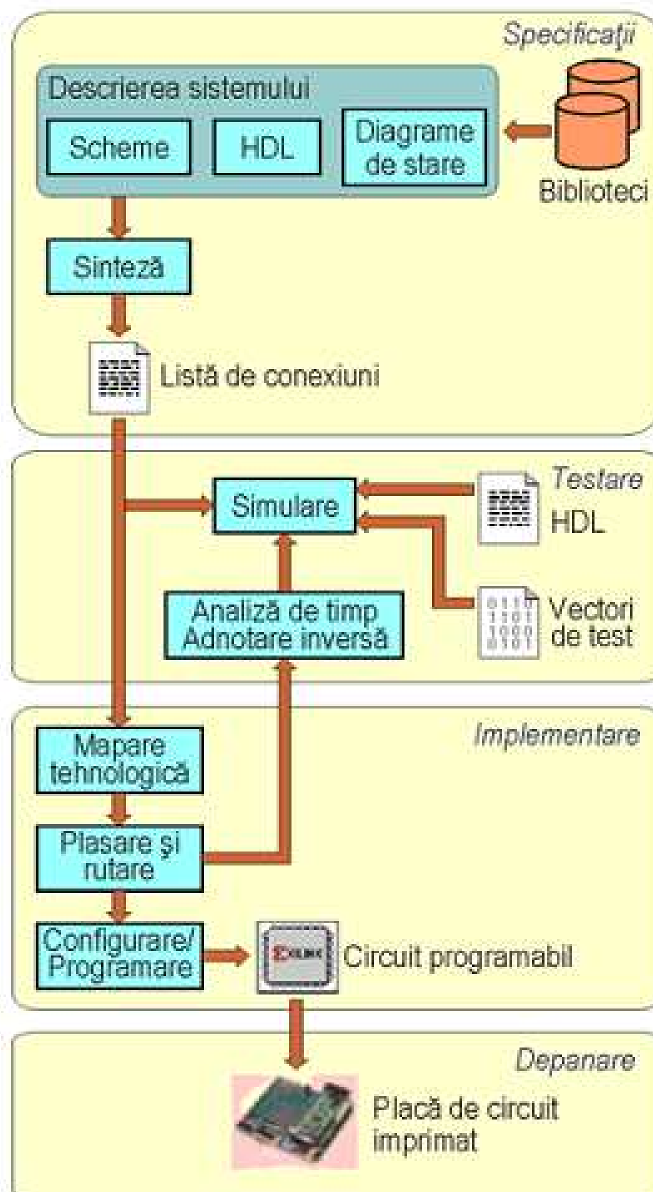
de dimensiune variabilã. Aceste arhitecturi sunt utilizate şi în circuitele integrate de tipul „sistem pe chip” – SOPC (system on programmable chip).

Părţile componente ale SOPC:

- Blocuri logice configurabile;
- matrice de interconectare;
- blocuri de intrare/ieşire;
- blocuri de memorie RAM încorporate;
- blocuri dedicate procesării digitale de semnal (DSP - ***D**igital **S**ignal **P**rocessing*), bazate pe multiplicatoare *hardware* şi acumulatoare;
- nuclee IP (Intellectual Property cores), care pot fi de 2 tipuri:
    - procesoare hard (microprocesoare sau microcontrolere în care sunt realizate anumite funcţii şi unităţi programabile;
    - procesoare soft cu structură uniformă şi posibilitate de reconfigurare a tuturor unităţilor din system.

Etapele de proiectare cu circuite programabile

Pentru proiectarea sistemelor digitale utilizând circuite programabile, cum sunt circuitele FPGA şi CPLD, se utilizează pachete de programe de proiectare asistată de calculator (CAD – *Computer Aided Design*). Aceste pachete de programe asistă proiectantul în toate etapele procesului de proiectare. Astfel, majoritatea pachetelor CAD pentru circuitele programabile asigură următoarele funcţii principale:



1. ***Specificarea*** (descrierea) sistemului digital; Principalele metode pentru descrierea sistemelor digitale:

- prin scheme logice,
- prin limbaje de descriere hardware (HDL – Hardware Description Language).

2. *Sinteza* descrierii, deci transformarea acesteia într-o listă de conexiuni conținând porți elementare şi interconexiunile dintre ele; Această translatare se realizează cu ajutorul unui program de sinteză din cadrul sistemului CAD.

Lista de conexiuni ("netlist") este o descriere compactă a sistemului digital sub formă textuală, în care sunt specificate componentele sistemului, interconexiunile dintre acestea şi pinii de intrare/ieşire. Această listă este prelucrată de celelalte componente ale sistemului CAD pentru realizarea etapelor următoare din cadrul procesului de proiectare.

3. ***Simularea*** funcţionării sistemului pe baza listei de conexiuni obţinute, înainte de implementarea într-un anumit circuit;

În această etapă se utilizează un program simulator pentru verificarea funcţionării sistemului proiectat. Această verificare se referă doar la aspectele funcţionale ale sistemului, fără a se lua în considerare întârzierile semnalelor, care vor fi cunoscute numai după implementare.

Pentru verificarea funcţională proiectantul furnizează simulatorului mai multe combinaţii ale valorilor semnalelor de intrare, o asemenea combinaţie fiind numită ***vector de test***. De asemenea, proiectantul poate specifica valorile semnalelor de ieşire care trebuie generate de sistem pentru fiecare vector de test.

4. ***Implementarea*** sistemului într-un circuit prin adaptarea listei de conexiuni pentru a se utiliza în mod eficient resursele disponibile ale circuitului;

a) Maparea tehnologică

Această etapă constă dintr-o serie de operaţii care realizează prelucrarea listei de conexiuni şi adaptarea acesteia la particularităţile şi resursele disponibile ale circuitului utilizat pentru implementare. Cele mai obişnuite operaţii sunt:

- adaptarea la elementele fizice ale circuitului,
- optimizarea şi verificarea regulilor de proiectare (de exemplu, testarea depăşirii numărului pinilor de I/E disponibili în cadrul circuitului). În timpul acestei etape, proiectantul selectează tipul circuitului programabil care va fi utilizat, capsula circuitului integrat, viteza şi alte opţiuni specifice circuitului respectiv.

În urma execuţiei operaţiilor din etapa de mapare tehnologică se generează un raport detaliat al rezultatelor tuturor programelor executate. Pe lângă mesaje de eroare şi de avertizare, se creează de obicei o listă cu resursele utilizate din cadrul circuitului.

b) Plasarea şi rutarea

Aceste operaţii sunt executate în cazul utilizării unui circuit FPGA pentru implementare.

Pentru proiectarea cu circuite CPLD, operaţia echivalentă este numită adaptare (“fitting”).

Plasarea este procesul de selectare a unor module sau blocuri logice ale circuitului programabil care vor fi utilizate pentru implementarea diferitelor funcţii ale sistemului digital.

Rutarea constă în interconectarea acestor blocuri logice utilizând resursele de rutare disponibile ale circuitului.

Majoritatea sistemelor CAD realizează operaţiile de plasare şi rutare în mod automat, astfel încât utilizatorul nu trebuie să cunoască detaliile arhitecturii circuitului utilizat pentru implementare.

c) Analiza de timp

Pachetele de programe CAD pentru proiectarea sistemelor digitale conțin de obicei un program numit analizor de timp, care poate furniza informații despre întârzierile semnalelor. Aceste informații se referă atât la întârzierile introduse de blocurile logice, cât şi la întârzierile datorate interconexiunilor.

Proiectantul poate utiliza informațiile despre întârzierile semnalelor pentru a realiza o nouă simulare a sistemului, în care să se țină cont de aceste întârzieri. Această operație prin care se furnizează simulatorului informații detaliate despre întârzierile semnalelor se numeşte adnotare inversă ("back-annotation").

Anumite sisteme permit utilizatorilor experți plasarea şi rutarea manuală a unor porțiuni critice ale sistemului digital pentru a obține performanțe superioare.

5. ***Configurarea*** **(programarea)** circuitului pentru ca acesta să realizeze funcția dorită.

Operația de configurare se referă la circuitele programabile bazate pe memorii volatile SRAM (Static Random Access Memory) şi constă din încărcarea informațiilor de configurare în memoria circuitului.

Operația de programare se referă la circuitele programabile bazate pe memorii nevolatile (cum sunt circuitele care conțin antifuzibile). Această operație se execută similar cu cea de configurare, dar informațiile de configurare sunt păstrate şi după întreruperea tensiunii de alimentare.

La sfârşitul operațiilor de plasare şi rutare, se generează un fişier care conține toate informațiile ce se referă atât la configurarea blocurilor logice ale circuitului, cât şi la specificarea interconexiunilor dintre blocurile logice. Fişierul în care se înscriu aceste informații conține, în principiu, un şir de biți **("bitstream")**, fiecare bit indicând starea închisă sau deschisă a unui comutator. Circuitele programabile conțin un număr mare de asemenea comutatoare, un comutator fiind realizat sub forma unui tranzistor sau a unei celule de memorie. Un bit de 1 din şirul de biți va determina închiderea unui comutator şi, deci, stabilirea unei conexiuni.

Biții din acest fişier de configurare sunt aranjați într-un anumit format pentru a realiza o corespondență între un bit şi comutatorul corespunzător.

Conținutul fişierului de configurare se transferă la circuitul programabil, aflat de obicei pe o placă de circuit imprimat împreună cu alte circuite.

Din cauza memoriei volatile, circuitul trebuie configurat din nou după fiecare întrerupere a tensiunii de alimentare. Informațiile de configurare pot fi păstrate într-o memorie nevolatilă PROM, existând posibilitatea configurării automate a circuitului din această memorie nevolatilă la aplicarea tensiunii de alimentare.

Configurarea sau programarea se pot realiza utilizând interfața paralelă a calculatorului. Pentru aceasta, este necesar ca placa cu circuitul programabil să conțină un conector pentru interfața paralelă, pentru transfer utilizându-se un cablu parallel (DB25). O altă posibilitate este utilizarea unui cablu special şi a unei metodologii de configurare propuse de organizația JTAG (Joint Test Advisory Group). Această metodologie este cunoscută şi sub numele de "Boundary-Scan".

6. Depanarea sistemului

În această ultimă etapă a procesului de proiectare se verifică funcționarea sistemului digital proiectat în condiții reale. O funcționare necorespunzătoare se poate datora nerespectării specificațiilor de proiectare, a specificațiilor circuitului utilizat pentru implementare, a unor aspecte legate de întârzierea semnalelor etc. Depanarea poate fi simplificată dacă circuitul se configurează astfel încât să conțină unele module speciale care permit citirea valorii unor semnale în timpul funcționării şi transferul acestor informații la calculator, utilizând un cablu JTAG şi un program special pentru vizualizarea semnalelor dorite.

Limbajul VHDL

În prezent se utilizează mai mule limbaje de descriere hardware HDL (Hardware Description Language): VHDL, Verilog, ABEL (Advanced Booleean Expression Language), AHDL.

Avantajele utilizării limbajelor de descriere hardware:

- Portabilitatea proiectelor pentru diferite tehnologii în care sunt realizate cipurile, în care urmează să se facă implementarea;
- Maniera de descriere a proiectului sub formă de cod îi conferă acestuia claritate mai bună decât dacă ar fi fost descris sub formă de schemă;
- Timp de proiectare redus;
- Opțiuni de optimizare a proiectului, cum ar fi cele de arie sau/şi viteză.
- Folosirea diferitelor construcții specifice HDL cum ar fi: pachetele (package) şi bibliotecile (library), permit reutilizarea lor în alte proiecte.

Limbajul VHDL a fost dezvoltat la mijlocul anilor '80 de către departamentul de apărare al Statelor Unite în colaborare cu IEEE (**I**nstitute of **E**lectrical and **E**lectronical **E**nginner). Limbajul a fost standardizat pentru prima dată de *IEEE* în 1987 (*VHDL-87*) şi a fost extins în 1993 (*VHDL-93*).

Abrevierea VHDL derivă din ***VHSIC H****ardware* ***D****escription Language* (limbajul de descriere hard VHSIC), abrevierea *VHSIC* însemnând la rândul ei ***V****ery* ***H****igh* ***S****peed* ***I****ntegrated* ***C****ircuit.*

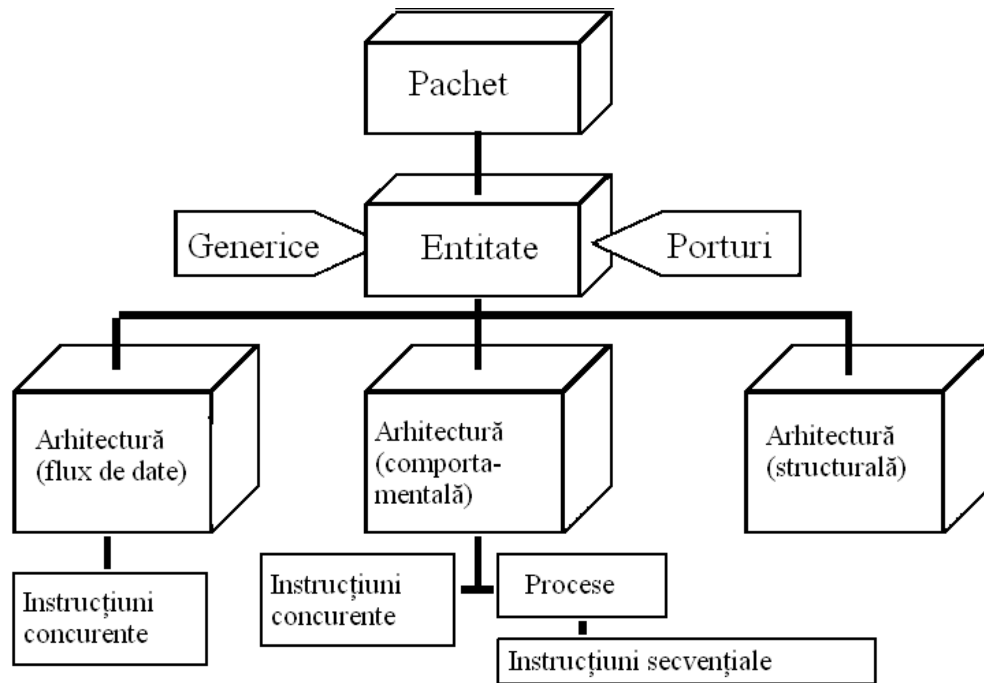
Caracteristicile de bază ale limbajului sunt:

- Descrierea func□ionării componentelor electronice de la nivelul de poartă logică până la procesoare □i sisteme de calcul.
- Descrierea ierarhică a sistemelor digitale. Modelele VHDL realizate pot fi utilizate ca blocuri în descriere unor circuite complexe;
- Descrierea structurală/comportamentală a sistemelor în scopul sintezei automate;
- Descrierea evenimentelor concurente, specifice func□ionării reale a circuitelor digitale;
- Verificarea funcțională (prin simulare) şi temporală a sistemelor prin intermediul unor programe speciale, numite ***test bench***. Ele con□in descrierea stimulilor □i a rezultatelor ce ar trebui ob□inute prin aplicarea acestora, în scopul depistării automate a unor erori func□ionale.

Particularită□ile limbajelor de descriere hardware □i , în particular al limbajului VHDL, sunt următoarele:

- Există unele deosebiri între un program VHDL pentru sinteza circuitului logic □i un program VHDL pentru simularea lui;
- Este posibil ca un program VHDL, perfect, din punct de vedere sintactic, să fie neimplementabil datorită descrierii eronate a unor fenomene fizice din circuit;
- În anumite situa□ii este necesară interven□ia proiectantului pentru a înlătura anumite anomalii din solu□ia generată automat.

Principalele părți (blocuri) care alcătuiesc un model VHDL complet sunt:



Arhitectura poate fi de trei tipuri:

1. Structurală
2. Comportamentală
3. Flux de data

La nivel ***structural*** sistemul este descris ca o colecţie de porţi şi conexiuni între ele. Această descriere se apropie de realizarea fizică a sistemului.

Modelarea structurală impune o proiectare ierarhizată în care se pot defini componente folosite de mai multe ori. Aceasta reduce semnificativ complexitatea proiectelor mari.

La nivel ***comportamental*** (***funcţional)*** sistemul este descris prin modul cum se comportă şi nu prin componentele sale. Această descriere foloseşte atât instrucţiuni secvenţiale care se execută în ordinea specificată, cât şi instrucţiuni concurente care se execută în paralel.

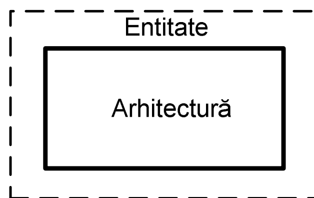
Reprezentarea de tip **Data Flow** descrie modul cum circulă datele prin sistem la nivel de transfer de date între registre (RTL). Această descriere foloseşte instrucţiuni concurente, care se execută în paralel.

