Apontamentos Teóricos da Disciplina de

SISTEMAS DIGITAIS 1

Licenciatura em Engenharia de Sistemas e Informática

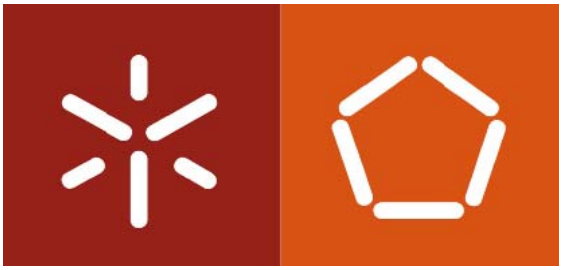
2º Ano -- 2º Semestre

António Joaquim Esteves
João Miguel Fernandes

Departamento de Informática

ESCOLA DE ENGENHARIA

UNIVERSIDADE DO MINHO



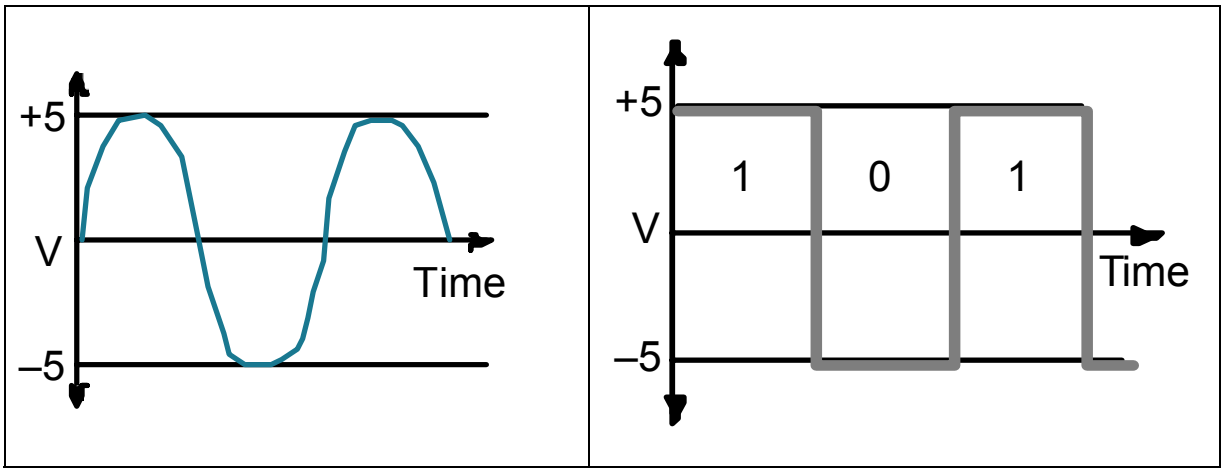
Março 2006

ÍNDICE

1. Introdução

1.1. Sistemas digitais vs. analógicos

Um sistema é um conjunto de partes relacionadas que funcionam como um todo para atingir um determinado objectivo. Um sistema possui entradas e saídas e apresenta um comportamento definido à custa de funções que convertem as entradas em saídas. Um **sistema analógico** processa sinais que variam ao longo do tempo e que podem assumir qualquer valor dum intervalo contínuo de tensão, corrente, pressão, … O mesmo se aplica ao **sistema digital**: a diferença está em nós desejarmos que isso não aconteça.

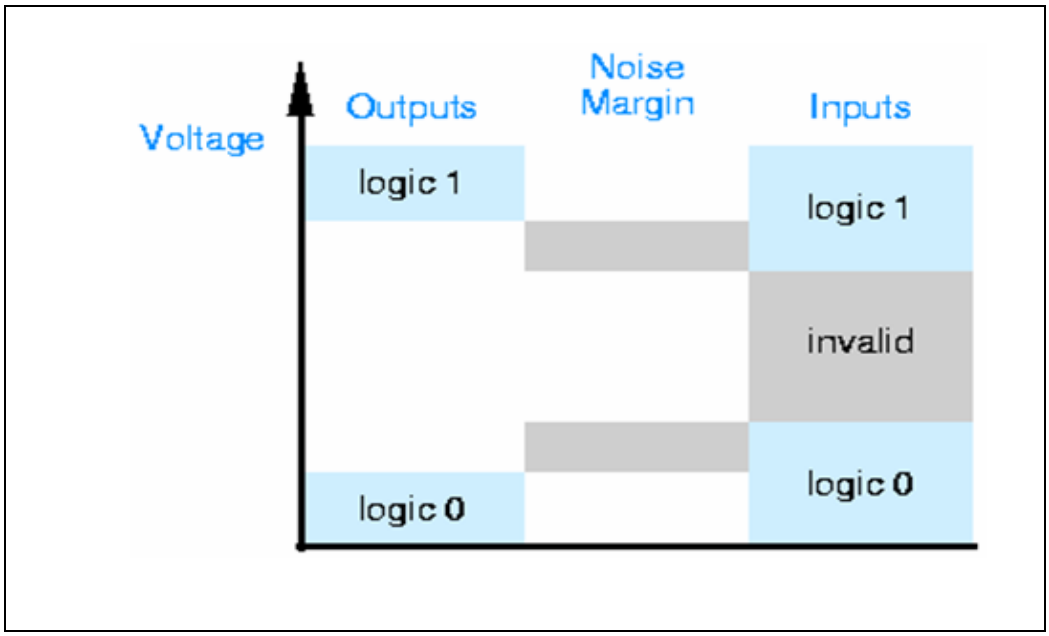


A vantagem mais importante dos sistemas digitais é a sua capacidade para operarem com sinais eléctricos que tenham sido degradados. Pelo facto de as saídas serem discretas, uma ligeira variação numa entrada continua a ser interpretada correctamente. Nos circuitos analógicos, um ligeiro erro na entrada provoca um erro na saída. O sistema binário é a forma mais simples de sistema digital. Um sinal binário é modelado de forma a que ele apenas assuma dois valores discretos: 0 ou 1, Baixo/*LOW* ou Alto/*HIGH*, Falso ou Verdadeiro.

1.2. Abstracção digital

Os circuitos digitais operam sobre tensões e correntes analógicas. A **abstracção digital** consiste em ignorar comportamento analógico na maior parte das situações, permitindo deste modo que os circuitos sejam modelados como se eles processassem apenas 0s e 1s.

Associação entre um intervalo de valores analógicos e cada um dos valores lógicos (0 e 1). À diferença entre os limites desses intervalos chama-se margem de ruído.

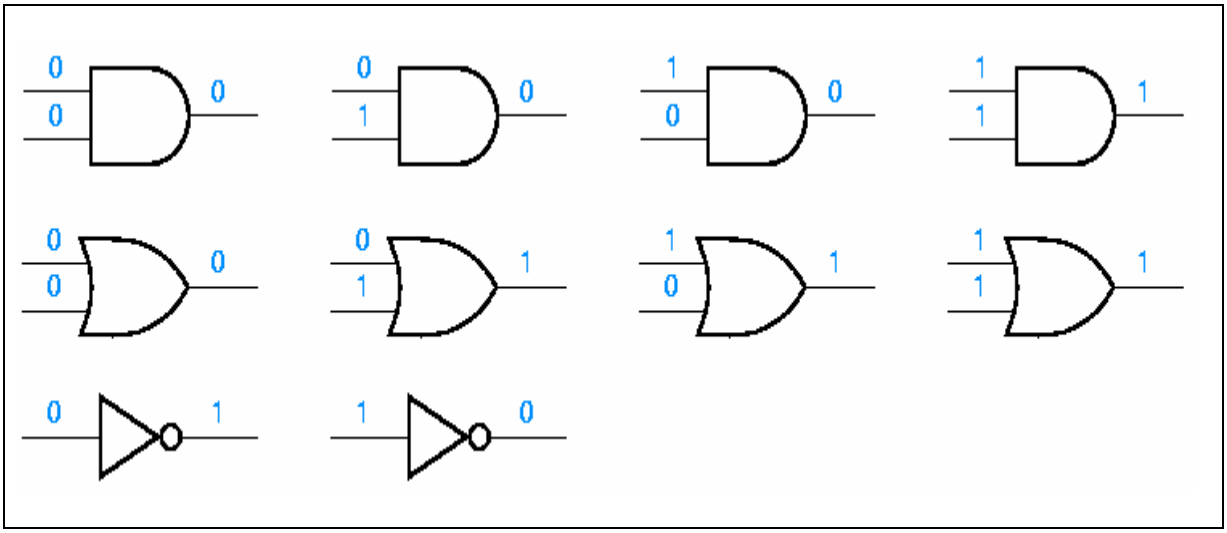


1.3. Sistemas síncronos vs. assíncronos

Um **sistema síncrono** é aquele em que os elementos mudam o seu valor em determinados instantes específicos. Um **sistema assíncrono** possui saídas que podem mudar de valor em qualquer instante. Por exemplo, considere-se um relógio digital com alarme, programado para tocar às 13:59. Num sistema síncrono, as saídas (HH, mm, …) mudam todas ao mesmo tempo: 12:59 → 13:00 → 13:01 → ... Num sistema assíncrono, as saídas não têm forçosamente que mudar em simultâneo: 12:59 → 13:59 → 13:00 → ...

1.4. Portas lógicas (gates)

As portas lógicas são o dispositivo digital mais elementar. Uma **porta lógica** possui uma ou mais entradas e gera uma saída que é uma função dos valores actuais das entradas. Uma porta lógica é um **circuito combinacional** porque as saídas dependem exclusivamente da combinação actual das entradas.



1.5. Flip-flops

Um **flip-flop** é um dispositivo que guarda um 0 ou um 1 na saída. O estado do flip-flop é o valor que ele guarda no presente instante. O valor guardado só pode ser alterado em determinados instantes, impostos por uma entrada de relógio (*clock*). Um circuito digital que inclui flip-flops é um **circuito sequencial**. A saída dum circuito sequencial depende, em qualquer instante, além do valor actual das entradas, da sequência de valores que no passado foi aplicada nas entradas. Um circuito sequencial possui memória dos eventos passados.

1.6. Ferramentas de CAD

O projecto de sistemas digitais não tem que recorrer obrigatoriamente a ferramentas de *software*. Contudo, as ferramentas de *software* são essenciais ao projecto de sistemas digitais. A utilização de HDLs (***Hardware Description Languages***), e das correspondentes ferramentas de simulação e síntese, está generalizada. Num ambiente CAD (*Computer-Aided Design*), as ferramentas melhoram a produtividade, ajudam a corrigir erros e a antever o comportamento. Algumas das tarefas que podem ser realizadas com a ajuda das ferramentas CAD são:

- Edição de esquemáticos;
- Compilação, simulação e síntese com HDLs;
- Análise da evolução dos sistemas ao longo do tempo;
- Simulação;
- Geração de vectores de teste.

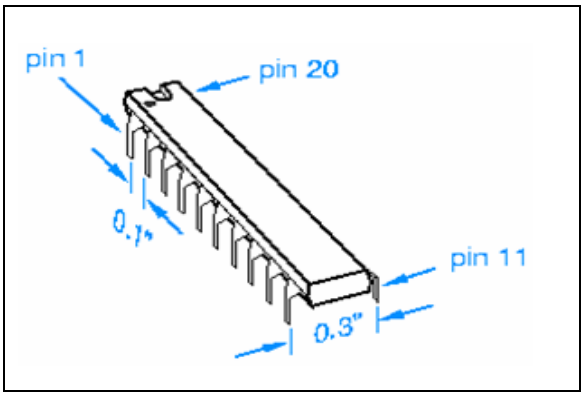
1.7. Circuitos integrados

Um **circuito integrado** (CI) é uma colecção de portas lógicas produzidas num único *chip*. Os CIs podem ser classificados, de acordo com o seu tamanho, em:

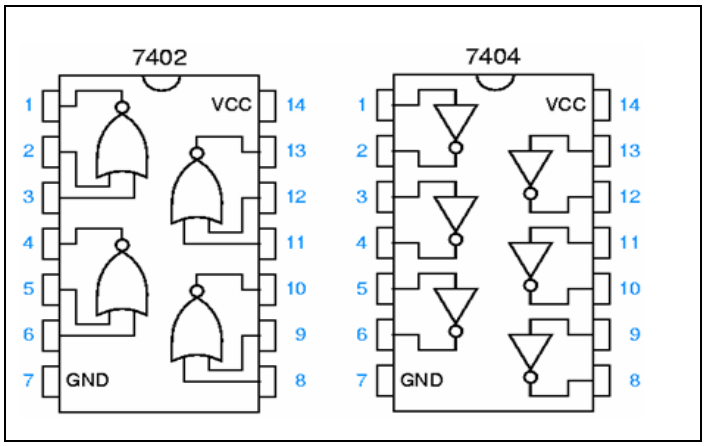
- **SSI** (*small scale integration*): de 1 a 20 portas lógicas - ANDs, Ors e NOTs;
- **MSI** (*medium scale integration*): 20 a 200 portas lógicas - descodificadores, registos e contadores;
- **LSI** (*large scale integration*): 200 a 200.000 portas lógicas - memórias de capacidade reduzida e PLDs simples;
- **VLSI** (*very large scale integration*): mais de 1 milhão de transístores - microprocessadores, memórias e PLDs complexas.

Como exemplo, pode dizer-se que o Pentium4 tem 42 milhões de transístores.

A figura seguinte mostra um exemplo de encapsulamento do tipo DIP (***D****ual* ***I****n-line* ***P****in package*).



O diagrama de pinos da figura seguinte mostra a correspondência entre os sinais do dispositivo e os pinos do encapsulamento. Actualmente, os CIs do tipo SSI são usados como "cola" para formar componentes maiores em sistemas complexos. Os CIs do tipo SSI têm vindo a ser substituídos de forma generalizada por dispositivos de lógica programável (PLDs).



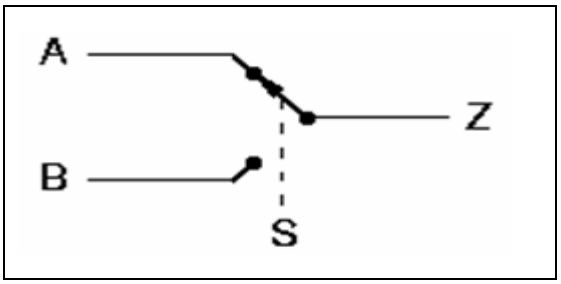
1.8. PLDs

Alguns CIs permitem que a sua funcionalidade lógica seja programada no próprio *chip* após terem sido fabricados. A maior parte destes CIs pode mesmo ser **re**programada, o que permite que alguns erros sejam corrigidos sem ter que o substituir ou retirar. As **PLDs:** possuem uma estrutura a dois-níveis, com portas AND e OR, e ligações programáveis pelo utilizador. As **CPLDs** (PLDs complexas) e as **FPGAs** (*Field Programmable Gate Arrays*) foram idealizadas com o intuito de implementarem sistemas de maior dimensão. A utilização de **HDLs** e das respectivas **ferramentas** permite que um projecto seja compilado, sintetizado e descarregado para o dispositivo em menos tempo.

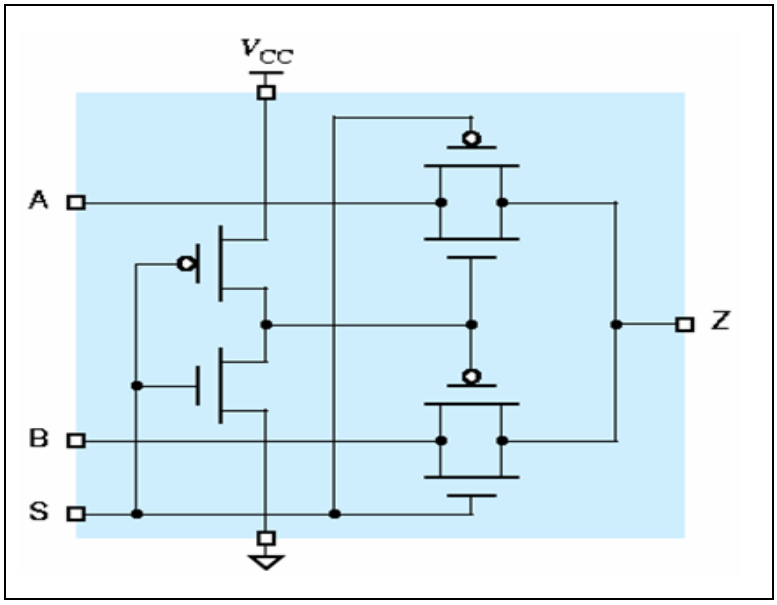
1.9. Níveis de abstracção no projecto de sistemas digitais

O projecto de sistemas digitais pode decorrer em vários níveis de representação e de abstracção. Embora se possa ganhar experiência a projectar em determinado nível, por vezes é preciso mudar (subir e descer) de nível para concluir com sucesso certos projectos. O nível mais baixo é o da física do dispositivo e dos processos de fabrico do CI. Este nível não é leccionado em SD1. O nível seguinte é o do transístor. Este nível também não é leccionado em SD1.

Para explicar o nível do transístor e os níveis seguintes, vamos usar um multiplexador com 2 bits de entrada (A e B), 1 bit de controlo (S) e 1 bit de saída (Z).

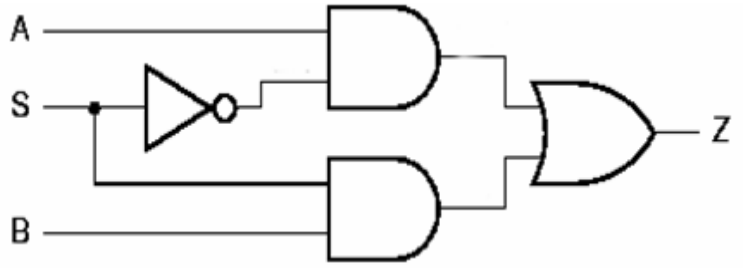


Para optimizar certas funções (ou módulos funcionais) é necessário projectá-las no nível do transístor. O multiplexador pode ser projectado em tecnologia CMOS, usando blocos estruturais à base de transístores. Utilizando esta abordagem, o multiplexador pode ser construído com 6 transístores apenas.

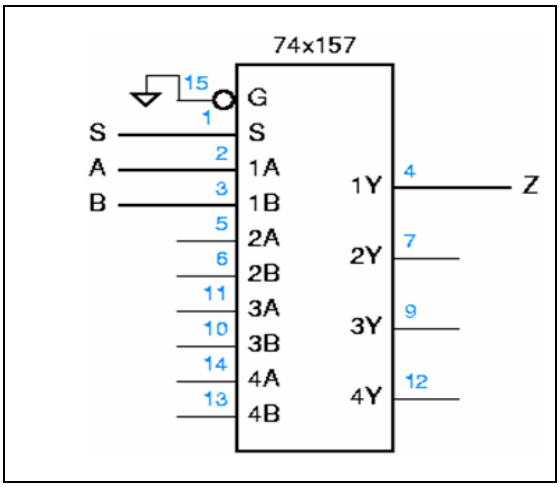


Segundo a forma tradicional de projectar um sistema, utiliza-se uma tabela de verdade para descrever a funcionalidade do multiplexador. Uma **tabela de verdade** contém todas as possíveis combinações dos valores de entradas e dos correspondentes valores das saídas. A partir da tabela de verdade, pode obter-se uma expressão minimizada para a saída do multiplexador: **Z = S'.A + S.B**
Esta expressão pode ser convertida num diagrama no nível da porta lógica.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Para as funções mais frequentes, a maioria das tecnologias digitais dispõe de blocos elementares predefinidos. O 74x157 da figura seguinte é um CI do tipo MSI que faz a multiplexagem de 2 entradas de 4-bits. A figura mostra o diagrama do CI 74x157, no nível do bloco. Os números em azul identificam os pinos num encapsulamento DIP de 16-pinos que contém o dispositivo.



Também se pode usar HDLs, como o VHDL, para descrever a funcionalidade do multiplexador no nível algorítmico (ver figura seguinte). A declaração ***entity*** especifica quais as entradas e saídas do circuito. A declaração **architecture** define o comportamento do multiplexador. Uma ferramenta de síntese pode processar esta descrição algorítmica e gerar um circuito numa determinada tecnologia.

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

entity Vchap1mux is
    port ( A, B, S: in  STD_LOGIC;
           Z:       out STD_LOGIC );
end Vchap1mux;

architecture Vchap1mux_arch of Vchap1mux is
begin
  Z <= A when S = '0' else B;
end Vchap1mux_arch;
```

2. Álgebra de Boole

O sucesso da tecnologia dos computadores baseia-se em 1º lugar na simplicidade com que se projectam circuitos digitais e na facilidade da sua produção. Os circuitos digitais são constituídos por unidades de processamento elementares, designadas por **portas lógicas**, e unidades de memória elementares, designadas por ***flip-flops***. A simplicidade do projecto de circuitos digitais deve-se ao facto de as entradas e as saídas de cada porta lógica ou flip-flop assumir apenas 2 valores: 0 e 1. As alterações no valor dos sinais são determinadas pelas leis da **álgebra de Boole**.
A álgebra de Boole permite **optimizar** funções. No projecto de circuitos digitais pode usar-se **técnicas de optimização de outras áreas**.

2.1. Sinais binários

A lógica digital esconde a realidade analógica, ao mapear uma gama infinita de valores reais em apenas 2 valores: 0 e 1. A um valor lógico, 0 ou 1, é comum chamar-se um dígito binário (bit). Com n bits, pode representar-se 2n entidades distintas. Quando um projectista lida com circuitos electrónicos, é comum usar os termos “BAIXO” e “ALTO”, em vez de “0” e “1”. Considerar que 0 é BAIXO e 1 é ALTO, corresponde a usar lógica positiva. A correspondência oposta a esta é designada de lógica negativa.

	State Representing Bit	
Technology	0	1
Pneumatic logic	Fluid at low pressure	Fluid at high pressure
Relay logic	Circuit open	Circuit closed
Complementary metal-oxide semiconductor (CMOS) logic	0–1.5 V	3.5–5.0 V
Transistor-transistor logic (TTL)	0–0.8 V	2.0–5.0 V
Fiber optics	Light off	Light on
Dynamic memory	Capacitor discharged	Capacitor charged
Nonvolatile, erasable memory	Electrons trapped	Electrons released
Bipolar read-only memory	Fuse blown	Fuse intact
Bubble memory	No magnetic bubble	Bubble present
Magnetic tape or disk	Flux direction "north"	Flux direction "south"
Polymer memory	Molecule in state A	Molecule in state B
Read-only compact disc	No pit	Pit
Rewriteable compact disc	Dye in crystalline state	Dye in noncrystalline state

2.2. *Sistemas combinacionais vs. sequenciais*

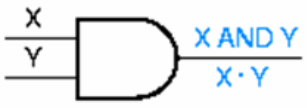
Um sistema lógico **combinacional** é aquele em que as saídas dependem apenas do valor actual das entradas. Um sistema combinacional pode ser descrito por uma **tabela de verdade**.

Além do valor actual das entradas, as saídas dum circuito lógico **sequencial** dependem também da sequência de valores por que passaram as entradas ⇒ memória. Um sistema sequencial pode ser descrito através duma **tabela de estados**.

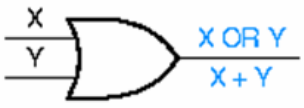
Um sistema combinacional pode conter qualquer número de portas lógicas mas não ciclos de realimentação (*feedback loops*). Um ciclo de realimentação é um caminho dum circuito, que permite a um sinal de saída duma porta ser propagado de volta para a entrada dessa porta.
Regra geral, os ciclos de realimentação introduzem um comportamento sequencial nos circuitos.

2.3. *Portas lógicas*

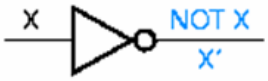
Com 3 tipos de porta elementares (AND, OR, NOT) consegue construir-se qualquer sistema digital combinacional, ou seja, formam um **conjunto completo**.



X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

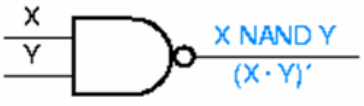


X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

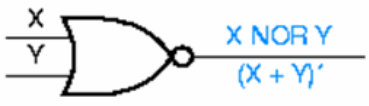


X	NOT X
0	1
1	0

Os símbolos e as tabelas de verdade do AND e do OR podem ser generalizados para portas com qualquer número de entradas. A bolha na saída do inversor representa um comportamento "invertido". Combinando numa única porta, um NOT com uma função AND ou OR, obtêm-se 2 novas funções lógicas: NAND e NOR.



X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



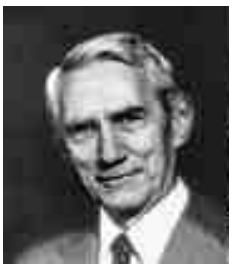
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

O símbolo e a tabela de verdade do NAND e do NOR também podem ser generalizados para portas com qualquer número de entradas.

2.4. Álgebra da comutação (switching)

Em 1854, G. Boole [1815-1865] introduziu o formalismo que ainda usamos para tratar a lógica de forma sistemática, a **álgebra de Boole**.
Em 1938, C. Shannon [1916-2001] utilizou esta álgebra para provar que as propriedades dos circuitos de comutação eléctricos podem ser representados por uma álgebra de Boole com 2-valores, a **álgebra da comutação**.


G. Boole


C. Shannon

Usando esta álgebra, pode equacionar-se proposicões (afirmações) que serão verdadeiras ou falsas. Combinando-as, geram-se novas proposições e pode concluir-se se elas são verdadeiras ou falsas. Shannon usava uma variável simbólica (por ex. X) para representar a condição associada a um sinal lógico, em que ele assumia um de 2 valores possíveis ("0" ou "1").

2.5. Axiomas

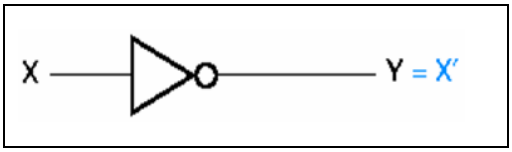
Os **axiomas** *(ou postulados)* dum sistema matemático são um conjunto mínimo de definições elementares, que se considera serem verdadeiras. O 1º par de axiomas incorpora a **abstracção digital** *(X só pode assumir 2 valores)*:

(A1) X=0 se X≠1 (A1') X=1 se X≠0

Este par de axiomas apenas difere na permuta dos símbolos 0 e 1. Este princípio aplica-se a todos os axiomas e está na origem da **dualidade**. O próximo par de axiomas incorpora a notação de função **inversor**:

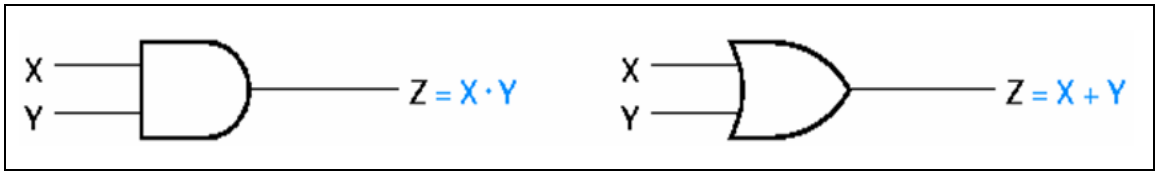
(A2) Se X=0, então X'=1 (A2') Se X=1, então X'=0

em que a plica (') denota a função inversor.



Os últimos 3 pares de axiomas enunciam a definição formal das operações **AND** *(multiplicação lógica)* e **OR** *(adição lógica)*:

(A3) 0·0 = 0 (A3') 1+1 = 1
(A4) 1·1 = 1 (A4') 0+0 = 0
(A5) 0·1 = 1·0 = 0 (A5') 1+0 = 0+1 = 1



Por convenção, numa expressão lógica contendo multiplicação e adição, a multiplicação tem **precedência**. A expressão X·Y+Y·Z' é equivalente a (X·Y)+(Y·Z'). Os axiomas A1-A5 e A1'-A5' definem de forma completa a **álgebra Boole**.

2.6. Teoremas

Os **teoremas** são declarações, que se sabe serem verdadeiras, que permitem manipular expressões algébricas de modo a que a análise seja mais simples e/ou a síntese dos circuitos correspondentes seja mais eficiente.

Apresentam-se a seguir os teoremas envolvendo apenas **uma** variável:

(T1)	X+0 = X	**(T1')**	X·1 = X	(Identidades)
(T2)	X+1 = 1	**(T2')**	X·0 = 0	(Elementos nulos)
(T3)	X+X = X	**(T3')**	X·X = X	(Idempotência)
(T4)	(X')' = X			(Involução)
(T5)	X+X' = 1	**(T5')**	X·X' = 0	(Complementos)

Pode provar-se que estes teoremas são verdadeiros. Segue-se a prova de T1:
[X=0] 0+0=0 (verdade, segundo A4')
[X=1] 1+0=1 (verdade, segundo A5')

Os teoremas envolvendo **2 ou 3** variáveis são:

(T6)	X+Y = Y+X	**(T6')**	X·Y = Y·X	(Comutatividade)
(T7)	(X+Y)+Z = X+(Y+Z)	**(T7')**	(X·Y)·Z = X·(Y·Z)	(Associatividade)
(T8)	X·Y+X·Z = X·(Y+Z)	**(T8')**	(X+Y)·(X+Z)=X+Y·Z	(Distributividade)
(T9)	X+X·Y = X	**(T9')**	X·(X+Y) = X	(Cobertura)
(T10)	X·Y+X·Y' = X	**(T10')**	(X+Y)·(X+Y') = X	(Combinação)
(T11)	X·Y+X'·Z+Y·Z = X·Y+X'·Z			(Consenso)
(T11')	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)			

Convém ter em tenção que o teorema T8' não é verdadeiro com inteiros ou reais. Os teoremas T9 e T10 são usados para **minimizar** funções lógicas.

Vários teoremas importantes são verdadeiros para um nº arbitrário de variáveis. Daqui resultam teoremas envolvendo ***n*** variáveis:

Teoremas da idempotência generalizada:

(T12)	X+X+ ... +X = X
(T12')	X·X· ... ·X = X

Teoremas de DeMorgan:

(T13) $(X1 \cdot X2 \cdot \ldots \cdot Xn)' = X1' + X2' + \ldots + Xn'$
(T13') $(X1 + X2 + \ldots + Xn)' = X1' \cdot X2' \cdot \ldots \cdot Xn'$

Teorema de DeMorgan generalizado:

(T14) $[F(X1,X2,\ldots,Xn,0,1,+,\cdot)]' = F(X1',X2',\ldots,Xn',1,0,\cdot,+)$

Teoremas da expansão de Shannon:

(T15) $F(X1,X2,\ldots,Xn) = X1 \cdot F(1,X2,\ldots,Xn) + X1' \cdot F(0,X2,\ldots,Xn)$
(T15') $F(X1,X2,\ldots,Xn) = [X1 + F(0,X2,\ldots,Xn)] \cdot [X1' + F(1,X2,\ldots,Xn)]$

Os teorema de DeMorgan (T13 e T13') para 2 varáveis (n=2) ficam:

$(X \cdot Y)' = X' + Y'$
$(X + Y)' = X' \cdot Y'$

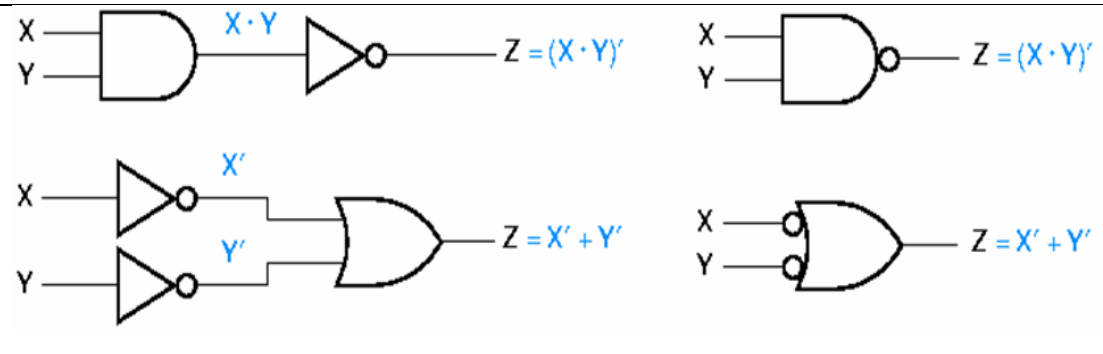
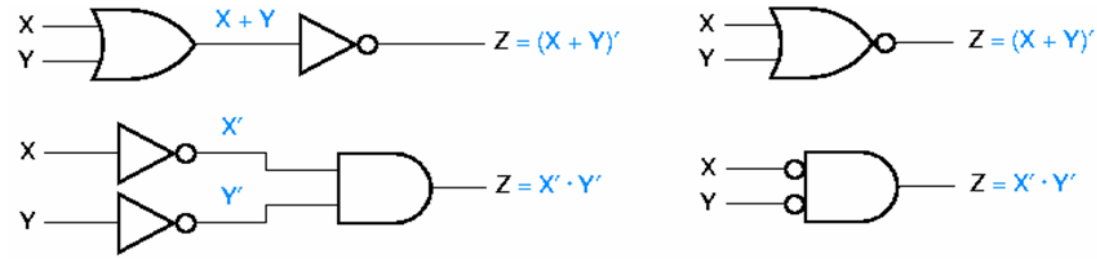
Augustus De Morgan [1806-1871]

O Teorema de DeMorgan estabelece um procedimento para **complementar funções** lógicas. Pode usar-se o teorema de DeMorgan para converter expressões AND-OR em expressões OR-AND, como se mostra no próximo exemplo.

Exemplo:

$Z = A'\,B'\,C + A'\,B\,C + A\,B'\,C + A\,B\,C'$ *(expressão AND-OR)*
$Z' = (A + B + C') \cdot (A + B' + C') \cdot (A' + B + C') \cdot (A' + B' + C)$ *(expressão OR-AND)*

Também se podem usar os teoremas de DeMorgan para saber a equivalência entre portas lógicas:

T13	
T13’	

Como a álgebra de Boole só possui 2 valores, também se pode demonstrar a validade dos teoremas através de **tabelas de verdade**. Para isso, constrói-se uma tabela de verdade para cada lado das equações presentes num teorema. O exemplo seguinte prova a veracidade dos teoremas de DeMorgan [T13 e T13’] para n=2:

(X+Y)’ = X’·Y’

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X+Y}$	$\overline{X}\cdot\overline{Y}$
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

(X·Y)’ = X’+Y’

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X\cdot Y}$	$\overline{X}+\overline{Y}$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

2.7. Dualidade

Os teoremas anteriores foram apresentados aos pares. A versão primária dum teorema pode ser obtida da versão secundária trocando “0” com “1” e “·” com “+”.

> *<u>Princípio da dualidade</u>: qualquer teorema ou identidade da Álgebra de Boole continua a ser verdadeiro quando se trocam todos os “0” com “1” e todos os “·” com “+”.*

A dualidade é importante porque duplica a utilidade de qualquer axioma/teorema da Álgebra de Boole e da manipulação de funções lógicas. Verifica-se que o dual duma expressão lógica é a mesma expressão em que “+” e “·” foram trocados:

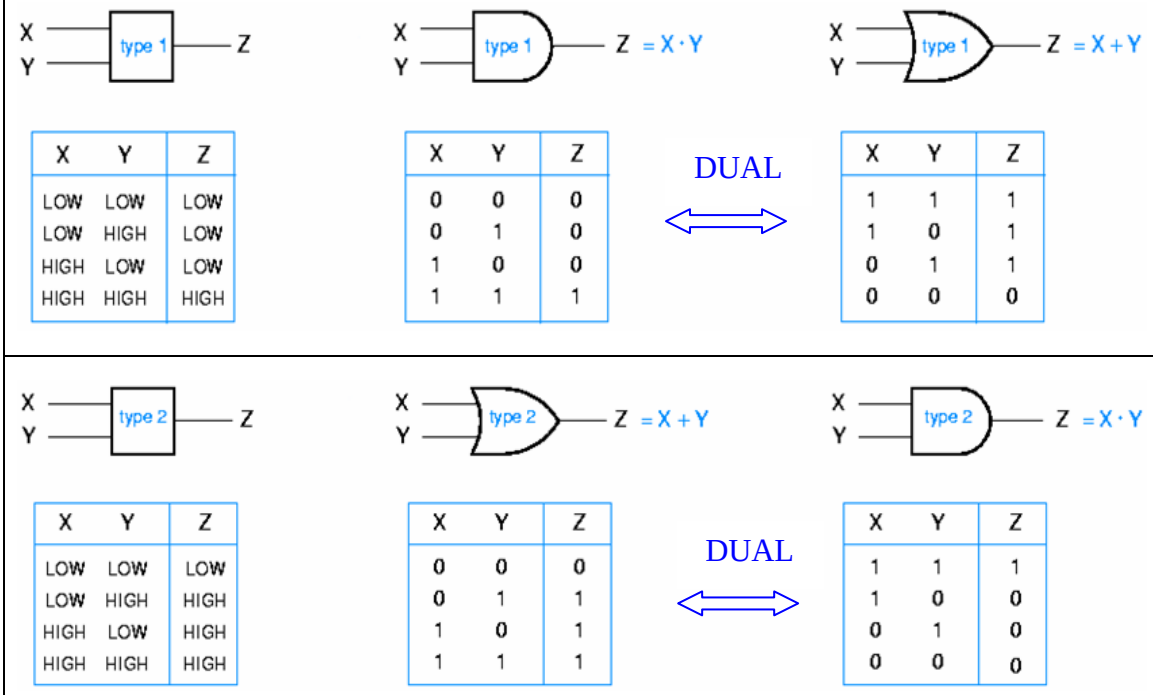
$F^D(X_1,X_2,...,X_n,0,1,+,\cdot,') = F(X_1,X_2,...,X_n,1,0,\cdot,+,')$.

Não se deve confundir a dualidade com os teoremas de DeMorgan:

$$[F(X_1,X_2,...,X_n,+,\cdot)]' = F(X_1',X_2',...,X_n',\cdot,+)$$
$$[F(X_1,X_2,...,X_n,+,\cdot)]' = F^D(X_1',X_2',...,X_n',+,\cdot)$$

A figura seguinte ilustra a aplicação prática da expressão do dual.

Função eléctrica	Lógica positiva	Lógica negativa



X	Y	Z
LOW	LOW	LOW
LOW	HIGH	LOW
HIGH	LOW	LOW
HIGH	HIGH	HIGH

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

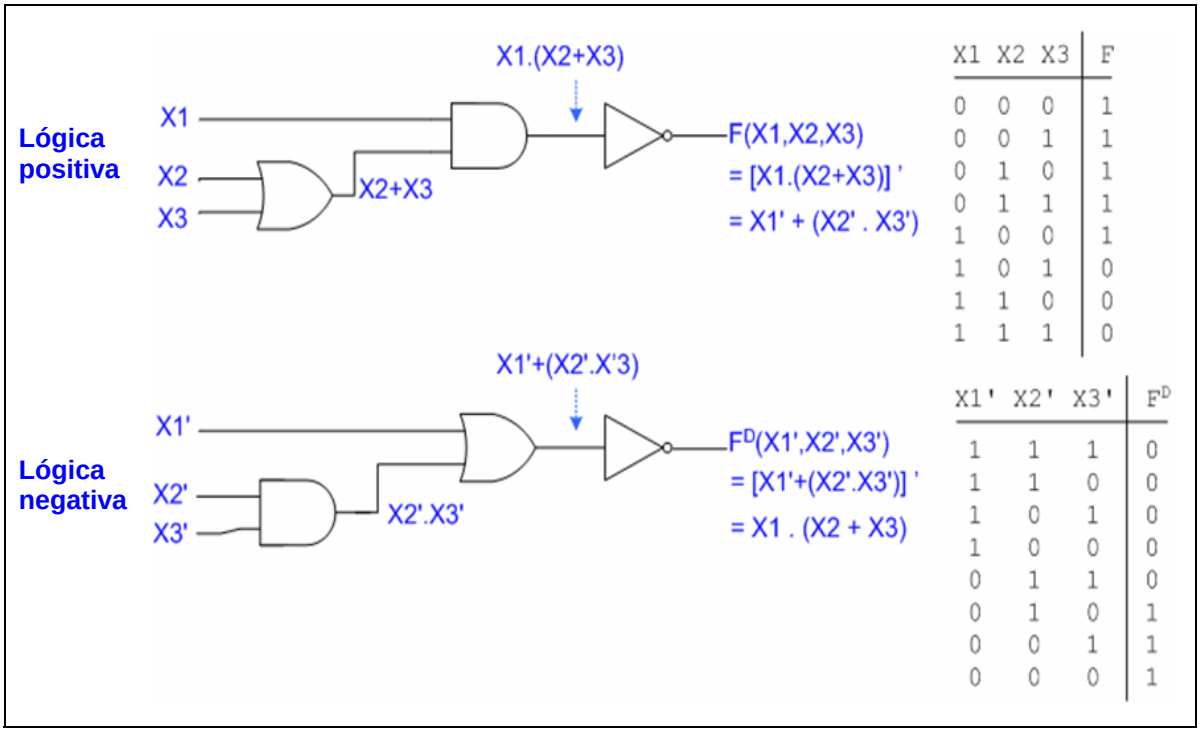
X	Y	Z
1	1	1
1	0	1
0	1	1
0	0	0

X	Y	Z
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	HIGH

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	Z
1	1	1
1	0	0
0	1	0
0	0	0

A figura seguinte é uma ilustração prática de $[F(X1,X2,...,Xn)]' = F^D(X1',X2',...,Xn')$.



X1	X2	X3	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

X1'	X2'	X3'	F^D
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	1

2.8. Representação normalizada

A representação mais elementar duma função lógica é a tabela de verdade. A **tabela de verdade** indica qual é a saída do circuito para cada combinação de entradas possível. A tabela de verdade duma função de **n**-variáveis possui $\mathbf{2^n}$ linhas.

Row	X	Y	Z	F
0	0	0	0	F(0,0,0)
1	0	0	1	F(0,0,1)
2	0	1	0	F(0,1,0)
3	0	1	1	F(0,1,1)
4	1	0	0	F(1,0,0)
5	1	0	1	F(1,0,1)
6	1	1	0	F(1,1,0)
7	1	1	1	F(1,1,1)

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Existem 2^8 funções lógicas de 3 variáveis diferentes, em que o número de linhas é $2^3 = 8$ e os valores possíveis por linha são {0,1} = 2.

Como as tabelas de verdade apenas são viáveis com poucas variáveis, é conveniente saber convertê-las para expressões algébricas. Apresentam-se agora algumas definições a usar com as expressões algébricas.

Um **literal** é uma variável ou o complemento duma variável. Exemplo: X, Y, X'.

Um **termo de produto** é um literal ou um produto lógico de 2 ou mais literais. Exemplo: Z' , W·X·Y , W·X'·Y'

A **soma-de-produtos** (**SOP**) é uma soma lógica de termos de produto. Exemplo: Z' + W·X·Y

Um **termo de soma** é um literal ou uma soma lógica de 2 ou mais literais. Exemplo: Z' , W+X+Y , W+X'+Y'

O **produto-de-somas** (**POS**) é um produto lógico de termos de soma. Exemplo: Z' · (W+X+Y)

Um **termo normal** é um termo de produto, ou de soma, em que cada variável só aparece uma vez. Exemplos de termos **não**-normais: W·X·X'·Z' , W'+Y'+Z+W'

Um **mintermo** de n-variáveis é um termo de produto normal com **n** literais. Exemplos com 4 variáveis: W·X·Y·Z' , W'·X'·Y·Z

Um **maxtermo** de n-variáveis é um termo de soma normal com **n** literais. Exemplos com 4 variáveis: W+X+Y+Z' , W'+X'+Y+Z

Há uma correspondência entre a tabela de verdade e os mintermos e maxtermos. Um mintermo é um termo de produto que é 1 numa linha da tabela de verdade, enquanto um maxtermo é um termo de soma que é 0 numa linha da tabela de verdade.
A tabela que se segue mostra, a título de exemplo, os mintermos e maxtermos para uma função de 3-variáveis F(X,Y,Z).

Row	X	Y	Z	F	Minterm	Maxterm
0	0	0	0	F(0,0,0)	X′·Y′·Z′	X+Y+Z
1	0	0	1	F(0,0,1)	X′·Y′·Z	X+Y+Z′
2	0	1	0	F(0,1,0)	X′·Y·Z′	X+Y′+Z
3	0	1	1	F(0,1,1)	X′·Y·Z	X+Y′+Z′
4	1	0	0	F(1,0,0)	X·Y′·Z′	X′+Y+Z
5	1	0	1	F(1,0,1)	X·Y′·Z	X′+Y+Z′
6	1	1	0	F(1,1,0)	X·Y·Z′	X′+Y′+Z
7	1	1	1	F(1,1,1)	X·Y·Z	X′+Y′+Z′

Um mintermo de **n**-variáveis pode ser representado por um inteiro com **n**-bits, que se designa por número do mintermo. No mintermo **i**, uma variável surge complementada se o bit correspondente na representação binária de **i** for 0; senão, a variável é não-complementada. Por exemplo, à linha 5 (101) corresponde o mintermo X·Y'·Z. No maxtermo **i**, uma variável surge complementada se o bit correspondente na representação binária de **i** for 1; senão, a variável é não-complementada. Por exemplo, à linha 5 (101) corresponde o maxtermo X'+Y+Z'. Para que a especificação dos mintermos e maxtermos faça sentido, é preciso conhecer o número de variáveis da função e a sua ordem (X,Y,Z nos exemplos).

A partir da correspondência entre a tabela de verdade e os mintermos, pode derivar-se uma representação algébrica dessa função lógica. A **soma canónica** duma função lógica é uma soma dos mintermos que correspondem a linhas da tabela de verdade para as quais a função é 1. Por exemplo, a partir da tabela seguinte obtém-se:

$F = \sum_{X,Y,Z} m(0,3,4,6,7) = X'\cdot Y'\cdot Z' + X'\cdot Y\cdot Z + X\cdot Y'\cdot Z' + X\cdot Y\cdot Z' + X\cdot Y\cdot Z$

A notação $\sum_{X,Y,Z} m(0,3,4,6,7)$ identifica 1 lista de mintermos e representa a soma dos mintermos 0, 3, 4, 6 e 7 envolvendo as variáveis X, Y e Z.

À lista de mintermos também se pode dar o nome de ***on-set*** da função lógica.

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

A partir da correspondência entre a tabela de verdade e os maxtermos, pode derivar-se uma representação algébrica dessa função lógica. O **produto canónico** duma função lógica é um produto dos maxtermos que correspondem a linhas da tabela de verdade para as quais a função é 0.

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Como exemplo, a partir da próxima tabela obtém-se:

$F = \prod_{X,Y,Z} M(1,2,5) = (X+Y+Z') \cdot (X+Y'+Z) \cdot (X'+Y+Z')$

A notação $\prod_{X,Y,Z} M(1,2,5)$ identifica uma lista de maxtermos e representa o produto dos maxtermos 1, 2 e 5 envolvendo as variáveis X, Y e Z.

À lista de maxtermos também se pode dar o nome de ***off-set*** da função lógica. É fácil converter uma lista de mintermos numa lista de maxtermos. Para uma função de **n**-variáveis, os mintermos e maxtermos pertencem ao conjunto {0, 1, …, 2^n-1}. Uma lista de mintermos ou de maxtermos é um subconjunto destes números. Para mudar dum tipo de lista para o outro, utiliza-se o subconjunto de números complementar.

Alguns exemplos:

$\sum_{A,B,C} m(0,1,2,3) = \prod_{A,B,C} M(4,5,6,7)$
$\sum_{X,Y} m(1) = \prod_{X,Y} M(0,2,3)$
$\sum_{W,X,Y,Z} m(1,2,3,5,8,12,13) = \prod_{W,X,Y,Z} M(0,4,6,7,9,10,11,14,15)$

Foram apresentadas 5 formas distintas de representar funções lógicas combinacionais:

- A tabela de verdade
- A soma algébrica de mintermos (a soma canónica)
- A lista de mintermos, com notação $\sum$
- Produto algébrico de maxtermos (o produto canónico)
- A lista de maxtermos, com notação $\prod$

Qualquer destas representações contém exactamente a mesma informação. A partir duma delas, pode derivar-se cada uma das outras 4 aplicando uma regra de conversão simples.

2.9. Exemplos

1. Para F = X·Y + X·Y'·Z + X'·Y·Z, obter a expressão de F' na forma produto de somas.

```
F' = (X.Y + X.Y'.Z + X'.Y.Z)'
   = (X.Y)'·(X.Y'.Z)'·(X'.Y.Z)'
   = (X'+Y').(X'+Y+Z').(X+Y'+Z')
```

2. Escreva a função G(X,Y,Z) = X + Y·Z como uma lista de mintermos

```
G = X + Y·Z
  = X·(Y+Y')·(Z+Z') + Y·Z·(X+X')          [T5]
  = XYZ + XYZ'+ XY'Z + XY'Z'+ XYZ + X'YZ
  = X'YZ + XY'Z' + XY'Z + XYZ'+ XYZ       [T3]
  = Σ x,Y,Z m(3,4,5,6,7)
```

3. Obter o produto de maxtermos para a função H = X'·Y' + X·Z.

