

CC4301

Arquitectura de Computadores

Profesor: Pablo Guerrero P.

Índice general

1. Diseño de circuitos digitales	13
1.1. Sistemas digitales	13
1.2. Sistemas combinacionales	14
1.2.1. Representaciones	14
1.2.2. Compuertas lógicas	15
1.2.3. Álgebra de Boole	16
1.2.4. Diferentes notaciones de operadores	17
1.2.5. Diseño de circuitos combinacionales	17
1.2.6. Simplificación de sumas de productos	18
1.2.7. Expansión de Shannon	19
1.2.8. Mapas de Karnaugh	20
1.2.9. Sumador 1 bit, 3 entradas	23
1.2.10. Sumador 4 bits, 2 entradas	24
1.2.11. Funciones incompletamente especificadas	25
1.3. Sistemas secuenciales	26
1.3.1. Diagrama de tiempo	26
1.3.2. Reloj	26
1.3.3. Circuito secuencial	27

1.3.4. Diagramas de estado	27
1.3.5. Especificacion informal	28
1.3.6. Flip flop data	29
1.3.7. Implementación de circuitos secuenciales	31
1.3.8. Tiempo de retardo	36
1.3.9. Elementos biestables	37
1.4. Diseño modular	47
1.4.1. Motivación	47
1.4.2. Configuraciones Modulares	47
1.4.3. Estrategias de comunicación	48
1.4.4. Selección de Entradas o Salidas	50
1.4.5. Compuerta tristate	56
1.4.6. Elementos Reseteables	57
1.4.7. Sumador 1 bit 3 entradas	58
1.4.8. Contador	59
1.4.9. Comparadores	61
1.4.10. Shifters	62
1.4.11. Multiplicadores	63
2. Arquitectura lógica de un computador	69
2.1. Representación númerica	69
2.1.1. Notación hexadecimal	70
2.1.2. Enteros	71
2.1.3. Comparación entre signed y unsigned	72
2.1.4. Obtención de $[[x]]_s$	72

ÍNDICE GENERAL	5
2.1.5. Interpretación de $[[x]]_s$	73
2.1.6. Suma de enteros	75
2.1.7. Resta de enteros	76
2.1.8. Punto flotante	76
2.1.9. Casos especiales	77
2.1.10. Codificacion BCD	78
2.1.11. Reducción	78
2.1.12. Extensión sin signo	79
2.1.13. Extensión con signo	79
2.2. Arquitectura lógica	80
2.2.1. Tipos de ISA	80
2.2.2. CPU	81
2.2.3. Memoria	82
2.2.4. Principio funcionamiento computador	83
2.2.5. Lenguaje de máquina vs Assembler	83
2.2.6. Programa almacenado en memoria	84
2.2.7. Tipos de instrucciones	85
2.2.8. Tipos de argumentos	85
2.2.9. Arquitectura Von Neumann	85
2.2.10. Arquitectura Harvard	86
2.3. Assembler x86	86
2.3.1. Registros	86
2.3.2. Registros de propósito general	87
2.3.3. Registros de índices	87
2.3.4. Registros de punteros	88

2.3.5. Registros de puntero de instrucción	88
2.3.6. Assembler	89
2.3.7. Sintaxis	89
2.3.8. Instrucciones aritmeticas	90
2.3.9. Multiplicación-división	90
2.3.10. Copia información (mov)	91
2.3.11. Copiar dirección de memoria (lea)	91
2.3.12. Direccionamiento indirecto	91
2.3.13. Extensión	92
2.3.14. Comparaciones	92
2.3.15. Saltos condicionales	92
2.3.16. Implementación if	93
2.3.17. Operaciones lógicas	93
2.3.18. Shift y rotate	93
2.3.19. Direccionamiento directo (o desplazamiento)	94
2.3.20. Direccionamiento indirecto	94
2.3.21. Ejemplos	95
2.4. Pila	96
2.4.1. Llamadas a subrutinas	96
2.4.2. Convención del llamador	96
2.4.3. Manejo stack en llamadas	97
2.4.4. Búffer overflow	98
2.4.5. Generación de ejecutable	99
2.4.6. Llamadas a sistema en Linux	99
2.4.7. Comparación de sintaxis	100

ÍNDICE GENERAL	7
2.4.8. Hello world	101
2.4.9. Formato de instrucción	103
2.4.10. Ejemplos	104
2.4.11. Variaciones opcode	104
2.4.12. Prefijos	104
2.5. ARM Assembler	105
2.5.1. Modos del Procesador	105
2.5.2. Modo User	105
2.5.3. Modos Privilegiados	105
2.5.4. Modos de Excepción	106
2.5.5. Modo System	106
2.5.6. Registros	106
2.5.7. Asignación	109
2.5.8. Operaciones aritméticas	109
2.5.9. Comparaciones	111
2.5.10. Operaciones lógicas	111
2.5.11. Saltos	112
2.5.12. Ejecución condicional	113
2.5.13. Load - Store	113
2.5.14. Stack	115
2.5.15. Ejemplo	117
3. Arquitectura física de un computador	119
3.1. Memoria	119
3.2. CPU	128

3.2.1.	M32	128
3.2.2.	CPU	128
3.2.3.	Ciclo Instrucción	129
3.2.4.	CPU-M32	129
3.2.5.	Sincronidad	134
3.2.6.	Operaciones	135
3.2.7.	Implementación ALU	137
3.2.8.	Circuito combinacional	137
3.2.9.	Registro síncrono	138
3.2.10.	Banco de registros	138
3.2.11.	Unidad de control	139
3.2.12.	Optimización	139
3.3.	Apéndice	140
3.3.1.	Preguntas	140
3.3.2.	Referencias	141
4.	Entrada/Salida	143
4.1.	Lectura / Escritura en Dispositivo E/S	143
4.2.	Mapeo en memoria	145
4.2.1.	Ejemplo: visor de calculadora	145
4.3.	Controlador de entrada y salida	148
4.3.1.	Típicas líneas de comunicación	148
4.4.	Comunicación	149
4.4.1.	Comunicación paralela	149
4.4.2.	Comunicación serial	149

