

Arquitectura de Computadoras 1

Alumno: Santiago Vietto

Docente: Luis Eduardo Toledo

Institución: UCC

Año: 2023

Álgebra de Boole

Introducción

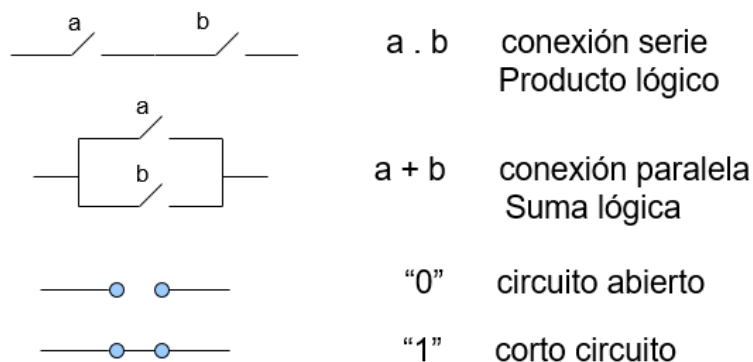
Definición

_ Se conoce con ese nombre en honor a George Boole, matemático inglés quien en 1854 publicó un libro donde se presentaba la teoría matemática de la lógica. Tenemos que saber que el algebra de Boole se utiliza para describir la interconexión de compuertas digitales y para transformar diagramas de circuitos en expresiones algebraicas.

_ Un álgebra de Boole es toda clase o conjunto de elementos que pueden tomar dos valores perfectamente diferenciados, que se suelen asignar a los números 0 y 1 de un código binario. Dichos elementos están relacionados mediante las operaciones binarias denominadas:

- Suma lógica (+)
- Producto lógico (.)
- Complementación o inversión (/), se coloca arriba de la letra.

_ A continuación, vemos unos ejemplos de operaciones binarias, pero con llaves, que representan las mismas:



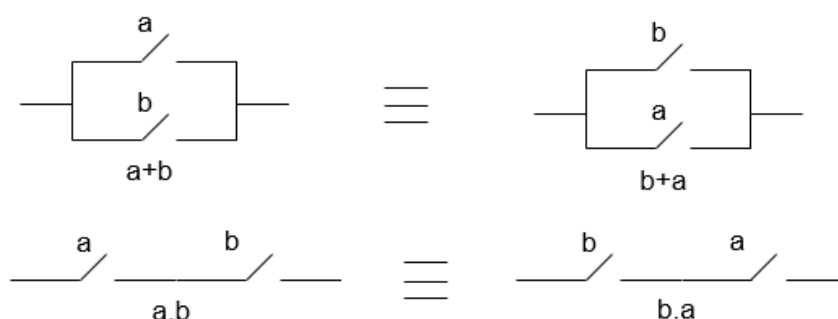
_ El algebra de Boole debe cumplir con los siguientes postulados.

Postulados

A)_ Ambas operaciones son **conmutativas**, es decir, si a y b son elementos del álgebra se verifica:

$$a+b = b+a$$

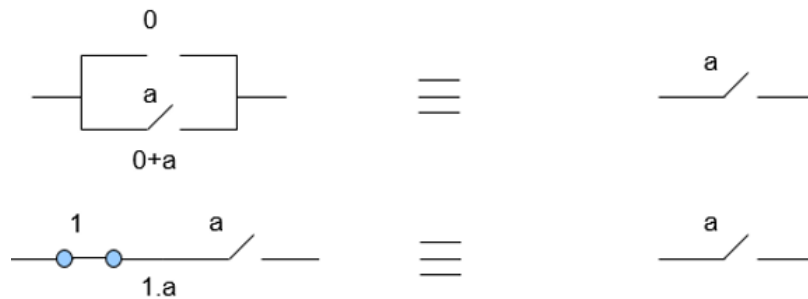
$$a.b = b.a$$



B)_ Posee dos **elementos neutros**, el 0 y el 1, que cumplen la propiedad de identidad con respecto a cada una de las operaciones suma lógica y producto lógico:

$$0+a = a$$

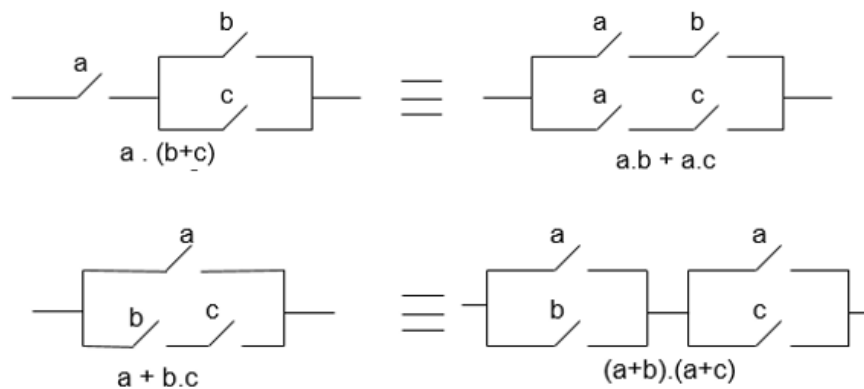
$$1.a = a$$



C)_ Cada operación es **distributiva** con respecto a la otra:

$$a.(b+c) = a.b+a.c$$

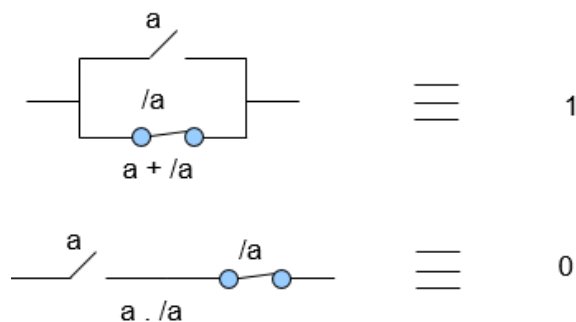
$$a+b.c = (a+b).(a+c)$$



D)_ Para cada elemento "a" del álgebra existe un elemento "/a" (se lo llama a **negado**) tal que:

$$a + /a = 1$$

$$a . /a = 0$$




Lógica, compuertas y tabla de verdad

Compuertas lógicas

_ A continuación, podemos observar los diferentes tipos de compuertas lógicas y los resultados que producen:


AND



$F = x \cdot y$


x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

OR




$F = x + y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0
x	F								
0	1								
1	0								


Buffer



$F = x$

x	F
0	0
1	1

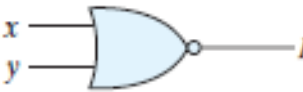
NAND



$F = (xy)'$

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0


NOR



$F = (x + y)'$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR (XOR)



$$F = xy' + x'y$$
$$= x \oplus y$$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR
or
equivalence



$$F = xy + x'y' \\ = (x \oplus y)'$$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

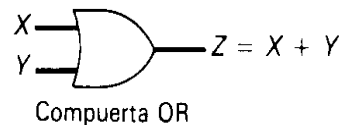
_ Los posibles valores binarios de la operación OR lógica son los siguientes:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$



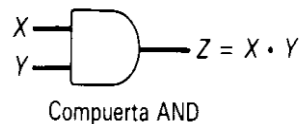
_ Los de la operación AND lógica son:

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$



_ Y los de la operación NOT:

$$x = 1 \quad /x=0 \text{ y viceversa.}$$

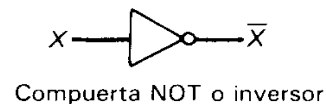


Tabla de verdad

_ Una tabla de verdad es una disposición de las combinaciones de las variables binarias que muestra la relación entre los valores que pueden tomar las variables y el resultado de la operación.

Tablas de verdad de las tres operaciones lógicas

AND			OR			NOT	
X	Y	X · Y	X	Y	X + Y	X	\bar{X}
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

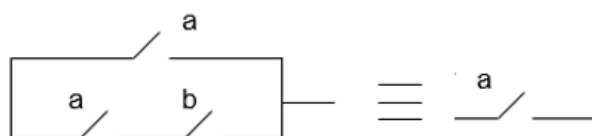
Teoremas

Principio de dualidad: dada una igualdad, si se cambia suma lógica (+) por producto lógico (.), producto lógico por suma lógica, ceros por unos y unos por ceros, la igualdad permanece válida. Por ejemplo:

	Adición	Producto
1	$a + \bar{a} = 1$	$a \cdot \bar{a} = 0$
2	$a + 0 = a$	$a \cdot 1 = a$
3	$a + 1 = 1$	$a \cdot 0 = 0$
4	$a + a = a$	$a \cdot a = a$
5	$a + b = b + a$	$a \cdot b = b \cdot a$
6	$a + (b + c) = (a + b) + c$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
7	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = a \cdot b + a \cdot c$
8	$a + a \cdot b = a$	$a \cdot (a + b) = a$
9	$\overline{(a + b)} = \bar{a} \cdot \bar{b}$	$\overline{(a \cdot b)} = \bar{a} + \bar{b}$

Ley de Absorción: el termino b es absorbido por a (sucede lo mismo en su forma dual):

$$a + a \cdot b = a$$



Teorema del consenso: dada la siguiente operación:

$$x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$$

_ Se demuestra que el tercer término es redundante y se puede eliminar.

$$\begin{aligned}
 x \cdot y + \bar{x} \cdot z + y \cdot z &= x \cdot y + \bar{x} \cdot z + y \cdot z (x + \bar{x}) \\
 &= x \cdot y + \bar{x} \cdot z + x \cdot y \cdot z + \bar{x} \cdot y \cdot z \\
 &= x \cdot y + x \cdot y \cdot z + \bar{x} \cdot z + \bar{x} \cdot y \cdot z \\
 &= x \cdot y (1 + z) + \bar{x} \cdot z (1 + y) \\
 &= x \cdot y + \bar{x} \cdot z
 \end{aligned}$$

Teorema de De Morgan: se aplica para obtener el complemento de una operación y se puede verificar por medio de la tabla de verdad en donde se asigna todos los valores posibles a X y a Y.

$$\overline{(x + y)} = \bar{x} \cdot \bar{y}$$

$$\overline{(x \cdot y)} = \bar{x} + \bar{y}$$

Tablas de verdad para verificar el teorema de DeMorgan

A.	X	Y	X + Y	$\overline{(X + Y)}$	B.	X	Y	\bar{X}	\bar{Y}	$\bar{X} \cdot \bar{Y}$
	0	0	0	1		0	0	1	1	1
	0	1	1	0		0	1	1	0	0
	1	0	1	0		1	0	0	1	0
	1	1	1	0		1	1	0	0	0

Identidades básicas del álgebra booleana

_ Observamos la siguiente tabla:

Identidades básicas del álgebra booleana

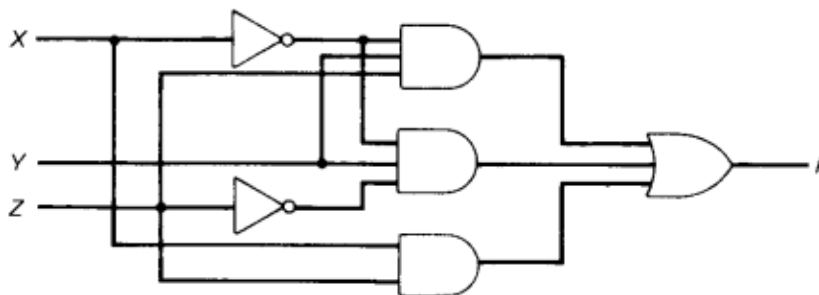
1. $X + 0 = X$	2. $X \cdot 1 = X$	
3. $X + 1 = 1$	4. $X \cdot 0 = 0$	
5. $X + \bar{X} = 1$	6. $X \cdot X = X$	
7. $X + \bar{X} = 1$	8. $X \cdot \bar{X} = 0$	
9. $\bar{\bar{X}} = X$		
10. $X + Y = Y + X$	11. $XY = YX$	Conmutativa
12. $X + (Y + Z) = (X + Y) + Z$	13. $X(YZ) = (XY)Z$	Asociativa
14. $X(Y + Z) = XY + XZ$	15. $X + YZ = (X + Y)(X + Z)$	Distributiva
16. $\overline{X + Y} = \bar{X} \cdot \bar{Y}$	17. $\overline{X \cdot Y} = \bar{X} + \bar{Y}$	DeMorgan

Manipulación Algebraica

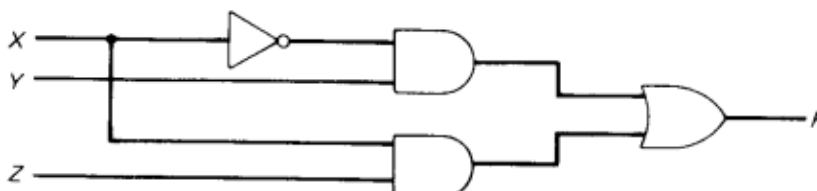
_ Consideramos el siguiente ejemplo:

$$\begin{aligned} F &= \bar{X}YZ + \bar{X}Y\bar{Z} + XZ \\ &= \bar{X}Y(Z + \bar{Z}) + XZ \\ &= \bar{X}Y \cdot 1 + XZ \\ &= \bar{X}Y + XZ \end{aligned}$$

por la identidad 14
por la identidad 7
por la identidad 2



(a) $F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$



(b) $F = \bar{X}Y + XZ$

Función

_ Una función del álgebra de Boole es una expresión algebraica en las que aparecen las variables de las cuales la función depende en forma directa o negada y relacionadas por las operaciones de suma lógica y/o producto lógico.

$$f(a,b,c) = a.b.c + /a.b./c + /b./c + a./b$$

$$f(a,b,c) = (a+b+c).(/a+b+/c).(/b+/c).(a+/b)$$

Término canónico

_ Un término es canónico cuando aparecen todas las variables de las cuales la función depende ya sea en forma directa o negada.

$$f(a,b,c) = a.b.c + /a.b./c + /b./c + a./b$$

 Producto canónico

$$f(a,b,c) = (a+b+c) . (/a+b+/c) . (/b+/c).(a+/b)$$

 Suma canónica

Función canónica

_ Una función es canónica cuando todos sus términos son canónicos. Puede ser expresada como suma de productos canónicos o como producto de sumas canónicas.

$$f(a,b,c) = a.b.c + /a.b./c + a./b./c + a./b./c$$

$$f(a,b,c) = (a+b+c).(/a+b+/c).(a+/b+/c).(a+/b+c)$$

_ En una función expresada como suma de productos canónicos, cada producto canónico representa un uno de la función. Basta que uno de los términos sea uno para que la función valga uno.

$$f(a,b,c) = a.b.c + /a.b./c + a./b./c + a./b./c$$

_ En una función expresada como producto de sumas canónicas, cada suma canónica representa un cero de la función. Basta que uno de los términos sea cero para que la función valga cero.

$$f(a,b,c) = (a+b+c).(/a+b+/c).(a+/b+/c).(a+/b+c)$$

_ Tanto la suma de productos como el producto de sumas constituyen la representación algebraica de la función.

Representación de la función mediante tabla de verdad:

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$f(a,b,c) = \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{b} \cdot c + a \cdot b \cdot c$

Representación de la función mediante:

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

- Suma de productos

$$f(a,b,c) = \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{b} \cdot c + a \cdot b \cdot c$$

- Producto de sumas

$$f(a,b,c) = (a+b+c) \cdot (a+b+\overline{c}) \cdot (a+\overline{b}+c) \cdot (\overline{a}+b+c)$$

Representación matemática de la función:

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

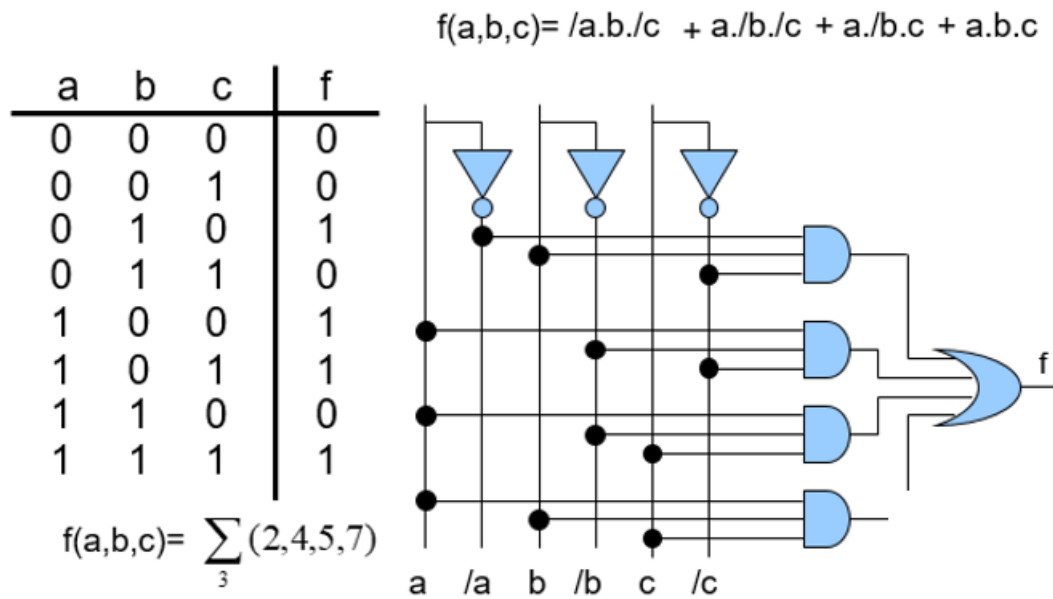
$$f(a,b,c) = \overbrace{\overline{a} \cdot b \cdot \overline{c}}^2 + \overbrace{a \cdot \overline{b} \cdot \overline{c}}^4 + \overbrace{a \cdot \overline{b} \cdot c}^5 + \overbrace{a \cdot b \cdot c}^7$$

$$f(a,b,c) = \sum_3 (2,4,5,7)$$

$$f(a,b,c) = (a+b+c) \cdot (a+b+\overline{c}) \cdot (a+\overline{b}+c) \cdot (\overline{a}+b+c)$$

$$f(a,b,c) = \prod_3 (7,6,4,1)$$

Representación esquemática de la función:



Minterms y Maxterms

Minterms and Maxterms for Three Binary Variables

x	y	z	Minterms		Maxterms	
			Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

$$f_2 = (x + y + z)(x + y + z')(x + y' + z)(x' + y + z)$$

$$= M_0 M_1 M_2 M_4$$

Representación de la función mediante mini y maxitérminos: en este caso, los números siguen la definición de mini y maxitérmino indicadas anteriormente.

a	b	c	f	
0	0	0	0	$f(a,b,c) = \overline{a.b.c} + a.\overline{b.c} + a.\overline{b.c} + a.b.c$
0	0	1	0	$f(a,b,c) = \sum (2,4,5,7)$
0	1	0	1	
0	1	1	0	Representación como minitérminos
1	0	0	1	
1	0	1	1	$f(a,b,c) = (a+b+c).(a+b+\overline{c}).(a+\overline{b}+\overline{c}).(\overline{a}+\overline{b}+c)$
1	1	0	0	
1	1	1	1	$f(a,b,c) = \prod(0,1,3,6)$

Representación como maxitérminos

Complemento de una función

_ El complemento de una función F se obtiene a partir de un intercambio de unos por ceros y de ceros por unos, en los valores de F de la tabla de verdad. El complemento puede determinarse en forma algebraica aplicando el teorema De Morgan intercambiando operaciones AND y OR, y complementando cada variable. Por ejemplo:

$$\begin{aligned}
 F1 &= \overline{X.Y.Z} + \overline{X.Y.Z} \\
 &= \overline{(\overline{X.Y.Z} + \overline{X.Y.Z})} \\
 &= \overline{(\overline{X.Y.Z}) . (\overline{X.Y.Z})} \\
 &= (\overline{\overline{X+Y+Z}}) . (\overline{\overline{X+Y+Z}}) \\
 &= (X+Y+Z) . (X+Y+Z)
 \end{aligned}$$

$$\begin{aligned}
 F2 &= X . (\overline{Y.Z} + Y.Z) \\
 &= \overline{X . (\overline{Y.Z} + Y.Z)} \\
 &= \overline{X} . \overline{(\overline{Y.Z} + Y.Z)} \\
 &= \overline{\overline{X}} . (\overline{\overline{Y.Z}} + \overline{Y.Z}) \\
 &= X . ((Y+Z) . (\overline{Y+Z}))
 \end{aligned}$$

Karnaugh

Simplificación por Karnaugh

Simplificación

_ Tenemos que saber que la simplificación produce un diagrama de circuitos lógicos con un número mínimo de compuertas y el número mínimo de entradas a las compuertas.