```
H = X'Y' + XZ
  = (X'Y'+X)(X'Y'+Z)              [T8'] e+A.B=(e+A).(e+B)
  = (X'+X).(Y'+X).(X'+Z).(Y'+Z)   [T8'] e+A.B=(e+A).(e+B)
  =     1 .(X+Y').(X'+Z).(Y'+Z)   [T5 ] X+X'=1
```

Em cada soma da expressão anterior falta uma variável:

```
X+Y' = X+Y'+ZZ' = (X+Y'+Z)(X+Y'+Z')       [T5'] [T8']
X'+Z = X'+Z+YY' = (X'+Y+Z)(X'+Y'+Z)       [T5'] [T8']
Y'+Z = Y'+Z+XX' = (X+Y'+Z)(X'+Y'+Z)       [T5'] [T8']
```

Combinando estes termos:

$$H = (X+Y'+Z)(X+Y'+Z')(X'+Y+Z)(X'+Y'+Z)(X+Y'+Z)(X'+Y'+Z)$$
$$= \prod_{X,Y,Z} M(2,3,4,6)$$

4. Obter a lista de maxtermos para H = X'·Y' + X·Z, usando a tabela de verdade que se segue.

X	Y	Z	W
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

A partir da tabela obtém-se:

$$H = \prod_{X,Y,Z} M(2,3,4,6)$$
$$H = \sum_{X,Y,Z} m(0,1,5,7)$$

Compare esta solução com a que se obteve no exemplo 3.

5. Obter uma expressão para J = XYZ + XYZ' + XY'Z + X'YZ com um número reduzido de operadores.

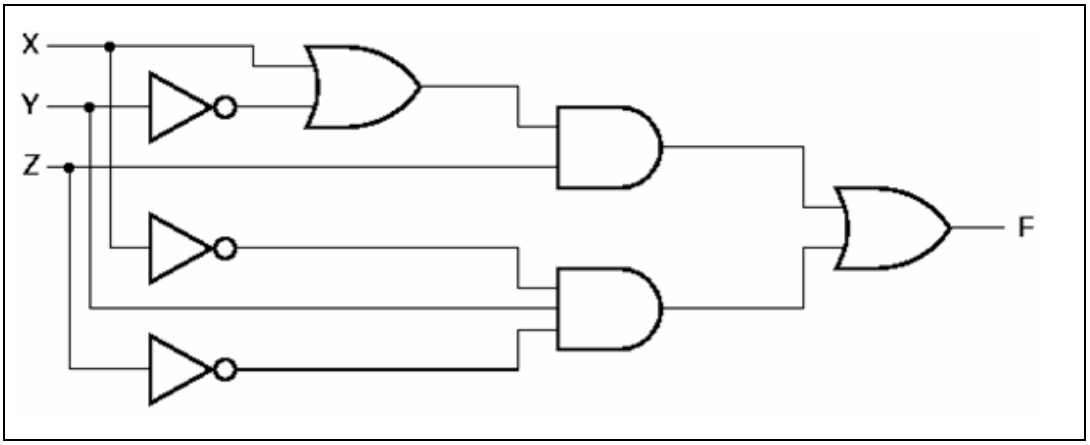
$$J = XYZ + XYZ' + XYZ + XY'Z + XYZ + X'YZ \qquad [T3]\ X+X=X$$
$$= XY(Z+Z') + X(Y+Y')Z + (X+X')YZ$$
$$= XY+XZ+YZ \qquad [T5]\ X+X'=1$$

3. Conceitos sobre Sistemas Combinacionais

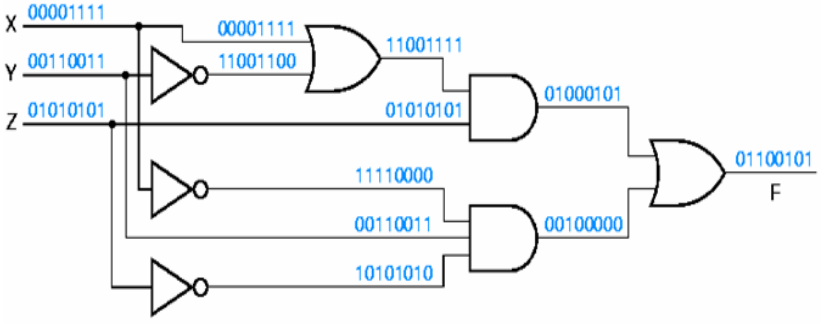
3.1. Análise, síntese e alterações a um circuito

Após obter a descrição formal da função lógica dum circuito, pode (i) determinar-se o seu comportamento para várias combinações de entradas; (ii) alterar-se a descrição algébrica de modo a induzir diferentes estruturas para o circuito; (iii) transformar a descrição algébrica numa forma normalizada (por exemplo, PLD); (iv) utilizar a descrição algébrica do circuito para analisar um sistema de maior dimensão, do qual este sistema seja parte integrante.

Considere-se o seguinte circuito.



Pode obter-se a **tabela de verdade** do circuito inspeccionando o seu comportamento para todas as combinações (2^n) das entradas.

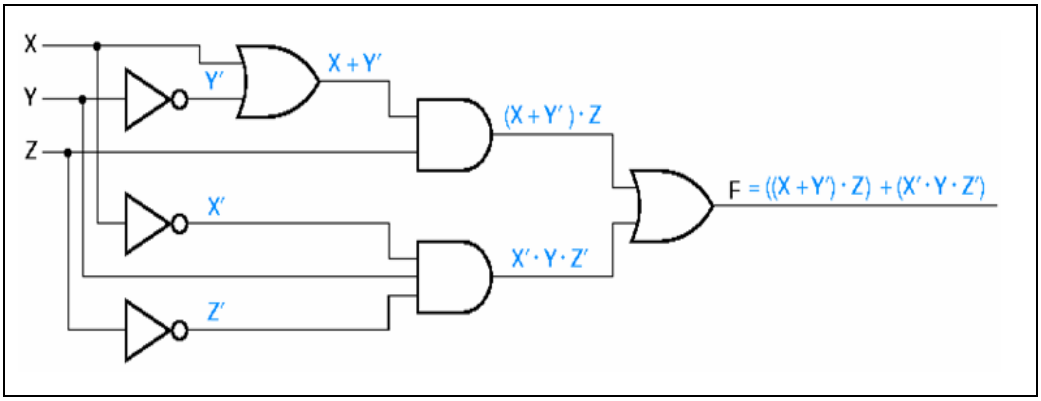


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

A partir da tabela de verdade pode extrair-se de forma directa uma expressão lógica (por exemplo, SOP).

Esta técnica é muito morosa e só é exequível se o número de variáveis de entrada for reduzido. A solução é recorrer à abordagem algébrica.

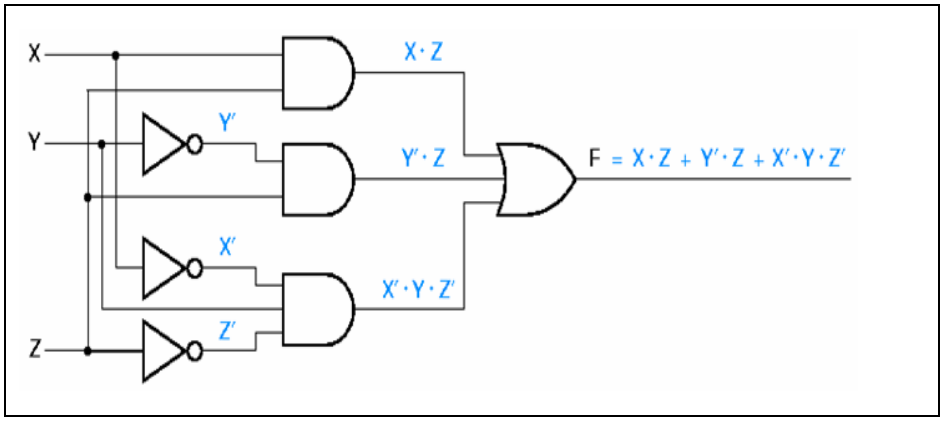
Abordagem algébrica: é uma abordagem em que se analisa o circuito desde as entradas até às saídas, construindo uma expressão correspondente aos operadores e à estrutura do circuito



Aplicando a abordagem algébrica ao circuito anterior obtém-se:
F = ((X+Y') · Z) + (X'·Y·Z')

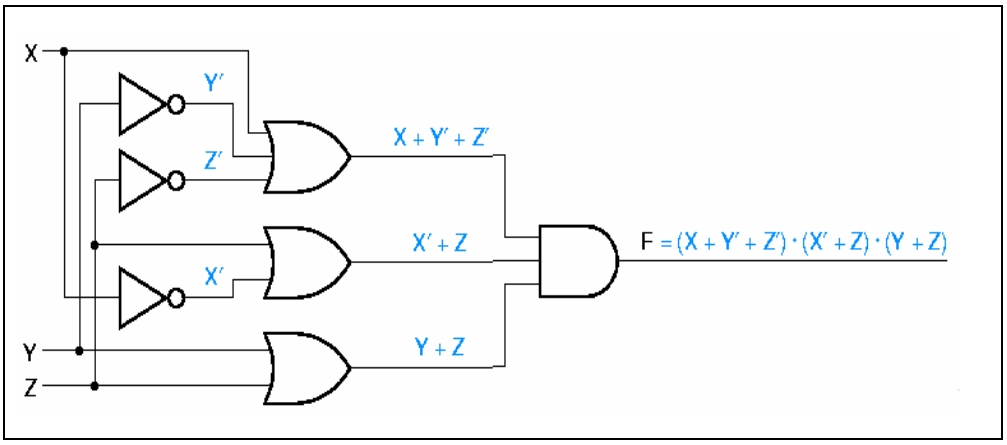
Após aplicar uma transformação algébrica obtém-se uma expressão e um circuito diferentes

F = ((X+Y') · Z) + (X'·Y·Z')
= (X·Z) + (Y'·Z) + (X'·Y·Z´)



A partir da expressão de F original, também se pode derivar uma expressão do tipo POS (aplicando T8' e T5):

F = ((X+Y') · Z) + (X'·Y·Z')
= (X+Y'+X') · (X+Y'+Y) · (X+Y'+Z') · (Z+X') · (Z+Y) · (Z+Z') **[T8']**
= 1 · 1 · (X+Y'+Z') · (Z+X') · (Z+Y) · 1 = **[T5]**
= (X+Y'+Z') · (X'+Z) · (Y+Z)



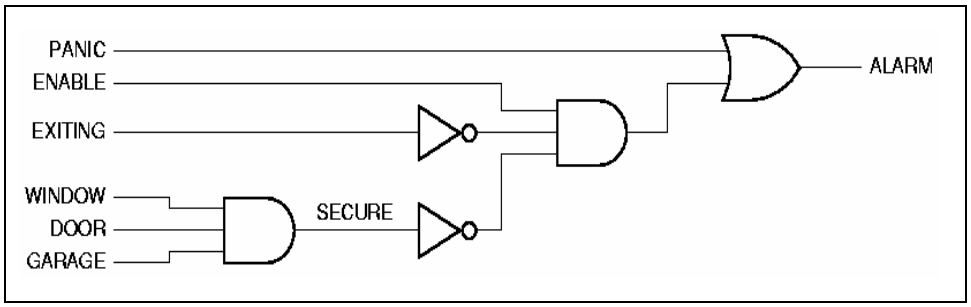
O ponto de partida para projectar um circuito combinacional é normalmente a sua descrição em linguagem natural (por exemplo, em português).

Exemplo: construir um circuito de alarme.

"A saída ALARM é 1 se a entrada PANIC for 1 **ou** se a entrada ENABLE for 1, a entrada EXITING for 0 e a casa não estiver segura. A casa é segura se as entradas WINDOW, DOOR e GARAGE forem todas 1".

ALARM = PANIC + ENABLE· EXITING'· SECURE'
SECURE = WINDOW· DOOR· GARAGE

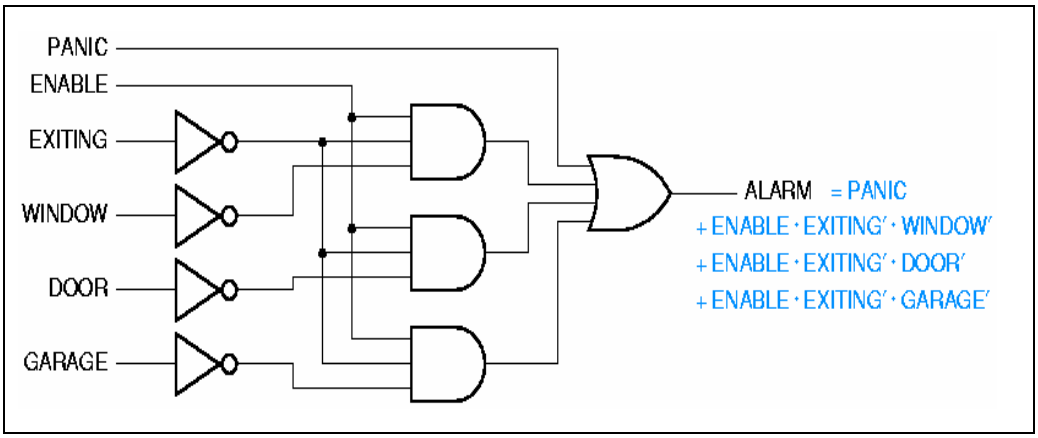
O circuito que concretiza a expressão de ALARM é o que se apresenta abaixo.



A concretização da expressão de ALARM na forma SOP é:

ALARM= PANIC + ENABLE· EXITING'· (WINDOW· DOOR· GARAGE)'
ALARM= PANIC + ENABLE· EXITING'· (WINDOW'+DOOR'+GARAGE')
ALARM= PANIC + ENABLE· EXITING'· WINDOW' +
ENABLE· EXITING'· DOOR' + ENABLE· EXITING'· GARAGE'

à qual corresponde o seguinte circuito:

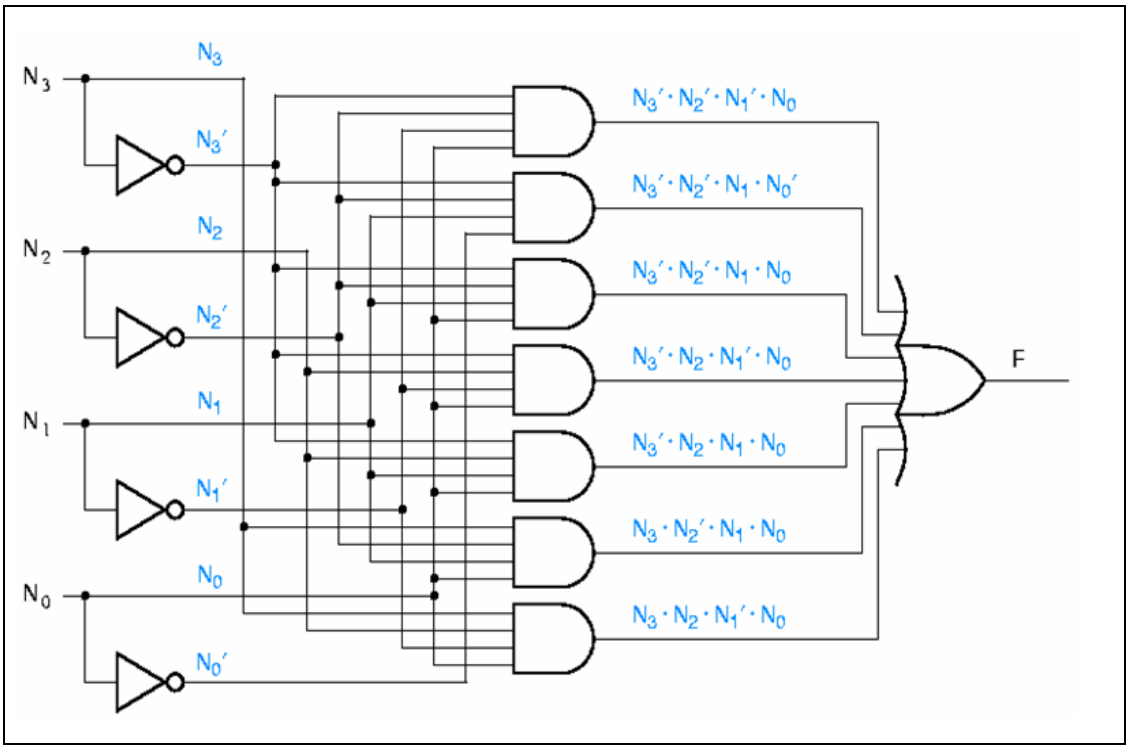


Outras vezes, a descrição começa com uma listagem das combinações das entradas para as quais uma determinada saída deve estar activa ou inactiva. Esta listagem equivale à tabela de verdade dessa saída.

Exemplo: construir um circuito que detecte números primos de 4-bits.
"Dada uma combinação de entrada $N=N_3N_2N_1N_0$ com 4-bits, o circuito gera 1 na saída quando N=1,2,3,5,7,11,13 e gera 0 nos outros casos."

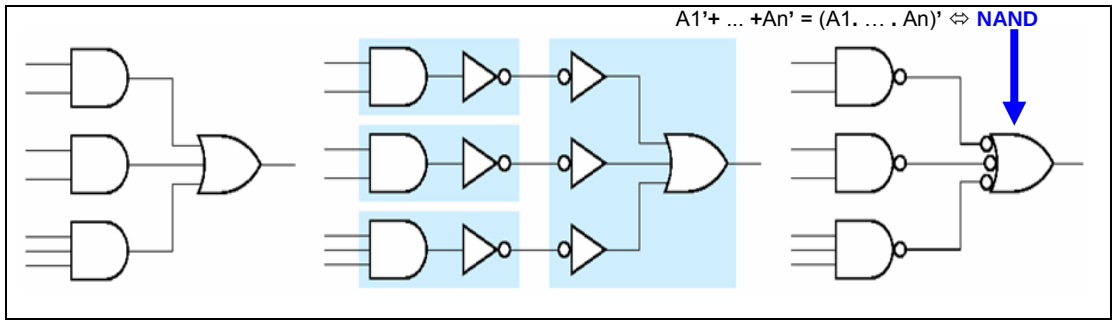
$$F = \sum_{N3,N2,N1,N0} m(1,2,3,5,7,11,13) =$$
$$N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0' + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 +$$
$$N_3' \cdot N_2 \cdot N_1 \cdot N_0 + N_3 \cdot N_2' \cdot N_1 \cdot N_0 + N_3 \cdot N_2 \cdot N_1' \cdot N_0$$

A concretização da expressão SOP da saída do detector é:

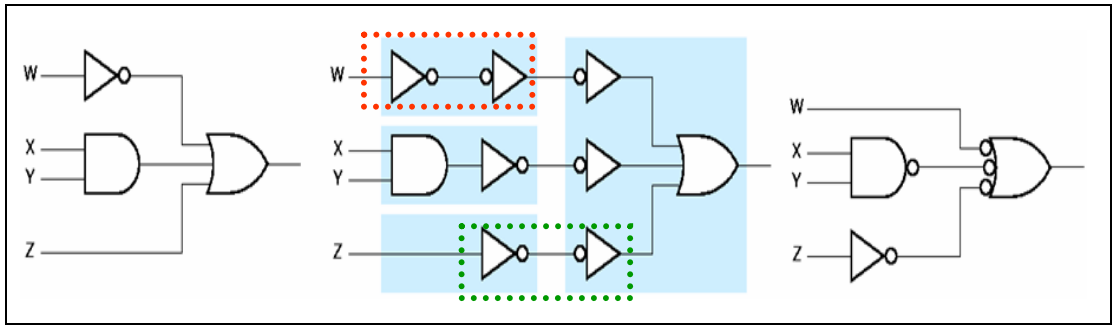


Até aqui foram apresentados métodos para projectar circuitos que usam apenas portas AND, OR e NOT. Em certas situações, o projectista pode querer usar portas NAND ou NOR, dado que são mais rápidas que AND's e OR's na maioria das tecnologias. Contudo, a generalidade das pessoas não desenvolve proposições lógicas usando o NAND e o NOR como elementos de ligação. Não se diz: "*Não gosto duma rapariga, se ela não for inteligente ou não for elegante* ***e*** *também se ela não for rica ou não for simpática*", ou seja, G'=(I'+E')·(R'+S'). É mais frequente dizer-se: "*Gosto duma rapariga, se ela for inteligente e elegante* ***ou*** *se ela for rica e simpática*", ou seja, G = (I·E) + (R·S).

Qualquer expressão lógica pode ser convertida numa expressão SOP equivalente e ser deste modo implementada com portas AND e OR. Um circuito AND-OR com 2-níveis pode ser convertido num circuito NAND-NAND com 2-níveis, através duma simples substituição de portas (ver o exemplo abaixo).



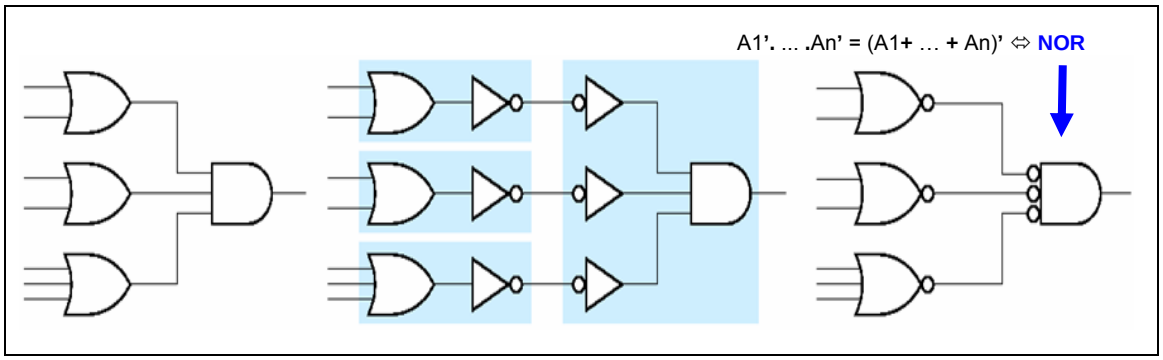
Se os termos (de produto) da expressão SOP incluírem apenas um literal, os inversores a aplicar a esse termo podem ser ou não necessários (ver o exemplo abaixo).



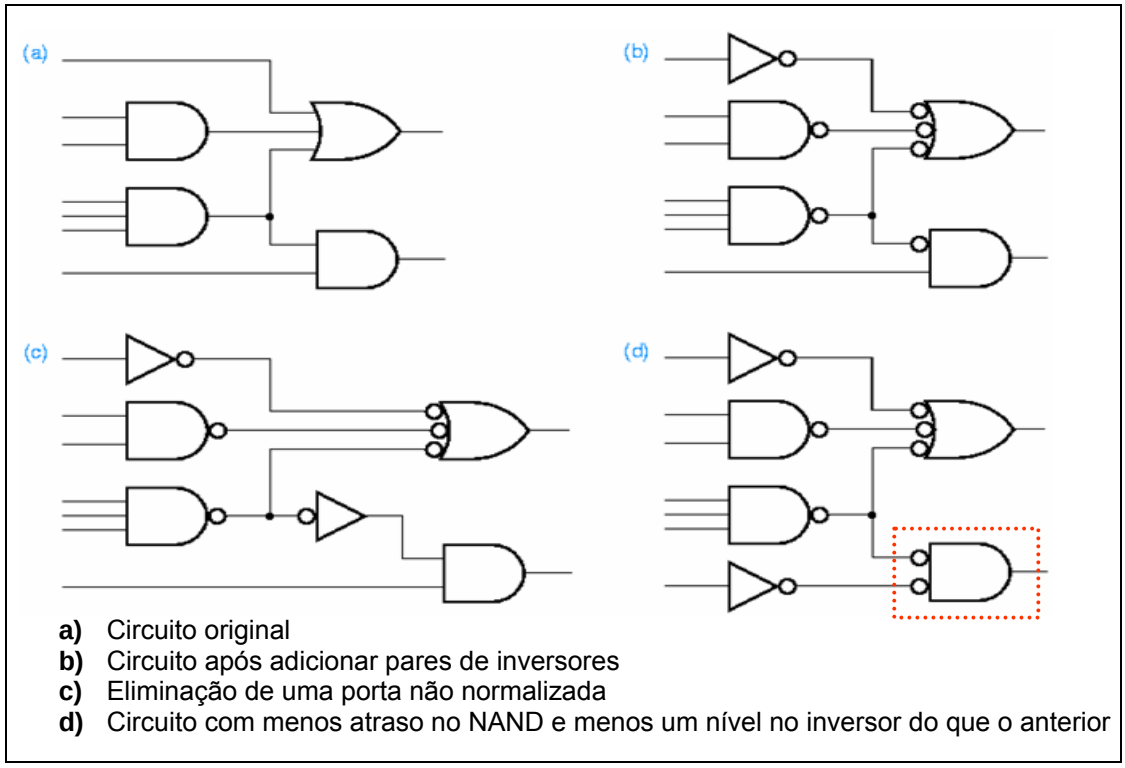
Vimos que qualquer expressão SOP pode ser concretizada de duas formas: através dum circuito AND-OR ou dum circuito NAND-NAND. Aplicando o princípio da dualidade a esta regra, obtemos uma declaração que também é verdadeira:

> *Qualquer expressão POS pode ser concretizada de duas formas: através dum circuito OR-AND ou dum circuito NOR-NOR.*

A figura seguinte mostra a aplicação do princípio da dualidade.

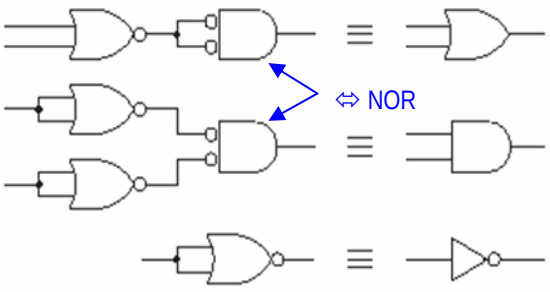


Estas alterações, como por exemplo a conversão para a estrutura NAND-NAND, podem ser aplicadas a qualquer circuito lógico.

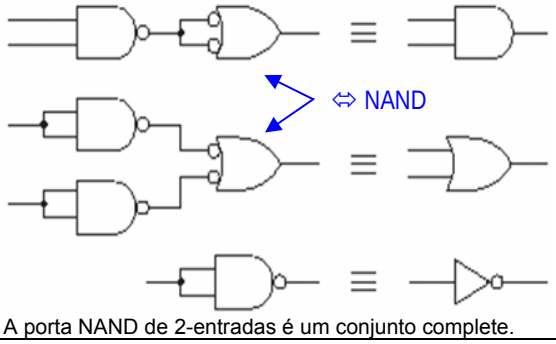


a) Circuito original
b) Circuito após adicionar pares de inversores
c) Eliminação de uma porta não normalizada
d) Circuito com menos atraso no NAND e menos um nível no inversor do que o anterior

Qualquer conjunto de tipos de porta lógica que permite concretizar qualquer função lógica é um **conjunto completo**. A porta AND de 2-entradas mais a porta OR de 2-entradas mais o inversor formam 1 conjunto complete. A figura seguinte mostra outros conjuntos completos: (i) a porta NAND de 2-entradas e (ii) a porta NOR de 2-entradas.



A porta NOR de 2-entradas é um conjunto complete.



A porta NAND de 2-entradas é um conjunto complete.

Qualquer função lógica pode ser expressa numa soma de produtos de literais. As portas AND e OR, com qualquer número de entradas, podem ser construídas a partir de portas do mesmo tipo com 2-entradas.

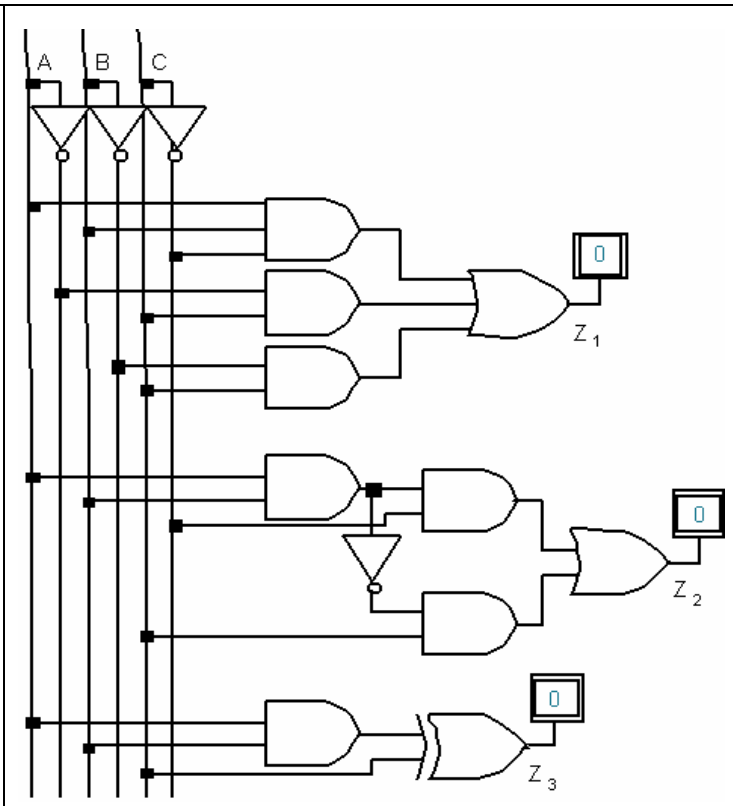
3.2. Minimização

Normalmente não é económico concretizar uma função lógica directamente a partir da 1ª expressão que ocorre. Isto porque as expressões canónicas (soma e produto) são especialmente consumidoras de recursos. A minimização de lógica emprega diversas técnicas para obter a implementação mais simples possível ao nível-da-porta para uma função. Contudo, o grau de simplificação depende da métrica usada. Três exemplos de métricas que podem ser usadas são:

- Número de literais;
- Número de portas (lógicas);
- Número de níveis de portas em cascata.

O **número de literais** mede a quantidade de ligações necessária para implementar uma função. O **número de portas** mede a área (*espaço ocupado*) do circuito. Existe uma relação directa entre o número de portas dum projecto e o número de circuitos integrados necessário à sua implementação. O **número de níveis de portas** mede o número de portas entre as entradas e as saídas do circuito. Quanto maior é o número de níveis, maior é o atraso no circuito. Verifica-se que ao adequar um circuito para apresentar um atraso mínimo, raramente se consegue uma implementação com o menor número de portas ou com as portas mais simples possíveis. Não é possível minimizar as três métricas ao mesmo tempo. Apresenta-se a seguir um exemplo que ilustra várias alternativas de simplificação duma função.

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



Dada a função

Z = A'B'C+A'BC+AB'C+ABC'=A'(B'+B)C+(A'+A)B'C+ABC'
= A'C+B'C+ABC'

i) A implementação Z_1 na figura anterior constitui uma concretização a dois-níveis *(dado que os inversores não contam)*, em que o número de literais é maior que nas outras alternativas.

ii) A implementação Z_2 é uma concretização multi-nível, que utilize portas com menos entradas mas possui um número de níveis maior:

Z = ABC'+A'C+B'C
= (AB)C'+(A'+B')C
= (AB)C'+(AB)'.C

iii) A implementação Z_3 é uma concretização que utilize uma porta mais complexa (XOR), que significa um atraso maior, mas emprega menos portas:

Z = (AB)C'+(AB)'.C = (AB) xor C

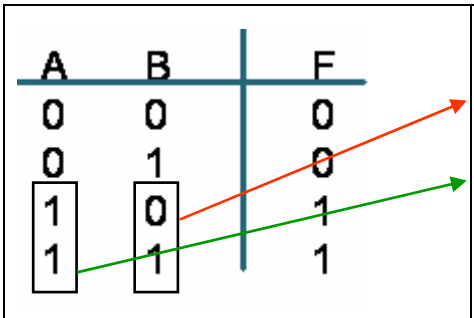
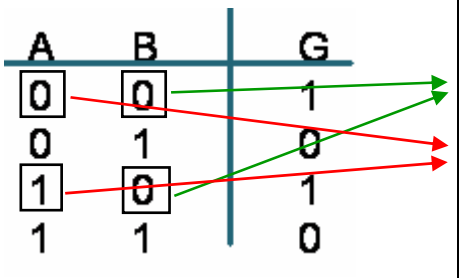
As técnicas de minimização reduzem o número e o tamanho das portas necessárias para construir um circuito, diminuindo assim o custo do sistema. Os métodos de minimização reduzem o custo dum circuito AND-OR ou OR-AND a 2-níveis através de:

- Minimizar o número de portas no primeiro nível;
- Minimizar o número de entradas em cada porta do primeiro nível;
- Minimizar o número de entradas em cada porta do segundo nível.

Os métodos de minimização não consideram o custo dos inversores à entrada. Isto resulta de se considerar que todas as variáveis de entrada, e seus complementares, estão disponíveis (adequado para implementações com PLD's). Também assumem que a função a minimizar está representada por uma tabela de verdade ou por uma lista de mintermos ou maxtermos. A minimização baseia-se nos teoremas **T10** e **T10'**:

produto·Y + produto·Y' = produto
(soma + Z) · (soma + Z') = soma

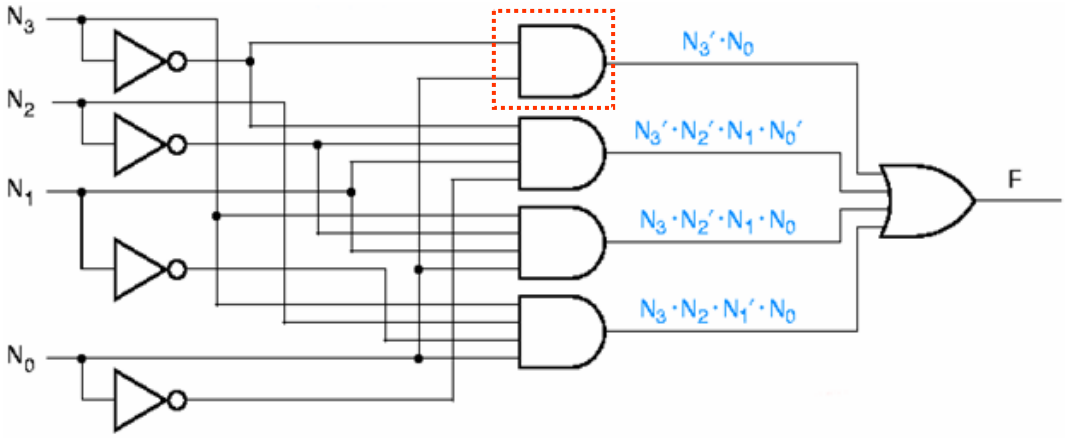
Estes teoremas indicam que "se dois termos diferem apenas numa variável, podem ser substituídos por um único termo com menos uma variável". Deste modo, poupa-se uma porta e a outra porta possui menos uma entrada.

<table>
<tr>
<td> 
</td>
<td>F = A B' + A B = A (B' + B) = A
O valor de B varia nas linhas do on-set
B é eliminado , A mantém-se
O valor de A não varia nas linhas do on-set</td>
</tr>
<tr>
<td> 
</td>
<td>G = A' B' + A B' = (A' + A) B' = B'
O valor de B não varia nas linhas do on-set
B mantém-se , A é eliminado
O valor de A varia nas linhas do on-set</td>
</tr>
</table>

Essência da simplificação:
Encontrar pares de elementos do ON-set em que apenas uma variável muda de valor. A variável que muda de valor pode ser eliminada.

Vamos aplicar esta técnica à expressão do detector de números primos.

$$F = \sum_{N3,N2,N1,N0} m(1,2,3,5,7,11,13) =$$
$$N_3'\cdot N_2'\cdot N_1'\cdot N_0 + N_3'\cdot N_2'\cdot N_1\cdot N_0 + N_3'\cdot N_2\cdot N_1'\cdot N_0 + N_3'\cdot N_2\cdot N_1\cdot N_0 + \ldots =$$
$$(N_3'\cdot N_2'\cdot N_1'\cdot N_0 + N_3'\cdot N_2'\cdot N_1\cdot N_0) + (N_3'\cdot N_2\cdot N_1'\cdot N_0 + N_3'\cdot N_2\cdot N_1\cdot N_0) + \ldots =$$
$$(N_3'\cdot N_2'\cdot N_0) + (N_3'\cdot N_2\cdot N_0) + \ldots = N_3'\cdot N_0 + \ldots$$

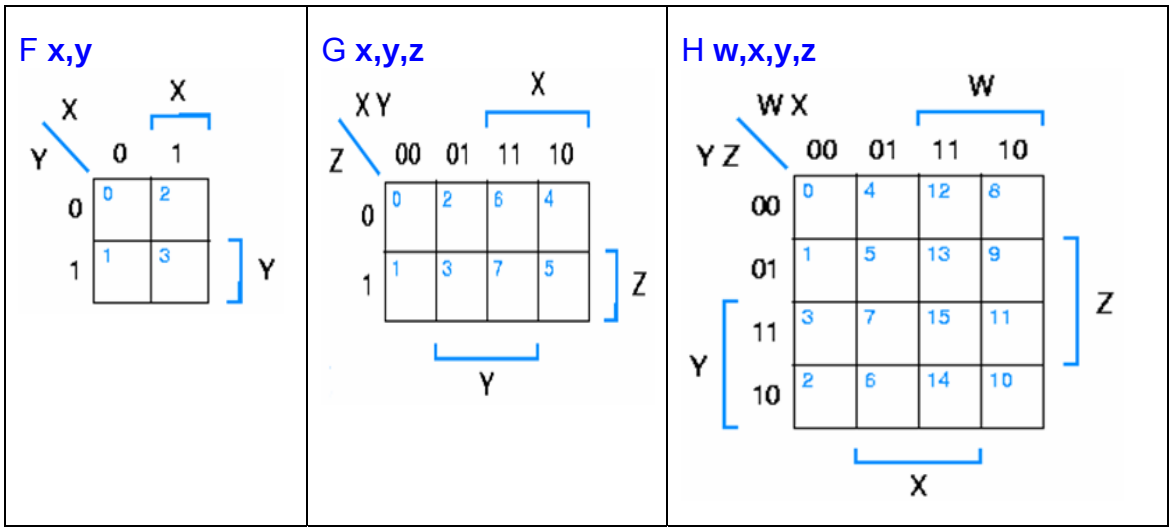


A porta AND assinalada é o resultado de simplificar 4 ANDs de 4 entradas.

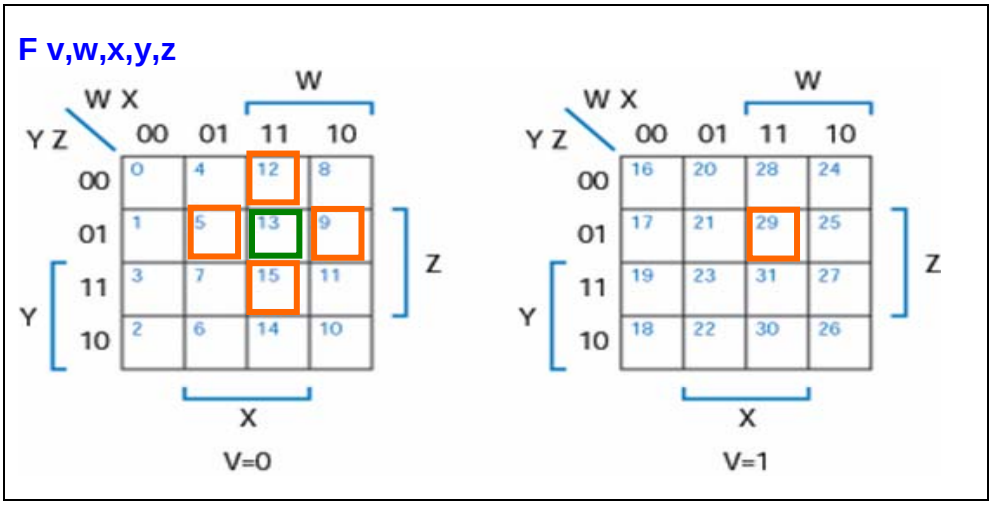
3.3. Mapas de Karnaugh

Não é fácil encontrar o par de termos que participa em cada simplificação. Um **mapa de Karnaugh** é uma representação gráfica para a tabela de verdade duma

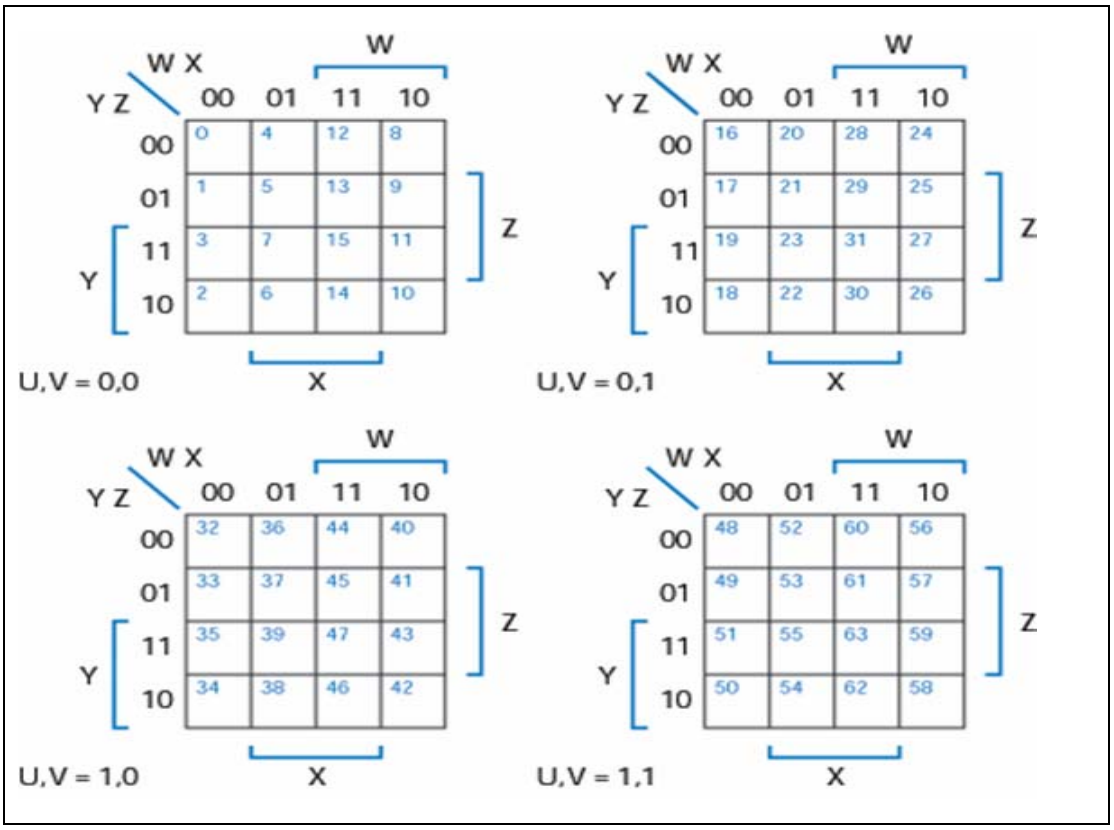
função lógica. O mapa para uma função lógica de **n**-entradas é um array com $\mathbf{2^n}$ células, uma por cada mintermo (ver a figura seguinte para os casos n=2, 3 e 4).



Os mapas de Karnaugh que se utilizam para representar funções com 5 e 6 variáveis não são tão adequados como os mapas de 2, 3 e 4 variáveis, dado a adjacência ser mais difícil de visualizar. Num mapa de **5 variáveis**, é necessário recorrer a 2 mapas de 4 variáveis colocados um ao lado do outro. Nesta representação, considera-se que um mapa é colocado por cima do outro, de modo a originar um objecto 3-dimensional. Neste caso, cada **célula** é adjacente de **5** células: **4** no mesmo mapa e **1** no outro (ver figura abaixo).

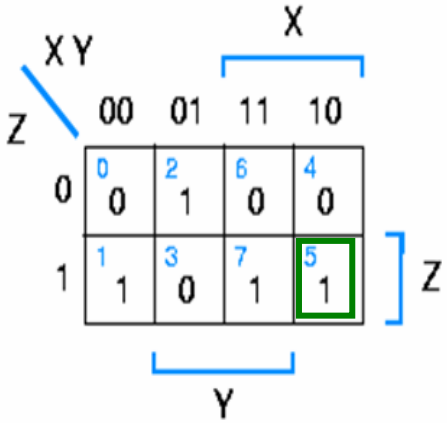


O mapa de Karnaugh que se segue é de **6** variáveis: **U**, **V**, **W**, **X**, **Y** e **Z**.



Para representar uma função lógica num mapa de Karnaugh, copiam-se os 1's e 0's da tabela de verdade para as células correspondentes do mapa. Cada célula do mapa corresponde a um mintermo da função.

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



A célula assinalada correspondente ao mintermo 5.

Na prática, apenas se copiam os 1's ou os 0's (não ambos) para as células, dependendo do tipo de expressão que se pretende obter (SOP ou POS).

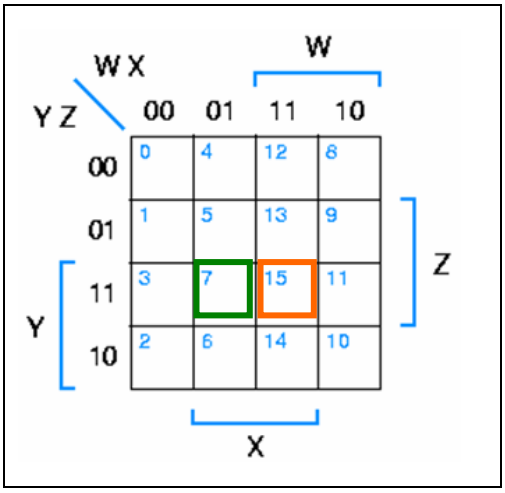
Q: Porque se usa uma ordenação estranha das linhas e colunas?

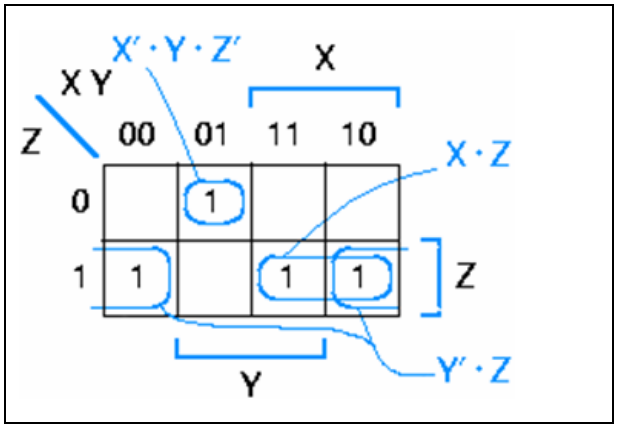
Porque assim, cada célula corresponde a uma combinação das entradas que difere apenas numa variável da combinação associada a cada uma das células vizinhas

imediatamente adjacentes. As células **7** e **15**, no mapa de 4 variáveis seguinte, diferem apenas no valor de W. Nos mapas de 3 e 4 variáveis, as células no canto esquerdo (direito) ou superior (inferior) também são vizinhas. As células 8 e 10 no mapa de 4 variáveis diferem apenas no valor de Y.

Uma vez que os pares de células-a-1 adjacentes correspondem a mintermos que diferem apenas numa variável, cada par pode ser combinado num único termo de produto, usando o teorema T10:

produto·Y + produto·Y' = produto





Para o mapa anterior, combinando as células 5 e 7:

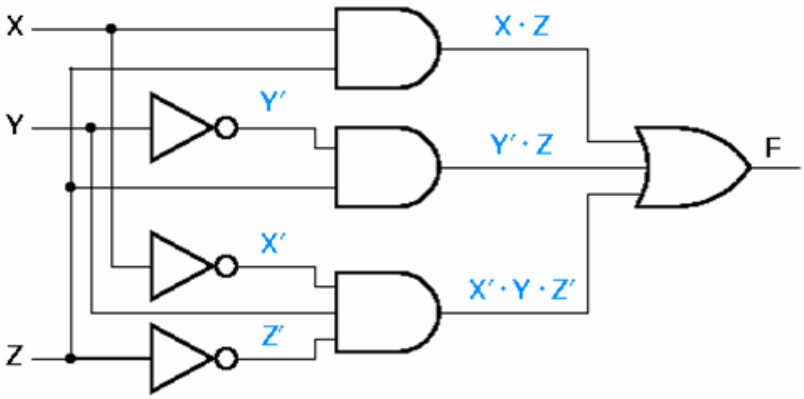
F = ... + X·Y'·Z + X·Y·Z
= ... + X·Z

Combinando as células 1 e 5:

F = X'·Y'·Z + X·Y'·Z + ...
= Y'·Z + ...

O mintermo 5 é incluído duas vezes. Não há problema porque X+X=X.

F = X·Z + Y'·Z + X'·Y·Z'



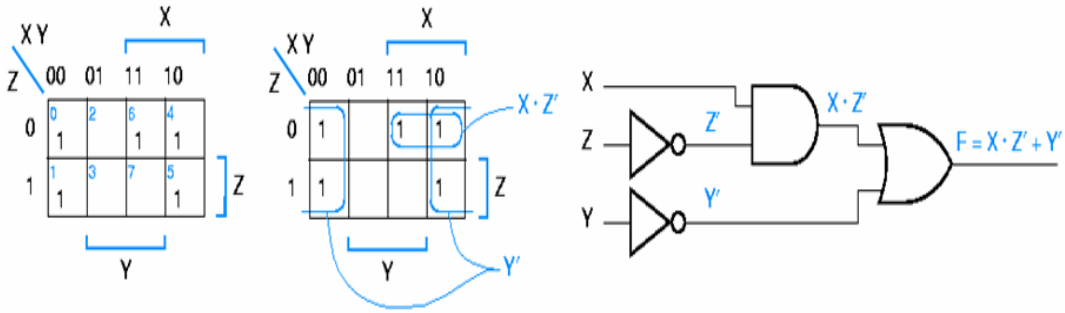
Concretização AND-OR da função F simplificada.

O procedimento utilizado para combinar células, pode ser estendido para permitir combinar **mais do que 2 células-a-1** num único termo.

F = $\sum_{X,Y,Z,}$ m(0,1,4,5,6) = X'·Y'·Z' + X'·Y'·Z + X·Y'·Z' + X·Y'·Z + X·Y·Z'

= [Y'·(X'·Z') + Y'·(X'·Z) + Y'·(X·Z') + Y'·(X·Z)] + X·Y·Z'
= Y'·[(X'·Z' + X·Z') + (X'·Z +X·Z)] + X·Y·Z' = Y'·[(X' + X).Z' + (X' +X)·Z] + X·Y·Z'

= Y'·(1.Z'+ 1.Z) + X·Y·Z' = Y'.(Z'+Z) + X·Y·Z' = Y'.1 + X·Y·Z' = Y' + X·Y·Z'



i) Simplificação com mapa de Karnaugh

ii) Concretização AND-OR de F simplificada com mapa de Karnaugh

Generalizando, pode combinar-se ***2^i* células-a-1** para originar um termo de produto com ***n-i*** literais (em que **n** = número de variáveis). A regra para combinar células-a-1 é descrita por:

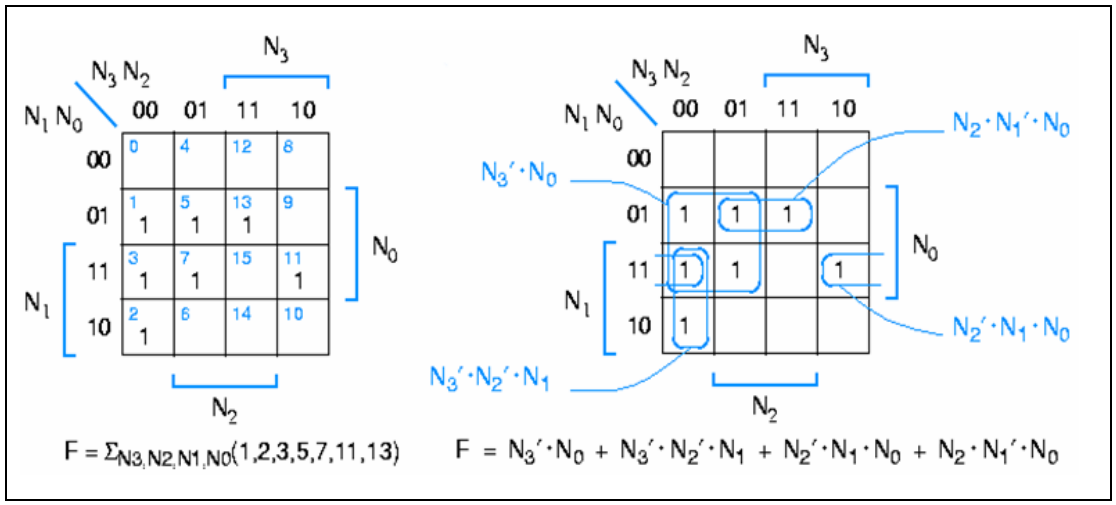
> ***Um conjunto de 2^i células-a-1 pode ser combinado se existirem i variáveis que assumem todas as possíveis combinações (2^i) dentro desse conjunto, enquanto as restantes n-i variáveis mantêm o mesmo valor em todo o conjunto.***
>
> ***O termo de produto resultante possui n-i literais, em que cada literal é: a variável complementada (se ela aparecer como 0 em todas as células-a-1) ou a variável não complementada (se ela aparecer como 1).***

Graficamente, podemos envolver conjuntos rectangulares de 2^n células-a-1 com um "rectângulo".

A partir de cada rectângulo que envolve células-a-1, obtém-se o correspondente termo de produto:

- Se o rectângulo cobre apenas zonas do mapa em que uma dada variável é 0 (1), então essa variável surge complementada (não complementada) no termo de produto.

- Se o rectângulo cobre zonas do mapa em que uma dada variável é 0 e 1, então essa variável não aparece no termo de produto.



Uma **soma mínima** para a função lógica F é uma expressão do tipo soma de produtos (SOP) para F tal que nenhuma outra expressão SOP para F possui menos termos de produto, e qualquer expressão SOP com o mesmo número de termos de produto possui pelo menos tantos literais como ela. A soma mínima possui o menor número de termos de produto possível (número de portas no 1° nível e número de entradas na porta do 2° nível) e o menor número de literais possível (número de entradas nas portas do 1° nível).
Uma função lógica **P implica** a função lógica **F** (P $\Rightarrow$ F) se para cada combinação de entradas em que P=1, então também F=1. Ou seja, F inclui ou cobre P. Um **implicante maior** duma função lógica F é um termo de produto normal P que implica F, de tal modo que se qualquer variável for eliminada de P, então o termo de produto resultante já não implica F. Em termos dum mapa de Karnaugh, um implicante maior de F é um conjunto de células-a-1 envolvido por um rectângulo, de tal modo que se tentarmos aumentá-lo (de modo a cobrir o dobro das células), ele vai cobrir um ou mais 0's.

> ***Teorema do implicante maior**: uma soma mínima é uma soma **de** implicantes maiores.*

Para encontrar uma soma mínima, **não é** necessário ter em conta qualquer termo de produto que não seja um implicante maior. A soma de todos os implicantes maiores duma função é designada por **soma completa**. A soma completa **não é** necessariamente uma soma mínima.

Algoritmo: obter a expressão SOP mínima através dum mapa de Karnaugh

Passo 1: Marcar os implicantes maiores

Escolher um "1" do ON-set ainda não coberto por qualquer implicante.

Encontrar os grupos de 1's (e X's) adjacentes desse elemento e que possuem a maior dimensão possível. A dimensão tem que ser 2^i.

Não esquecer a adjacência entre a linha superior e inferior, a coluna esquerda e direita e os cantos.

Repetir o Passo 1 para cada "1" do ON-set, de modo a encontrar todo os implicantes maiores.

Passo 2: Marcar os implicantes essenciais

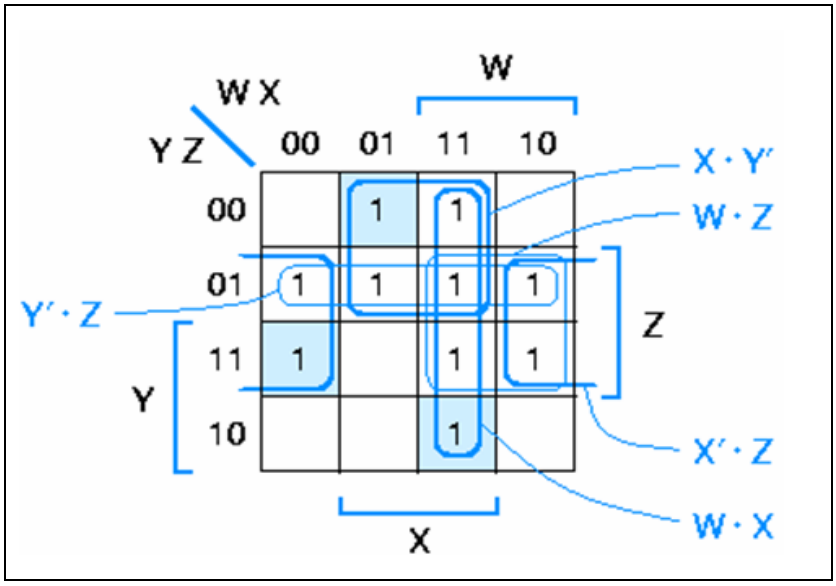
Visitar cada "1" do ON-set. Se o "1" for coberto por um único implicante maior, então este implicante é essencial e aparece na expressão final. Os restantes 1's cobertos pelo implicante não precisam ser revisitados.

Repetir o Passo 2 até que todos os implicantes essenciais tenham sido encontrados.

Passo 3: Cobertura adicional

Se existirem 1's não cobertos pelos implicantes essenciais, seleccionar o menor número de implicantes maiores que cobra todos esses 1's. Tentar várias alternativas de cobertura.

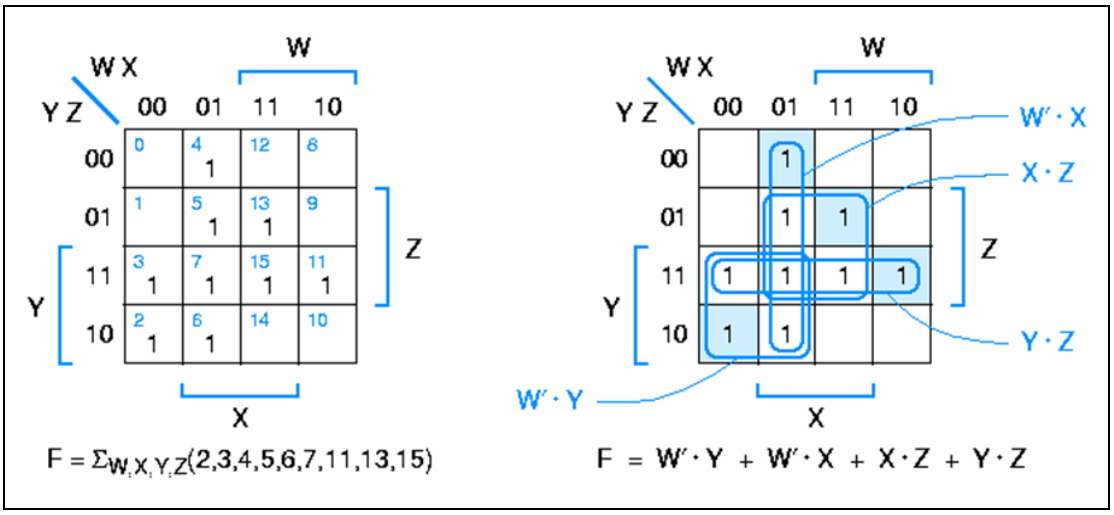
Considere a função F = $\sum_{W,X,Y,Z,}$ m(1,3,4,5,9,11,12,13,14,15). Esta função possui 5 implicantes maiores. A soma mínima inclui apenas 3 implicantes maiores: F = X·Y' + X'·Z + W·X. Como se decide quais os implicantes maiores a incluir na expressão minimizada duma função?



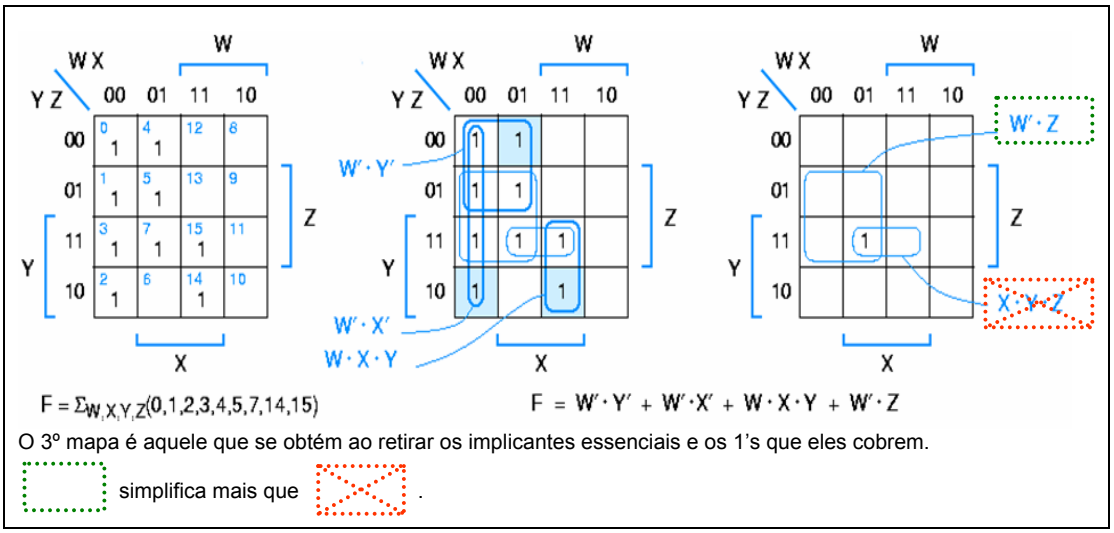
Para responder à pergunta anterior, apresentam-se mais algumas definições.

Uma **célula-a-1 distinguida** duma função lógica é uma combinação de entradas que é coberta por um único implicante maior. Um **implicante essencial** duma função lógica é um implicante maior que cobre uma ou mais células-a-1distinguidas. Os implicantes essenciais têm que aparecer em qualquer soma mínima. O 1º passo na selecção dos implicantes consiste em identificar as células-a-1 distinguidas e incluir os correspondentes implicantes essenciais na soma. Depois, se houver células-a-1 não cobertas pelos implicantes essenciais, falta encontrar a melhor forma de as cobrir.

Exemplo em que todos os implicantes maiores são essenciais:



Exemplo em que nem todos os implicantes maiores são essenciais:



O 3º mapa é aquele que se obtém ao retirar os implicantes essenciais e os 1's que eles cobrem.

simplifica mais que .

Aplicando o princípio da dualidade, pode minimizar-se expressões do tipo produto de somas (POS) se se funcionar com os 0's do mapa de Karnaugh. Cada 0 do mapa corresponde a um maxtermo. Uma forma mais fácil de encontrar o produto mínimo de F consiste em obter a soma mínima de F'. Obter F' é simples: os 1's de F' são os 0's de F. Após obter a soma mínima para F' *(F'sop)*, complementa-se o

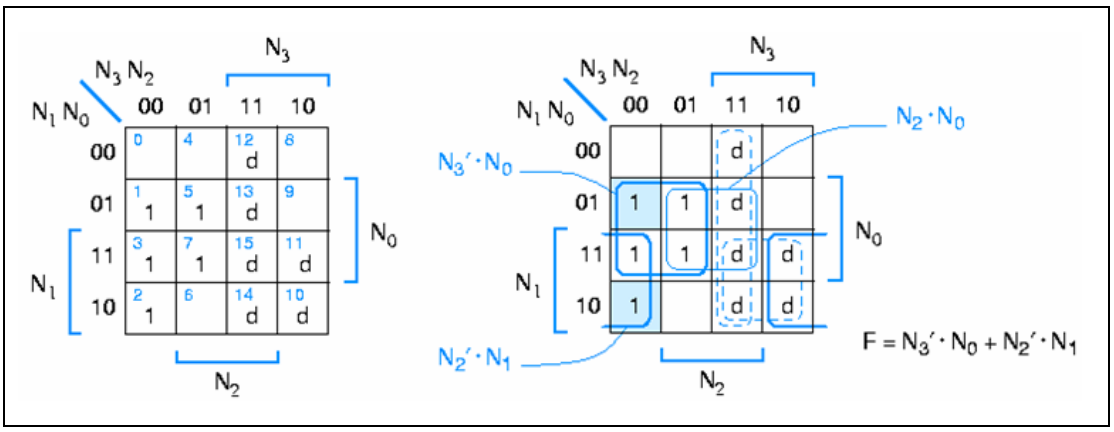
resultado obtido aplicando o teorema de DeMorgan generalizado [T14] , de modo a obter o produto mínimo para F (*Fpos*).

Um exemplo: F' = X·Y' + X'·Z + W·X
F = (X'+Y) · (X+Z') · (W'+X')

Por vezes, não interessa qual é o valor da saída da função para certas combinações das entradas. A estas saídas chama-se ***don't cares***.

Um exemplo: um detector de números primos em que a entrada *N* com 4-bits é sempre um dígito BCD, ou seja, os mintermos 10-15 nunca ocorrem.

$$F = \sum_{N3,N2,N1,N0} m\,(1,2,3,5,7) + \sum_{N3,N2,N1,N0} d\,(10,11,12,13,14,15)$$



Ao introduzir *don't cares* (X's), o procedimento usado para envolver os conjuntos de 1's com rectângulos é alterado do seguinte modo:

- Permite-se que os X's sejam incluídos nos conjuntos de 1's a envolver com rectângulos, para formar conjuntos tão grandes quanto possível. Desta forma reduz-se o número de variáveis nos implicantes maiores que lhe correspondem.

- Não se considera qualquer conjunto que contenha apenas X's. Incluir na função os termos de produto formados a partir de X's aumentaria desnecessariamente o seu custo, ou seja, a função ficaria menos minimizada.

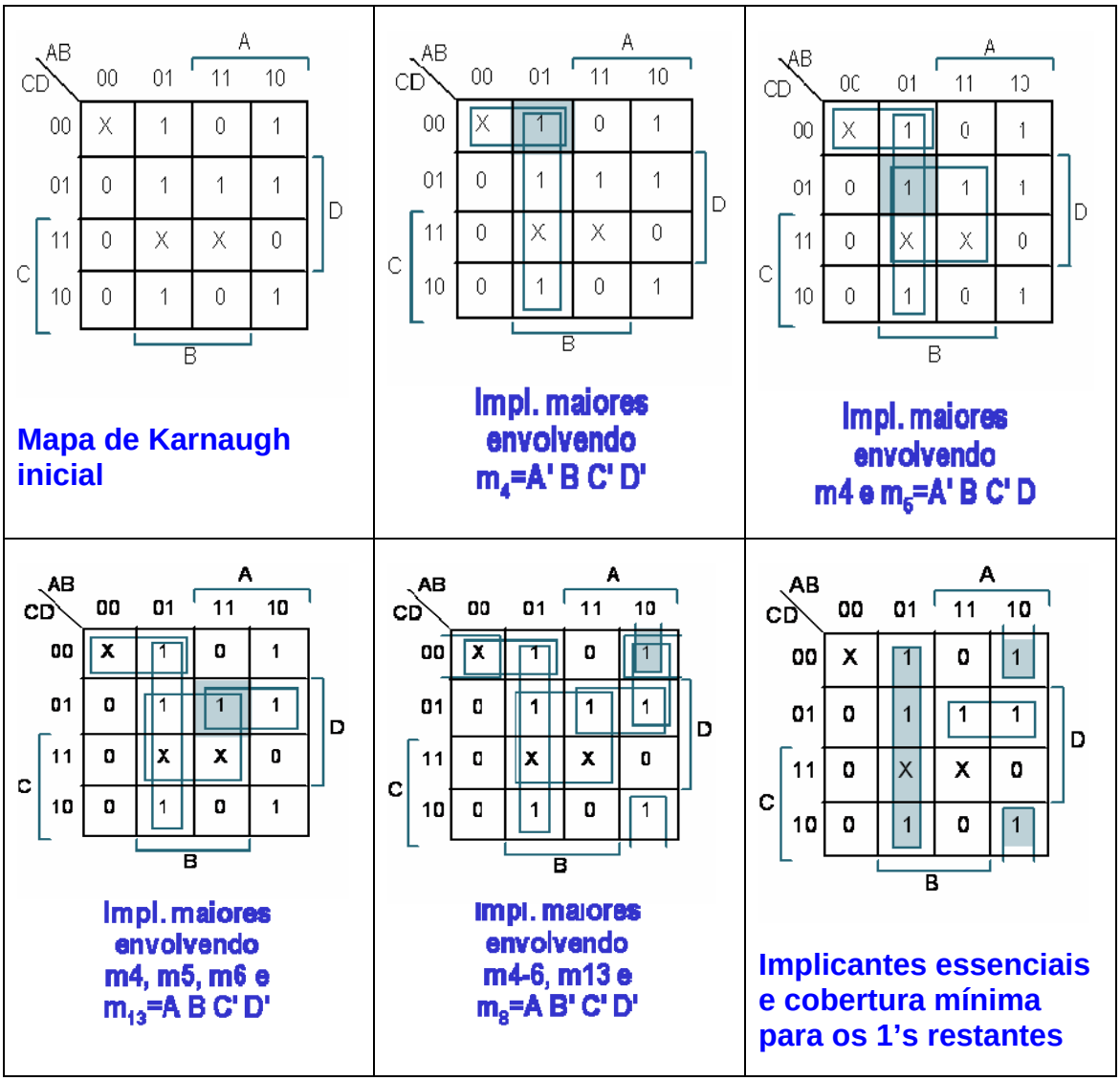
O resto do procedimento mantém-se válido. Por exemplo:

- Visitam-se apenas as células-a-1distinguidas e não as células-a-X distinguidas.

- Inclui-se na função simplificada apenas (i) os implicantes essenciais que correspondem a esses 1's e (ii) outros implicantes maiores que sejam necessários para cobrir todos os 1's do mapa.

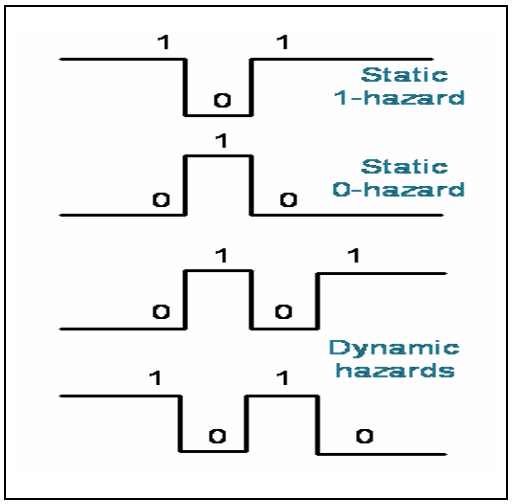
Apresenta-se agora, como exemplo, a simplificação da função:

$F = \sum_{A,B,C,D} m\,(4,5,6,8,9,10,13) + \sum_{A,B,C,D} d\,(0,7,15)$



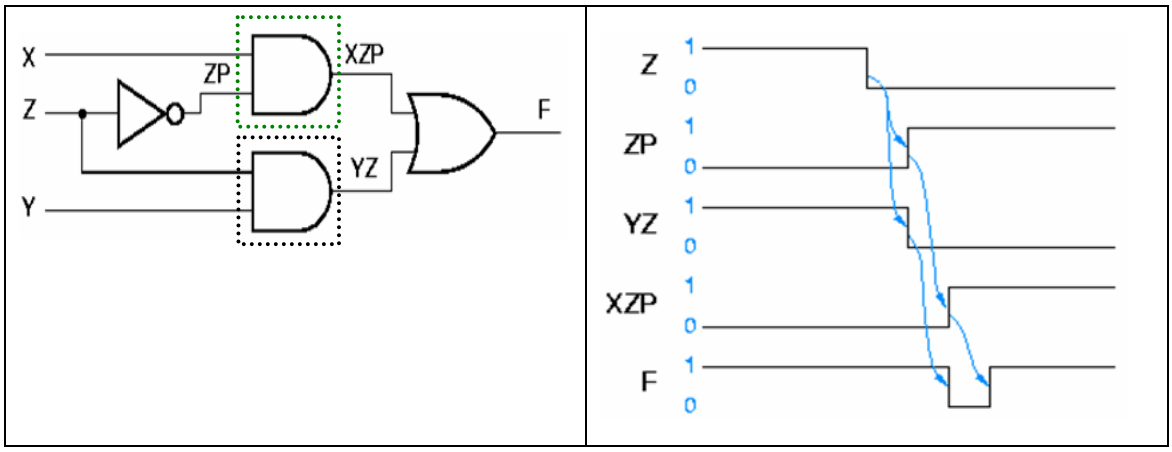
3.4. Hazards

Devido aos atrasos nos componentes electrónicos, um circuito pode originar um *glitch*. Um ***glitch*** é uma variação de curta duração no valor duma saída, quando não se espera nenhuma variação. Designa-se de ***hazard*** a situação em que existe a possibilidade de o circuito gerar um *glitch*. Ocorre um ***hazard* estático** quando existe a possibilidade de uma saída sofrer uma transição momentânea em condições em que se esperava que ela se mantivesse inalterada. Ocorre um ***hazard* dinâmico** quando for possível uma saída mudar mais do que uma vez, em condições em que se esperava que ela tivesse uma única transição (de 0 → 1 ou de 1→0).



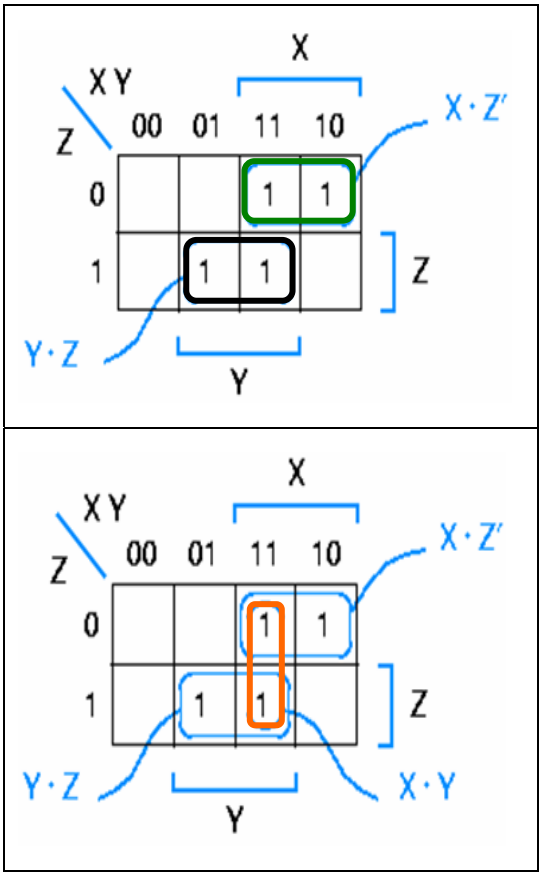
Um ***hazard* num 1 (0) estático** é um par de combinações de entradas que diferem apenas numa variável de entrada e em que ambas produzem 1 (0) na saída, de tal modo que pode ocorrer um 0 (1) momentâneo na saída, durante uma transição na variável de entrada que distingue essas combinações.

A figura seguinte mostra um circuito que apresenta um *hazard* quando X=Y=1 e Z transita de 1 → 0.

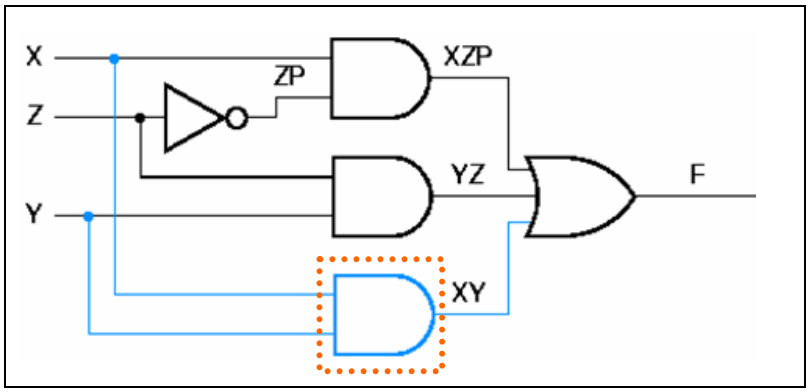


Os métodos usados para eliminar *hazards* consideram que apenas uma entrada varia em cada instante. Este pressuposto equivale a efectuar um deslocamento através de células vizinhas num mapa de Karnaugh. Os mapas de Karnaugh podem ser usados para detectar *hazards* estáticos em circuitos com estrutura AND-OR ou OR-AND. Num circuito bem projectado e que implemente uma soma de produtos a 2-níveis só podem ocorrer *hazards* em 1's estáticos (não em 0's estáticos).

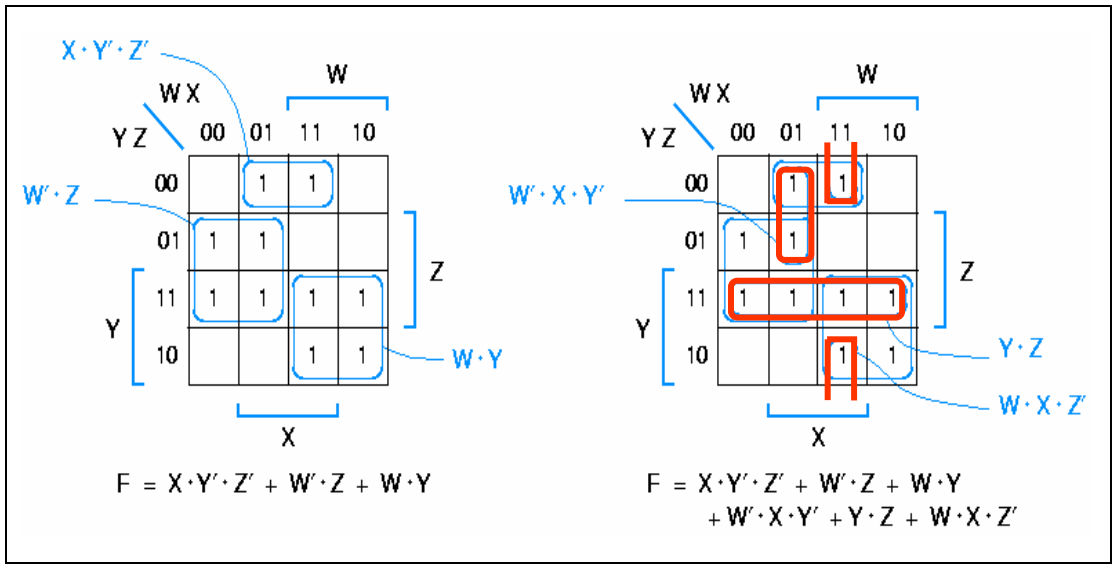
Para o mapa da figura seguinte, como não há um termo de produto que cobre ambas as combinações XYZ=111 e XYZ=110, é possível que se gere um breve *glitch* a 0 na saída (se o AND que muda para 0 o fizer antes do AND que muda para 1). Para eliminar o *hazard*, deve incluir-se no circuito uma porta AND extra.



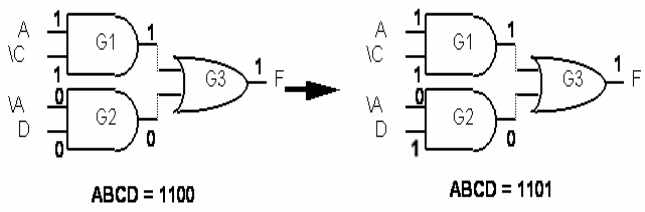
O circuito que resulta da eliminação do *hazard* com a porta AND extra é:



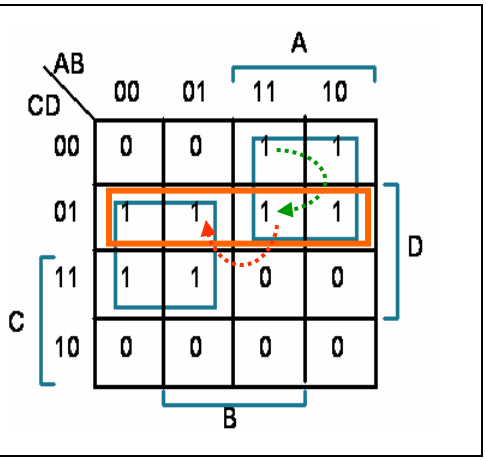
Outro exemplo, em que se usam 3 AND's extra para eliminar os *hazards estáticos*:

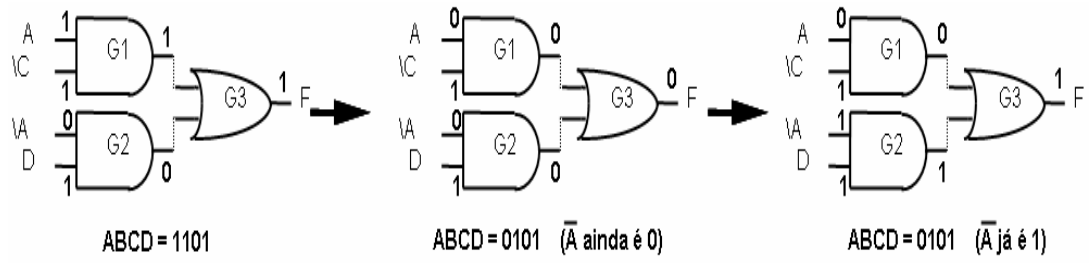


Para terminar, apresenta-se mais um exemplo, em que se mostra a ocorrência e eliminação de um *glitch* na função F = $\sum_{A,B,C,D,}(1,3,5,7,8,9,12,13)$.



i) Não ocorre um glitch (situação verde no mapa).





ii) Ocorre um glitch, que é eliminado com o termo extra C'.D (situação laranja no mapa).

4. VHDL

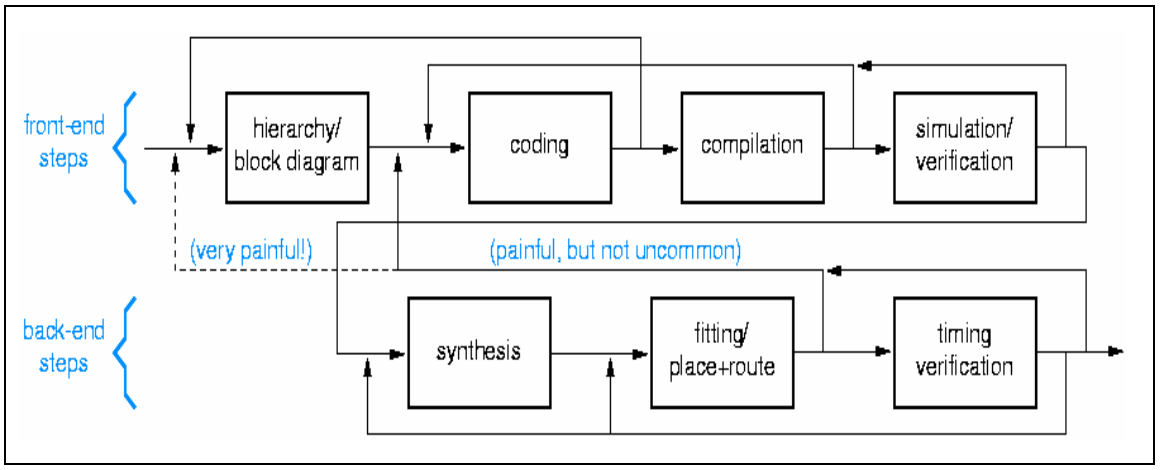
O VHDL foi desenvolvido na década de 80 pelo DoD e pelo IEEE. VHDL é um acrónimo de VHSIC Hardware Description Language; VHSIC é um acrónimo de Very High Speed Integrated Circuit.

O VHDL possui as seguintes características:

- Os projectos devem ser decompostos de forma **hierárquica**.
- Cada elemento dum projecto possui uma **interface** e uma especificação do seu **comportamento**.
- A especificação dum comportamento pode usar **um algoritmo ou uma estrutura** para definir o modo de operação do elemento.
- Pode modelar **concorrência**, **temporização** e o conceito de **relógio** *(clocking).*
- Permite **simular** a operação lógica e o comportamento temporal dum projecto.

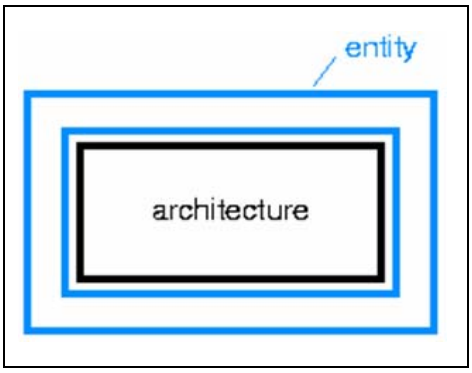
4.1. Fluxo de projecto

O VHDL começou por ser uma linguagem de **documentação** e **modelação**, que permitia especificar e simular o comportamento dos projectos. Actualmente existem **ferramentas de síntese** comerciais que geram a estrutura dos circuitos lógicos directamente a partir de especificações em VHDL.



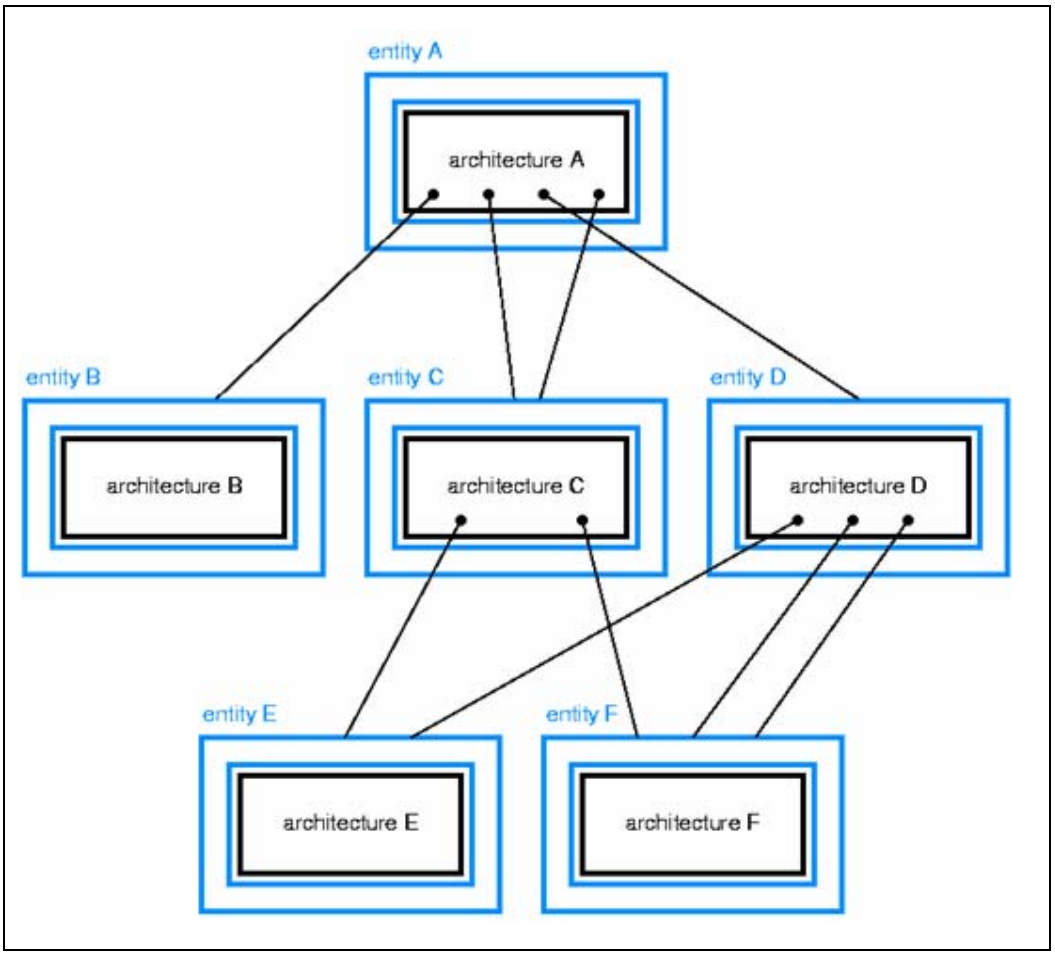
4.2. Entidades e arquitecturas

O VHDL foi desenvolvido tendo em consideração muitos dos princípios da programação estruturada. Muitas ideias do VHDL foram importadas do Pascal e do Ada. Uma das ideias chave do VHDL é a utilização de interfaces que definem a fronteira dos módulos (de hardware), ao mesmo tempo que escondem os pormenores relativos ao seu interior. Em VHDL, a **entidade** (entity) é uma declaração das entradas e saídas dum módulo. A **arquitectura** (architecture) é uma descrição detalhada da estrutura ou comportamento interno dum módulo.



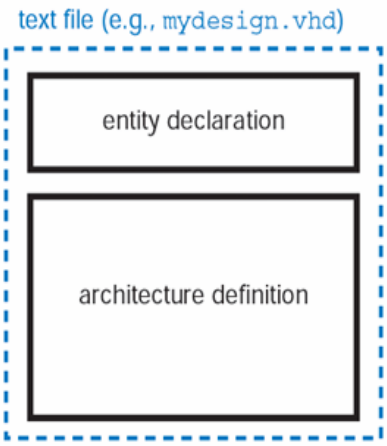
Uma arquitectura pode usar outras entidades. Uma arquitectura de nível superior pode usar uma entidade de nível inferior várias vezes. Várias arquitecturas de nível superior podem usar a mesma entidade de nível inferior. Estas facilidades são o suporte básico para projectar sistemas de forma hierárquica.

As **configurações** definem qual a arquitectura a usar em cada ocorrência duma entidade.



Num ficheiro texto, contendo código VHDL, a declaração da entidade e a definição da arquitectura estão separadas.