Structura unui cod VHDL

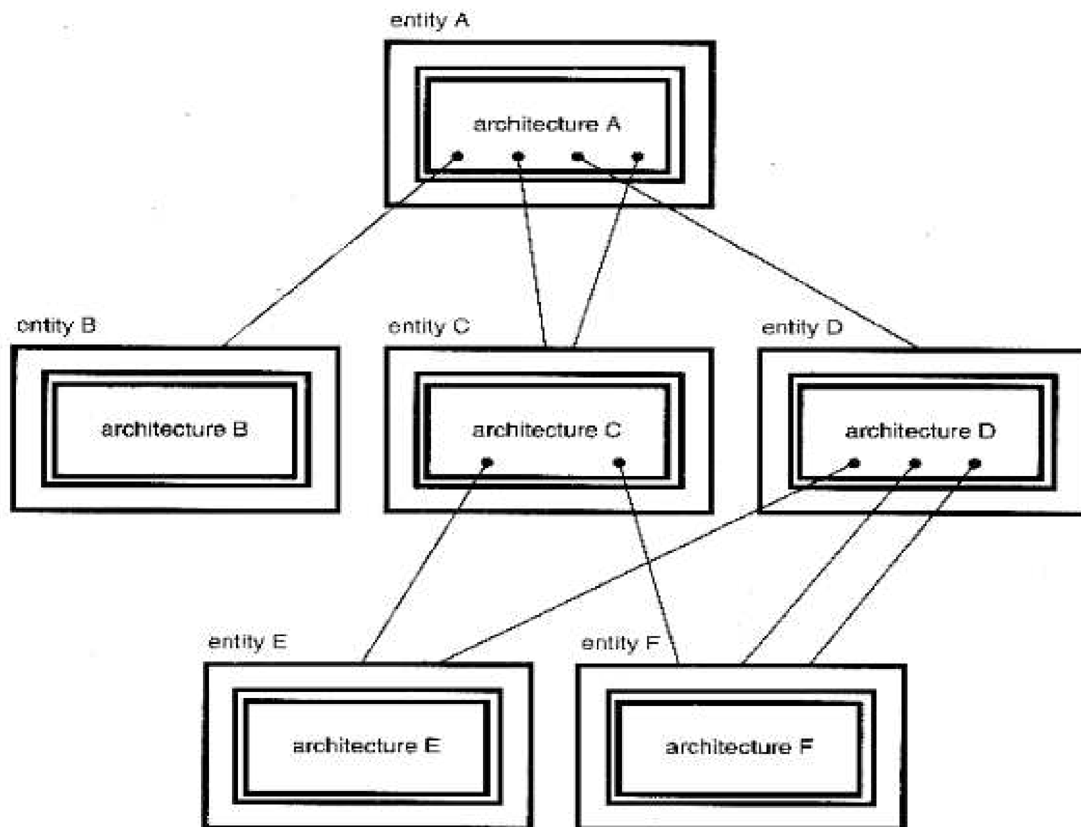
VHDL a fost proiectat ca şi program structurat, împrumutându-se unele idei de la limbajele de programare soft Pascal şi Ada.

O idee de bază este acea de a definii o interfaţă a modulului hard în timp ce detaliile interne sunt ascunse.

Astfel o entitate (*entity*) *VHDL* este o simplă declarare a intrărilor şi ieşirilor modulului în timp ce arhitectura (*architecture*) *VHDL* este o descriere structurală sau comportamentală detailată a funcţionării modulului.

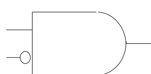


Acest concept formează bazele proiectării ierarhice a sistemelor, şi anume arhitectura entității de la nivelul cel mai superior poate conține (*instantiate*) alte entități ale căror arhitecturi sunt “invizibile” de la nivelele superioare. O arhitectură de la nivel superior poate folosii entități de la nivelul inferior de mai multe ori, iar mai multe arhitecturi de la nivel superior pot folosii aceeaşi entitate de la nivel inferior fiecare la rândul ei.



Exemplu de program VHDL:

Inhibiție (BUT/NOT)



```
entity BUT_NOT  is   -- poarta SI cu o intrare inversata
        Port (X,Y:   in   BIT;
                Z:       out   BIT) ;
 End BUT_NOT;

Architecture BUT_NOT_arch  of  BUT_NOT  is
        Begin
                Z <='1' when X='1'  and  Y='0' else  '0';
 End BUT_NOT_arch;
```

Aici, entity, port, end, is, in, out, architecture, begin, when, else – sunt *cuvinte rezervate* sau *cuvinte cheie*. Acestea nu pot fi folosite de utilizator ca nume de semnale sau identificatori.

Identificatorii sunt definiți de utilizator. În exemplul de mai sus aceşti identificatori sunt: BUT_NOT, X, Y, Z □i BIT.

BIT este un identificator preexistent şi poate fi redefinit.

Identificatorii pot conține numai simboluri alfanumerice (A-Z, a-z, 0-9) şi caracterul „underscore” (_). caracterul „underscore” nu poate apărea dublat sau în ultima poziție. Cuvintele cheie şi identificatorii pot fi scrişi atât cu majuscule cât şi cu litere mici, nu sunt “*case sensitive*”.

Cuvintele cheie definite în VHDL:

abs	acces	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	proctected
pure	range	record	register
reject	rem	report	return
rol	ror	select	severity
signal	shared	sla	sll
sra	srl	subtype	then
to	transport	type	unaffected
units	until	use	variable
wait	when	while	with
xnor	xor		

Sintaxa pentru declararea entității:

entity nume_entitate **is**
    port (nume-semnale : ***mod*** tip-semnale;
        nume-semnale : ***mod*** tip-semnale;
        ...
        nume-semnale : ***mod*** tip-semnale) ;
end nume-entitate;

mod = unul din următoarele patru cuvinte rezervate, specificând direcția semnalului:

- ***in*** – pentru semnal de intrare în entitate;
- ***out*** – pentru semnal de ieşire din entitate, valoarea semnalului nu poate fi "citită" înăuntrul entității, ci numai de alte entități care-l folosesc;
- ***buffer*** – defineşte un semnal de ieşire din entitate, iar valoare lui poate fi citită şi în interiorul arhitecturii din entitate;
- ***inout*** – defineşte un semnal de intrare/ieşire din entitate, acesta se foloseşte frecvent pentru a descrie pini three-state.

Sintaxa pentru declararea arhitecturii:

```
arhitecture  nume_arhitectură  of  nume_entitate  is
     Zona de declarații (tipuri, semnale, constante,
                          funcții, proceduri, componente)
       begin
               instrucțiuni_concurente
       end  nume_arhitectură;
```

Numele entității (*entity-name*) trebuie să fie acelaşi cu cel folosit la definirea entității. Numele arhitecturii (*architecture-name*) este un identificator definit de utilizator, dacă se doreşte poate fi la fel cu cel al entității sau diferit.

Reprezentări numerice

Reprezentarea numerică obişnuită este reprezentarea zecimală.
VHDL permite reprezentarea de tip întreg şi real.
Tipul întreg : 13, 25, 45E6
Tipul real : 1.2 , 3.14E-2

Pentru reprezentarea unui număr în altă bază se foloseşte notarea:
baza#număr#

Exemple: 2#101101# reprezentare binară
8#356# reprezentare octală
16#1C# reprezentare hexazecimală

Pentru a face citirea numerelor mai uşoară se admit caractere underscore: 2#1010_1100_1110#

Caractere, şiruri de caractere şi şiruri de biți

Pentru a putea folosi caractere se folosesc ghilimele simple:
'a', 'A', '.'
Şirurile de caractere se plasează între ghilimele duble:

”caracter”

Un şir de biți reprezintă o secvențş de valori 0 şi 1. Şiruri de biți pot fi reprezentate în sistemele binar, octal şi hexazecimal (mai compact):

Exemple:

Binar B”1001_1011”
Octal O”324”
Hexazecimal X”3D4A”

Obiecte de tip date: semnale, variabile şi constante

Un obiect de tip dată este creat de o declarație a obiectului şi are asociată o valoare şi un tip.

Constante

O constantă poate avea o singură valoare de un tip specificat şi nu se poate modifica pe parcursul modelării.

Sintaxa declarării constantei:
constant *nume-constantă* **:** *nume-tip* **: =** *valoare;*
Constantele se declară la începutul unei arhitecturi şi se pot folosi apoi oriunde în arhitectură. Constantele declarate într-un proces se pot folosi doar în procesul respectiv.

Exemple:

constant X: integer := 24;
constant BUS_SIZE: integer :=32; -- reprezintă lățimea componentei
constant Y: integer : = BUS_SIZE-1; -- numărul de biți ai lui Y
constant Z: character := ‘Z’; -- sinonim cu valoarea de înaltă impedanță
constant T: time := 2 ns;
Constantele pot fi definite în pachet, entitate, arhitectură i proces.

Variabile

O variabilă poate avea o singură valoare la un moment dat, dar ea poate fi şi actualizată folosind o instrucțiune de actualizare. Variabila se actualizează fără nici o întârziere, imediat ce instrucțiunea este executată. Variabilele se declară doar în interiorul proceselor.

Sintaxa declarării variabilelor:

variable *nume-variabile : tip-variabile [:= valori inițiale];* --[…] opțional

Exemple:

variable CT : bit :=0 ;
variable VAR : boolean := FALSE;
variable SUM : integer **range** 0 **to** 256 := 16 ; -- 16 este valoarea inițială
variable X_BIT : bit_vector (7 **downto** 0) ; -- defineşte un vector de 8 biți
Pentru actualizarea variabilei se poate folosi o instrucțiune de asignare:
nume_variabilă := expresie;

Semnale

Semnalele reprezintă porturile de intrare/ieşire şi conexiunile interne ale unui circuit. Semnalele se definesc în cadrul entității.

entity nume_entitate **is**
  port (nume-semnale : ***mode*** tip-semnale;
    nume-semnale : ***mode*** tip-semnale;
    ...
    nume-semnale : ***mode*** tip-semnale) ;
end nume-entitate;

De asemenea se pot definii şi semnale interne arhitecturii, dar ele vor acționa doar local.

Sintaxa pentru definirea unui semnal intern:
signal *nume_semnal: tip_semnal [:=valori inițiale]*; -- nu este specificat modul

Exemple:

signal SUMA, CARRY : std_logic;
signal CLK : bit ;
signal DATA_BUS : bit_vector (0 to 7) ;
signal VAL : integer range 0 to 100 ;

Semnalele sunt actualizate când se execută o instrucțiune de asignare, cu o anumită întârziere:

SUMA <= (A xor B) **after** 2 ns ;

În particular se poate astfel specifica şi o formă de undă:

signal unda : std_logic ;
unda <= ‘0’, ‘1’ **after** 5 ns, ‘0’ **after** 10 ns, ‘1’ **after** 20 ns ;

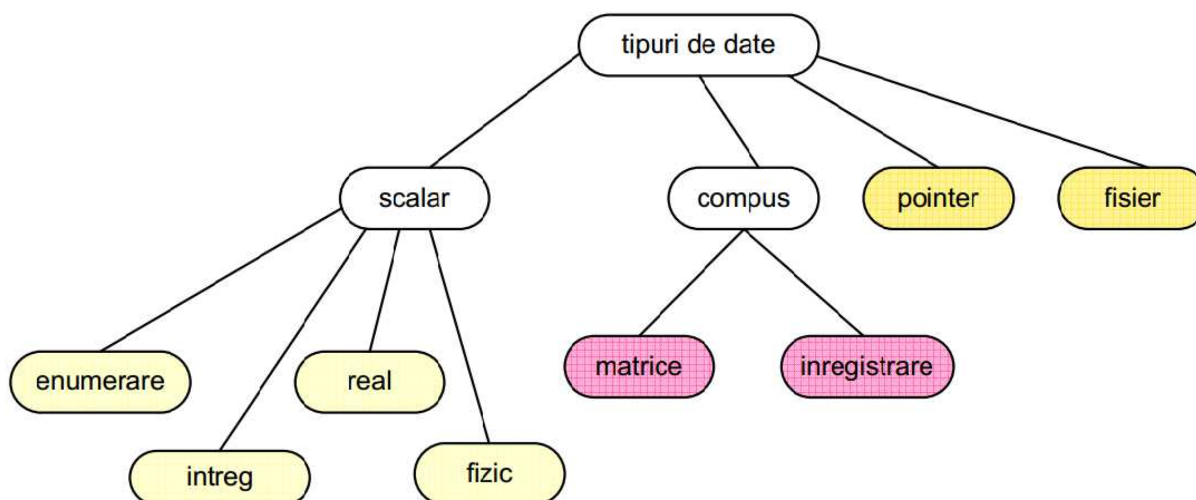
Tipuri de date

In VHDL există doua feluri de tipuri: tipuri SCALARE si tipuri COMPUSE.

Tipurile scalare includ numere, cantitati fizice, enumerari si tipuri predefinite.

Tipurile compuse sint matrice □i înregistrări.

În VHDL sânt definite □i tipurile 'access' (pointeri) si 'file' (fisiere).



Toate semnalele, variabilele şi constantele dintr-un program *VHDL* trebuie să aibă asociat un tip. În cadrul tipului se specifică un set de valori pe care le poate lua obiectul (semnal, variabilă …) şi de asemenea există şi un set de operatori (+, AND etc) asociați tipului respectiv.

Tipurile de date predefinite din limbajul VHDL sunt descrise în pachetul (eng. package) ***standard*** din biblioteca ***std .***

Tipurile predefinite din limbajul VHDL

Tip	Domeniul de valori	Exemple
Bit	‘0’, ‘1’	signal B: bit :=1;
Bit_vector	un □ir de elemente de tip bit	signal BUS: bit_vector (7 downto 0);
Boolean	TRUE, FALSE	variable X: boolean :=true;
Character	orice caracter permis în VHDL	variable Z: character :=’a’ ;
Integer	Domeniul include numerele întregi de la -(2^{31}-1) până la +(2^{31}-1).	constant A: integer :=12;
Natural	De la 0 pâna la valoarea maximă admisă	variable C: natural :=4;
Positive	De la 1 pâna la valoarea maximă admisă	variable D: positive :=10;
String	Un □ir de elemente de tip caracter	variable list: string :=”abcde&*%”;

Tipul *character* (caracter) include toate caracterele din setul de caractere ISO* (*International Organization of Standardization*) exprimate pe 8 biți, primele 128 fiind caractere ASCII.

Tipul fizic:

Tipul fizic este un tip numeric de reprezentare a unor cantităţi fizice (lungime, timp, volţi). Declaraţia de tip include specificarea unei unităţi de măsură de bază şi eventual un număr de unităţi de măsură secundare, care reprezintă multiplii ai unităţii de bază.

Exemplu:

type length **is range** 0 **to** 1E9
units
um;
mm = 1000 um;
cm = 10 mm;
m = 1000 mm;
end units;

Există tipul predefinit 'time', folosit in simulări VHDL pentru specificarea întârzierilor.

type **time** is range **interval_maxim_din_implementare**
units
fs;
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;

Înregistrări: Înregistrările în VHDL sânt colecţii de elemente, care pot avea tipuri diferite.

Exemplu:

type instruction **is**
record
op_code : processor_op;
address mode : mode;
operand1,operand2 : integer range 0 to 15;
end record;

Cele mai folosite tipuri în programele *VHDL* sunt aşa numitele tipuri definite de utilizator (*user defined types*), unul dintre acestea este ***tipul enumerare*** (*enumerated type*) definit printr-o înşiruire de valori.

Exemple de declarare a tipului enumerare:

Type nume-tip **is** (listă de valori);

Lista de valori reprezintă enumerarea tuturor elementelor posibile pentru respectivul tip, separate prin virgulă. Valorile enumerate pot fi atât caractere (un caracter este cuprins între ghilimele simple) cât şi identificatori definiți de utilizator.

Exemple.
type oct_digit **is** (`0`, `1`, `2`,`3`,`4`,`5`,`6`,`7`) ;
type PC_OPER **is** (load, store, add, sub) ;
type MY_WORD **is range** 31 **downto** 0 ;
type cmos_level **is range** 0.0 to 3.3 ;

Exemple de obiecte care folosesc aceste tipuri:
variable ALU_OP : PC_OPER ;
signal SIG : oct_digit ;
Dacă nu se inițializează semnalul, inițializarea implicită este valoarea extremă in stânga domeniului.

Limbajul *VHDL* permite definirea de subtip (*subtype*) corespunzător unui tip. Valorile unui subtip trebuie să se afle într-un subdomeniu de valori ale tipului de bază, continuu față de valorile tipului de bază.

Exista o serie de tipuri de enumerări predefinite:

type severity-level is (note,warning,error,failure);
type boolean is (false,true);
type bit is ('0','1');
type character **is** (NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FSP, GSP, RSP, USP, ` `, `!`, … `z`, `{`, `|`, `}` , `~`, DEL);

Subtype nume-subtip **is** nume-tip valoare-inițială **to** valoare-finală; -- ordine crescătoare.

Subtype nume-subtip **is** nume-tip valoare-inițială **downto** valoare-finală; -- ordine descrescătoare.

Exemple:
Subtype twoval_logic **is** std_logic **range** `0` **to** `1` ;
Subtype negint **is** integer **range** `-2147483647` **to** `-1` ;
Subtype data_word **is** my_word **range** 7 **downto** 0 ;

VHDL include două subtipuri integer predefinite:
Natural : subtype natural is integer range 0 to *cel mai mare întreg*;
Positive: subtype positive is integer range 1 to **cel mai mare întreg**;

Un tip foarte important este ***std_logic*,** care este un tip standard definit de utilizator şi este parte a package-ului standard IEEE 1164. Acest pachet include tipul de date std_ulogic care are 9 valori logice utile în simularea unui semnal logic dintr-un circuit real.

`U` - neinitializat
`X` - puternic necunoscut
`0` - puternic 0 -- poate fi sintetizat
`1` - puternic 1 -- poate fi sintetizat
`Z` - impendanta inalta -- poate fi sintetizat
`W` - slab necunoscut
`L` - slab 0
`H` - slab 1
`-` indiferent.

O altă categorie foarte importantă de tip definit de utilizator este tipul matrice (***array type***). Acest tip defineşte o matrice ca fiind un set de elemente de acelaşi tip în care fiecare element este selectat printr-un indice de matrice (*array index*)

Versiuni de sintaxă folosite în declararea unei matrici:
Type nume-tip **is array** (valoare-initiala **to** valoare-finala) **of** tip-element;
Type nume-tip **is array** (valoare-initiala **downto** valoare-finala) **of** tip-element;

Exemple:
Type luni_count **is array** (1 **to** 12) **of** integer;
Type byte **is array** (7 **downto** 0) **of** std_logic;

Constant WORD_LEN: integer :=32
type word **is array** (WORD_LEN-1 **downto** 0) **of** word;
type matrice3x2 **is array** (1 to 3, 1 to 2) **of** natural ; -- matrice bidimensională

Se consideră implicit că elementele unei matrici sunt ordonate de la stânga la dreapta, astfel cel mai din stânga element al matricilor luni_count, byte, word este 1, 7, 31.

Uneori este util să nu se specifice domeniul.
type nume **is array** (tip **range <>) of** tip_elemente ;

Accesarea unui singur element din matrice:

W(5) , unde W este semnal aferent matricii word.