_ La expresión más simple no es necesariamente única. La función del álgebra de Boole puede tener más de un mínimo.

_ Las funciones en la forma de suma de productos canónicos se pueden simplificar mediante manipulaciones algebraicas; sin embargo, es complicado. El método del mapa de Karnaugh nos da una forma sistemática para hacerlo.

Adyacencia lógica

_ Dos términos son adyacentes lógicamente cuando difieren solamente en el estado de una variable, por ejemplo:

$$a./b.c + a.b.c$$

_ Como podemos observar en dicha operación, los términos difieren en la variable b. En ese caso se puede simplificar porque los dos términos se pueden agrupar como:

$$a.c.(b+/b)$$

$$a.c$$

_ Y queda solamente a.c porque $(b+/b)$ es igual a 1.

Mapa de Karnaugh

_ Este mapa se basa en que cada casilla del mapa es adyacente, físicamente y lógicamente, con la que tiene al lado.

		c				
		d	00	01	11	10
a	b					
	00		0	1	3	2
	01		4	5	7	6
	11		12	13	15	14
	10		8	9	11	10

Mapa de dos variables

_ Hay cuatro minitérminos en una función booleana con dos variables, por lo tanto, el mapa de dos variables consta de cuatro cuadrados, uno para cada termino mínimo. A continuación, vemos la representación:

m_0	m_1
m_2	m_3

	Y	0	1
X	0	$\bar{X}\bar{Y}$	$\bar{X}Y$
	1	$X\bar{Y}$	XY

_ A modo de ejemplo:

$$m_1 + m_2 + m_3 = \bar{X}Y + X\bar{Y} + XY = X + Y$$

_ Consideramos los siguientes casos, en donde XY es el mínimo termino de m3 por lo que simplemente se coloca un 1:

	Y	0	1
X	0		
	1		1

(a) XY

	Y	0	1
X	0		1
	1	1	1

(b) $X + Y$

_ Y en el caso de $X + Y$ se encierra a el 1 que puede pertenecer tanto a X como a Y, por ende, se realiza una simplificación algebraica:

$$\bar{X}Y + X\bar{Y} + XY = \bar{X}Y + X(\bar{Y} + Y) = (\bar{X} + X)(Y + X) = X + Y$$

Mapa de tres variables

_ Hay ocho minitérminos para tres variables binarias. Por lo tanto, un mapa de tres variables consta de ocho cuadrados:

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

		YZ			Y
		00	01	11	10
X	0	$\bar{X}\bar{Y}\bar{Z}$	$\bar{X}\bar{Y}Z$	$\bar{X}YZ$	$\bar{X}Y\bar{Z}$
	1	$X\bar{Y}\bar{Z}$	$X\bar{Y}Z$	XYZ	$XY\bar{Z}$

Z

_ A continuación, realizamos algunas operaciones de ejemplo:

$$m_0 + m_2 = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} = \bar{X}\bar{Z}(\bar{Y} + Y) = \bar{X}\bar{Z}$$

$$m_4 + m_6 = X\bar{Y}\bar{Z} + XY\bar{Z} = X\bar{Z}(\bar{Y} + Y) = X\bar{Z}$$

$$m_5 + m_7 = X\bar{Y}Z + XYZ = XZ(\bar{Y} + Y) = XZ$$

$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z} \\ &= \bar{X}\bar{Z}(\bar{Y} + Y) + X\bar{Z}(\bar{Y} + Y) \\ &= \bar{X}\bar{Z} + X\bar{Z} = \bar{Z}(\bar{X} + X) = \bar{Z} \end{aligned}$$

_ Ahora, se nos presenta la situación de simplificar las siguientes funciones booleanas:

1)_

$$F(X, Y, Z) = \Sigma m(2, 3, 4, 5)$$

		YZ		Y	
		00	01	11	10
X	0			1	1
X	1	1	1		

$$F = \bar{X}Y + X\bar{Y}$$

2)_

$$F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7)$$

		YZ		Y	
		00	01	11	10
X	0			1	
X	1	1		1	1

$$F_1 = YZ + X\bar{Z}$$

3)_

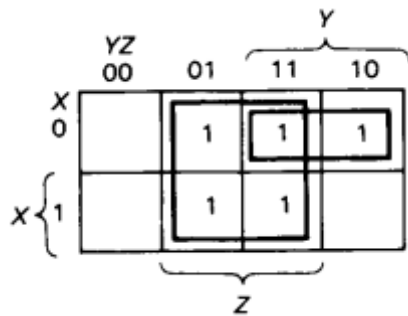
$$F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6)$$

		YZ		Y	
		00	01	11	10
X	0	1			1
X	1	1	1		1

$$F_2 = \bar{Z} + X\bar{Y}$$

4)_

$$F(X, Y, Z) = \sum m(1, 2, 3, 5, 7)$$

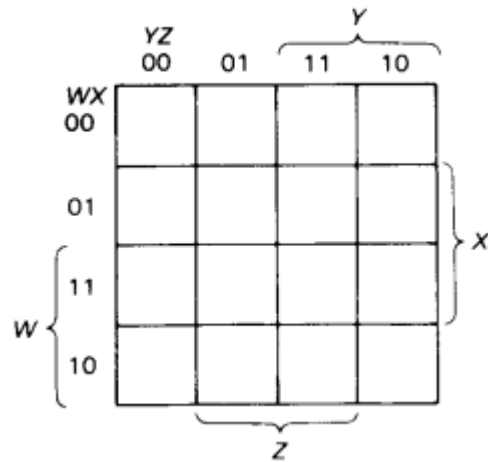


$$F = Z + \bar{X}Y$$

Mapa de cuatro variables

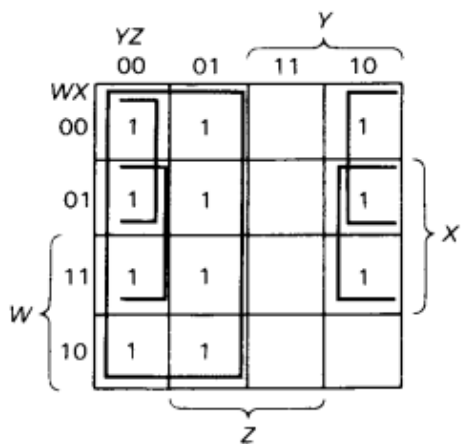
_ Hay 16 minitérminos para cuatro variables binarias y, en consecuencia, un mapa de cuatro variables consta de 16 cuadrados. A continuación, vemos la representación:

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}



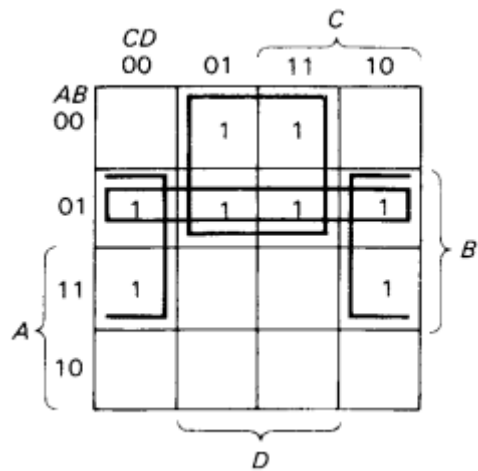
_ Como ejemplo, debemos simplificar la siguiente función:

$$F(W, X, Y, Z) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$



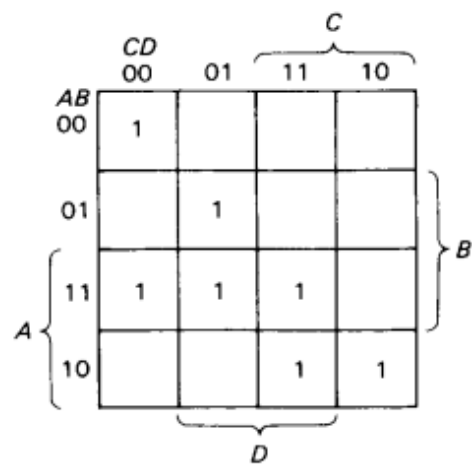
$$F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$$

_ Es importante considerar en estos casos que la forma de resolver es determinando grupos de cuatro unos adyacentes, llamados primos. Es decir, siempre agrupamos de a dos, cuatro, ocho, etc, siempre y cuando estén unidos. Por ejemplo:

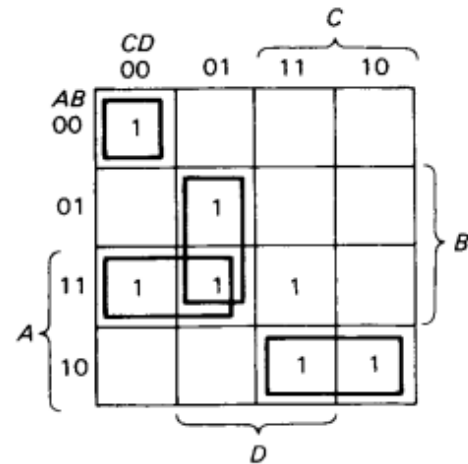


$$F = \overline{A}D + B\overline{D}$$

_ Otro ejemplo:



(a) Graficación de los minterminos



(b) Implicantes primos esenciales

$$F = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{B}\overline{C}D + A\overline{B}\overline{C} + A\overline{B}C + \begin{cases} ACD \\ \text{o} \\ ABD \end{cases}$$

Reglas

_ A continuación, vemos un ejemplo:

$$f(a,b,c) = \bar{a}.b.\bar{c} + a.\bar{b}.\bar{c} + a.\bar{b}.c + a.b.c$$

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$$f(a,b,c) = \sum_3 (2,4,5,7)$$

a \ c	0	1
b		
00		
01	1	
11		1
10	1	1

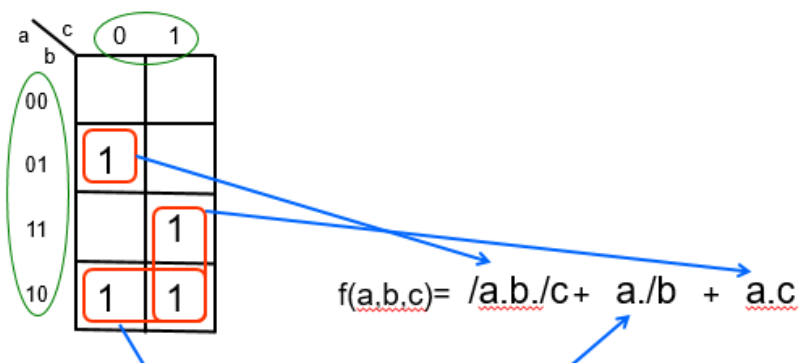
1)_ Se deben agrupar todos los unos del mapa con la menor cantidad de agrupamientos posibles y con la mayor cantidad posibles de unos por agrupamiento.

a \ c	0	1
b		
00		
01	1	
11		1
10	1	1

2)_ Los agrupamientos de los unos pueden ser hechos de a uno, dos, cuatro, ocho, etc.

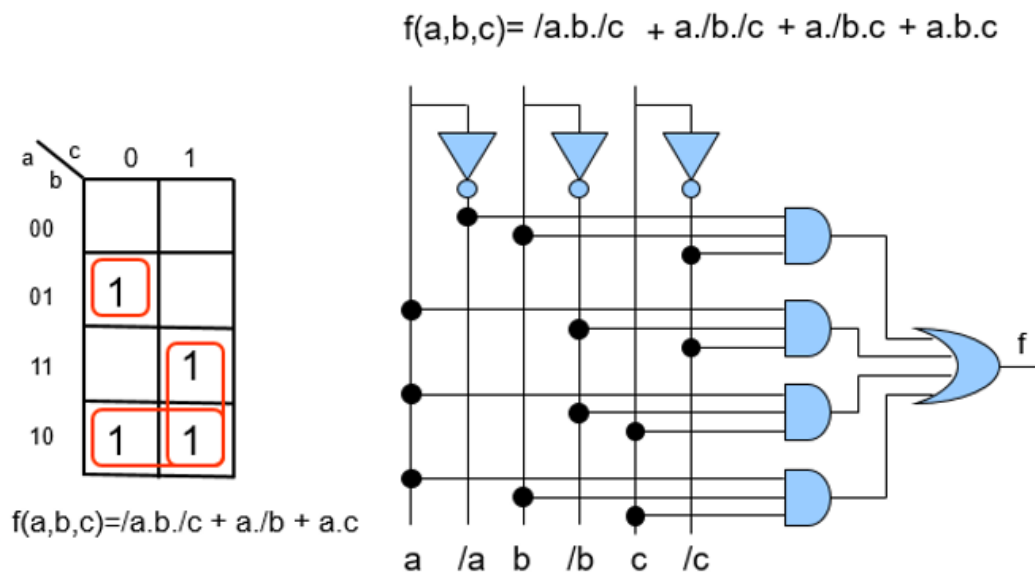
Extracción de la función

_ Para extraer la función, habrá tantos términos en la función como agrupamientos haya en el mapa. Las variables adoptarán la forma directa o negada de acuerdo a si en el mapa valen uno o cero.



_ Las variables que cambian dentro del agrupamiento desaparecen.

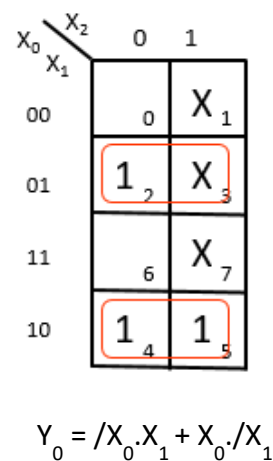
Implementación de la función



Condiciones “sin cuidado” o “no importa”

_ Hay aplicaciones donde la función no se especifica para ciertas combinaciones de las variables. Estas condiciones sin cuidado se pueden aprovechar en un mapa de Karnaugh para simplificar más la función.

X_0	X_1	X_2	Y_0
0	0	0	0
0	0	1	X ?
0	1	0	1
0	1	1	X ?
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	X ?



_ Cuando se simplifica la función, se puede optar por incluir cada término “no importa” con los unos o con los ceros, dependiendo de que combinación genere la expresión más simple.

_ En el siguiente ejemplo, los minterminos F son las combinaciones de las variables que hacen que la función sea igual a 1, y los minterminos de d son aquellos que no importa a los que se les puede asignar 0 o 1. En este caso obtenemos dos resultados de F que son algebraicamente desiguales, pero ambas son aceptables:

$$F(W, X, Y, Z) = \sum m(1, 3, 7, 11, 15)$$

$$d(W, X, Y, Z) = \sum m(0, 2, 5)$$

		Y			
		YZ		11	10
W	WX	00	01	11	10
	00	X	1	1	X
	01	0	X	1	0
	11	0	0	1	0
	10	0	0	1	0

$$F = YZ + \overline{W}X$$

		Y			
		YZ		11	10
W	WX	00	01	11	10
	00	X	1	1	X
	01	0	X	1	0
	11	0	0	1	0
	10	0	0	1	0

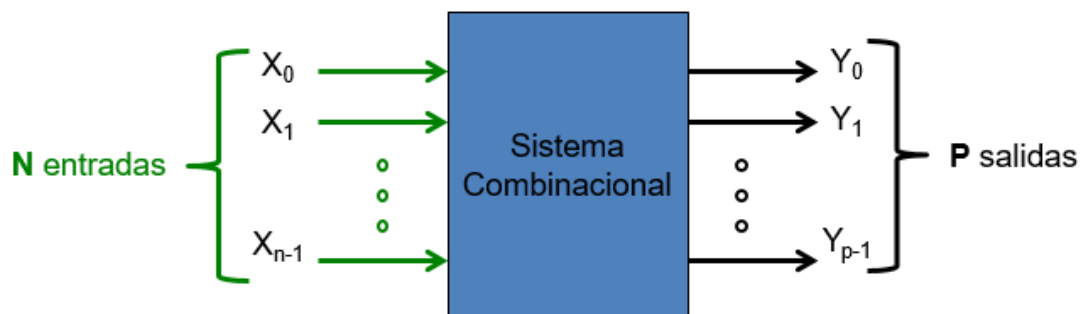
$$F = YZ + \overline{W}Z$$

Sistemas combinacionales

Introducción

Definición

_ Un sistema combinacional es un sistema en donde las salidas dependen pura y exclusivamente del valor de las entradas.



X_0	X_1	X_2	Y_0	Y_1
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

$$Y_0 = \sum_3(2,4,5,7)$$

$$Y_1 = \sum_3(1,2,3,7)$$

_ Cada salida es una función del algebra de Boole. A continuación, vemos la simplificación:

$$Y_0 = \sum_3(2,4,5,7)$$

$$Y_1 = \sum_3(1,2,3,7)$$

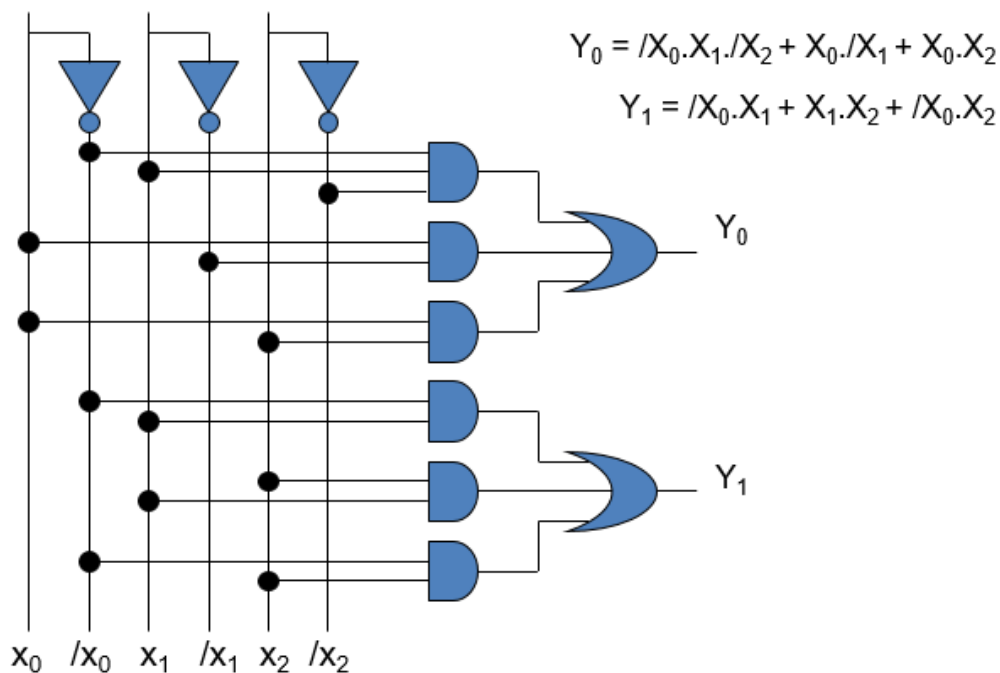
$X_0 \backslash X_1$	0	1
0	0	1
1	1	0
2	0	1
3	1	0
4	1	0
5	0	1
6	0	1
7	1	0

$X_0 \backslash X_1$	0	1
0	0	1
1	1	0
2	1	1
3	0	1
4	1	0
5	0	1
6	0	1
7	1	0

$$Y_0 = /X_0.X_1./X_2 + X_0./X_1 + X_0.X_2$$

$$Y_1 = /X_0.X_1 + X_1.X_2 + /X_0.X_2$$

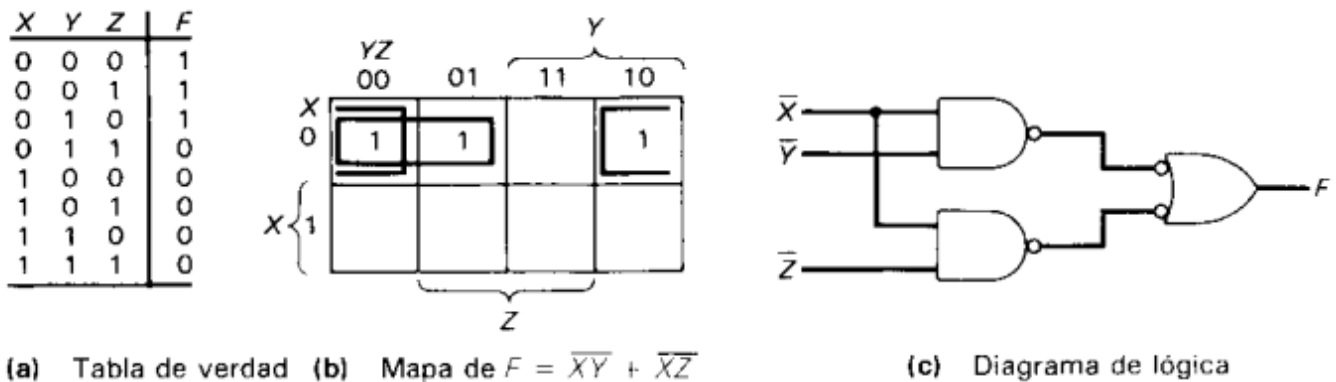
_ Luego mostramos la implementación:



_ Un circuito combinatorio consta de variables de entrada, compuertas lógicas y variables de salida, en donde las compuertas lógicas aceptan señales de las entradas y generan señales en las salidas. Este proceso transforma información binaria de datos de entrada en datos de salida requeridos. Cada variable de entrada y salida existe físicamente como una señal binaria que representa el equivalente a unos y ceros lógicos. Para n variables de entrada, existen 2^n combinaciones binarias posibles, y para cada combinación binaria de las variables de entrada existe un valor binario de salida posible.