```
entity Inhibit is
  port (X,Y: in BIT;
        Z:   out BIT);
end Inhibit;

architecture Inhibit_arch of Inhibit is
begin
  Z <= '1' when X='1' and Y='0' else '0';
end Inhibit_arch;
```



A linguagem não distingue maiúsculas de minúsculas. Os comentários começam com 2 hífenes "--" e terminam no fim da linha. O VHDL possui palavras (chave) reservadas: port, is, in, out, begin, end, entity, architecture, if, case, ...

A **sintaxe** da declaração duma **entidade** é:

```
entity entity-name is
  port(signal-names : mode signal-type;
       signal-names : mode signal-type;
         ...
       signal-names : mode signal-type);
end entity-name;
```

Numa entidade, mode indica qual a direcção dum porto da interface. Um **porto** é uma entrada, saída ou entrada/saída dum módulo. As possibilidades para a direcção dum porto são:

- in: entrada da entidade;
- out: saída da entidade;
- buffer: saída da entidade (o seu valor pode ser lido dentro da arquitectura);
- inout: entrada e saída da entidade.

Nesta declaração, signal-type designa um tipo de sinal predefinido ou um tipo definido pelo utilizador.

Sinal é o objecto primário utilizado para descrever sistemas, equivale a um "fio" físico e tem um historial de valores passados associado. Os sinais funcionam como canais de comunicação entre instruções concorrentes. A **sintaxe** da declaração dum **sinal** é:

signal *nome_sinal* : *tipo_sinal* := *valor_inicial_opcional* ;

Os sinais podem ser declarados explicitamente na declaração dum *package*, numa arquitectura, num bloco ou num subprograma. A declaração dum porto duma entidade é uma declaração implícita dum sinal.

Variável é similar a um sinal, mas sem o equivalente físico e sem o historial de valores.

A **sintaxe** da definição duma **arquitectura** é:

```
architecture architecture-name of entity-name is
   type declarations
   signal declarations
   constant declarations
   function definitions
   procedure definitions
   component declarations
begin
   concurrent-statement
    ...
   concurrent-statement
end architecture-name ;
```

As declarações (tipos, sinais,…,componentes) podem surgir por qualquer ordem. Em "*signal declarations*" definem-se os sinais internos à arquitectura.

4.3. Tipos

Qualquer sinal, variável e constante tem um tipo associado. O **tipo** especifica o conjunto de valores permitidos a um objecto e os operadores que podem ser aplicados a esse objecto. Esta definição de tipo equivalente a um tipo de dados abstracto, um conceito similar ao de classe em OO. VHDL é uma linguagem fortemente "tipada" mas que possui apenas os seguintes tipos predefinidos:

```
bit             character       severity_level
bit_vector      integer         string
boolean         real            time
```

- **integer** inclui os inteiros no intervalo -2.147.483.647 a +2.147.483.647
- **boolean** possui 2 valores: *true* e *false*
- **character** inclui os caracteres do conjunto ISO 8-bit.

A tabela em baixo inclui os operadores predefinidos para os tipos **integer** e **boolean**. A funcionalidade dos operadores menos triviais, rem e mod, é agora descrita:

A rem B → módulo = A – int(A/B)*B e sinal = sinal de A
A mod B → módulo = A – int(A/B)*B [quando A e B são do mesmo sinal]
módulo = A + ceil(|A/B|)*B [quando A e B são de sinais diferentes]
sinal = sinal de B

integer Operators		boolean Operators	
+	addition	and	AND
-	subtraction	or	OR
*	multiplication	nand	NAND
/	division	nor	NOR
mod	modulo division	xor	Exclusive OR
rem	modulo remainder	xnor	Exclusive NOR
abs	absolute value	not	complementation
**	exponentiation		

Os tipos definidos pelo utilizador são comuns em VHDL. Um tipo enumerado é definido através duma lista de valores permitidos.

```
type type-name is (value-list);

subtype subtype-name is type-name start to end;
subtype subtype-name is type-name start downto end;

constant constant-name : type-name := value;
```

Os 9 níveis permitidos pelo tipo **STD_LOGIC** são:

```
type STD_ULOGIC is (
   'U',   -- Uninitialized
   'X',   -- Forcing  Unknown
   '0',   -- Forcing  0
   '1',   -- Forcing  1
   'Z',   -- High Impedance
   'W',   -- Weak     Unknown
   'L',   -- Weak     0
   'H',   -- Weak     1
   '-');  -- Don't care
subtype STD_LOGIC is resolved STD_ULOGIC;
```

Outros exemplos são:

- **type** traffic_light_state **is** (reset, stop, start, go);
- **subtype** int32 **is integer range** 31 downto 0;
- **constant** BUS_SIZE: **integer** := 32;

O utilizador também pode definir **arrays** usando as palavras-chave **to**, **downto** e **range** para definir a dimensão. Um array é um conjunto ordenado de elementos do mesmo tipo.

```
type type-name is array(start to end) of element-type;

type type-name is array(start downto end) of element-type;

type type-name is array(range-type) of element-type;

type type-name is array(range-type range start to end) of element-type;

type type-name is array(range-type range start downto end) of element-type;
```

```
type monthly_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 STD_LOGIC;
constant NUM_REGS: integer := 8;
type reg_file is array (1 to NUM_REGS) of word;
type statecount is array (traffic_light_state) of integer;
```

O conteúdo dos elementos dum *array* pode ser especificado **por posição**, colocando a lista de valores a atribuir entre parênteses:

```
B := ('1','1','0','1','1','0','0','1');    -- tipo byte
```

O conteúdo do *array* também pode ser especificado **usando índices**:

```
W := (0=>'0', 3=>'0', 9=>'0',others=>'1'); -- tipo word
```

O conteúdo dum *array* STD_LOGIC pode ser especificado **usando strings**:

```
B := "11011001";
W := "0110111110111111";
```

Também se pode especificar uma parcela dum *array*:

```
B(2 to 4):="101";   W(9 downto 0):="0101011010";
```

Pode juntar-se *arrays* usando o operador de concatenação (&):

```
'0'&'1'&"1Z"          é equivalente a "011Z".
B(6 downto 0)& B(7)   equivale a rodar o array B um 1-
bit à esquerda.
```

4.4. Funções e procedimentos

Uma **função** aceita um conjunto de argumentos e devolve um resultado. Tanto os argumentos como o resultado devem ter um tipo. O corpo da função é um

conjunto de instruções executadas em sequência. A sintaxe da definição de função é:

```
function function-name(
      signal-names  : signal-type;
      signal-names  : signal-type;
        ...
      signal-names  : signal-type
)  return return-type is
   type declarations
   constant declarations
   variable declarations
   function definitions
   procedure definitions
begin
   sequential-statement
    ...
   sequential-statement
end function-name;
```

Exemplo de definição e utilização duma função ButNot:

```
architecture Inhibit_archf of Inhibit 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(X,Y);
end Inhibit_archf;
```

Normalmente é necessário converter um sinal de um tipo para outro. Usando o tipo de array

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

apresentam-se a seguir 2 funções de conversão de tipos: de **std_logic** para **inteiro** e de **inteiro** para **std_logic**.

```
function CONV_INTEGER (X: STD_LOGIC_VECTOR) return INTEGER is
  variable RESULT: INTEGER;
begin
  RESULT := 0;
  for i in X'range loop
    RESULT := RESULT * 2;
    case X(i) is
      when '0' | 'L'  => null;
      when '1' | 'H'  => RESULT := RESULT + 1;
      when others     => null;
    end case;
  end loop;
  return RESULT;
end CONV_INTEGER;
```

```
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
    return STD_LOGIC_VECTOR is
  variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
  variable temp: integer;
begin
  temp := ARG;
  for i in 0 to SIZE-1 loop
    if (temp mod 2) = 1 then result(i) := '1';
    else result(i) := '0';
    end if;
    temp := temp / 2;
  end loop;
  return result;
end;
```

Um **procedimento** é idêntico a uma função, mas não devolve um resultado. Enquanto a invocação duma função pode ser usada em vez duma expressão, a invocação dum procedimento pode ser usada em vez duma instrução. Como os argumentos dum procedimento podem ter uma direcção do tipo **out** ou **inout**, é possível um procedimento "devolver" resultado(s).

4.5. Bibliotecas e packages

Uma **biblioteca** é o local onde o compilador de VHDL guarda a informação relativa a um determinado projecto (intermédia, da simulação e da síntese). Para qualquer projecto, o compilador cria e utiliza a biblioteca **work**. Um projecto pode usar múltiplos ficheiros, cada um com unidades (entidades / arquitecturas) diferentes. Quando um ficheiro é compilado, os resultados são guardados na biblioteca work. Nem toda a informação necessária a um projecto deve estar na biblioteca work. O projectista pode recorrer a definições ou funções que são comuns a vários projectos (por exemplo, incluídas na **biblioteca IEEE**). Um projecto pode especificar que vai usar bibliotecas que contêm definições partilháveis.

Um exemplo: **library ieee;**

A especificação duma biblioteca permite aceder a todas as entidades e arquitecturas guardadas nessa biblioteca, mas não dá acesso aos tipos, subtipos, funções, procedimentos,... Um ***package*** é um ficheiro com definições de objectos (sinais, tipos, constantes, funções, procedimentos, componentes) que podem ser utilizados nos projectos. A cláusula seguinte permite a um projecto usar todas as definições do **package standard 1164 do IEEE**:

```
use ieee.std_logic_1164.all;
```

A sintaxe da definição dum *package* é:

```
package package-name is
   type declarations
   signal declarations
   constant declarations
   component declarations
   function declarations
   procedure declarations
end package-name ;
package body package-name is
   type declarations
   constant declarations
   function definitions
   procedure definitions
end package-name ;
```

4.6. Projecto estrutural

O corpo duma arquitectura é uma série de **instruções concorrentes**. Cada instrução concorrente é executada em simultâneo com as outras instruções concorrentes incluídas no mesmo corpo de arquitectura. As instruções concorrentes são necessárias para simular o modo paralelo em que os elementos de *hardware* funcionam. A instrução concorrente mais elementar é a instanciação dum **componente**. Apresenta-se a seguir a sintaxe da instanciação dum componente:

```
label: component-name port map (signal1, signal2, ..., signaln);

label: component-name port map (port1=>signal1, port2=>signal2, ..., portn=>signaln);
```

A primeira linha da figura anterior estabelece um mapeamento entre portos e sinais externos **por posição**, enquanto a segunda linha define um mapeamento

explícito entre portos e sinais externos. Ainda nesta figura, component-name é o nome duma entidade definida anteriormente.

Por cada instanciação dum componente é criada uma instância da entidade respectiva. Antes de instanciar um componente ele tem que ser declarado na parte declarativa da arquitectura usando o construtor **component**. A declaração dum componente é essencialmente o mesmo que a declaração da interface da entidade correspondente. A sintaxe da declaração dum componente é:

```
component component-name
   port (signal-names  : mode signal-type;
         signal-names  : mode signal-type;
           ...
         signal-names  : mode signal-type);
end component;
```

Os componentes usados numa arquitectura podem ter sido definidos anteriormente no projecto em causa, ou podem estar definidos numa biblioteca.

O exemplo que se segue apresenta a descrição estrutural do detector de números primos.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prime is
    port ( N: in STD_LOGIC_VECTOR (3 downto 0); P: out STD_LOGIC );
end prime;
architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L: STD_LOGIC;
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;                    (i)
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O:out STD_LOGIC);end component;   (ii)
begin
  U1: INV port map (N(3), N3_L);
  U2: INV port map (N(2), N2_L);
  U3: INV port map (N(1), N1_L);
  U4: AND2 port map (N3_L, N(0), N3L_N0);
  U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
  U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
  U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
  U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, P);              (iii)
end prime1_arch;
```

(i) sinais internos;
(ii) declaração de componentes;
(iii) instanciação de componentes.

Uma arquitectura que utiliza componentes é uma **descrição estrutural**, uma vez que descreve a estrutura de interligação entre os sinais e as entidades que concretizam essa entidade. A instrução **generate** permite criar estruturas repetitivas. A sintaxe do ciclo **for…generate** é:

```
label: for identifier in range generate
         concurrent-statement
       end generate;
```

identifier – é uma variável implicitamente declarada.

Exemplo de um inversor de 8 bits descrito com um ciclo for … generate.

```
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 INV port (I: in STD_LOGIC; O:out STD_LOGIC); end component;
begin
  g1: for b in 1 to 8 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end inv8_arch;
```

Pode definir-se **constantes genéricas** na declaração duma entidade. A próxima figura mostra a sintaxe da declaração duma entidade usando constantes genéricas. As constantes genéricas permitem definir uma entidade parametrizada.

```
entity entity-name is
  generic (constant-names : constant-type;
           constant-names : constant-type;
             ...
           constant-names : constant-type);
  port (signal-names  : mode signal-type;
        signal-names  : mode signal-type;
          ...
        signal-names  : mode signal-type);
end entity-name;
```

Cada constante pode ser usada dentro da arquitectura em que é declarada e a atribuição dum valor a essa constante ocorre apenas quando a entidade for instanciada noutra arquitectura. Ao instanciar um componente, para atribuir valores às constantes genéricas utiliza-se uma cláusula **generic map**.

Circuito inversor dos bits dum barramento de largura parametrizável (**WID**) e descrito com um ciclo **for … generate** (ver figura).

```
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 INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in WID-1 downto 0 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end businv_arch;
```

Descreve-se agora a entidade que instancia o inversor de barramento com 3 valores de WID: 8,16 e 32,

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

entity businv_example 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 businv_example;

architecture businv_ex_arch of businv_example 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 businv_ex_arch;
```

4.7. Projecto fluxo de dados (dataflow)

Se uma arquitectura emprega apenas instruções concorrentes, o que ela descreve é o fluxo de dados e as operações que são aplicadas aos dados dentro do circuito. Este estilo de descrição é designado por **fluxo de dados**. Dois tipos de instrução concorrente que se usam numa descrição fluxo de dados são:

- Atribuição concorrente dum valor a um sinal – a largura e o tipo de ambos os lados da instrução têm que ser compatíveis;
- Atribuição concorrente e condicional dum valor a um sinal.

A sintaxe dos 2 tipos de atribuição concorrente é:

```
signal-name <= expression;

signal-name <= expression when boolean-expression else
               expression when boolean-expression else
                  ...
               expression when boolean-expression else
               expression;
```

Exemplo de uma descrição fluxo de dados para o detector de números primos, usando operadores predefinidos como **and**, **or**, **not** e **atribuições simples.**

```
architecture prime2_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0      <= not N(3)                                and N(0);
  N3L_N2L_N1  <= not N(3) and not N(2) and     N(1)          ;
  N2L_N1_N0   <=              not N(2) and     N(1) and N(0);
  N2_N1L_N0   <=                  N(2) and not N(1) and N(0);
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime2_arch;
```

Exemplo de uma descrição fluxo de dados usando **atribuições condicionais**.

```
architecture prime3_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0     <= '1' when N(3)='0' and N(0)='1' else '0';
  N3L_N2L_N1 <= '1' when N(3)='0' and N(2)='0' and N(1)='1' else '0';
  N2L_N1_N0  <= '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0';
  N2_N1L_N0  <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime3_arch;
```

para cobrir todas as combinações restantes

Outro tipo de instrução concorrente é a **atribuição selectiva a um sinal**, idêntica a um construtor CASE. A sintaxe da atribuição selectiva é:

```
with expression select
  signal-name <= signal-value when choices,
                 signal-value when choices,
                    . . .
                 signal-value when choices;
```

Exemplo de uma descrição fluxo de dados para o detector de números primos usando uma **atribuição selectiva ao sinal F**. As várias escolhas devem ser mutuamente exclusivas e cobrir todos os casos.

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

escolha simples

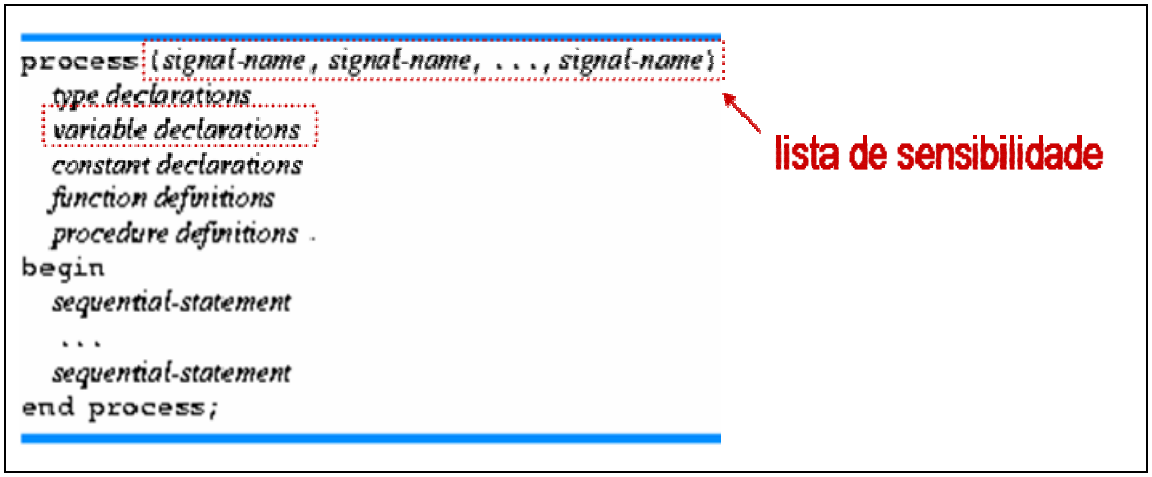
lista

O código seguinte constitui outra descrição fluxo de dados para o mesmo detector.

```
architecture prime5_arch of prime is
begin
  with CONV_INTEGER(N) select
    F <= '1' when 1|2|3|5|7|11|13,
         '0' when others;
end prime5_arch;
```

4.8. Projecto comportamental (ou funcional)

O principal construtor usado em descrições comportamentais é o **processo**, o qual consiste numa série de instruções sequenciais que são executadas em paralelo com outras instruções e processos concorrentes. Um processo tem um tempo de simulação nulo. Um processo em VHDL é assim uma instrução concorrente, com sintaxe:



Um processo não pode declarar sinais, apenas variáveis, utilizadas para guardar informação relativa ao estado do processo. A sintaxe da definição duma variável é:

variable *nome_variavel* : *tipo_variavel*;

Um processo em VHDL está num de 2 estados: em execução ou suspenso. A lista de sinais incluída na definição dum processo (**lista de sensibilidade**) determina quando é que ele é executado. Um processo está inicialmente suspenso. Quando um sinal da lista de sensibilidade muda de valor, o processo retoma a execução, desde a 1ª instrução até à última. Se um sinal da lista de sensibilidade mudar de valor durante a execução do processo, este será executado outra vez. A execução continua até o processo terminar a execução sem que nenhum destes sinais tenha mudado de valor. Na simulação, a execução do corpo dum processo (até ele ser suspenso) decorre num tempo de simulação nulo. Após ter retomado a execução, um processo correctamente escrito será suspenso ao fim de algumas execuções. Contudo, é possível escrever (incorrectamente) um processo que nunca é suspenso.
Um exemplo: um processo com uma única instrução "X <= not X;" e uma lista de sensibilidade igual a "(X)". Como X muda em cada execução, o processo executa indefinidamente num tempo de simulação nulo. Na prática, os simuladores conseguem detectar estas situações e terminar a simulação.

A instrução de atribuição sequencial dum valor a um sinal possui a mesma sintaxe que a versão concorrente, mas ocorre no corpo dum processo em vez de numa arquitectura:

***nome-sinal* <= expressão;**

A instrução de atribuição dum valor a uma variável possui a seguinte sintaxe:

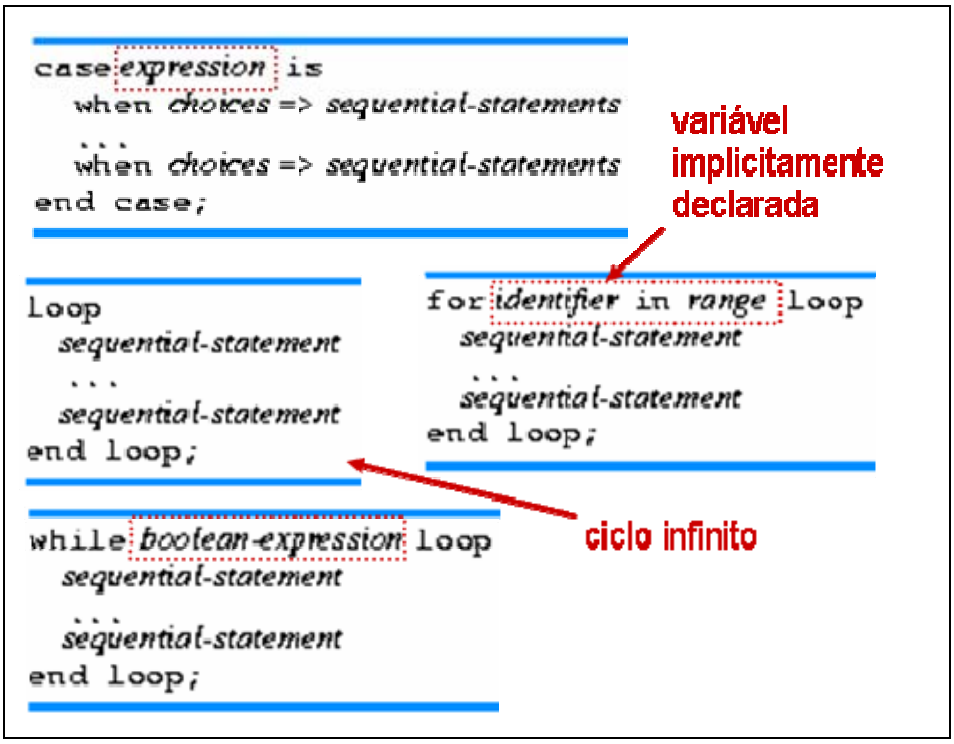
***nome-variavel* := expressão;**

Exemplo de uma descrição comportamental para o detector de números primos, em que se usa um processo com atribuições a variáveis.

```
architecture prime6_arch of prime6 is
begin
  process(N)
    variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  begin
    N3L_N0      := not N(3)                                and N(0);
    N3L_N2L_N1  := not N(3) and not N(2) and     N(1)           ;
    N2L_N1_N0   :=               not N(2) and     N(1) and N(0);
    N2_N1L_N0   :=                   N(2) and not N(1) and N(0);
    P <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
  end process;
end prime6_arch;
```

Além da atribuição, pode usar-se outras instruções sequenciais, descritas por alguns dos construtores mais populares, tais como: **if**, **case**, **loop**, **for** e **while**. O **case** é mais legível que um **if** com múltiplas cláusulas if/elsif e pode ser sintetizado de forma mais adequada. As 2 figuras seguintes incluem a sintaxe destes construtores.

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
else sequential-statement
end if;

if boolean-expression  then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
else sequential-statement
end if;
```



No caso do case, as várias escolhas devem ser mutuamente exclusivas e cobrir todos os casos.