ÍNDICE GENERAL	9
4.5. Ciclo busy-waiting	152
4.6. Interrupciones	152
4.6.1. Interrupciones (por HW)	152
4.6.2. Habilitación	153
4.6.3. Implementación	153
4.6.4. Interrupciones por SW	154
4.6.5. Dirección en memoria	154
4.6.6. Ciclo de ejecución	154
4.6.7. Resguardo registros	155
4.6.8. Deshabilitación interrupciones	155
4.6.9. Código de resguardo	155
4.6.10. Ejemplo	156
4.6.11. Demora atención	157
4.6.12. Evaluación	157
4.6.13. Problema	158
4.7. Direct Memory Access (DMA)	158
4.7.1. Interfaz DMA	158
4.7.2. Configuración DMA	159
4.7.3. Procedimiento DMA	159
4.7.4. Observaciones	159
4.7.5. Evaluación	160
4.8. Ejercicios propuestos	160
4.8.1. Problema 1	160
4.8.2. Problema 2	160

5. Arquitecturas avanzadas	163
5.1. Memoria caché	163
5.1.1. Problema	163
5.1.2. Experimento	163
5.1.3. Localidad espacial	164
5.1.4. Localidad temporal	164
5.1.5. Grados! de asociatividad	165
5.1.6. Lectura	165
5.1.7. Observaciones	167
5.1.8. Políticas para la escritura	167
5.1.9. Ejemplo	167
5.1.10. Eficiencia	168
5.1.11. Problema	168
5.1.12. Jerarquía de memoria	169
5.1.13. Jerarquía de Buses	169
5.2. Pipeling	169
5.2.1. Partes instrucción	169
5.2.2. Pipeline	171
5.2.3. Hazards	171
5.2.4. Data Hazards	171
5.2.5. Structural Hazard	173
5.2.6. Control Hazard	173
5.2.7. Pipeling	173
5.2.8. Delayed Branches	175
5.2.9. Niveles de pipelining	175

ÍNDICE GENERAL	11
5.2.10. Branch Prediction	178
5.3. Arquitecturas superescalares	178
5.3.1. Grados de pipelining	178
5.3.2. Implementación	179
5.3.3. Diagrama de pipeline	179
5.3.4. Dependencia	180
5.3.5. Ejecución fuera de orden	181
5.3.6. Register Renaming	182
5.3.7. Ejecución especulativa	183
5.4. Multi-core chips	183
5.4.1. Symmetric multiprocessing (SMP)	183
5.4.2. Multi Core Chips	184
5.4.3. Caché Coherence	184
5.4.4. MESI	185
5.4.5. Request For Ownership (RFO)	187
5.4.6. Ejercicios resueltos	187
5.5. Anexos	191

Capítulo 1

Diseño de circuitos digitales

1.1. Sistemas digitales

Características:

1. Operan en tiempo discreto



2. Entradas y salidas digitales: es decir, para operar reciben números y retornan números (en nuestro caso 0s o 1s)



Existen 2 tipos de sistemas digitales:

1. Sistemas combinacionales

2. Sistemas secuenciales

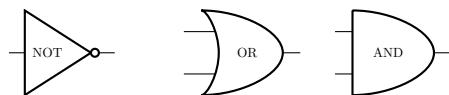
1.2. Sistemas combinacionales

Los sistemas combinacionales no poseen memoria, por ende la salida en un instante de tiempo k es una función de las entradas en ese mismo instante de tiempo:

$$y^k = f(x^k)$$

Ejemplos:

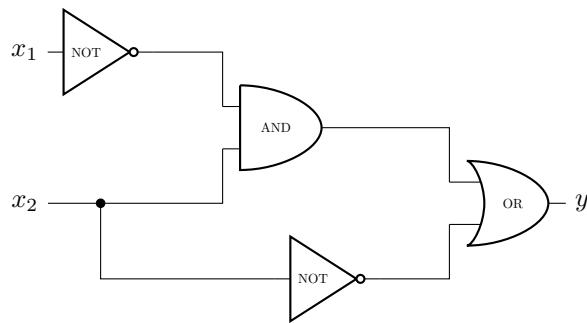
- Compuertas lógicas:



- Combinaciones no retroalimentadas
 - Operaciones lógicas: mux, demux, cod, decod.
 - Operaciones aritméticas: sumador, multiplicador.

1.2.1. Representaciones

Podemos representar los circuitos combinacionales de dos formas:



- Tablas de verdad: Para cada combinación de entradas se especifican las salidas.
Las entradas deben ir en orden creciente

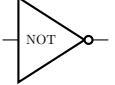
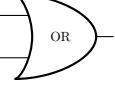
n entradas

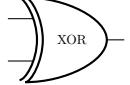
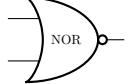
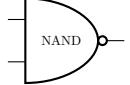
$$2^n \left\{ \begin{array}{c|c|c} & \overbrace{x_1 \quad x_2}^{\text{Entradas}} & y \\ \hline 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array} \right. \quad \leftarrow \text{salida}$$

- Fórmulas algebraicas: Se usa notación algebraica que se deriva directamente del circuito. Ejemplo: fórmula circuito 1

$$y = \neg x_1 \wedge x_2 \vee \neg x_2$$

1.2.2. Compuertas lógicas

Símbolo	Significado	Tabla de verdad	Fórmula algebraica															
	Negación	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>x</th><th>y</th></tr> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td>0</td></tr> </table>	x	y	0	1	1	0	$y = \neg x$									
x	y																	
0	1																	
1	0																	
	O lógico	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>x_1</th><th>x_2</th><th>y</th></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	x_1	x_2	y	0	0	0	0	1	1	1	0	1	1	1	1	$y = x_1 \vee x_2$
x_1	x_2	y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
	Y lógico	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>x_1</th><th>x_2</th><th>y</th></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	x_1	x_2	y	0	0	0	0	1	0	1	0	0	1	1	1	$y = x_1 \wedge x_2$
x_1	x_2	y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																