O matrice literală se defineşte (*array literals*) prin înşiruirea între paranteze a valorilor elementelor. Elementele variabilei B de tipul byte pot primi toate valoarea *1* logic scriind o expresie de forma:

B := (‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’);

Limbajul *VHDL* permite de asemenea notații mai scurte, de exemplu pentru a atribui valoarea *1* logic tuturor biților variabilei *W* de tip *word*, în afară de LSB din fiecare se va scrie expresia:

W := (0=>’0’, 8=>’0’, 16=>’0’, 24=>’0’, others=>’1’);

Expresiile anterioare pot fi rescrise folosind şirurile de caractere, după cum urmează:

B := “11111111”;

W:= “11111110111111101111111011111110”;

Este posibil de asemenea să se facă referire la un subset de valori (*slice*) dintr-o matrice, specificând începutul şi sfârşitul indicilor elementelor din subset, exemple: M(6 to 9), B(3 downto 0), W(15 downto 8) etc.

Cel mai important tip de matrice des întâlnit în programele *VHDL* este cel aparținând standardului logic definit de utilizator IEEE 1164 (*std_logic_vector*), definiția acestui standard este:

type STD_LOGIC_VECTOR **is array** (natural range < >) **of** STD_LOGIC

Datorită faptului că *VHDL*-ul este un limbaj puternic tipizat, adesea apare necesitatea convertiri unui semnal sau variabile dintr-un anumit tip în altul. Packageul IEEE 1164 conține câteva astfel de funcții de conversie, de exemplu din *BIT* în *STD_LOGIC* sau invers. O conversie foarte utilizată şi care nu este definită, din cauză că proiecte diferite pot avea nevoie de o interpretare diferită a numerelor (ex. numere cu semn sau fără semn), este conversia din *STD_LOGIC_VECTOR* în *integer*.

Conversii suportate de package-ul std_logic_1164	
Conversia	**Funcția**
std_ulogic -> bit	to_bit(*expresie*)
std_logic_vector -> bit_vector	to_bitvector(*expresie*)
std_ulogic_vector -> bit_vector	to_bitvector(*expresie*)
bit -> std_ulogic	To_StdULogic(*expresie)*
bit_vector -> std_logic_vector	To_StdLogicVector(*expresie*)
bit_vector -> std_ulogic_vector	To_StdUlogicVector(*expresie*)
std_ulogic -> std_logic_vector	To_StdLogicVector(*expresie*)
std_logic -> std_ulogic_vector	To_StdUlogicVector(*expresie*)

Atribute

În VHDL există două categorii de atribute:

1. Predefinite ca parte a standardului 1076;

2. Introduse de utilizator.

Atributele predefinite sunt întotdeauna aplicate ca un prefix numelui unui semnal, variabile sau tip.

Atributele sunt folosite pentru e returna diferite tipuri de informaţie.

Exemple de atribute predefinite:

nume_semnal`event - întoarce valoarea booleană True dacă semnalul a avut o tranziţie şi False dacă nu;

nume_semnal`active - întoarce valoarea booleană True dacă a existat o atribuire a valorii semnalului şi False dacă nu.

Operatori VHDL

Limbajul VHDL suportă diferite clase de operatori care operează pe semnale, constante şi variabile.

Clasa						
1. Operatori logici	and	or	nand	nor	xor	xnor
2. Operatori relaţionali	=	/=	<	<=	>	>=
3. Operatori de deplasare	sll	srl	Sla	sra	rol	ror
4. Operatori aditivi	+	–	&			
5. Operatori unari	+	-				
6. Operatori multiplicativi	*	/	mod	rem		
7. Alţi operatori	**	abs	Not			

Prioritatea operatorilor este maximă pentru operatorii din clasa 7. Dacă nu sunt folosite paranteze, ei se execută primii. Operatorii din aceeaşi clasă au aceeaşi prioritate şi se execută de la dreapta la stînga într-o expresie.

Operatorii logici sunt definiţi pentru tipurile bit, boolean, std.logic şi pentru vectorii de aceste tipuri.

Operatorii nand şi nor nu sunt asociativi. De exemplu expresia: **X nand Y nand Z** va da o eroare de sintaxă. Corect se va scri: **(X nand Y) nand Z.**

Operatorii relaţionali testează relaţia dintre două tipuri scalare şi oferă ca rezultat o ieşire booleană TRUE sau FALSE. De menţionat ca simbolul operatorului „mai mic sau egal" este acelaşi cu cel al operatorului de atribuire pentru semnale şi variabile.

Operatorii de deplasare execută o deplasare sau rotire la nivel de bit într-un vector de elemente de tip bit (std.logic) sau boolean.

Operatorii aditivi sunt folosiţi pentru operaţii aritmetice pe orice tip numeric de operanzi. Operatorul de concatenare este folosit pentru a concatena doi vectori, de exemplu prin concatenarea '0' & '1' & '1Z' se obţine "011Z". Pentru a folosi aceşti operatori trebuie specificate pachetele std_logic_unsigned sau std_logic_arith pe lângă pachetul std_logic_1164.

Operatorii unari se folosesc pentru a specifica semnul unui tip numeric.

Operatorii de multiplicare se folosesc pentru a executa funcţii matematice pe tipurile numerice (întreg sau virgulă mobilă).

mod – împărțire modulo;
rem - împărțire modulo cu rest;
** - ridicare la putere (pentru tipurile numerice);
abs - valoare absolută (pentru tipurile numerice);
not – negare (pentru tipurile bit şi boolean);

Conversiile dintre std_logic_vector şi tipurile numerice:

Data type of a	To data type	Conversion function/type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Sa presupunem ca sunt declarate următoarele semnale:
s i g n a l sl, s 2 , s3: std_logic_vector (3 downto *0) ;*
s i g n a l ul, u2, u3: unsigned (3 downto 0) ;

Operațiile:
u l <= s l ;
s2 <= u 3 ;
nu sunt corecte, deoarece au loc atribuiri între diferite tipuri.
Corect se va scrie:
u l <= unsigned(s1);
s2 <= std_logic_vector(u3);
Operații aritmetice:
u4 <= u2 + ul; -- corect
s5 <= s2 + sl; -- incorect, std_logic_vector nu susține operatori aritmetici
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ok

Exemple cu operatorul ***concatenare***
signal a1: std_logic;
signal a4: std_logic_vector (3 downto 0);
signal b8, c8: std_logic_vector (7 downto 0);
...
b8 <= a4 & a 4 ;
c8 <= a1 & a1 & a4 & " 0 0 " ;

Operatorul & poate fi utilizat şi pentru a deplasa datele. Cu toate că operatorii de deplasare sunt definiți în limbajul VHDL, ei uneori nu pot fi sintetizați automat de compilator.

signal a : std_logic_vector (7 downto 0);

```
signal rot, shl, sha : std_logic_vector (7 downto 0);
...
rot <= a (2 downto 0) & a (7 downto 3); -- deplasare ciclica cu 3 biti la dreapta
shl <= "000"& a(7 downto 3);  -- deplasare logica cu 3 biti la dreapta
sha <= a ( 7 ) & a ( 7 ) & a ( 7 ) & a ( 7 downto 3 ) ; -- deplasare aritmetica cu 3
-- biti la dreapta
```

Biblioteci şi package-uri

O bibliotecă (*library*) *VHDL* este locul unde compilatorul depune toate informațiile despre un anumit proiect şi anume toate fişierele intermediare folosite în analiza, simularea şi sinteza proiectului.

Pentru un proiect *VHDL* dat compilatorul creează şi foloseşte în mod automat o bibliotecă numită *work*.

Pentru a utiliza biblioteca standard care conține elemente (funcții, tipuri, etc) predefinite se foloseşte directiva *library* la începutul codului *VHDL*, ex.

library ieee;

Astfel se obține accesul la toate entitățile şi arhitecturile aflate în biblioteca respectivă.

Pentru a accesa tipurile definite se folosesc pachetele (*package*-urile).

Un *package* este un fişier care conține obiecte definite folosite în mod curent. Tipurile de obiecte care se află într-un *package* sunt: semnale, definiri de tipuri, constante, funcții, proceduri şi declarări de componente.

Un proiect poate folosi un *package* dacă în codul *VHDL* se include directiva *use,* de exemplu pentru a apela toate componentele definite în package-ul standard IEEE 1164 scriem următoarea secvență de cod:

use ieee.std_logic_1164.all;,

unde *ieee* este numele unei librării care a fost dat împreună cu directiva *library.* Sintaxa *std_logic_1164* este numele unui fişier care conține componentele definite, iar sufixul *all* îi spune compilatorului să le folosească pe toate.

În locul respectivului sufix se poate scrie numai numele unei anumite componente care să fie luată în considerare de compilator.

Exemple de pachete:

std_logic_1164 - defineşte tipurile de date standard;

std_logic_arith - conține funcții aritmetice, de conversie şi comparație pentru tipurile de date signed, unsigned, integer, std_logic şi std_logic_vector;

std_logic_unsigned;

std_logic_misc - defineşte tipuri suplimentare, subtipuri, constante şi funcții pentru pachetul std_logic_1164.

Definirea package-urilor nu se limitează doar la cele standard aflate în biblioteci, utilizatorul poate el însuşi să-şi definească package-uri.

Sintaxa de declarare a unui pachet este următoarea:

package numele_pachetului **is**

declarațiile pachetului
end package numele_pachetului ;
package body numele_pachetului **is**
declarațiile corpului pachetului
end package body numele_pachetului;

De exemplu, în proiectarea structurală, funcțiile de bază pentru componentele AND2, OR2, NAND2, NOR2, XOR2 etc., sunt necesare ca să fie definite înainte să fie utilizată una din ele. Acest lucru poate fi făcut într-unul dintre pachete, de ex. **base_func** pentru fiecare dintre aceste componente, după cum urmează :

```
-- declarația pachetului
library ieee ;
use ieee.std_logic_1164.all ;
package  basic_func is

-- declarația pentru AND2
        component  AND2
                generic (DELAY: timp  :=5ns);
                port (in1, in2: in std_logic; out1:  out  std_logic);
        end component;

--declarația pentru OR2
        component   OR2
                generic (DELAY: timp  :=5ns);
                port (in1, in2: in std_logic; out1:  out  std_logic);
        end component;

end package  basic_func ;

-- declarațiile corpului pachetului
library ieee ;
use ieee.std_logic_1164.all;
package body basic_func  is

        -- poartă AND cu două intrări
        entity AND2   is
                generic (DELAY:  timp);
                port (in1, in2:  in std_logic;  out1:  out  std_logic);
        end AND2;
        arhitecture  model_conc  of AND2  is
                begin
                        out1 <= in1  and  in2  after  DELAY;
        end model_conc;
```

```
-- poartă OR  cu două intrări
entity OR2   is
        generic (DELAY:  timp);
        port (in1, in2:  in std_logic; out1:  out  std_logic);
end  OR2;
arhitecture  model_conc2  of AND2  is
        begin
                out1 <= in1  or  in2  after  DELAY;
end model_conc2;

end package body basic_func;
```

În program a fost inclusă o întârziere (delay) de 5 ns. Totuşi, ar trebui de notat că specificațiile întârzierii sunt ignorate de către Foundation synthesis tool. A trebuit să utilizăm tipul predefinit std_logic care este declarat în pachetul std_logic_1164. S-au inclus în acest pachet **library** şi clauza **use**. Acest pachet are nevoie să fie compilat şi plasat într-o librărie. O să numim aceasta ***library my_func***. Pentru a utiliza componentele acestui pachet, o dată ce a fost declarat, se utilizează *library* şi clauza *use* :

```
library  ieee, my_func;
use  ieee.std_logic_1164.all, my_func.basic_func.all;
```

Funcții şi proceduri

La fel ca şi în alte limbaje de nivel înalt şi în *VHDL* o funcție acceptă un anumit număr de argumente şi returnează un rezultat. Atât argumentele cât şi rezultatul returnat de o funcție sunt de un tip predeterminat.

Sintaxa *VHDL* prin care se defineşte o funcție:

```
        function nume-funcție (
                nume-semnal : tip-semnale;
                ...
                nume-semnal : tip-semnale;
        ) return tip-întoarcere is
        declarații de tip
        declarații de constante
        declarații de variabile
        definiții de funcții
        definiții de proceduri
    begin
        instrucțiune secvențială
```

```
        ...
        Instrucțiune secvențială
    end nume funcție;
```

După ce se dă un nume funcției (*function -name*) se poate definii o listă de parametri formali care vor fi folosiți în structura funcției, toți aceşti parametrii trebuie să fie de acelaşi tip. În cadrul unei funcții se pot definii local tipuri, constante, variabile şi de asemenea funcții şi proceduri imbricate (*nested*). Între cuvintele cheie *begin* şi *end* este cuprinsă o serie de instrucțiuni secvențiale care sunt executate ori de câte ori este apelată funcția . În cadrul funcției cuvântul cheie *return* indică momentul din care funcția returnează o valoare care trebuie să fie de tipul celei definite la declararea funcției.

Exemplu:

```
entity BUT_NOT  is   -- poarta SI cu o intrare inversata
        Port (X1,X2:   in   BIT;
              Z:       out  BIT) ;
End BUT_NOT;
Architecture BUT_NOT_archf  of  BUT_NOT  is
        function ButNot (A, B: bit)  return bit is
          begin
             if B = `0` then return A;
             else return `0
             end if;
          end ButNot;
Begin
        Z <= ButNot (X1,X2);
End BUT_NOT_archf;
```

În limbajul *VHDL* mai este definită noțiunea de procedură (*procedure*) care este similară cu funcția doar că nu returnează o valoare. La fel cum o funcție poate fi apelată în locul unei expresii şi o procedură poate fi apelată în locul unei declarații. Dacă argumentele unei proceduri sunt declarate de tipul *out* sau *inout*, va exista totuşi o valoare returnată.

Stiluri de proiectare şi descriere a circuitelor

O arhitectură este formată dintr-o serie de instrucțiuni concurente. Fiecare instrucțiune se execută simultan cu celelalte instrucțiuni din arhitectură. Instrucțiunile concurente sunt necesare pentru a putea simula comportamentul circuitelor modelate, în care componentele interacționează unele cu altele.

Într-o arhitectură *VHDL* dacă ultima instrucțiune modifică starea unui semnal care este folosit de prima instrucțiune din arhitectură atunci simulatorul se va întoarce la prima instrucțiune şi o va reactualiza cu noua valoare, cu alte cuvinte simulatorul va continua să propage schimbările şi să reactualizeze valorile semnalelor până când circuitul simulat se stabilizează.

Sintaxa *VHDL* conține câteva instrucțiuni concurente şi de asemenea un mecanism de grupare a instrucțiunilor secvențiale astfel încât ele să funcționeze ca şi o sigură instrucțiune concurentă. Utilizate în diferite modalități, aceste instrucțiuni generează trei stiluri diferite de proiectare şi descriere a circuitelor:

- proiectare structurală;
- proiectare ca flux de date;
- proiectare comportamentală.

Proiectare structurală

În proiectarea structurală se definesc exact elementele şi interconexiunile dintre elementele circuitului. O descriere structurală pură este echivalentă cu o descriere schematică sau cu un netlist (listă de conexiuni între elementele unui circuit).

Cea mai elementară instrucțiune concurentă, folosită în descrierea structurală este instrucțiunea *component:*

etichetă: nume-componentă **port-map** (semnal1, semnal2, ..., semnal n);
etichetă: nume-componentă **port-map** (port1=> semnal1, port2=>semnal2, …portn=>semnaln);

Numele componentei este numele unei entități definite anterior care trebuie utilizată sau instanțiată (multiplicată).

Fiecare instrucțiune *component* care invocă numele entității creează o "clonă" (instanță) respectivei entități care se va distinge printr-un nume unic dat de o etichetă (*label*).

Cuvântul cheie *port map* introduce o listă prin care porturile entității sunt asociate unor semnale din arhitectura curentă.

Această listă se poate scrie în două moduri: pozițional şi asociativ.

În modul pozițional de atribuire semnalele din listă sunt asociate cu porturile entității în aceeaşi ordine în care apar în entitate.

În modul de atribuire asociativ fiecare port al entității se asociază unui semnal folosind operatorul "=>", iar asocierea poate fi făcută în orice ordine.

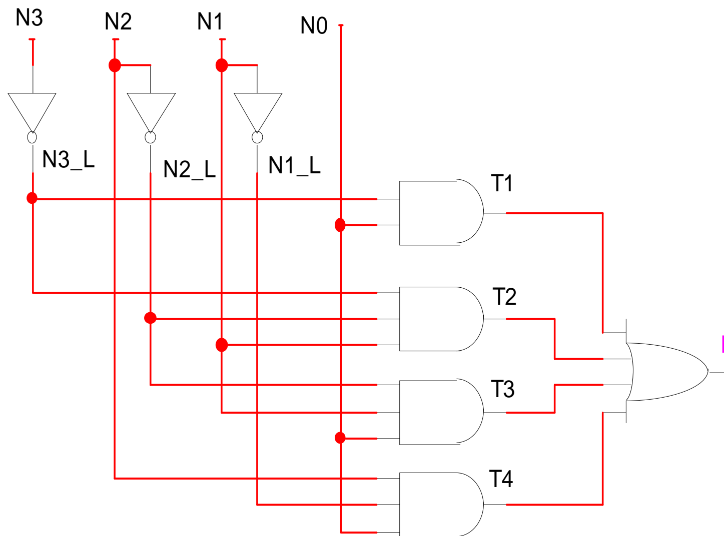
Înainte de a fi instanțiată într-o arhitectură, o componentă trebuie să fie declarată în cadrul arhitecturii, sintaxa de declarare este:

component nume-componentă
port (nume-semnale: mod tip-semnale;
nume-semnale: mod tip-semnale;
...
nume-semnale: mod tip-semnale);
end component;

Componentele folosite în arhitectură pot să fie de două tipuri:
a) componente descrise anterior care sunt specifice proiectului,
b) componente care sunt apelate din bibliotecile standard.