Exemplos da descrição comportamental para o detector de números primos usando um construtor **if** ou **case,** necessariamente incluídos num processo.

```
architecture prime7_arch of prime is
begin
  process(N)
    variable NI: INTEGER;
  begin
    NI := CONV_INTEGER(N);
    if NI=1 or NI=2 then P <= '1';
    elsif NI=3 or NI=5 or NI=7 or NI=11 or
          NI=13 then P <= '1';
    else P <= '0';
    end if;
  end process;
end prime7_arch;
```

```
architecture prime8_arch of prime is
begin
  process(N)
  begin
    case CONV_INTEGER(N) is
      when 1 => P <= '1';
      when 2 => P <= '1';
      when 3 | 5 | 7 | 11 | 13 => P <= '1';
      when others => P <= '0';
    end case;
  end process;
end prime8_arch;
```

Exemplo da descrição comportamental para o detector de números primos usando um ciclo **for, em que** o tamanho da entrada - N - passou para 8 bits.

```
architecture prime9_arch of prime9 is
begin
  process(N)
  variable NI: INTEGER;
  variable prime: boolean;
  begin
    NI := CONV_INTEGER(N);
    prime := true;
    if NI=1 or NI=2 then null; -- boundary cases
    else for i in 2 to 253 loop
           if NI mod i = 0 then
             prime := false; exit;
           end if;
         end loop;
    end if;
    if prime then P <= '1'; else P <= '0'; end if;
  end process;
end prime9_arch;
```

4.9. Dimensão temporal

Nenhum dos exemplos anteriores aborda a dimensão temporal associada com o funcionamento dos circuitos: tudo se passa num tempo de simulação nulo. O VHDL modela adequadamente os aspectos relacionados com tempo. Em VHDL pode especificar-se um atraso usando a palavra-chave **after** na instrução de atribuição dum valor a um sinal.

```
Z <= '1' after 4ns when X='1' else
     '0' after 3ns;
```

Esta instrução modela uma porta que apresenta um atraso de 4ns quando a saída Z muda de 0→1 e apenas 3ns quando muda de 1→0. Deste modo, ao

simular o modelo dum circuito obtém-se uma boa aproximação para o seu comportamento temporal.
Outra forma de incorporar informação temporal numa descrição VHDL consiste em usar a instrução **wait**. Esta instrução sequencial pode ser usada para suspender um processo durante um intervalo determinado. É frequente usar a instrução wait para ajudar a descrever os padrões que se aplicam nas entradas dum circuito que está a ser simulado. Um exemplo:

```
entity InhibitTestBench is
end InhibitTestBench;

architecture InhibitTB_arch of InhibitTestBench is
component Inhibit port (X,Y: in BIT; Z: out BIT); end component;
signal XT, YT, ZT: BIT;
begin
  U1: Inhibit port map (XT, YT, ZT);
  process
  begin
    XT <= '0'; YT <= '0';
    wait for 10 ns;
    XT <= '0'; YT <= '1';
    wait for 10 ns;
    XT <= '1'; YT <= '0';
    wait for 10 ns;
    XT <= '1'; YT <= '1';
    wait; -- this suspends the process indefinitely
  end process;
end InhibitTB_arch;
```

A instrução **wait** pode assumir uma de várias variantes:

wait;
wait on lista_sinais;
wait until condição;
wait for tempo;

4.10. Simulação

A partir do momento em que se dispõe duma descrição VHDL sem erros de sintaxe e de semântica, pode aplicar-se a descrição num simulador para verificar se o circuito descrito funciona como esperado. O processo de simulação arranca no instante zero de simulação. Neste instante, o simulador inicia todos os sinais com o seu valor por defeito. Também se iniciam os sinais e variáveis para os quais o código VHDL declare explicitamente valores iniciais.
A seguir, o simulador começa a executar "todos" os processos (e instruções concorrentes) da descrição do circuito. Para simular a execução de todos os processos, o simulador usa (i) uma lista com eventos calendarizados (com ocorrência temporal escalonada) e (ii) uma matriz com os sinais a que cada processo é sensível. No instante zero de simulação todos os processos são escalonados para serem executados. Selecciona-se um dos processos e executam-se todas as suas instruções sequenciais, incluindo os ciclos. Quando a execução deste processo terminar, escolhe-se outro para execução e repete-

se este procedimento até que todos os processos tenham sido executados. Completa-se assim um **ciclo de simulação**. Durante a execução, um processo pode atribuir novos valores a sinais. Os novos valores não são atribuídos imediatamente. Antes são colocados na lista de eventos e a sua efectivação é escalonada para um tempo determinado.

Se uma atribuição tiver um tempo de simulação explícito (usando a cláusula after, por exemplo), então ela será colocada na lista de eventos e escalonada para que ocorra ao fim desse tempo. Caso contrário, a atribuição é concretizada "imediatamente". Na realidade a atribuição é escalonada para ocorrer num instante que é dado pela soma do tempo actual com um atraso minúsculo (***delta delay***). O *delta delay* é um intervalo infinitamente curto, que mesmo somando ao tempo de simulação actual um número qualquer de *delta delay*'s ainda se obtém o tempo de simulação actual. O conceito de *delta delay* permite que um processo seja executado múltiplas vezes (se necessário) num tempo de simulação nulo. Quando se completa um ciclo de simulação, percorre-se a lista de eventos em busca dos sinais em que vai ocorrer a alteração mais próxima. A proximidade pode ser tão diminuta quanto um *delta delay*, ou então ser um atraso real. No 2º caso o tempo de simulação avança. Em qualquer dos casos, as alterações num sinal, escalonadas para esse instante, serão efectuadas. Os processos que forem sensíveis às alterações do sinal que acaba de mudar serão escalonados para execução no próximo ciclo de simulação. A simulação evoluirá indefinidamente até a lista de eventos estar vazia.

O mecanismo da lista de eventos permite simular o funcionamento de processos concorrentes num sistema uni-processador e garante um funcionamento correcto mesmo quando os processos exigem várias execuções antes de estabilizar.

Exemplo com 2 processos:

```
process (x,y)
begin
   x <= '1' after 10 ns;
   y <= x   after  3 ns;
end process;

process
begin
   z <= x or y;
   wait for 5ns;
end process;
```

A próxima figura contém o código do *testbench* para ALU de 1 bit.

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

entity testAlu1bit is
end entity test_alu1bit;

architecture tst of testAlu1bit is

  component alu1bit is
    port (
    a, b, c  : in std_logic;
    sel      : in std_logic_vector
               (1 downto 0);
    res, f   : out std_logic);
  end component alu1bit;

  signal i1  : std_logic := '0';
  signal i2  : std_logic := '0';
  signal ci  : std_logic := '0';
  signal op  : std_logic_vector
               (1 downto 0) := "00";
  signal res : std_logic;
  signal co  : std_logic;

begin
  -- instanciar o sistema
  -- a testar
  ALU1: alu1bit
    port map (
      a      => i1  ,
      b      => i2  ,
      c      => ci  ,
      sel    => op  ,
      res    => res ,
      f      => co  );
```

```
  process (i1) is
  begin
    if i1='1' then
      i1 <= '0' after 10ns;
    elsif i1='0' then
      i1 <= '1' after 10ns;
    end if;
  end process;

  process (i2) i
  begin
    if i2='1' then
      i2 <= '0' after 20ns;
    elsif i2='0' then
      i2 <= '1' after 20ns;
    end if;
  end process;

  process (ci) is
  begin
    if ci='1' then
      ci <= '0' after 40ns;
    elsif ci='0' then
      ci <= '1' after 40ns;
    end if;
  end process;

  process (op) is
  begin
    if op="00" then
      op <= "01" after 80ns;
    elsif op="01" then
      op <= "10" after 80ns;
    elsif op="10" then
      op <= "11" after 80ns;
    elsif op="11" then
      op <= "00" after 80ns;
    end if;
  end process;
end architecture;
```

4.11. Síntese

O VHDL foi pensado para ser uma linguagem de descrição e simulação. Mais tarde, foi também adoptado no processo de síntese. A linguagem possui várias características e construtores NÃO sintetizáveis. O subconjunto da linguagem e o estilo de programas apresentados até aqui são em boa medida sintetizáveis pela maior parte das ferramentas comerciais.

O código que descreve um circuito pode ter um impacto enorme na qualidade do circuito que se consegue sintetizar. Como exemplo pode dar-se o das estruturas de controlo encadeadas, do tipo if-elsif-elsif-else. Estas estruturas podem originar uma série de portas lógicas em cadeia usadas para testar as várias condições. É preferível usar um case ou uma atribuição selectiva, desde que as condições de selecção (*choices*) sejam mutuamente exclusivas.

Os ciclos dentro de processos são geralmente desdobrados para criar várias cópias da lógica combinacional que executa as instruções do corpo do ciclo. Quando se pretende apenas uma cópia da lógica combinacional a executar as instruções do corpo do ciclo, é necessário especificar um circuito sequencial.

Quando um processo inclui instruções condicionais que não cobrem todas as combinações das entradas, o compilador sintetiza uma *latch* para guardar o valor da(s) saída(s) nos casos não cobertos. Geralmente, estas *latches* não são desejadas.

Algumas das características e construtores que não são sintetizáveis em todas as ferramentas são: memória dinâmica, ficheiros e apontadores.

5. Questões Práticas de Sistemas Combinacionais

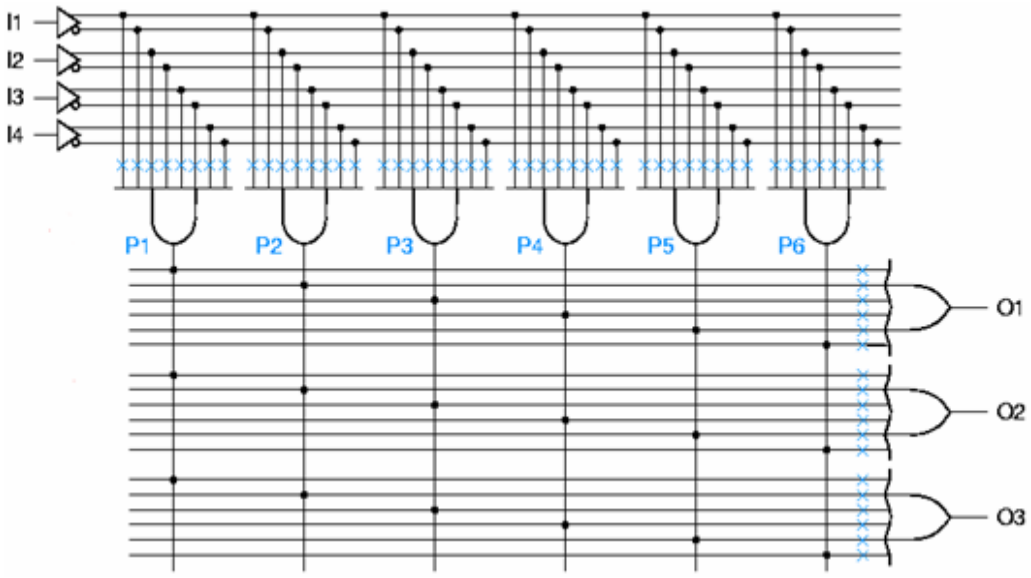
5.1. PLDs (dispositivos de lógica programável)

Os primeiros PLDs a surgir foram os **Programmable Logic Arrays** (PLAs). Uma PLA é um dispositivo combinacional, com estrutura a 2-níveis AND-OR, que pode ser programado para efectuar qualquer expressão lógica do tipo soma-de-produtos. O tipo de expressão implementável numa PLA está limitada pelo:

- Número de entradas da PLA (**n**)
- Número de saídas da PLA (**m**)
- Número de termos de produto da PLA (**p**)

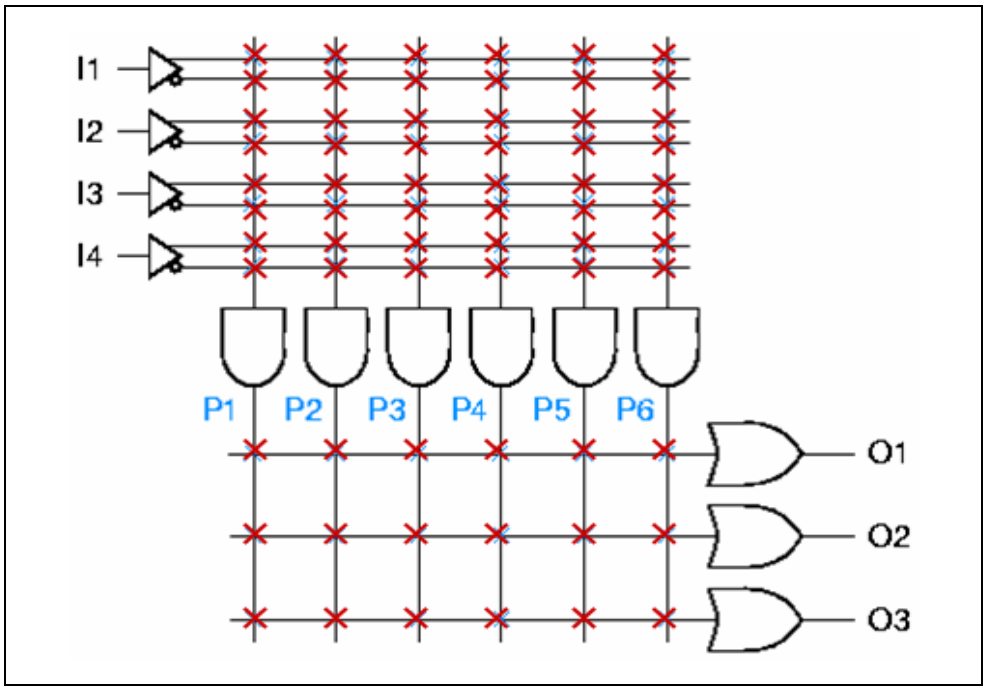
O dispositivo é definido como "uma PLA **n x m** com **p** termos de produto". Normalmente, p << 2^n. Uma PLA **n x m** com **p** termos de produto contém **p** ANDs de **2n**-entradas e **m** ORs de **p**-entradas.

Cada entrada *In* está ligada a um *buffer* que gera uma cópia desse sinal (*In*) e o seu complemento (/*In*). As ligações possíveis de estabelecer estão assinaladas com **X**s. O dispositivo é programado estabelecendo as ligações necessárias. As ligações são feitas através de fusíveis (células memória).



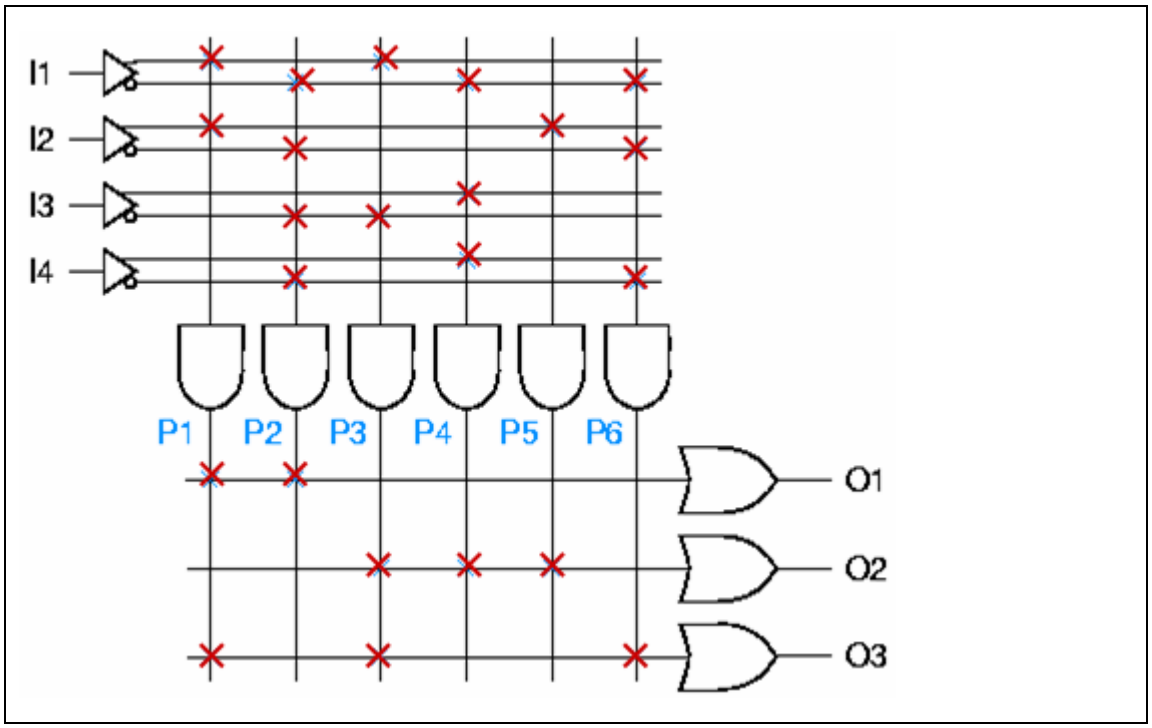
Uma PLA **4x3** com **6** termos de produto.

A próxima figura mostra uma representação mais compacta da PLA **4x3** com **6** termos de produto.



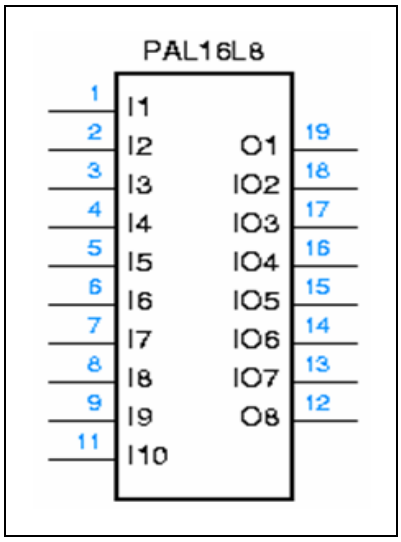
Um exemplo de utilização da PLA:

```
O1 = P1+P2       = I1·I2  + I1'·I2'·I3'·I4'
O2 = P3+P4+P5    = I1·I3' + I1'·I3·I4 + I2
O3 = P1+P3+P6    = I1·I2  + I1·I3'    + I1'·I2'·I4'
```

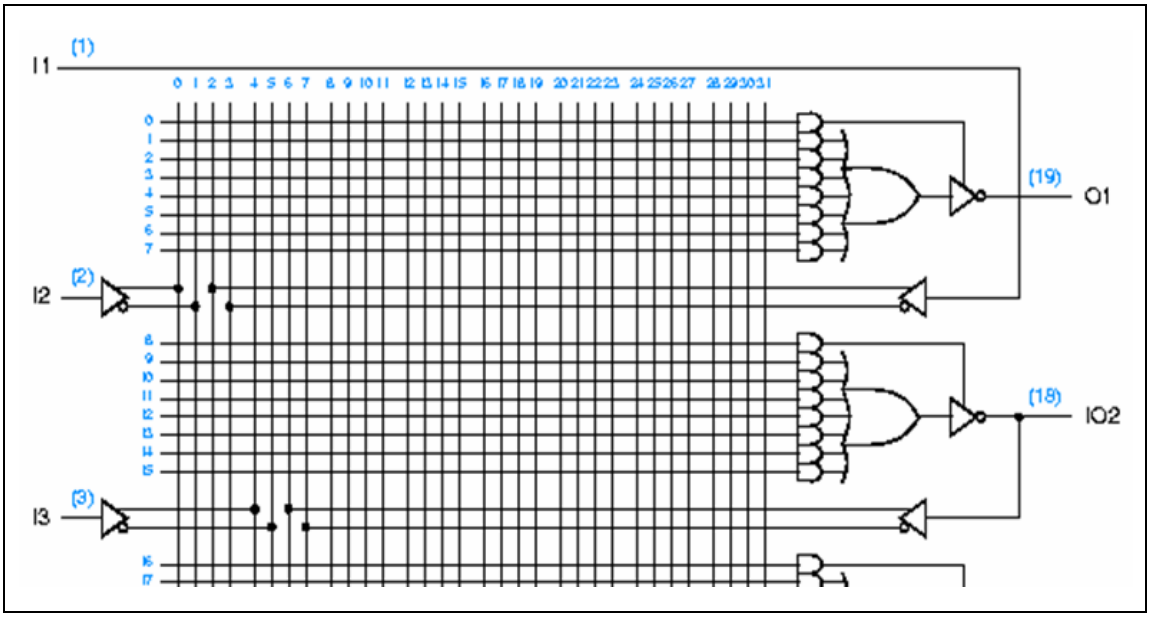


Outro tipo de PLD, que mais não é do que um caso particular de PLA, é o **Programmable Array Logic** (PAL). Um dispositivo PAL tem um *array* de ORs fixo. Numa PAL, os termos de produto não são partilhados por várias saídas.

Cada saída dispõe dum conjunto único e fixo de termos de produto que pode usar. Uma PAL é normalmente mais rápida do que uma PLA equivalente.



Apresenta-se a seguir uma vista parcial do diagrama lógico da PAL **16L8**.



A PAL 16L8 possui 10 entradas *I1..I10*, 2 saídas *O1 e O8*, 6 entradas/saídas *IO2..IO7*, *64*32* fusíveis e cada OR recebe 7 termos de produto.

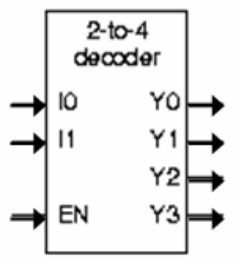
5.2. Descodificadores

Um **descodificador** é um circuito que converte uma entrada codificada numa saída codificada, sendo os códigos de entrada e saída diferentes. Habitualmente, o código de entrada tem menos bits que o de saída. O descodificador mais comum é o descodificador n-para-2^n ou binário. Usa-se um descodificador binário quando é preciso activar uma saída de entre $\mathbf{2^n}$ possíveis, com base no valor duma entrada de **n**-bits.

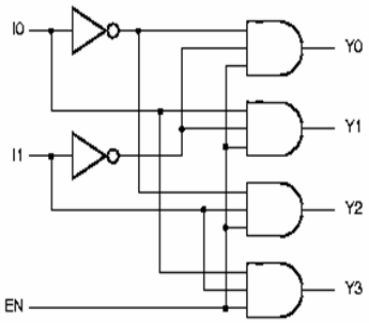
Descodificador 2-para-4:

Inputs			Outputs			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Tabela de verdade.



Símbolo.



Esquemático.
(por exemplo, Y0=EN./I1./I0)

Um CI 74x139 contém dois descodificadores 2-para-4 independentes.

Símbolo:

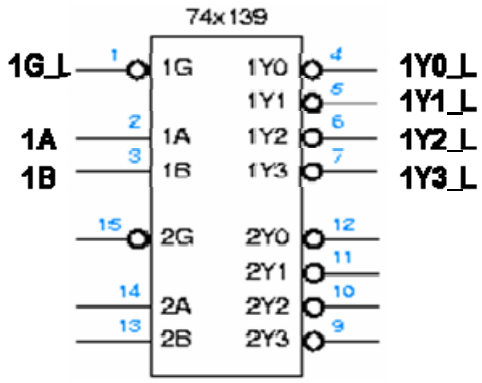
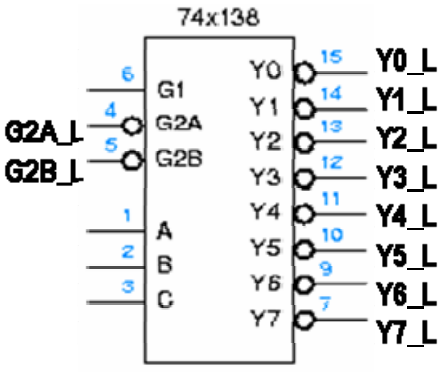


Tabela de verdade para um dos descodificadores:

Inputs			Outputs			
G_L	B	A	Y3_L	Y2_L	Y1_L	Y0_L
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

Os sinais com bolha são activos a zero
(anexa-se **_L** ao nome para indicar esse facto).

UM CI 74x138 contém um descodificador 3-para-8:

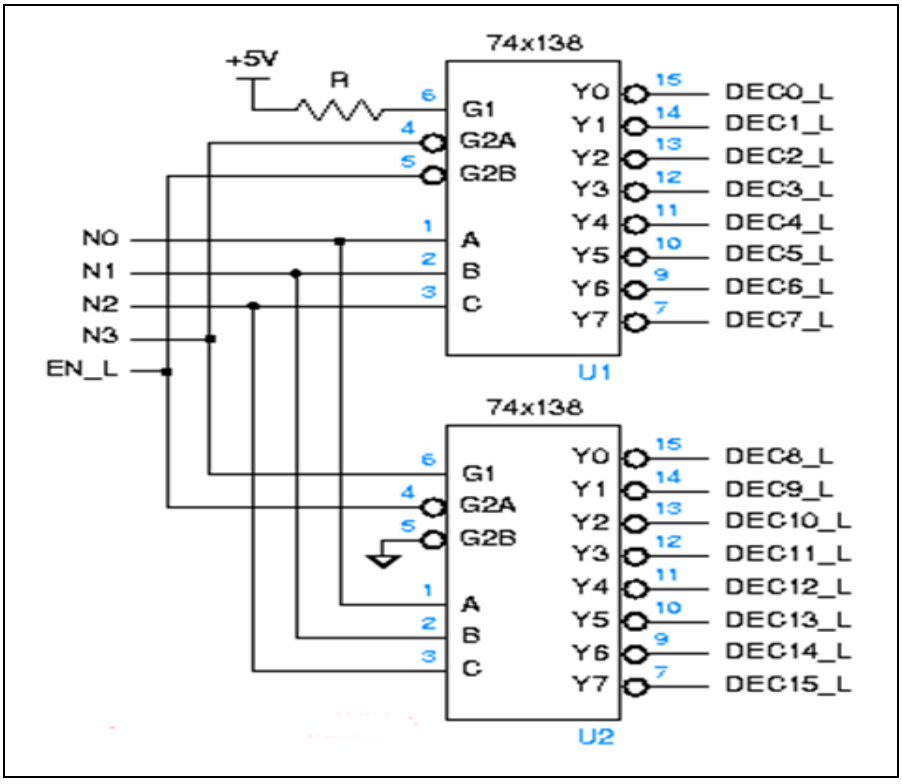


Símbolo do 74x138.

Inputs						Outputs							
G1	G2A_L	G2B_L	C	B	A	Y7_L	Y6_L	Y5_L	Y4_L	Y3_L	Y2_L	Y1_L	Y0_L
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

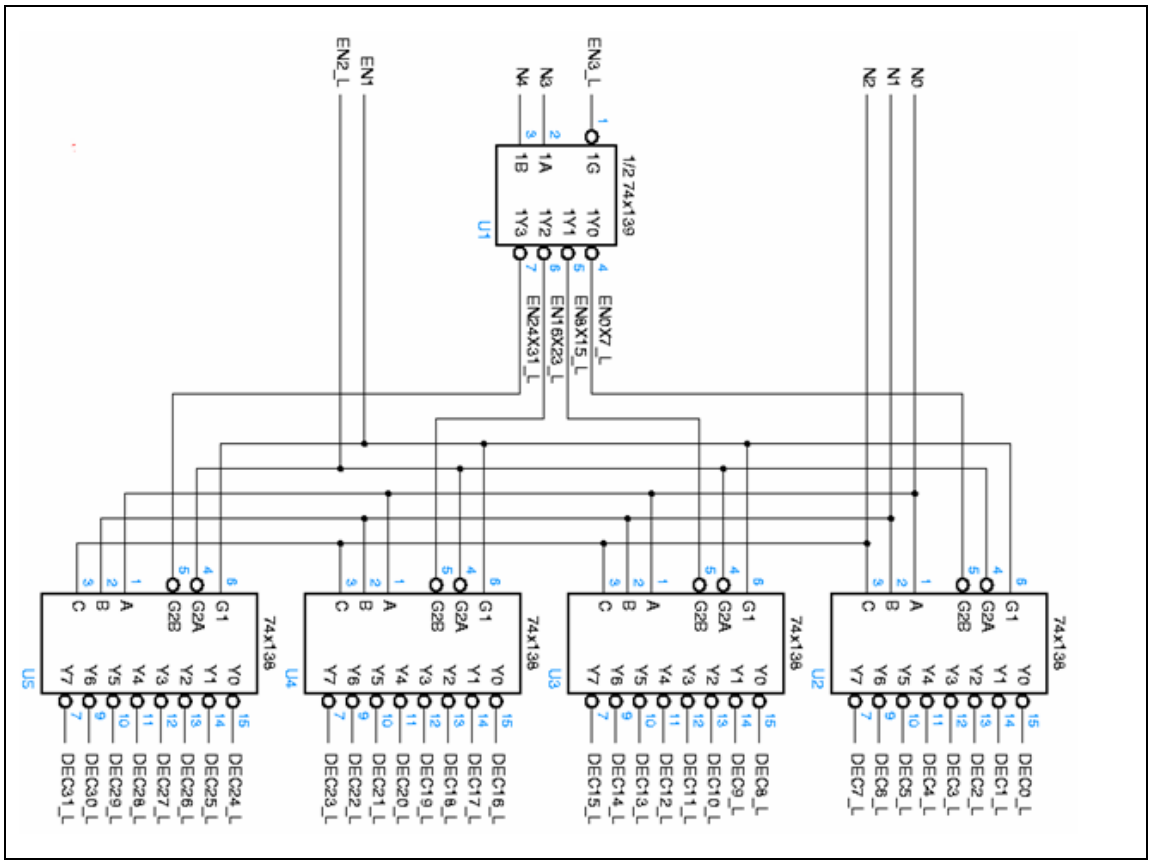
Tabela de verdade do descodificador do 74x138.

Utilizando vários descodificadores é possível descodificar entradas com mais bits. Por exemplo, pode utilizar-se 2 descodificadores 3-para-8 para construir um descodificador 4-para-16 (ver figura em baixo). O descodificador de cima (U1) só está habilitado a funcionar quando N3=0 e o descodificador de baixo (U2) quando N3=1. EN_L=0 habilita ambos.



Para descodificar entradas ainda com mais bits, pode usar-se uma hierarquia de descodificadores. Por exemplo, pode construir-se um descodificador 5-para-32 com um descodificador 2-para-4 e quatro 3-para-8 (ver figura seguinte). O descodificador 2-para-4 descodifica os 2 bits mais significativos, gerando 4

enables para os descodificadores 3:8. Os 4 descodificadores 3-para-8 descodificam os 3 bits menos significativos (0:7, 8:15, 16:23 e 24:31).



Um descodificador pode ser descrito em VHDL de várias formas. A forma mais primitiva *(e pouco legível)* consistiria em usar uma descrição **estrutural** equivalente ao circuito lógico da página 67.

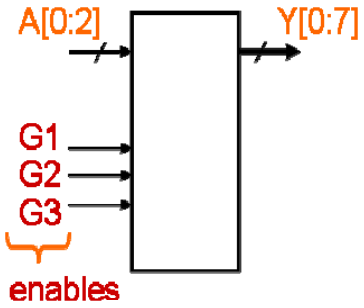
```
library IEEE;
use IEEE.std_logic_1164.all;
entity V2to4dec is
  port (I0, I1, EN: in STD_LOGIC;
        Y0, Y1, Y2, Y3: out STD_LOGIC );
end V2to4dec;
architecture V2to4dec_s of V2to4dec is
  signal NOTI0, NOTI1: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (I0,NOTI0);
  U2: inv port map (I1,NOTI1);
  U3: and3 port map (NOTI0,NOTI1,EN,Y0);
  U4: and3 port map (   I0,NOTI1,EN,Y1);
  U5: and3 port map (NOTI0,   I1,EN,Y2);
  U6: and3 port map (   I0,   I1,EN,Y3);
end V2to4dec_s;
```

A segunda alternativa (*mais legível*) consistiria em usar uma descrição **fluxo de dados**. Como exemplo, apresenta-se o **descodificador 3:8 com enable**.

```
entity decoder3to8 is
    port (
     A  : in  std_logic_vector(2 downto 0);
     EN : in  std_logic;
     Y  : out std_logic_vector (7 downto 0)  );
end entity decoder3to8;
```

```
architecture dataFlow of decoder3to8 is
    signal Y_i : std_logic_vector (7 downto 0) ;  -- sinal auxiliar
    begin
    with A select Y_i <=
            "00000001"  when  "000",
            "00000010"  when  "001",
            "00000100"  when  "010",
            "00001000"  when  "011",
            "00010000"  when  "100",
            "00100000"  when  "101",
            "01000000"  when  "110",
            "10000000"  when  "111",
            "00000000"  when  others;
    Y <= Y_i when EN='1' else "00000000";
end architecture dataFlow;
```

Outra alternativa para o descodificador 3:8 seria uma descrição **comportamental**. Neste caso, a entrada *A*, os 3 enables e a saída *Y* são todos activos a 1 (ver figura).

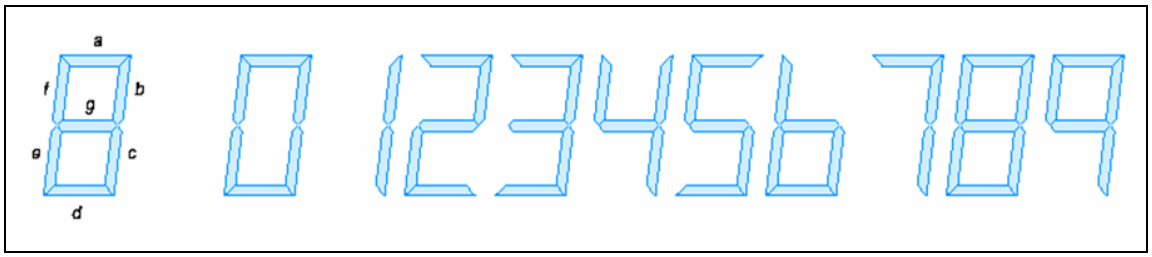



```
architecture V3to8dec_c of V3to8dec is
begin
process (G1, G2, G3, A)
  variable i: INTEGER range 0 to 7;
  begin
    Y <= "00000000";
    if (G1 and G2 and G3) = '1' then
      for i in 0 to 7 loop
        if i=CONV_INTEGER(A) then Y(i) <= '1'; end if;
      end loop;
    end if;
  end process;
end V3to8dec_c;
```

5.3. Descodificadores de 7-segmentos

Os **visores de 7-segmentos** usam-se em relógios, calculadoras e outros instrumentos que precisem mostrar informação digital. Um dígito é desenhado iluminando alguns dos 7 segmentos A a G.

Um **descodificador de 7-segmentos** recebe como entrada um dígito BCD com 4-bits e tem como saídas os 7 sinais que fazem iluminar cada um dos segmentos.



Descodificador 7-segmentos do CI 74x49:

Inputs					Outputs						
BI_L	D	C	B	A	a	b	c	d	e	f	g
0	x	x	x	x	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	1	0	1	1	0	1	1	0	1
1	0	0	1	1	1	1	1	1	0	0	1
1	0	1	0	0	0	1	1	0	0	1	1
1	0	1	0	1	1	0	1	1	0	1	1
1	0	1	1	0	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1	0	0	0	0
1	1	0	0	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	0	0	1	1
1	1	0	1	0	0	0	0	1	1	0	1
1	1	0	1	1	0	0	1	1	0	0	1
1	1	1	0	0	0	1	0	0	0	1	1
1	1	1	0	1	1	0	0	1	0	1	1
1	1	1	1	0	0	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0

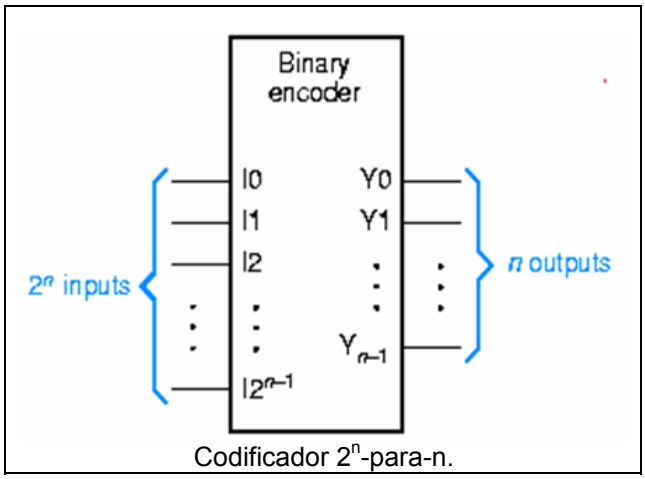
BI_L: permite que os segmentos iluminem (BI_L=1) ou não (BI_L=0)

TPC 1:
Obter as expressões minimizadas para as saídas do descodificador de 7-segmentos ao lado usando mapas de Karnaugh *(sem BI_L).*

TPC 2:
Escrever em VHDL uma descrição fluxo de dados para o descodificador de 7-segmentos.

5.4. Codificadores

Um **codificador** é um circuito que codifica uma entrada numa saída com um número de bits normalmente inferior ao da entrada. O codificador mais simples de construir é o codificador **2^n-para-n** ou **binário**. A sua funcionalidade é oposta à do descodificador binário.



Codificador 2^n-para-n.

Codificador 8-para-3:

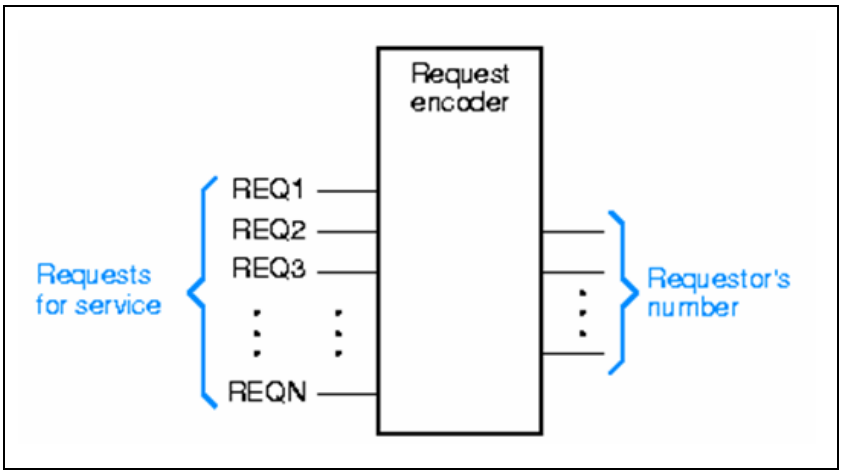
Assume-se que apenas uma entrada está activa em cada instante. As equações das saídas do codificador são:

Y2 = I4 + I5 + I6 + I7
Y1 = I2 + I3 + I6 + I7
Y0 = I1 + I3 + I5 + I7

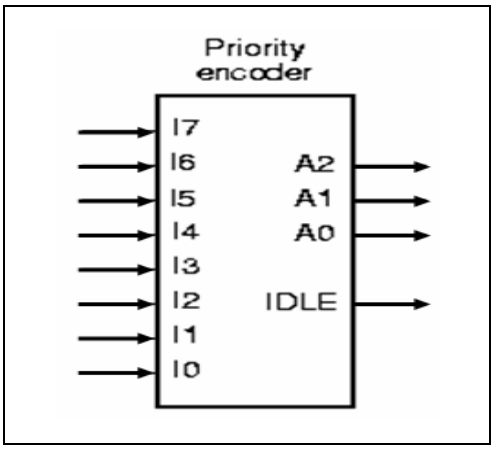
I7 I6 I5 I4 I3 I2 I1 I0	Y2 Y1 Y0
0 0 0 0 0 0 0 1	0 0 0
0 0 0 0 0 0 1 0	0 0 1
0 0 0 0 0 1 0 0	0 1 0
0 0 0 0 1 0 0 0	0 1 1
0 0 0 1 0 0 0 0	1 0 0
0 0 1 0 0 0 0 0	1 0 1
0 1 0 0 0 0 0 0	1 1 0
1 0 0 0 0 0 0 0	1 1 1

Q: O que acontece se 2 entradas estiverem activas, por ex. I2 e I4? Porquê?

Para implementar um **codificador de pedidos** (*request encoder*), o codificador binário não funciona porque assume que apenas uma entrada está activa.



Se for possível efectuar múltiplos pedidos em simultâneo, deve atribuir-se uma prioridade a cada sinal de entrada REQ_i . Quando se efectuam múltiplos pedidos, o dispositivo (**codificador de prioridade**) coloca na saída o número do pedido mais prioritário.



A entrada *I7* é a que possui maior prioridade. Nas saídas *A2:A0* é colocado o número da entrada mais prioritária activa, se houver alguma activa. A saída *IDLE* indica se há (*IDLE=0*) ou não alguma entrada activa (*IDLE=1*). Para obter as expressões das saídas, usam-se as variáveis intermédias *H0:H7*. *Hi* é 1, se *Ii* for a entrada a 1 mais prioritária:

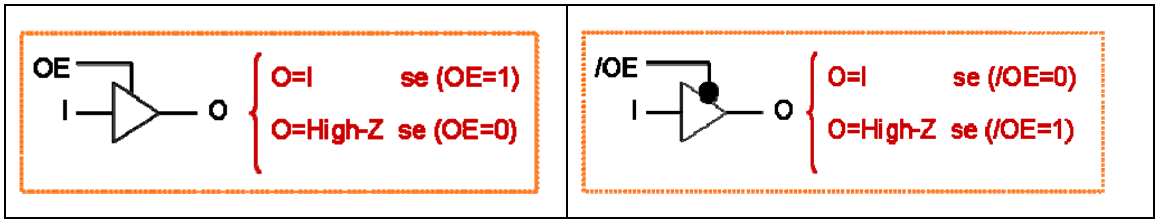
H7 = I7
H6 = I6·I7'
H5 = I5·I6'·I7'
H4 = I4·I5'·I6'·I7'
... *(equações idênticas para H3 a H0)*

A0 = H1 + H3 + H5 + H7
A1 = H2 + H3 + H6 + H7
A2 = H4 + H5 + H6 + H7
IDLE = I0'·I1'·I2'·I3'·I4'·I5'·I6'·I7'

5.5. *Buffers Tri-state*

Algumas saídas de dispositivos podem assumir um 3º estado eléctrico, que não é 0 nem 1, mas sim alta impedância (high-Z) ou estado flutuante. Este estado equivale a ter a saída desligada do resto do circuito. Os dispositivos que têm uma saída passível de estar em alta impedância possuem uma entrada de controlo *(output enable*) que define se a saída está em High-Z ou não.

Um ***buffer tri-state*** é um circuito que coloca na saída a entrada ou então coloca a saída em alta impedância.



Os *buffers tri-state* permitem ligar várias saídas a um mesmo ponto dum circuito, desde que apenas um OE esteja activo em cada instante. Esta situação é necessária quando vários dispositivos ligam a um barramento comum.
A descrição VHDL que segue, para um *buffer tri-state*, usa o nível '**Z**' do tipo STD_LOGIC para representar o estado High-Z.

```
architecture funcional1 of bufTriState is
begin
   O <= I when OE='1' else 'Z';
end funcional1;
```

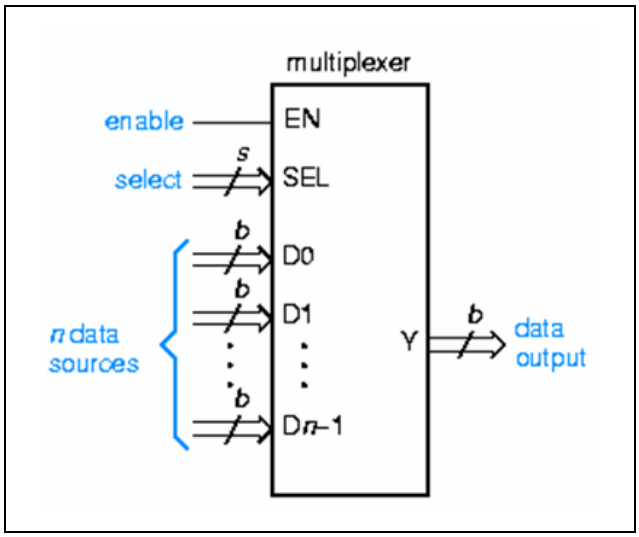
5.6. *Multiplexadores*

Um **multiplexador** (ou apenas **mux**) é um comutador digital que encaminha a informação de uma das **n** entradas para a única saída (ver figura abaixo). A entrada SEL, com **s** bits, permite seleccionar uma de entre **n** entradas, logo:

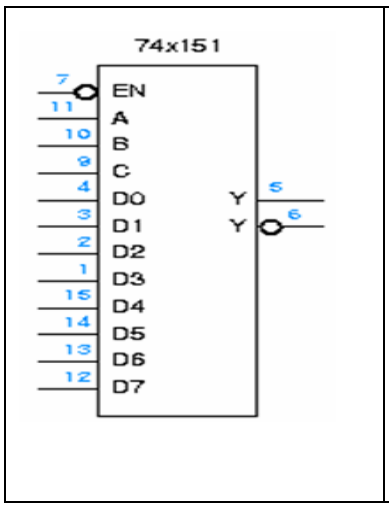
$s = \lceil \log_2 n \rceil$.

Quando o *enable* está inactivo (EN=0), Y=0. Quando está activo (EN=1), o mux funciona.

Os multiplexadores são usados nos computadores, entre os registos do processador e a ALU, para seleccionar os registos que ligam às entradas da ALU.

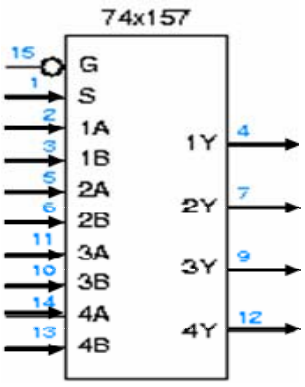


O CI 74x151 implementa um multiplexador 8:1 (8 entradas de 1-bit: D0..D7). As entradas selectoras são A,B e C, em que C é o bit mais significativo (msb). A entrada de *enable* EN_L é activa a zero. O CI disponibiliza duas versões da saída: uma activa a zero (Y) e outra activa a 1 (Y_L).



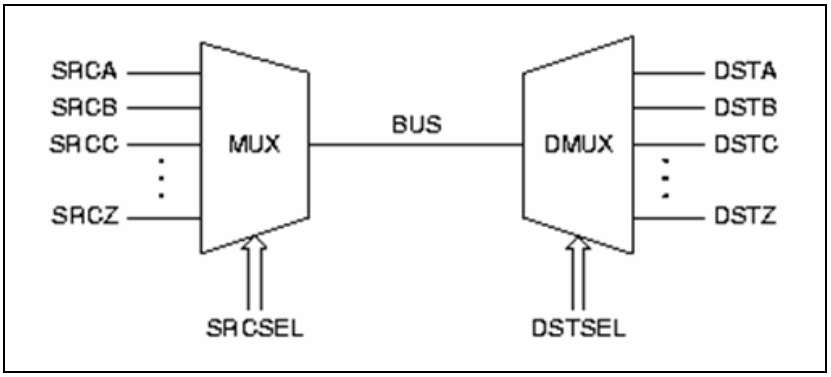
Inputs				Outputs	
EN_L	C	B	A	Y	Y_L
1	×	×	×	0	1
0	0	0	0	D0	D0′
0	0	0	1	D1	D1′
0	0	1	0	D2	D2′
0	0	1	1	D3	D3′
0	1	0	0	D4	D4′
0	1	0	1	D5	D5′
0	1	1	0	D6	D6′
0	1	1	1	D7	D7′

A figura que se segue apresenta o **multiplexador 2:1 de 4-bits** 74x157. A entrada selectora é S e a entrada de *enable* G_L é activa a zero. A notação da tabela de verdade foi estendida para que as entradas apareçam nas colunas das saídas, tornando a tabela mais clara e mais compacta.

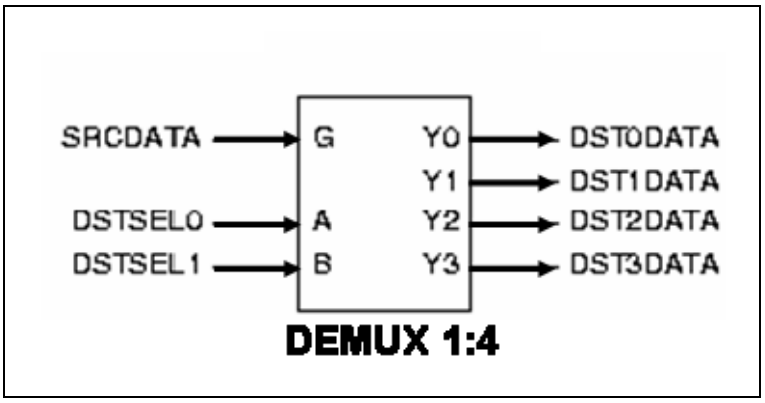


Inputs		Outputs			
G_L	S	1Y	2Y	3Y	4Y
1	×	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

Pode usar-se um multiplexador para seleccionar uma de **n** origens dos dados a enviar para um barramento. Por outro lado, pode usar-se um **desmultiplexador** para encaminhar os dados que flúem num barramento para um de **m** destinos (ver figura).



A funcionalidade dum desmultiplexador é inversa daquela que apresenta um multiplexador. Um desmultiplexador de 1-bit e **n**-saídas possui uma entrada de dados e **s** entradas selectoras para escolher para qual das $n=2^s$ saídas encaminha a única entrada de dados.



O exemplo da próxima figura descreve um MUX 4:1, em VHDL estilo fluxo de dados com uma atribuição selectiva. O MUX possui entradas de 8 bits (A, B, C e D).

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

entity mux4in8b is
    port (
        S: in STD_LOGIC_VECTOR (1 downto 0);      -- Select inputs, 0-3 ==> A-D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Data bus input
        Y: out STD_LOGIC_VECTOR (1 to 8)          -- Data bus output
    );
end mux4in8b;

architecture mux4in8b of mux4in8b is
begin
    with S select Y <=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
      (others => 'U') when others; -- this creates an 8-bit vector of 'U'
end mux4in8b;
```

O exemplo que se segue descreve um MUX 4:1, em VHDL estilo comportamental com uma instrução CASE. O MUX possui entradas de 8 bits (A, B, C e D).

```
architecture mux4in8p of mux4in8b is
begin
process(S, A, B, C, D)
  begin
    case S is
      when "00" => Y <= A;
      when "01" => Y <= B;
      when "10" => Y <= C;
      when "11" => Y <= D;
      when others => Y <= (others => 'U');  -- 8-bit vector of 'U'
    end case;
  end process;
end mux4in8p;
```

Em VHDL é fácil descrever um multiplexador que apresente uma estratégia de selecção da entrada a colocar na saída bem particular e rebuscada, usando as potencialidades do CASE / SELECT e da condição por defeito. Por exemplo recorrendo a: *(others => 'U')*.

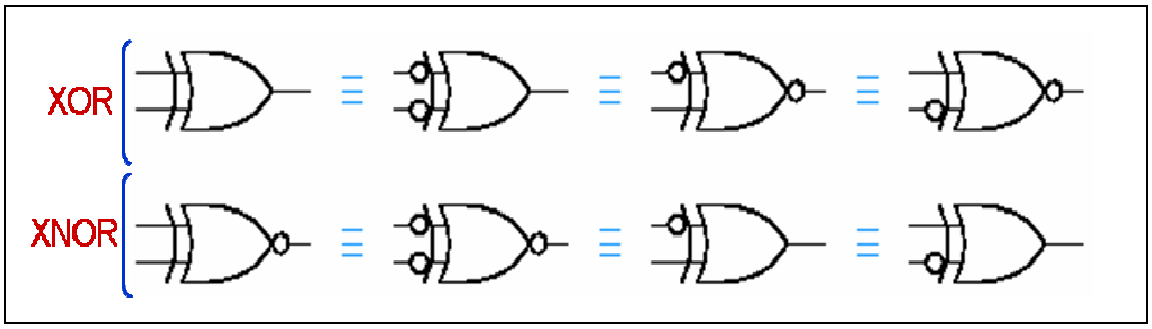
5.7. Portas XOR e circuitos detectores de paridade

Uma porta **OR-exclusivo (XOR)** é uma porta de 2-entradas cuja saída é 1, se exactamente uma das entradas for 1. Uma porta XOR gera um 1 na saída se as entradas forem diferentes.
Uma porta **NOR-exclusivo (XNOR)** é precisamente o oposto: gera um 1 na saída se as entradas forem iguais. A operação XOR é por vezes representada pelo símbolo ⊕.

$X \oplus Y \quad = X' \cdot Y + X \cdot Y'$
$(X \oplus Y)' = X' \cdot Y' + X \cdot Y$

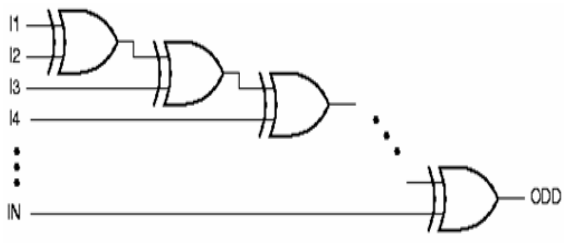
X	Y	X⊕Y (XOR)	(X⊕Y) (XNOR)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Pode usar-se um de 4 símbolos para representar uma porta XOR e XNOR:

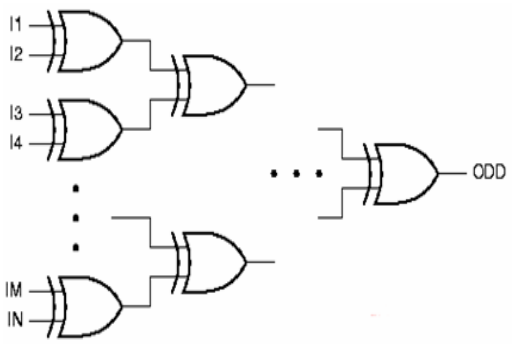


Estas alternativas resultam da aplicação da seguinte regra: quaisquer 2 sinais (entradas ou saída) duma porta XOR ou XNOR podem ser complementados sem que a função lógica implementada seja alterada. Escolhe-se o símbolo que é mais representativo da função lógica implementada.