Símbolo	Significado	Tabla de verdad	Fórmula algebraica															
	O exclusivo	<table border="1"> <thead> <tr> <th>x_1</th><th>x_2</th><th>y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x_1	x_2	y	0	0	0	0	1	1	1	0	1	1	1	0	$y = x_1 \oplus x_2$
x_1	x_2	y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
	O negado	<table border="1"> <thead> <tr> <th>x_1</th><th>x_2</th><th>y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x_1	x_2	y	0	0	1	0	1	0	1	0	0	1	1	0	$y = \neg(x_1 \vee x_2)$
x_1	x_2	y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
	Y negado	<table border="1"> <thead> <tr> <th>x_1</th><th>x_2</th><th>y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x_1	x_2	y	0	0	1	0	1	1	1	0	1	1	1	0	$y = \neg(x_1 \wedge x_2)$
x_1	x_2	y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

1.2.3. Álgebra de Boole

Un álgebra de Boole es una estructura algebraica formada por un conjunto y operaciones. En nuestro caso analizaremos $\langle A, \vee, \wedge \rangle$.

Este conjunto cumple con ser álgebra de Boole si y solo si:

- **Neutros:**
 $\exists 0 \in A : \forall a \in A, a \vee 0 = a$
 $\exists 1 \in A : \forall a \in A, a \wedge 1 = a$
- **Distributividad:** $\forall a, b, c \in A :$
 $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
 $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
- **Negación:** $\forall a \in A, \exists \neg a \in A :$
 $a \wedge \neg a = 0$
 $a \vee \neg a = 1$
- **Asociatividad:** $\forall a, b, c \in A :$
 $a \vee (b \vee c) = (a \vee b) \vee c = a \vee b \vee c$
 $a \wedge (b \wedge c) = (a \wedge b) \wedge c = a \wedge b \wedge c$
- **Commutatividad:** $\forall a, b \in A :$

$$\begin{aligned} a \vee b &= b \vee a \\ a \wedge b &= b \wedge a \end{aligned}$$

Teoremas básicos de un álgebra de Boole

- **Idempotencia:** $\forall a \in A :$

$$\begin{aligned} a \vee a &= a \\ a \wedge a &= a \end{aligned}$$

- **Dominación:** $\forall a \in A :$

$$\begin{aligned} a \vee 1 &= 1 \\ a \wedge 0 &= 0 \end{aligned}$$

- **Cancelación:** $\forall a, b \in A :$

$$\begin{aligned} a \vee (a \wedge b) &= a \\ a \wedge (a \vee b) &= a \end{aligned}$$

- **Leyes de De Morgan:** $\forall a, b \in A :$

$$\begin{aligned} \neg(a \vee b) &= \neg a \wedge \neg b \\ \neg(a \wedge b) &= \neg a \vee \neg b \end{aligned}$$

1.2.4. Diferentes notaciones de operadores

Un mismo operador se puede representar de distintas maneras:

<i>OR</i>	$+$	\cup	\vee	$ $	\parallel
<i>AND</i>	\cdot	\cap	\wedge	$\&$	$\&\&$
<i>NOT</i>	\bar{a}	$'$	\neg	\sim	!

1.2.5. Diseño de circuitos combinacionales

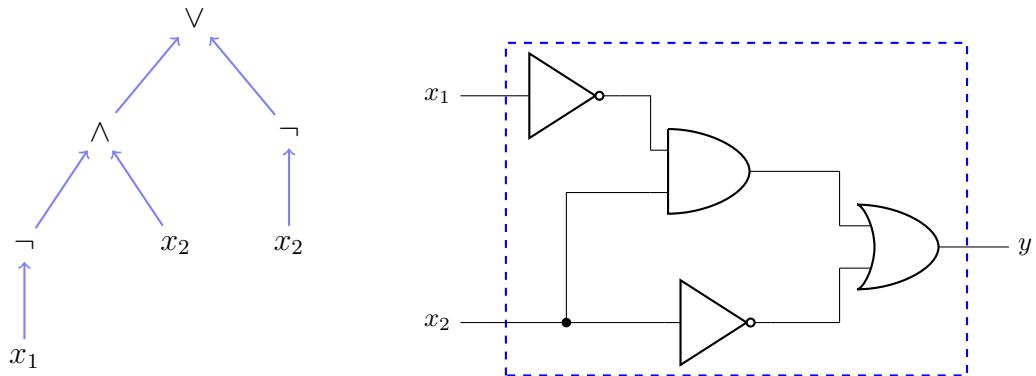
Para diseñar un circuito combinacional a partir de una tabla de verdad seguimos los siguientes pasos:

- Obtención de ecuación: Seleccionamos las entradas cuya salida es 1, luego para cada una de estas salidas tomamos sus entradas y creamos una ecuación usando solo operadores AND, de tal manera que esta ecuación sea verdadera para las entradas dadas. Finalmente unimos todas las ecuaciones usando operadores OR.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

$\Rightarrow \neg x_1 \wedge x_2 \Rightarrow \vee \Rightarrow (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$
 $\Rightarrow x_1 \wedge \neg x_2 \Rightarrow \vee \Rightarrow (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$

- Diseño a partir de ecuación: Se construye el árbol de operaciones de la expresión y se reemplazan los operadores por las respectivas compuertas lógicas.



1.2.6. Simplificación de sumas de productos

- Método 1:
Sea $v = f(\{x_i\})$, $w = g(\{x_i\})$

$$vw \vee \neg vw = w$$

Esta regla nos dice que dado un OR podemos eliminar los elementos tales que dicha variable esta a un lado del OR y su negado se ubica al otro lado del OR.
Ejemplo:

$$\begin{aligned} x_1 x_2 \neg x_3 \vee x_1 \neg x_2 \neg x_3 &= \\ x_1 \neg x_3 & \end{aligned}$$

- Método 2:
Sea $w = f(\{x_i\}) :$

$$\begin{aligned} & (x_j \wedge \neg x_k \wedge w) \vee \\ & (\neg x_j \wedge x_k \wedge w) \vee \\ & (x_j \wedge x_k \wedge w) = \\ & (x_j \vee x_k) \wedge w \end{aligned}$$