Procedimiento de diseño

_ Como ejemplo, tenemos que diseñar un circuito combinatorio con tres entradas y una salida. La salida debe ser un 1 lógico cuando el valor binario de las entradas sea menor que tres y 0 lógico en caso contrario. Utilizar solo compuertas NAND. Entonces:



_ Como podemos observar, F es 1 cuando las entradas son 0, 1 y 2, caso contrario F es 0, y luego se representa el diagrama con compuertas NAND.

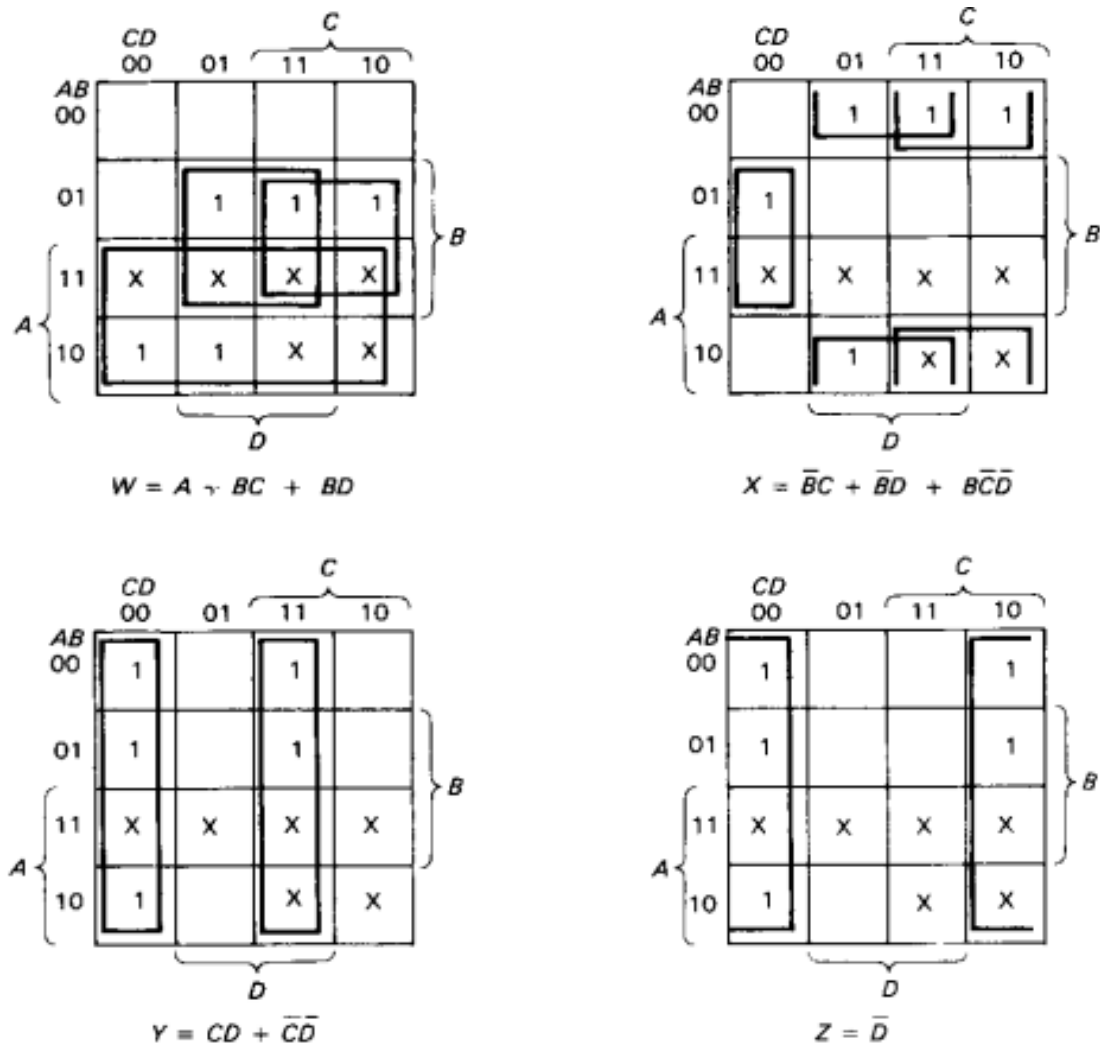
Convertidor de código

_ Cuando un circuito combinatorio tiene dos o más salidas, cada salida debe expresarse en forma independiente como funciones de todas las variables de entrada. Por ende, un ejemplo de un circuito de múltiples salidas es un convertidor de código que es un circuito que traduce información de un código binario a otro.

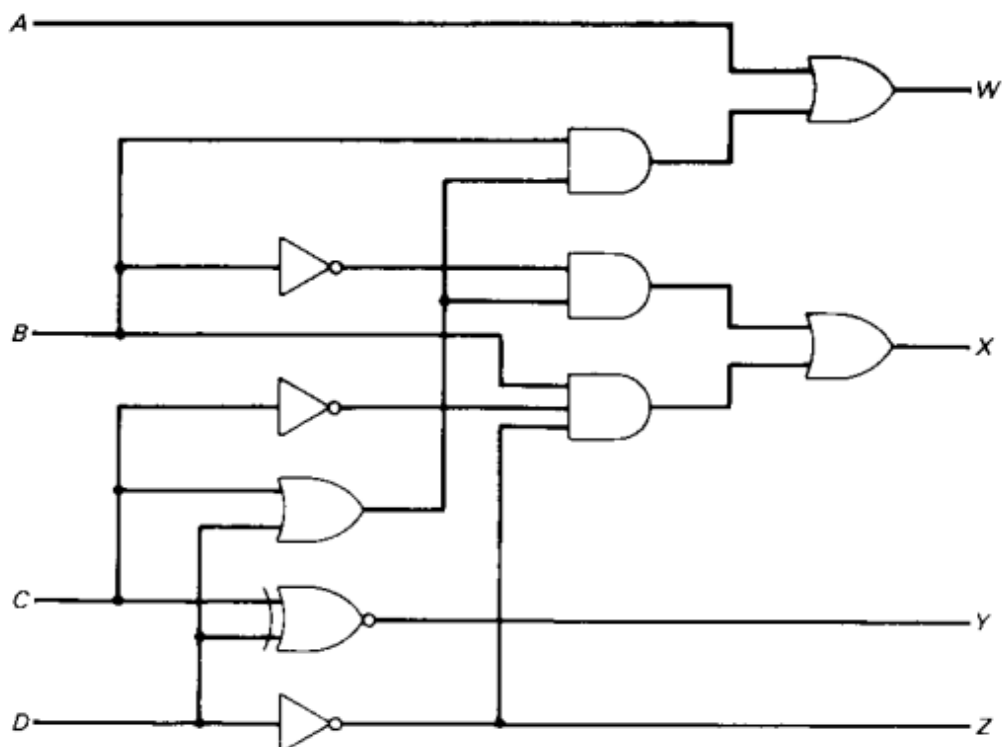
Tabla de verdad del ejemplo del convertidor de código

Dígito decimal	Entrada BCD				Salida exceso-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

_ Realizamos el mapa del convertidor:



_ Luego realizamos el diagrama de lógica:



Semisumador (sumador medio)

_ Un semisumador es un circuito aritmético que genera la suma de dos dígitos binarios. Este tiene dos entradas y dos salidas. Las variables de entrada son los bits sumando y consumando que se sumaran y las variables de salida producen la suma y el acarreo. Las funciones booleanas de las dos salidas se pueden obtener fácilmente con la tabla de verdad:

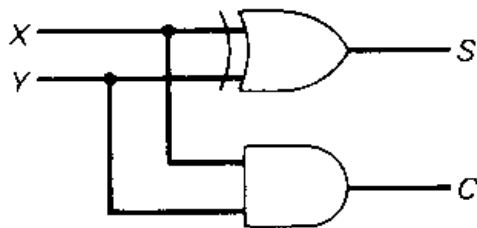
Tabla de verdad del semisumador

Entradas		Salidas	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = \bar{X}Y + X\bar{Y} = X \oplus Y$$

$$C = XY$$

_ El semisumador se puede construir con una compuerta OR excluyente y una AND:



Sumador completo

_ Un sumador completo es un circuito combinatorio que forma la suma aritmética de tres bits de entrada. Consta de tres entradas y dos salidas. La variable S da el valor del bit menos significativo de la suma y la variable binaria C es el acarreo de salida. Como vemos en la tabla de verdad:

- Cuando todos los bits de entrada valen 0, todas las salidas valen 0.
- La salida S es igual a 1 solo cuando una entrada es igual a 1 o cuando las tres entradas son iguales a 1.
- La salida C tiene un acarreo de 1 i dos o tres entradas son iguales a 1.

Tabla de verdad del sumador completo

Entradas			Salidas	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$C = XY + XZ + YZ$$

	YZ		Y	
	00	01	11	10
X				
0		1		1
1	1		1	

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$= X \oplus Y \oplus Z$$

	YZ		Y	
	00	01	11	10
X				
0			1	
1		1	1	1

$$C = XY + XZ + YZ$$

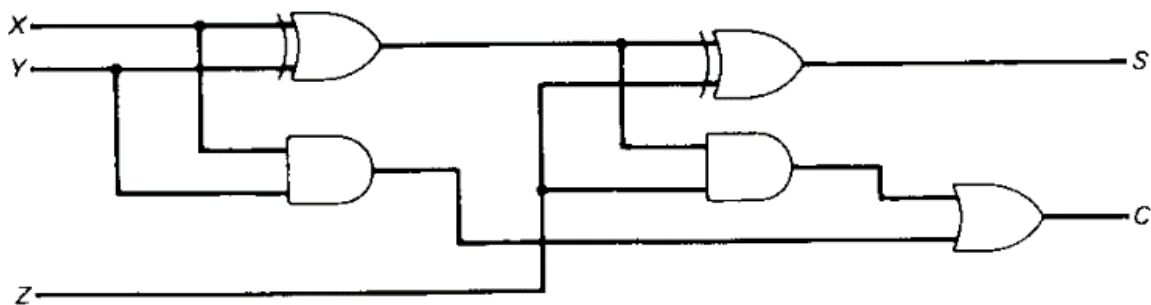
$$= XY + Z(XY + \bar{X}Y)$$

$$= XY + Z(X \oplus Y)$$

_ El diagrama de lógica consta de dos semisumadores y una compuerta OR:

$$S = X \oplus Y \oplus Z$$

$$C = XY + Z(X \oplus Y)$$



Complemento a la base y a la base-1

_ Existen dos tipos de complementos para cada sistema de base β . El complemento a la base y el complemento a la base-1. Cuando el valor de la base se sustituye en el nombre, los dos tipos se conocen como complemento a 2 y complemento a 1 en el sistema binario, y como complemento a 10 y complemento a 9 en el sistema decimal.

_ Dado un número N en base β que tiene n dígitos, el complemento a $(\beta-1)$ de N se define como: $(\beta^n-1)-N$

_ Y el complemento a β de N se define como: β^n-N

_ Donde β^n representa un número que consta de un 1 seguido de n ceros. En el caso de números binarios $\beta=2$ y (2^n-1) es un número binario representado por n unos.

Complemento a uno

_ En este caso cambiamos los unos por ceros y los ceros por unos, es decir, alternamos:

$$1101011 \rightarrow 0010100$$

Complemento a dos

_ Analizamos lo siguiente:

$(10101100)_2 \rightarrow$ complemento a 2

$$\begin{array}{r} 11111^2 \\ \text{~~~~~} \\ 100000000 \\ - 10101100 \\ \hline 001010100 \end{array}$$

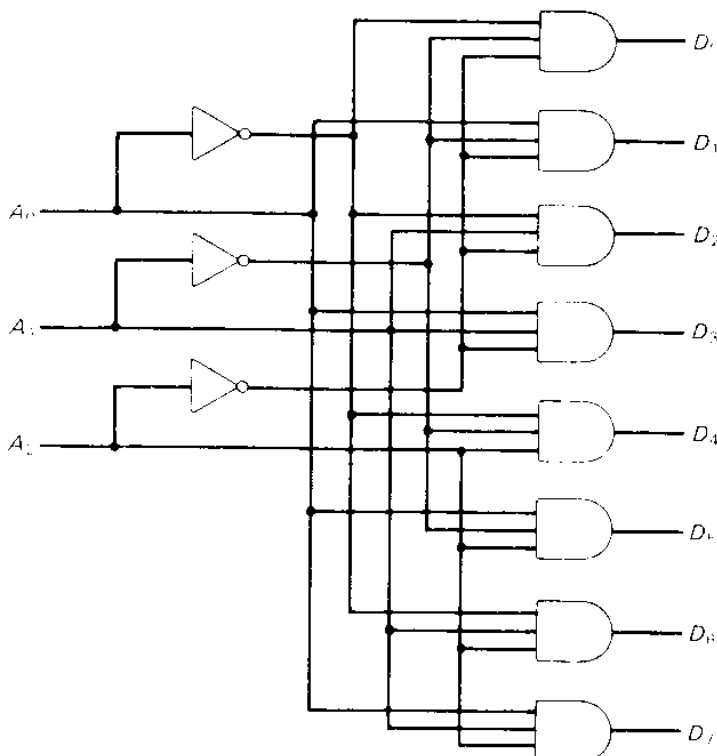
_ Una forma más práctica, consiste en invertir unos por ceros y ceros por unos y al resultado le sumo uno:

$$\begin{array}{r} 10101100 \\ 01010011 \\ + 1 \\ \hline 01010100 \end{array}$$

Circuitos combinatorios

Decodificador

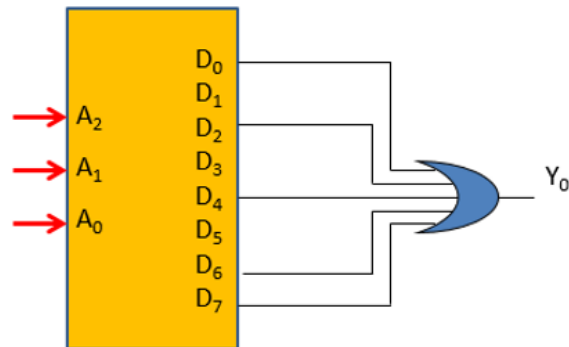
_ Un decodificador es un circuito combinatorio que convierte información binaria de las n entradas codificadas a un máximo de 2^n salidas únicas. A continuación, vemos una representación:



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

_ Realizamos un ejemplo de la implementación:

A_2	A_1	A_0	Y_0
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

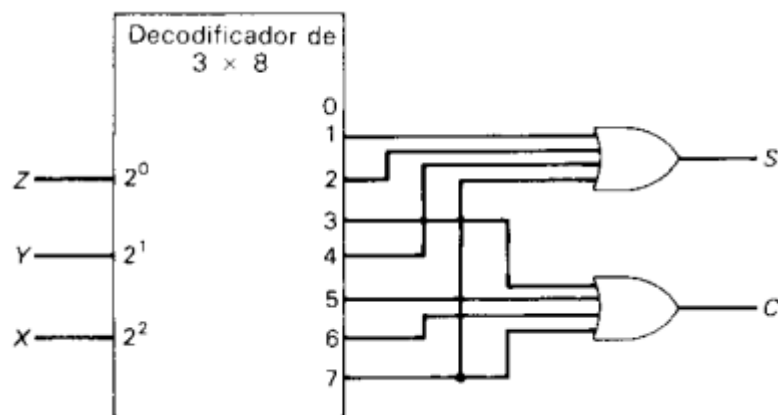


$$Y_0 = \sum_3 (0, 2, 4, 6, 7)$$

_ Otro ejemplo a resolver:

$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

$$C(X, Y, Z) = \sum m(3, 5, 6, 7)$$



Codificador

_ Un codificador es un circuito combinatorio que realiza la operación inversa de un decodificador.

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

DECODIFICADOR

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

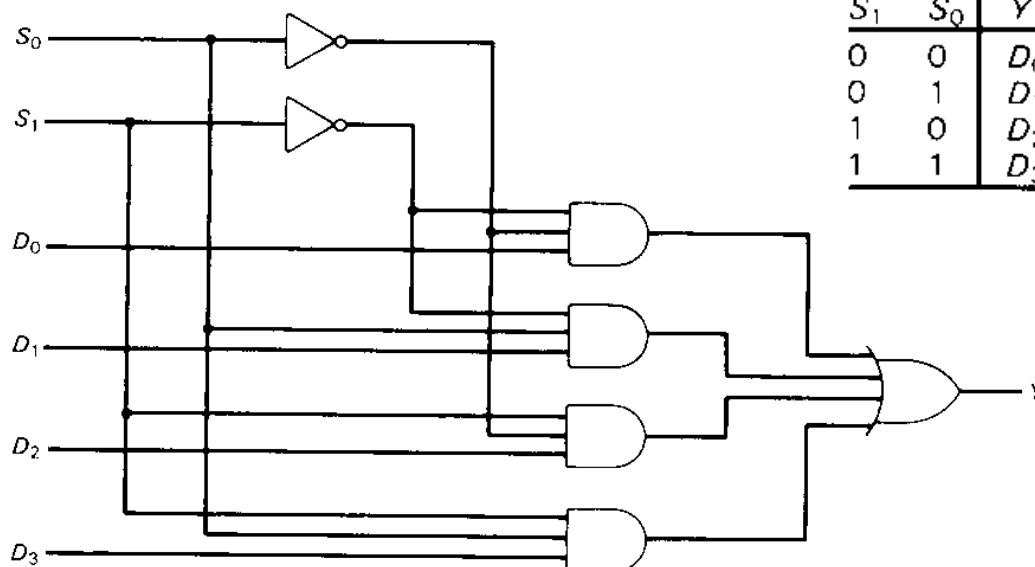
CODIFICADOR

Multiplexor

_ Un multiplexor es un circuito combinatorio que selecciona información binaria de una entre muchas líneas de entrada y la dirige a una sola línea de salida. Hay 2^n líneas de entrada y n líneas de selección.