Exemplu. Codul VHDL pentru un detector de numere prime pe 4 biți.

Numerele prime sunt: 1, 2, 3, 5, 7, 11, 13.
Funcția minimizată: $F = \bar{N}_3 N_0 + \bar{N}_3 \bar{N}_2 N_1 + \bar{N}_2 N_1 N_0 + N_2 \bar{N}_1 N_0$
Circuitul logic:



$$T_1 = \bar{N}_3 N_0$$
$$T_2 = \bar{N}_3 \bar{N}_2 N_1$$
$$T_3 = \bar{N}_2 N_1 N_0$$
$$T_4 = N_2 \bar{N}_1 N_0$$

Programul structural VHDL pentru detectorul de numere prime:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity prime is
        port ( N: in STD_LOGIC_VECTOR (3 downto 0);
                F: out STD_LOGIC ) ;
end prime;

architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L  :  STD_LOGIC;
signal T1, T2, T3, T4  :  STD_LOGIC;
component  G_INV  port  (x :  in   STD_LOGIC; Q:  out   STD_LOGIC) ;
end component ;
component  G_AND2  port  (x, y : in STD_LOGIC; Q:  out   STD_LOGIC) ;
end component ;
component  G_AND3  port  (x, y, z : in STD_LOGIC; Q:  out   STD_LOGIC) ;
end component ;
component G_OR4  port (x, y, z, w : in STD_LOGIC; Q:  out   STD_LOGIC ;
end component ;

begin
        U1 : G_INV  port map  (N(3) , N3_L) ;
        U2 : G_INV  port map  (N(2) , N2_L) ;
        U3 : G_INV  port map  (N(1) , N1_L) ;
        U4 : G_AND2  port map (N3_L,  N(0), T1) ;
        U5 : G_AND3  port map (N3_L, N2_L,  N(1), T2) ;
```

```
        U6 : G_AND3  port map (N2_L, N(1), N(0),   T3) ;
        U7 : G_AND3  port map (N(2), N1_L, N(0), T4) ;
        U8 : G_OR4  port map (T1, T2, T3, T4, F) ;
end prime1;
```

Prin declararea entităţii se declară intrările şi ieşirile circuitului. În cadrul arhitecurii sunt declarate toate semnalele care vor fi folosite, de asemenea şi numele componentelor **(G_INV, G_AND2, G_AND3, G_OR4)** care trebuie definite în acelaşi proiect în fişiere VHDL de nivel ierarhic mai jos.

Exemplu de descriere a porții logice AND cu două intrări:

```
library ieee;
use ieee.std_logic_1164.all;
entity G_And2 is
port (    x: IN std_logic;
          y: IN std_logic;
          Q: OUT std_logic);
end G_And2;
architecture G_And2_beh of G_And2 is
begin
  Q <= x and y;
end G_And2_beh;
```

Deoarece descrierea circuitului este concurentă în orice ordine am introduce componentele se va sintetiza acelaşi circuit, iar funcționarea va fi aceeaşi.

Există proiecte în care este necesară crearea mai multor copii a unui element în cadrul unei arhitecturi. De exemplu, un sumator pe *n* biți se poate crea din *n* sumatoare pe un bit.

Limbajul *VHDL* include o instrucțiune (*generate*) care permite crearea unei structuri repetitive folosind un fel de buclă, fără a fi necesar să instanțiem separat fiecare element.

Sintaxa unei bucle *for- generate* :

```
label: for identificator  in domeniu  generate
          instrucțiune concurentă
end generate;
```

Identificatorul este implicit declarat ca variabilă compatibilă cu domeniul (*range*). Instrucțiunile concurente se execută câte o dată pentru fiecare valoare posibilă a identificatorului inclusă în domeniu. Identificatorul trebuie folosit în cadrul instrucțiunii concurente.

Exemplu. Crearea unei porți inversoare pe 8-biți.

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;

entity inv8 is
        port ( X: in STD_LOGIC_VECTOR  (1  to  8);
                Y: out  STD_LOGIC_VECTOR  (1  to  8) );
end inv8;

architecture inv8_arch  of inv8  is
component G_INV  port (I: in  STD_LOGIC;
                              Q: out STD_LOGIC);
end component;
begin:
        g1: for b  in  1 to 8  generate
            U1: G_INV  port map (X(b) , Y(b)) ;
        end generate;
end inv8_arch;
```

Programul pentru inversorul pe un bit trebuie inclus în proiectul curent:

```
library ieee;
use ieee.std_logic_1164.all;
entity G_INV is
  port( I  : in std_logic;
       Q : out std_logic);
end G_INV;
architecture func of G_INV is
begin
  Q <= not I;
end func;
```

Valoarea constantelor trebuie să fie cunoscută în momentul compilării unui program VHDL. În unele aplicații este util să se proiecteze şi să se compileze o entitate împreună cu arhitectura corespunzătoare care să conțină unii parametrii (cum ar fi lățimea magistralei) care nu sunt specificați. Această facilitate este introdusă de constanta *generic*.

Genericele nu se modifică în timpul simulării. Datele conținute în genericele transmise entității sau componentei se pot utiliza pentru a modifica rezultatele simulării, dar rezultatele nu pot modifica genericele.

```
entity nume-entitate  is
        generic ( nume constante : tip-constante;
                    …
                    nume-constante:  tip-constante);
        port ( nume-semnale: mod tip-semnale;
                 …
                 nume-semnale: mod tip-semnale);
end nume-entitate;
```

Una sau mai multe constante generice pot fi definite într-o declarație de entitate înaintea declarației de porturi. O valoare îi va fi atribuită numai când entitatea va fi instanțiată în cadrul altei arhitecturi. În cadrul definirii componentelor (*component*), constantelor *generic* li se atribuie valori folosind instrucțiunea *generic map,* asemănătoare cu instrucțiunea *port map.*

Exemplu. Definirea unui inversor de magistrală cu o lățime impusă de utilizator.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity businv is
        generic ( WIDTH: positive);
        port ( X: in STD_LOGIC_VECTOR  (WIDTH -1  downto  0);
                Y: out  STD_LOGIC_VECTOR  (WIDTH -1  downto  0) );
end businv;

architecture businv_arch  of businv  is
component G_INV  port (I: in  STD_LOGIC;
                       Q: out STD_LOGIC);
end component;
begin
        g1: for b  in  WIDTH -1  downto  0 generate
            U1: G_INV  port map (X(b) , Y(b)) ;
        end generate;
end businv_arch;
```

În următorul exemplu sunt instanțiate mai multe exemplare ale acestui inversor, fiecare pentru o altă lățime.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity businv1 is
        port ( IN8: in STD_LOGIC_VECTOR  (7  downto  0);
                OUT8: out  STD_LOGIC_VECTOR  (7  downto  0);
                 IN16: in STD_LOGIC_VECTOR  (15  downto  0);
                OUT16: out  STD_LOGIC_VECTOR  (15  downto  0);
                IN32: in STD_LOGIC_VECTOR  (31  downto  0);
                OUT32: out  STD_LOGIC_VECTOR  (31  downto  0) );
end businv1;

architecture businv1_arch  of businv1  is
component businv
```

```
        generic ( WIDTH: positive);
        port ( X: in STD_LOGIC_VECTOR  (WIDTH -1  downto  0);
               Y: out  STD_LOGIC_VECTOR  (WIDTH -1  downto  0) );
end component;

begin:
       U1: businv  generic map (WIDTH=>8)  port map (IN8, OUT8);
       U2: businv  generic map (WIDTH=>16)  port map (IN16, OUT16);
       U3: businv  generic map (WIDTH=>32)  port map (IN32, OUT32);
end businv1_arch;
```

Proiectare de tip flux de date

În limbajul VHDL există un şir de instrucţiuni ***concurente*** care permit descrierea circuitului din punctul de vedere al fluxului de date şi al prelucrării acestuia în circuit.

1. Instrucţiunea de atribuire concurentă directă de semnal

(*signal-name<=expression*).

Tipul expresiei trebuie să fie compatibil cu cel al semnalului.

Pentru a exemplifica atribuirea concurentă de semnal arhitectura detectorului de numere prime este rescrisă pentru fluxul de date.

```
architecture   prime2_arch of prime is
        signal T1, T2, T3, T4  :  STD_LOGIC;
                begin
                        T1  <=  not N(3) and N(0) ;
                        T2  <= not N(3) and not  N(2) and N(1) ;
                        T3  <= not N(2) and N(1) and N(0) ;
                        T4  <= N(2) and not N(1) and N(0) ;
                        F    <= T1 or T2 or T3 or T4;
                end prime2_arch;
```

Spre deosebire de descrierea anterioară (descriere structurală) porţile şi conexiunile dintre ele nu mai apar explicit, ci se folosesc operatori *VHDL* definiţi implicit in biblioteca IEEE (*and, or, not*). Se observă că operatorul *not* are prioritatea cea mai mare, astfel că nu este necesar să se utilizeze paranteze pentru expresii de genul *"not N(2)"*.

2. Instrucţiunea de atribuire concurentă condiţională de semnal when - else.

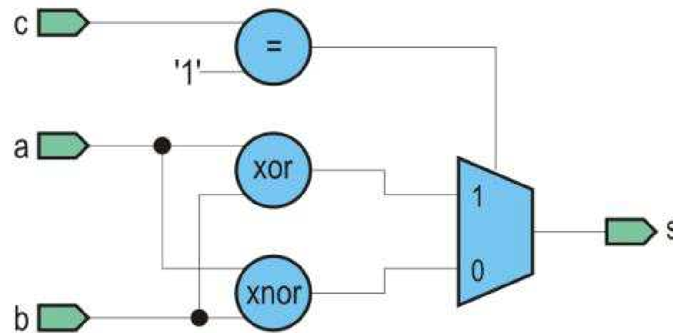
```
Nume-semnal  <= expresie  when espresie-booleană  else
                 . . .
                 expresie  when espresie-booleană  else
                 expresie;
```

In acest caz o expresie booleană combină termeni booleeni individuali folosind operatori impliciţi (*and, or, not*). Termenii booleeni sunt de obicei variabile sau rezultate ale unor comparaţii făcute cu ajutorul operatorilor relaţionali: = , /= (*inegaliate*), > , >= , < , <= (*mai mic sau egal*)

Instrucţiunea de atribuire condiţională este implementată printr-un multiplexor care selectează una din expresiile sursă.

Circuitul rezultat pentru următoarea instrucţiune:

```
s <= a xor b when c = '1' else
not (a xor b);
```



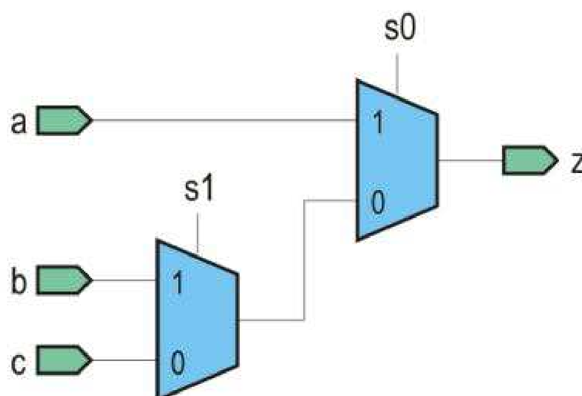
Circuitul este cel generat iniţial de utilitarul de sinteză, dar operatorul de egalitate va fi minimizat ulterior la o simplă conexiune, astfel încât semnalul c să controleze multiplexorul în mod direct.

Exemplul anterior reprezintă forma cea mai simplă a unei instrucţiuni de atribuire condiţională, în care există o singură condiţie testată. Un alt exemplu în care există două condiţii testate este următorul:

```
z <= a when s0 = '1' else
b when s1 = '1' else
c;
```

Condiţiile sunt evaluate în ordinea în care sunt scrise, fiind selectată pentru atribuire prima expresie a cărei condiţie este adevărată. Aceasta echivalează din punct de vedere hardware cu o serie de multiplexoare cu două căi, prima condiţie controlând multiplexorul cel mai apropiat de ieşire.

Circuitul rezultat pentru acest exemplu:



Pentru acest circuit, condiţiile au fost deja optimizate, astfel încât semnalele s0 şi s1 controlează multiplexoarele în mod direct. Din acest circuit se poate

observa că atunci când ***s0*** este '1', este selectat semnalul ***a*** indiferent de valoarea semnalului ***s1***. Dacă ***s0*** este '0', atunci ***s1*** selectează între intrările ***b*** şi ***c.***

Dacă există un număr mare de ramuri ale instrucţiunii, la implementare rezultă un şir lung de multiplexoare. De acest aspect trebuie să se ţină cont la proiectare: cu cât o expresie sursă apare mai târziu în lista de selecţie, cu atât semnalele din această expresie vor traversa mai multe multiplexoare în circuitul rezultat la sinteză.

Exemplu. Arhitectura detectorului de numere prime in care se foloseşte atribuirea concurentă condiţională de semnal.

```
architecture   prime3_arch of prime is
signal T1, T2, T3, T4  :  STD_LOGIC;
    begin
T1  <=  '1' when N(3)='0' and N(0)='1' else '0' ;
T2  <=  '1' when N(3)='0' and N(2)='0' and N(1)='1' else '0' ;
T3  <=  '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0' ;
T4  <=  '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0' ;
F   <= T1 or T2 or T3 or T4;
 end prime3_arch;
```

Comparaţia unui bit de tip *std_logic* cum ar fi *N(3)* se face in funcţie de caracterele literale *'1'* sau *'0'*, iar rezultatul returnat este de tip boolean.

Rezultatul acestor comparaţii se combină intr-o expresie booleană plasată intre cuvintele cheie *when, else.* Clauza *else* este obligatorie deoarece setul de condiţii dintr-o expresie trebuie să acopere toate combinaţiile posibile.

3. ***Instrucţiunea de atribuire concurentă selectivă de semnal:***

```
with  expresie  select
        nume-semnal <= valoare-semnal  when  opţiuni,
                       valoare-semnal  when  opţiuni,
                        . . .
                       valoare-semnal  when  opţiuni;
```

In acest tip de instrucţiune, se evaluează expresia dată, iar cand una dintre valori se potriveşte cu una dintre opţiuni (*choices*) atunci identificatorului *nume-semnal* i se va atribui valoarea corespunzătoare *valoare-semnal.* Opţiunea corespunzătoare fiecărei clauze *when* poate să fie o singură valoare sau o listă de valori separate intre ele prin bara verticală (|). Cuvantul cheie *others* poate fi folosit impreună cu ultima clauză *when* pentru a acoperii toate valorile pe care le poate lua expresia de evaluare.

Această instrucţiune este echivalentă funcţional cu instrucţiunea secvenţială ***case.***

Exemplu. Arhitectura detectorului de numere prime in care se foloseşte atribuirea concurentă selectivă de semnal. Toate opţiunile pentru care *F* ia valoarea

‘*1*’ ar fi putut fi scrise folosind o singură clauză *when,* dar pentru o ințelegere mai bună s-au folosit mai multe clauze.

```
architecture   prime4_arch of prime is
        begin
            with  N select
                F  <=  `1`  when "0001" |  "0010" | "0011",
                       `1`  when  | "0101"  | "0111" ,
                       `1`  when "1011"  | "1101"  ,
                       `0`  when others ;
        end prime4_arch;
```

Exemple.

1. Poarta SAU EXCLUSIV cu două intrări. În arhitectură se utilizează o instrucțiune de atribuire condițională.

```
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port (a, b: in std_logic;
        x: out std_logic);
end xor2;
architecture arh1_xor2 of xor2 is
  begin
     x <= '0' when a = b else
           '1';
  end arh1_xor2;
```

2. Definirea porții SAU EXCLUSIV cu două intrări, în cadrul arhitecturii se utilizează o instrucțiune de atribuire selectivă.

```
library ieee;
use ieee.std_logic_1164.all;
  entity xor2 is
     port (a, b: in std_logic;
           x: out std_logic);
  end xor2;
architecture arh_xor2 of xor2 is
    signal tmp: std_logic_vector (1 downto 0);
      begin
        tmp <= a & b;
        with tmp select
        x <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '0' when "11";
```

end arh_xor2;

Proiectare comportamentală (secvențială)

Nu întotdeauna este posibil să descriem direct comportarea unui circuit logic prin utilizarea unei instrucțiuni concurente. Pentru majoritatea descrierilor comportamentale este necesară folosirea unor elemente suplimentare de limbaj.

Instrucțiunile secvențiale se utilizează în procese, funcții şi proceduri.

1. Un element de descriere comportamentală cheie este **procesul**. Un proces (*process*) este o colecție de instrucțiuni secvențiale care se execută in paralel cu alte instrucțiuni concurente şi cu alte procese.

Instrucțiunea *process* are următoarea sintaxă:

process (nume-semnal, nume-semnal, …, nume-semnal)
  declarații de tipuri
  declarații de variabile
  declarații de constante
  definiții de funcții
  definiții de proceduri
 begin
   instrucțiune-secvențială
   ….
   instrucțiune-secvențială
 end process;

Intr-un proces sunt vizibile numai tipurile, semnalele, constantele, funcțiile şi procedurile care fac parte din aceeaşi arhitectură cu procesul, însă toate aceste elemente cu excepția semnalelor pot fi definite şi local in proces.

In cadrul unui proces, variabilele au rolul de a păstra stări, ele nu sunt vizibile in afara procesului. În funcție de modul de utilizare, o variabilă poate genera sau nu un semnal correspondent într-o implementare fizică a circuitului modelat.

Sintaxa *VHDL* pentru definirea unei variabile :

variable *nume-variabile : tip-variabile ;*

Semnalele care se află în paranteze alături de cuvântul cheie *process,* determină dacă procesul rulează sau dacă este suspendat, această listă de semnale poartă numele de listă de sensibilitate.