Ejemplo:

$$\begin{aligned} (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2) \\ = x_1 \vee x_2 \end{aligned}$$

1.2.7. Expansión de Shannon

Cualquier función se puede descomponer como un OR entre 2 expresiones de la siguiente forma:

$$\begin{aligned} f(x_1, \dots, x_i, \dots, x_n) = \\ x_i f(x_1, \dots, 1, \dots, x_n) \vee \neg x_i f(x_1, \dots, 0, \dots, x_n) \end{aligned}$$

Luego si $x_i = 1$

$$\begin{aligned} \underbrace{x_i}_{1} f(x_1, \dots, \boxed{1}, \dots, x_n) \vee \underbrace{\neg x_i}_{0} f(x_1, \dots, \boxed{0}, \dots, x_n) = \\ f(x_1, \dots, 1, \dots, x_n) \vee \overbrace{f(x_1, \dots, 0, \dots, x_n)}^{0} = \\ f(x_1, \dots, 1, \dots, x_n) \end{aligned}$$

Si $x_i = 0$

$$\begin{aligned} \underbrace{x_i}_{0} f(x_1, \dots, \boxed{1}, \dots, x_n) \vee \underbrace{\neg x_i}_{1} f(x_1, \dots, \boxed{0}, \dots, x_n) = \\ \overbrace{f(x_1, \dots, 1, \dots, x_n)}^{0} \vee f(x_1, \dots, 0, \dots, x_n) = \\ f(x_1, \dots, 0, \dots, x_n) \end{aligned}$$

Como se observa, la función nunca cambia su valor final.

1.2.8. Mapas de Karnaugh

Para visualizar mejor los productos que se pueden reducir se usan tablas de verdad reordenadas, llamadas mapas de Karnaugh.

Los mapas de Karnaugh son tablas de verdad de doble entrada, estas tablas trabajan con una sola salida, por ende si tenemos varias salidas será necesario crear varios mapas de Karnaugh.

Algunas variables cambian con las filas, el resto con las columnas.

Ejemplo:

x_2	x_1	x_0	y	
0	0	0	1	$\neg x_2 \neg x_1 \neg x_0$
0	0	1	1	$\neg x_2 \neg x_1 x_0$
0	1	0	0	
0	1	1	1	$\neg x_2 x_1 x_0$
1	0	0	1	$x_2 \neg x_1 \neg x_0$
1	0	1	0	
1	1	0	0	
1	1	1	0	

Creamos el mapa de Karnaugh, donde la primera columna son los posibles valores que puede tomar x_2 , y la primera fila son los posibles valores y combinaciones que puede tomar el par x_1x_0 :

$x_2 \setminus x_1 x_0$	00	01	11	10
0				
1				

Luego completamos de acuerdo a los valores entregados en la tabla de verdad:

$x_2 \setminus x_1 x_0$	00	01	11	10
0	1	1	1	0
1	1	0	0	0

A continuación agrupamos los 1 en rectángulos de tamaño potencia de 2:

$x_2 \setminus x_1 x_0$	00	01	11	10
0	1	1	1	0
1	1	0	0	0

Finalmente para cada rectángulo obtenemos su expresión usando opeadores AND. Para esto seleccionamos las entradas cuyo valor no cambia en ninguna casilla del rectángulo, si dicha entrada es 1 agregamos la entrada, si es 0 agregamos la entrada negada. Por último, unimos las expresiones de cada rectángulo usando operadores OR.

$$\neg x_1 \neg x_0 \vee \neg x_2 x_0$$

Gray code

Para crear la tabla de Karnaugh vamos colocando los grupos de números de modo tal que sus vecinos solo tengan 1 bit diferente.

$x_2 \setminus x_1 x_0$	00	01	11	10
0	0	0	0	0
1	1	0	0	1

Circularidad

Los mapas de Karnaugh cumplen con la propiedad de ser circulares.

$x_2 \setminus x_1 x_0$	00	01	11	10
0	0	0	0	0
1	1	0	0	1

Reciclaje

En los mapas de Karnaugh se pueden aprovechar los 1 más de una vez.

$x_2 \setminus x_1 x_0$	00	01	11	10
0	1	0	0	0
1	1	0	0	1

Además debemos tener claro que siempre se deben construir rectángulos lo más grandes posible. Agrupando 2^n elementos se eliminan n variables.

Ejemplo: (debemos mantener gray code)

$x_3 x_2 \setminus x_1 x_0$	00	01	11	10
00	1	1	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

$$x_2 x_0 \vee \neg x_2 \neg x_0 \vee \neg x_3 \neg x_1 x_0$$

Con n variables

Se debe mantener gray code.

Se puede construir recursivamente.

- 1 bit: 0 1
- 2 bits: 00 01 11 10
- 3 bits: 000 001 011 010 110 111 101 100
- N bits: 0GC(N-1) 1Reflejar(GC(N-1))

1.2.9. Sumador 1 bit, 3 entradas

Queremos sumar 3 bits x_0, x_1, x_2 . Deseamos obtener 2 salidas:

Resultado(r)

Acarreo(carry, c)



El acarreo se hace 1 cuando la suma supera 1.

x_2	x_1	x_0	c	r
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

Creamos una tabla de Karnaugh para cada salida.