Tabla de funciones

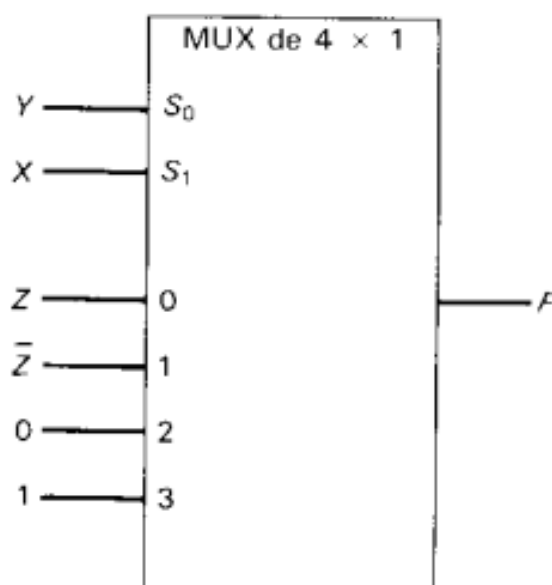
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3



_ Se puede implementar una función booleana de n variables con un multiplexor que tiene $n-1$ entradas de selección. Las primeras $n-1$ variables de la función se conectan a las entradas de selección del multiplexor. La variable que resta de la función se la utiliza para la entrada de datos. A continuación, vemos la ejecución de una función booleana con un multiplexor:

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	0
1	1	0	1
1	1	1	1

(a) Tabla de verdad

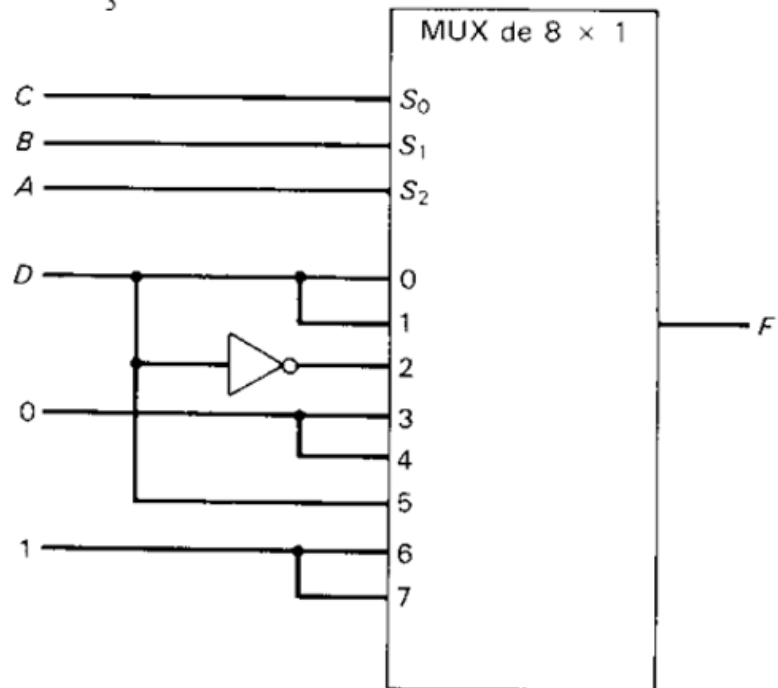


(b) Ejecución con el multiplexor

_ A continuación, tenemos otro ejemplo de la implementación:

$$F(A,B,C,D) = \sum_3 (1,3,4,11,12,13,14,15)$$

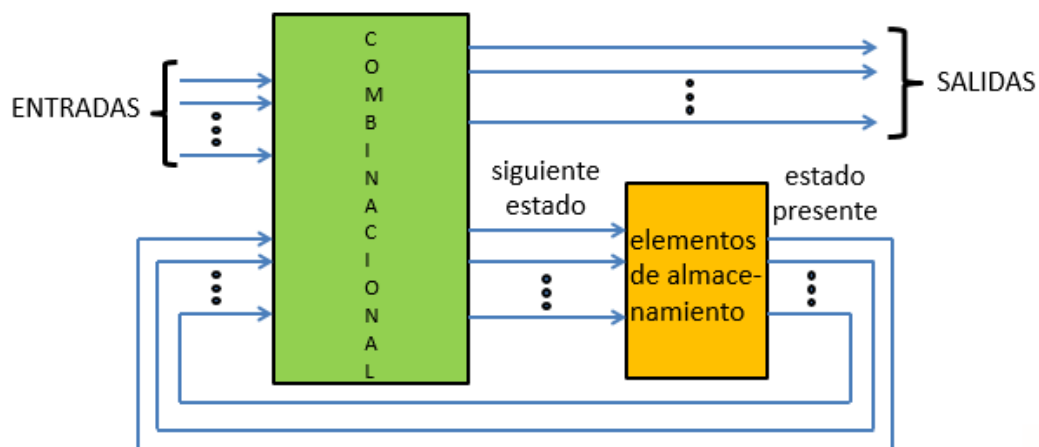
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



Sistemas secuenciales

Introducción

_ En un sistema secuencial, las salidas no dependen únicamente de las entradas sino también de los estados internos. El circuito secuencial recibe información binaria de entradas externas, las cuales, junto con el estado presente de los elementos de almacenamiento, determinan el valor binario de las salidas, así como la condición para cambiar el estado de los elementos de almacenamiento.



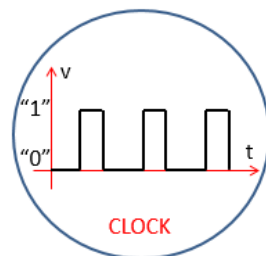
_ Hay dos tipos de circuitos secuenciales y estos son:

Circuito secuencial asíncrono: depende del orden en el que cambien las entradas, y el estado del circuito puede ser afectado en cualquier instante.

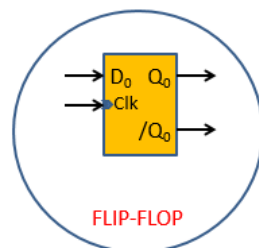
Circuito secuencial síncrono: sistema cuyo comportamiento puede definirse a partir del conocimiento de sus señales en instantes discretos. La sincronización se logra mediante un generador de señales de reloj que produce una señal periódica de pulsos de reloj. Por ende, los circuitos secuenciales síncronos que usan pulsos de reloj en la entrada se llaman circuitos secuenciales controlados por reloj, y los elementos de almacenamiento que se emplean en estos circuitos secuenciales se denominan multivibradores biestables o Flip-Flops.

_ Entonces, se introduce dos conceptos:

- Tiempo



- Memoria



Flip-Flop

_ Estos son dispositivos de almacenamiento binario que pueden contener un bit de información. Normalmente, un circuito secuencial utilizara muchos Flip-Flops para almacenar tantos bits como sea necesario. Un circuito multivibrador biestable tiene dos salidas, una para el valor normal y una para el valor complementado del bit que esta almacenado en él. La información binaria puede entrar en un Flip-Flop de diversas maneras, lo que da origen a distintos tipos del mismo.

Flip-Flop D: este depende solo de la entrada D (dato de entrada) y es independiente del estado presente. Este multivibrador no tiene condición de cambio. Y para calcular el número de Flip-Flops dentro del sistema usamos $\log_2(N)$, donde N es el número de estados. Se puede expresar como:

(c) Flip-flop D		
Q(t)	Q(t + 1)	D
0	0	0
0	1	1
1	0	0
1	1	1

$$Q_{t+1} = D_t$$

(c) Flip-flop D		
D	Q(t + 1)	Operación
0	0	Reiniciación
1	1	Iniciación

_ En donde:

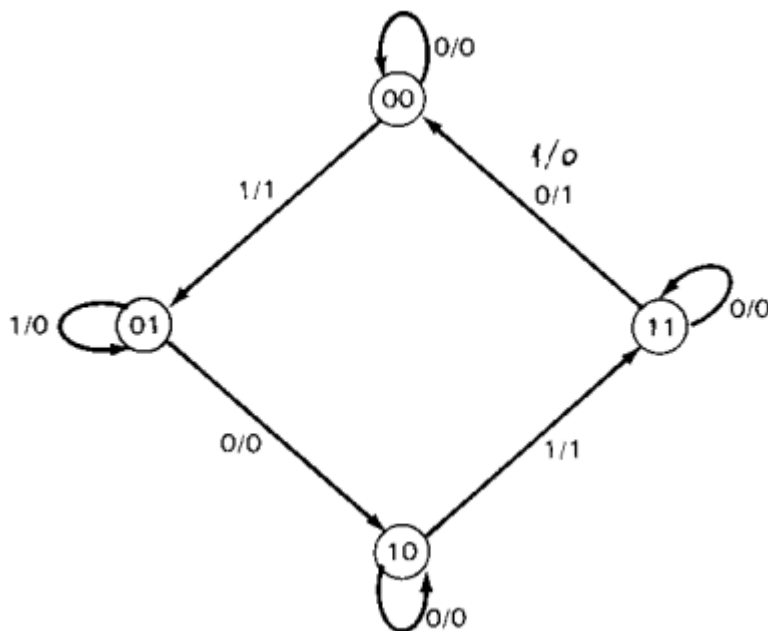
- Q_t : representa el estado presente antes de la aplicación del pulso.
- $Q(t+1)$: representa el estado siguiente, un periodo despues.

Diseño con multivibradores biestables D

_ El procedimiento de diseño se ejecuta siguiendo los pasos que se indican a continuación:

- 1)_ Se obtiene la tabla de estados a partir del planteamiento del problema o del diagrama de estados.
- 2)_ Se determinan las ecuaciones de entrada de los Flip-Flops a partir de las condiciones de estado siguiente de la tabla de estados.
- 3)_ Se derivan las funciones de salida si hay condiciones de entrada en la tabla de estados.
- 4)_ Se simplifican las ecuaciones de entrada y las funciones de salida.
- 5)_ Se realiza el diagrama de lógica con Flip-Flops D y compuertas combinatorias, según lo que indican las ecuaciones de entrada y las funciones de salida.

_ Entonces a modo de ejemplo tenemos el siguiente diagrama de estados, en donde a las salidas o estados del multivibrador biestable las representamos como A y B, a la entrada con X y a la salida con Y:



_ Una vez analizado el diagrama lo que hacemos es obtener la tabla de estados que se obtiene directamente de los estados.

Tabla de estados del ejemplo de diseño

Estado presente		Entrada X	Estado siguiente		Salida Y
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

_ Las ecuaciones de entrada se obtienen a partir de los valores del estado siguiente, y la función de salida está dada por los valores binarios de Y :

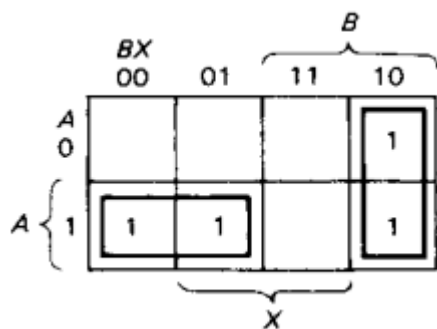
$$A(t + 1) = D_A(A, B, X) = \sum m(2, 4, 5, 6)$$

$$B(t + 1) = D_B(A, B, X) = \sum m(1, 3, 5, 6)$$

$$Y(A, B, X) = \sum m(1, 5)$$

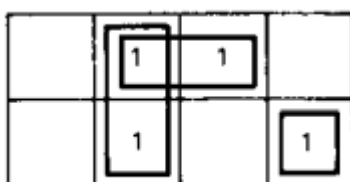
_ Ahora simplificamos las funciones anteriores:

1)_ $A(t + 1) = D_A(A, B, X) = \sum m(2, 4, 5, 6)$



$$D_A = A\bar{B} + B\bar{X}$$

2)_ $B(t + 1) = D_B(A, B, X) = \sum m(1, 3, 5, 6)$



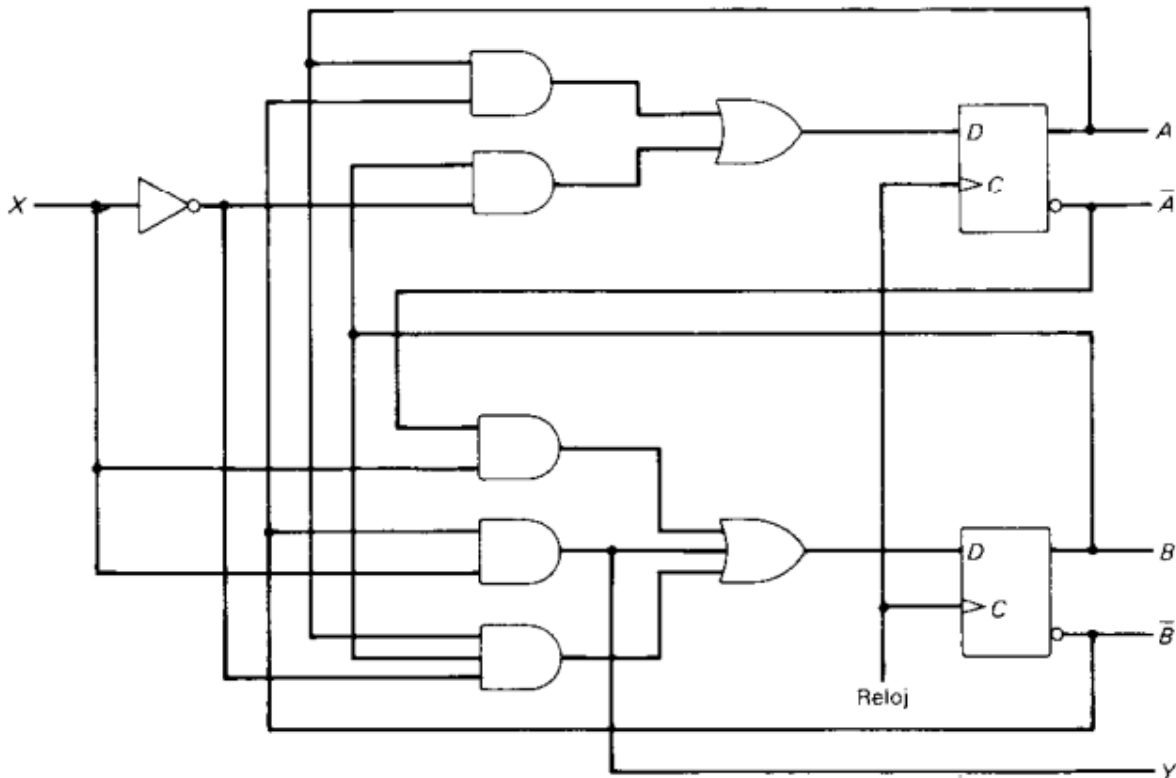
$$D_B = \bar{A}X + \bar{B}X + AB\bar{X}$$

3)_ $Y(A, B, X) = \sum m(1, 5)$

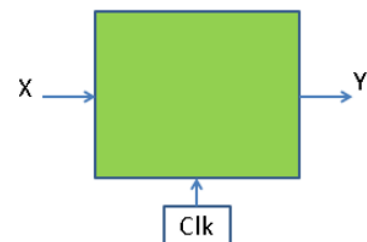
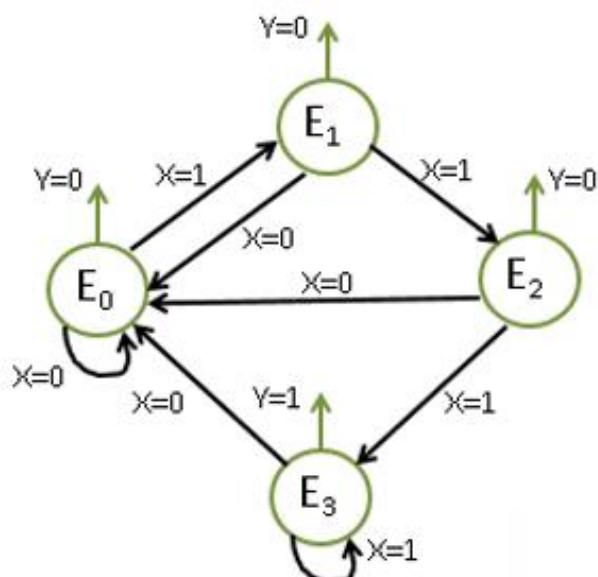
	1		
	1		

$$Y = \overline{B}X$$

_ Por último, realizamos el diagrama de lógica secuencial:



_ A modo de segundo ejemplo, supongamos que tenemos un sistema cuya salida Y se activa ante la presencia de tres unos consecutivos en la entrada X. Dicho sistema hace referencia a una caja negra que contiene información y de la cual se obtendrá el diagrama de estados.

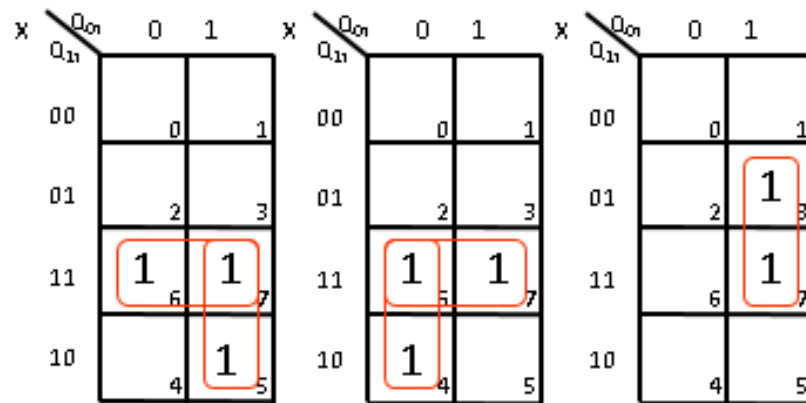


ESTADO	Q_1	Q_0
E0	0	0
E1	0	1
E2	1	0
E3	1	1

Procedemos a crear la tabla de verdad con la información anterior:

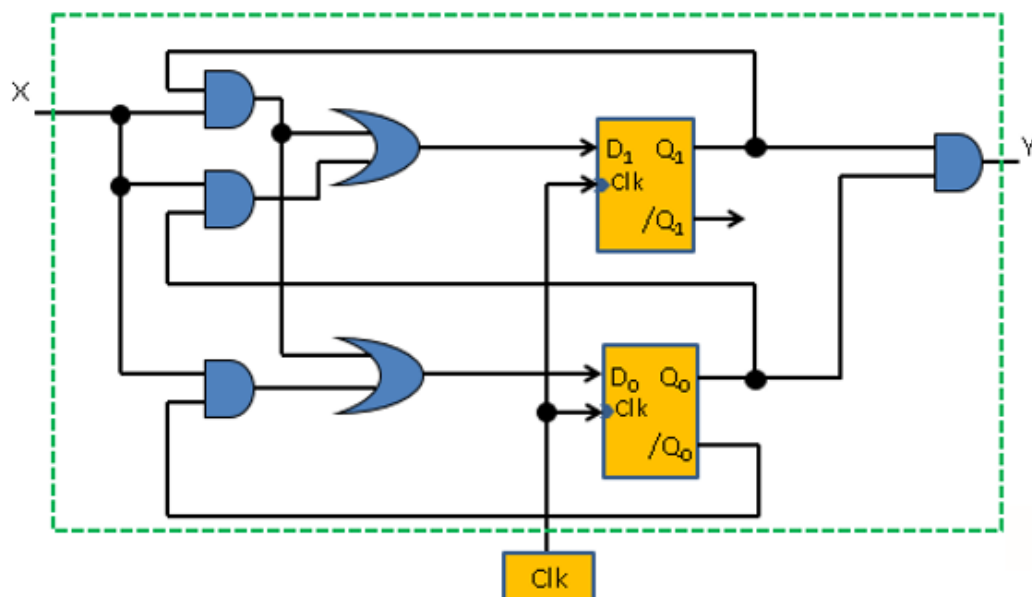
X	Q_{1t}	Q_{0t}	Q_{1t+1}	Q_{0t+1}	Y_t
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	1

_ Luego, una vez que determinamos las funciones, aplicamos Karnaugh para simplificar:



$$Q_{-1t+1} = X_t Q_{-1t} + X_t Q_{0t} \quad Q_{0t+1} = X_t Q_{-1t} + X_t / Q_{0t} \quad Y_t = Q_{-1t} \cdot Q_{0t}$$

_ Una vez que obtenemos los resultados realizamos la implementación:



Diseño con estados no usados

_ Hay ocasiones en los que un circuito secuencial utiliza un número menor de estados máximo posible, en donde los estados que no se usan, no se incluyen en la tabla de estados. Entonces cuando se simplifican las ecuaciones de entrada, los estados no usados se pueden tratar como condiciones “no importa”.