Presupunem că un proces este suspendat inițial. Dacă unul dintre semnalele aflate în lista de sensibilitate îşi schimbă valoarea, procesul intră în execuție, începând cu prima instrucțiune şi până la ultima. Dacă valoarea oricărui semnal din lista de sensibilitate se modifică datorită execuției procesului, acesta se execută din nou. Astfel procesul va rula până în momentul în care nici unul dintre semnale nu-şi mai schimbă valoarea. Toate evenimentele din cadrul unui proces au loc, în cadrul unei simulări într-un "timp de simulare" egal cu zero.

Un proces descris corect în *VHDL* se va suspenda după una sau mai multe rulări, trebuie să se evite modelarea proceselor care nu se suspendă niciodată. Lista de sensibilitate este opțională, procesele care nu au listă de sensibilitate încep să ruleze în cadrul simulării la momentul zero, aceste procese sunt utile pentru modelarea *test bench*-urilor.

Limbajul *VHDL* are câteva tipuri de instrucțiuni secvențiale.

1. ***Atribuirea secvențială de semnal*** (sequential signal-assignment):

   nume-semnal **<=** *expresie ;*

Instrucțiunea are aceeaşi sintaxă ca şi varianta concurentă, dar se utilizează doar în interiorul procesului, şi nu în arhitectură.

2. ***Atribuirea secvențială de variabilă***:

   nume-variabilă **: =** *expresie ;*

Exemplu. Arhitectura detectorului de numere prime rescrisă ca proces.

```
architecture   prime5_arch of prime is
        begin
           process (N)
                variable T1, T2,T3,T4 :  STD_LOGIC;
            begin
                T1  :=  not N(3) and N(0) ;
                T2  := not N(3) and not  N(2) and N(1) ;
                T3  := not N(2) and N(1) and N(0) ;
                T4  := N(2) and not N(1) and N(0) ;
                F    <= T1 or T2 or T3 or T4;
            end process ;
  end prime5_arch;
```

În cadrul acestei arhitecturi (*prime5_arch*) există doar o singură instrucțiune concurentă, aceasta este instrucțiunea *process.* În lista de sensibilitate a procesului apare intrarea *N*, iar în cadrul procesului se definesc variabile, definirea de semnale nu este permisă. Ieşirile porților AND trebuie definite ca variabile, deoarece definițiile de semnale nu sunt acceptate într-un proces.

3. Instrucțiunea if

Sintaxa:

a) **if** expresie-booleană **then** instrucțiune secvențială
   end if;

Expresia booleană este testată şi, dacă ea este adevărată (TRUE), se va executa o instrucțiune secvențială.

b) **if** expresie-booleană **then** instrucțiune secvențială ;
   else instrucțiune secvențială ;
   end if;

În această formă se mai adaugă şi clauza *else* urmată de o altă instrucţiune secvenţială care se execută în cazul în care expresia testată este falsă (nu se verifică).

c) **if** expresie-booleană **then** instrucţiune secvenţială ;
elsif expresie-booleană **then** instrucţiune secvenţială ;
. . .
elsif expresie-booleană **then** instrucţiune secvenţială ;
end if;

d) **if** expresie-booleană **then** instrucţiune secvenţială ;
elsif expresie-booleană **then** instrucţiune secvenţială ;
. . .
elsif expresie-booleană **then** instrucţiune secvenţială ;
else instrucţiune secvenţială **;**
end if;

În aceste forme se introduce cuvântul cheie *elsif*. O instrucţiune secvenţială condiţionată de clauza *elsif* se execută dacă expresia booleană căreia i s-a aplicat aceasta este adevărată şi toate expresiile precedente sunt false. Instrucţiunea secvenţială corespunzătoare clauzei finale *else* se execută dacă toate expresiile booleene precedente au fost false.

Exemplu. Codul VHDL pentru detectorului de numere prime folosindu-se instrucţiunea *if*.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity prime is
     port ( N: in STD_LOGIC_VECTOR (3 downto 0);
            F: out STD_LOGIC) ;
end prime;
architecture   prime6_arch of prime is
  begin
    process (N)
      variable NI : INTEGER;
        begin
          NI  :=   to_integer(unsigned (N));
    if NI=1  or NI=2  or   NI=3 or NI=5  or NI=7  or NI=11  or NI=13  then F<='1' ;
     else F <=        '0' ;
   end if ;
   end process ;
  end prime6_arch;
```

În acest caz se foloseşte o variabilă *NI* pentru a păstra valoarea întreagă (de tip integer) rezultată în urma convertiri intrării *N*.

4. ***Instrucțiunea case***

Se foloseşte când trebuie aleasă o singură alternativă din multitudinea de alternative oferite de valorile pe care le poate lua un semnal sau o expresie.

Sintaxa instrucțiunii *case:*

```
case expresie  is
        when opțiuni  => instrucțiuni-secvențiale ;
        . . .
        when opțiuni  => instrucțiuni-secvențiale ;
        end case ;
```

Această instrucțiune evaluează o expresie dată, alege valoarea care se potriveşte din una din opțiuni (*choices*) şi execută instrucțiunea secvențială corespunzătoare.

Opțiunile (*choices*) sunt reprezentate de o singură valoare sau de un set de valori separate prin bare verticale (|), aceste opțiuni trebuie să se excludă una pe cealaltă şi trebuie să includă toate valorile posibile ale expresiei evaluate, altfel se foloseşte clauza *others*.

Exemplu. Arhitectura pentru detectorul de numere prime, folosind instrucțiunea *case.*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity prime is
        port ( N: in STD_LOGIC_VECTOR (3 downto 0);
              F: out STD_LOGIC) ;
end prime;

architecture   prime7_arch of prime is
begin
           process (N)
              begin
                 case to_integer(unsigned (N))   is
                    when 1  => F <= '1' ;
                    when 2  => F <= '1' ;
    when 3 |  5 |  7  |   11  |   13    => F <= '1' ;
                    when others  =>  F <= '0' ;
                 end case ;
            end process ;
 end prime7_arch;
```

5. **Instrucțiunile** ***for loop, while loop*** **şi** ***loop***

Se utilizează pentru a indeplini mai multe iterații în procese, funcții şi proceduri.

Sintaxa:

a) **for** identificator **in** domeniu **loop**
instrucțiune-secvențială;
. . .
instrucțiune-secvențială;
end loop;

Variabila *identificator* se declară implicit şi este de acelaşi tip cu domeniul (*range*), pe care îl parcurge de la stânga la dreapta câte o valoare la fiecare iterație.

Domeniul (range) poate fi:

5 **downto** 0
0 **to** 5

În limbajul VHDL, instruc□iunea ***loop*** are mai multe particularită□i, care o deosebesc de alte limbaje de programare. În unele limbaje de programare, identificatorului din interiorul ciclului i se poate atribui o valoare oarecare. În limbajul VHDL acest lucru nu este posibil.

O altă particularitate este faptul că identificatorul este declarat doar în interiorul instruc□iunii ***for loop*** □i nu este necesară declararea explicită a lui în proces, func□ie sau procedură. Din această cauză identificatorul nu poate fi utilizat în calitate de valoare returnată în func□ii □i proceduri. Dacă în interiorul procedurii, dar în afara ciclului ***loop***, există o altă variabilă cu acela□i nume, ea este tratată ca o variabilă aparte.

```
process (i)
  begin
    ...
   x <= i+1;   -- x este un semnal
   ...
   for  i   in 1 to 10   loop
     q(i) := a;   -- q este o variabilă
   end loop;
   ...
end process;
```

Exemplu. Un generator al bitului de paritate pentru un cuvânt binar de 10 biți, utilizând instrucțiunea **for-loop.**

```
library ieee;
use ieee.std_logic_1164.all;
entity   parity10  is
  port  (D: in std_logic_vector(0 to 9);
         ODD: out std_logic);
         constant WIDTH: integer := 10;
end parity10;
```

```
architecture par of parity10 is
  begin
    process  (D)
      variable x: Boolean;
        begin
         x := false;
          for i in 0 to D'length - 1 loop  -- se utilizează atributul length
            if D(i) = '1' then
              x := not x;
           end if;
         end loop;
             if x  then
               ODD <= '1';
             else
               ODD <= '0';
             end if;
 end process;
 end par;
```

Programul va genera bitul de imparitate, dacă variabila x va primi inițial valoarea TRUE.

b) while expresie-booleană **loop**

```
while expresie-booleană  loop
     instrucțiune-secvențială;
     . . .
     instrucțiune-secvențială;
     end loop;
```

În acest tip de instrucțiune expresia booleană este testată înaintea de fiecare iterație, iar bucla se execută numai dacă valoarea expresiei este adevărată.

Instrucțiunea se foloseşte pentru descrierea proceselor secvențiale în programe speciale de generare a stimulilor de intrare, numite test-bench. Nu se recomandă utilizarea ei în programele de descriere a circuitelor, deoarece compilatorul nu întotdeauna poate sintetiza instrucțiunea.

Exemplu. Generator al semnalului de ceas, care poate fi utilizat într-un test-bench. Semnalul de ceas se va genera până când fanioanele ***error_flag*** sau ***done*** nu vor fi egale cu 1.

```
process
  begin
    while error_flag /= '1' and done /= '1' loop
        Clock <= not Clock;
```

```
        wait for CLK_PERIOD/2;
    end loop;
end process;
```

c) loop **--** ciclu infinit, se utilizează cu instr. next sau exit

```
   instrucțiune-secvențială;
   . . .
   instrucțiune-secvențială;
   end loop;
```

La fel ca şi instrucțiunea while-loop nu poate fi sintetizată.

6. Instrucțiunile *exit* şi *next*

Sunt instrucțiuni secvențiale care se pot executa în cadrul unei instrucțiuni *loop.*

Instrucțiunea ***exit*** **t**ransferă comanda către instrucțiunea ce urmează imediat după sfârşitul buclei. Poate fi utilizată pentru toate tipurile de instrucțiuni ***loop.***

Poate fi:

- necondițională: ***exit***
- condițională: ***exit when***

Sintaxa:

exit [label] [when condition] ;

Exemplu. Generator al semnalului de ceas, care poate fi utilizat într-un test-bench în care se folose□te instrucțiunea ***exit***.

```
process
  begin
    loop
      Clock <= not Clock;
      wait for CLK_PERIOD/2;
      if done = '1' or error_flag = '1' then
        exit;
      end if;
    end loop;
end process;
```

Exemplu. Utilizarea instrucțiunii condiționale ***exit***.

```
L2: loop
   A:=A+1;
 exit L2 when A>10;
 end loop L2;
```

Instrucțiunea ***next*** face ca toate instrucțiunile rămase până la sfârşitul buclei să fie ocolite şi să înceapă o nouă execuție a buclei.

Sintaxa:
next [label] [when condition] ;

Exemplu. Un numărător de unită□i pentru cuvinte binare de 16 bi□i.

```
library ieee;
use ieee.numeric_bit.all;
entity count_ones is
 port (v: in bit_vector (15 downto 0);
 count: out signed (3 downto 0));
end count_ones;
architecture functional of count_ones is
begin
process (v)
 variable result: signed (3 downto 0);
 begin
 result := (others => '0');
 for i in v'range loop
 next when v(i) = '0';
 result := result + 1;
 end loop;
 count <= result;
 end process;
end functional;
```

7. Instrucțiunea *wait*

În locul unei liste de sensibilitate, un proces poate conține o instrucțiune wait. Utilizarea unei instrucțiuni wait are două scopuri:

- Suspendarea execuției unui proces;
- Specificarea condiției care va determina activarea procesului suspendat.

La întâlnirea unei instrucțiuni wait, procesul în care apare această instrucțiune este suspendat. Atunci când se îndeplineşte condiția specificată în cadrul instrucțiunii wait, procesul este activat şi se execută instrucțiunile acestuia până când se întâlneşte din nou instrucțiunea wait.

Limbajul VHDL permite ca un proces să conțină mai multe instrucțiuni wait. Atunci când se utilizează pentru modelarea logicii combinaționale în vederea sintezei, un proces poate conține însă o singură instrucțiune wait.

Dacă un proces conține o instrucțiune wait, acesta nu poate conține o listă de sensibilitate. Formele instrucțiunii wait:

- wait on *listă_de_sensibilitate*. Procesul se execută când apare o modificare a valorii semnallor din lista de sensibilitate; Exemplu: ***wait on*** a,b;

- wait until *expresie_condițională*; Procesul se suspendă până când condiția specificată devine adevărată datorită modificării unuia din semnalele care apar în expresia condițională. Dacă nici un semnal din această expresie nu se modifică,

procesul nu va fi activat, chiar dacă expresia condiţională este adevărată. Exemplu: ***wait until*** ((x*y)<100);

- wait for *expresie_de_timp*. Permite suspendarea execuției unui proces pentru un timp specificat, de exemplu: **wait for** 10 ns;

Exemplele următoare prezintă mai multe forme ale instrucțiunii wait until:

wait until *semnal* = *valoare*;
wait until *semnal*'**event and** *semnal* = *valoare*;
wait until not *semnal*'**stable and** *semnal* = *valoare*;

unde *semnal* este numele unui semnal, iar *valoare* este valoarea care se testează. Dacă semnalul este de tip bit, atunci pentru valoarea '1' se aşteaptă frontul crescător al semnalului, iar pentru '0' frontul descrescător.

Instrucțiunea wait until se poate utiliza pentru implementarea unei funcționări sincrone. În mod obişnuit, semnalul testat este un semnal de ceas. De exemplu, aşteptarea frontului crescător al unui semnal de ceas se poate exprima în următoarele moduri:

wait until clk = '1';
wait until clk'**event and** clk = '1';

Pentru descrierile destinate sintezei, instrucțiunea wait until trebuie să fie prima din cadrul procesului. Din această cauză, logica sincronă descrisă cu o instrucțiune wait until nu poate fi resetată în mod asincron.

Exemplu. Implementarea unui circuit cu resetare sincronă, utilizând instrucțiunile ***wait*** şi ***loop.*** Semnalul RESET trebuie verificat imediat după fiecare instrucțiune ***wait***.

```
process
  begin
    RESET_LOOP: loop
      wait until CLOCK'event and CLOCK = '1';
      next RESET_LOOP when (RESET = '1');
        X <= A;
     wait until CLOCK'event and CLOCK = '1';
      next RESET_LOOP when (RESET = '1');
        Y <= B;
  end loop RESET_LOOP;
end process;
```

Exemplu. Utilizarea procesului cu listă de sensibilitate □i fără listă de sensibilitate, dar cu instruc□iunea wait.

```
PROCESS (clk)
  VARIABLE last_clk : std_logic := 'X';
BEGIN
  IF (clk /= last_clk ) AND (clk = '1') THEN
   q <= din AFTER 25 ns;
END IF;

last_clk := clk;

END PROCESS;
```

```
PROCESS
  VARIABLE last_clk : std_logic := 'X';
BEGIN
  IF (clk /= last_clk ) AND (clk = '1') THEN
   q <= din AFTER 25 ns;
  END IF;

  last_clk := clk;

  WAIT ON clk;
END PROCESS;
```

Instruc□iunea wait este situată la sfâr□itul procesului pentru a permite instruc□iunilor din proces de fi executate o dată.

Sinteza circuitelor modelate cu limbajul *VHDL*

Limbajul *VHDL* a fost mai întâi dezvoltat ca limbaj de descriere şi simulare a circuitelor şi abia mai târziu acest limbaj a fost adaptat pentru sinteză. Astfel că limbajul are multe caracteristici şi construcții care nu pot fi sintetizate.

Recomandări pentru sinteză:

- Structurile de control seriale de tipul *if-elsif-elsif-else* pot fi sintetizate sub forma unui lanț serial de porți logice, ceea ce favorizează întârzierile, astfel că în acest caz este mai bine să se folosească instrucțiunile de selecție *case* sau *with* dacă condițiile se exclud reciproc.
- În cazul în care se folosesc instrucțiuni condiționale în cadrul unui proces, dacă pentru o anumită combinație a intrărilor se omite specificarea valori pe care trebuie să o ia aibă ieşirea, compilatorul va crea un latch (bistabil) la ieşire care să poată păstra vechea valoare a semnalului de ieşire care altfel ar trebui să se schimbe. În general generarea unui astfel de latch nu este dorită.
- Instrucțiunile de ciclare creează în general copii multiple ale logicii combinaționale descrise în cadrul instrucțiunii. Dacă se doreşte folosirea unei singure astfel copii a logici combinaționale într-o secvență de paşi atunci trebuie să se proiecteze un circuit secvențial.

Definirea întârzierilor

Fără specificarea întârzierilor, modelarea circuitului are lor întru-un timp de simulare egal cu zero.

Limbajul VHDL permite modelarea întârzierilor.

Unul dintre cuvintele cheie care permit introducerea întârzierilor este *after* care poate fi asociat cu orice tip de atribuire de semnal (secvențială, concurențială, condițională şi selectată).

Arhitectura porți logice interdicție (*BUT-NOT*), poate fi rescrisă după cum urmează:

Z <= '1' *after 4ns when* X = '1' *and* Y ='0' *else '0' after 3 ns;*

Această poartă este astfel modelată încât semnalul de ieşire față de cel de intrare va avea o întârziere de 4ns la tranziția din *0* în *1* şi 3ns la tranziția din *1* în *0*. O altă instrucțiune care invocă dimensiunea timp este instrucțiunea secvențială *wait.* Aceasta poate fi folosită pentru a suspenda executarea unui proces pentru o anumită perioadă de timp.

Aceste două instrucțiuni care invocă dimensiunea timp nu sunt sintetizabile, ele sunt utile însă la modelarea test-bench-urilor.

Exemplu. Programul, numit *test-bench*, care foloseşte instrucțiunea *wait* pentru a genera formele de undă simulate necesare la testarea funcționării porții logice interdicție.