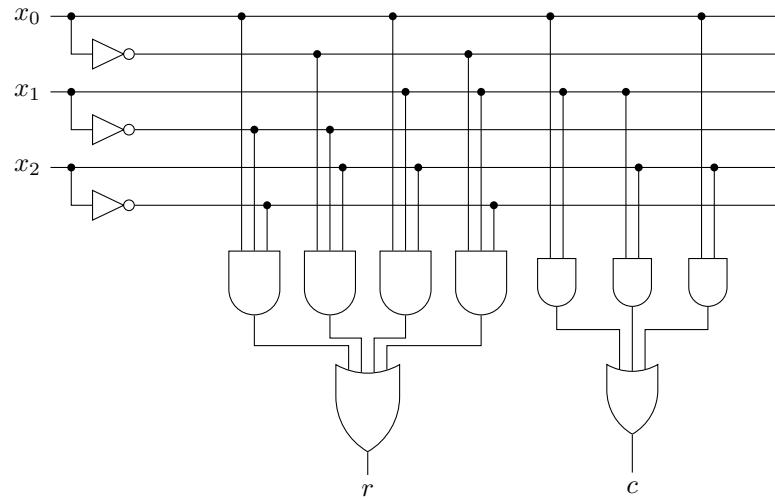
$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$r = \neg x_2 \neg x_1 \neg x_0 \vee \neg x_2 \neg x_1 x_0 \vee x_2 x_1 x_0 \vee \neg x_2 x_1 \neg x_0$$

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

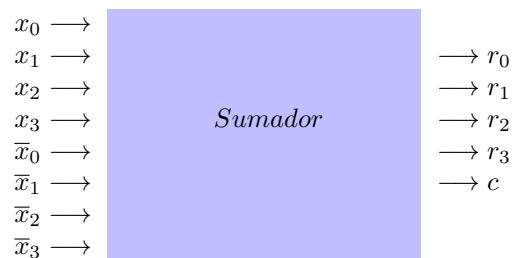
$$c = x_2 x_1 \vee x_1 x_0 \vee x_2 x_0$$

Diagrama circuitual:



1.2.10. Sumador 4 bits, 2 entradas

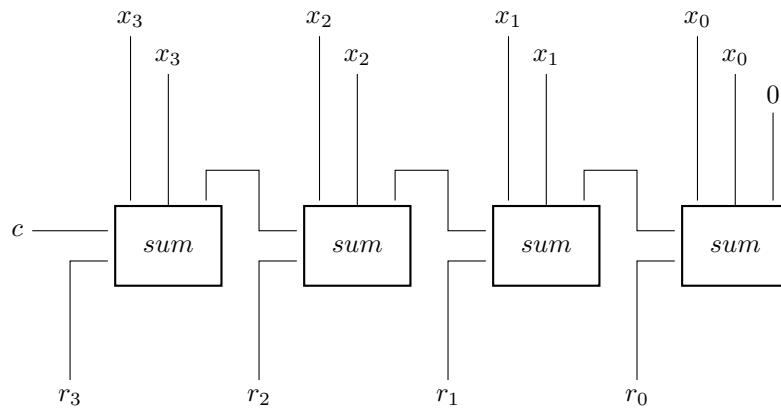
Este sumador tiene 4 salidas y 1 carry



Tamaño de la tabla de verdad de este sumador es:

$$2^8 = 256 \text{ filas!}$$

Solución modular:



1.2.11. Funciones incompletamente especificadas

Puede haber ciertas combinaciones de entradas para las cuales ciertas salidas sean irrelevantes (don't care, \times).

$x_3x_2 \setminus x_1x_0$	00	01	11	10
00	1	0	1	1
01	0	\times	1	1
11	\times	\times	\times	\times
10	1	1	\times	\times

Cada \times puede quedar fuera o dentro del rectángulo de acuerdo a la conveniencia.

$x_3x_2 \setminus x_1x_0$	00	01	11	10
00	1		1	1
01	0	\times	1	1
11	\times	\times	\times	\times
10	1	1	\times	\times

$$\neg x_2 \neg x_0 \vee x_3 \vee x_1$$

1.3. Sistemas secuenciales

Tienen memoria, por ende la salida en un tiempo k puede depender de las entradas anteriores a ese tiempo k .

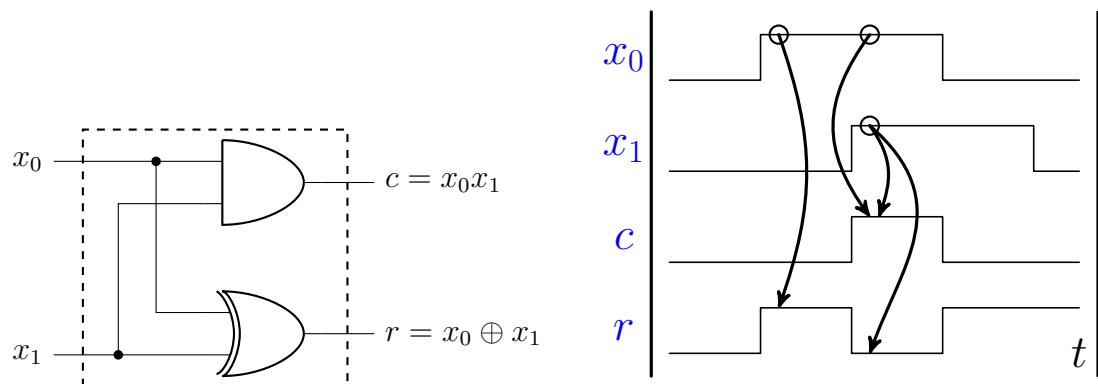
$$y^k = f(\{x^i\}_{i=0,\dots,k-1})$$

Se puede asumir que tienen un estado interno:

$$\begin{aligned} y^k &= f(x^{k-1}, s^k) \\ s^k &= g(s^{k-1}, x^{k-1}) \end{aligned}$$

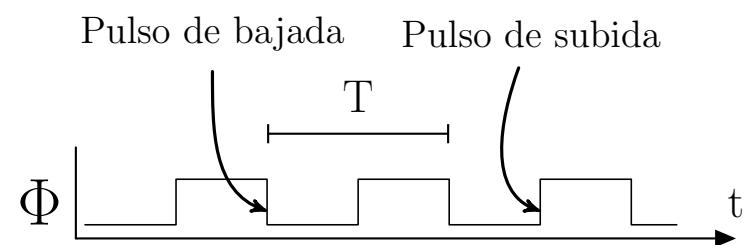
1.3.1. Diagrama de tiempo

Muestra los valores de señales en el tiempo. Pueden explicitar causalidades.
Retardo de compuertas: $10ps = 10^{-11}s$



1.3.2. Reloj

Genera una señal que cambia periódicamente entre 0 y 1.