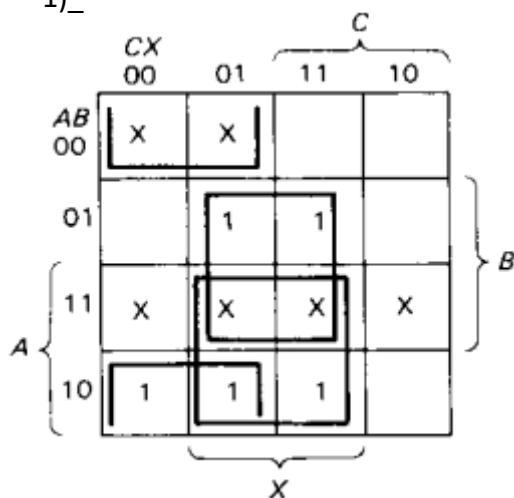
_ Analizamos la siguiente tabla de estados, en la que se definen tres multivibradores biestables A, B y C y una entrada X, pero no hay columna de salida. Como podemos observar hay tres estados no usados que no están incluidos en la tabla y son 000, 110 y 111, pero en este caso tenemos que contar los valores de X, por ende, los estados que no se usan son 0000, 0001, 1100, 1101, 1110 y 1111, ya que no aparecen en la tabla.

Tabla de estados del segundo ejemplo de diseño

Estado presente			Entrada	Estado siguiente		
A	B	C		A	B	C
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	1	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0

_ Por ende realizamos los mapas para obtener las tres ecuaciones de entrada:

1)_



$$D_A = AX + BX + \overline{BC}$$

2)_

x	x	1	
1			
x	x	x	x

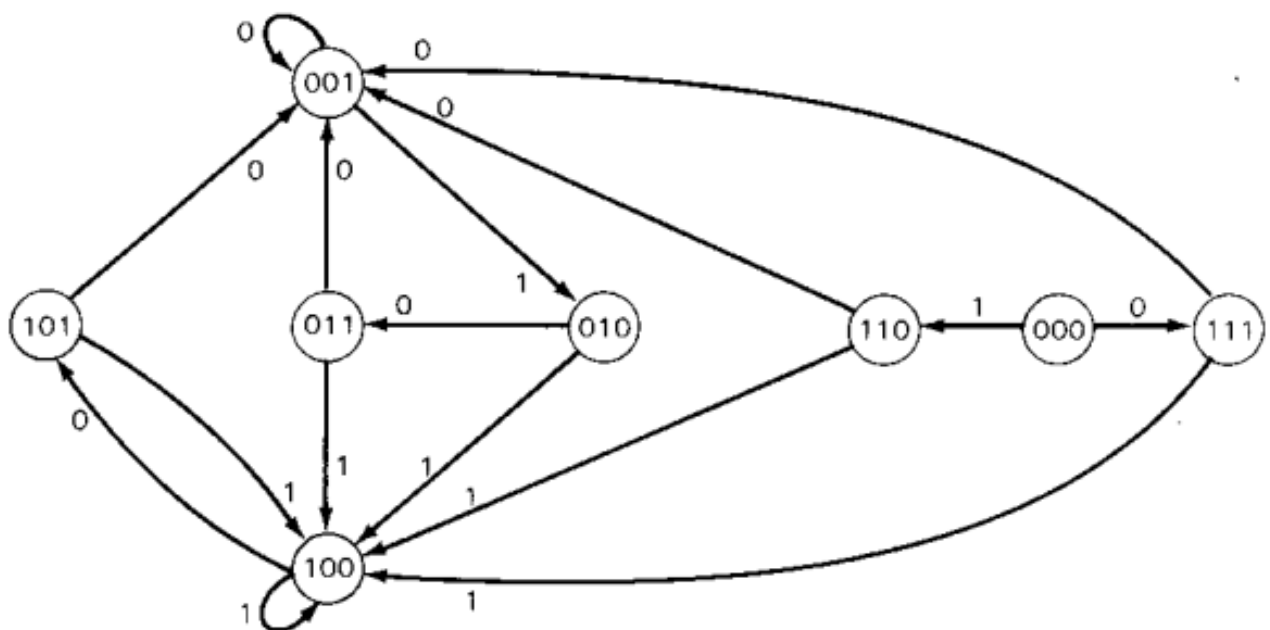
$$D_B = \overline{A}\overline{C}\overline{X} + \overline{A}\overline{B}X$$

3)_

x	x		1
1			1
x	x	x	x
1			1

$$D_C = \overline{X}$$

_ Por último, realizamos el diagrama de estados:



Diseño de una CPU

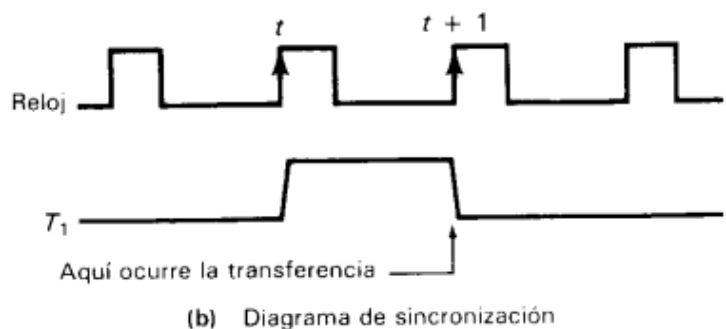
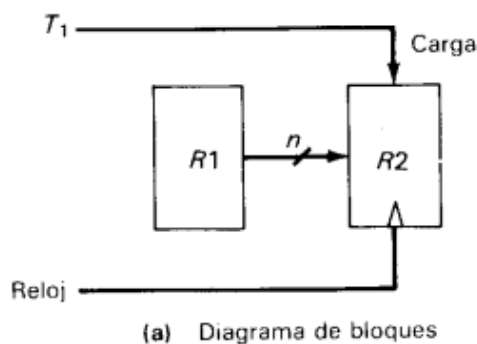
Transferencia de registros

_ Los registros de un sistema digital se designan por letras mayúsculas, y a veces seguidas de numerales, que denotan la función de un registro. Algunos son PC, IR, R1 y R2. La transferencia de información de un registro a otro se designa en forma simbólica por medio de un operador de reemplazo. Por ejemplo:

$$R2 \leftarrow R1$$

_ La instrucción anterior se refiere a una transferencia del contenido del registro R1 al registro R2, es decir, designa un reemplazo del contenido de R2 por el contenido de R1. El contenido de R1 no sufre alteración después de la transferencia. A continuación, vemos otra transferencia, pero con la siguiente condición:

$$\text{Si } (T_1 = 1) \text{ entonces } (R2 \leftarrow R1)$$



_ Entonces, observamos la siguiente tabla:

Simbolos básicos de transferencias de registros

Simbolo	Descripción	Ejemplos
Letras (y numerales)	Denota un registro	AR, R2
Paréntesis ()	Denota una parte de un registro	R2(0-7), R2(L)
Flecha -	Denota transferencia de información	R2 ← R1
Coma ,	Separa dos microoperaciones	R2 ← R1, R1 ← R2
Corchetes o paréntesis cuadrados []	Especifican una dirección de la memoria	DR ← M[AR]

Microoperaciones

_ Una microoperación es una operación elemental que se realiza con los datos almacenados en registros. El tipo de microoperaciones que se encuentran más frecuentemente en las computadoras digitales se clasifican en cuatro categorías:

- Las microoperaciones de transferencia de registros transfieren información binaria de un registro a otro.
- Las microoperaciones aritméticas efectúan operaciones aritméticas con números almacenados en registros.
- Las microoperaciones lógicas efectúan operaciones de manipulación de bits con datos no numéricos almacenados en registros.
- Las microoperaciones de corrimiento realizan operaciones de corrimiento del contenido de los registros.

Microoperaciones aritméticas

_ Las más básicas son la adición, sustracción, incremento, disminución y corrimiento. A continuación, observamos en la tabla:

Microoperaciones aritméticas

Designación simbólica	Descripción
$R0 \leftarrow R1 + R2$	Contenido de $R1$ más $R2$ transferido a $R0$
$R0 \leftarrow R1 - R2$	Contenido de $R1$ menos $R2$ transferido a $R0$
$R2 \leftarrow \overline{R2}$	Complemento del contenido de $R2$ (complemento a 1's)
$R2 \leftarrow \overline{R2} + 1$	Complemento a 2's del contenido de $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ más el complemento a 2's de $R2$ (sustracción)
$R1 \leftarrow R1 + 1$	Incremento del contenido de $R1$ (cuenta ascendente)
$R1 \leftarrow R1 - 1$	Decremento del contenido de $R1$ (cuenta descendente)

_ Las operaciones de multiplicación y división son válidas, pero no están en la tabla, ya que no son parte del conjunto básico de microoperaciones, por ende:

- Operación multiplicación: representada por el símbolo *.
- Operación división: representada por el símbolo /.

Microoperaciones lógicas

_ Observamos la siguiente tabla:

Microoperaciones lógicas

Designación simbólica	Descripción
$R \leftarrow \overline{R}$	Complementa todos los bits del registro R
$R0 \leftarrow R1 \wedge R2$	Microoperación AND lógica (pone a bits en ceros)
$R0 \leftarrow R1 \vee R2$	Microoperación OR lógica (inicia a bits)
$R0 \leftarrow R1 \oplus R2$	Microoperación XOR lógica (complementa a bits)

AND: se utiliza para anular a cero a un bit o un grupo en un registro. La relación es:

$$X \cdot 0 = 0$$

$$X \cdot 1 = X$$

_ En donde: $\beta_1 \wedge \beta_2$

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

OR: se utiliza para iniciar un grupo o un bit en un registro. La relación es:

$$X + 0 = X$$

$$X + 1 = 1$$

_ En donde: $\beta_1 \vee \beta_2$

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

XOR: se utiliza para complementar un grupo o un bit en un registro. La relación es:

$$X \oplus 0 = X$$

$$X \oplus 1 = \neg X$$

_ En donde: $\beta_1 \oplus \beta_2$

$$0 \oplus 0 = 0$$

$$0 \oplus 1 = 1$$

$$1 \oplus 0 = 1$$

$$1 \oplus 1 = 0$$

Corrimiento

_ El corrimiento se utiliza para transferir datos en serie. También se utilizan operaciones aritméticas de, lógicas y de control. Tenemos en cuenta que el contenido de un registro se puede correr a la derecha o a la izquierda.

_ En una operación de rotación, la salida en serie se conecta a la entrada en serie y los bits del registro giran sin que haya pérdida de información.

Corrimiento aritmético: es una microoperación que desplaza un número binario con signo a la izquierda (multiplicando un número binario con signo por 2) y a la derecha (dividiendo un número binario por 2). Estos dejan intacto el bit del signo an la multiplicación y la división.

_ Observamos la siguiente tabla:

Microoperaciones de corrimiento

Designación simbólica	Descripción
$R \leftarrow \text{shl } R$	Corrimiento a la izquierda del registro R
$R \leftarrow \text{shr } R$	Corrimiento a la derecha del registro R
$R \leftarrow \text{rol } R$	Rotación a la izquierda del registro R
$R \leftarrow \text{ror } R$	Rotación a la derecha del registro R
$R \leftarrow \text{asl } R$	Corrimiento aritmético a la izquierda de R
$R \leftarrow \text{asr } R$	Corrimiento aritmético a la derecha de R

Desbordamiento

_ Cuando se suman dos números de n dígitos cada uno y la suma ocupa $n+1$ dígitos, decimos que hay un desbordamiento. Esto sucede con números binarios o decimales tengan signo o no. El desbordamiento es un problema en las computadoras digitales porque la longitud de los registros es finita.

_ La detección del desbordamiento despues de la adición de dos números binarios depende de si los números se consideran con sin signo, entonces:

- Cuando se suman dos números sin signo, se detecta un desbordamiento desde el acarreo final en la posición más significativa.
- Cuando se suman cuando se suman dos números con signo, el bit del signo se trata como parte del número y el acarreo final no indica desbordamiento.

_ Un desbordamiento no puede ocurrir despues de una adición si un número es positivo y el otro es negativo, ya que esta adición va a producir un resultado menor al valor de los dos números originales. Ahora, un desbordamiento puede ocurrir cuando los números que se suman son ambos positivos o ambos negativos.

_ A continuación, vemos un ejemplo en donde sumamos dos números binarios que representan el +70 y el +80 que se almacenan en dos registros de 8 bits, pero tenemos que saber que el intervalo de números que puede alojar cada registro es +127 binario en el caso de los positivos y -128 binario en el caso de los negativos. Como vemos, la suma supera el valor binario de los positivos, por ende, excede la capacidad del registro de 8 bits.

acarreo: 0 1

+ 70	0 1000110
<u>+ 80</u>	<u>0 1010000</u>
+ 150	1 0010110

acarreo: 1 0

- 70	1 0111010
<u>- 80</u>	<u>1 0110000</u>
- 150	0 1101010

Suma binaria

_ A continuación analizamos la siguiente lógica de la adición:

$0 + 0 = 0 \rightarrow$ resultado 0, acarreo 0.

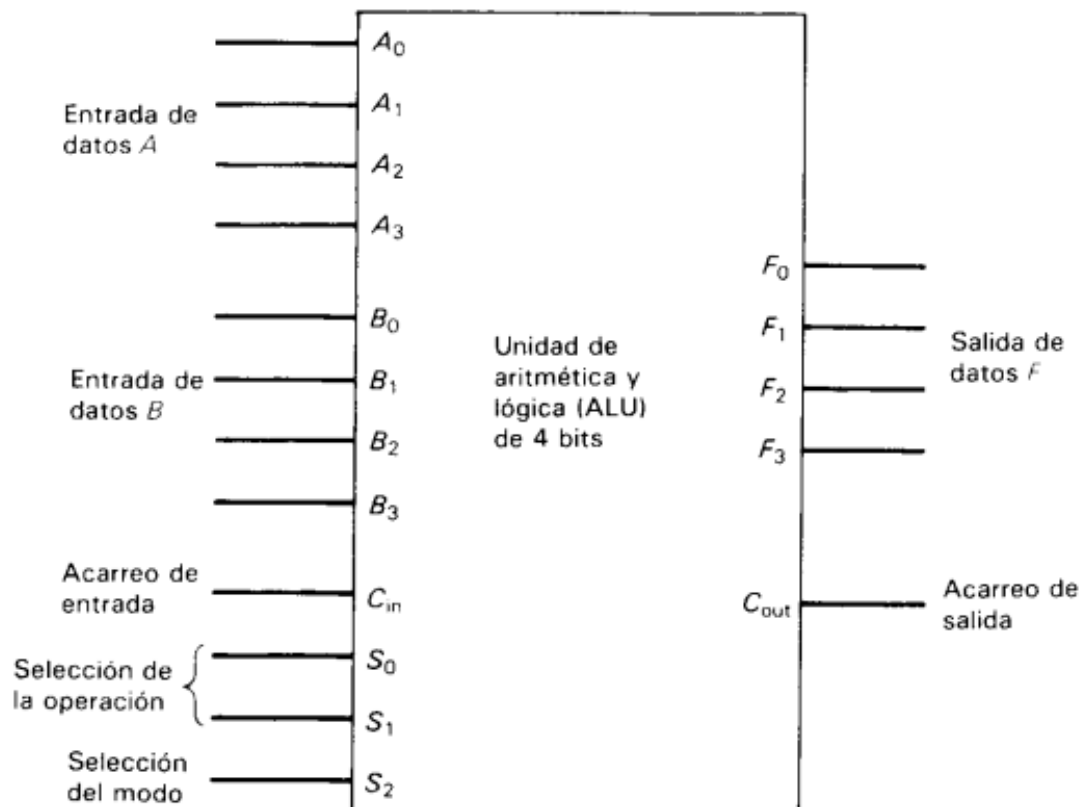
$0 + 1 = 1 \rightarrow$ resultado 1, acarreo 0.

$1 + 0 = 1 \rightarrow$ resultado 1, acarreo 0.

$1 + 1 = 10 \rightarrow$ resultado 0, acarreo 1.

Unidad aritmética lógica (ALU)

_ La ALU es un circuito combinatorio que realiza un conjunto de microoperaciones de aritmética y lógica básica. La ALU tiene un número de líneas de selección que sirven para elegir una operación determinada en la unidad. K variables de selección pueden especificar hasta 2^k operaciones distintas. A continuación, vemos un diagrama de bloque de una ALU de 4 bits en donde tenemos las cuatro entradas de datos de A que se combinan con las cuatro entradas de B para generar una operación en las salidas de F, luego tenemos la entrada de selección del modo S_2 que distingue entre operaciones aritméticas y lógicas, y tenemos aparte las dos entradas de selección de funciones S_0 y S_1 que especifican la operación aritmética o lógica en particular que se va a generar. El acarreo de entrada C_{in} se utiliza a menudo como una cuarta variable de selección para operaciones aritméticas y de esta forma es posible duplicar el número de operaciones aritméticas de cuatro a ocho.



_ A continuación, vemos la tabla de funciones de la ALU en donde cada operación se selecciona a través de las variables S_2 , S_1 , S_0 y C_{in} . En donde el acarreo de entrada se utiliza para seleccionar solo una operación aritmética.

Tabla de funciones de la ALU

<u>Selección de la operación</u>				Operación	Función
S_2	S_1	S_0	C_{in}		
0	0	0	0	$F = A$	Transferencia de A
0	0	0	1	$F = A + 1$	Incremento de A
0	0	1	0	$F = A + B$	Adición
0	0	1	1	$F = A + B + 1$	Adición con acarreo
0	1	0	0	$F = A + \overline{B}$	A más complemento a 1's de B
0	1	0	1	$F = A + \overline{B} + 1$	Sustracción
0	1	1	0	$F = A - 1$	Decremento de A
0	1	1	1	$F = A$	Transferencia de A
1	0	0	0	$F = A \wedge B$	AND
1	0	1	0	$F = A \vee B$	OR
1	1	0	0	$F = A \oplus B$	XOR
1	1	1	0	$F = \overline{A}$	Complemento de A

Palabra de control

_ Hay 16 entradas de selección binarias en la unidad y su valor combinado especifica una palabra de control. La palabra de control de 16 bits se define de la siguiente manera:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A			B			D			F				H		

_ La palabra de control, como podemos observar, consta de cinco partes llamadas campos, donde cada campo está designado por una letra. Cuatro de los campos contienen tres bits cada uno y un campo tiene cuatro bits.