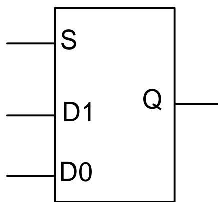
```
entity BUT_NOT_TB is
end BUT_NOT_TB ;
architecture BUT_NOT_TB_arch  of  BUT_NO_TB  is
component BUT_NOT port (X,Y : in  BIT ; Z : out  BIT) ;  end
component ;
signal XI, YI, ZI :  BIT ;
  begin
    U1 : BUT_NOT  port map  (XI, YI, ZI) ;
       process
         begin
           XI <= `0` ;  YI <= `0` ;
           wait for 10 ns ;
           XI <= `0` ;  YI <= `1` ;
           wait for 10 ns ;
           XI <= `1` ;  YI <= `0` ;
           wait for 10 ns ;
           XI <= `1` ;  YI <= `1` ;
           wait ;  -- suspendă procesul pentru un timp nedetrminat
         end process ;
    end BUT_NOT_TB_arch ;
```

Descrierea circuitelor combinaționale în VHDL

Multiplexoare

Multiplexorul este un comutator digital care transmite la ieşire datele de la una dintre cele n surse de la intrare.

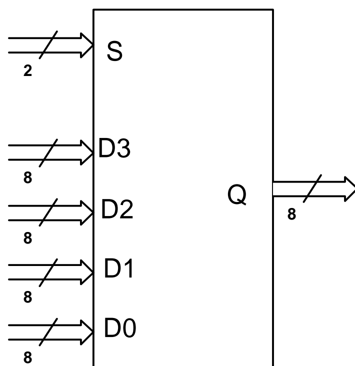
Mux $2 \rightarrow 1$



Program VHDL bazat pe flux de date:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux2_1 is
port (s: in std_logic;
         D1,D0 : in std_logic;
         Q : out std_logic);
end mux2_1;
architecture mux2_1_arch of mux2_1 is
begin
Q <= D1 when (s = '1') else D0;
end mux2_1_arch;
```

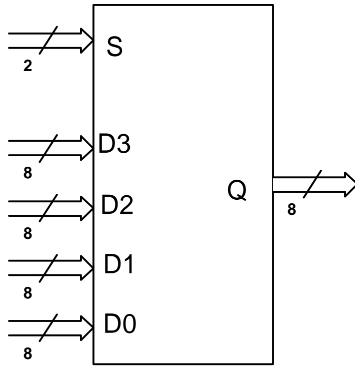
Mux $4 \rightarrow 1$ (8*biti pe linie*)



Program VHDL bazat pe flux de date, utilizând instrucțiunea ***select***

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4in8b is
port(
   s: in std_logic_vector (1 downto 0);
   D3, D2, D1, D0: in std_logic_vector(1 to 8);
   Q: out std_logic_vector(1 to 8)  );
end mux4in8b;
architecture mux4in8b_arch of mux4in8b is
  begin
     with S select Q <=
        D0  when "00",
        D1  when "01",
        D2  when "10",
        D3  when "11",
        (others  => 'U' ) when others;  -- creaza un
 --vector de 8 biti  din `U`, valoare
-- neinitializata
end mux4in8b_arch;
```

$Mux\ 4 \rightarrow 1\ (8 biti\ pe\ linie)$

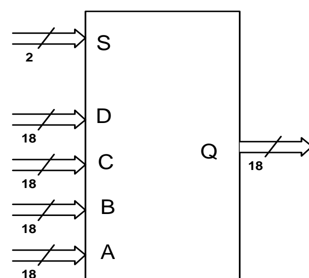


Program VHDL comportamental

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4in8b is
port(    S: in std_logic_vector (1 downto 0);
         D3, D2, D1, D0: in std_logic_vector(1 to 8);
         Q: out std_logic_vector(1 to 8)  );
end mux4in8b;
architecture mux4in8b_arch of mux4in8b is
  begin
    process (S, D3, D2, D1, D0)
      begin
       case  S is
       when "00"  =>  Q  <=  D0;
       when "01"  =>  Q  <=  D1;
       when "10"  =>  Q  <=  D2;
       when "11"  =>  Q  <=  D3;
       when others   =>  Q  <=  (others =>'U') ;
      end case;
end process;
end mux4in8b_arch;
```

Multiplexor specializat care selectează una din cele patru magistrale de intrare de 18 biți **A, B, C** şi **D**, pentru a comanda o magistrală de ieşire de 18 biți **Q.**

$Mux\ 4 \rightarrow 1\ (18 biti\ pe\ linie)$



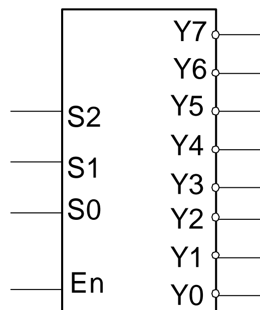
S2	S1	S0	Q
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	B

```
library IEEE; -- comportamental
use IEEE.std_logic_1164.all;
entity mux4in18 is
port(
   S: in std_logic_vector (2 downto 0);
   A, B, C, D: in std_logic_vector(1 to 18);
   Q: out std_logic_vector(1 to 18)  );
end mux4in18;
architecture mux4in18b_arch of mux4in18 is
   begin
   process (S, A, B, C, D)
  -- variable i : INTEGER;
      begin
      case  S   is
when "000"|"010" |"100" |"110"  =>  Q  <= A;
        when "001"|"111"  =>  Q  <=  B;
        when "011"  =>  Q  <=  C;
       -- when "101"  =>  Q  <=  D;
        when others   =>  Q  <=  (others =>'U') ;
     end case;
  end process;
end mux4in18b_arch;
```

Decodificatoare

Decodificatorul este un circuit care activează o singură ieşire din 2^n posibile, corespunzător codului binar aplicat la cele n intrări.

***DC* 3 → 8**



Program VHDL comportamental pentru un decodificator 3→8 cu intrări active pe 0.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dec3to8 is
port ( S: in std_logic_vector (2 downto 0); --intrari de selectare
        EN: in std_logic; -- enable
        Y: out std_logic_vector (7 downto 0)); -- iesirile sunt active pe zero
end dec3to8;
architecture dec3to8_arch  of  dec3to8  is
begin
process (S,EN)
begin
  y <= "11111111";
```

```
  if (en = '1') then
     case S  is
when "000" => Y(0) <= '0';
when "001" => Y(1) <= '0';
when "010" => Y(2) <= '0';
when "011" => Y(3) <= '0';
when "100" => Y(4) <= '0';
when "101" => Y(5) <= '0';
when "110" => Y(6) <= '0';
when "111" => Y(7) <= '0';
when others => null;
     end case;
   end if;
end process;
end dec3to8_arch ;
```

În cazul în care vreuneia dintre ieşiri nu i se atribuie nici o valoare, sintetizatorul va presupune ca ieşirea respectivă trebuie să-şi păstreze valoarea curentă şi astfel se va genera un latch pentru ieşirea respectivă, cea ce înseamnă a irosi din resursele cipului în care urmează a fi implementat modelul şi de asemenea întârzieri în plus.

Un program VHDL comportamental mai flexibil care nu include tabelul de adevăr în cod:

```
library IEEE;
 use IEEE.std_logic_1164.all;
USE ieee.numeric_std.ALL;
entity dec3to8 is
port ( S: in std_logic_vector (2 downto 0); --intrari de selectare
       EN: in std_logic; -- enable
       Y: out std_logic_vector (7 downto 0)); -- iesirile sunt active pe zero
end dec3to8;
architecture dec3to8_comp  of  dec3to8  is
   begin
     process (S,EN)
       variable i : INTEGER range 0  to 7;
         begin
           Y <= "11111111";
           if (EN = '1') then
             for  i   in   0   to   7  loop
             if   i = to_integer(unsigned(S))   then   Y(i)  <=  '0' ;
            end if;
           end loop;
          end if;
```

```
      end process;
end dec3to8_comp ;

library IEEE;
 use IEEE.std_logic_1164.all;
USE ieee.numeric_std.ALL;
entity dec3to8 is
port ( S: in std_logic_vector (2 downto 0); --intrari de selectare
        EN: in std_logic; -- enable
        Y: out std_logic_vector (7 downto 0)); -- iesirile sunt active pe zero
end dec3to8;
architecture dec3to8_comp  of  dec3to8  is
   begin
      process (S,EN)
          begin
             Y <= "11111111";
             if (EN = '1') then
               Y(to_integer(unsigned(S))) <= '0';
            end if;
      end process;
end dec3to8_comp ;

library IEEE;
 use IEEE.std_logic_1164.all;
USE ieee.numeric_std.ALL;
entity dec3to8 is
port ( S: in std_logic_vector (2 downto 0); --intrari de selectare
        EN: in std_logic; -- enable
        Y: out std_logic_vector (7 downto 0)); -- iesirile sunt active pe zero
end dec3to8;
architecture dec3to8_comp  of  dec3to8  is
   begin
      process (S,EN)
        variable i :INTEGER;
          begin

             Y <= "11111111";
             if (EN = '1') then
               i := to_integer(unsigned(S));
               Y(i) <= '0';
            end if;
      end process;
end dec3to8_comp ;
```

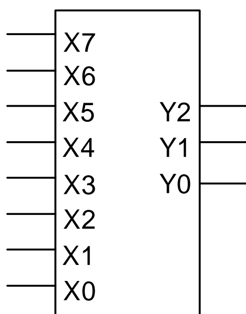
Acest program poate fi adaptat cu mai multă uşurinţă pentru a realiza decodoare binare de orice dimensiuni.

O altă variantă de cod VHDL pentru decodificator, de data aceasta bazat pe conceptul de flux de date:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dec3to8_alt  is
port (S: in std_logic_vector(2 downto 0);
        En: in std_logic;
        Y_L: out std_logic_vector(0 to 7));
end dec3to8_alt;
architecture dec3to8_alt  of  dec3to8_alt  is
signal YI : STD_LOGIC_VECTOR (0 to 7);
begin
  with S select YI  <=
"01111111"   when   "000" ,
"10111111"   when   "001" ,
"11011111"   when   "010" ,
"11101111"   when   "011" ,
"11110111"   when   "100" ,
"11111011"   when   "101" ,
"11111101"   when   "110" ,
"11111110"   when   "111" ,
"11111111"   when    others;
   Y_L  <=  YI  when En ='1'   else "11111111";
end dec3to8_alt;
```

Codificatoare

Un codificator binar cu 2^n intrări şi n ieşiri va genera la ieşire un cuvânt de cod la ieşire când se va activa o linie la intrare.



Program VHDL bazat pe flux de date

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CD is port (
   X : in  std_logic_vector (7 downto 0);
   Y : out std_logic_vector (2 downto 0));
end CD;
architecture CD_arch of CD is
begin
        Y <=
                "000" when X(0)='1' else
                "001" when X(1)='1' else
                "010" when X(2)='1' else
                "011" when X(3)='1' else
                "100" when X(4)='1' else
```

```
                         "101" when X(5)='1' else
                         "110" when X(6)='1' else
                         "111" when X(7)='1' else
                         "000";
              end CD_arch;
```

Un codificator prioritar poate avea la intrare mai multe linii activate. El va genera la ieşire un cuvânt de cod care corespunde liniei activate cu prioritatea cea mai mare.

Pentru descrierea unui codificator prioritar se va folosi construcţia VHDL *if-then-else* care descrie cel mai bine o ierarhie de priorităţi. Această construcţie se utilizează doar în procese, deci descrierea este comportamentală.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CD_PR is port (
  X : in  std_logic_vector (7 downto 0);
  Y : out std_logic_vector (2 downto 0));
end CD_PR;
architecture CD_PR_arch of CD_PR is
begin
process(X)
begin
        Y <= "000";
        if      X(7)='1' then Y <= "111";
        elsif   X(6)='1' then Y <= "110";
        elsif   X(5)='1' then Y <= "101";
        elsif   X(4)='1' then Y <= "100";
        elsif   X(3)='1' then Y <= "011";
        elsif   X(2)='1' then Y <= "010";
        elsif   X(1)='1' then Y <= "001";
        elsif   X(0)='1' then Y <= "000";
        end if;
end process;
end CD_PR_arch;
```

Deoarece toate instrucţiunile în interiorul unui process se execută secvenţial, prioritatea cea mai mare o are prima instrucţiune. Astfel intarea X(7) are prioritatea cea mai mare.

Sumatoare binare

Sumator cu transport succesiv pe 8 biţi.

Expresiile logice pentru un sumatior complet de 1 bit:

$$S = a \oplus b \oplus c_{in}$$

$$C_{out} = a \cdot b + c_{in} \cdot (a + b)$$

Pentru descrierea sumatorului se foloseşte instrucţiunea **loop** cu ajutorul căreia se generează logica specifică sumatorului.

În cod se defineşte un semnal temporar signal "*c: std_logic_vector(7 downto 0);*", pentru a păstra valoarea semnalului carry care se propagă intern.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity adder8 is port (
   a,b: in std_logic_vector (7 downto 0); -- semnalele de intrare
   cin: in std_logic; -- transport la intrare
   sum: out std_logic_vector(8 downto 0); -- semnale de ieşire
   cout: out std_logic); -- transport la ieşire
end adder8;
architecture adder8_arch of adder8 is
signal c: std_logic_vector(8 downto 0);
  begin
    process (a,b,cin,c)
      begin
        c(0) <= cin;
        for i in 0 to 7 loop
          sum(i) <= a(i) xor b(i) xor c(i);
          c(i+1) <= (a(i) and b(i)) or (c(i) and (a(i) or b(i)));
        end loop;
      cout <= c(8);
    end process;
end adder8_arch;
```

În următorul exemplu se prezintă descrierea unui sumator cu transport succesiv în care se utilizează un generic ***n***, care specifică dimensiunea sumatorului. În cadrul entităţii, acest generic este utilizat pentru definirea dimensiunii porturilor. În cadrul arhitecturii, genericul ***n*** este utilizat pentru specificarea domeniului buclei ***for.***

```
library ieee;
use ieee.std_logic_1164.all;
  entity add_succ is
    generic (n: natural);
    port (a, b: in std_logic_vector (n-1 downto 0);
           cin: in std_logic;
           s: out std_logic_vector (n-1 downto 0);
           cout: out std_logic);
  end add_succ;
architecture structural of add_succ is
```

```
  begin
    process (a, b, cin)
      variable c: std_logic;
        begin
          c := cin;
          for i in 0 to n-1 loop
            s(i) <= a(i) xor b(i) xor c;
            c := (a(i) and b(i)) or (a(i) and c) or  (b(i) and c);
          end loop;
            cout <= c;
    end process;
end structural;
```

În următorul exemplu se descrie un sumator pe 4 biți utilizând instrucțiuni concurente de asignare a semnalelor şi instrucțiunea ***for generate***. În acest exemplu, semnalul de transport ***c*** s-a transformat într-un vector. Fiecare bit de transport este un semnal separat, iar numărul de semnale necesare depinde de parametrul generic **n**. Dimensiunea vectorului este dată de parametrul generic, această dimensiune fiind mai mare cu 1 față de dimensiunea operanzilor care se adună pentru a furniza un transport de ieşire.

```
library ieee;
use ieee.std_logic_1164.all;
entity add_succ is
generic (n: natural := 4);
port (a, b: in std_logic_vector (n-1 downto 0);
        cin: in std_logic;
        s: out std_logic_vector (n-1 downto 0);
        cout: out std_logic);
end add_succ;
architecture structural of add_succ is
  signal c: std_logic_vector (n downto 0);
    begin
      c(0) <= cin;
      gen: for i in 0 to n-1 generate
              s(i) <= a(i) xor b(i) xor c(i);
              c(i+1) <= (a(i) and b(i)) or (a(i) and c(i)) or
                          (b(i) and c(i));
    end generate;
        cout <= c(n);
end structural;
```

O altă descriere a sumatorului foloseşte operatorii aritmetici. Aceşti operatori sunt aplicabili doar tipurilor reale, întregi şi fizice. Ei nu pot fi aplicați tipurilor BIT_VECTOR şi STD_LOGIC_VECTOR. Operatorii aritmetici sunt

definiți în pachhetul IEEE.std_logic_arith. Acest pachet defineşte două tipuri tablou noi, SIGNED şi UNSIGNED şi un set de funcții de comparare pentru operanzii de tipurile INTEGER, SIGNED şi UNSIGNED.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity adder  is port (
   a,b: in unsigned (7 downto 0);
    c: in signed (7 downto 0);
    d: in std_logic_vector (7 downto 0);
     s: out unsigned (8 downto 0);
     t: out signed (8 downto 0);
     u: out signed (7 downto 0);
     v: out std_logic_vector(8 downto 0)  );
   end adder8;
architecture adder_arch of adder8 is
  begin
    s <= ('0'& a) + ('0' & b);
     t <= a+c ;
     u <= c + signed (d) ;
     v <= c - unsigned (d) ;
end adder_arch;
```

Rezultatul ***s*** are 9 biți pentru a afla transportul rezultant.Operatorul de concatenare **&** este folosit pentru a extinde operanzii ***a*** şi ***b*** astfel ca funcția de adunare să returneze bitul de transport în poziția MSB a rezultatului.

Lungimea unui rezultat este, în mod normal, egală cu cea mai mare dintre lungimile operanzilor. Însă când un operand UNSIGNED se combină cu un operand SIGNED sau INTEGER, lungimea acestui operand se măreşte cu 1 pentru a include şi un bit de semn 0, apoi se stabileşte lungimea rezultatului. Astfel, rezultatul, ***t,*** are tot lungimea de 9 biți.

În mod normal, dacă vreunul dintre operanzi este de tipul SIGNED, rezultatul este SIGNED. La fel este şi pentru tipul UNSIGNED. Dar dacă rezultatul se atribuie unui semnal sau unei variabile de tipul STD_LOGIC_VECTOR, atunci rezultatul SIGNED sau UNSIGNED se converteşte în acel tip.

Rezultatul ***u*** este de tip SIGNED, de 8 biți. Intrarea ***d*** este convertită în SIGNED. În ultima instrucțiune ***d*** este convertit la UNSIGNED, fiind extins automat cu un bit şi scăzut din ***c***, rezultând ***v***, de 9 biți.

Descrierea circuitelor secvențiale în VHDL

Semnalele generate la ieşirile unui circuit secvențial depind atât de semnalele de intrare, cât şi de starea circuitului.

Starea prezentă a circuitului este determinată de o stare anterioară şi de valorile semnalelor de intrare.

Circuitele secvenţiale conţin elemente de memorare şi circuite combinaţionale.

Efectul de memorare se datorează unor legături inverse (bucle de reacţie) prezente în schemele logice ale acestor circuite.

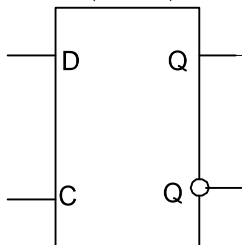
În cazul circuitelor secvenţiale sincrone, modificarea stării se poate realiza la momente bine definite de timp sub controlul unui semnal de ceas. În cazul circuitelor secvenţiale asincrone, modificarea stării poate fi cauzată de schimbarea aleatoare în timp a valorii unui semnal de intrare.

Comportamentul unui circuit asincron este mai puţin sigur, evoluţia stării fiind influenţată şi de timpii de întârziere ai componentelor circuitului. Trecerea între două stări stabile se poate realiza printr-o succesiune de stări instabile, aleatoare.

Circuitele secvenţiale sincrone sunt mai fiabile şi au un comportament predictiv. Toate elementele de memorare ale unui circuit sincron îşi modifică simultan starea, ceea ce elimină apariţia unor stări intermediare instabile. Prin testarea semnalelor de intrare la momente bine definite de timp se reduce influenţa întârzierilor şi a eventualelor zgomote.

Bistabile

Descrierea unui bistabil sincron de tip D acţionat pe nivelul semnalului de ceas (latch).



Tabelul de tranziţie

D_t	Q_{t+1}	nQ_{t+1}
0	0	1
1	1	0

Program VHDL comportamental