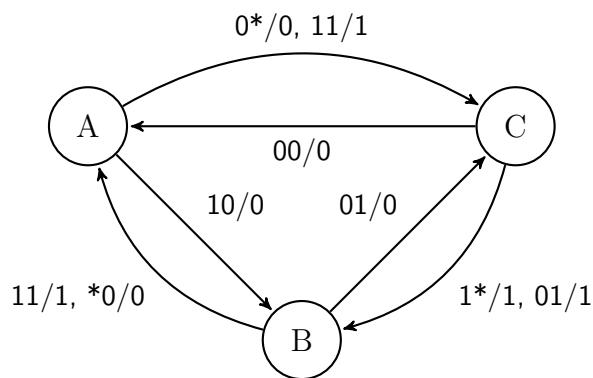
1.3.3. Circuito secuencial

Los circuitos secuenciales tienen un número finito de estados.

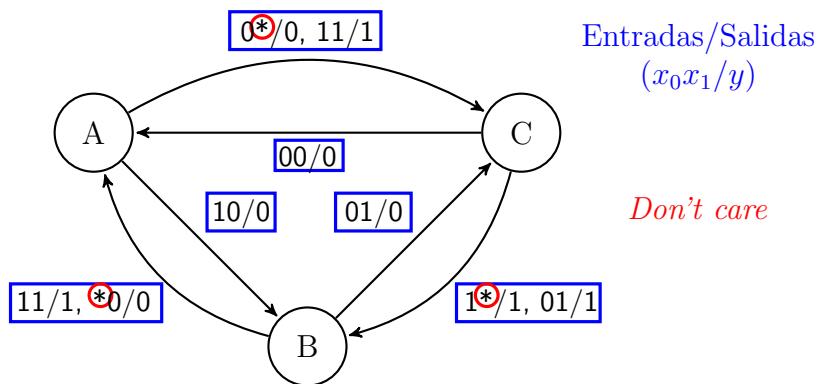
Tienen una entrada de reloj: En pulsos de bajada se calculan las salidas, en el resto se mantiene constante.

Las salidas se calculan en función de las entradas y del estado interno.

1.3.4. Diagramas de estado



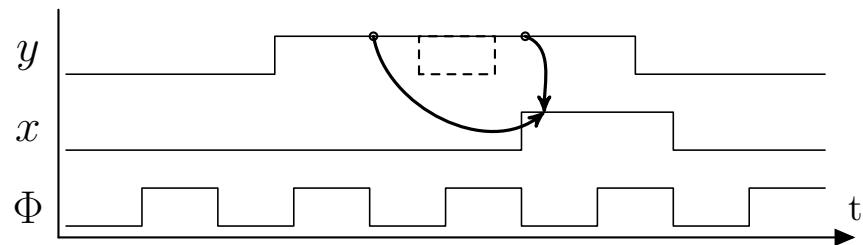
Especifican formalmente el comportamiento de un circuito secuencial.



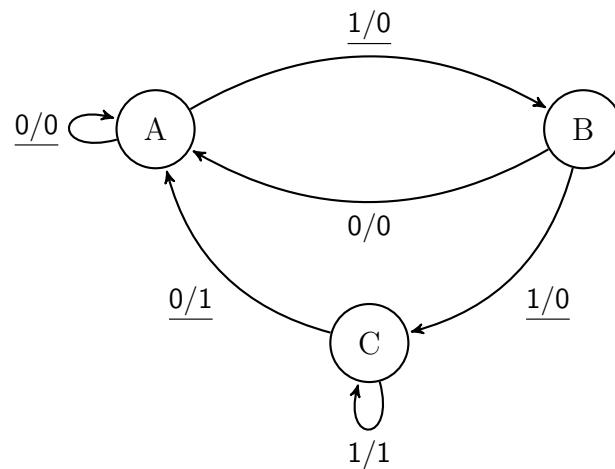
1.3.5. Especificación informal

Un diagrama de tiempo puede describir informalmente un circuito secuencial.

Ejemplo: detector de secuencias 1 - 1

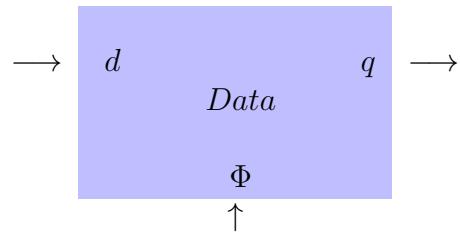


A partir del diagrama de tiempo podemos construir el digrama de estados:

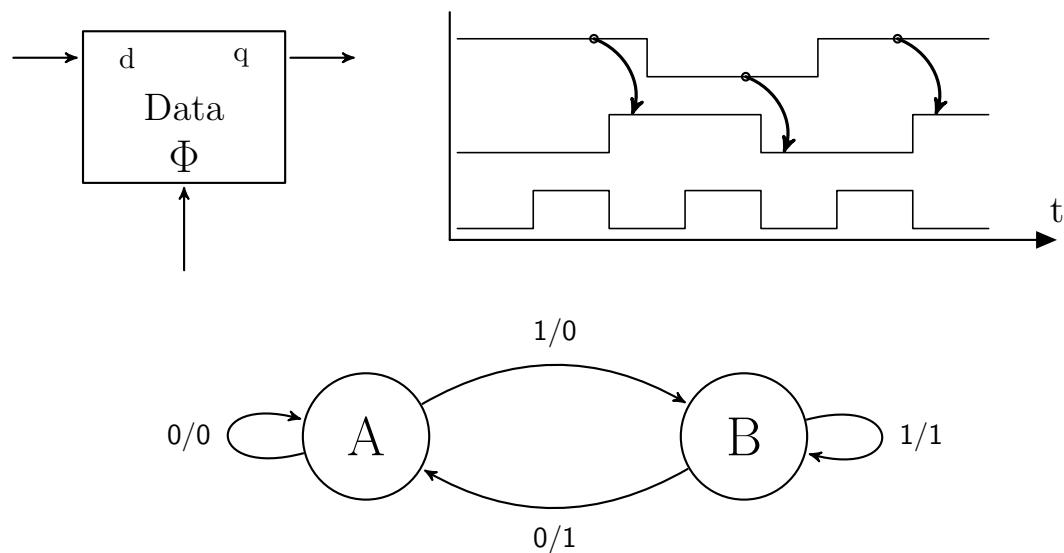


Las transiciones subrayadas son las necesarias para ser compatibles con el diagrama de tiempo.