- Los tres bits de A seleccionan un registro fuente para la entrada de la ALU.
- Los tres bits de B seleccionan un registro para la segunda entrada de la ALU.
- Los tres bits de D seleccionan un registro destino.
- Los cuatro bits de F seleccionan una de 12 operaciones en la ALU.
- Los tres bits de H seleccionan el tipo de corrimiento en la unidad de corrimiento.

_ La palabra de control de 16 bits, cuando se aplica a las entradas de selección, especifica una microoperación en particular. Las funciones de todas las variables de especificación se especifican en la siguiente tabla:

Codificación de la palabra de control de la unidad procesadora

Código binario	Función de campos de selección					
	A	B	D	F con $C_{in} = 0$	F con $C_{in} = 1$	H
000	Entrada	Entrada	Ninguno	$F = A$	$F = A + 1$	No hay corrimiento
001	R1	R1	R1	$F = A + \overline{B}$	$F = A + B + 1$	SHL
010	R2	R2	R2	$F = A + \overline{B}$	$F = A - B$	SHR
011	R3	R3	R3	$F = A - 1$	$F = A$	Bus = 0
100	R4	R4	R4	$F = A \wedge B$	—	—
101	R5	R5	R5	$F = A \vee B$	—	ROL
110	R6	R6	R6	$F = \overline{A} \oplus B$	—	ROR
111	R7	R7	R7	$F = \overline{A}$	—	—

ARM Assembly

Introducción

_ El lenguaje Assembly (ensamblador) es la representación legible por el hombre del lenguaje nativo de la computadora. Cada instrucción de lenguaje ensamblador especifica tanto la operación a realizar como los operandos sobre los que operar. Introducimos instrucciones aritméticas simples y mostramos cómo se escriben estas operaciones en lenguaje ensamblador. Luego definimos los operandos de la instrucción ARM, es decir, registros, memoria y constantes.

Instrucciones

Adición: la operación más común que realizan las computadoras es la adición. El siguiente ejemplo muestra el código para sumar las variables b y c, y escribir el resultado en a. El código que se muestra arriba es en un lenguaje de alto nivel (usando la sintaxis de C, C++ y Java) y luego se reescribe abajo en el lenguaje ensamblador ARM.

- High-Level Code:
a = b + c;
- ARM Assembly Code:
ADD a, b, c

_ La primera parte de la instrucción de montaje, ADD, se llama mnemotécnica e indica qué operación se va a realizar. La operación se realiza en b y c, los operandos fuente (source operands), y el resultado se escribe en a, el operando de destino (destination operand).

Sustracción: en el caso de la resta tenemos el siguiente ejemplo, que muestra que la resta es similar a la suma. El formato de la instrucción es el mismo que el de la instrucción ADD excepto por la especificación de operación, SUB. Las variables a, b, y c son todas operandos.

- High-Level Code:
a = b - c;
- ARM Assembly Code:
SUB a, b, c

_ Las instrucciones con un número consistente de operandos, en este caso, dos fuentes y un destino, son más fáciles de codificar y manejar en el hardware. Un código de alto nivel más complejo se traduce en múltiples instrucciones ARM, como se muestra en el siguiente ejemplo:

- High-Level Code:
a = b + c - d; // single-line comment /* multiple-line comment */
- ARM Assembly Cod:
ADD t, b, c ; t = b + c
SUB a, t, d ; a = t - d

_ En el lenguaje ensamblador ARM, sólo se usan los comentarios de una línea., en donde estos comienzan con un punto y coma (;) y continúan hasta el final de la línea. En este programa de ARM se usa una variable temporal t para almacenar el resultado intermedio.

Operandos: registros, memoria y constantes

_ Una instrucción opera en los operandos (operands). Pero las computadoras operan con 1 y 0, no con nombres de variables. Las instrucciones necesitan una ubicación física desde la cual recuperar los datos binarios. Los operandos pueden estar almacenados en registros o en memoria, o pueden ser constantes almacenadas en la propia instrucción. Los ordenadores utilizan varios lugares para mantener operandos con el fin de optimizar la velocidad y la capacidad de datos. Se accede rápidamente a los operandos almacenados como constantes o en registros, pero sólo contienen una pequeña cantidad de datos. Se debe acceder a los datos adicionales desde la memoria, que es grande pero lenta. ARM (anterior a ARMv8) se llama arquitectura de 32 bits porque opera con datos de 32 bits.

Registros

_ Las instrucciones necesitan acceder rápidamente a los operandos para que puedan correr rápido. Pero los operandos almacenados en la memoria tardan mucho tiempo en recuperarse. Por lo tanto, la mayoría de las arquitecturas especifican un pequeño número de registros que contienen operandos de uso común. La arquitectura ARM utiliza 16 registros, llamados conjunto de registros o archivo de registros (register set o register file). Cuantos menos registros, más rápido se puede acceder a ellos. Un archivo de registro se construye típicamente a partir de una pequeña matriz SRAM

_ Retomamos nuevamente el ejemplo de la suma, en donde esta vez especificamos a que registro pertenece cada variable:

- High-Level Code:
 $a = b + c;$
- ARM Assembly Code:
; R0= a, R1 = b, R2 = c
ADD R0, R1, R2 ; a = b + c

_ Este ejemplo de código muestra la instrucción ADD con operandos de registro. Los nombres de registro ARM están precedidos por la letra 'R'. Las variables a, b y c se colocan arbitrariamente en R0, R1 y R2. El nombre R1 se pronuncia "registro 1" o "R1" o "registro R1". La instrucción suma los valores de 32 bits contenidos en R1 (b) y R2 (c) y escribe el resultado de 32 bits en R0 (a).

Registro temporal: como vimos anteriormente, el siguiente ejemplo muestra el código en ARM usando un registro, el R4, para almacenar el cálculo intermedio de $b + c$. Y a esto se le denomina registro temporal, es decir, como si fuese una variable auxiliar.

- High-Level Code:
 $a = b + c - d;$
- ARM Assembly Code:
; R0= a, R1 = b, R2 = c, R3 = d; R4 = t
ADD R4, R1, R2 ; t = b + c
SUB R0, R4, R3 ; a = t - d

Conjunto de registros

_ La tabla enumera el nombre y el uso de cada uno de los 16 registros de ARM. R0 a R12 se usan para almacenar variables; R0 a R3 también tienen usos especiales durante las llamadas a procedimientos. R13, R14 y R15 también se llaman SP, LR y PC.

Nombre	Uso
R0	Argument / return value / temporary variable
R1 – R3	Argument / temporary variables
R4 – R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Ejercicio: traduzca el siguiente código de alto nivel al lenguaje ensamblador ARM.
Supongamos que las variables a, b y c están en los registros R0, R1 y R2, y f, g, h, i y j están en los registros R3, R4, R5, R6 y R7.

a = b - c;
f = (g + h) - (i + j);

_ La solución es:

```
; ARM código ensamblador
; R0= a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j
SUB R0, R1, R2      ; a = b - c
ADD R8, R4, R5      ; R8 = g + h
ADD R9, R6, R7      ; R9 = i + j
SUB R3, R8, R9      ; f = (g + h) - (i + j)
```

Constantes/Inmediatos

_ Además de las operaciones de registro, las instrucciones de ARM pueden utilizar operandos constantes o inmediatos (immediate operands). Estas constantes se denominan inmediatas, porque sus valores están disponibles inmediatamente en la instrucción y no requieren un registro o acceso a la memoria. En código ensamblador, el inmediato está precedido por el símbolo # y puede escribirse en decimal o hexadecimal. Las constantes hexadecimales en el lenguaje ensamblador ARM comienzan con 0x. Los inmediatos son números sin signo de 8 a 12 bits.

Instrucción de movimiento (MOV): es una forma útil de inicializar o asignar los valores de los registros. MOV también puede tomar un operando de fuente de registro. Por ejemplo, el siguiente código copia el contenido del registro R7 en R1:

```
MOV R1, R7
```

_ A continuación, tenemos un ejemplo de la operación con constantes:

- High-Level Code:
a = a + 4;
b = a - 12;
- ARM Assembly Code:
; R7= a, R8 = b
ADD R7, R7, #4 ; a = a + 4
SUB R8, R7, #0xC ; b = a - 12

_ En este caso vemos un ejemplo en el que inicializamos variables inmediatas o con constantes:

- High-Level Code:
i = 0;
x = 4080;

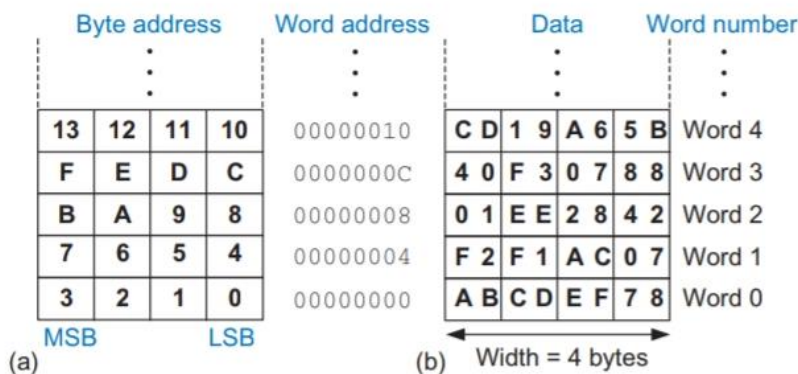
- ARM Assembly Code:
; R4= i, R5 = x
MOV R4, #0 ; i = 0
MOV R5, #0xFF0 ; x = 4080

Memoria

_ Si los registros fueran el único espacio de almacenamiento para los operandos, nos limitaríamos a programas simples con no más de 15 variables. Sin embargo, los datos también pueden ser almacenados en la memoria. Mientras que el archivo de registro es pequeño y rápido, la memoria es más grande y lenta. Por esta razón, las variables de uso frecuente se mantienen en registros. En la arquitectura ARM, las instrucciones operan exclusivamente en registros, por lo que los datos almacenados en la memoria deben ser movidos a un registro antes de que puedan ser procesados.

_ Utilizando una combinación de memoria y registros, un programa puede acceder a una gran cantidad de datos con bastante rapidez. La arquitectura ARM utiliza direcciones de memoria de 32 bits y palabras de datos de 32 bits. El ARM utiliza una memoria direccionable por bytes (byte-addressable). Es decir, cada byte de la memoria tiene una dirección única.

_ Una palabra de 32 bits consta de cuatro bytes de 8 bits, por lo que cada dirección de palabra es un múltiplo de 4. El byte más significativo (MSB) está a la izquierda y el menos significativo (LSB) a la derecha. Tanto la dirección de la palabra de 32 bits como el valor de los datos están dados en hexadecimal. Por ejemplo, la palabra de datos 0xF2F1AC07 se almacena en la dirección de memoria 4. Por convención, la memoria se dibuja con las direcciones de memoria baja hacia abajo y las direcciones de memoria alta hacia arriba.



Instrucción de registro de carga (LDR): ARM proporciona esta instrucción para leer una palabra de datos de la memoria en un registro. La instrucción LDR especifica la dirección de la memoria utilizando un registro base (base register) (R5) y un offset (8). Recordemos que cada palabra de datos es de 4 bytes, por lo que la palabra número 1 está en la dirección 4, la palabra número 2 está en la dirección 8, y así sucesivamente. La dirección de la palabra es cuatro veces el número de la palabra misma. La dirección de memoria se forma añadiendo el contenido del registro base (R5) y el offset. A continuación, vemos un ejemplo de lectura de la memoria:

- High-Level Code:
a = mem[2];
- ARM Assembly Code:
; R7= a
MOV R5, #0 ; base address = 0
LDR R7, [R5, #8] ; R7 <= data at memory address (R5+8)

_ En este ejemplo se carga la palabra de memoria 2 en un (R7). Después de que se ejecuta la instrucción de registro de carga (LDR), R7 mantiene el valor 0x01EE2842, que es el valor de los datos almacenados en la dirección de memoria 8.

Instrucción de registro de almacenamiento (STR): ARM utiliza esta instrucción para escribir una palabra de datos de un registro en la memoria. A continuación, tenemos un ejemplo de escritura en la memoria:

- High-Level Code:
mem[5] = 42;
- ARM Assembly Code:
MOV R1, #0 ; base address = 0
MOV R9, #42
STR R9, [R1, #0x14] ; value stored at memory address (R1+20) = 42

_ El código anterior escribe el valor 42 del registro R9 en la palabra de memoria 5.

Instrucciones para el procesamiento de datos

_ La arquitectura ARM define una variedad de instrucciones de procesamiento de datos (a menudo llamadas instrucciones lógicas y aritméticas en otras arquitecturas).

Instrucciones lógicas:

_ Las operaciones lógicas de ARM incluyen AND, ORR (OR), EOR (XOR), y BIC (bit clear). Cada una de ellas opera bit a bit en dos fuentes y escribe el resultado en un registro de destino. La primera fuente es siempre un registro y la segunda fuente es un registro inmediato u otro registro. Otra operación lógica, MVN (MoVe y Not), actúa de forma poco precisa en la segunda fuente (un inmediato o registro) y escribe el resultado en el registro de destino.

Instrucción bit clear (BIC): es útil para enmascarar bits, es decir, forzar a los bits no deseados a 0. BIC R6, R1, R2 calcula R1 y no R2. En otras palabras, BIC borra los bits que se afirman en R2. En este caso, los dos bytes superiores de R1 se limpian o enmascaran (masked), y los dos bytes inferiores desenmascarados de R1, 0xF1B7, se colocan en R6. Cualquier subconjunto de bits de registro puede ser enmascarado.

Instrucción ORR: es útil para combinar los campos de bits de dos registros. Por ejemplo:

0x347A0000 ORR 0x000072FC = 0x347A72FC.

_ En la siguiente imagen podemos ver ejemplos de estas operaciones sobre los dos valores de la fuente 0x46A1F1B7 y 0xFFFF0000. Se ven los valores almacenados en el registro de destino después de que la instrucción se ejecute.

		Source registers			
R1		0100 0110	1010 0001	1111 0001	1011 0111
R2		1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

Instrucciones de desplazamiento:

_ Las instrucciones de desplazamiento (Shift Instructions), cambian el valor de un registro a la izquierda o a la derecha, dejando caer trozos del final. La instrucción de giro rota el valor en un registro a la derecha hasta 31 bits. Nos referimos a ambos, desplazar y rotar genéricamente como operaciones de desplazamiento (shift operations).

Operaciones de desplazamiento: estas son:

- LSL: desplazamiento lógico a la izquierda.
- LSR: desplazamiento lógico a la derecha.
- ASR: desplazamiento aritmético a la derecha.
- ROR: rotación a la derecha.
- No hay instrucción ROL porque la rotación a la izquierda se puede realizar con una rotación a la derecha por una cantidad complementaria.

_ Los desplazamientos a la izquierda (left shift) siempre llenan los bits menos significativos con ceros. Sin embargo, los desplazamientos hacia la derecha pueden ser lógicos (el desplazamiento de 0 en los bits más significativos) o aritméticos (el bit del signo se desplaza en los bits más significativos). La cantidad por la que se puede desplazar puede ser un inmediato o un registro.

_ Como podemos observar en la siguiente imagen, R5 se desplaza por la cantidad inmediata, y el resultado se coloca en el registro de destino.

		Source register			
R5		1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

_ Desplazar un valor a la izquierda por N equivale a multiplicarlo por 2^N . Del mismo modo, desplazar aritméticamente un valor a la derecha por N equivale a dividirlo por 2^N .

_ Los desplazamientos lógicos también se utilizan para extraer o ensamblar campos de bits. La siguiente imagen muestra el código de ensamblaje y los valores de registro resultantes para las operaciones de desplazamiento en las que la cantidad de desplazamiento se mantiene en un registro, R6.

		Source registers			
R8		0000 1000	0001 1100	0001 0110	1110 0111
R6		0000 0000	0000 0000	0000 0000	0001 0100
Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

_ Esta instrucción utiliza el modo de direccionamiento de registro desplazado (register-shifted register), en el que un registro (R8) se desplaza en la cantidad (20) que se mantiene en un segundo registro (R6).

Instrucciones de multiplicación:

_ La multiplicación es algo diferente de otras operaciones aritméticas. Multiplicar dos números de 32 bits produce un producto de 64 bits. La arquitectura del ARM proporciona instrucciones de multiplicación (Multiply Instructions) que dan como resultado un producto de 32 o 64 bits.

Instrucción multiply (MUL): multiplica dos números de 32 bits y produce un resultado de 32 bits. Esta instrucción es útil para multiplicar números pequeños cuyo resultado cabe en 32 bits. El siguiente código multiplica los valores en R2 y R3 y coloca los bits menos significativos del producto en R1; los 32 bits más significativos del producto se descartan.

MUL R1, R2, R3

Instrucción UMULL (sin signo, multiplica largo) y SMULL (con signo, multiplica largo): ambas multiplican dos números de 32 bits y producen un producto de 64 bits. Por ejemplo, el siguiente código realiza una multiplicación sin signo de R3 y R4. Los 32 bits menos significativos del producto se colocan en R1 y los 32 bits más significativos se colocan en R2.

UMULL R1, R2, R3, R4

_ Cada una de estas instrucciones también tiene una variante de multiplicación, MLA, SMLAL, y UMLAL, que añade el producto a una suma de 32 ó 64 bits. Estas instrucciones pueden mejorar el rendimiento matemático en aplicaciones como la

multiplicación de matrices y el procesamiento de señales consistentes en multiplicaciones y sumas repetidas.

Banderas de condición:

_ Las instrucciones ARM establecen opcionalmente banderas de condición (condition flags) basadas en si el resultado es negativo, cero, etc. Las instrucciones subsiguientes se ejecutan condicionalmente, dependiendo del estado de esos indicadores de condición.

Indicadores de condición ARM: también llamados indicadores de estado (status flags), son negative (N), zero (Z), carry (C) y overflow (V), como se indica en la tabla. Estos banderines son fijados por la ALU y se mantienen en los 4 bits superiores del registro de estado del programa actual (Current Program Status Register - CPSR) de 32 bits.



Flag	Nombre	Descripción
N	Negative	El resultado de la instrucción es negativo, es decir, el bit 31 del resultado es 1.
Z	Zero	El resultado de la instrucción es cero.
C	Carry	La instrucción hace que se lleve a cabo.
V	oVerflow	La instrucción causa un desbordamiento.

Instrucción comapare (CMP): la forma más común de establecer los bits de estado es con esta instrucción, que resta el segundo operando de la fuente del primero y establece los indicadores de condición basados en el resultado. Por ejemplo, si los números son iguales, el resultado será cero y se establece el indicador Z. Si el primer número es un valor sin signo que es mayor o igual que el segundo, la resta producirá una ejecución y se fija el indicador C.

_ Las instrucciones posteriores pueden ejecutarse condicionalmente dependiendo del estado de las banderas. La instrucción mnemotécnica es seguida por una condición mnemotécnica que indica cuándo ejecutar.

_ En la siguiente tabla se enumeran el campo de condición de 4 bits (cond), el mnemotécnico de condición, el nombre y el estado de los indicadores de condición que dan lugar a la ejecución de la instrucción (CondEx).

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\bar{Z}
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	\bar{C}
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	\bar{N}
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	\bar{V}
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	Z OR \bar{C}
1010	GE	Signed greater than or equal	$\bar{N} \oplus \bar{V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\bar{N} \oplus \bar{V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Ejemplo: supongamos que un programa realiza las siguientes operaciones, por un lado, la comparación fija el indicador Z si R4 y R5 son iguales, y el ADDEQ se ejecuta sólo si se fija el indicador Z.

```
CMP R4, R5
ADDEQ R1, R2, R3
```

_ Otras instrucciones de procesamiento de datos fijarán las banderas de condición cuando la instrucción mnemotécnica sea seguida de una S.