```
library IEEE;
use IEEE.std_logic_1164.all;
entity D_latch is
port (D, Clk : in std_logic;
                Q , nQ : out std_logic);
end entity D_latch;
architecture D_comp of D_latch is
begin
p0: process (Clk) is
begin
if (Clk = '1') then
Q <= D;
nQ <= not D;
end if;
end process p0;
end architecture D_comp;
```

Pornind de la această descriere, compilatorul VHDL „deduce logic” un circuit latch, întrucât codul nu arată cum trebuie de procedat în cazul în care clk nu este 1. Bistabilul creat va păstra valoarea ***Q*** între apelările procesului.

În general, un compilator VHDL deduce un circuit latch pentru un semnal căruia i se atribuie o valoare în cadrul unei instrucțiuni *if* sau *case*, dacă nu se iau în considerație toate combinațiile de intrare.

Majoritatea schemelor digitale modelate cu VHDL sunt sisteme sincrone, realizate cu bistabile active pe front (flip-flop). Pentru descrierea comportării activate pe front se foloseşte atributul ***event***. Atributul event poate fi ataşat unei denumiri de semnal pentru ca valoarea rezultată să fie *true* dacă semnalul îşi schimbă valoarea şi *false* în caz contrar.

Folosind atributul event putem crea un model de bistabil activ pe front. Construcția ***if (clk`event)*** întoarce valoarea true pe oricare front al semnalului de tact. Construcția ***if (clk`event and clk=`1`)*** întoarce valoarea true doar pe frontul crescător al semnalului de tact.

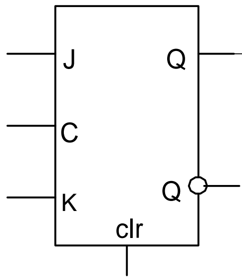
Codul VHDL pentru un bistabil sincron de tip D acționat pe frontul crescător al semnalului de ceas cu intrare asincronă de resetare.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity vdff is
port (D, Clk, Clr : in std_logic;
              Q , nQ : out std_logic);
end entity vdff;
architecture D_comp of vdff is
begin
p0: process (Clk, Clr) is
variable state: std_logic;
begin
      if clr='1' then Q <= '0'; nQ <= '1';
elsif rising_edge(Clk) then
--  eslif (clk'event and clk = '1') then  Q <= D;  nQ <= not D; acela□i efect
state := D;
end if;
Q <= state;
nQ <= not state;
end process p0;
end architecture D_comp;
```

Procesul utilizat pentru descrierea bistabilului este sensibil numai la modificările semnalului de ceas clk. Tranziția semnalului de intrare ***D*** nu determină activarea procesului. În acest model intrarea asincronă de ştergere CLR este dominantă față de orice comportare determinată de semnalul de tact CLK, deci este prima evaluată în clauza „if". Când CLR este negat, se evaluează clauza „elsif".

Descrierea unui bistabil de   Program comportamental

tip JK cu resetare asincronă



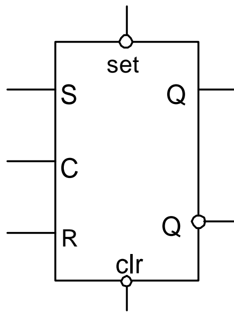

```
library ieee;
use ieee.std_logic_1164.all;
entity JK_FF is
port (   clk, J, K, clr:     in std_logic;
         Q, Qn:  out std_logic );
end JK_FF;
architecture behv of JK_FF is
   signal s: std_logic;
   signal x: std_logic_vector(1 downto 0);
begin
    x  <= J & K; -- vector JK concatenat;
process (clk, clr) is
   begin
     if (clr = '1') then s <= '0';
    elsif (rising_edge(clk)) then
       case (x) is
when "11" => s <= not s;
when "10" => s <= '1';
when "01" => s <= '0';
when others =>    null;
         end case;
         end if;
   end process;
   Q <= s; --instrucţiuni concurente
   Qn <= not s;
end behv;
```

Funcţia rising_edge este definită în pachetul std_logic_1164, având rolul de a detecta frontul crescător al unui semnal. Această funcţie se poate utiliza în locul expresiei (clk'event and clk = '1'), dacă semnalul clk este de tip std_logic. În acelaşi pachet este definită şi funcţia falling_edge, care detectează fronturile descrescătoare ale semnalelor. Aceste funcţii sunt preferate de către unii proiectanţi deoarece la simulare funcţia rising_edge, de exemplu, va asigura că tranziţia este de la '0' la '1' şi nu va ţine cont de alte tranziţii, cum este cea de la 'U' la '1'.

Descrierea unui bistabil de tip SR cu resetare şi setatare asincronă.

Program comportamental

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_arith.all;
use ieee. std_logic_unsigned.all;
entity SR-FF is
  port ( S,R,CLK,clr,set: in std_logic;
          Q, nQ: out std_logic);
```




```
end SR-FF;
architecture SR_arch of SR-FF is
  begin
    process (CLK,clr, set)
      variable x: std_logic;
        begin
          if (clr = '0')  then
            x:='0';
          elsif (set = '0')  then
              x:='1';
          elsif (CLK='1' and CLK'EVENT) then
            if( S='0' and R='0') then
                x:=x;
            elsif (S='1' and R='1') then
                x:='Z';
            elsif (S='0' and R='1')  then
                x:='0';
            else
                x:='1';
            end if;
        end if;

  Q<=x;
  nQ<=not x;
end process;
end SR_arch;
```

Registre

Descrierea unui registru de 8 biți. D şi Q sunt vectori. Registrul are un semnal de validare a ceasului (ce).

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
  port (clk, ce: in std_logic;
          D: in std_logic_vector (7 downto 0);
          Q: out std_logic_vector (7 downto 0));
end reg8;
architecture reg8_arch of reg8 is
    begin
      process (clk)
        begin
          if (clk'event and clk = '1') then
```

```
            if (ce = '1') then Q <= D;
            end if;
         end if;
      end process;
end reg8_arch;
```

Registre de deplasare

Un registru de deplasare este un circuit secvențial care deplasează la stânga sau la dreapta conținutul registrului cu o poziție în fiecare ciclu de ceas. De obicei, intrările unui registru de deplasare sunt reprezentate de semnalul de ceas, o intrare serială de date, un semnal de setare/resetare sincronă sau asincronă şi un semnal de validare a ceasului. În plus, un registru de deplasare poate avea semnale de control şi de date pentru încărcarea paralelă sincronă sau asincronă. Datele de ieşire ale unui registru de deplasare pot fi accesate fie serial, atunci când este accesibil numai conținutul ultimului bistabil pentru restul circuitului, fie în paralel, atunci când este accesibil conținutul mai multor bistabile.
Utilitarele de sinteză ale diferitor firme conțin resurse dedicate (ex. primitivele SRL16 şi SRL32, XILINX) care permit o implementare eficientă a registrelor de deplasare fără utilizarea unor bistabile suplimentare. Totuşi, aceste resurse permit numai operații de deplasare la stânga şi au un număr limitat de semnale de intrare/ieşire: ceas, validarea ceasului, intrare serială de date şi ieşirea sincronă.

Există mai multe posibilități pentru descrierea registrelor de deplasare în limbajul VHDL:
- Utilizarea operatorului de concatenare: reg <= reg (6 downto 0) & SI;
- Utilizarea construcțiilor for loop;
- Utilizarea operatorilor de deplasare predefiniți (sll, srl, sla, sra).

Descrierea unui registru de deplasare la stânga de 8 biți cu semnale de validare a ceasului, intrare serială şi ieşire serială. Pentru descrierea registrului de deplasare se utilizează o construcție for loop.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8_depl is
  port (clk, ce, si: in std_logic;
          so:          out std_logic);
end reg8_depl;
architecture reg_depl of reg8_depl is
   signal tmp: std_logic_vector (7 downto 0);
    begin
      process (clk)
         begin
          if (clk'event and clk = '1') then
            if (ce = '1') then
            for i in 0 to 6 loop
               tmp(i+1) <= tmp(i);
```

```
            end loop;
              tmp(0) <= si;
            end if;
          end if;
      end process;
   so <= tmp(7);
end reg_depl;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_bit.all;
entity test is
port(clk,il,ir :in bit;
 s :in bit_vector(1 downto 0);
 i : in bit_vector(3 downto 0);
 q : out bit_vector(3 downto 0));
end test;
architecture beh2 of test is
signal qtmp: bit_vector(5 downto 0);
begin
 process(clk)
   begin
     if (clk = '1' and clk'event) then
          case s is
       when "00" => qtmp <= qtmp;
 when "01" =>   qtmp <=il&i&ir;
 when "10" =>   qtmp <= (il&qtmp(4 downto 1)&ir) sll 1;
 when "11" =>   qtmp<= (il&qtmp(4 downto 1)&ir) srl 1 ;
 when others =>   null;
     end case;
       end if;
 end process;
 q<=qtmp(4 downto 1);
end beh2;
```

Numărătoare

Descriea unui numărător de 3 biți.

```
library ieee;
use ieee.std_logic_1164.all;
entity num3 is
    port (clk: in std_logic;
```

```
        num: out integer range 0 to 7);
end num3;
architecture num3_integer of num3 is
  signal tmp: integer range 0 to 7;
    begin
      cnt: process (clk)
        begin
          if (clk'event and clk = '1') then
            tmp <= tmp + 1;
         end if;
      end process cnt;
       num <= tmp;
end num3_integer;
```

În exemplul anterior, pentru semnalul num, care este de tip integer, se utilizează operatorul de adunare. Majoritatea utilitarelor de sinteză permit această utilizare, convertind tipul integer la tipul bit_vector sau std_logic_vector.

Utilizarea tipului integer pentru porturi pune însă unele probleme:

1) Pentru a utiliza valoarea num într-o altă porțiune a proiectului pentru care interfața are porturi de tip std_logic, trebuie efectuată o conversie de tip.

2) Vectorii aplicați în timpul simulării codului sursă nu pot fi utilizați pentru simularea modelului generat în urma sintezei. Pentru codul sursă, vectorii trebuie să fie valori întregi. Modelul generat în urma sintezei necesită vectori de tip std_logic.

Deoarece operatorul nativ + al limbajului VHDL nu este definit pentru tipurile bit sau std_logic, acest operator trebuie redefinit înainte de adunarea operanzilor care au aceste tipuri. Standardul IEEE 1076.3 defineşte funcții pentru redefinirea operatorului + pentru următoarele perechi de operanzi: (unsigned, unsigned), (unsigned, integer), (signed, signed) şi (signed, integer). Aceste funcții sunt definite în pachetul numeric_std al standardului 1076.3.

În următorul exemplu se utilizează tipul unsigned pentru ieşirea numărătorului.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity num3 is
  port (clk: in std_logic;
        num: out unsigned (2 downto 0));
end num3;
architecture num3_unsigned of num3 is
  signal tmp: unsigned (2 downto 0);
    begin
      cnt: process (clk)
       begin
          if (clk'event and clk = '1') then
            tmp <= tmp + 1;
```

```
            end if;
        end process cnt;
        num <= tmp;
  end num3_unsigned;
```

De obicei, utilitarele de sinteză pun la dispoziție pachete suplimentare care redefinesc operatorii pentru tipul std_logic. Deşi acestea nu sunt pachete standard, ele se utilizează adesea de către proiectanți, deoarece permit operații aritmetice şi relaționale cu tipul std_logic, din acest punct de vedere fiind chiar mai utile decât pachetul numeric_std. De asemenea, aceste pachete nu necesită utilizarea a două tipuri suplimentare (signed, unsigned) în plus față de tipul std_logic_vector şi nici a funcțiilor necesare conversiei între aceste tipuri. La utilizarea unuia din aceste pachete pentru operațiile aritmetice, utilitarul de sinteză va utiliza pentru tipul std_logic_vector o reprezentare fără semn sau una cu semn (în complement față de 2) şi va genera componentele aritmetice corespunzătoare.
În următorul exemplu se prezintă descrierea modificată a numărătorului din exemplele precedente pentru a se utiliza pachetul std_logic_unsigned şi tipul std_logic_vector pentru ieşirea numărătorului.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num3 is
  port (clk: in std_logic;
         num: out std_logic_vector (2 downto 0));
end num3;
architecture num3_std_logic of num3 is
  signal tmp: std_logic_vector (2 downto 0);
    begin
      cnt: process (clk)
       begin
         if (clk'event and clk = '1') then
             tmp <= tmp + 1;
         end if;
       end process cnt;
         num <= tmp;
end num3_std_logic;
```

Numărător de 8 biți cu semnale asincrone de resetare şi setare.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num8 is
  port (clk: in std_logic;
         rst, set: in std_logic;
```

```
            en, load: in std_logic;
            data: in std_logic_vector (7 downto 0);
             num: out std_logic_vector (7 downto 0));
end num8;
architecture arh_num8 of num8 is
   signal tmp: std_logic_vector (7 downto 0);
     begin
       cnt: process (rst, set, clk)
            begin
                if (rst = '1') then
                    tmp <= (others => '0');
                 elsif (set = '1') then
                    tmp <= (others => '1');
                 elsif (clk'event and clk = '1') then
                    if (load = '1') then
                         tmp <= data;
                 elsif (en = '1') then
                         tmp <= tmp + 1;
                 end if;
             end if;
        end process cnt;
          num <= tmp;
end arh_num8;
```

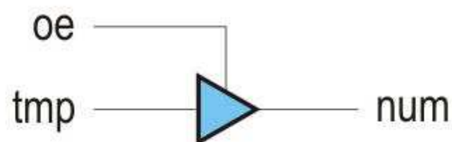
Majoritatea circuitelor programabile dispun de ieşiri cu trei stări sau semnale bidirecționale de I/E. În plus, anumite circuite dispun de buffere interne cu trei stări. Un semnal cu trei stări poate avea valorile '0', '1' şi 'Z', toate acestea fiind permise de tipul std_logic.

Descrierea modificată a numărătorului din exemplul precedent pentru a utiliza ieşiri cu trei stări. Acest numărător nu dispune de un semnal de setare asincronă.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity num8 is
   port (clk, rst: in std_logic;
            en, load: in std_logic;
            oe: in std_logic;
            data: in std_logic_vector (7 downto 0);
            num: out std_logic_vector (7 downto 0));
end num8;
architecture arh_num8 of num8 is
      signal tmp: std_logic_vector (7 downto 0);
        begin
    cnt: process (rst, clk)
```