Ao colocar **N-1** portas XOR em cascata constrói-se um circuito com **N** entradas e uma única saída que funciona como um **detector de paridade impar**, que gera um 1 na saída quando tiver um número impar de entradas a 1. Se negarmos a saída de qualquer dos 2 circuitos da figura seguinte, obtém-se um **detector de paridade par**, em que a saída é 1 quando o circuito tiver um número par de entradas a 1.



i) Detector de paridade impar com estrutura *daisy-chain*



ii) Detector de paridade impar com estrutura tipo árvore (mais rápida)

VHDL dispõe de operadores **xor** e **xnor** nativos. Uma porta XOR de 3-entradas pode ser especificada pelo seguinte código VHDL em estilo fluxo de dados.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity vxor3 is
  port (A, B, C: in STD_LOGIC;
        Y: out STD_LOGIC);
end vxor3;
architecture vxor3 of vxor3 is
begin
  Y <= A xor B xor C;
end vxor3;
```

O próximo exemplo descreve, em VHDL comportamental, um detector de paridade com uma entrada de 9-bits.

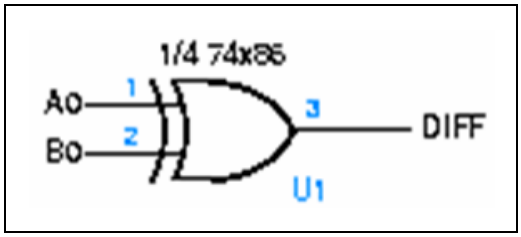
```
library IEEE;
use IEEE.std_logic_1164.all;
entity parity9 is
  port (I: in STD_LOGIC_VECTOR (1 to 9);
        EVEN, ODD: out STD_LOGIC);
end parity9;
architecture parity9p of parity9 is
begin
process (I)
  variable p : STD_LOGIC;
  begin
    p := I(1);
    for j in 2 to 9 loop
      if I(j) = '1' then p := not p; end if;
    end loop;
    ODD <= p;
    EVEN <= not p;
  end process;
end parity9p;
```

Pode acontecer que esta descrição comportamental não seja eficientemente sintetizada pela ferramenta de síntese, o que pode implicar ter que recorrer a uma descrição estrutural.

5.8. Comparadores

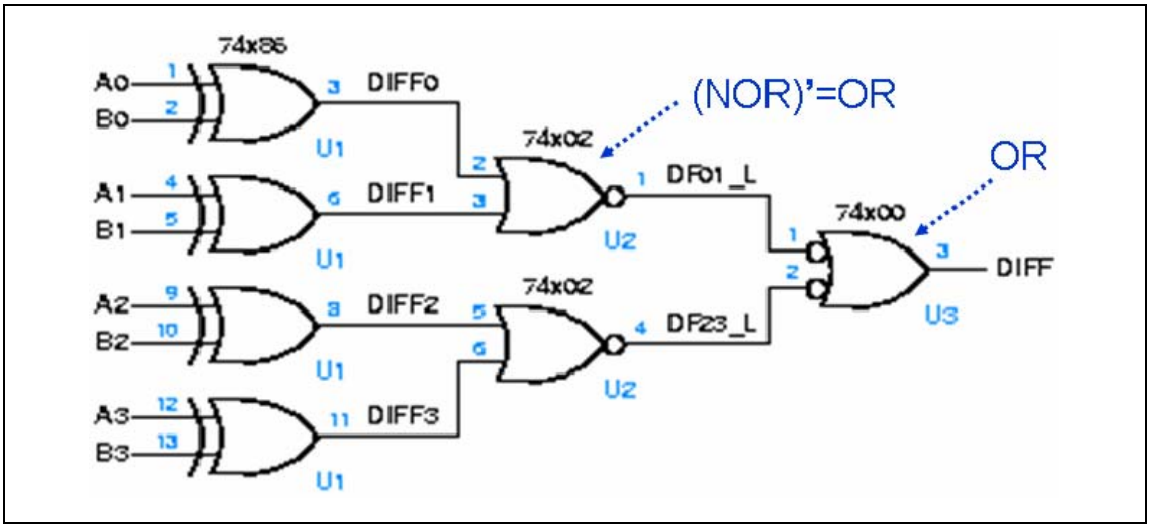
Comparar 2 palavras binárias é uma operação comum num computador. Um circuito que compara 2 palavras binárias e indica se elas são iguais ou diferentes é um **comparador**. Alguns comparadores (i) distinguem entre entradas que representam números com e sem sinal e (ii) indicam uma relação aritmética entre as entradas (maior que ou menor que). Estes circuitos são geralmente denominados por **comparadores de amplitude**.

Uma porta XOR e XNOR pode ser vista como sendo 1 comparador de 1-bit.



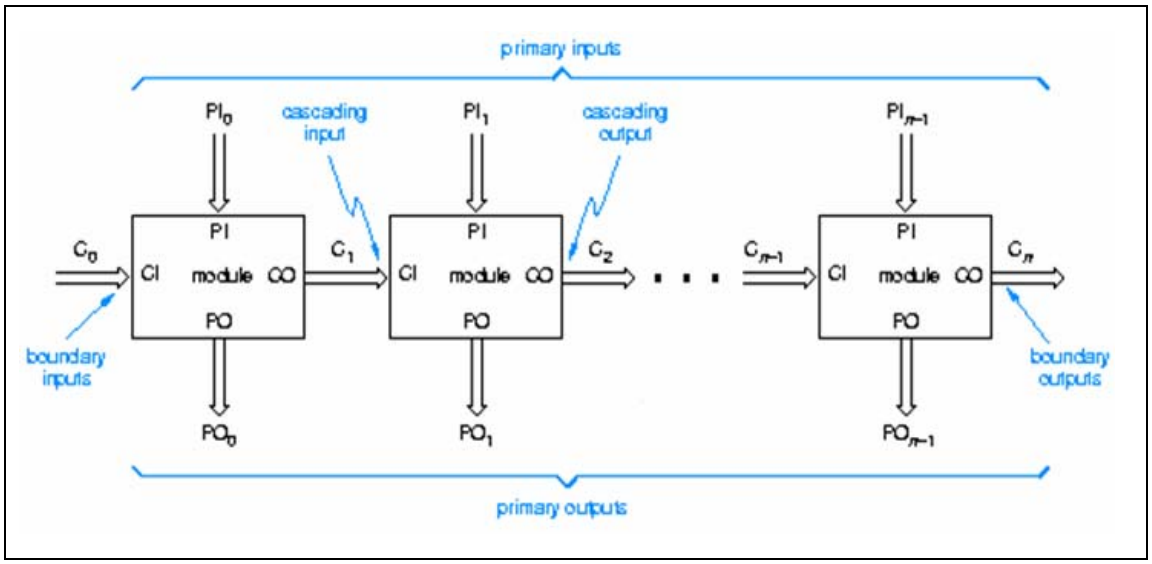
A saída DIFF é activada quando as entradas forem diferentes.

O próximo exemplo é um comparador de 4-bits que efectua o OR entre as saídas de 4 portas XOR.



A saída DIFF é activada quando algum par de bits da entrada (Ai , Bi) for diferente. O circuito pode ser facilmente adaptado para funcionar com palavras de qualquer número de bits.

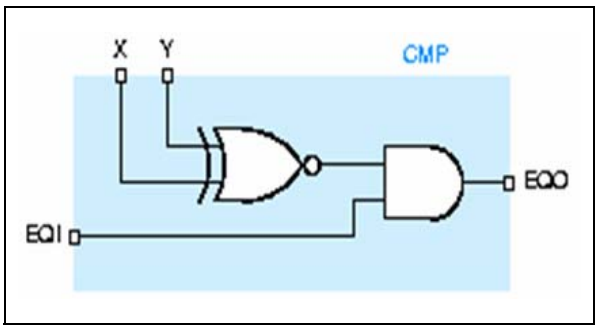
Um **circuito iterativo** genérico é um circuito combinacional com a seguinte estrutura:



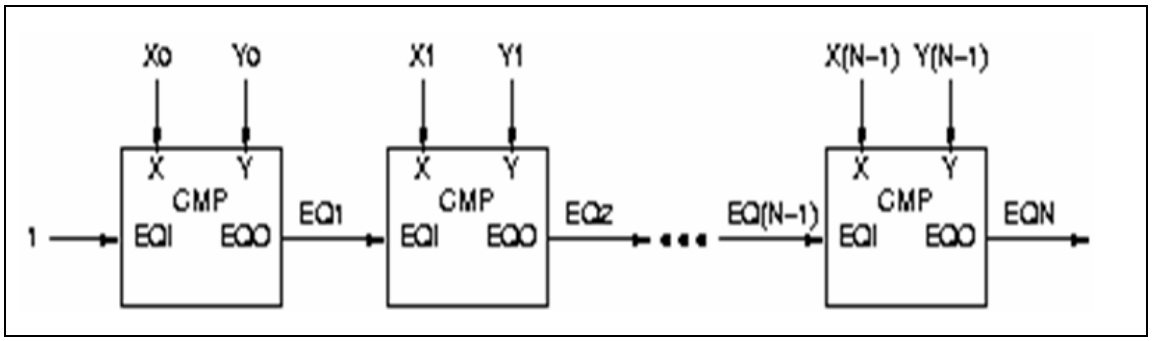
O circuito é composto por **n** módulos idênticos, cada um com entradas e saídas primárias e ainda com entradas e saídas em cascata. As entradas em cascata mais à esquerda (*boundary inputs*) estão normalmente ligadas a valores fixos.

Pode comparar-se dois valores **X** e **Y** de n-bits efectuando a comparação bit-a-bit (como o componente CMP), usando em cada etapa apenas um bit $\mathbf{EQ_i}$ que indica se todos os pares de bits comparados até ai são iguais ou não:

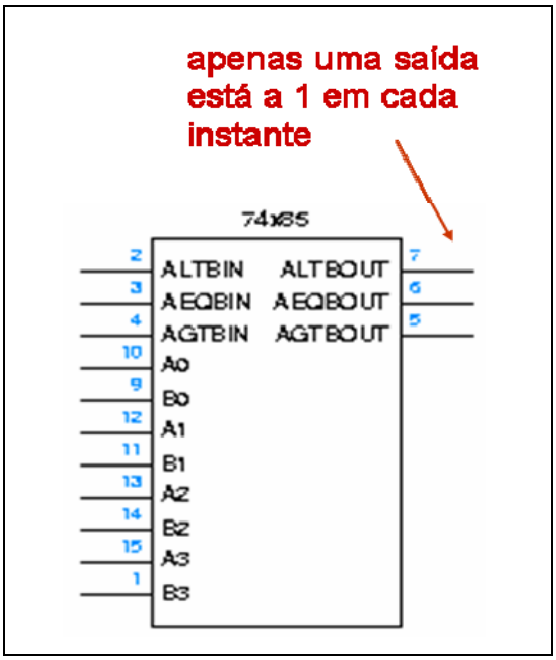
1. Colocar EQ_0 a 1 e *i* a 0.
2. Se EQ_i é 1 e X_i=Y_i, colocar EQ_{i+1} a 1.
   Senão colocar EQ_{i+1} a 0.
3. Incrementar *i*.
4. Se $i < n$ regressar à etapa 2.



O algoritmo anterior é implementado pelo seguinte circuito:



Estão disponíveis no mercado vários comparadores MSI. O CI 74x85 implementa um comparador de 4-bits.



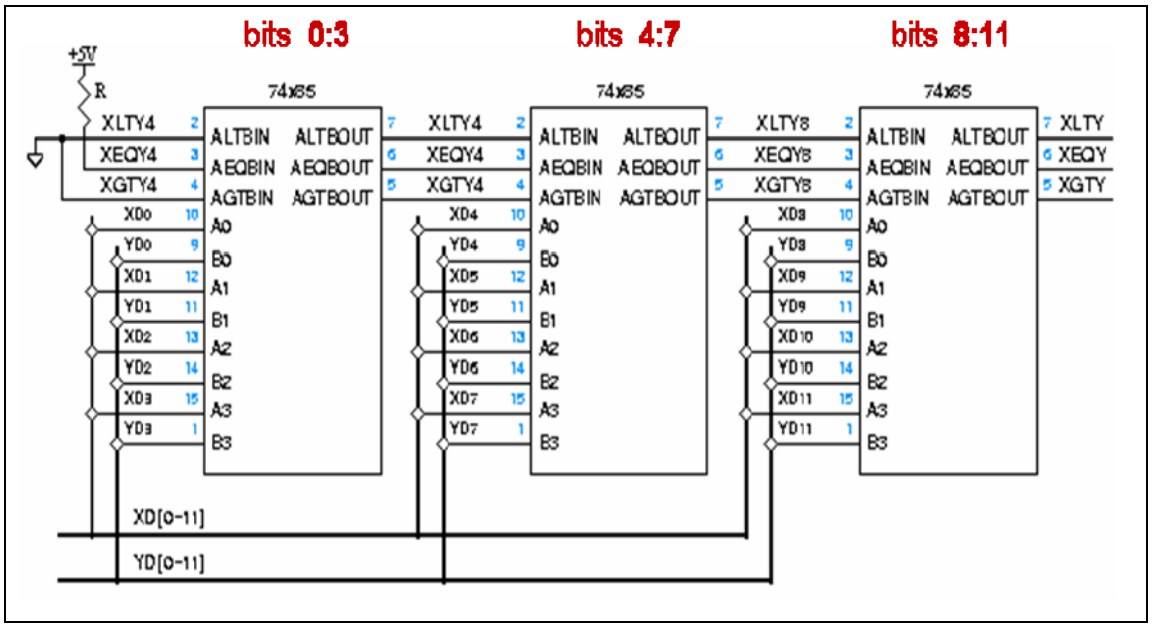
Este CI dispõe de saídas com indicação de "***maior que***", "***menor que***" e "***igual a***". O CI 74x85 possui ainda entradas em cascata para combinar múltiplos circuitos de modo a construir comparadores com mais do que 4 bits.

$$AGTBOUT = (A>B) + (A=B) \cdot AGTBIN$$
$$AEQBOUT = (A=B) \cdot AEQBIN$$
$$ALTBOUT = (A<B) + (A=B) \cdot ALTBIN$$

Por exemplo, (A>B) é dado por:

$$A3 \cdot B3' + (A3 \oplus B3)' \cdot A2 \cdot B2' + (A3 \oplus B3)' \cdot (A2 \oplus B2)' \cdot A1 \cdot B1' + (A3 \oplus B3)' \cdot (A2 \oplus B2)' \cdot (A1 \oplus B1)' \cdot A0 \cdot B0'$$

Com 3 circuitos 74x85, pode construir-se um comparador de 12-bits.



O VHDL dispõe de operadores para comparar qualquer dos tipos de dados nativos. Os operadores igualdade (**=**) e desigualdade (**/=**) aplicam-se a todos os tipos. Para comparar *arrays* e *records*, os operandos devem ter a mesma dimensão e estrutura, e os operandos são comparados elemento-a-elemento.
Os outros operadores do VHDL (**>**, **<**, **>=**, **<=**) aplicam-se apenas a inteiros, tipos enumerados e *arrays* unidimensionais de inteiros ou de tipos enumerados.

5.9. Somadores, subtractores e ALUs

A adição (ou soma) é a operação aritmética mais frequente nos sistemas digitais. Um **somador** combina dois operandos aritméticos aplicando as regras da adição. Nos números sem sinal e nos números com sinal em complemento para 2 usam-se as mesmas regras da adição, logo os mesmos somadores.
Um somador pode efectuar a subtracção através da soma do minuendo com o subtraendo complementado. Também se pode construir um circuito **subtractor** que efectua de forma directa a subtracção. Uma **ALU** (**U**nidade **A**ritmética e **L**ógica) efectua a soma, a subtracção e outras operações lógicas.

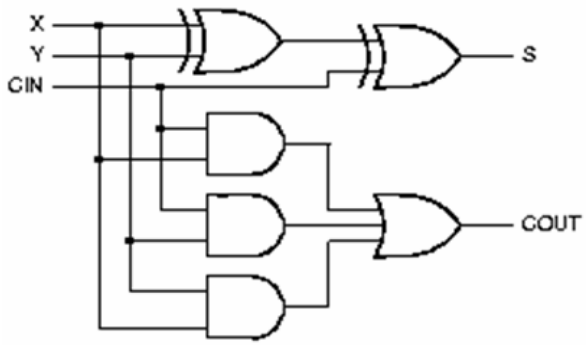
O somador mais simples, designado por **semi-somador**, soma dois operandos **X** e **Y** de 1-bit, produzindo um resultado com 2-bits. O resultado da soma varia de 0 a 2, logo é necessário 2 bits para o expressar. O bit menos significativo do resultado é designado por **HS** (*half sum*) e o bit mais significativo do resultado é designado por **CO** (*carry out*). As equações que definem o resultado da soma são:

$$HS = X \oplus Y = X \cdot Y' + X' \cdot Y$$
$$CO = X \cdot Y$$

Para somar operandos com mais do que um bit, é preciso considerar o transporte de uma soma (ao nível do bit) para a seguinte.

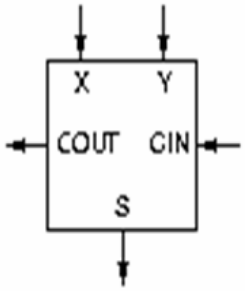
O bloco elementar com que se constrói um circuito que executa esta operação é um **somador completo**. Além das entradas **X** e **Y** a adicionar, um somador completo possui um bit de transporte como entrada: **CIN**. O resultado da soma dos 3 bits varia de 0 a 3 e pode ser expresso por duas saídas de 1 bit: **S** e **COUT**. As equações que definem o resultado da soma são:

$$S = X \oplus Y \oplus CIN$$
$$COUT = X \cdot Y + X \cdot CIN + Y \cdot CIN$$



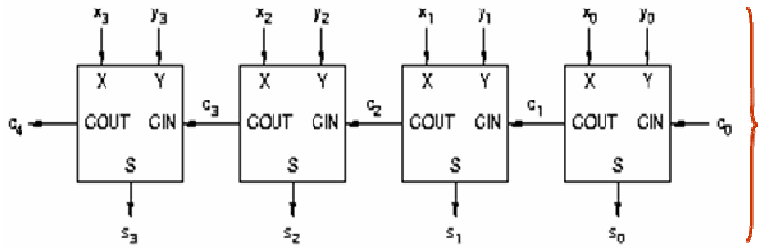
i) Esquemático do somador completo

ii) Símbolo do somador completo



iii) Símbolo alternativo do somador completo

Duas palavras em binário, cada uma com **n** bits, podem ser somadas usando um somador de *ripple*. Um **somador de *ripple*** é composto por uma cadeia de **n** somadores completos, onde cada somador processa um bit:



É lento porque o transporte é propagado do somador 0 (LS) até ao 3 (MS)

A entrada de transporte do somador do bit menos significativo (c_0) é colocada normalmente a 0. A saída de transporte de cada somador completo é ligada à entrada de transporte do somador completo seguinte e que é mais significativo do que ele.

A operação de subtracção binária é análoga à adição binária. Um **subtractor completo** possui como entradas **X** (minuendo), **Y** (subtraendo) e **BIN** (*borrow in*) e como saídas **D** (diferença) e **BOUT** (*borrow out*). As equações que definem o resultado da subtracção são:

D = X ⊕ Y ⊕ BIN
BOUT = X'·Y + X'·BIN + Y·BIN

manipulando estas equações obtém-se:

D = X ⊕ Y' ⊕ BIN' ← usando Y⊕BIN=Y'⊕BIN'
BOUT' = X·Y' + X·BIN' + Y'·BIN' ← T. DeMorgan e distributividade da '+'

Um subtractor completo pode ser construído com um somador completo:
X-Y = X+ (-Y) = X+Y'+1

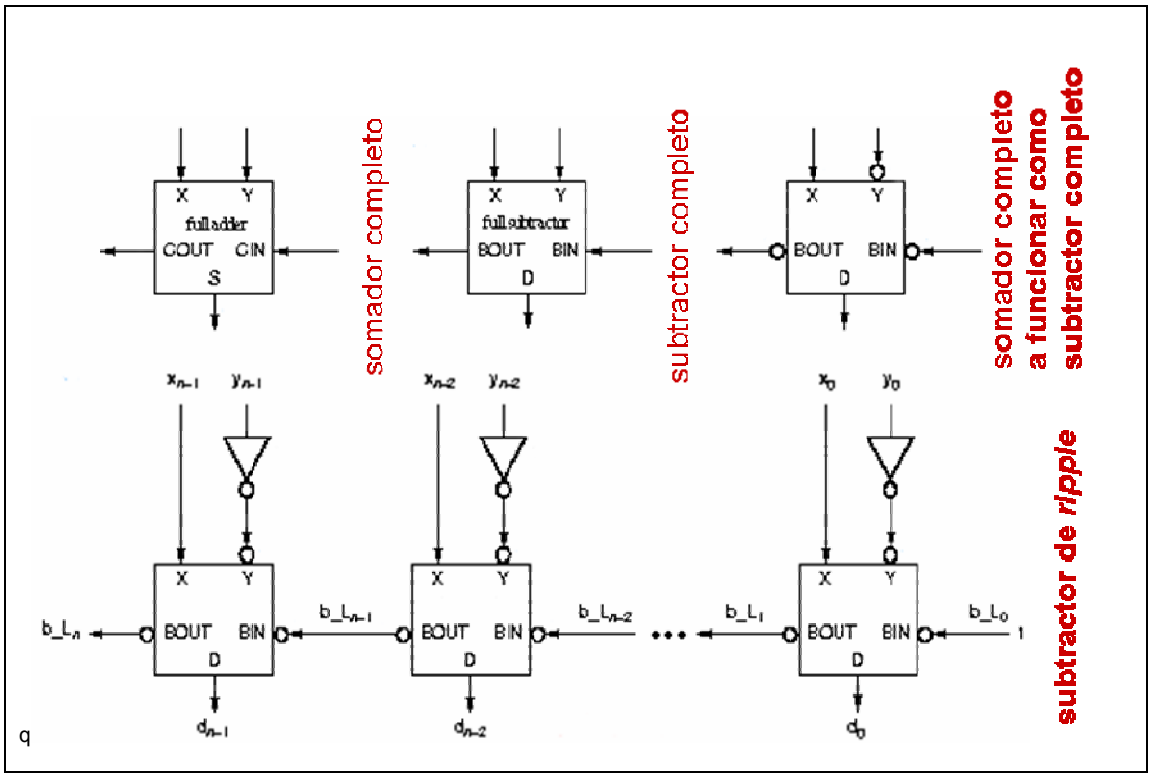
Nesta expressão o "1" resulta de BIN' ter substituído CIN, logo CIN[0]=0 passou a BIN[0]=1. Ainda nesta expressão, Y' é o complemento para 1 de Y.

Comparando as equações da <u>soma</u>
S = X ⊕ Y ⊕ CIN
COUT = X· Y + X· CIN + Y· CIN

com as equações da <u>subtracção</u>
D = X ⊕ Y' ⊕ BIN'
BOUT' = X·Y' + X·BIN' + Y'·BIN'

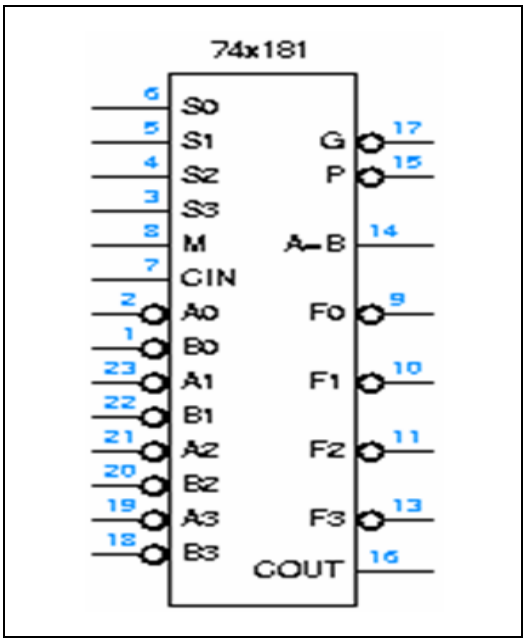
Constata-se que um <u>somador pode funcionar de subtractor</u> se:

S	for substituído por	D
X	for substituído por	X
Y	for substituído por	Y'
CIN	for substituído por	BIN'
COUT	for substituído por	BOUT'



Uma **ALU** é um circuito combinacional que pode efectuar diversas operações aritméticas e lógicas com um par de operandos de **b**-bits. A operação a efectuar é especificada por um conjunto de entradas selectoras. Tipicamente, uma ALU implementada num CI do tipo MSI possui 2 operandos de 4-bits e 3 a 5 entradas selectoras da operação a efectuar, permitindo escolher uma operação de entre 32 possíveis.

Por exemplo, o CI 74x181 implementa uma ALU de 4-bits. A operação a efectuar pela ALU do 74x181 é seleccionada pelas entradas **M** e **S3-S0**.



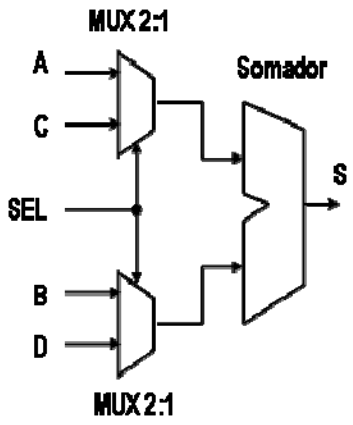
A tabela de verdade da ALU do 74x181 é:

Inputs				Function	
S3	S2	S1	S0	M = 0 (arithmetic)	M = 1 (logic)
0	0	0	0	F = A minus 1 plus CIN	F = A′
0	0	0	1	F = A · B minus 1 plus CIN	F = A′ + B′
0	0	1	0	F = A · B′ minus 1 plus CIN	F = A′ + B
0	0	1	1	F = 1111 plus CIN	F = 1111
0	1	0	0	F = A plus (A + B′) plus CIN	F = A′ · B′
0	1	0	1	F = A · B plus (A + B′) plus CIN	F = B′
0	1	1	0	F = A minus B minus 1 plus CIN	F = A ⊕ B′
0	1	1	1	F = A + B′ plus CIN	F = A + B′
1	0	0	0	F = A plus (A + B) plus CIN	F = A′ · B
1	0	0	1	F = A plus B plus CIN	F = A ⊕ B
1	0	1	0	F = A · B′ plus (A + B) plus CIN	F = B
1	0	1	1	F = A + B plus CIN	F = A + B
1	1	0	0	F = A plus A plus CIN	F = 0000
1	1	0	1	F = A · B plus A plus CIN	F = A · B′
1	1	1	0	F = A · B′ plus A plus CIN	F = A · B
1	1	1	1	F = A plus CIN	F = A

M=1 é utilizado para efectuar operações lógicas que não usam transporte.

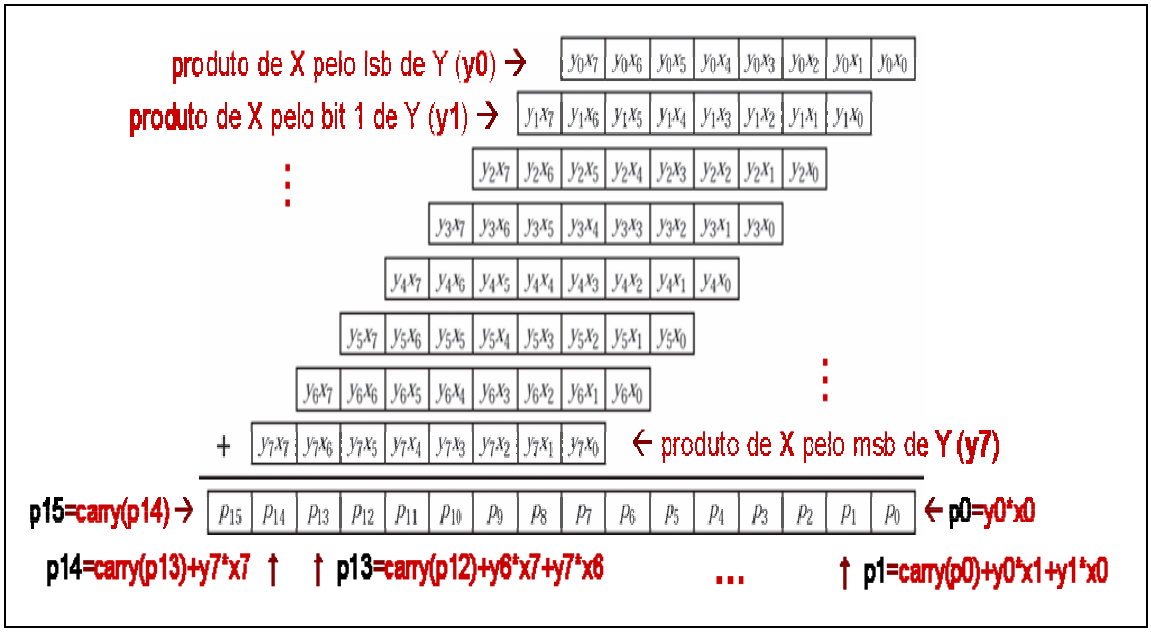
Na figura a baixo, apresenta-se um circuito descrito em VHDL de modo a que ao sintetizar se partilhe um somador entre duas somas: A+B e C+D. Deste modo, poupa-se lógica porque em vez de dois somadores usa-se apenas um somador mais dois MUX2:1 para seleccionar as entradas (A ou C; B ou D).

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

entity vaddshr is
    port (
        A, B, C, D: in SIGNED (7 downto 0);
        SEL: in STD_LOGIC;
        S: out SIGNED (7 downto 0)
    );
end vaddshr;

architecture vaddshr_arch of vaddshr is
begin
    S <= A + B when SEL = '1' else C + D;
end vaddshr_arch;
```



5.10. Multiplicadores

O algoritmo tradicionalmente usado para multiplicar números binários emprega deslocamentos e somas na obtenção do resultado. Contudo, não é essa a única forma possível de implementar a multiplicação.
Dadas duas entradas de **n**-bits (**X**, **Y**), pode escrever-se a tabela de verdade que expressa o produto **P=X*Y** de **2n**-bits através duma função combinacional de **X** e **Y**. A maior parte das estratégias de implementação de multiplicadores combinacionais baseiam-se no algoritmo tradicional com deslocamentos e somas.

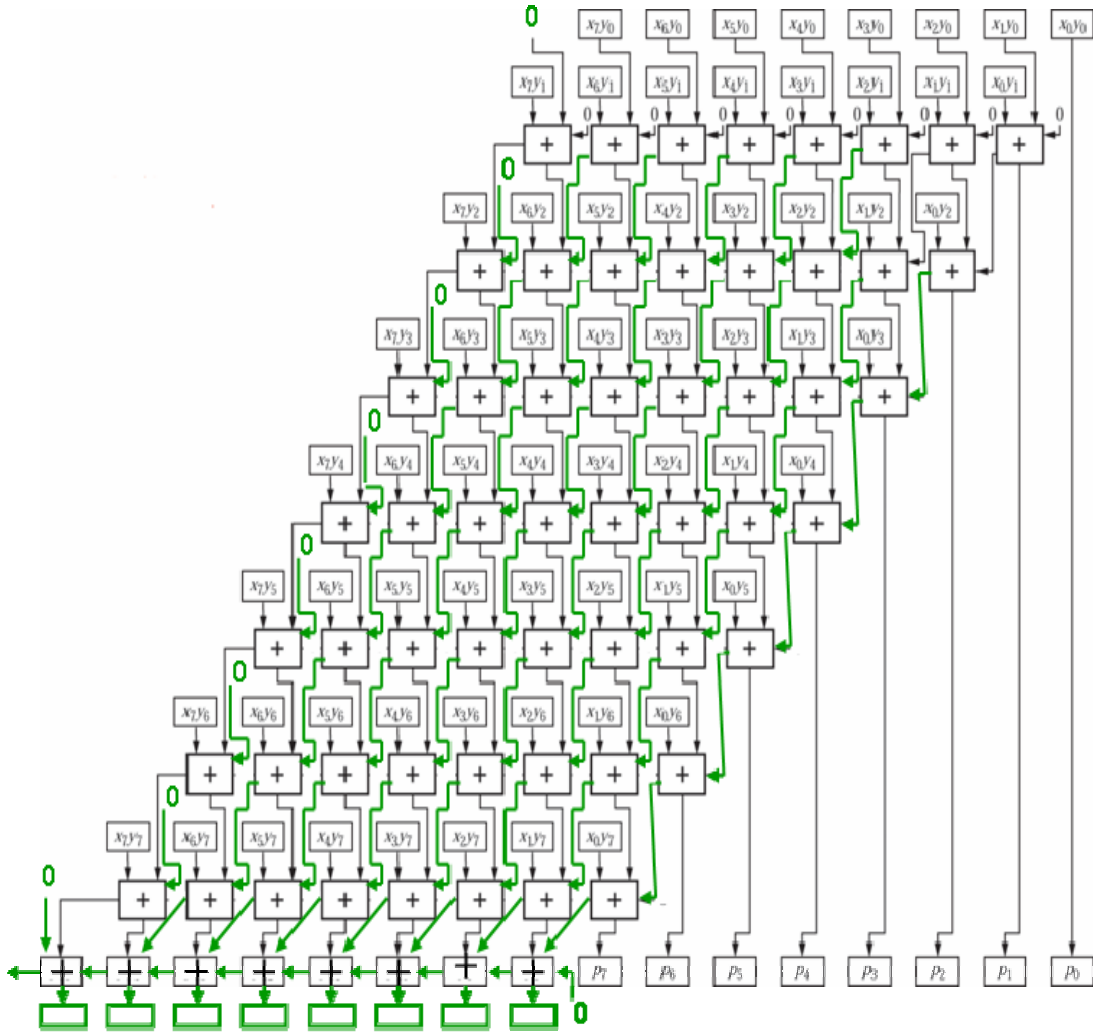


O esquemático seguinte implementa a estratégia de multiplicação anterior.

Pode descrever-se em VHDL comportamental a estratégia de multiplicação anterior, usando uma descrição que imita essa estrutura. Para representar as ligações entre somadores de 1 bit usam-se arrays bidimensionais, como se ilustra a seguir:

```
type ARRAY8x8 is array (7 downto 0) of std_logic_vector(7 downto 0);
variable CARRY : ARRAY8x8;
```

Contudo a biblioteca IEEE **std_logic_arith** fornece o operador "*" para operandos com sinal (SIGNED) e sem sinal (UNSIGNED). Este operador é descrito usando um algoritmo idêntico ao anterior e permite escrever um programa que multiplica 2 valores sem sinal com uma única linha de código:



Os blocos “+” são somadores completos de 1 bit.
O pior atraso do circuito é **15** * atraso dum somador completo.

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

entity vmul8x8i is
    port (
        X: in UNSIGNED (7 downto 0);
        Y: in UNSIGNED (7 downto 0);
        P: out UNSIGNED (15 downto 0)
    );
end vmul8x8i;