Ejemplo: el siguiente código restará R7 de R3, pondrá el resultado en R2, y pondrá las banderas de condición. Todas las instrucciones de procesamiento de datos afectarán a los indicadores N y Z en función de si el resultado es cero o tiene el conjunto de bits más significativo. ADDS y SUBS también influyen en V y C, y los desplazamientos influyen en C.

```
SUBS R2, R3, R7
```

Ejecución condicional

_ El siguiente código muestra instrucciones que se ejecutan condicionalmente. La primera instrucción, CMP R2, R3, se ejecuta incondicionalmente y establece las banderas de condición. Allí las instrucciones de mantenimiento se ejecutan condicionalmente, dependiendo de los valores de las banderas de condición. Supongamos que R2 y R3 contienen los valores 0x80000000 y 0x00000001. La comparación calcula $R2 - R3 = 0x80000000 - 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF$ con una ejecución (C=1). Las fuentes tenían signos opuestos y el signo del resultado difiere del signo de la primera fuente, por lo que el resultado se desborda (V=1). Las banderas restantes (N y Z) son 0. ANDHS se ejecuta porque C=1. EORLT se ejecuta porque N es 0 y V es 1. Intuitivamente, ANDHS y EORLT se ejecutan porque $R2 \geq R3$ (unsigned) y $R2 < R3$ (signed), respectivamente. ADDEQ y ORRMI no se ejecutan porque el resultado de R2-R3 no es cero (es decir, $R2 \neq R3$) o negativo.

- ARM Assembly Code:
CMP R2, R3
ADDEQ R4, R5, #78
ANDHS R7, R8, R9
ORRMI R10, R11, R12
EORLT R12, R7, R10

Branching

_ Una computadora realiza diferentes tareas dependiendo de la entrada. Por ejemplo, declaraciones if/else, switch/case, y los bucles, en donde todos ejecutan condicionalmente el código dependiendo de alguna prueba. Una forma de tomar decisiones es utilizar la ejecución condicional para ignorar ciertas instrucciones. Esto funciona bien para declaraciones simples si se ignora un pequeño número de instrucciones, pero es un desperdicio para declaraciones con muchas instrucciones en el cuerpo, y es insuficiente para manejar bucles. Así, ARM y la mayoría de las otras arquitecturas usan Branch Instructions para saltar sobre secciones de código o repetir código.

_ Un programa normalmente se ejecuta en secuencia, con el contador de programa (PC) incrementándose en 4 después de cada instrucción para apuntar a la siguiente instrucción, (recordar que las instrucciones tienen 4 bytes de longitud y ARM es una arquitectura de dirección inversa).

_ Las instrucciones Branch sirven para cambiar el contador de programa. Los branch también se llaman jump en algunas arquitecturas. ARM incluye dos tipos de ramas o branches:

- Branch simple (B)
- Branch and Link (BL): se usa para llamadas a funciones.

_ Como otras instrucciones ARM, los Branch pueden ser:

Incondicionales: el siguiente ejemplo de código muestra una ramificación (branching) incondicional usando la instrucción de branch B. Cuando el código llega a la instrucción B TARGET, el branch es tomado, es decir, la siguiente instrucción ejecutada es la instrucción SUB justo después de la etiqueta llamada TARGET.

- ARM Assembly Code:
ADD R1, R2, #17 ; R1 = R2 + 17
B TARGET ; branch to TARGET
ORR R1, R1, R3 ; not executed
AND R3, R1, #0xFF ; not executed
TARGET SUB R1, R1, #78 ; R1 = R1 - 78

Condicionales: las Branch Instructions pueden ejecutarse condicionalmente usándolas nemotécnicas de condición de la tabla anterior. El siguiente ejemplo de código ilustra el uso de BEQ (combina las instrucciones branch B y EQ de equal), en donde la ramificación es dependiente de la igualdad ($Z=1$). Cuando el código llega a la instrucción BEQ, el indicador de condición Z es 0 (es decir, $R0 \neq R1$), por lo que la rama no se toma, es decir, la siguiente instrucción que se ejecuta es la instrucción ORR.

- ARM Assembly Code:
MOV R0, #4 ; R0 = 4
ADD R1, R0, R0 ; R1 = R0 + R0 = 8
CMP R0, R1 ; set flags based on $R0-R1 = -4$.
NZCV = 1000 BEQ THERE ; branch not taken ($Z \neq 1$)
ORR R1, R1, #1 ; R1 = R1 OR 1 = 9
THERE
ADD R1, R1, #78 ; R1 = R1 + 78 = 87

Etiquetas (labels): el código de ensamblaje utiliza etiquetas para indicar las ubicaciones de las instrucciones en el programa. Estas no pueden ser palabras reservadas, como nemotécnicas de instrucciones. También, el compilador de ARM exige que las etiquetas no deben tener sangría, y las instrucciones deben ir precedidas de un espacio en blanco.

Declaraciones condicionales

_ En los casos de if, if/else, y switch/case, estas son declaraciones condicionales que se utilizan comúnmente en los lenguajes de alto nivel. Cada una de ellas ejecuta condicionalmente un bloque de código que consiste en una o más declaraciones.

Condicional if: una declaración if ejecuta un bloque de código sólo cuando se cumple una condición. A continuación, vemos la analogía de la traducción de una declaración if en código de ensamblaje ARM, en donde en el código de alto nivel simplemente se compara que, si las manzanas son iguales a las naranjas, se realiza como consecuencia la operación que vemos dentro de la condición, de lo contrario se realiza la operación que esta por fuera:

- High-Level Code:

```
if (apples == oranges)
    f = i + 1;

f = f - i;
```
- ARM Assembly Code:

```
; R0= apples, R1 = oranges, R2 = f, R3 = i
CMP R0, R1          ; apples == oranges ?
BNE L1              ; if not equal, skip if block
ADD R2, R3, #1       ; if block: f = i + 1
L1
SUB R2, R2, R3       ; f = f - i
```

_ El código Assembly de la declaración if prueba la condición opuesta a la del código de alto nivel. Ya que, en el ejemplo anterior, el código de alto nivel prueba para manzanas que sean iguales a naranjas, y el código de ARM prueba para las manzanas distintas de naranjas usando BNE para saltar el bloque if si la condición no se cumple. En caso contrario, en donde las manzanas son iguales a naranjas, no se toma el branch, y se ejecuta el bloque if. Escribiendo el código anterior de manera más compacta seria:

- ARM Assembly Code:

```
CMP R0, R1          ; apples == oranges ?
ADDEQ R2, R3, #1     ; f = i + 1 on equality (i.e.. Z = 1)
SUB R2, R2, R3       ; f = f - i
```

_ Esta solución con ejecución condicional es más corta y también más rápida porque implica una instrucción menos.

Condicional if/else: las declaraciones if/else ejecutan uno de los dos bloques de código dependiendo de una condición. Cuando se cumple la condición de la declaración if, se ejecuta el bloque if, de lo contrario, se ejecuta el bloque else. A continuación, vemos la analogía de la traducción de una declaración if/else en código de ensamblaje ARM:

- High-Level Code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```
- ARM Assembly Code:

```
; R0= apples, R1 = oranges, R2 = f, R3 = i
CMP R0, R1          ; apples == oranges?
BNE L1              ; if not equal, skip if block
ADD R2, R3, #1       ; if block: f = i + 1
B L2                ; skip else block
L1
SUB R2, R2, R3       ; else block: f = f - i
L2
```

_ En este caso, el código de alto nivel prueba para manzanas que sean iguales a naranjas, y el código de ensamblaje prueba para manzanas distintas de naranjas. Entonces, si esa condición opuesta es verdadera, BNE se salta el bloque if y ejecuta el bloque else. De lo contrario, el bloque if se ejecuta y termina con una rama incondicional (B) más allá del bloque else.

_ De nuevo, porque cualquier instrucción puede ejecutarse condicionalmente y porque las instrucciones dentro del bloque if no cambian las banderas de condición, este puede comprimirse como:

- ARM Assembly Code:
CMP R0, R1 ; apples == oranges?
ADDEQ R2, R3, #1 ; f = i + 1 on equality (i.e., Z= 1)
SUBNE R2, R2, R3 ; f = f - i on not equal (i.e., Z= 0)

Condicional switch/case: estas declaraciones ejecutan uno de varios bloques de código dependiendo de las condiciones. Si no se cumplen las condiciones, se ejecuta el bloque por defecto. Una declaración de caso equivale a una serie de declaraciones anidadas if/else.

_ El siguiente ejemplo de código muestra dos fragmentos de código de alto nivel con la misma funcionalidad, es decir, ambos calculan si dispensar 20, 50 o 100 dólares de un cajero automático dependiendo del botón pulsado. Y luego vemos la analogía en ARM:

- High-Level Code:
switch (button){
 case 1: amt = 20; break;
 case 2: amt = 50; break;
 case 3: amt = 100; break;
 default: amt = 0;
}
- ARM Assembly Code:
; R0= button, R1 = amt
CMP R0, #1 ; is button 1 ?
MOVEQ R1, #20 ; amt = 20 if button is 1
BEQ DONE ; break

CMP R0, #2 ; is button 2 ?
MOVEQ R1, #50 ; amt = 50 if button is 2
BEQ DONE ; break

CMP R0, #3 ; is button 3 ?
MOVEQ R1, #100 ; amt = 100 if button is 3
BEQ DONE ; break
MOV R1, #0 ; default amt = 0
DONE

_ A continuación, vemos una equivalencia del ejemplo anterior en alto nivel, pero con la sentencia if/else:

- High-Level Code:
/* equivalent function using if/else statements */
if (button == 1)
 amt = 20;
else if (button == 2)
 amt = 50;
else if (button == 3)
 amt = 100;
else
 amt = 0;

Bucles

_ Los bucles ejecutan repetidamente un bloque de código dependiendo de una condición. Los bucles while y for, son construcciones de bucle comunes utilizadas por los lenguajes de alto nivel. Los bucles son especialmente útiles para acceder a grandes cantidades de datos similares almacenados en la memoria.

Ciclo While: estos ejecutan repetidamente un bloque de código mientras no se cumpla o no una condición. A continuación, vemos un ejemplo de la analogía. El siguiente bucle while, en el código, determina el valor de x de tal manera que $2^x = 128$, en donde se va a ejecutar siete veces, hasta que pow = 128 y ahí se frena el ciclo.

- High-Level Code:
int pow = 1;
int x = 0;

while (pow != 128){
 pow = pow * 2;
 x = x + 1;
}

• ARM Assembly Code:
; R0= pow, R1 = x
MOV R0, #1 ; pow = 1
MOV R1, #0 ; x = 0
WHILE
CMP R0, #128 ; pow != 128 ?
BEQ DONE ; if pow == 128, exit loop
LSL R0, R0, #1 ; pow = pow * 2
ADD R1, R1, #1 ; x = x + 1
B WHILE ; repeat loop
DONE

_ Al igual que las declaraciones if/else, el código ensamblador de los bucles while prueba la condición opuesta a la del código de alto nivel. Si esa condición opuesta es VERDADERA (en este caso, $R0 == 128$), el bucle while termina. Si no es así ($R0 \neq 128$), la rama no se toma y el cuerpo del bucle se ejecuta.

Ciclo for: estos son una forma abreviada conveniente que combina la inicialización, la comprobación de la condición y el cambio de la variable en un solo lugar. El formato del bucle for es:

```
for (initialization; condition; loop operation)
    statement
```

_ El código de inicialización se ejecuta antes de que comience el bucle for. La condición se comprueba al principio de cada bucle. Si la condición no se cumple, el bucle sale. La operación de bucle se ejecuta al final de cada bucle.

- High-Level Code:

```
int i;
int sum = 0;

for (i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}
```

- ARM Assembly Code:

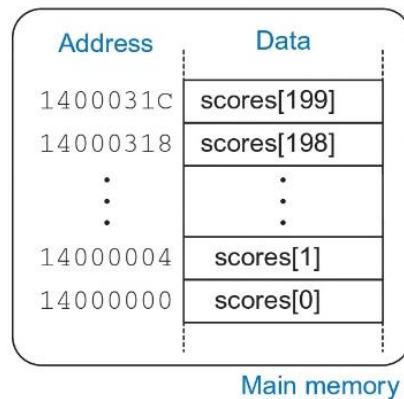
```
; R0= i, R1 = sum
MOV R1, #0          ; sum = 0
MOV R0, #0          ; i = 0 loop initialization
FOR CMP R0, #10      ; i < 10 ? check condition
BGE DONE            ; if (i >= 10) exit loop
ADD R1, R1, R0       ; sum = sum + i loop body
ADD R0, R0, #1       ; i = i + 1 loop operation
B FOR                ; repeat loop
DONE
```

_ Este ejemplo de código suma los números del 0 al 9. La variable de bucle, en este caso i, se inicializa a 0 y se incrementa al final de cada iteración de bucle. El bucle for se ejecuta siempre y cuando i sea inferior a 10. El bucle comprueba la condición < para continuar, por lo que el código de ensamblaje comprueba la condición opuesta, >=, para salir del bucle.

Memoria

_ Para facilitar el almacenamiento y el acceso, los datos similares pueden agruparse en un array (arreglo). Un array almacena su contenido en direcciones de datos secuenciales en la memoria. Cada elemento del array se identifica con un número llamado índice. El número de elementos de la matriz se denomina la longitud de la

matriz (length of the array). El siguiente grafico muestra un conjunto de 200 elementos de puntuación almacenados en la memoria.



Accediendo a un arreglo utilizando un bucle for: el siguiente código es un algoritmo de inflación de grado que añade 10 puntos a cada una de las puntuaciones. Podemos ver que no se muestra el código para inicializar el arreglo de puntuaciones. El índice del arreglo es una variable (i) en lugar de una constante, por lo que debemos multiplicarlo por 4 antes de agregarlo a la dirección base.

- High-Level Code

```
int i;
int scores[200];
...
```

```
for (i = 0; i < 200; i = i + 1)
    scores[i] = scores[i] + 10;
```

- ARM Assembly Code

```
; R0= array base address, R1 = i
; Initialization code...
MOV R0, #0x14000000          ; R0 = base address
MOV R1, #0                   ; i = 0
LOOP
CMP R1, #200                 ; i < 200?
BGE L3                       ; if i ≥ 200, exit loop
LSL R2, R1, #2               ; R2 = i*4
LDR R3, [R0, R2]             ; R3 = scores[i]
ADD R3, R3, #10              ; R3 = scores[i] + 10
STR R3, [R0, R2]             ; scores[i] = scores[i] + 10
ADD R1, R1, #1               ; i = i + 1
B LOOP                       ; repeat loop
L3
```

_ ARM puede escalar (multiplicar) el índice, añadirlo a la dirección base y cargarlo desde la memoria en una sola instrucción. En lugar de la secuencia de instrucciones LSL y LDR del código anterior, podemos usar una sola instrucción:

```
LDR R3, [R0, R1, LSL #2]
```

_ R1 es escalado (desplazado a la izquierda por dos) y luego añadido a la dirección base (R0). Por lo tanto, la dirección de memoria es $R0 + (R1 \times 4)$.

Modos de indexación:

_ Además de escalar el registro del índice (índex), ARM proporciona un direccionamiento Offset (compensado), Pre-Indexed (pre-indexado) y Post-Indexed (post-indexado) para permitir un código denso y eficiente para los accesos a los arreglos y las llamadas a las funciones.

Direccionamiento de compensación: (Offset addressing), calcula la dirección como el registro base \pm el desplazamiento; el registro base no cambia.

Direccionamiento pre-indexado: (Pre-indexed addressing), calcula la dirección como el registro base \pm el desplazamiento y actualiza el registro base a esta nueva dirección.

Direccionamiento post-indexado: (Post-indexed addressing), calcula la dirección sólo como el registro base y luego, después de acceder a la memoria, el registro base se actualiza al registro base \pm la desviación.

_ En la siguiente tabla se dan ejemplos de cada modo de indexación. En cada caso, el registro base es R1 y el offset es R2. El offset se puede restar escribiendo -R2. El offset también puede ser un inmediato en el rango de 0-4095 que puede ser sumado (por ejemplo, #20) o restado (por ejemplo, #-20).

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

Ciclo for utilizando post-indexado: este ejemplo de código muestra el bucle for del ejemplo de código anterior, reescrito para utilizar la post-indexación, eliminando el ADD para incrementar i.

- High-Level Code:
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)
 scores[i] = scores[i] + 10;

- ARM Assembly Code:
 - ; R0= array base address
 - ; initialization code...
 - MOV R0, #0x14000000 ; R0 = base address
 - ADD R1, R0, #800 ; R1 = base address + (200*4)
 - LOOP
 - CMP R0, R1 ; reached end of array?
 - BGE L3 ; if yes, exit loop
 - LDR R2, [R0] ; R2 = scores[i]
 - ADD R2, R2, #10 ; R2 = scores[i] + 10
 - STR R2, [R0], #4 ; scores[i] = scores[i] + 10
 - ; then R0 = R0 + 4
 - B LOOP ; repeat loop
 - L3

Bytes y caracteres

_ Los números en el rango [-128, 127] pueden ser almacenados en un solo byte en lugar de una palabra entera. Debido a que hay mucho menos de 256 caracteres en el teclado de un idioma inglés, en donde los caracteres ingleses son a menudo representados por bytes. El lenguaje C, por ejemplo, utiliza el tipo char para representar un byte o un carácter. Las primeras computadoras carecían de un mapeo estándar entre bytes y caracteres ingleses, por lo que el intercambio de texto entre computadoras era difícil. Es por eso que, en 1963, la Asociación Americana de Normalización publicó el Código Estándar Americano para el Intercambio de Información (ASCII), que asigna a cada carácter del texto un valor de byte único.

- Los valores ASCII se dan en hexadecimal.
- Las letras minúsculas y mayúsculas difieren en 0x20 (32).
- Byte significa 8 bits
- Halfword significa 16 bits (dos bytes)
- Word significa 32 bits (cuatro bytes)

_ A continuación, podemos ver una tabla que posee algunos caracteres en código ASCII:

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x

29)	39 9	49 I	59 Y	69 i	79 y
2A *	3A :	4A J	5A Z	6A j	7A z
2B +	3B ;	4B K	5B [6B k	7B {
2C ,	3C <	4C L	5C \	6C l	7C
2D -	3D =	4D M	5D]	6D m	7D }
2E .	3E >	4E N	5E ^	6E n	7E ~
2F /	3F ?	4F 0	5F _	6F o	

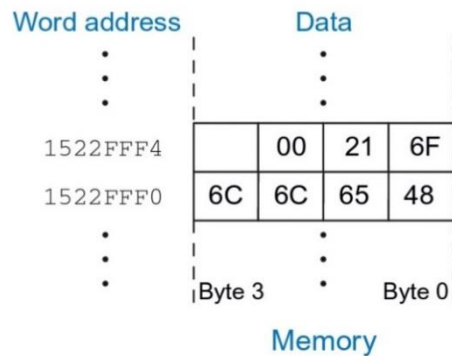
_ ARM proporciona un byte de carga (LDRB), un byte de carga firmado (LDRSB) y un byte de almacenamiento (STRB) para acceder a los bytes individuales de la memoria. LDRB cero-extiende el byte, mientras que LDRSB firma-extiende el byte para llenar todo el registro de 32 bits. STRB almacena el byte menos significativo del registro de 32 bits en la dirección del byte especificado en la memoria.

_ Los tres se observan en la imagen a continuación, con la dirección base R4 siendo 0. En donde:

- LDRB: carga el byte en la dirección de memoria 2 en el byte menos significativo de R1 y llena los restantes bits del registro con 0.
- LDRSB: carga este byte en R2 y extiende el byte en los 24 bits superiores del registro.
- STRB: almacena el byte menos significativo de R3 (0x9B) en el byte 3 de la memoria; reemplaza 0xF7 con 0x9B. Los bytes más significativos de R3 son ignorados.