1.3.6. Flip flop data

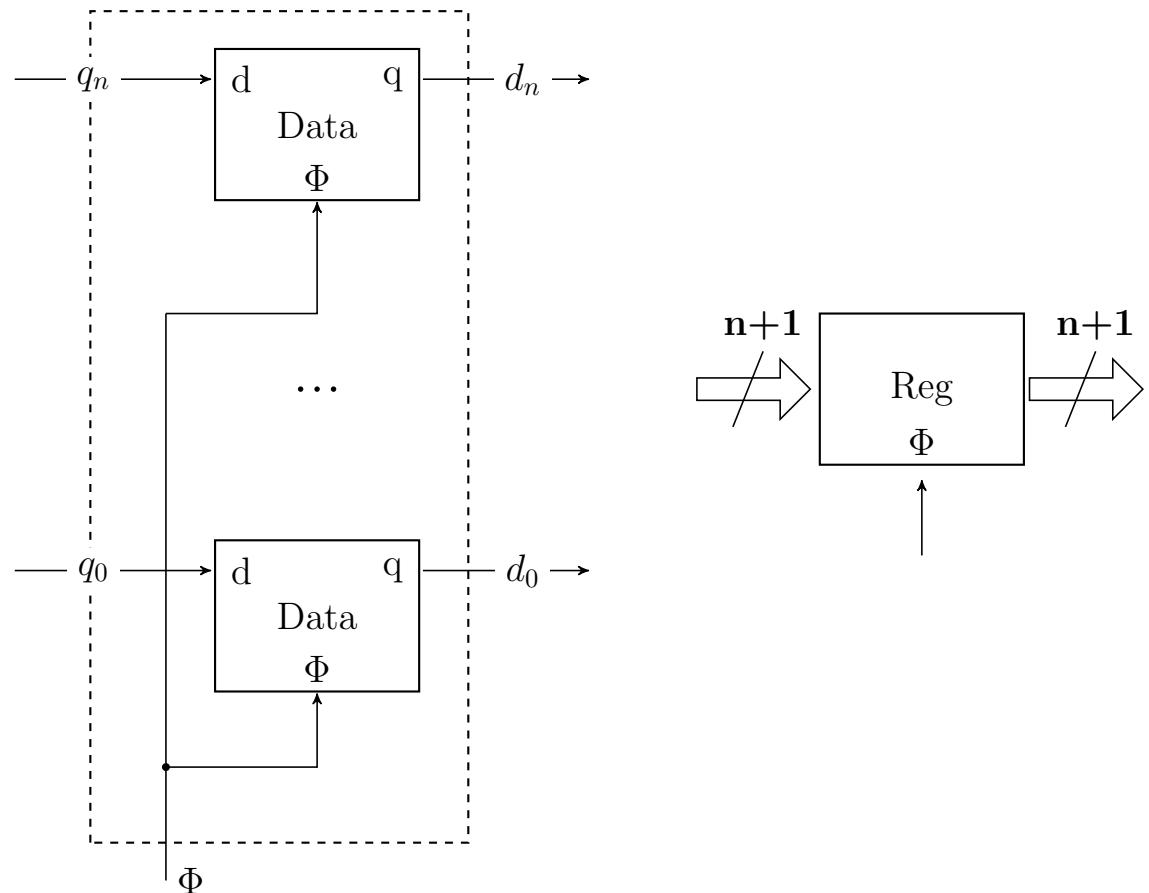


Es un circuito secuencial mínimo.
Permite almacenar un valor por un ciclo.



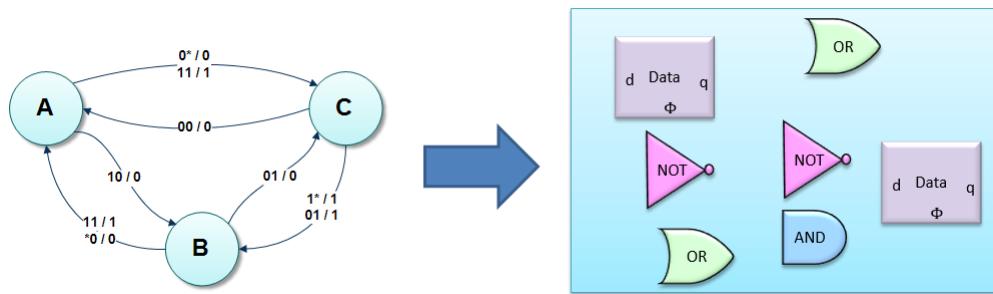
Almacenamiento de varios bits

Para esto colocamos varios flip flop en paralelo.