```
    begin
      if (rst = '1') then
        tmp <= (others => '0');
      elsif rising_edge (clk) then
        if (load = '1') then
          tmp <= data;
      elsif (en = '1') then
          tmp <= tmp + 1;
         end if;
      end if;
  end process cnt;
  oep: process (oe, tmp)
    begin
      if (oe = '0') then
        num <= (others => 'Z');
     else
        num <= tmp;
      end if;
  end process oep;
end arh_num8;
```

În această descriere se utilizează un semnal suplimentar oe pentru controlul ieşirilor cu trei stări. Procesul etichetat cu oep descrie ieşirile cu trei stări ale numărătorului. Dacă semnalul oe nu este activat, ieşirile sunt trecute în starea de înaltă impedanță. Descrierea procesului oep este în concordanță cu funcționarea unui buffer cu trei stări:



Proiectarea automatelor de stare cu VHDL

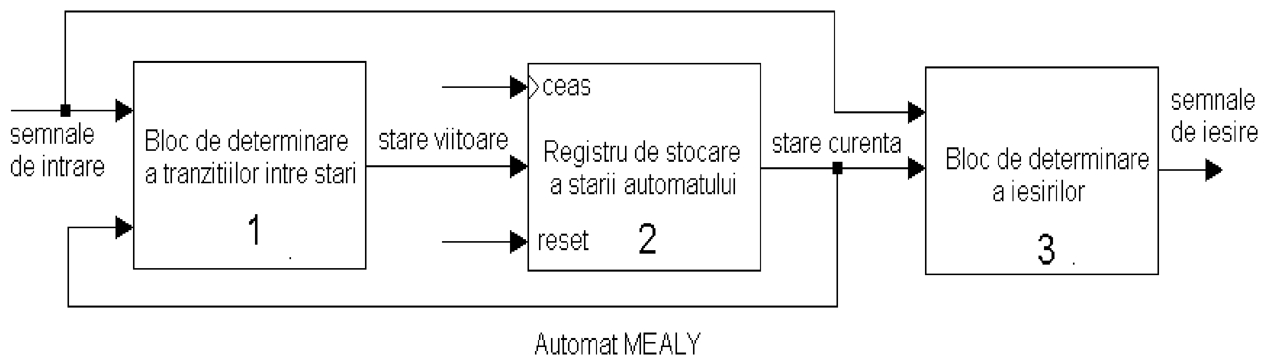
În domeniul circuitelor digitale, în majoritatea cazurilor, proiectanții trebuie să proiecteze circuite care efectuează anumite secvențe specifice de operații, de exemplu, controlere utilizate pentru comanda funcționării altor circuite. Automatele de stare reprezintă o metodă foarte eficientă pentru modelarea circuitelor secvențiale. Prin modelarea automatelor de stare într-un limbaj de descriere hardware şi utilizarea programelor de sinteză proiectanții se pot concentra doar asupra modelării secvenței dorite de operații fără a-şi pune probleme în privința implementării circuitului. Această operație este lăsată programelor de sinteză.

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

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

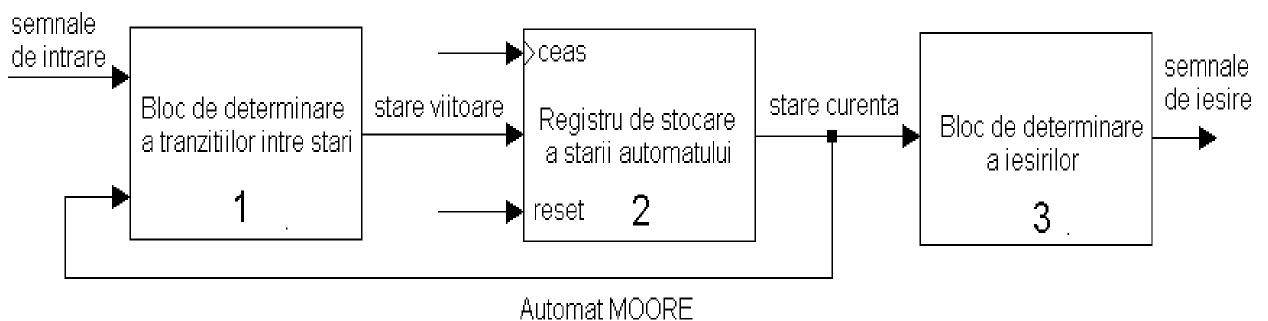
După tipul funcției pentru stabilirea valorilor de la ieşire, automatele de stare se împart în două categorii: *Mealy* şi *Moore*.

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



Automat MEALY

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



Automat MOORE

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

Implementarea in VHDL

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

Particularită□i:

1. Tipul stărilor este definit de utilizator. De obicei se utilizează tipul enumerare.
   TYPE *state_type* **IS** (idle, tap1, tap2, tap3, tap4);
2. Se folosesc semnale sau variabile interne în arhitectură pentru memorarea stărilor.
   SIGNAL filter : *state_type*;
3. Pentru a determina tranzi□ia în starea următoare se folose□te insruc□iunea **case.**
4. Pentru a determina semnalele de ie□ire se folose□te fie un **proces** combina□ional cu instruc□iunea **case**, fie **asignarea condi□ională** sau **selectivă** de semnal.

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

1. Un singur proces
   - conține atât tranziția stărilor cât şi funcțiile de ieşire;
2. Două procese
   - proces combinațional care conține logica stării următoare şi logica de ieşire;
   - proces secvențial pentru actualizarea stării prezente, sincron cu semnalul de ceas;
3. Trei procese
   - proces combinațional care conține logica stării următoare ;
   - proces secvențial pentru actualizarea stării prezente, sincron cu semnalul de ceas;
   - proces combinațional care conține logica de ieşire;

Inițializare sincronă şi asincronă

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

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

```
state_machine  :  process (clk)
 begin
   if (clk'EVENT AND clk = '1') then
    if (reset = '0') then
```

```
      state_vector <= idle;
     else
   -- condițiile de tranziție a stărilor
  end if;
 end if;
end process state_machine;
```

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

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

```
state_machine : process (clk, reset)
 begin
  if (reset = '0') then               -- resetare asincronă pe `0`
   state_vector <= idle;              -- trecere în starea inițială
  elsif (clk'EVENT AND clk = '1') then -- sincron cu frontul pozitiv
  -- condițiile de tranziție a stărilor
  end if;
end process state_machine;
```

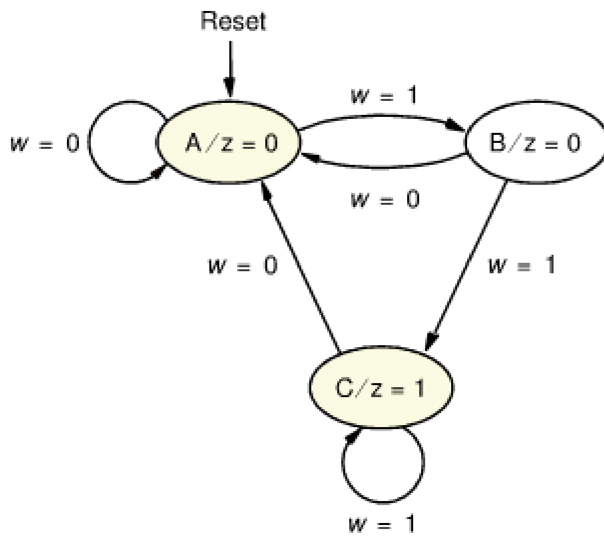
Semnalul reset are prioritate față de semnalul clk, deoarece instrucțiunile se evaluează în ordine succesivă, deci condiția de resetare este analizată prima.

Reprezentarea automatelor finite

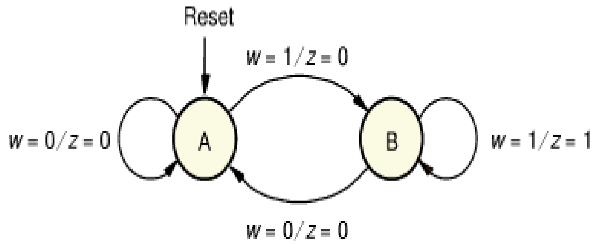
Automatele finite pot fi reprezentate prin digrame de stare (sisteme simple) sau scheme bloc (sisteme complexe). Ambele reprezentări utilizează nota□ii simbolice care arată tranzi□ia dintre stări □i valorile de la ie□ire în dependen□ă de anumite condi□ii.

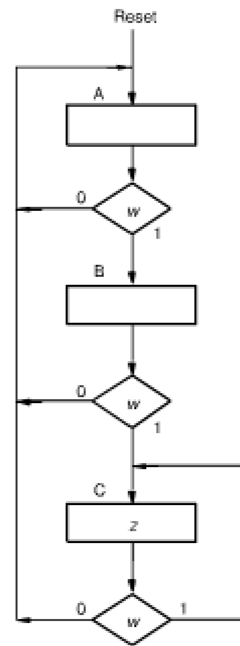
Exemplu de diagramă de stare pentru automatul Moore:

Exemplu de schemă bloc (Automatul Moore):

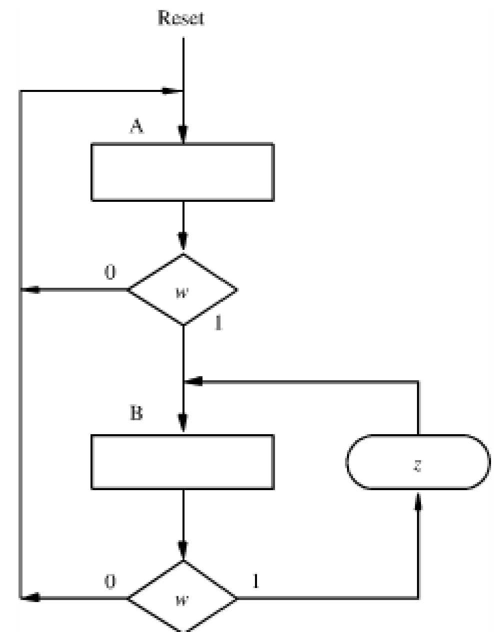


Pentru automatul Mealy:





Pentru automatul Mealy:

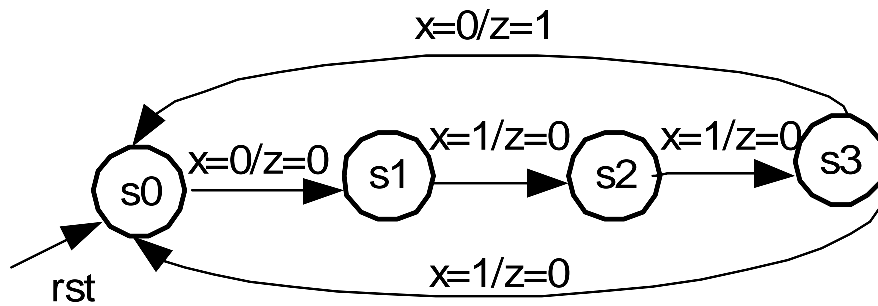


Exemple de automate.

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

Automatul Mealy.

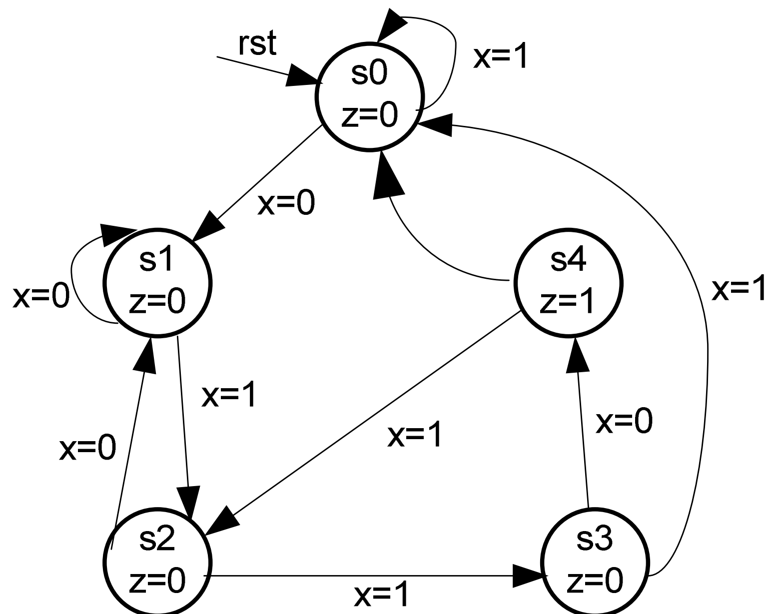
Diagrama de stare:



Codul VHDL pentru automatul Mealy descris cu 2 procese.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity mealy2 is
Port (CLK, rst,x: in std_logic;
            Z: out std_logic);
End entity mealy2;
Architecture Mealy2_arch of mealy2 is
Type State is (s0, s1, s2, s3);
Signal cs, ns: state;
Begin
Secv: process (CLK, rst) is
begin
If (rising_edge(clk)) then
If (rst='1') then cs<=s0;  else cs<=ns;
end if;
end if;
end process secv;
com: process (cs, x) is
begin
z<='0';
case cs is
when s0 =>  if (x='0')  then ns<=s1; else ns<=s0; end if;
when s1=>  if (x='0')  then ns<=s1; else ns<=s2; end if;
when s2 =>  if (x='0')  then ns<=s1; else ns<=s3; end if;
when s3 =>  if (x='0')  then ns<=s1; z<='1'; else ns<=s0; end if;
end case;
end process com;
end architecture Mealy2_arch;
```

Diagrama de stare pentru automatul Moore



Codul VHDL pentru automatul Moore descris cu 2 procese.

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

```
end case;
end process com;
end architecture Moore2_arch;
```

Codul VHDL pentru automatul Moore descris cu 3 procese.

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

Definirea şi codificarea stărilor

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

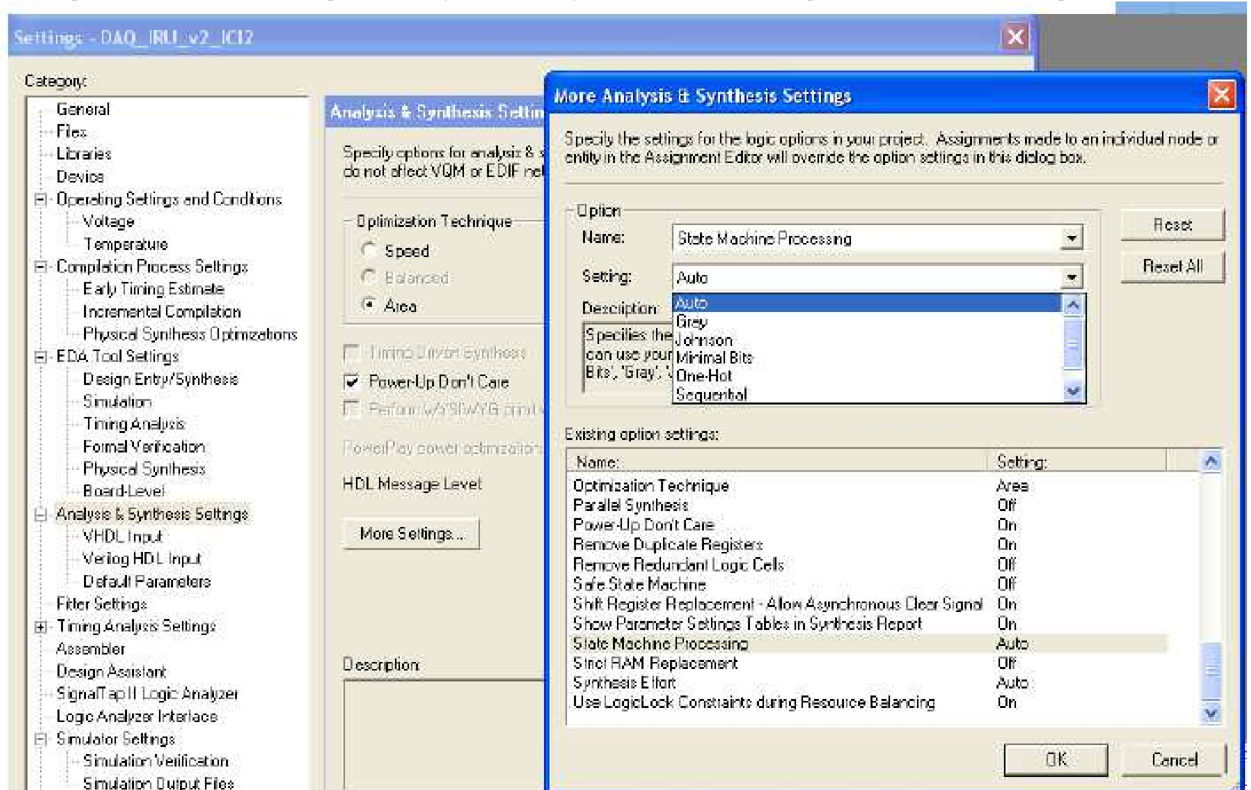
1. Codificare binară consecutivă.
2. Codificarea Gray.
3. Codificarea cu un bistabil pe stare.
4. Codificarea definită de utilizator.

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

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

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

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



1. Codificare binară consecutivă (Sequential).

Avantajul: număr minim de bistabile ($\log_2 n$);
Dezavantajul: Func□ii complexe de tranzi□ie a stărilor; logică complicată de decodare a stării curente; mai multe bistabile î□i pot schimba starea în acela□i timp.

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

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

2. Codul Gray.

Avantajul: Este cea mai sigură implementare, deoarece tranziția de la o stare la alta va fi determinată de schimbarea valorii unui singur bistabil, deci este exclusă apariția unor stări intermediare. Este important în special în cazul automatelor asincrone.

Dezavantaje: Func□ii complexe de tranzi□ie a stărilor; logică complicată de decodare a stării curente;

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

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

Tehnica de codificare cu un bistabil pe stare utilizează *n* bistabile pentru a reprezenta un automat cu *n* stări. Pentru fiecare stare există câte un bistabil, un singur bistabil fiind setat la un moment dat. Decodificarea stării prezente constă în simpla identificare a bistabilului care este setat. Tranziția dintr-o stare în alta constă în modificarea stării bistabilului corespunzător stării vechi din 1 în 0 şi a stării bistabilului corespunzător stării noi din 0 în 1.

Avantajul principal al automatelor care utilizează codificarea cu un bistabil pe stare este că numărul de porți necesare pentru decodificarea informației de stare pentru ieşiri şi pentru tranzițiile stărilor este cu mult mai redus decât numărul de porți necesare în cazul utilizării altor tehnici. Această diferență de complexitate creşte pe măsură ce numărul de stări devine mai mare.

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

În funcție de arhitectura circuitului utilizat pentru implementare, un automat de stare care utilizează codificarea cu un bistabil pe stare poate necesita o cantitate semnificativ mai redusă de resurse pentru implementare decât un automat care

utilizează alte metode de codificare. De asemenea, logica stării următoare necesită, de obicei, un număr mai redus de nivele logice între registrele de stare, ceea ce permite o frecvență mai ridicată de funcționare. Codificarea cu un bistabil pe stare nu reprezintă însă soluția optimă în toate cazurile, în principal datorită faptului că necesită un număr mai mare de bistabile decât codificarea secvențială. În general, codificarea cu un bistabil pe stare este avantajoasă atunci când arhitectura circuitului programabil utilizat conține un număr relativ mare de bistabile şi un număr relativ redus de porți logice între bistabile. De exemplu, această codificare este cea mai avantajoasă pentru automatele de stare implementate cu circuite FPGA, care conțin un număr mai mare de bistabile decât circuitele CPLD.

Toleranța la defecte a automatelor de stare

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

Automatul poate rămâne definitiv în această stare ilegală, sau poate activa o combinație ilegală a ieşirilor, ceea ce poate cauza alte efecte nedorite. Automatele de stare pot fi proiectate astfel încât să fie tolerante la defecte prin adăugarea unei logici care să asigure ieşirea din stările ilegale.

Posibilită□i de proiectare a automatelor de stare sigure.

1. Utilizarea construc□iei **when others.**

when others => stare <= idle;

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

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

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

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

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

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

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

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

```
 type stare is (s0, s1, s2, nedefinit);
signal cs, ns: stare;
…
case   cs is
```

```
   …
  when nedefinit => ns<=s0;
end case;

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