architecture vmul8x8i_arch of vmul8x8i is
begin
  P <= X * Y;
end vmul8x8i_arch;
```

6. Conceitos sobre Sistemas Sequenciais

6.1. Circuitos combinacionais vs sequenciais

Os circuitos lógicos podem ser classificados de combinacionais ou sequenciais. Um **circuito combinacional** é aquele em que as saídas só dependem das entradas actuais. Exemplo: comando com botão para escolher o canal da TV. Um **circuito sequencial** é aquele em que as saídas dependem das entradas actuais, mas também da sequência de valores por que passaram as entradas. Exemplo: comando para escolher o canal da TV com um botão para ir para o canal próximo/anterior (botão "+/-"). Não é possível descrever o comportamento dum circuito sequencial simplesmente com uma tabela que relacione as entradas com as saídas.

Para saber para onde vai evoluir um circuito sequencial, é preciso conhecer em que situação ele se encontra actualmente. Ou seja, o **estado** desse circuito deve ser memorizado.

6.2. Estado

O estado dum circuito sequencial é o conjunto de **variáveis de estado**, que guarda a informação relativa ao passado/presente desse circuito, necessária para determinar o seu comportamento futuro. No exemplo do comando para escolher o canal da TV, o número do canal actual é o estado actual. Conhecido o estado actual, pode sempre prever-se o próximo estado em função das entradas actuais. Num circuito digital, as variáveis de estado são valores binários e correspondem a sinais internos desse circuito. Um circuito com **n** variáveis de estado binárias pode ter até $\mathbf{2^n}$ estados.
Um circuito sequencial também pode ser designado de **máquina de estados finita**, ou seja, máquina com um número de estados finito.

As mudanças de estado acontecem em instantes impostos por um **sinal de relógio**. Um sinal de relógio é activo no nível alto se as mudanças de estado acontecerem no bordo ascendente do relógio ou quando o relógio está no nível ALTO. Caso contrário, é activo no nível baixo. O período do relógio (**T**) coincide com o intervalo entre transições sucessivas (do relógio) na mesma direcção. A frequência do relógio (**f**) é o inverso do período do relógio ($f = 1 / T$).

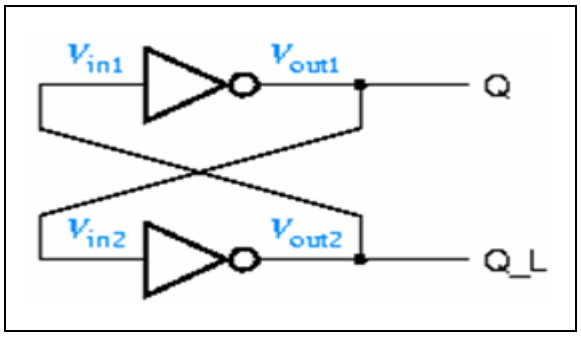
Neste módulo abordam-se dois tipos de circuitos sequenciais:

- Circuitos sequenciais simples*:* usam portas lógicas elementares (ANDs, ORs,…) e ciclos (ou caminhos) com *feedback* para criar elementos de memória (***latches*** e ***flip-flops***).
- Máquinas de estados síncronas com um sinal relógio: usam *latches* e *flip-flops* para criar circuitos que funcionam sob o controlo dum sinal de relógio.

6.3. Elementos bi-estáveis

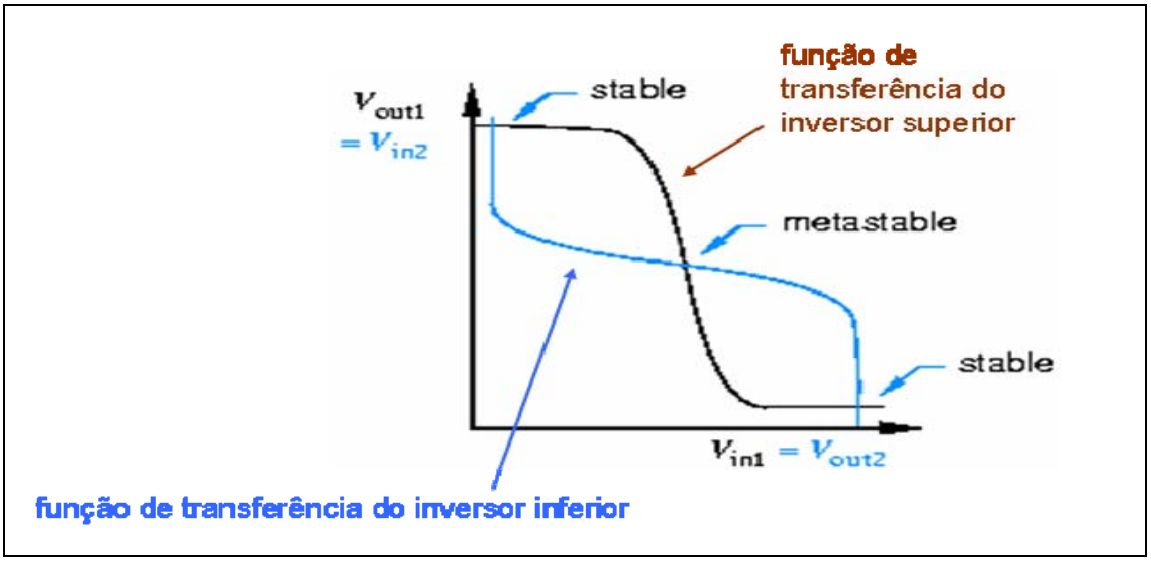
O circuito sequencial mais simples de todos é um circuito sem entradas e construído com um par de inversores interligados de modo a estabelecer um ciclo com *feedback*. A este circuito dá-se o nome de **bi-estável** porque possui dois estados (ou situações) estáveis:

- Quando **Q** está no nível ALTO, o inversor inferior tem a saída no nível BAIXO, forçando deste modo o inversor superior a colocar a sua saída no nível ALTO (como se assumiu inicialmente).
- Quando **Q** está no nível BAIXO, o inversor inferior tem a saída no nível ALTO, forçando deste modo o inversor superior a colocar a sua saída no nível BAIXO (como se assumiu inicialmente).



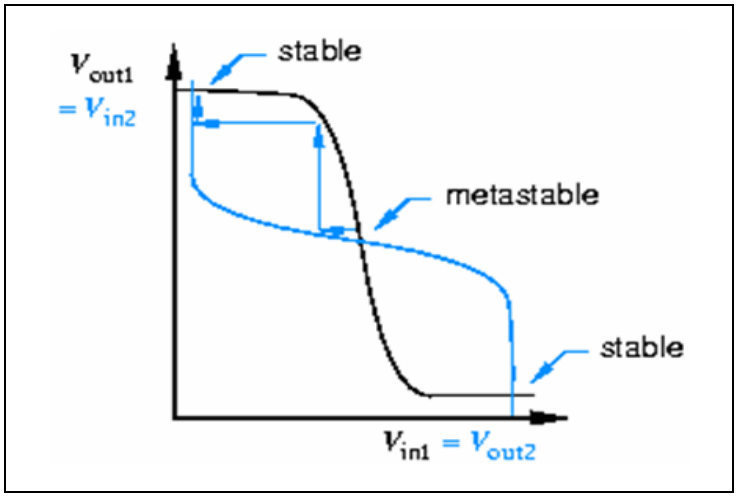
Pode usar-se uma única variável de estado (sinal **Q**) para definir o estado do circuito. Logo, há 2 estados possíveis: Q=0 e Q=1.

O elemento bi-estável é tão simples que não possui entradas, o que impede que o seu estado seja controlado. Quando o circuito é alimentado, ele assume um estado aleatório e permanece nele indefinidamente. Efectuando uma análise do bi-estável segundo uma perspectiva analógica percebe-se melhor o seu funcionamento.

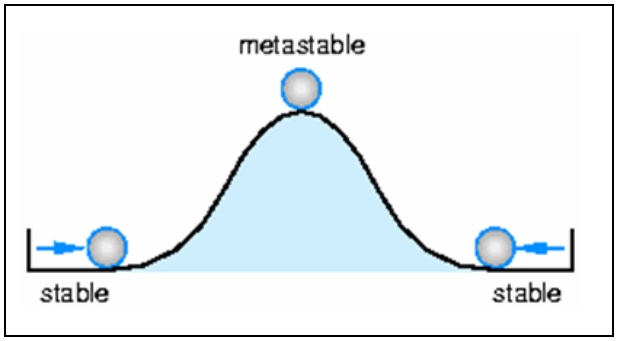


O bi-estável está em equilíbrio se as tensões de entrada e de saída em ambos os inversores assumirem um valor constante e consistente com (i) as ligações do ciclo de *feedback* e (ii) a função de transferência dos inversores.

O bi-estável está em equilíbrio nas posições assinaladas com "stable". Há um 3º ponto de equilíbrio, assinalado com "metastable", que ocorre quando $\mathbf{V_{out1}}$ e $\mathbf{V_{out2}}$ não são nem **0** nem **1** lógico. Se não houvesse ruído e o circuito atingisse o ponto meta-estável, poderia permanecer nele indefinidamente. O ponto é meta-estável porque o ruído tenderá a levar o circuito para uma das posições estáveis.



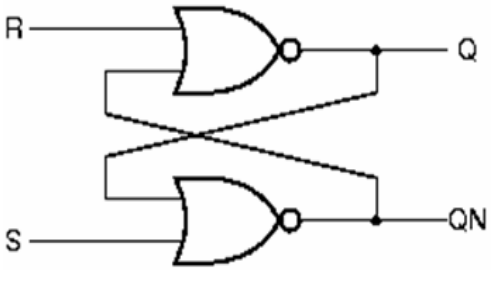
Na figura seguinte mostra-se uma analogia do ponto de meta-estabilidade com uma bola lançada sobre o pico duma montanha.



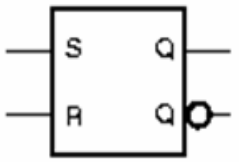
6.4. Latches e flip-flops

As *latches* e os *flips-flops* são os blocos elementares com os quais se constrói a maior parte dos circuitos sequenciais. Um ***flip-flop*** é um dispositivo sequencial que amostra as suas entradas e que altera as suas saídas apenas em instantes determinados por um sinal de relógio. Uma ***latch*** é um dispositivo sequencial que observa todas as suas entradas continuamente e altera as suas saídas em qualquer momento, independentemente de qualquer sinal de relógio.

Pode construir-se uma ***latch* S-R** com portas NOR (**S**=set, **R**=reset). Normalmente **QN** é o complemento de **Q**. Se S e R estiverem ambos a 0, o circuito comporta-se como o elemento bi-estável. Deve activar-se S ou R para forçar o ciclo de *feedback* a atingir o estado (estável) desejado. A entrada **S** define (***sets*** ou *presets*) a saída Q a 1 e a entrada **R** define (***resets*** ou *clears*) a saída Q a 0.

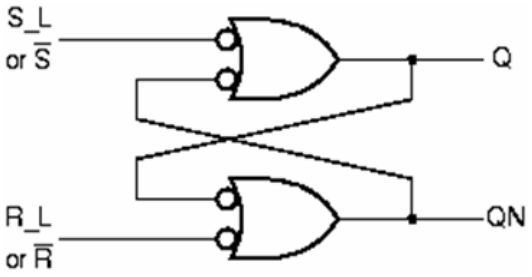


S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0



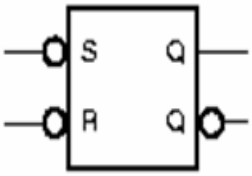
Pode construir-se uma *latch* S-R, com entradas de *set* e *reset* activas no nível baixo, com portas NAND. O funcionamento desta ***latch* S-R** é idêntico ao da anterior, com 2 diferenças:

- **S_L** e **R_L** são activas no nível baixo, logo a *latch* mantém o seu estado quando S_L=R_L=1.
- Quando S_L e R_L estiverem ambas activas, ambas as saídas ficam a 1 (e não a 0).

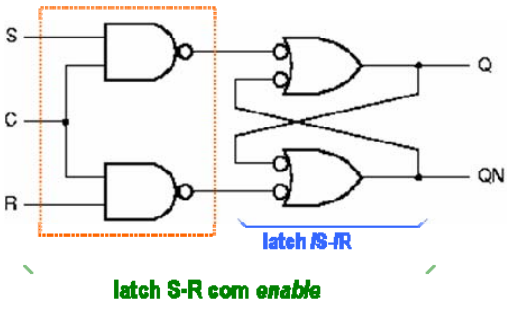


Um OR com as entradas negadas equivale a 1 NAND.
Em relação à latch S-R, /S e /R trocaram de posição.

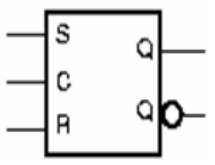
S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN



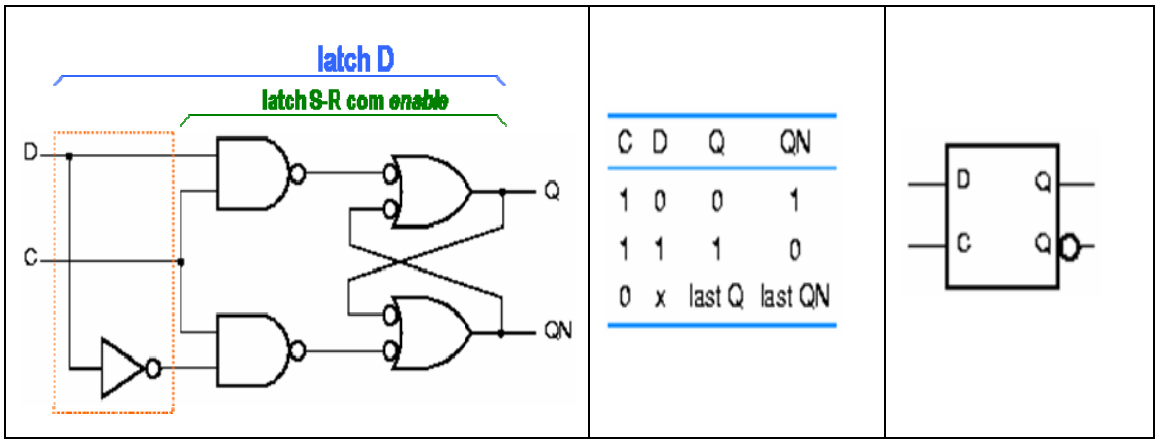
Uma *latch* S-R (ou /S-/R) reage ao valor das entradas em qualquer momento. Contudo, pode ser alterada para reagir ao valor das entradas apenas quando uma entrada de ***enable*** (**C**) estiver activa. O circuito alterado comporta-se tal como a *latch* S-R quando C=1 e mantém o estado quando C=0.



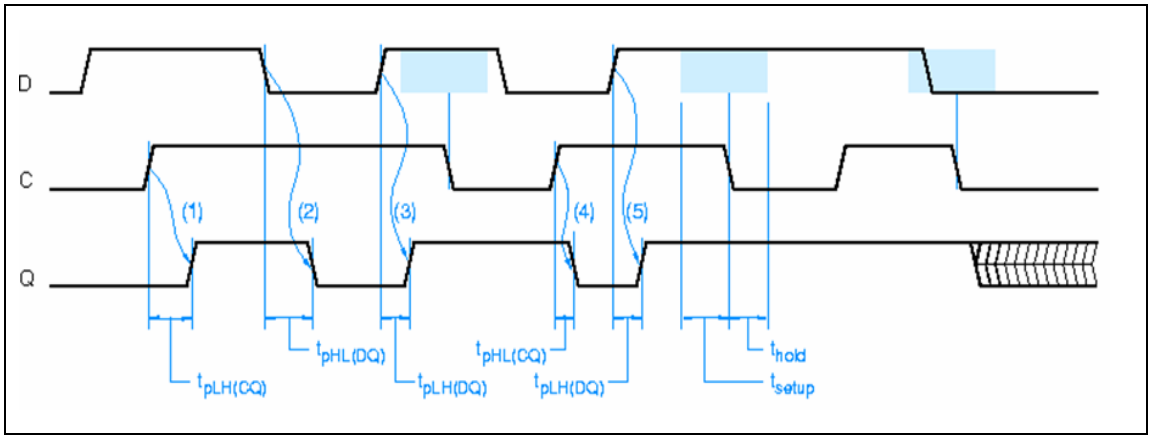
S	R	C	Q	QN
0	0	1	last Q	last QN
0	1	1	0	1
1	0	1	1	0
1	1	1	1	1
x	x	0	last Q	last QN



Quando a finalidade da utilização duma *latch* é guardar um bit de informação, a **latch D** é a mais recomendada. Pode construir-se uma *latch* D a partir duma *latch* S-R.

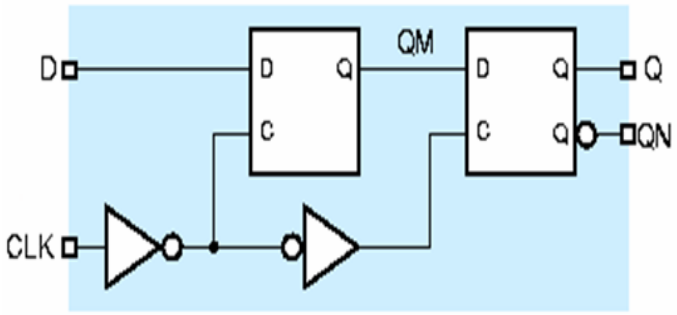


Esta *latch* elimina a situação problemática da *latch* S-R, que ocorre quando S e R são activadas (a 1) em simultâneo. Quando C=1, a *latch* está aberta / transparente e a saída Q acompanha a entrada D. Quando C=0, a *latch* está fechada e a saída Q mantém o último valor.



Há um atraso associado com a propagação dos sinais desde as entradas até à saída Q. No intervalo definido pelos ***setup time*** mais o ***hold time***, em torno do bordo descendente de **C**, a entrada D deve permanecer fixa. Se estes 2 tempos não forem respeitados, a saída da *latch* assumirá um valor imprevisível.

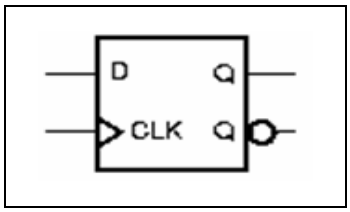
Um **flip-flop D positive-edge-triggered** *(FF D sensível à transição positiva)* é um circuito construído com um par de *latches* D que amostra a entrada D e altera as saídas Q e QN apenas no bordo ascendente do sinal CLK.



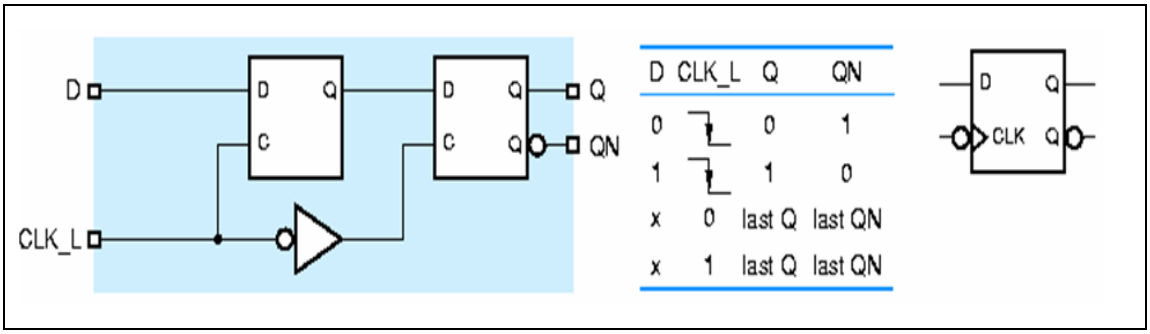
D	CLK	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

A primeira *latch* chama-se mestre e está aberta quando CLK=0. Quando CLK muda para 1, a *latch* mestre fecha. A segunda *latch*, o escravo, está aberta enquanto CLK=1, mas a saída muda de valor apenas no início desse intervalo, dado que o mestre está fechado nesse intervalo.

O triângulo na entrada CLK é um indicador de entrada dinâmica e assinala um comportamento sensível às transições (*edge-triggered*).

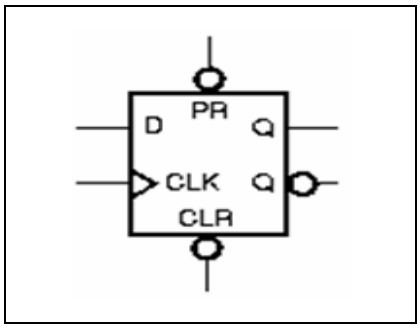


Num **flip-flop D negative-edge-triggered** *(FF D sensível à transição negativa)* inverte-se a entrada de relógio e a entrada D (as saídas Q e QN) passam a ser amostradas (alteradas) no bordo descendente do CLK.

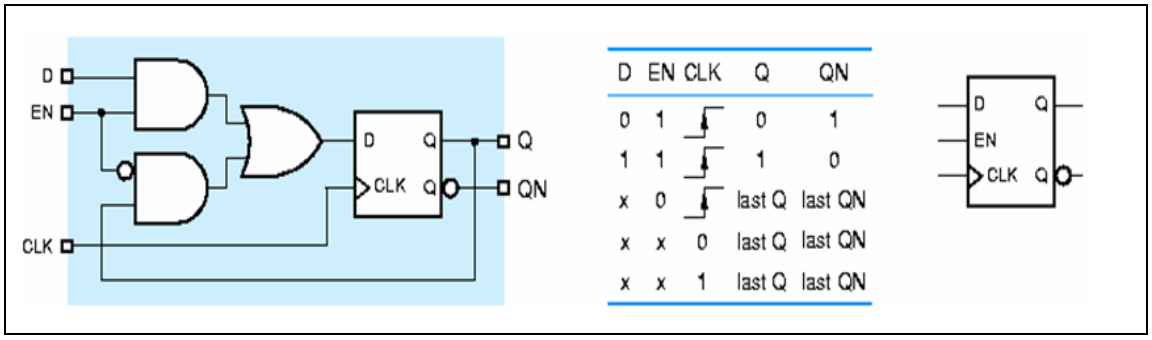


No intervalo definido pelos *setup time* mais o *hold time*, em torno dos bordos do CLK a que o FF é sensível, a entrada D deve permanecer fixa.

Alguns *flip-flops* D possuem 2 entradas assíncronas que servem para forçar o seu estado, independentemente das entradas CLK e D: PR e CLR.

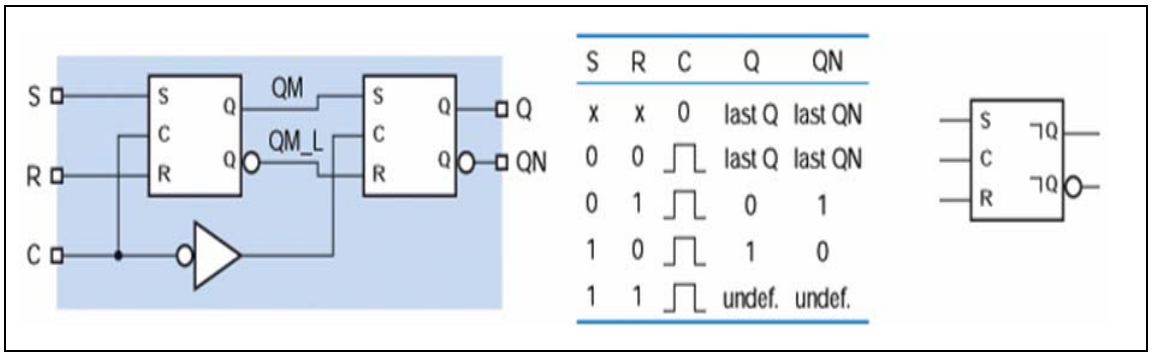


Estas entradas têm a mesma funcionalidade do set (S) e do reset (R) da *latch* S-R. As entradas assíncronas devem ser usadas nas fases de inicialização e teste dos circuitos. Alguns *flip-flops* D têm ainda a possibilidade de manter inalterado o último valor por que passou a saída. Para esse fim, adiciona-se ao FF D uma entrada de *enable*.



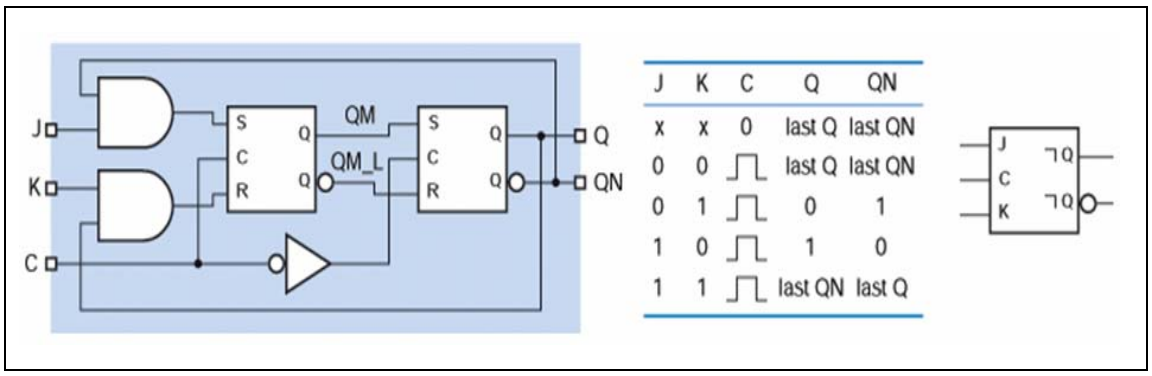
Q: Na figura anterior, porque não se aplica o *enable* no CLK, usando-se assim apenas um AND?

As *latches* S-R são úteis em sistemas de controlo, em que é comum ter condições independentes para colocar bits de controlo a 0/1. Quando se pretende que um dado bit de controlo seja apenas alterado em certos instantes determinados por um sinal de relógio, então exige-se um ***flip-flop* S-R**.



O comportamento deste flip-flop é o seguinte: possui uma estrutura mestre/escravo, não é verdadeiramente sensível à transição, o último valor guardado em QM (com C=1) só passa para a saída Q quando C mudar de 1 para 0 e se S=R=1, antes de C passar de 1 para 0, o valor da saída é imprevisível.

O problema que ocorre quando S e R estão ambos a 1 é resolvido no **flip-flop J-K** com estrutura mestre/escravo. As entradas J e K são análogas a S e R. No entanto, activar J só activa a entrada S da *latch* mestre se Q=0 (QN=1). Activar K só activa a entrada R da *latch* mestre se Q=1. Deste modo, se as entradas J e K forem activadas simultaneamente, as saídas do *flip-flop* mudam para o estado oposto do actual.

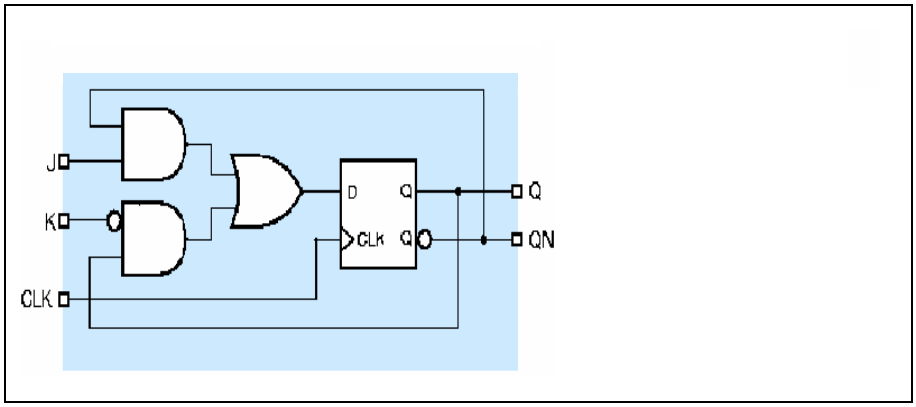


J	K	C	Q	QN
x	x	0	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	last QN	last Q

Com um *flip-flop* D *edge-triggered* pode construir-se um ***flip-flop J-K edge-triggered*** *(sensível às transições)*. O *flip-flop* JK amostra o valor das entradas (J e K) e altera as saídas (Q e QN) no bordo ascendente do CLK de acordo com a equação característica: **Q* = J . Q' + K' . Q**

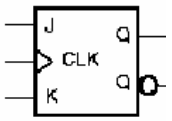
No intervalo definido pelos *setup time* mais o *hold time*, em torno do bordo ascendente de CLK, as entradas J e K devem permanecer fixas.

O *flip-flop* JK tem aplicação comum em máquinas de estado porque gera menos lógica combinacional.

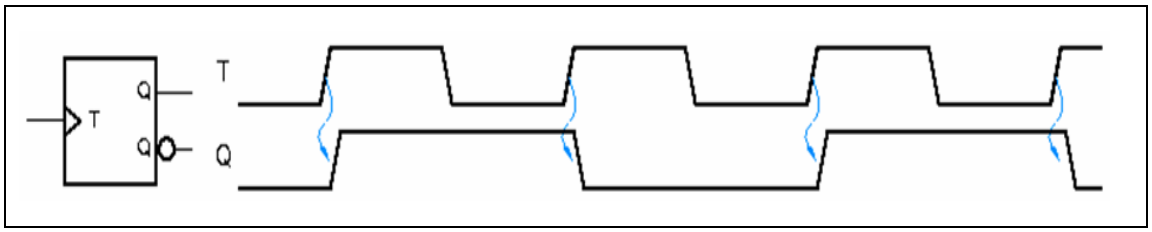


J	K	CLK	Q	QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	last QN	last Q

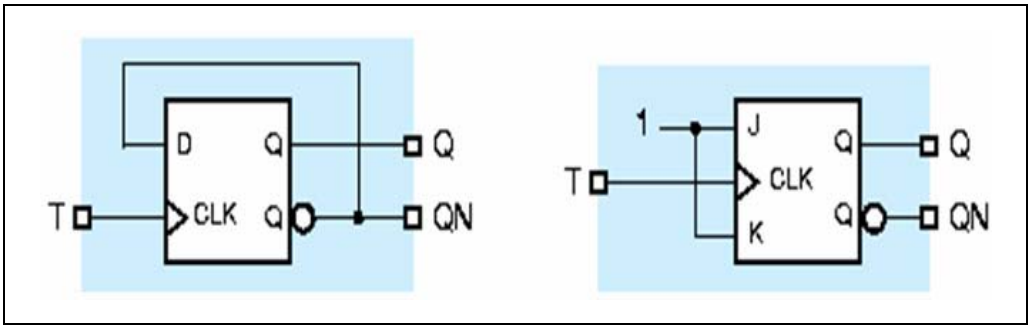
(c)



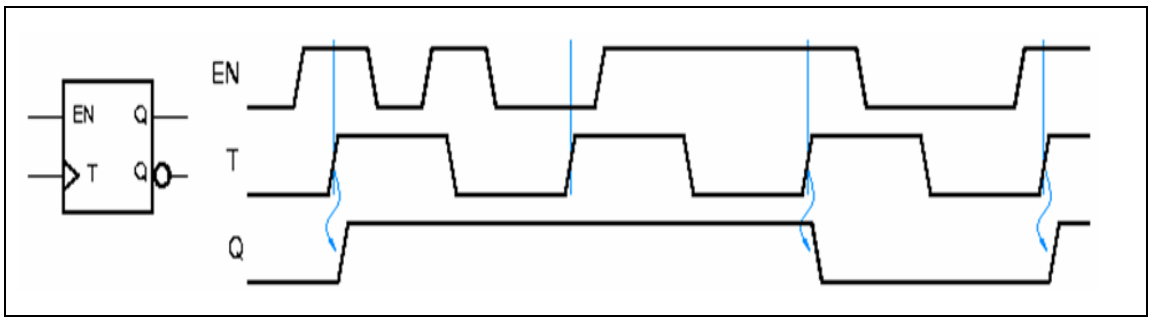
O ***flip-flop* T** (de *toggle*) muda de estado em cada transição 0 para 1 do sinal de relógio.



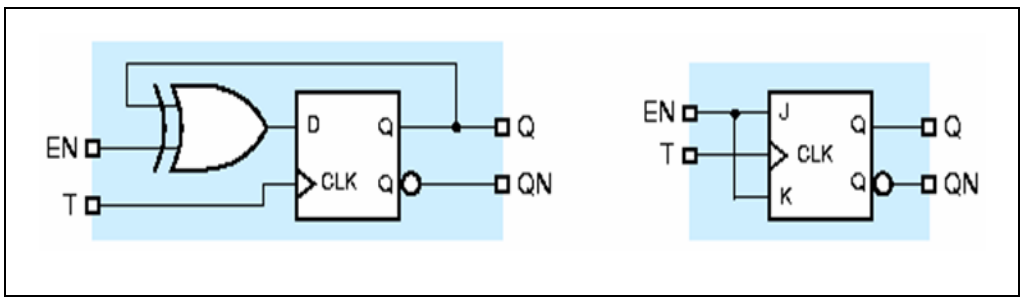
A saída Q do *flip-flop* possui uma frequência que é metade da frequência da entrada T. Pode usar-se um *flip-flop* D ou um J-K para construir o *flip-flop* T.



O *flip-flop* T também pode ter uma entrada de *enable*. Neste caso, o *flip-flop* só muda de estado no bordo ascendente do relógio (T) se a entrada de *enable* EN estiver activa.



Também se pode usar um *flip-flop* D e J-K para obter um *flip-flop* T com *enable*.



6.5. Projecto de máquinas de estados

Uma **máquina de estados finita** (FSM) pode ser definida formalmente por um quinteto **<S,I,O,F,G>**, em que:

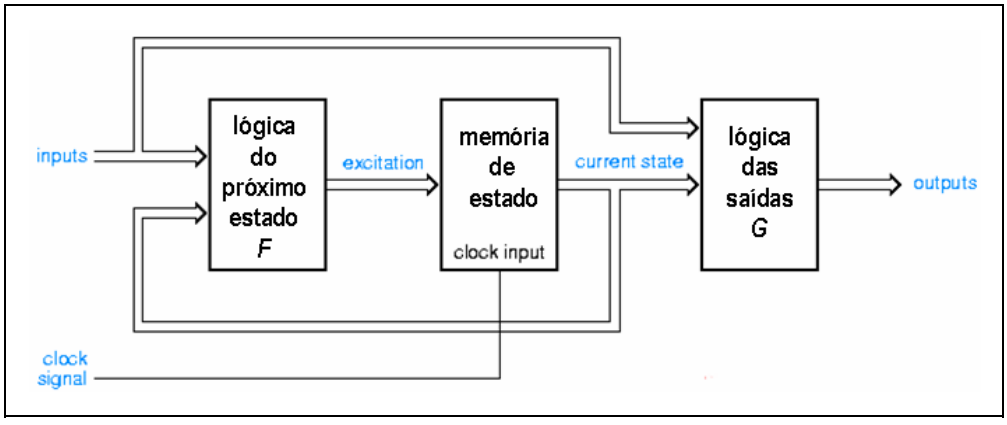
- **S** representa o conjunto de estados.
- **I** representa o conjunto de entradas.
- **O** representa o conjunto de saídas.
- **F** representa a função que gera o próximo estado.
- **G** representa a função que gera as saídas.

A função **F** atribui a cada combinação (estado, entradas) outro estado: **F:SxI→S**. A função **G** obtém o valor das saídas para o estado actual.

Existem 2 tipos de FSM, aos quais correspondem 2 definições diferentes para a função **G** que gera as saídas. Nas máquinas do tipo **Moore**, a função G baseia-se no estado: **G:S→O** e as saídas dependem apenas do estado da FSM. Nas máquinas do tipo **Mealy**, a função G baseia-se nas entradas e estado: **G:SxI→O** e as saídas dependem do estado e das entradas da FSM.

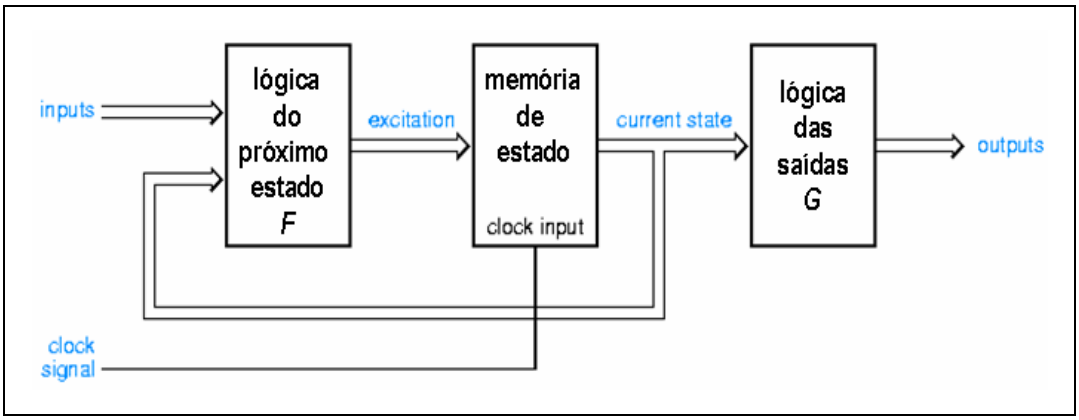
O meta-modelo FSM assume que o tempo está dividido em intervalos uniformes e que as transições ocorrem apenas no início de cada intervalo. Para definir esses intervalos, a que se chama ciclos de relógio, usa-se um sinal de relógio. Cada modelo FSM pode ser implementado com *flip-flops* e portas lógicas.

Genericamente, uma máquina de estados síncrona e do tipo Mealy possui a seguinte estrutura:



A **memória de estado** é implementada com um conjunto de *flip-flops* que guarda o estado actual da máquina. Os *flip-flops* partilham o sinal de relógio. As funções F e G são circuitos estritamente combinacionais.

Genericamente, uma máquina de estados síncrona e do tipo Moore possui a seguinte estrutura:



A única diferença entre as máquinas de Mealy e de Moore reside na forma como as saídas são geradas. Regra geral, uma máquina de Mealy tem algumas saídas de Mealy e outras de Moore. Para que a máquina seja o mais rápida possível, o bloco G tem que ser o mais simples possível (apenas fios). Para conseguir este objectivo, usam-se variáveis de estado que coincidem com as saídas.

Os passos envolvidos no projecto duma máquina de estados síncrona são:

- Analisar a descrição ou especificação em linguagem natural do sistema.
- Desenhar um diagrama de estados, usando nomes simbólicos para os estados.
- Construir a tabela de estados e saídas.
- [opcional] Minimizar o número de estados da tabela.
- Escolher um conjunto de variáveis de estado e atribuir uma combinação (dessas variáveis) a cada estado.
- Substituir na tabela o nome dos estados pela combinação (de variáveis) que lhe corresponde.
- Escolher um tipo de *flip-flop* para a memória de estado.
- Construir uma tabela de excitação que mostre quais os valores a aplicar na entrada dos *flip-flops* de modo a obter o próximo estado desejado, para cada combinação de estados e entradas.
- Obter a expressão para cada saída do bloco do próximo estado (excitação dos F/Fs).
- Obter a expressão para cada saída da FSM.

Exemplo duma máquina de estados:

Projectar uma máquina de estados que possui como entradas A e B e como saída Z. A saída Z é 1 se:

- A apresentar o mesmo valor nos 2 ciclos de relógio anteriores, ou
- B estiver fixo a 1 desde a última vez que a condição anterior se verificou.

Nos outros casos, a saída Z é 0.

Nesta fase, a especificação pode parecer pouco clara. O projectista deve transformar uma especificação ambígua, escrita em linguagem natural, numa tabela de estados sem ambiguidades. A máquina é do tipo Moore, uma vez que

as saídas só dependem do estado actual, ou seja, dependem apenas do que ocorreu nos ciclos de relógio anteriores.

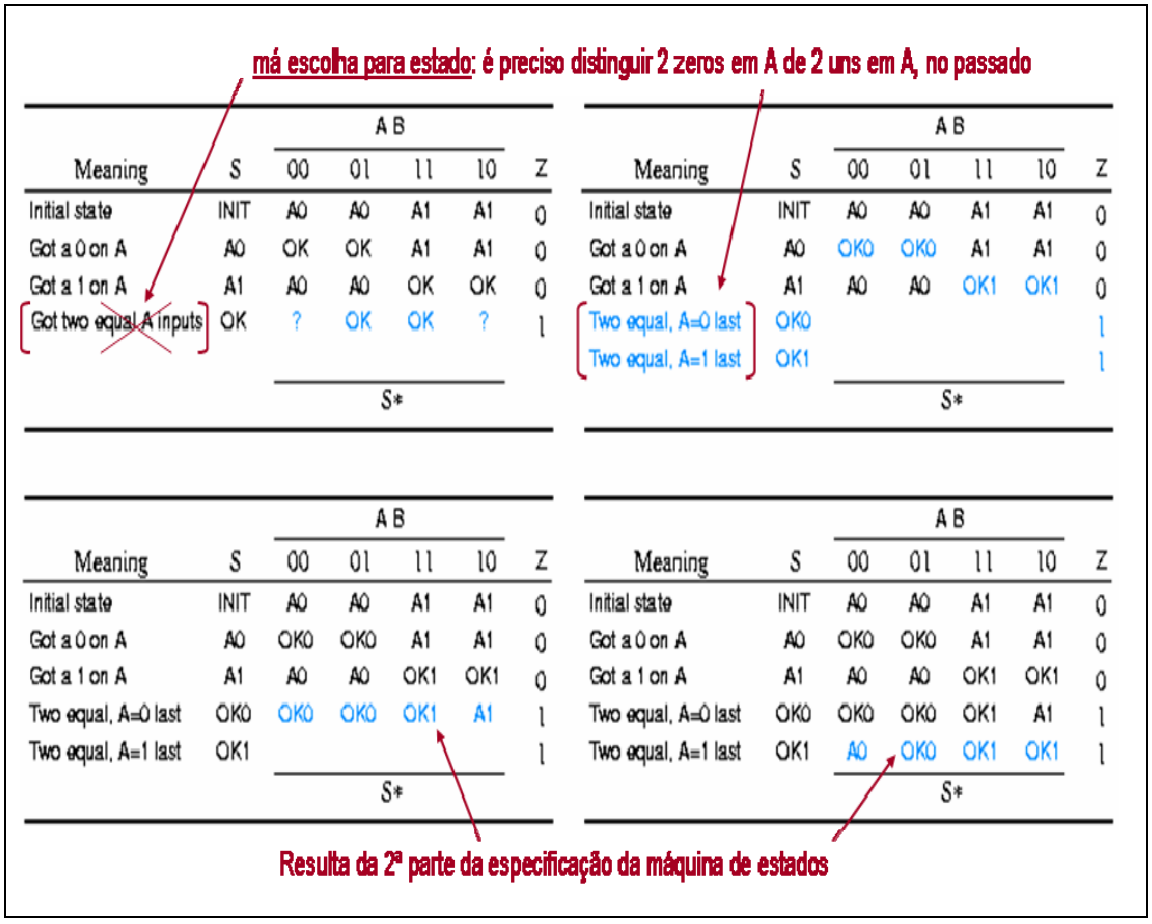
Processo de construção da tabela de estados e saídas (S=estado actual, S*=próx.estado):

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT					0
...						
...						
...						
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0					0
Got a 1 on A	A1					0
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1					0
Got two equal A inputs	OK					1
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK					1
		S*				



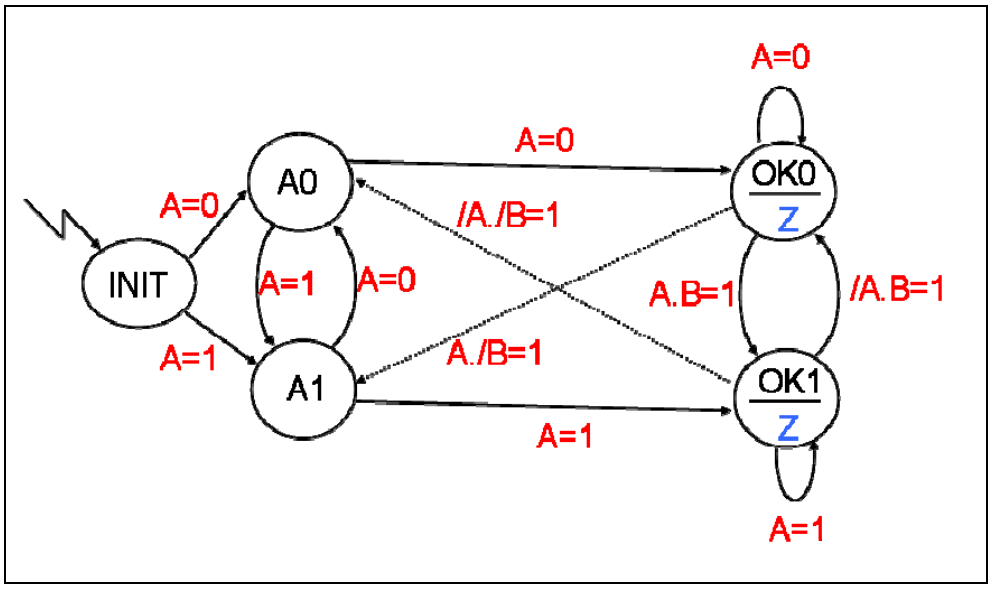
Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK	?	OK	OK	?	1
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0					1
Two equal, A=1 last	OK1					1
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1					1
		S*				

Meaning	S	A B 00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1	A0	OK0	OK1	OK1	1
		S*				

A máquina apresenta o seguinte diagrama de estados:



No próximo passo determina-se quantas variáveis binárias são necessárias para representar todos os estados da tabela de estados e saídas. De seguida, atribui-se uma combinação (das variáveis de estado binárias) a cada estado. A combinação atribuída a um estado particular é o código do estado.

Com **n** *flip-flops*, pode codificar-se $\mathbf{2^n}$ estados. O número de *flip-flops* necessário para codificar os **s** estados é $\lceil \mathbf{\log_2 s} \rceil$. No presente problema, como existem 5 estados, exige-se 3 *flip-flops*.

A tabela de estados e saídas é:

	AB				
S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
			S*		

Há várias **alternativas de codificação dos** 5 **estados**.

A **atribuição mais simples** dos **s** códigos aos estados consiste em usar os **s** valores binários iniciais que aparecem na contagem binária ordenada. Esta atribuição nem sempre é aquela que conduz às expressões mais simplificadas para as saídas do bloco que gera o próximo estado e para as saídas da FSM.

	Assignment			
State Name	Simplest Q1–Q3	Decomposed Q1–Q3	One-hot Q1–Q5	Almost One-hot Q1–Q4
INIT	000	000	00001	0000
A0	001	100	00010	0001
A1	010	101	00100	0010
OK0	011	110	01000	0100
OK1	100	111	10000	1000

A atribuição de códigos aos estados tem um forte impacto no "custo" (área) do circuito e deve ter em consideração a selecção dos elementos de memória e a abordagem a usar na concretização dos 2 blocos de lógica combinacional (próximo estado e saídas). Na maior parte dos problemas, para escolher a melhor atribuição de códigos aos estados, seria preciso experimentar todas as atribuições possíveis. Não é exequível para fazer isto manualmente!!! No presente exemplo, existem 6.720 formas diferentes de atribuir 5 (das 8) combinações de 3-bits aos 5 estados. Em alternativa, o projectista deve guiar-se pela sua experiência e por orientações práticas para obter uma boa atribuição de códigos aos estados.

Orientações para a atribuição de códigos aos estados:

- Escolher um código inicial que seja fácil de forçar com o mecanismo de *reset* dos F/Fs (normalmente será 000...0 ou 111...1).
- Minimizar o número de variáveis de estado que mudam em cada transição de estado.
- Maximizar o número de variáveis de estado que se mantêm inalteradas num grupo de estados interrelacionados.
- Explorar as simetrias existentes na especificação do problema e as simetrias que lhe correspondem na tabela de estados. Por exemplo, se um estado (ou grupo de estados) tiver um significado semelhante a outro estado (grupo), após escolher a atribuição para o primeiro estado (grupo) deve escolher-se uma atribuição similar (diferindo em 1,2,… bits) para o segundo.
- Decompor o conjunto de variáveis de estado em bits ou campos de bits, cada um relacionado com um aspecto funcional das entradas ou das saídas da FSM.
- Ponderar a hipótese de usar mais variáveis de estado do que o valor mínimo, por forma a tornar possível a decomposição anterior.

Algumas das orientações anteriores foram incorporadas na atribuição do tipo "**decomposed**". Ao estado **INIT** foi atribuído o código **000**, um código fácil de forçar através da entrada assíncrona dos flip-flops (CLR). O estado INIT nunca mais é acedido após o arranque da máquina. Por isso, usou-se o bit (variável de estado) **Q1** para indicar se o estado actual é ou não INIT. **Q2** e **Q3** permitem

distinguir os outros 4 estados. **Q2** está relacionado com o facto de a saída ser 1 ou não no estado actual e **Q3** está relacionado com o valor anterior de A.

State Name	Decomposed Q1–Q3
INIT	000
A0	100
A1	101
OK0	110
OK1	111

A atribuição do tipo **one-hot** pode ser aplicada a qualquer máquina de estados. Este tipo de atribuição utiliza mais variáveis de estado do que o valor mínimo: usa 1 bit por estado. Esta opção conduz normalmente a expressões simples para as saídas dado que cada *flip-flop* só é colocado a 1 nas transições para um único estado.

A atribuição do tipo **almost one-hot** segue a estratégia *one-hot* excepto para o estado inicial, em que se usa a combinação 00…0.

State Name	One-hot Q1–Q5	Almost One-hot Q1–Q4
INIT	00001	0000
A0	00010	0001
A1	00100	0010
OK0	01000	0100
OK1	10000	1000

No presente exemplo existem códigos de estado não usados porque o número de estados é menor do que o número de combinações das variáveis de estado. Como lidar com os códigos não usados?

Numa abordagem com risco mínimo, considera-se que a máquina pode atingir um estado com código não usado, por exemplo ao ocorrer uma falha no *hardware*. Para qualquer estado com código não usado, implementa-se uma transição explícita para regressar a um estado seguro. Numa abordagem com custo mínimo, considera-se que a máquina nunca atinge um estado com código não usado. Nos códigos de estado não usados, as entradas na tabela de estados relativas ao próximo estado podem ser definidas como “don’t care”.

Para obter a versão final da tabela de (transições de) estados e saídas substitui-se o nome dos estados pelo seu código. A tabela obtida mostra qual é o próximo estado para cada combinação (estado actual, entradas). Para o

exemplo em estudo, a tabela de transições de estados e saídas foi obtida efectuando uma atribuição do tipo *decomposed*.

State Name	Decomposed Q1–Q3
INIT	000
A0	100
A1	101
OK0	110
OK1	111

i) Atribuição do tipo *decomposed* usada.

estado actual — entradas — saída

	AB				
Q1Q2Q3	00	01	11	10	Z
000	100	100	101	101	0
100	110	110	101	101	0
101	100	100	111	111	0
110	110	110	111	101	1
111	100	110	111	111	1
	Q1*Q2*Q3*				

próximo estado

ii) Tabela de transições de estados e saídas.

No próximo passo obtém-se a **tabela de excitação**. Esta tabela define quais os valores a aplicar nas entradas dos *flip-flops* por forma a que a máquina transite para o próximo estado desejado. A estrutura e o conteúdo da tabela de excitação dependem do tipo de *flip-flop* escolhido para implementar a memória de estado (D, J-K, T, ...). Actualmente, a maioria dos projectos com máquinas de estado utiliza *flip-flops* D, por existirem em todas as tecnologias (CIs SSI, PLDs, FPGAs) e por serem fáceis de usar.

A equação característica do *flip-flop* D é: Q* = D. No caso de se usar *flip-flops* D, a tabela de excitação é idêntica à tabela de transições de estados, excepto os nomes das colunas que deixam de ser o próximo estado (Q1* Q2* Q3*) para ser a entrada dos F/Fs (D1 D2 D3).

	AB				
Q1Q2Q3	00	01	11	10	Z
000	100	100	101	101	0
100	110	110	101	101	0
101	100	100	111	111	0
110	110	110	111	101	1
111	100	110	111	111	1
	D1 D2 D3				

Se estiver completa, a tabela de excitação funciona como tabela de verdade das saídas combinacionais D1, D2 e D3, ao exprimi-las em função das entradas A e B e do estado Q1, Q2 e Q3. Se estiver completa, a tabela anterior também funciona como tabela de verdade das saídas da FSM, ao exprimir a saída combinacional Z em função do estado Q1, Q2 e Q3.
Com base na informação da tabela de excitação, pode usar-se mapas de Karnaugh para obter a expressão mínima para cada saída combinacional. A tabela de excitação apresentada não está completa porque não especifica o valor do próximo estado e da saída, em todas as combinações (estado, entradas). Ou seja, a informação relativa aos estados não usados não consta da tabela.

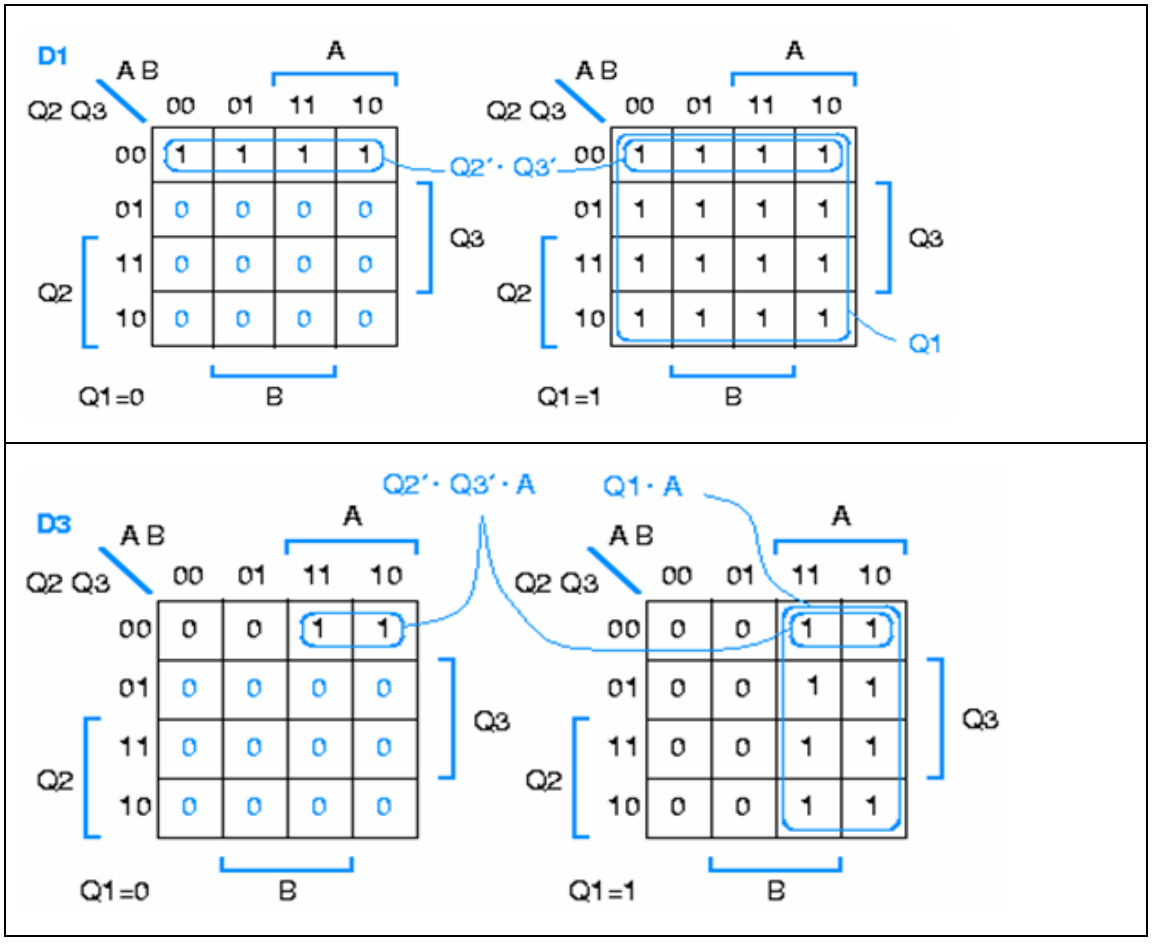
No presente exemplo, o tratamento dos estados não usados seguirá as duas abordagens mencionadas: risco mínimo e custo mínimo. Na abordagem de risco mínimo, em qualquer estado não usado (Q1Q2Q3 ≡ 001, 010, 011) o próximo estado é INIT ≡ 000.

A partir dos mapas em baixo (mais o mapa de D2 que falta) obtém-se as expressões:

D1 = Q1 + Q2'· Q3'
D2 = Q1·Q3'·A'+Q1·Q3·A+Q1·Q2·B
D3 = Q1· A + Q2'· Q3'· A

A tabela de excitação mostra que **Z** está activa nos estados 110 e 111:

Z = Q1· Q2· Q3' + Q1· Q2· Q3
= Q1· Q2



Na abordagem de custo mínimo, em qualquer estado não usado (Q1Q2Q3 ≡ 001, 010, 011) o próximo estado é *don't care*. A partir dos mapas em baixo (mais o mapa de D2 que falta) obtém-se as expressões:

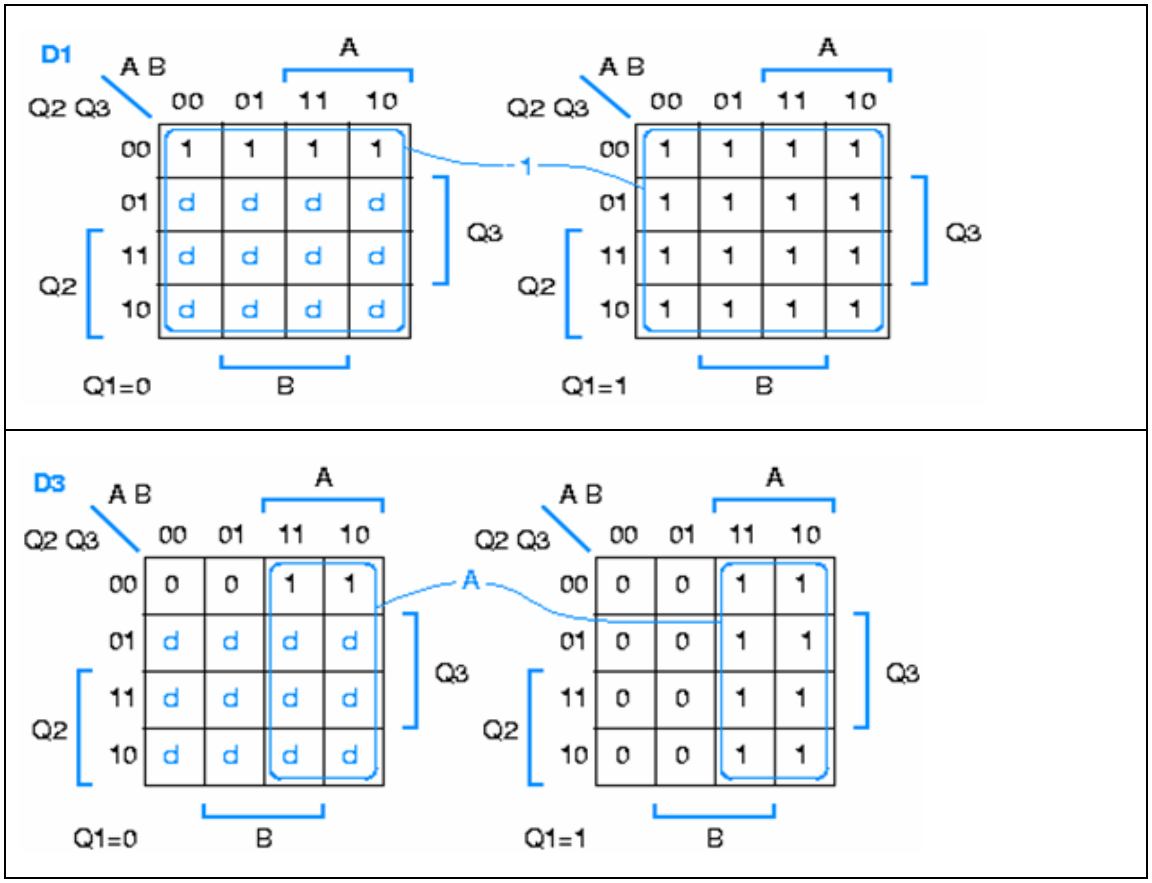
D1 = 1

D2 = Q1·Q3'·A'+Q3·A+Q2·B

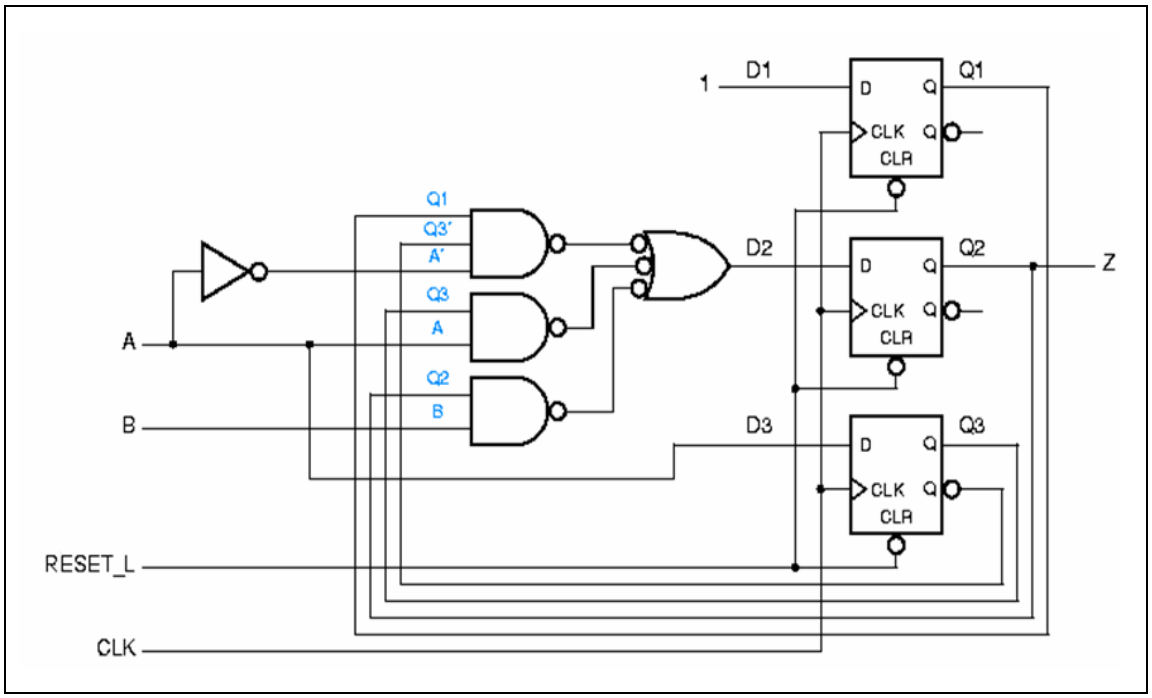
D3 = A

Como Z está activa nos estados 110 e 111 e é *don't care* nos estados não usados, obtém-se:

Z = Q2

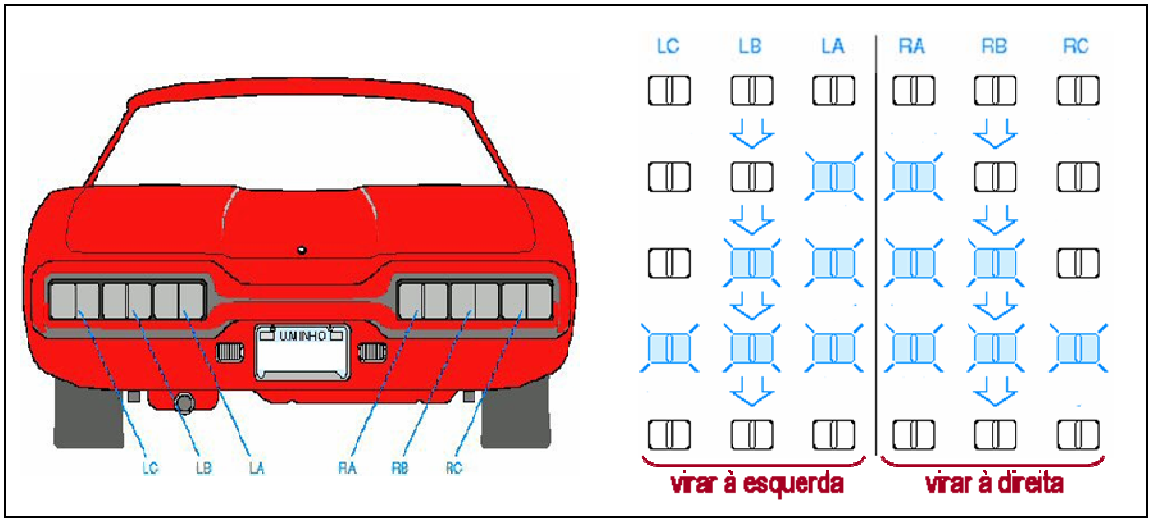


O diagrama lógico da máquina de estados projectada com a abordagem de custo mínimo encontra-se na próxima figura.



É frequente usar **diagramas de estados** para projectar máquinas de estados. Projectar um diagrama de estados é similar, mas ainda mais simples, do que escrever uma tabela de estados. Contudo, é fácil conceber um diagrama de estados com algumas ambiguidades, que são impossíveis de ocorrer numa tabela de estados. Num diagrama de estados incorrectamente projectado, é comum o próximo estado não ser especificado em certas combinações (estado, entradas). Também pode ocorrer a situação em que alguma combinação (estado, entradas) permite transitar para mais do que um estado.

O próximo exemplo é uma máquina de estados que controla as luzes de trás num carro. Existem 3 luzes do lado direito (RA, RB e RC) e outras 3 do lado esquerdo (LA, LB e LC). A máquina de estados possui 2 entradas LEFT / RIGHT, indicando um pedido para virar à esquerda / direita. Possui também uma entrada de emergência HAZ indicando um pedido para as 6 luzes piscarem.

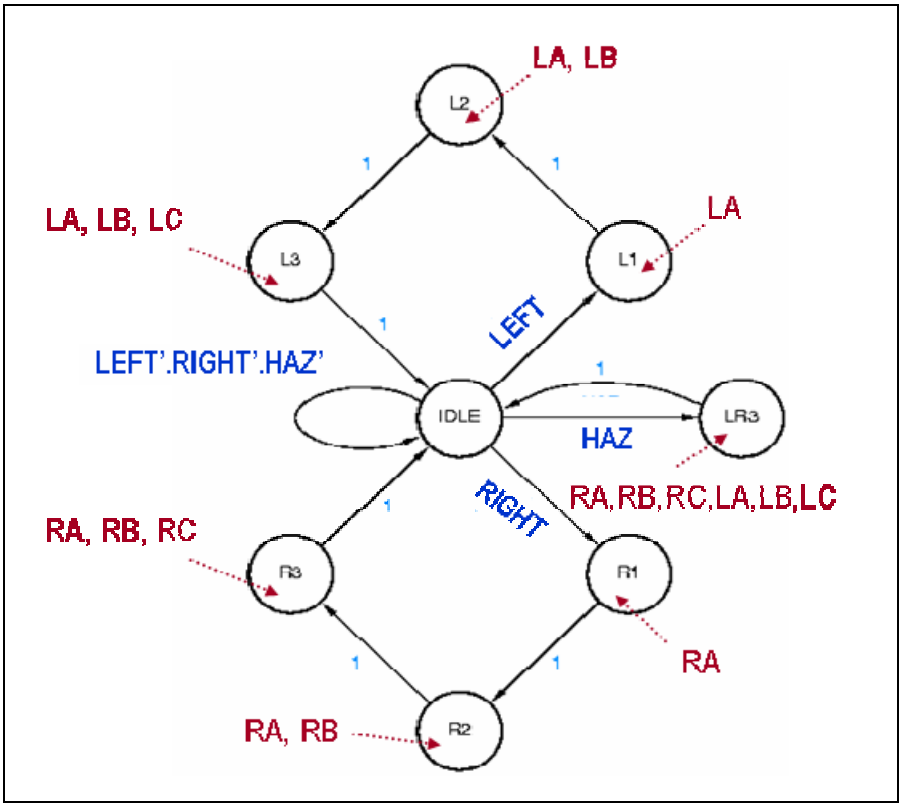


O diagrama de estados e a tabela das saídas (auxiliar do diagrama de estados) para o controlador de luzes do carro são agora apresentados:

todas as luzes apagadas

Output Table

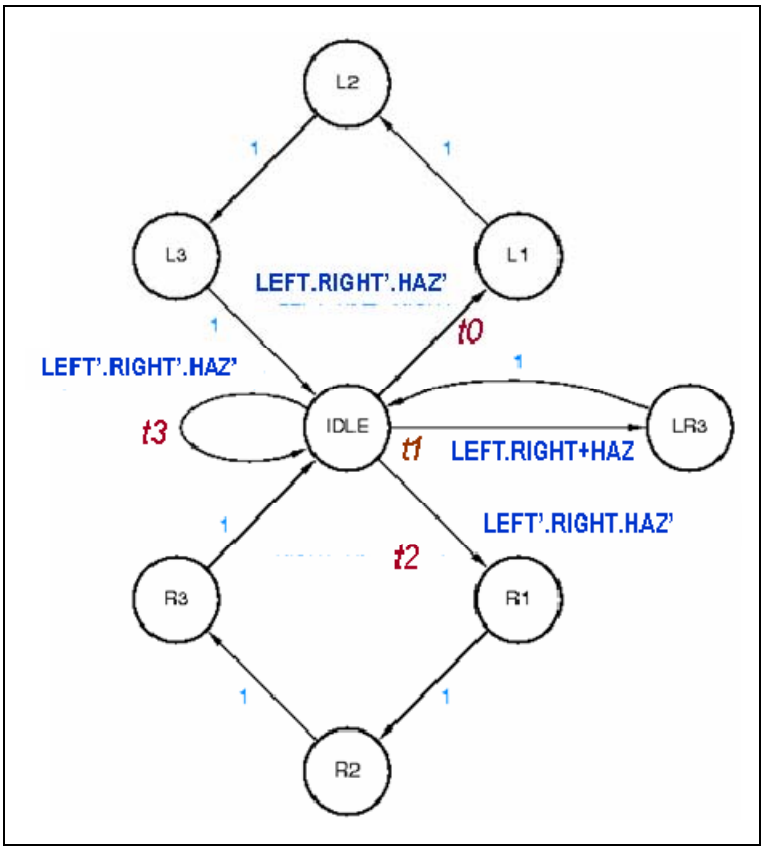
State	LC	LB	LA	RA	RB	RC
IDLE	0	0	0	0	0	0
L1	0	0	1	0	0	0
L2	0	1	1	0	0	0
L3	1	1	1	0	0	0
R1	0	0	0	1	0	0
R2	0	0	0	1	1	0
R3	0	0	0	1	1	1
LR3	1	1	1	1	1	1



Este diagrama é limitado, só está correcto se não houver activação simultânea de várias entradas. O diagrama de estados deve ser alterado para:

- Atribuir uma prioridade superior à entrada HAZ.
- Considerar a activação simultânea de LEFT e RIGHT como uma situação de emergência.

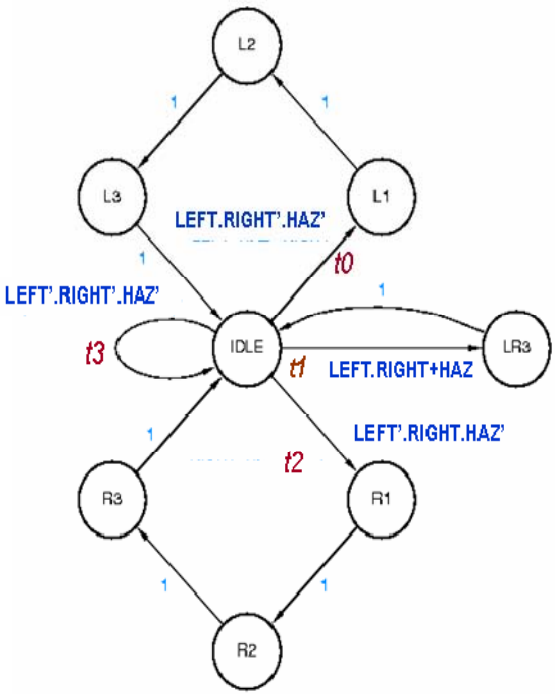
O novo diagrama de estados já não tem ambiguidades porque as condições associadas às várias transições que partem do mesmo estado são mutuamente exclusivas entre si e no conjunto cobrem todas as combinações das entradas.



Convém demonstrar que o novo diagrama de estados não tem ambiguidades:

LEFT	RIGHT	HAZ	Transição	Próximo Estado
0	0	0	*t3*	IDLE
0	0	1	*t1*	LR3
0	1	0	*t2*	R1
0	1	1	*t1*	LR3
1	0	0	*t0*	L1
1	0	1	*t1*	LR3
1	1	0	*t1*	LR3
1	1	1	*t1*	LR3

Transições a partir do estado **IDLE.**

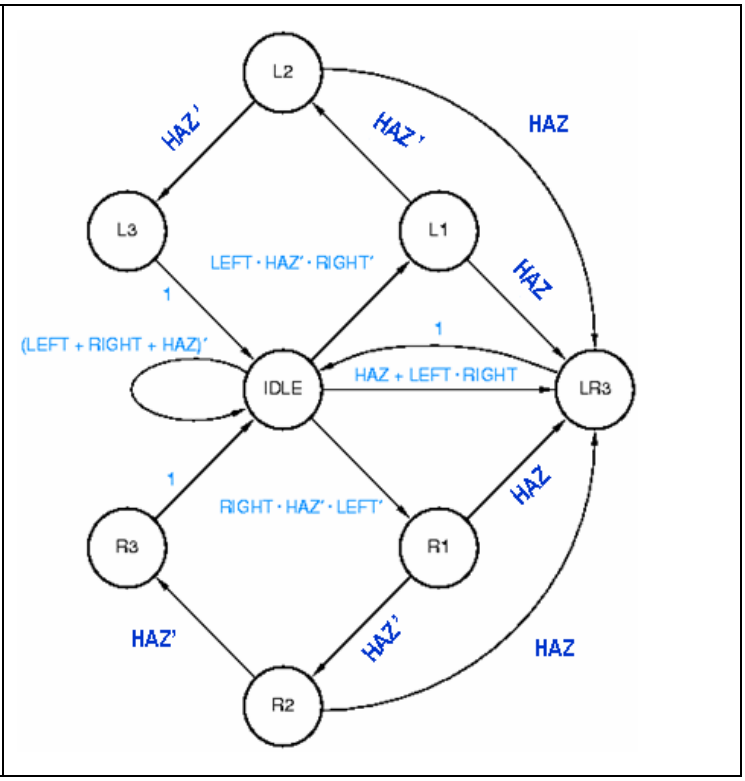


Antes de sintetizar a FSM, vai introduzir-se uma alteração: permite-se que a FSM vá para o estado LR3 o mais cedo possível, ou seja, antes de terminar uma sequência L1→L2→L3 ou R1→R2→R3 já iniciada, desde que tenha ocorrido um pedido de emergência através da activação de HAZ.

A FSM possui 8 estados, logo exige-se 3 *flip-flops* para a memória de estado. Convém explicar a atribuição de códigos aos estados:

- IDLE=000.
- Q2 distingue as partes simétricas R1→R2→R3 e L1→L2→L3.
- Usa-se código gray em Q1Q0 nas sequências IDLE→L1→L2→L3 e IDLE→R1→R2→R3 para minimizar as transições nas variáveis de estado.

State	Q2	Q1	Q0
IDLE	0	0	0
L1	0	0	1
L2	0	1	1
L3	0	1	0
R1	1	0	1
R2	1	1	1
R3	1	1	0
LR3	1	0	0



Apresenta-se agora a tabela de transições de estado. A tabela foi alterada para ser mais compacta, passando de **64** para **15** linhas. Isto foi conseguido substituindo as combinações das entradas por expressões.

S	Q2	Q1	Q0	Transition Expression	S+	Q2+	Q1+	Q0+
IDLE	0	0	0	(LEFT + RIGHT + HAZ)'	IDLE	0	0	0
IDLE	0	0	0	LEFT · HAZ' · RIGHT'	L1	0	0	1
IDLE	0	0	0	HAZ + LEFT · RIGHT	LR3	1	0	0
IDLE	0	0	0	RIGHT · HAZ' · LEFT'	R1	1	0	1
L1	0	0	1	HAZ'	L2	0	1	1
L1	0	0	1	HAZ	LR3	1	0	0
L2	0	1	1	HAZ'	L3	0	1	0
L2	0	1	1	HAZ	LR3	1	0	0
L3	0	1	0	1	IDLE	0	0	0
R1	1	0	1	HAZ'	R2	1	1	1
R1	1	0	1	HAZ	LR3	1	0	0
R2	1	1	1	HAZ'	R3	1	1	0
R2	1	1	1	HAZ	LR3	1	0	0
R3	1	1	0	1	IDLE	0	0	0
LR3	1	0	0	1	IDLE	0	0	0

6.6. Máquinas de estados em VHDL

A linguagem VHDL não possui qualquer construtor específico para descrever máquinas de estados. A maioria dos conceitos, necessários para descrever máquinas de estados síncronas em VHDL, já foi apresentada. O **processo** e o mecanismo da **lista de eventos** (usado pelos simuladores para guardar o historial das mudanças que ocorrem nos sinais) são o suporte base para descrever circuitos sequenciais em VHDL. Para modelar o comportamento "síncrono com o relógio" exige-se apenas uma característica do VHDL ainda não introduzida: o atributo **event**, que pode ser anexado ao nome dum sinal para originar um valor que será igual a **true** se o sinal tiver mudado de valor e é igual a **false** em caso contrário.
É frequente usarem-se tipos de dados enumerados e o construtor CASE para descrever máquinas de estados.

Pode usar-se o atributo **event** para modelar um *flip-flop* D

- Com um reset assíncrono (CLR) e
- Sensível às transições positivas do relógio (CLK).

A influência da entrada de reset sobrepõe-se ao comportamento que depende da entrada de relógio. Para qualquer transição do sinal CLK (0→1 ou 1→0), CLK'event é igual a true.

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

entity VposDff is
  port (CLK, CLR, D: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end VposDff;

architecture VposDff_arch of VposDff is
begin
  process (CLK, CLR)
  begin
    if CLR='1' then Q <= '0'; QN <= '1';
    elsif CLK'event and CLK='1' then Q <= D; QN <= not D;
    end if;
  end process;
end VposDff_arch;
```

A seguir apresentam-se mais 2 formas de descrever um *flip-flop* D sensível às transições positivas, mas sem entrada de *reset*.

```
process
  wait until CLK'event and CLK='1';
  Q <= D;
end process;

Q <= D when CLK'event and CLK='1' else Q;
```

Em VHDL, é possível descrever de várias formas uma máquina de estados como a do 1º exemplo apresentado neste capítulo. A primeira abordagem consiste em construir a tabela de estados e saídas no papel e depois convertê-la manualmente para código VHDL. Esta tabela já foi obtida antes e é repetida a seguir. A 1ª coisa a fazer nesta abordagem é definir um tipo enumerado Sreg-type, cujos valores permitidos são identificadores correspondentes ao nome dos vários estados: INIT, A0, A1, OK0 e OK1. Depois declara-se um sinal Sreg desse tipo enumerado, utilizado para guardar o estado actual da máquina.