	Memory					Registers					
Byte Address	3	2	1	0	R1	00	00	00	8C	LDRB	R1, [R4, #2]
Data	F7	8C	42	03	R2	FF	FF	FF	8C	LDRSB	R2, [R4, #2]
					R3	xx	xx	xx	9B	STRB	R3, [R4, #3]

Cadena de caracteres: son una serie de caracteres que tienen una longitud variable, por lo que los lenguajes de programación deben proporcionar una forma de determinar la longitud o el final de la cadena. En C, por ejemplo, el carácter nulo (0x00) significa el final de una cadena. A modo de ejemplo, la siguiente imagen muestra la cadena "¡Hola!" (0x48 65 6C 6C 6F 21 00) almacenada en la memoria. Dicha cadena tiene siete bytes de longitud y se extiende desde la dirección 0x1522FFF0 a 0x1522FFF6. El primer carácter de la cadena (H=0x48) se almacena en la dirección de byte más bajo (0x1522FFF0).



Ejercicio: el siguiente código de alto nivel convierte una serie de 10 entradas de caracteres de minúsculas a mayúsculas restando 32 de cada entrada de la serie. Hay que traducirlo al lenguaje de ARM. La idea es aplicar el uso de LDRB y STRB para acceder a un arreglo de caracteres. Recuerda que la diferencia de direcciones entre los elementos de la matriz es ahora de 1 byte, no de 4 bytes. Asumimos que R0 ya tiene la dirección base de la matriz.

```
// chararray[10] declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
    char array[i] = char array[i] - 32;
```

_ La solución es:

```
; ARM código ensamblador
; R0= base address of chararray (initialized earlier), R1 = i
MOV R1, #0          ; i = 0
LOOP
CMP R1, #10         ; i < 10 ?
BGE DONE            ; if (i >=10), exit loop
LDRB R2, [R0, R1]    ; R2 = mem[R0+R1] = chararray[i]
SUB R2, R2, #32      ; R2 = chararray[i] - 32
STRB R2, [R0, R1]    ; chararray[i] = R2
ADD R1, R1, #1       ; i = i + 1
B LOOP              ; repeat loop
DONE
```

Llamado a la función

_ Las funciones tienen entradas, llamadas argumentos, y una salida, llamada valor de retorno. Las funciones deben calcular el valor de retorno y no causar otros efectos secundarios no deseados. Cuando una función llama a otra, la función llamante (the caller) y la función llamada (the callee), deben acordar dónde poner los argumentos y el valor de retorno. En ARM, el caller coloca convencionalmente hasta cuatro argumentos en los registros R0-R3 antes de hacer la llamada a la función, y el callee coloca el valor de retorno en el registro R0 antes de terminar. Entonces, ambas funciones saben dónde encontrar los argumentos y el valor de retorno, incluso si el

caller y el callee fueron escritos por personas diferentes. El callee no debe interferir con el comportamiento del que llama. Este significa que el callee debe saber a dónde volver después de completar y no debe pisotear ningún registro o memoria que necesite la persona que llama.

_ La persona que llama almacena la dirección del remitente en el link register (LR) al mismo tiempo salta al callee usando la instrucción de branch and link (BL). El callee no debe sobrescribir ningún estado arquitectónico o memoria del que el caller depende. Específicamente, el callee debe dejar los saved register (R4-R11, y LR) y la pila, una porción de memoria utilizada para las variables temporales, sin modificar.

Función de llamada y return: ARM utiliza la instrucción branch and link (BL) para llamar una función y mueve el link register al PC (MOV PC, LR) para volver de una función. A continuación, vemos una analogía:

- High-Level Code:

```
int main() {  
    simple();  
    ...  
}
```

```
// void means the function returns no value
```

```
void simple() {  
    return;  
}
```

- ARM Assembly Code:

```
0x00008000 MAIN      ...  
...                  ...  
0x00008020 BL SIMPLE          ; call the simple function  
...  
  
0x0000902C SIMPLE MOV PC, LR      ; return
```

_ Este código de ejemplo muestra la función main() llamando a la función simple(). En este caso main() es el caller, y simple() es el callee. La función simple() se llama sin argumentos de entrada y no genera ningún valor de retorno; sólo regresa al caller. En este ejemplo las direcciones de las instrucciones se dan a la izquierda de cada instrucción de ARM en hexadecimal.

_ BL (branch and link) y MOV PC, LR son las dos instrucciones esenciales necesarias para una llamada de función y return. BL realiza dos tareas:

- Almacena la dirección de retorno de la siguiente instrucción (la instrucción después de BL) en el registro de enlace (LR).
- Se ramifica hacia la instrucción objetivo.

_ En el ejemplo anterior, la función principal llama a la función simple() ejecutando la instrucción de branch and link (BL). BL se ramifica a la etiqueta SIMPLE y almacena

0x00008024 en LR. La función simple() vuelve inmediatamente ejecutando la instrucción MOV PC, LR, copiando el retorno de la dirección del LR al PC. La función principal entonces continúa ejecutando en esta dirección (0x00008024).

Entrada de argumentos y retorno de valor: la función simple() en el ejemplo anterior no recibe ninguna entrada de la función de llamada main() y no devuelve ninguna salida. Por convención en ARM, las funciones usan R0-R3 para los argumentos de entrada y R0 para el valor de retorno. A continuación, vemos una analogía:

- High-Level Code:

```
int main() {  
    int y;  
    ...  
    y = diffofsums(2, 3, 4, 5);  
    ...  
}  
  
int diffofsums(int f, int g, int h, int i) {  
    int result;  
    result = (f + g) - (h + i);  
    return result;  
}
```

- ARM Assembly Code:

```
; R4 = y  
MAIN  
...  
MOV R0, #2;  
argument 0 = 2  
MOV R1, #3          ; argument 1 = 3  
MOV R2, #4          ; argument 2 = 4  
MOV R3, #5          ; argument 3 = 5  
BL DIFFOFSUMS       ; call function  
MOV R4, R0           ; y = returned value  
...  
  
;R4 = result  
DIFFOFSUMS  
ADD R8, R0, R1       ; R8 = f + g  
ADD R9, R2, R3       ; R9 = h + i  
SUB R4, R8, R9       ; result = (f + g) - (h + i)  
MOV R0, R4           ; put return value in R0  
MOV PC, LR           ; return to caller
```


_ En este ejemplo de código, la función DIFFOFSUMS se llama con cuatro argumentos y devuelve un resultado. El resultado es una variable local, que elegimos mantener en R4. Según la convención ARM, la función de llamada main() coloca los argumentos de la función de izquierda a derecha en los registros de entrada, R0-R3. La función llamada, DIFFOFSUMS, almacena el valor de retorno en el registro de retorno, R0. Cuando se llama a una función con más de cuatro argumentos, los argumentos de entrada adicionales se colocan en la pila.

The Stack

_ La pila es una memoria que se utiliza para guardar información dentro de una función. La pila se expande (utiliza más memoria) a medida que el procesador necesita consumir o arañar (scratch) más espacio, y se contrae (usa menos memoria) cuando el procesador ya no necesita las variables almacenadas allí. La pila posee la propiedad last-in-first-out (LIFO), en donde por ejemplo como una pila de platos, el último plato apilado (pushed) en la pila (el plato superior) es el primero que puede ser sacado (popped off). Cada función puede asignar el espacio de la pila para almacenar, pero debe ubicarla antes de regresar. La parte superior de la pila (top of the stack) es el espacio asignado más recientemente. Mientras que una pila de platos crece en el espacio, la pila de ARM crece en la memoria. La pila se expande a direcciones de memoria más bajas cuando un programa necesita más espacio para arañar. A continuación, vemos una imagen de una pila con un solo dato apilado y en el tope, y luego vemos otra imagen de la misma pila, pero con dos datos nuevos agregados, cambiando el tope de la misma:



_ El puntero de la pila, SP (R13), es un registro de ARM ordinario que, por convención, apunta a la parte superior de la pila. Un puntero es un nombre elegante para una dirección de memoria. SP apunta a (da la dirección de) datos. Por ejemplo, en la imagen (a), el puntero de la pila, SP, mantiene el valor de la dirección 0xBEFFFAE8 y apunta al valor de los datos 0xAB000001. El puntero de la pila (SP) comienza en una dirección de memoria alta y disminuye para expandirse según sea necesario. La imagen (b) muestra la pila expandiéndose para permitir dos palabras de datos más de almacenamiento temporal. Para ello, el SP disminuye en ocho para convertirse en 0xBEFFFAE0. Dos palabras de datos adicionales, 0x12345678 y 0xFFEEDDCC, se almacenan temporalmente en la pila.

_ Uno de los usos importantes de la pila es guardar y restaurar registros que son usados por una función. Una función debe calcular un retorno, pero no tienen otros efectos secundarios no deseados. En particular, no debe modificar ningún registro aparte de R0, el que contiene el valor de retorno.

_ La función DIFFOFSUMS en el código anterior viola esta regla porque modifica R4, R8 y R9. Si el principal hubiera estado usando estos registros antes de la llamada a los diffofsums, su contenido se habría corrompido por la llamada de la función. Para resolver este problema, una función guarda los registros en la pila antes de que los modifica, y luego los restaura de la pila antes de que vuelva. Específicamente, realiza los siguientes pasos:

- 1)_ Hace espacio en la pila para almacenar los valores de uno o más registros.
- 2)_ Almacena los valores de los registros en la pila.
- 3)_ Ejecuta la función utilizando los registros.
- 4)_ Restaurar los valores originales de los registros de la pila.
- 5)_ Distribuye el espacio en la pila.

_ Entonces, a continuación, vemos un ejemplo de código que muestra una versión mejorada de DIFFOFSUMS que salva y restaura R4, R8 y R9:

- ARM Assembly Code:
; R4 = result
DIFFOFSUMS
SUB SP, SP, #12 ; make space on stack for 3 registers
STR R9, [SP, #8] ; save R9 on stack
STR R8, [SP, #4] ; save R8 on stack
STR R4, [SP] ; save R4 on stack

ADD R8, R0, R1 ; R8 = f + g
ADD R9, R2, R3 ; R9 = h + i
SUB R4, R8, R9 ; result = (f + g) - (h + i)
MOV R0, R4 ; put return value in R0

LDR R4, [SP] ; restore R4 from stack
LDR R8, [SP, #4] ; restore R8 from stack
LDR R9, [SP, #8] ; restore R9 from stack
ADD SP, SP, #12 ; deallocate stack space

MOV PC, LR ; return to caller

_ La siguiente imagen muestra la pila antes, durante, y después de una llamada a la función DIFFOFSUMS del código anterior. La pila comienza a 0xBEF0F0FC. El diffofsums hace espacio para tres palabras en la pila disminuyendo el puntero de la pila SP en 12. Entonces almacena los valores actuales mantenidos en R4, R8 y R9 en el espacio recién

asignado. Ejecuta el resto de la función, cambiando los valores en estos tres registros. Al final de la función, el DIFFOFSUMS restaura los valores de estos registros de la pila, reparte su espacio de la pila y regresa. Cuando la función vuelve, R0 mantiene el resultado, pero no hay otros efectos secundarios, es decir, R4, R8, R9, y SP tienen los mismos valores que antes de la llamada de la función.

_ El espacio de la pila que una función se asigna a sí misma se llama stack frame. El stack frame de DIFFOFSUMS tiene tres palabras de profundidad. El principio de modularidad nos dice que cada función debe acceder sólo a su propia pila marco, no los marcos que pertenecen a otras funciones.

Cargar y almacenar múltiples registros: guardar y restaurar registros en la pila es una operación tan común que ARM proporciona instrucciones de cargar múltiples y almacenar múltiples (LDM y STM) que están optimizados para este propósito. En el ejemplo de código a continuación reescribe los diffofsums utilizando las instrucciones mencionadas. La pila contiene exactamente la misma información que en el ejemplo anterior, pero este código es mucho más corto.

- ARM Assembly Code:
; R4 = result
DIFFOFSUMS
STMFD SP!, {R4, R8, R9} ; push R4/8/9 on full descending stack
ADD R8, R0, R1 ; R8 = f + g
ADD R9, R2, R3 ; R9 = h + i
SUB R4, R8, R9 ; result = (f + g) - (h + i)
MOV R0, R4 ; put return value in R0
LDMFD SP!, {R4, R8, R9} ; pop R4/8/9 off full descending stack
MOV PC, LR ; return to caller

_ El LDM y el STM vienen en cuatro sabores para las pilas llenas y vacías descendentes y ascendentes (FD, ED, FA, EA).

- El SP! en las instrucciones indica que se deben actualizar los datos relativos al puntero de la pila y actualizar el puntero de la pila después de la tienda o la carga.
- PUSH y POP son sinónimos de STMFD SP!, {regs} y LDMFD SP!, {regs}, respectivamente, y son la forma preferida de guardar registros en la pila convencional de descenso completo.

Registros preservados y no preservados: los dos ejemplos de código anterior suponen que todos los registros utilizados (R4, R8 y R9) deben guardarse y restaurarse. Si la función de llamada no utiliza esos registros, el esfuerzo de salvarlos y restaurarlos es inútil. Para evitar este desperdicio, ARM divide los registros en categorías preservadas (preserved) y no preservadas (nonpreserved).

_ Los registros preservados incluyen el R4-R11. Los registros no preservados son R0-R3 y R12. SP y LR (R13 y R14) también deben ser preservados. Una función debe guardar y

restaurar cualquiera de los registros preservados que se desee utilizar, pero puede cambiar los registros no preservados libremente.

- ARM Assembly Code:
; R4 = result
DIFFOFSUMS
PUSH {R4} ;save R4 on stack
ADD R1, R0, R1 ;R1 = f + g
ADD R3, R2, R3 ;R3 = h + i
SUB R4, R1, R3 ;result = (f + g) - (h + i)
MOV R0, R4 ;put return value in R0
POP {R4} ;pop R4 off stack
MOV PC, LR ;return to caller

_ Este código muestra una versión mejorada de DIFFOFSUMS que sólo ahorra R4 en la pila. También ilustra el PUSH y el POP. El código reutiliza los registros de argumento no preservados R1 y R3 para mantener las sumas intermedias cuando esos argumentos ya no sean necesarios.

_ Recordar que cuando una función llama a otra, la primera es caller y el último es el callee. El callee debe guardar y restaurar cualquier registro conservado que desee utilizar. El callee puede cambiar cualquiera de los registros no conservados. Por lo tanto, si la persona que llama tiene datos activos en un registro no conservado, debe guardar ese registro no conservado antes de hacer la llamada de la función y luego necesita restaurarla después. Por estas razones, los registros preservados también se llaman "callee-save", y los no preservados se llaman "caller-save".

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

_ En el cuadro se resumen los registros que se preservan. R4-R11 son generalmente utilizados para mantener las variables locales dentro de una función, por lo que deben ser salvados. LR también debe ser guardado, para que la función sepa dónde volver.

_ R0-R3 y R12 se utilizan para mantener los resultados temporales. Estos cálculos normalmente se completan antes de que se haga una llamada de función, así que no son preservados, y es raro que el que llama necesite salvarlos. R0-R3 son a menudo sobrescritos en el proceso de llamar a una función. Por lo tanto, deben ser salvados por el caller si éste depende de cualquiera de sus propios argumentos después de que una función llamada regrese. R0 ciertamente no debería preservarse, porque el callee devuelve su resultado en este registro. El Registro de Estado del Programa Actual

(CPSR) contiene las banderas de condición. No se preserva a través de las llamadas de función.

_ La pila sobre el stack pointer se conserva automáticamente mientras que el callee no escribe a las direcciones de memoria por encima de SP. De esta manera, no modifica el stack frame de cualquier otra función. El puntero de la pila se conserva, porque el callejero ubica su marco de pila antes volviendo a sumar la misma cantidad que restó de SP en el comienzo de la función.

Código DIFFOFSUMS optimizado:

- ARM Assembly Code:
DIFFOFSUMS
ADD R1, R0, R1 ;R1 = f + g
ADD R3, R2, R3 ;R3 = h + i
SUB R0, R1, R3 ;return (f + g) – (h + i)
MOV PC, LR ;return to caller

Nonleaf Function Calls: una función que no llama a otras se llama función hoja (leaf function); DIFFOFSUMS es un ejemplo. Ahora, una función que sí llama a los demás se llama función sin hojas (nonleaf function). Las funciones no relacionadas con las hojas son algo más complicadas porque puede que tengan que guardar los registros no preservados en la pila antes de llamar otra función y luego restaurar esos registros después. Específicamente:

- Caller save rule: antes de una llamada de función, la persona que llama debe guardar los registros no preservados (R0-R3 y R12) que necesite después de la llamada. Después de la llamada, debe restaurar estos registros antes de usarlos.
- Callee save rule: antes de que una callee perturbe cualquiera de los registros preservados (R4-R11 y LR), debe guardar los registros. Antes de que regrese, debe restaurar estos registros.

_ El siguiente ejemplo de código demuestra una función nonleaf f1 y una función leaf f2 incluyendo todo lo necesario para guardar y preservar los registros. Supongamos que f1 mantiene i en R4 y x en R5. Vemos que f2 mantiene r en R4. f1 usa los registros preservados R4, R5 y LR, por lo que inicialmente los empuja en el apilar de acuerdo con la regla de guardado del callee. Utiliza el R12 para mantener el resultado intermedio (a - b) para no tener que preservar otro registro para este cálculo. Antes de llamar a f2, f1 empuja a R0 y R1 en la pila de acuerdo con la regla de guardar del caller porque estos son registros no preservados que la f2 podría cambiar y que la f1 seguirá necesitando después de la llamada.

- High-Level Code:
int f1(int a, int b) {
 int i, x;

 x = (a + b)*(a - b);

```

    for (i=0; i < a; i++){
        x = x+f2 (b+i);
    return x;
}

```

```

int f2(int p){
    int r;
    r = p+5;
    return r+p;
}

```

- ARM Assembly Code:

```
; R0 = a, R1 = b, R4 = i, R5 = x
```

```
F1
```

```
PUSH {R4, R5, LR}      ;save preserved registers used by f1
```

```
ADD R5, R0, R1          ;x = (a + b)
```

```
SUB R12, R0, R1         ;temp = (a - b)
```

```
MUL R5, R5, R12         ;x = x * temp = (a + b) * (a - b)
```

```
MOV R4, #0              ;i = 0
```

```
FOR
```

```
CMP R4, R0              ;i < a?
```

```
BGE RETURN             ;no: exit loop
```

```
PUSH {R0, R1}          ;save nonpreserved registers
```

```
ADD R0, R1, R4          ;argument is b + i
```

```
BL F2                   ;call f2(b+i)
```

```
ADD R5, R5, R0          ;x = x + f2(b+i)
```

```
POP {R0, R1}           ;restore nonpreserved registers
```

```
ADD R4, R4, #1          ;i++
```

```
B    FOR                ;continue for loop
```

```
RETURN
```

```
MOV R0, R5              ;return value is x
```

```
POP {R4, R5, LR}       ;restore preserved registers
```

```
MOV PC, LR              ;return from f1
```

```
;R0 = p, R4 = r
```

```
F2
```

```
PUSH {R4}               ;save preserved registers used by f2
```

```
ADD R4, R0, #5           ;r = p + 5
```

```
ADD R0, R4, R0           ;return value is r + p
```

```
POP {R4}                ;restore preserved registers
```

```
MOV PC, LR              ; return from f2
```

_ Aunque el R12 también es un registro no preservado que la f2 podría sobrescribir, f1 ya no necesita el R12 y no tiene que guardarlo. f1 entonces pasa el argumento a f2 en R0, hace la llamada a la función, y usa el resultado en R0. f1 entonces restaura R0 y R1

porque todavía los necesita. Cuando f1 se hace, pone el valor de retorno en R0, restaura los registros preservados R4, R5, y LR, y regresa. f2 guarda y restaura R4 de acuerdo con la regla de guardado del callee.