O corpo da arquitectura possui duas instruções concorrentes:

- Um processo que reage apenas ao sinal CLOCK e implementa todas as transições de estado.
- Uma atribuição selectiva que gera a saída Z da máquina de Moore.

	AB				
S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
			S*		

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

entity smexamp is
  port ( CLOCK, A, B: in STD_LOGIC;
         Z: out STD_LOGIC );
end;

architecture smexamp_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
signal Sreg: Sreg_type;
begin

  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = '1' then
      case Sreg is
        when INIT => if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= A1;   end if;
        when A0 =>   if    A='0' then Sreg <= OK0;
                     elsif A='1' then Sreg <= A1;  end if;
        when A1 =>   if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= OK1;  end if;
        when OK0 =>  if    A='0' then Sreg <= OK0;
                     elsif A='1' and B='0' then Sreg <= A1;
                     elsif A='1' and B='1' then Sreg <= OK1;  end if;
        when OK1 =>  if    A='0' and B='0' then Sreg <= A0;
                     elsif A='0' and B='1' then Sreg <= OK0;
                     elsif A='1' then Sreg <= OK1;            end if;
        when others => Sreg <= INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <= '0' when INIT | A0 | A1,
         '1' when OK0 | OK1,
         '0' when others;

end smexamp_arch;
```

Como é feita a atribuição de códigos aos estados? Ou seja, qual a estratégia e qual o número de variáveis de estado usados?

A ferramenta de síntese tem a liberdade de substituir os identificadores dos estados pelos valores inteiros (ou binários) que desejar. Regra geral, a estratégia seguida para atribuir os códigos é do tipo mais simples, usando códigos cujo valor segue a ordem pela qual os estados estão listados no tipo enumerado. As ferramentas de síntese actuais permitem ao projectista usar uma codificação diferente.

Uma das formas é usar a interface (gráfica) da ferramenta para escolher o tipo de codificação. A outra consiste em usar uma instrução **attribute** (ver (1) na figura seguinte). O atributo definido pelo utilizador **enum_encoding** exige a biblioteca SYNOPSYS e é passado à ferramenta de síntese (ver (2) na figura seguinte).

```
library IEEE;
use IEEE.std_logic_1164.all;
(1) library SYNOPSYS;
    use SYNOPSYS.attributes.all;
...
architecture smexampe_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
(2) attribute enum_encoding of Sreg_type: type is
          "0000 0001 0010 0100 1000";
signal Sreg: Sreg_type;
...
```

Outra forma de forçar um tipo de codificação dos estados consiste em definir o registo de estado mais explicitamente. Neste exemplo, define-se um subtipo **Sreg_type** baseado em **std_logic** com a largura desejada para o registo de estado. Depois, é possível definir uma constante desse subtipo para cada estado. Deste modo, força-se o tipo de codificação (**valores**) e os estados continuam a ser acedidos por nomes simbólicos (**nomes**).

```
library IEEE;
use IEEE.std_logic_1164.all;
...
architecture smexampc_arch of smexamp is
subtype Sreg_type is STD_LOGIC_VECTOR (1 to 4);
constant INIT: Sreg_type := "0000";
constant A0  : Sreg_type := "0001";
constant A1  : Sreg_type := "0010";
constant OK0 : Sreg_type := "0100";
constant OK1 : Sreg_type := "1000";
signal Sreg: Sreg_type;
...
```

Exemplo do sistema que controla as luzes de trás dum carro (**página 110**), em que a máquina de estados é agora descrita em VHDL. Usam-se códigos de estado iguais aos valores das saídas **(página 108),** dispensando assim a lógica para gerar as saídas.

```
entity Vtbird is
  port ( CLOCK, RESET, LEFT, RIGHT, HAZ: in STD_LOGIC;
         LIGHTS: buffer STD_LOGIC_VECTOR (1 to 6) );
end;

architecture Vtbird_arch of Vtbird is
constant IDLE: STD_LOGIC_VECTOR (1 to 6) := "000000";
constant L3  : STD_LOGIC_VECTOR (1 to 6) := "111000";
constant L2  : STD_LOGIC_VECTOR (1 to 6) := "110000";
constant L1  : STD_LOGIC_VECTOR (1 to 6) := "100000";
constant R1  : STD_LOGIC_VECTOR (1 to 6) := "000001";
constant R2  : STD_LOGIC_VECTOR (1 to 6) := "000011";
constant R3  : STD_LOGIC_VECTOR (1 to 6) := "000111";
constant LR3 : STD_LOGIC_VECTOR (1 to 6) := "111111";
begin
  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then LIGHTS <= IDLE; else
        case LIGHTS is
          when IDLE => if HAZ='1' or (LEFT='1' and RIGHT='1') then LIGHTS <= LR3;
                       elsif LEFT='1'                         then LIGHTS <= L1;
                       elsif RIGHT='1'                        then LIGHTS <= R1;
                       else                                        LIGHTS <= IDLE;
                       end if;
          when L1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L2; end if;
          when L2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L3; end if;
          when L3   => LIGHTS <= IDLE;
          when R1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R2; end if;
          when R2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R3; end if;
          when R3   => LIGHTS <= IDLE;
          when LR3  => LIGHTS <= IDLE;
          when others => null;
        end case;
      end if;
    end if;
  end process;
end Vtbird_arch;
```

7. Questões Práticas de Sistemas Sequenciais

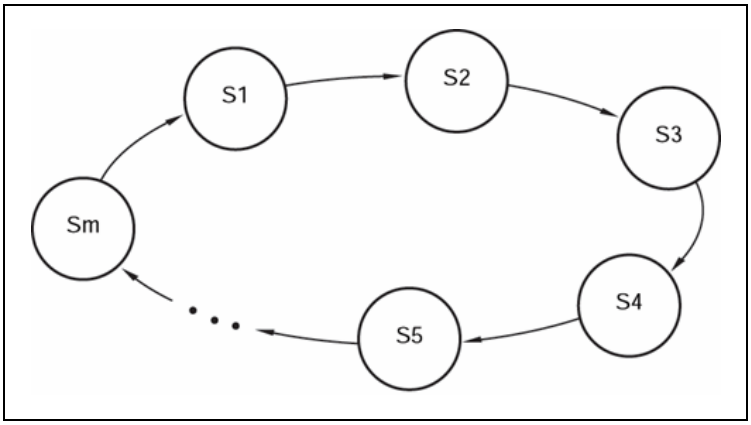
7.1. Introdução

A maior parte dos sistemas digitais reais é do tipo sequencial. O comportamento sequencial resulta de ciclos de realimentação, de *latches* ou de *flip-flops* que fazem o estado actual do sistema depender do historial das entradas. Para que o número de estados de alguns sistemas digitais reais (como um computador) não atinja um valor extremamente elevado, é habitual decompô-los em partes, como por exemplo: caminhos de dados (*data paths*), bancos de registos e unidades de controlo. Um sistema típico possui várias unidades funcionais com uma interface bem conhecida para elas poderem ser interligadas. Por sua vez, uma unidade funcional pode também ser construída à custa de blocos mais simples passíveis de ser tratados de forma hierárquica e em níveis de abstracção diferentes.

Apesar de os sistemas digitais poderem ser muito complexos, a maior parte deles é construído em torno de uma máquina de estados. Durante a década de 80, o processo utilizado para projectar uma máquina de estados era o que se usou no 1º exemplo do módulo anterior. Ou seja, escrevia-se a tabela de estados e saídas que após algumas transformações gerava a tabela de verdade para cada saída combinacional. A síntese manual da máquina terminava com a simplificação das expressões das saídas através de mapas de karnaugh (ou de outro método qualquer). Actualmente, a maior parte dos projectistas descreve a tabela de estados e saídas das máquinas de estados numa HDL (como ABEL, VHDL ou Verilog). Uma ferramenta de síntese para essa HDL processa a descrição da máquina de modo a efectuar uma síntese equivalente à efectuada manualmente e obtém uma implementação numa determinada tecnologia (PLD, CPLD, FPGA ou ASIC) escolhida, desde que seja suportada pela ferramenta.

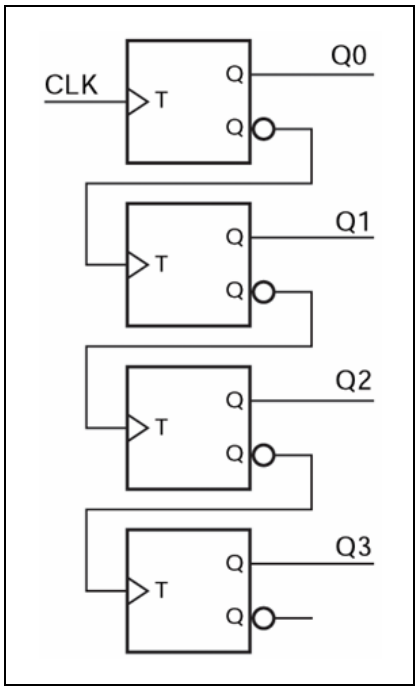
7.2. Contadores

Chama-se **contador** a um circuito sequencial síncrono que é descrito por um diagrama de estados que possui apenas um ciclo. O ***modulus*** do contador coincide com o número de estados desse ciclo.



A um contador com **m** estados chama-se um contador módulo-m. Um contador com um modulus que não é potência de 2 possui alguns estados que não são visitados no modo normal de funcionamento.

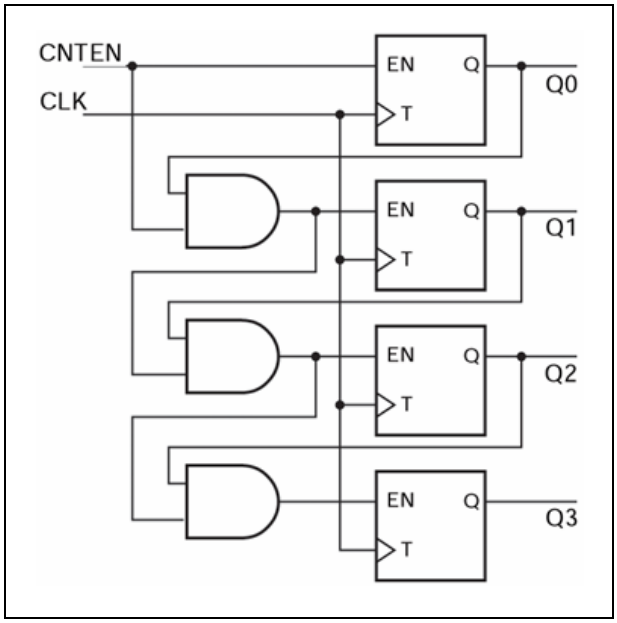
O tipo de contador mais comum é o **contador binário de n-bits**. Este contador é implementado com **n** *flip-flops* e possui $\mathbf{2^n}$ estados, os quais são visitados seguindo a sequência 0, 1, 2,…, 2^n-1, 0, 1, … Cada estado é codificado usando o valor binário de **n**-bits que correspondente à posição desse estado na sequência 0, 1, 2,…, 2^n-1. Um exemplo: se um estado estiver na posição $0 \leq \mathbf{p} \leq 2^n-1$, o seu código é o valor binário que representa **p** em n-bits.
Um contador binário de **n**-bits pode ser construído apenas com **n** *flip-flops* tipo T, sem qualquer tipo de lógica combinacional. Numa implementação deste tipo, cada bit $\mathbf{Q_i}$ do contador muda quando o bit $\mathbf{Q_{i-1}}$ imediatamente antes (e menos significativo) mudar de 1 para 0. O contador é designado por **contador de *ripple*** porque a informação de transporte é propagada desde o LSB até ao MSB.



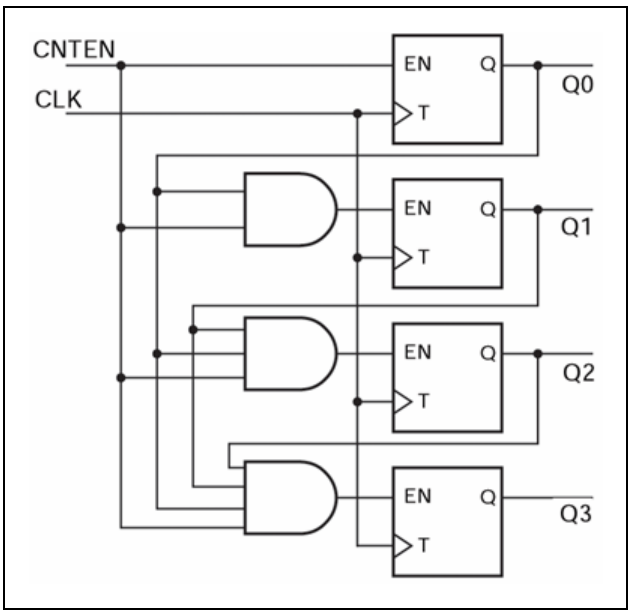
Apesar de exigir menos lógica do que qualquer outro tipo de contador, o contador de *ripple* também é mais lento. O pior cenário, que ocorre quando o MSB tiver que mudar, a saída só é válida passados $\mathbf{n*t_{TQ}}$ sobre a transição ascendente do relógio.(em que $\mathbf{t_{TQ}}$ é o tempo de atraso que a entrada demora a afectar a saída dum *flip-flop* tipo T).
Num contador síncrono, o sinal de relógio de todos os *flip-flops* está ligado a um sinal CLK comum. Deste modo, a saída de todos os *flip-flops* muda de valor no mesmo instante, passado um atraso t_{TQ} após a transição ascendente de CLK.

Um **contador síncrono de *ripple*** exige que se use *flip-flops* T com *enable*. Assim, a saída Q muda de valor na transição ascendente da entrada T mas apenas se a entrada EN estiver activa. A saída Q de cada *flip-flop* T muda de

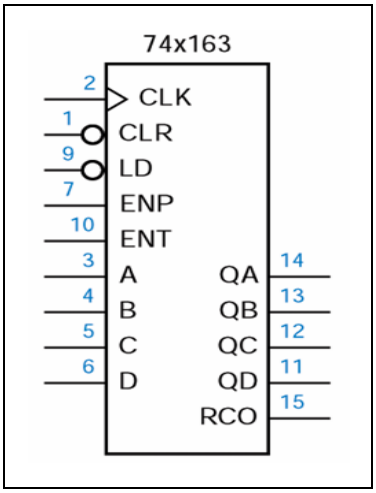
valor se CNTEN estiver activa e todas as saídas Q de ordem inferior forem **1**. Tal como o contador de *ripple*, o contador síncrono também pode ser construído com uma quantidade de lógica por bit fixa. No exemplo seguinte, um *flip-flop* T com *enable* e uma porta AND de 2 entradas. A este tipo de contador também se chama **contador série síncrono** porque o sinal de *enable* se propaga em série desde o LSB até ao MSB.



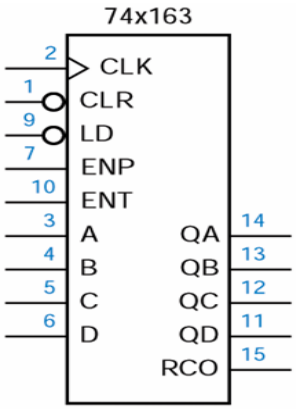
Se o período do relógio for demasiado reduzido, poderá não dar tempo para que o sinal de *enable* se propague desde o LSB até ao MSB. Este problema é eliminado se cada entrada EN for gerada por uma porta AND própria sem recurso aos EN's de ordem inferior (ver figura em baixo). A alteração introduzida resulta no tipo de contador com estrutura mais rápida: o **contador síncrono paralelo**.



O CI **74x163** contém um contador síncrono binário de 4-bits. O contador é construído com *flip-flops* tipo D. A entrada síncrona LD (***load***) permite colocar o valor das entradas D-C-B-A nas saídas Q. A entrada síncrona CLR (***clear***) permite fazer o *reset* das saídas Q. A saída RCO (***ripple carry*** *do MSB*) é 1, durante um ciclo de relógio, quando todas as saídas Q e a entrada **ENT** estiverem a 1.

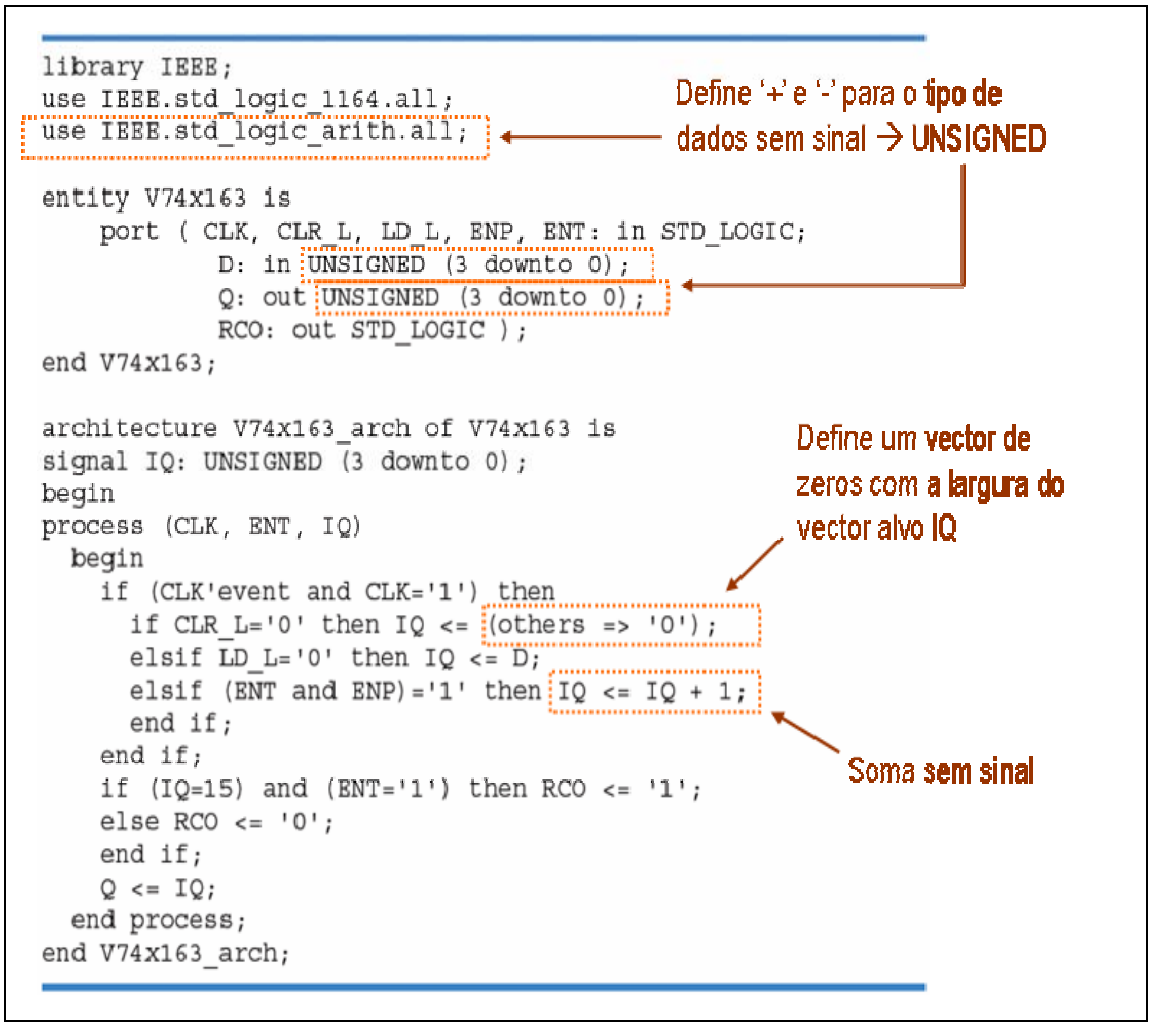


Apresenta-se agora o símbolo e a tabela de estados do contador síncrono binário de 4-bits 74x163.



Inputs				*Current State*				*Next State*			
CLR_L	*LD_L*	*ENT*	*ENP*	*QD*	*QC*	*QB*	*QA*	*QD**	*QC**	*QB**	*QA**
0	x	x	x	x	x	x	x	0	0	0	0
1	0	x	x	x	x	x	x	D	C	B	A
1	1	0	x	x	x	x	x	QD	QC	QB	QA
1	1	x	0	x	x	x	x	QD	QC	QB	QA
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1
1	1	1	1	0	0	1	1	0	1	0	0
1	1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	0	1	0	1	0	1	1	0
1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0	1	0	0	1
1	1	1	1	1	0	0	1	1	0	1	0
1	1	1	1	1	0	1	0	1	0	1	1
1	1	1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	0	0	1	1	0	1
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0

A próxima figura contém a descrição em VHDL comportamental para o contador síncrono binário de 4-bits 74x163.



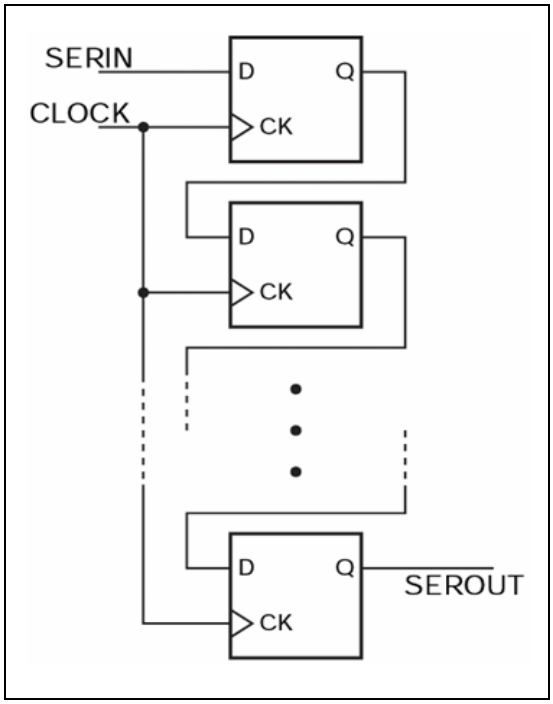

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity V74x163 is
    port ( CLK, CLR_L, LD_L, ENP, ENT: in STD_LOGIC;
           D: in UNSIGNED (3 downto 0);
           Q: out UNSIGNED (3 downto 0);
           RCO: out STD_LOGIC );
end V74x163;

architecture V74x163_arch of V74x163 is
signal IQ: UNSIGNED (3 downto 0);
begin
process (CLK, ENT, IQ)
  begin
    if (CLK'event and CLK='1') then
      if CLR_L='0' then IQ <= (others => '0');
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=15) and (ENT='1') then RCO <= '1';
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74x163_arch;
```

A figura que se segue descreve um contador semelhante ao 74x163 mas que conta em excesso de 3 (3, 4, …, 11, 12, 3, …).

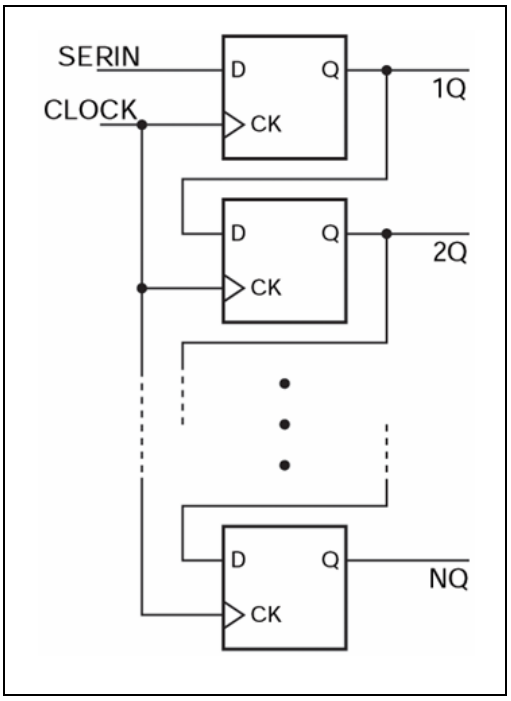
```
architecture V74xs3_arch of V74x163 is
signal IQ: UNSIGNED (3 downto 0);
begin
process (CLK, ENT, IQ)
  begin
    if CLK'event and CLK='1' then
      if CLR_L='0' then IQ <= (others => '0');
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' and (IQ=12) then IQ <= ('0','0','1','1');
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=12) and (ENT='1') then RCO <= '1';
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74xs3_arch;
```

7.3. Registos de deslocamento (shift registers)

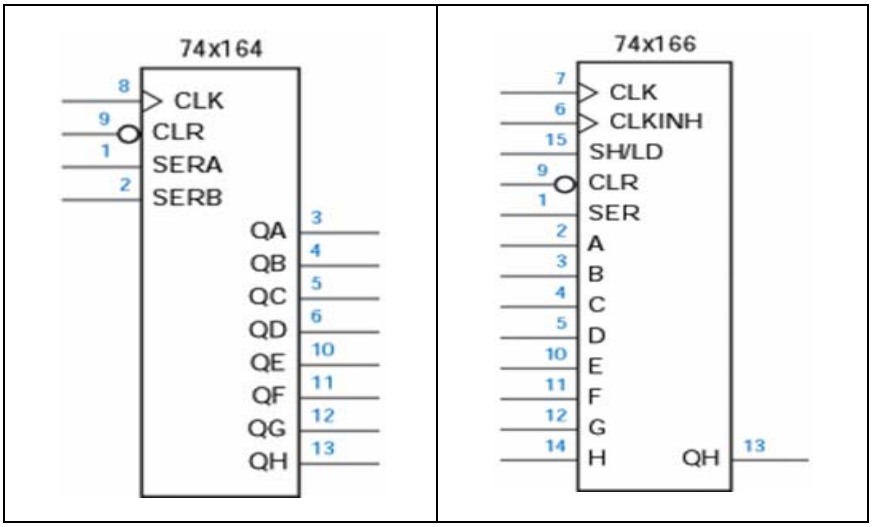
Um **registo de deslocamento** é um registo de **n**-bits com a capacidade de deslocar em uma posição o valor armazenado, em cada ciclo de relógio. Um registo de deslocamento com entrada e saída série possui apenas uma entrada (SERIN) e uma saída (SEROUT). Na entrada SERIN aplica-se um novo bit a deslocar até à saída da cadeia de F/Fs, um F/F em cada ciclo do relógio. O bit aplicado na entrada surge na saída SEROUT passados **n** ciclos do relógio e sai do registo no ciclo seguinte (**n+1**). Um registo de deslocamento com entrada e saída série, e com **n**-bits, permite atrasar um sinal em **n** ciclos do relógio.



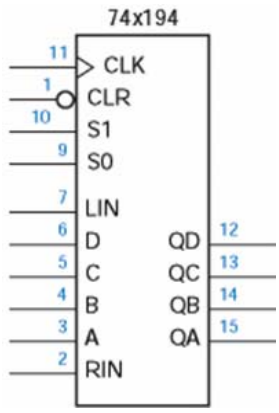
Um registo de deslocamento com entrada série e saída paralela possui uma saída para cada bit armazenado, colocando-os disponíveis em 1Q a NQ. Os registos deste tipo são utilizados para efectuar conversões de série para paralelo. Pode ainda construir-se um registo de deslocamento com entrada paralela e saída série ou com entrada e saída paralela. O registo de deslocamento com entrada e saída paralela é útil na implementação das operações de **SHIFT** e **ROTATE**.



O registo de deslocamento com 8-bits dos CIs 74x164 e 74x166 são unidireccionais dado que só efectuam o deslocamento dos bits numa direcção. O CI **74x164** é do tipo entrada série e saída paralela e possui uma entrada de *reset* CLR assíncrona. O CI **74x166** é do tipo entrada paralela e saída série, também com uma entrada de *reset* CLR assíncrona. O 74x166 regista a entrada quando SH/LD é **0** e desloca quando for **1**.



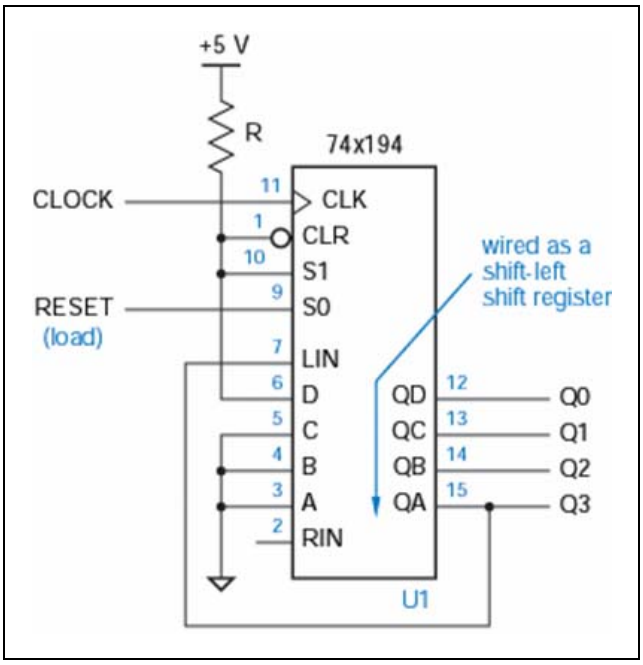
O CI MSI **74x194** é um registo de deslocamento com 4-bits e bidireccional, do tipo entrada e saída paralela. O registo de deslocamento é bidireccional porque desloca os bits armazenados em ambas as direcções (direita/esquerda), de acordo com as entradas de controlo S1 e S0. O CI 7x194 é considerado um registo de deslocamento universal, dado poder funcionar como qualquer um dos tipos de registo de deslocamento menos genérico apresentado.



Function	Inputs S1	Inputs S0	Next state QA^0	Next state QB^0	Next state QC^0	Next state QD^0
Hold	0	0	QA	QB	QC	QD
Shift right	0	1	RIN	QA	QB	QC
Shift left	1	0	QB	QC	QD	LIN
Load	1	1	A	B	C	D

7.4. Contadores baseados em registos de deslocamento

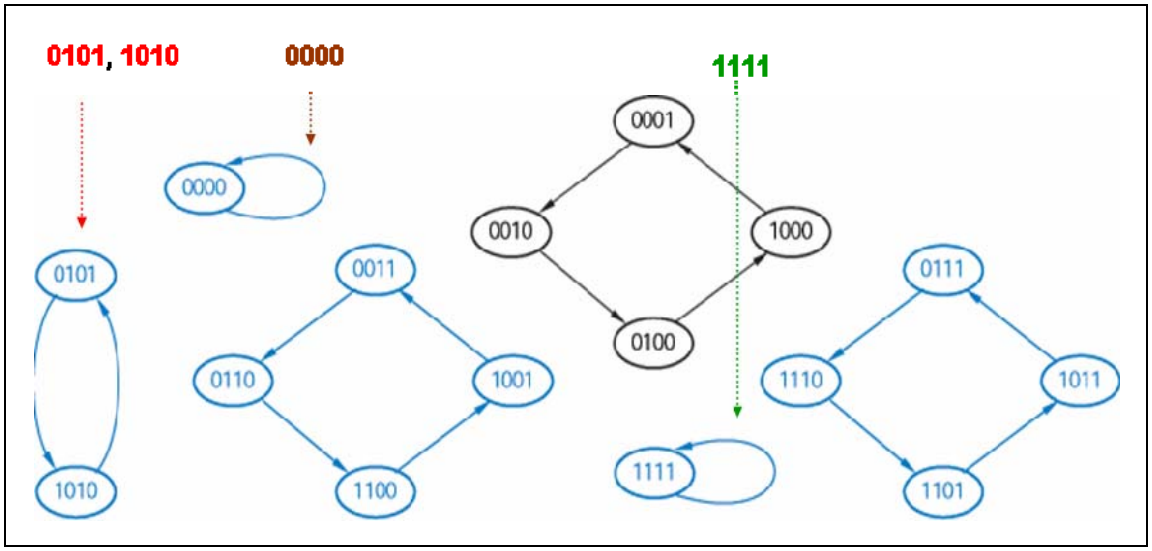
Pode combinar-se um registo de deslocamento com lógica combinacional para dar origem a uma máquina de estados com um diagrama de estados cíclico. Este tipo de circuito chama-se **contador baseado num registo de deslocamento** (*shift-register counter*).



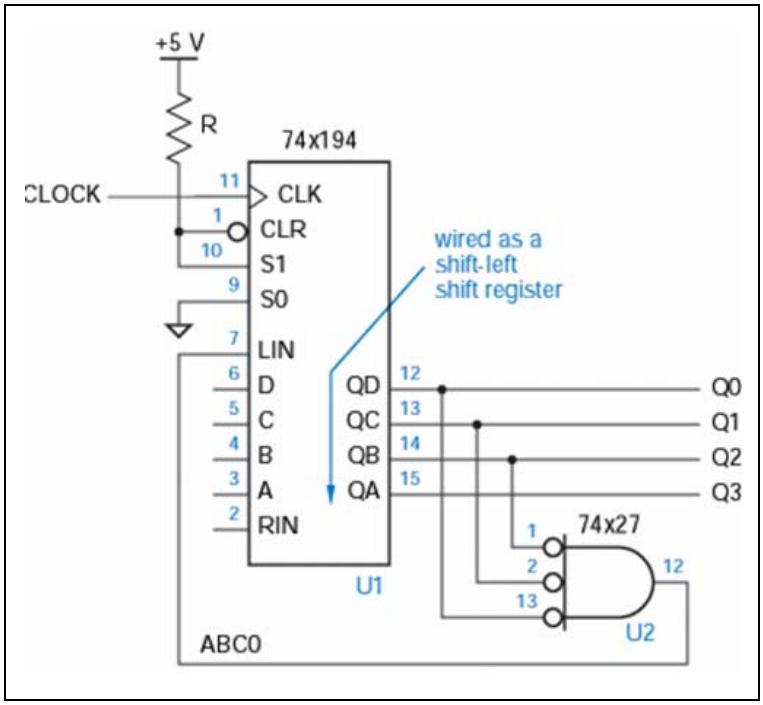
•Um contador baseado num registo de deslocamento é diferente dum contador binário porque não conta por ordem (ascendente ou descendente) e é utilizado em aplicações de controlo.
O contador baseado num registo de deslocamento mais simples de todos emprega um registo de deslocamento de **n**-bits para obter um contador com **n** estados, e dá pelo nome de **contador em anel**.

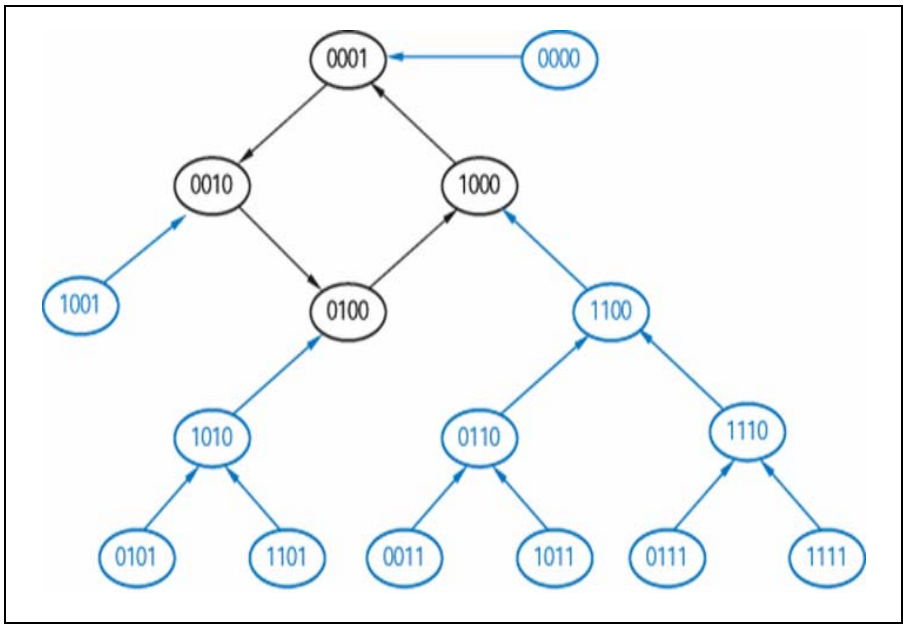
Pode obter-se um contador em anel de 4-bits a partir dum CI 74x194 (reg. de deslocamento). Quando o RESET está a 1, o contador em anel coloca

ABCD=0001 na saída Q3Q2Q1Q0 (estado inicial). No modo normal de funcionamento, com RESET=0, o contador efectua o deslocamento à esquerda do valor das saídas. O circuito não é robusto porque se o contador atingir um estado fora do ciclo normal de 4 estados (**0000**, **0101**, **1010**, **1111**, etc), vai permanecer fora do ciclo.



Pode projectar-se um contador em anel com mecanismo de auto-correcção de modo a que todos os estados fora do ciclo normal possuam transições para estados dentro desse ciclo. Um exemplo é o contador de 4-bits que se obtém com um CI **74x194** e uma porta NOR (ver figura seguinte). A porta NOR é usada para aplicar um **1** em LIN, apenas quando os 3 LSBs (Q2Q1Q0) forem **0**.

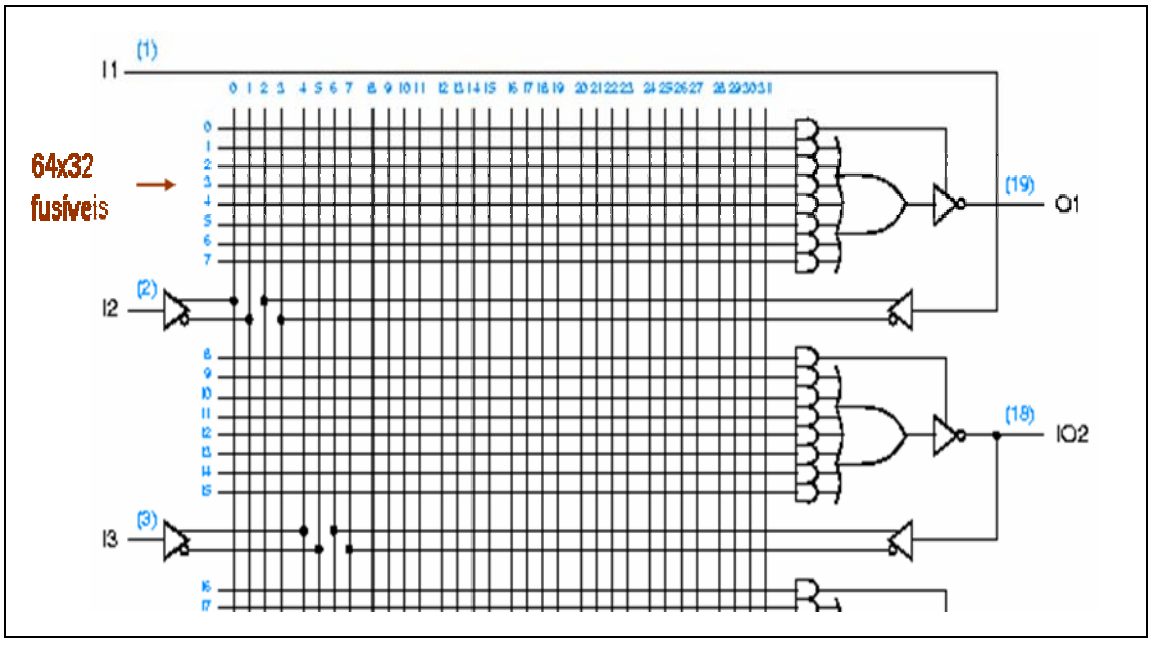




Deste modo, todos os **estados fora do ciclo normal** conduzem a um **estado dentro desse ciclo**. Independentemente do estado anormal atingido, a correcção para o estado normal **0001** faz-se num máximo de 4 ciclos de relógio, dispensando a utilização do RESET.

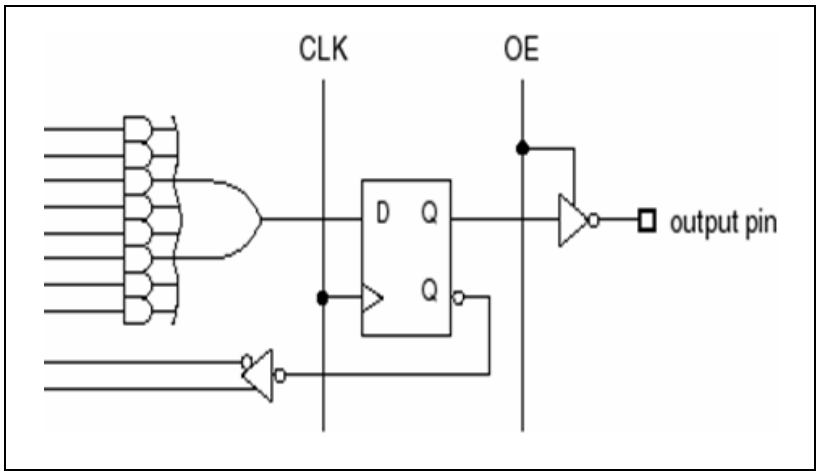
7.5. PLDs sequenciais

Vamos relembrar a **PAL combinacional 16L8** através duma vista parcial do seu diagrama lógico.



Esta PAL possui **10** entradas primárias (I1..I10), **0** saídas registadas, **2** saídas combinacionais (O1 e O8), **6** entradas/saídas (IO2..IO7), cada OR recebe **7** termos de produto e cada AND recebe **16** entradas.

A **PAL sequencial 16R8** possui um *flip-flop* tipo D entre a saída de cada OR do array AND-OR e uma saída da PAL. O array AND-OR da PAL 16R8 é igual ao da PAL 16L8. Todos os *flip-flops* partilham o mesmo sinal de relógio CLK. Cada *flip-flop* liga a um *buffer tri-state*, que por sua vez liga ao exterior da PAL através dum pino. Todos os *buffers* partilham o mesmo sinal de *output enable* OE.



As entradas do array AND-OR da PAL 16R8 são as entradas primárias I1 a I8 da PAL (na forma directa e complementada) mais as saídas dos 8 *flip-flops* (na forma directa e complementada). O facto de as saídas dos *flip-flops* alimentarem o array AND-OR facilita a implementação do bloco que gera as transições de estado nas máquinas de estados. As características da PAL 16R8 são: **8** entradas primárias, **8** saídas registadas, **0** saídas combinacionais, **0** entradas/saídas, cada OR recebe **8** termos de produto e cada AND recebe **16** entradas.
A variante **16R6** de PAL dispõe de **2** entradas/saídas (sendo as saídas combinacionais) e **6** saídas registadas.

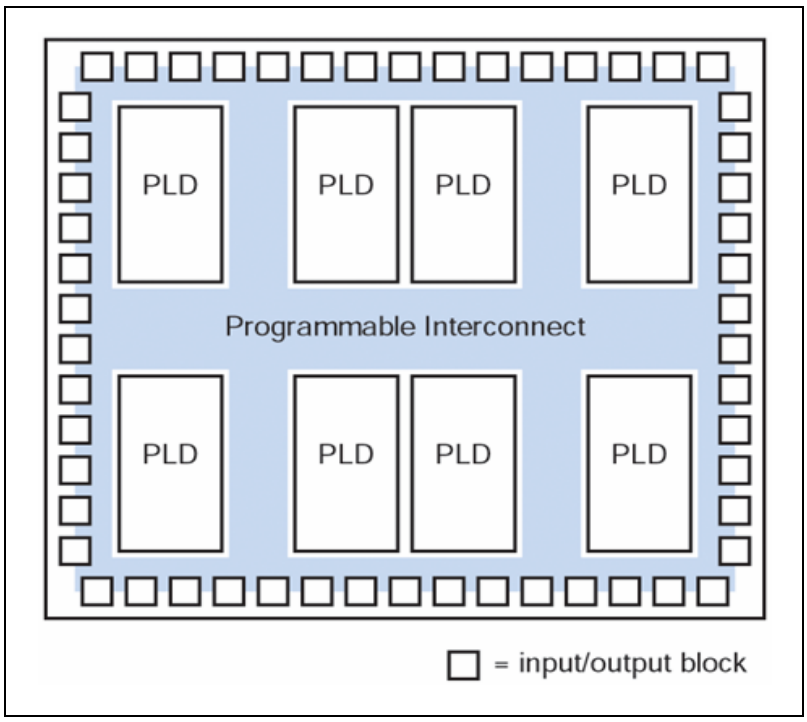
8. CPLDs e Memórias

8.1. PLDs complexas (CPLDs)

Introdução

À medida que a tecnologia de fabrico de CIs avançou, surgiu o interesse natural em produzir PLDs cada vez maiores para tirar partido duma densidade de transístores (dentro do *chip*) cada vez maior.

Uma **CPLD** é um dispositivo que incorpora num único *chip* uma colecção de PLDs interligadas por uma estrutura programável. Esta estrutura permite que as PLDs sejam interligadas do mesmo modo que o seriam fora do *chip*. A família de CPLDs 9500 da Xilinx vai ser usada como exemplo para a arquitectura duma CPLD.



Como se pode expandir a arquitectura das PLDs simples?

Hipótese de expansão:
Aumentar o número de entradas e saídas duma PLD convencional, ou seja, partindo das PALs 16V8, 20V8, 22V10 poderia chegar-se a PALs mais densas, tais como 32V16 ou 128V64.

Problemas desta alternativa:

- Aumentar ***n*** vezes o número de entradas e saídas exige n^2 vezes mais espaço no *chip,* logo é uma solução dispendiosa. Isto porque ao

aumentar **n** vezes um *array* de ANDs de tamanho (a*b) corresponderia um novo *array* de ANDs de tamanho (n*a)*(n*b)=n^2*(a*b).

- Quando se aumenta o número de entradas, a lógica combinacional fica cada vez mais lenta dado que o número de entradas do *array* de ANDs aumenta.

Solução: Utilizar várias PLDs interligadas por 1 estrutura programável relativamente reduzida.

•Esta arquitectura é menos genérica do que uma PLD de grande dimensão, mas a utilização duma ferramenta ***fitter*** liberta o projectista da tarefa de atribuir a cada bloco tipo PLD (da CPLD) uma parte do sistema a implementar.

CPLDs da Xilinx

Tendo por base o mesmo bloco PLD (designado por **Bloco Funcional** na Xilinx) pode construir-se variantes da mesma família de CPLD diferindo no:

- Número de blocos PLD;
- Número de pinos de entrada/saída (I/O).

Muitas CPLDs possuem menos pinos de I/O do que células base (**macrocélulas**):

- Algumas macrocélulas fornecem lógica interna mas não ligam as saídas ao exterior do *chip*.
- É possível obter encapsulamentos com um número de pinos diferente, mas com a mesma lógica interna.
- É possível obter CPLDs com o mesmo encapsulamento, mas com lógicas internas diferentes (ver tabela seguinte).

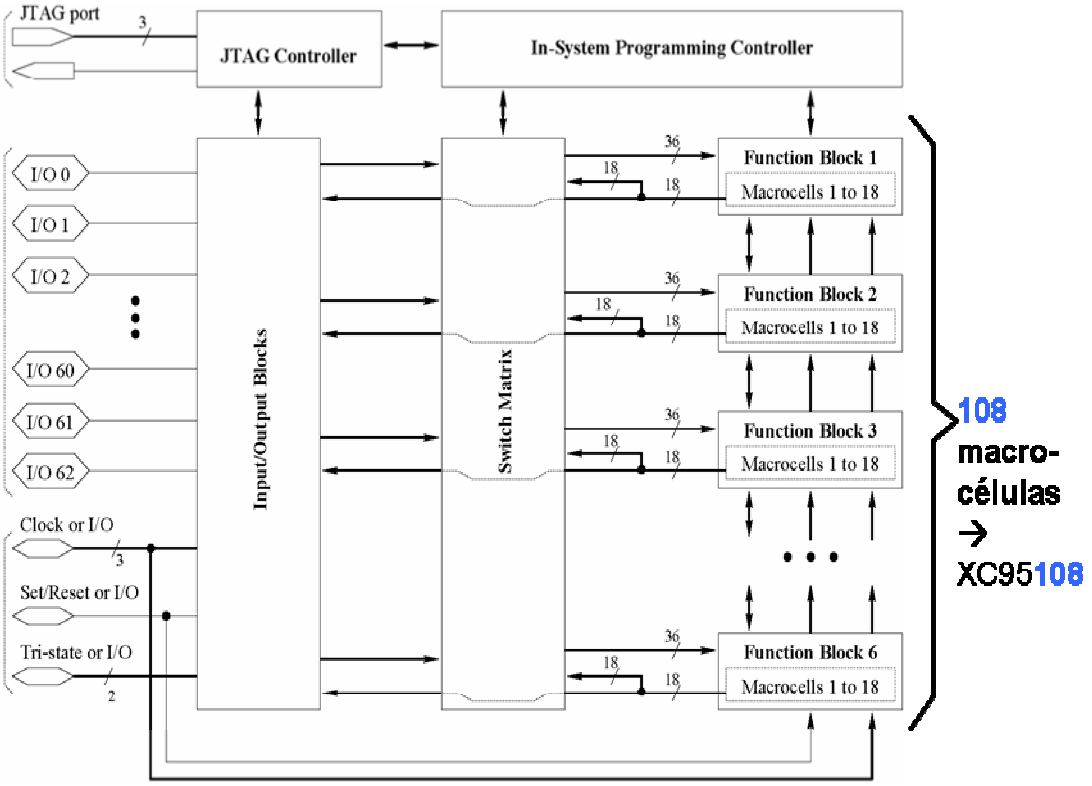
	Part Number					
	XC9536	XC9572	XC95108	XC95144	XC95216	XC95288
FBs / macrocells	2 / 36	4 / 72	6 / 108	8 / 144	12 / 216	16 / 288
Package	**Device I/O Pins**					
44-pin VQFP	34					
44-pin PLCC	34	34				
48-pin CSP	34					
84-pin PLCC		69	69			
100-pin TQFP		72	81	81		
100-pin PQFP		72	81	81		
160-pin PQFP			108	133	133	
208-pin HQFP					166	168
352-pin BGA					166	192

a mesma lógica interna e diferentes encapsulamentos

o mesmo encapsulamento e lógicas internas diferentes

A arquitectura das CPLDs XC95xx da Xilinx é composta por:

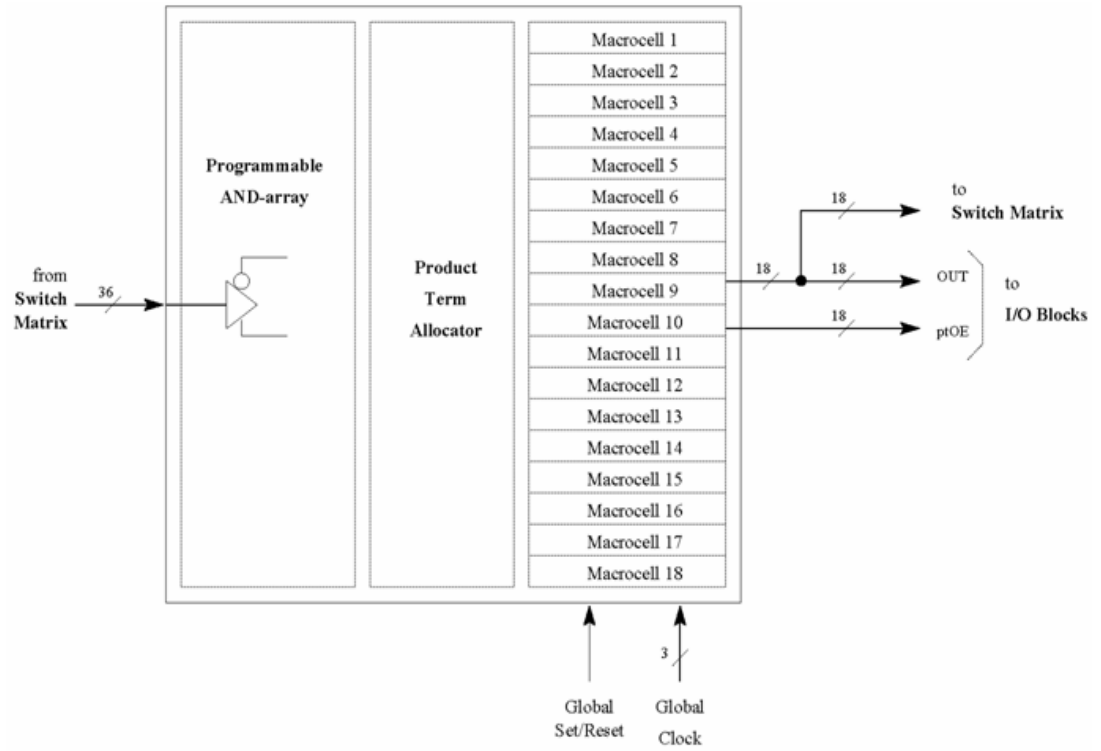
- Um conjunto de **blocos funcionais** e
- de **blocos de entrada/saída** (IOBs) interligados por
- uma **matriz de comutação** que encaminha as entradas e saídas para os blocos funcionais (ver figura).



Arquitectura das CPLDs da família 9500 da Xilinx.

Cada **bloco funcional** possui 18 macrocélulas, cada uma implementa uma função combinacional ou registada. As entradas de cada bloco são os sinais de relógio globais, um sinal de *set/reset* global, sinais de *enable* e até 36 entradas genéricas. Cada bloco gera até 18 saídas para a matriz de comutação e/ou IOBs. Com as 36 entradas e o complemento dessas 36 entradas pode gerar-se até 90 termos de produto: 18 macrocélulas * 5 termos por macrocélula = 90 termos.

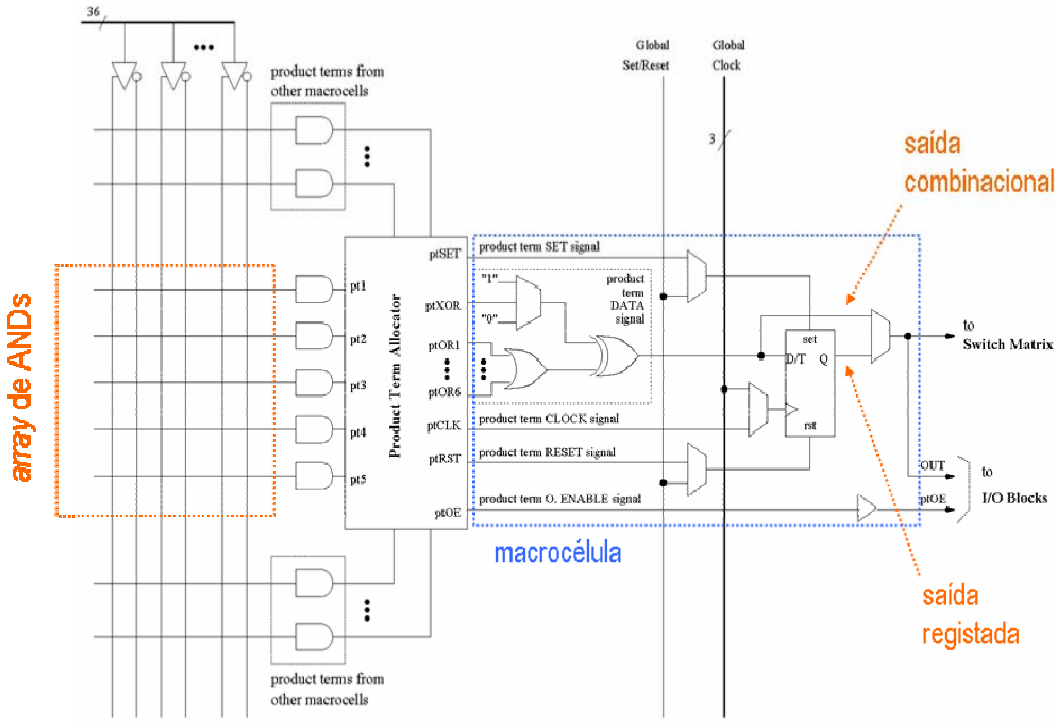
Existem caminhos de realimentação dentro do bloco funcional, permitindo que as saídas desse bloco possam ser usadas como entradas do *array* de ANDs do mesmo bloco.



Bloco funcional das CPLDs da família 9500 da Xilinx.

Cada **macrocélula** pode implementar uma função combinacional ou registada. Os contributos para os termos de produto são gerados pelo alocador de termos de produto a partir de:

- 5 entradas ***pt1*** a ***pt5*** provenientes directamente do *array* de ANDs.
- alguns termos de produto provenientes de outras macrocélulas: 1 da célula acima e 1 da célula abaixo, no exemplo a apresentar.



Macrocélula das CPLDs da família 9500 da Xilinx.

O alocador de termos de produto gera contributos para:

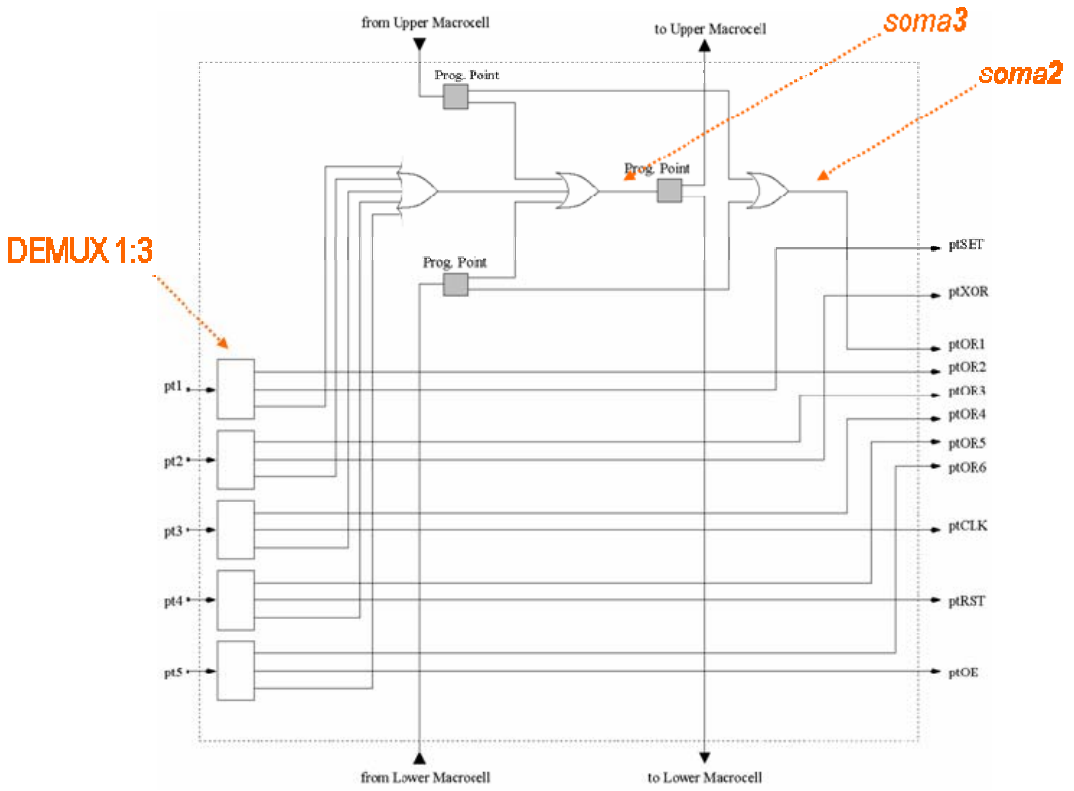
- A saída duma função combinacional.
- A entrada de dados / o sinal de relógio / os sinais de *set* e de *reset* duma função registada.
- Uma saída de *enable.*

O **alocador de termos de produto** controla o modo como os termos de produto são utilizados na implementação de cada função lógica. Uma função lógica pode envolver todos os 90 termos de produto, mas se usar apenas 15 termos (5 da macrocélula e 5/5 da macrocélula acima/abaixo) o atraso é mínimo. O alocador de termos de produto pode gerar somas de produto parciais a usar nessa ou em outras macrocélulas.

Por exemplo, o alocador de termos de produto gera 2 sinais:

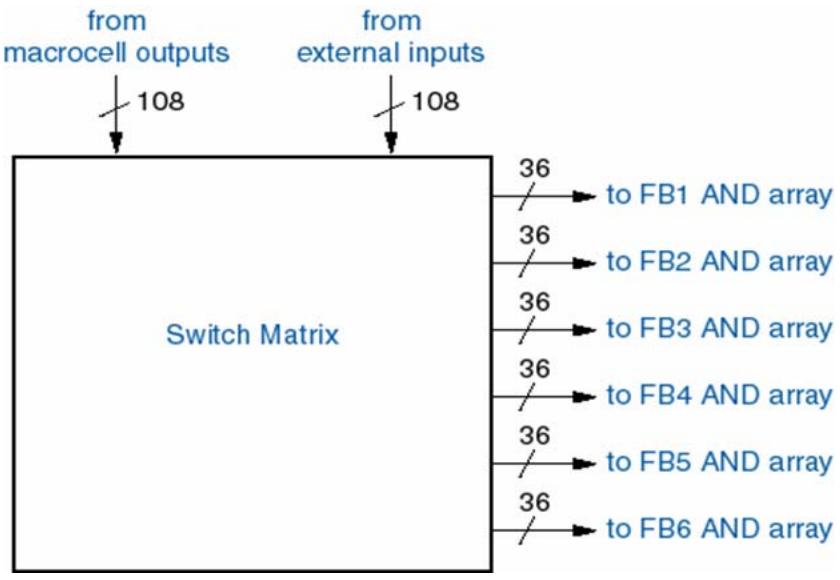
- O primeiro sinal é uma soma de produtos parcial, envolvendo 3 termos de produto, a enviar para as macrocélulas vizinhas (sinal ***soma3*** na figura em baixo).

- O segundo sinal é uma soma de 2 termos de produto, provenientes de macrocélulas vizinhas, que implementa a função lógica da própria macrocélula (sinal ***soma2*** na figura em baixo).



Alocador de produtos das CPLDs da família 9500 da Xilinx.

A **matriz de comutação** disponibiliza caminhos programáveis entre as entradas e as saídas. As entradas são as saídas dos *buffers* de entrada dos IOBs e as 18 saídas de cada FB, enquanto as saídas ligam às 36 entradas de cada FB.



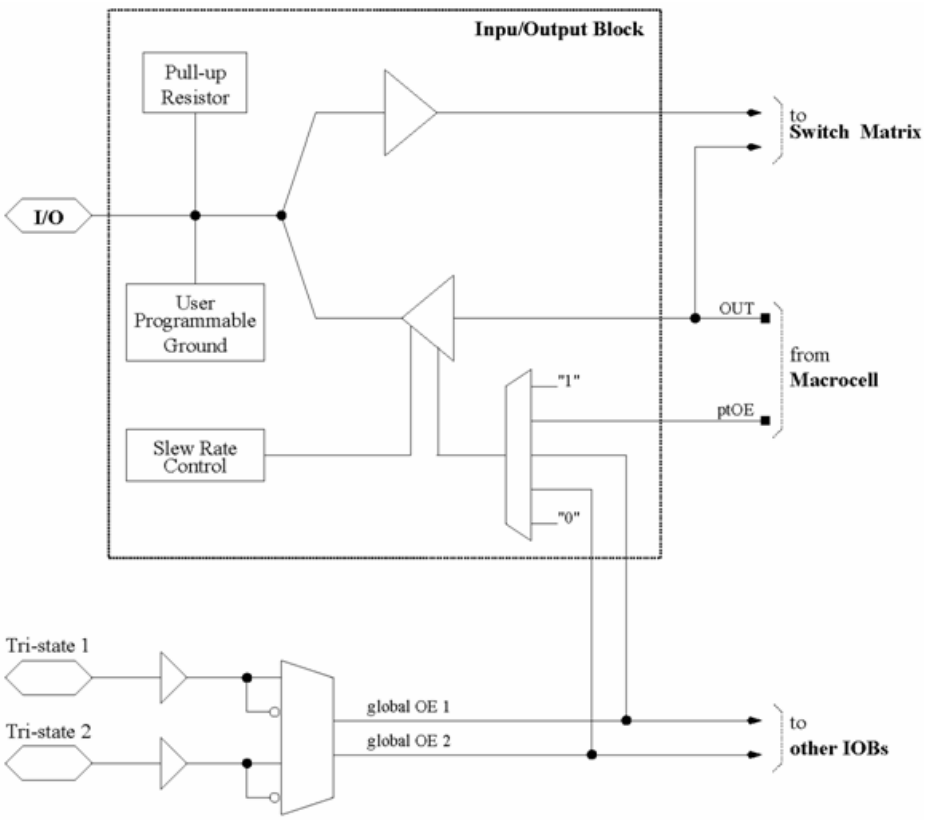
Matriz de comutação das CPLDs da família 9500 da Xilinx.

O **bloco de entrada/saída** (IOB) funciona como interface entre a lógica interna e os pinos. Cada IOB contém:

- Um *buffer* de entrada.
- Um *buffer tri-state* para saída.
- Lógica para gerar o sinal que controla o *buffer* de saída.

Este sinal pode ser gerado de várias formas:

- É o sinal de *output enable* gerado internamente pelo alocador de termos de produto.
- É um de entre os dois sinais de *output enable* globais/externos (***OE1*** ou ***OE2***).
- É um sinal fixo a '0' (*buffer tri-state sempre desligado => pino só de entrada*).
- É um sinal fixo a '1' (*buffer tri-state sempre ligado => pino só de* saída).



Bloco de entrada/saída das CPLDs da família 9500 da Xilinx.

8.2. Memórias: ROMs e RAMs

Qualquer circuito sequencial possui pelo menos um tipo de memória, uma vez que cada *flip-flop* e *latch* guarda um bit de informação. Contudo, o termo **memória** é usado para identificar um dispositivo em que os bits estão armazenados de forma estruturada, normalmente em forma de *array* bidimensional, em que se acede a uma linha de bits de cada vez.

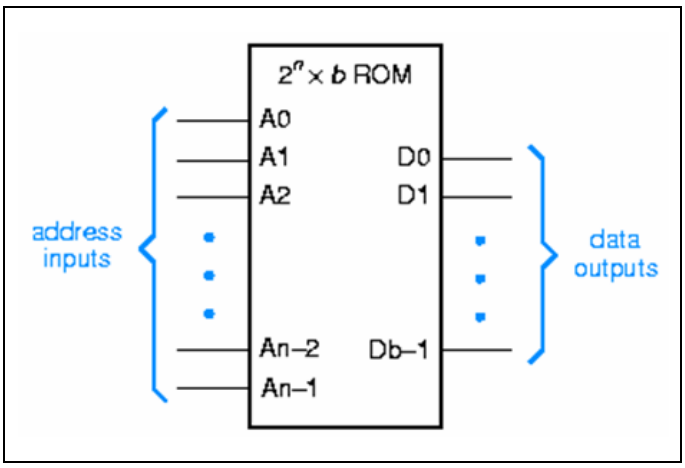
O campo de aplicação da memória é vasto e variado:

- Em microprocessadores (uPs);
- Em sistemas baseados em uPs ou microcontroladores, para guardar os dados a processar e/ou as instruções a executar;
- Em dispositivos de armazenamento portáteis: cartões SD/CF, *flash drivers*, leitores MP3
- Em sistemas áudio/vídeo (como leitores/gravadores de CD/DVD) para guardar uma parte da informação a processar e assim melhorar o desempenho.

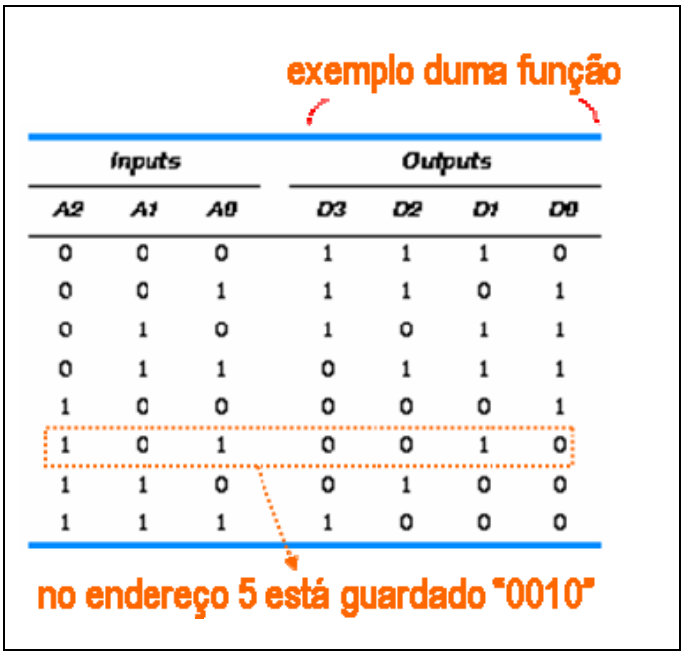
No CPU dum microprocessador, a **ROM** pode ser usada para guardar informação que define os procedimentos básicos a executar no arranque do sistema. A ***cache*** dos microprocessadores também é uma memória com menos capacidade mas maior desempenho. A **memória principal** é uma memória de maior capacidade mas menor desempenho.

ROMs

Uma memória só de leitura *(ROM - Read-Only Memory)* é um circuito combinacional com **n** entradas e **b** saídas (ver figura). As entradas definem o endereço de entrada e as saídas os dados de saída. Uma ROM pode ser vista como um dispositivo que guarda a tabela de verdade duma função lógica combinacional com **n** entradas e **b** saídas.



A tabela de verdade duma função lógica combinacional com **3** entradas e **4** saídas pode ser guardada numa ROM de tamanho 2^3x4. Os dados de saída da ROM coincidem com o valor das saídas na linha *(da tabela de verdade)* seleccionada pelos bits de endereço.

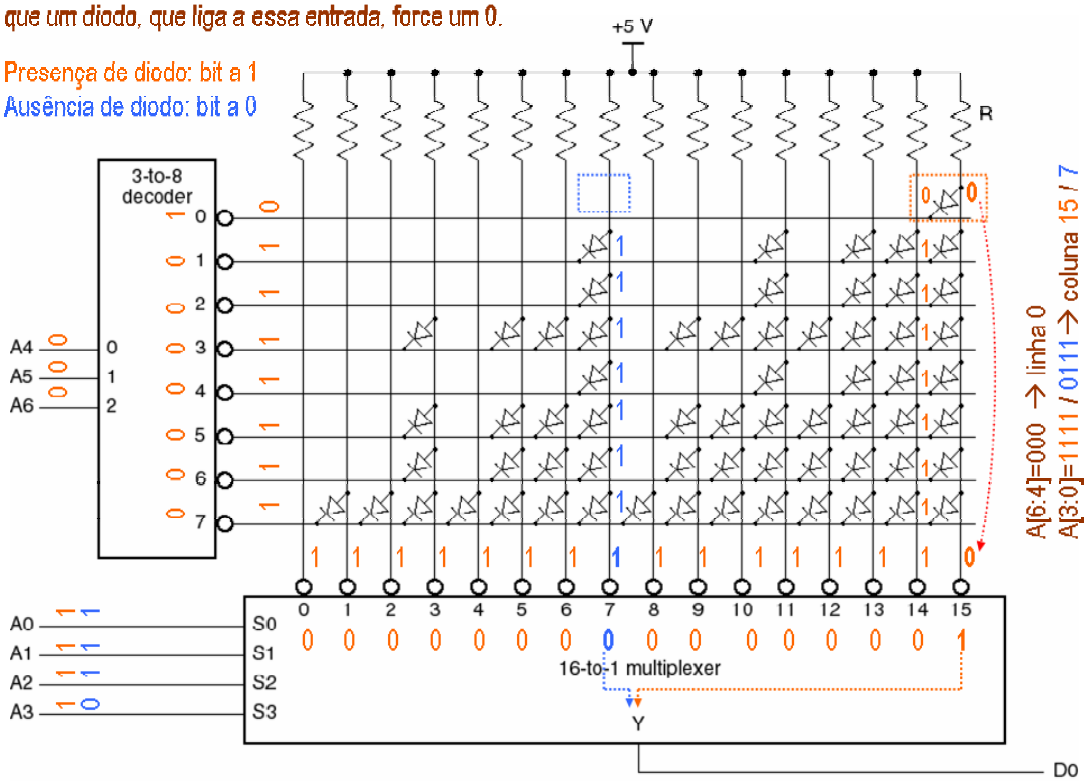


Como uma ROM é um circuito combinacional, não é verdadeiramente uma memória. Mas pode pensar-se na ROM como um dispositivo que guarda a informação definida no momento em que ela foi fabricada. A ROM é uma memória não-volátil, dado que mantém o conteúdo mesmo quando se desliga a alimentação.

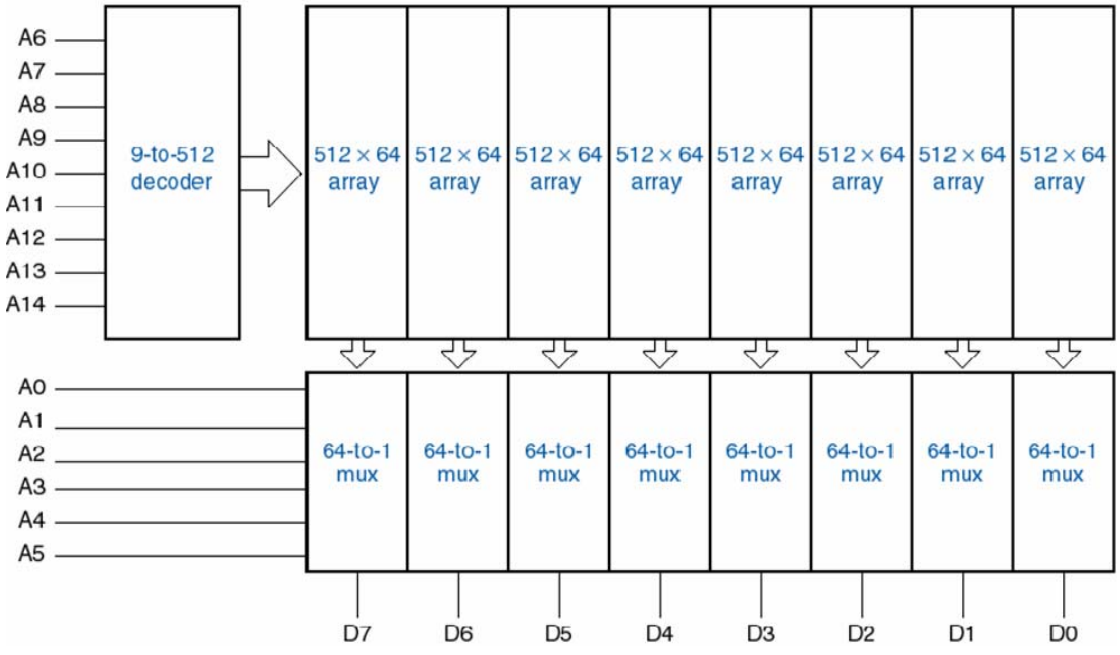
Estrutura interna duma ROM:

- Depende da tecnologia usada no seu fabrico.
- Mas é habitual incluir/não-incluir um díodo ou transístor para "programar" um 1/0 numa dada posição da matriz da ROM.
- A próxima figura ilustra a estrutura duma ROM 128x1, em que a descodificação do endereço se processa em 2 dimensões para reduzir o tamanho do descodificador.
- A ROM 128x1 está organizada segundo uma matriz bidimensional de 8 linhas x 16 colunas.
- O descodificador usa os **3** bits MS do endereço para seleccionar uma linha.
- O MUX 16:1 usa os **4** bits LS do endereço para seleccionar uma coluna.

Para que uma entrada do MUX seja 0 só é necessário
que um diodo, que liga a essa entrada, force um 0.



Uma ROM de tamanho 32K com múltiplas saídas (D7:D0) possui 2^{15}x8 bits.
Usa 1 descodificador de 9:512, 8 matrizes de 512 linhas x 64 colunas e 8 MUX 64:1.



Actualmente, as ROM são fabricadas num único CI e não com componentes discretos. Uma ROM com capacidade de alguns Mbits custa menos de 5€.

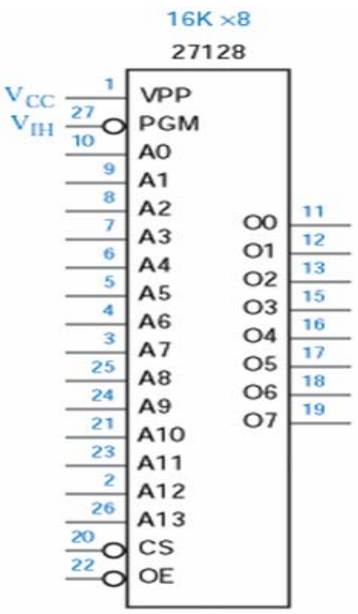
O conteúdo duma ROM pode ser "programado" usando um de vários métodos. Para programar o conteúdo duma *mask-programmable ROM*, fornece-se ao fabricante uma lista (ficheiro) que define o padrão de ligações e de não-ligações a efectuar na(s) matriz(es) da ROM. Este processo de fabrico é dispendioso e só se usa para produções em grande escala.

O processo de programação duma ROM programável (PROM) é idêntico ao da *mask ROM*, excepto que o conteúdo da PROM pode ser gravado usando um programador de PROMs. Uma PROM é fabricada com todos os bits no mesmo valor, normalmente a 1. O programador permite mudar o valor dos bits para 0, nas posições requeridas.

Uma *erasable programmable ROM* (EPROM) é programável tal como uma PROM, mas o seu conteúdo pode ser apagado e colocado no estado de tudo a 1, usando uma luz ultravioleta. Provavelmente a aplicação mais frequente das EPROMs é para guardar o código a executar pelo microprocessador ou microcontrolador dos sistemas embebidos. É comum utilizar EPROMs durante o desenvolvimento do código a usar nos sistemas embebidos, dado que esse código vai ser alterado repetidamente durante a fase de depuração. Como as ROMs e as PROMs são mais baratas do que as EPROMs similares, quando o desenvolvimento do código estiver concluído, substitui-se a EPROM por uma ROM ou PROM para que a produção do sistema seja menos dispendiosa.

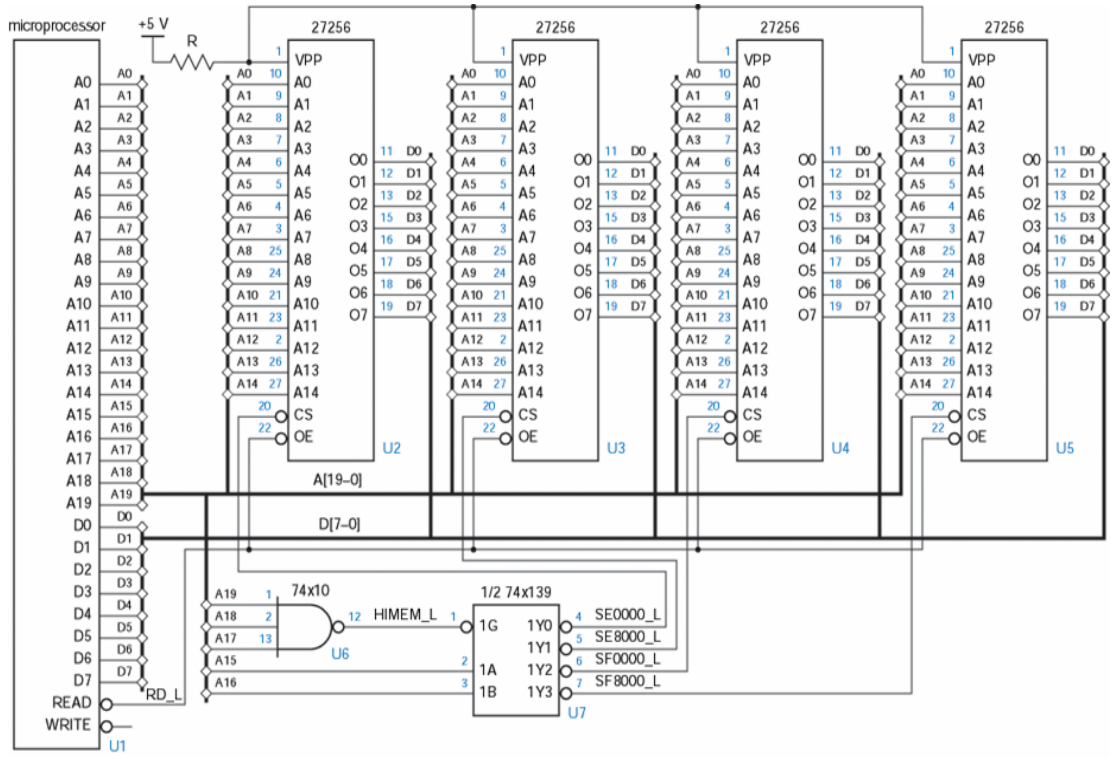
Uma *electrically erasable programmable ROM* (EEPROM) é idêntica a uma EPROM, excepto que o conteúdo da EEPROM pode ser apagado electricamente (pela aplicação duma tensão). As EEPROMs não são alternativa às RAMs porque o tempo de escrita é muito superior e não estão vocacionadas para serem reescritas indefinidamente (apenas milhares de vezes).

Muitas vezes as saídas da ROM (**O7:O0**) ligam a um barramento comum, ao qual ligam vários dispositivos com o objectivo de nele escrever, ainda que em instantes distintos (ver próxima figura). Assim, muitas ROMs possuem saídas em *tri-state* e uma entrada **OE** (*Output Enable*) que deve ser activada para que as saídas apareçam no exterior do *chip*. Para facilitar o desenho de certos circuitos em que há múltiplas ROMs ligadas ao mesmo barramento, embora apenas uma tenha a saída activa em cada instante, as ROMs possuem uma entrada **CS** (*Chip Select*). Neste caso, para que as saídas em *tri-state* estejam disponíveis no exterior é necessário que as entradas OE e CS estejam ambas activas.



Uma ROM de 16K x 8 bits.

Descodificação de endereços para gerar os 4 sinais **CS** usados na leitura de 4 ROMs 32Kx8, o mesmo é dizer 128Kx8:



HIMEM_L=0 se A19:A17=111, ou seja, as ROMs estão nos 128K endereços + significativos.
SE0000_L=0 se A16:A15=00, ou seja, selecciona a ROM U2.

Algumas das **vantagens em usar ROMs** no projecto de circuitos:

- O projecto dos circuitos é rápido e simples.
- O circuito resultante é normalmente mais rápido do que um circuito com vários CIs SSI/MSI/PLDs.
- A funcionalidade implementada pela ROM pode ser alterada facilmente, mudando apenas o conteúdo nela guardado, sem ter que se alterar qualquer lógica no seu exterior.
- O preço das ROMs, por ser um dispositivo estruturado, diminui constantemente.
- A densidade das ROMs aumenta constantemente, alargando o tipo de problemas que se pode resolver com um único *chip*.

Algumas das **desvantagens em usar ROMs** no projecto de circuitos:

- Em circuitos simples ou medianamente complexos, uma solução baseada em ROMs pode ser mais cara, consumir mais ou ser mais lenta do que um circuito com vários CIs SSI/MSI/PLDs.
- Para implementar funções com mais de 20 entradas, uma solução baseada em ROMs não é exequível devido à limitação imposta pelo tamanho das ROMs disponíveis. Por exemplo, implementar um somador de 16 bits com ROMs exigiria muitos milhões de bits (2^{32}x16).

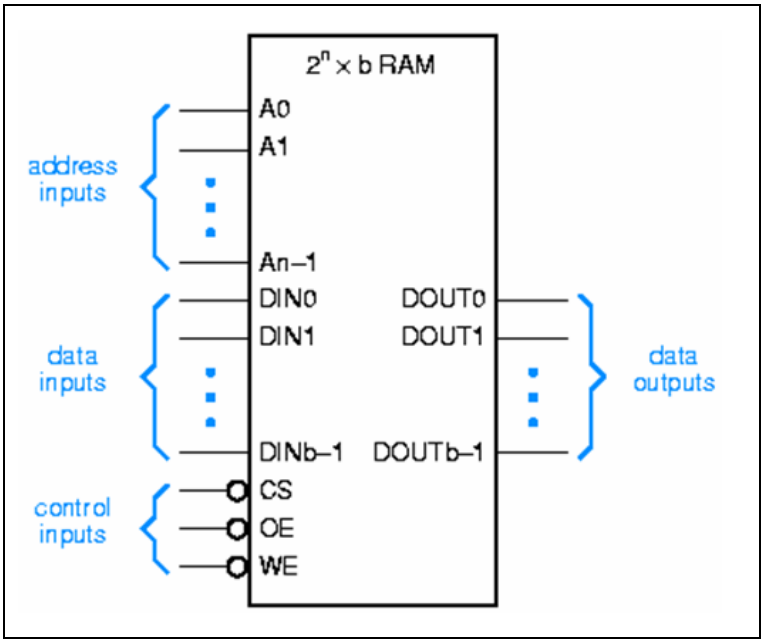
RAMs

A designação memória de leitura/escrita (***RWM*** - ***R****ead/****W****rite* ***M****emory*) aplica-se a *arrays* de memória que permitem que a informação seja guardada e lida em qualquer momento. Actualmente, a maior parte das memórias RWM é do tipo RAM.

Numa memória de acesso aleatório (***RAM - R****andom-****A****ccess* ***M****emory*), o tempo que demora a ler ou escrever um bit não depende da sua localização. Segundo a definição anterior, as ROMs também são memórias de acesso aleatório, mas a designação "RAM" geralmente só se aplica a RAMs que suportam leitura e escrita. Numa memória RAM estática (***SRAM***), o conteúdo escrito numa determinada posição mantém-se enquanto o *chip* estiver alimentado, a não ser que se escreva outro valor nessa posição. Uma memória RAM dinâmica (***DRAM***), só mantém o conteúdo guardado em cada posição se ele for reposto *(refreshed)* periodicamente. O refrescamento consegue-se lendo o conteúdo de determinada posição e reescrevendo-o na mesma posição.

A maior parte das RAMs são memórias do tipo volátil. Ou seja, quando se desliga a alimentação perde-se o seu conteúdo. Contudo, algumas RAMs mantêm o conteúdo mesmo quando se desliga a alimentação. Neste caso, são memórias do tipo não-volátil.

Uma RAM de **2^nxb** bits possui como entradas um endereço de **n** bits, **b** bits de dados e (3) sinais de controlo e como saídas apenas **b** bits de dados:



Os sinais de controlo são semelhantes aos da ROM apresentada, adicionando-se o sinal ***WE*** que habilita a escrita na RAM: CS, OE e WE. Quando se activa as entradas WE e CS, o valor da entrada de dados é escrito na posição da RAM seleccionada pelo endereço.

Cada posição, ou célula, de memória numa RAM estática comporta-se como uma *latch* D, e não como um *flip-flop* D sensível às transições do relógio. Ou seja, sempre que WE e CS estiverem activos, a *latch* associada com a posição seleccionada está aberta: a saída segue a entrada de dados. O valor que fica guardado na posição em causa é aquele que estiver presente na saída de dados da *latch* quando ela fecha.

Normalmente, as RAMs estáticas permitem apenas 2 tipos de acesso: leitura e escrita.

Para efectuar uma leitura:

- Coloca-se um endereço na entrada de endereço, mantendo as entradas de controlo **CS** e **OE** activas.
- O valor guardado na *latch* associada com a posição seleccionada é colocado na saída de dados (**DOUT** na RAM ilustrada atrás).

Para efectuar uma escrita:

- Coloca-se um endereço na entrada de endereço e um valor (a escrever) na entrada de dados **DIN**.
- Após isso, activam-se as entradas de controlo **CS** e **WE**.

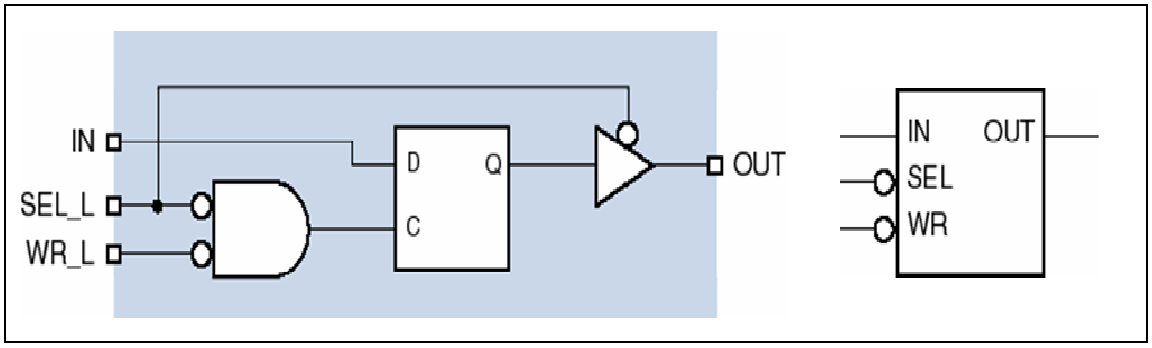
- A *latch* associada com a posição seleccionada abre e o valor presente em **DIN** é guardado na saída dessa *latch*.

A funcionalidade de cada célula da memória SRAM é a mesma que o circuito em baixo:

- O comportamento é o de uma *latch* D e ***não*** o de um *flip-flop* D sensível às transições.
- Quando **SEL_L** estiver activo, o valor guardado na *latch* é colocado na saída **OUT**.
- Quando **SEL_L** e **WR_L** estiverem ambos activos, a *latch* abre e o valor presente em **IN** é guardado na *latch*.

Numa operação de escrita é preciso garantir que:

- O endereço está estável antes da ordem de escrita.
- Os dados a escrever ficam estáveis antes de terminar a operação de escrita.



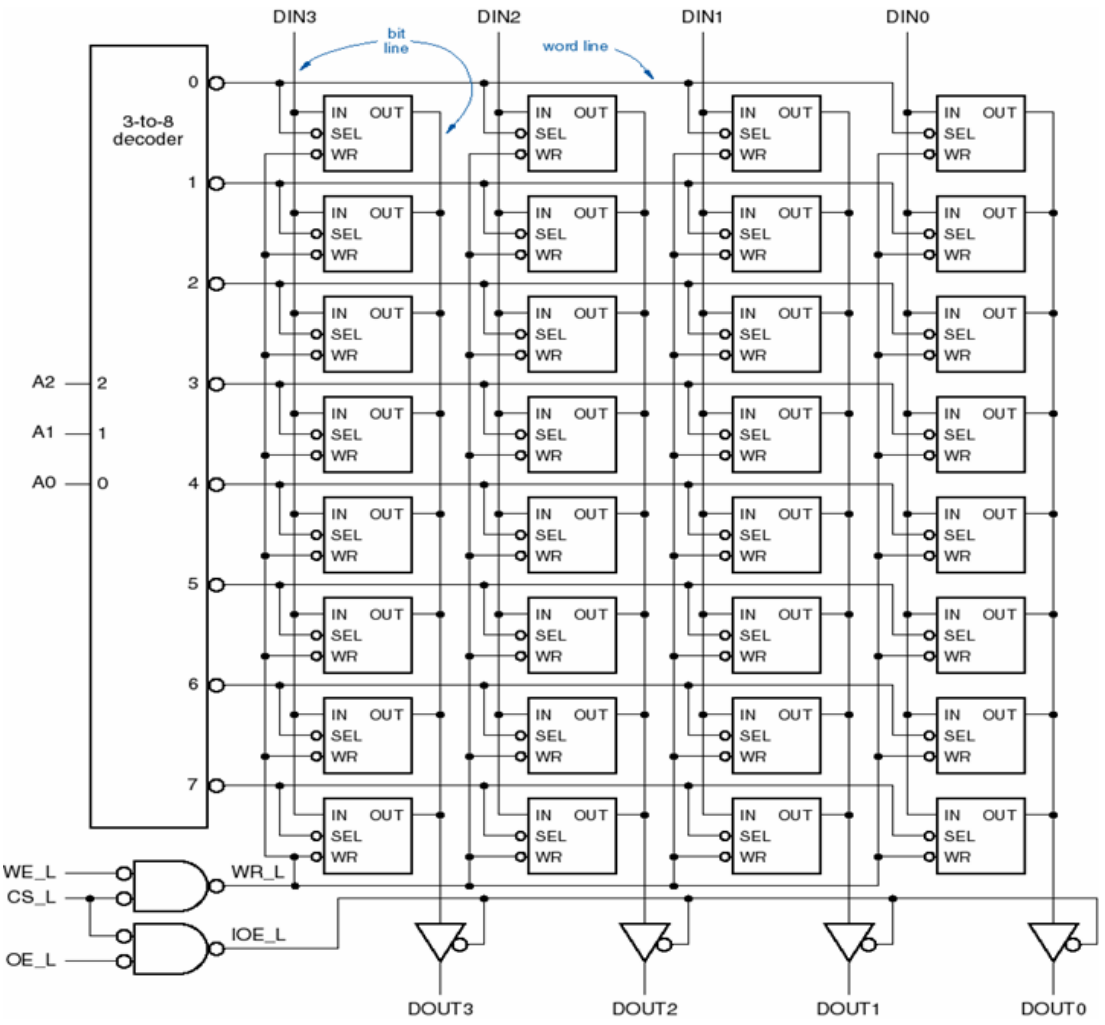
Organização interna duma SRAM:

Combinando várias células de memória como a anterior e juntando alguma lógica de controlo, obtém-se uma SRAM completa, por exemplo com tamanho 8x4 (ver figura a seguir).

Sinais de controlo:

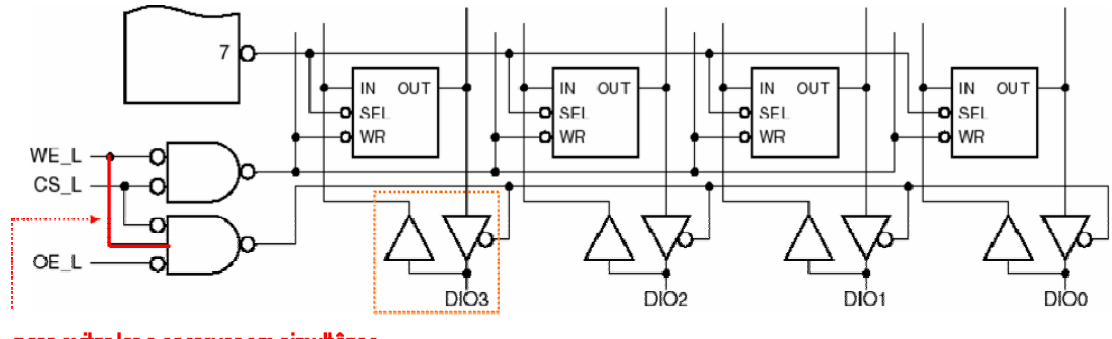
- *Chip select* (**CS_L**)
- *Output enable* (**OE_L**)
- *Write enable* (**WE_L**)

O descodificador 3:8 activa um (dos 8 sinais) **SEL** de modo a habilitar a leitura/escrita de/em todas as células da linha correspondente ao endereço A2:A0 aplicado.



Organização interna duma SRAM.

Para que a entrada e saída dos dados da SRAM se faça pelos mesmos pinos (bidireccionais) a organização interna apenas tem que ser alterada ligeiramente:

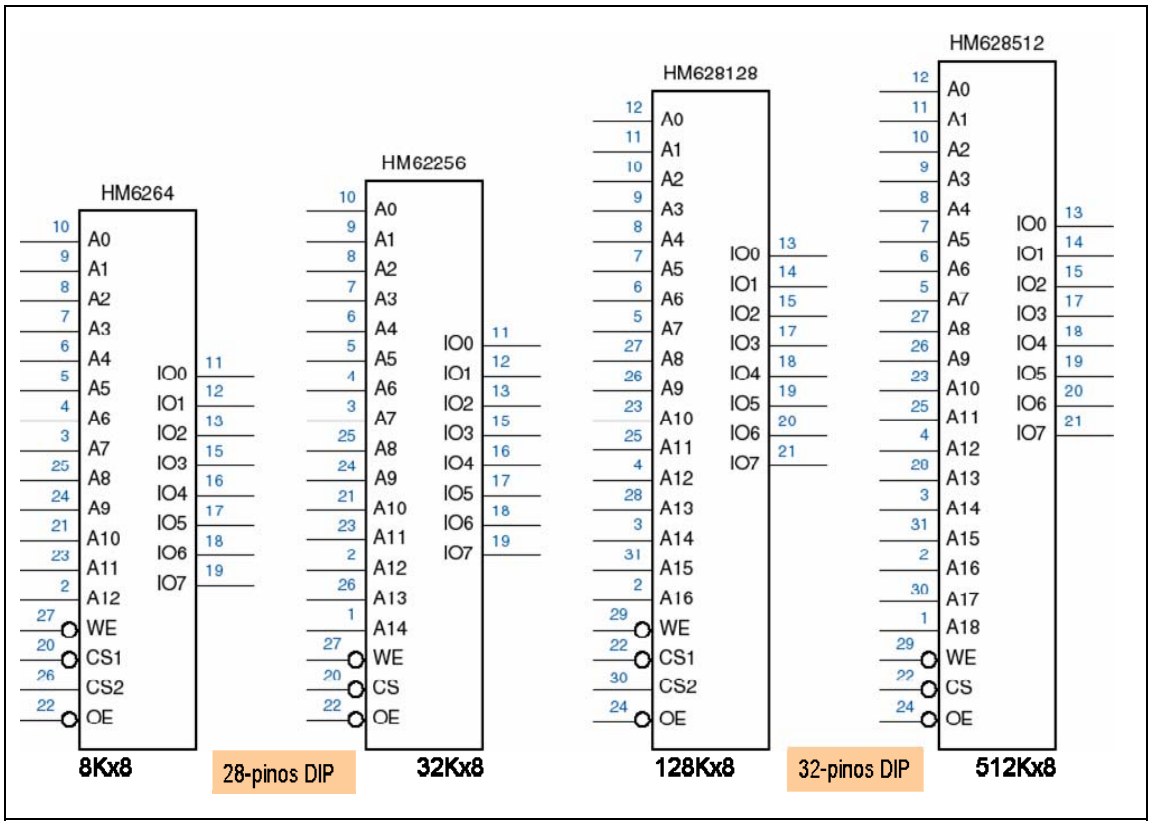


para evitar ler e escrever em simultâneo

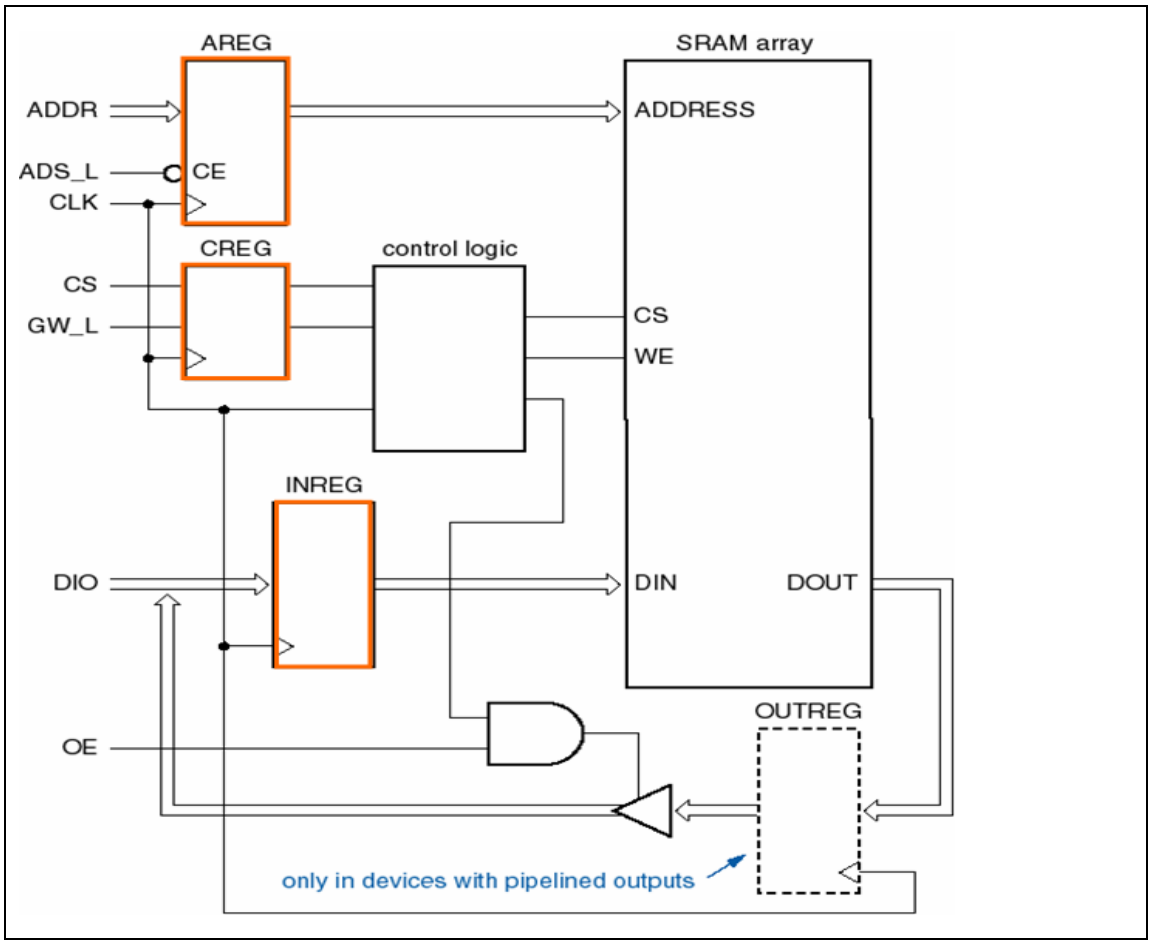
Qual o interesse em usar os mesmos pinos para os dados a ler e a escrever?

- Para reduzir o número de pinos, útil em memórias de grande capacidade.
- Porque a maior aplicação das memórias é na ligação a um microprocessador através de barramentos, sendo o barramento de dados normalmente bidireccional.

Exemplos de CIs SRAM, com encapsulamento e interface idênticos aos das ROMs:

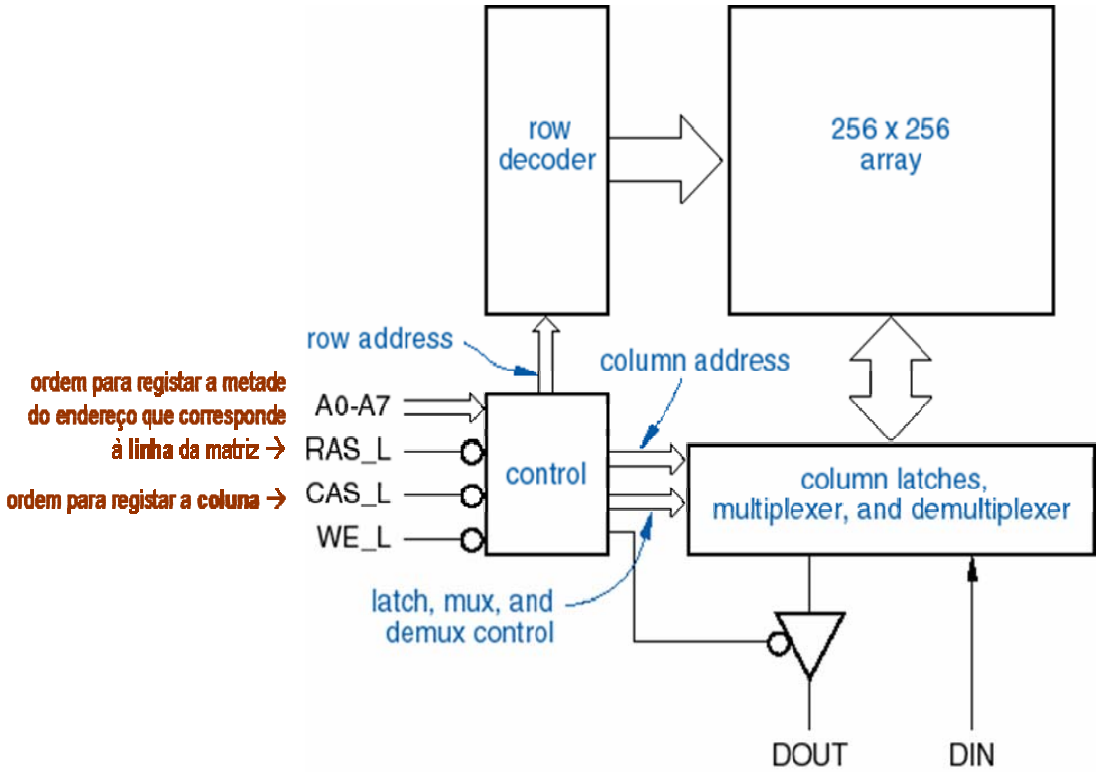


As memórias do tipo **SRAM síncrona** continuam a ter células baseadas numa *latch* mas usam endereços, sinais de controlo e dados registados. Na SRAM a seguir, o endereço **ADDR** é registado em **AREG**, os sinais de controlo **CS/GW_L** são registados em **CREG** e os dados de entrada **DIO** registados em **INREG**. Registar o endereço e os sinais de controlo facilita a utilização da SRAM em sistemas síncronos que operam com frequências elevadas.



A implementação da *latch* D usada nas células da SRAM emprega 4 portas lógicas discretas ou 4 a 6 transístores. A memória **RAM dinâmica** (***DRAM***) surge como forma de diminuir a quantidade de lógica usada por cada célula e conseguir assim uma maior capacidade com o mesmo espaço. A célula base da memória DRAM guarda a informação num condensador, que é acedido através dum único transístor. Ao fim de alguns milissegundos a informação guardada no condensador precisa ser reposta para que um "1" não passe a "0". Para repor (refrescar) a informação guardada, lê-se a célula e de seguida ela é reescrita com a mesma informação. Para reduzir o tempo usado no refrescamento, o processo refresca uma linha da matriz de cada vez. As DRAMs possuem normalmente várias matrizes para explorar o paralelismo de funcionamento, aumentando o débito de informação a escrever/ler.

Mostra-se agora uma DRAM de 64Kx1 bits organizada segundo uma matriz de 256x256 células. O endereço de 16 bits é aplicado em **A7:A0** em 2 etapas**,** sendo as metades registadas no bordo descendente de **RAS_L** e **CAS_L**.



DRAM de 64Kx1 bits organizada segundo uma matriz de 256x256 células.

9. Bibliografia

- John F. Wakerly, ***Digital Design Principles and Practices*****,** Prentice-Hall International, 3ª edição actualizada, 2001.

- M. Morris Mano, ***Digital Design***, Prentice-Hall International, 1997.

- Randy H. Katz, ***Contemporary Logic Design***, The Benjamin/Cummings Publishing Company, 1993.

- Daniel D. Gajski, ***Principles of Digital Design***, Prentice-Hall, 1997.

- Douglas L. Perry, ***VHDL: Programming by Example***, 4th edition, McGraw-Hill, 2002, ISBN 0-07-140070-2 .