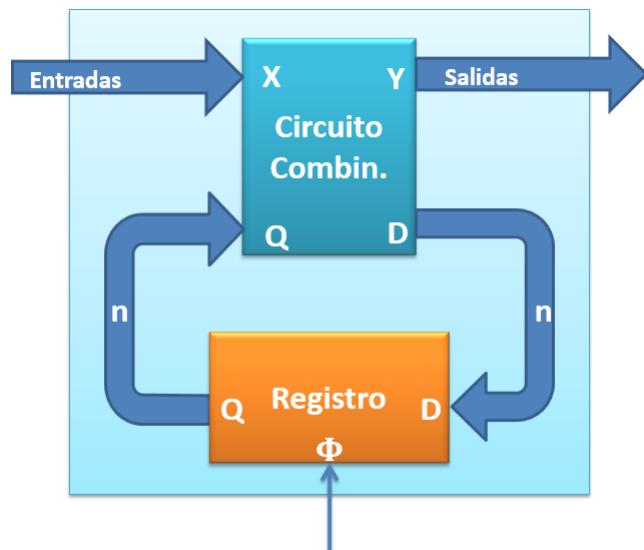


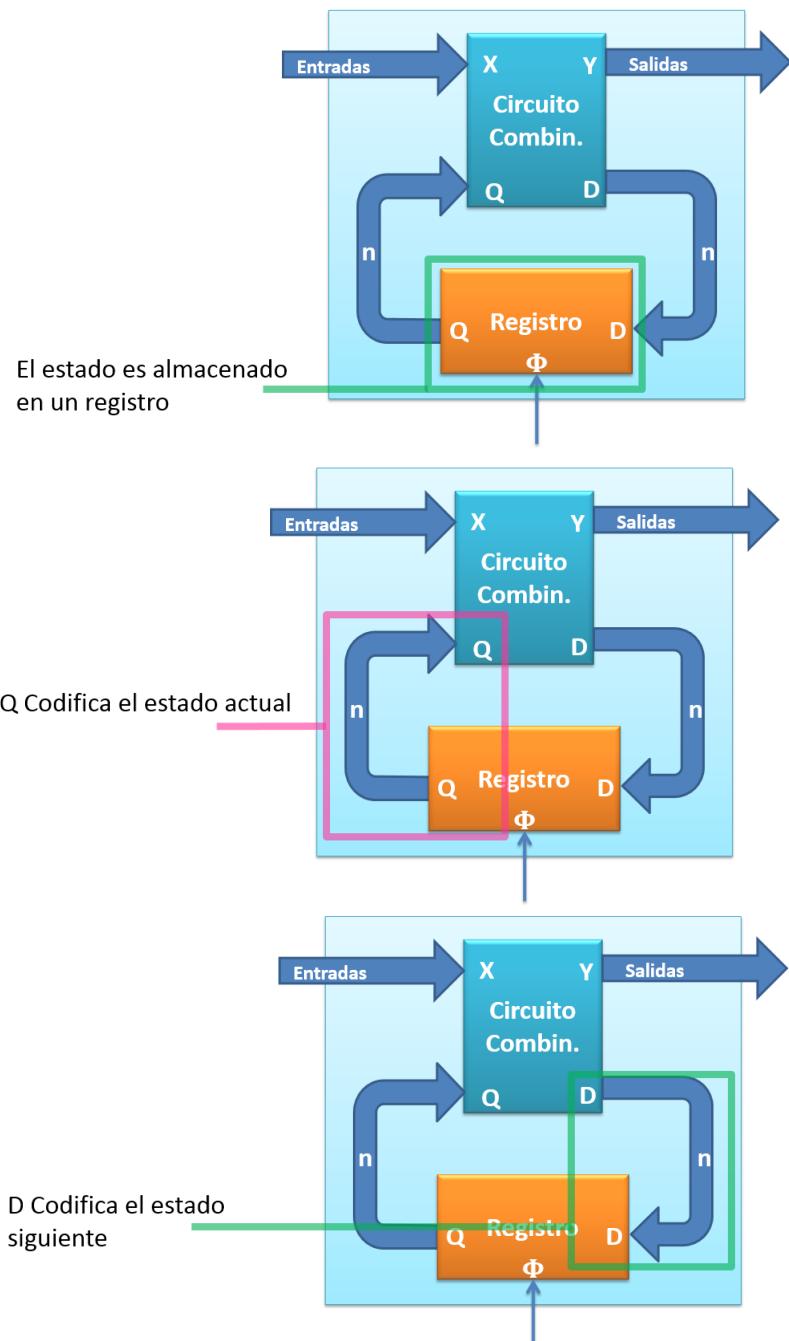
1.3.7. Implementación de circuitos secuenciales

Sabemos que los circuitos secuenciales usan elementos combinacionales y de memoria.
 ¿Cómo combinarlos?



Forma general



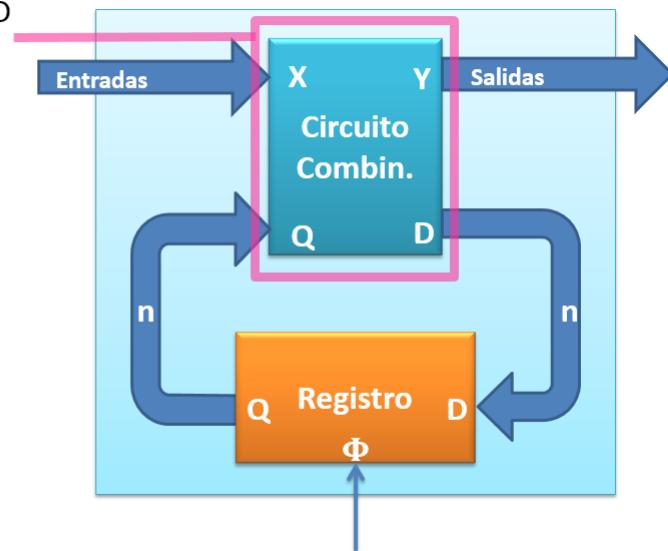


Calcula:

- Estado siguiente, D
- Salida, Y

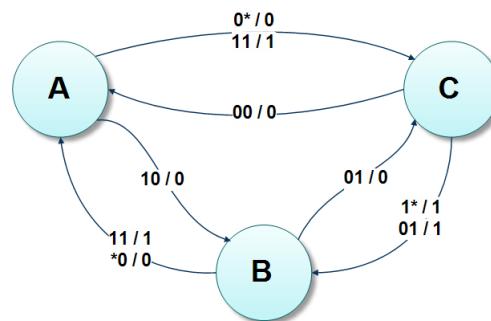
En función de:

- Estado actual, Q
- Entradas, X



Metodología

¿Cómo diseñamos el circuito para una máquina de estados dada?



Asignamos a cada estado un número binario. Como hay 3 estados necesitamos $\lceil \log_2 3 \rceil$ flip flop data.

Estado	q_1	q_0
A	0	0
B	0	1
C	1	0

El estado no forma parte de la interfaz del circuito por lo que cualquier permutación de una codificación es válida.

$E.$	q_1	q_0	x_1	x_0	d_1	d_0	y
A	0	0	0	0	1	0	0
A	0	0	0	1	1	0	0
A	0	0	1	0	0	1	0
A	0	0	1	1	1	0	1
B	0	1	0	0	0	0	0
B	0	1	0	1	1	0	0
B	0	1	1	0	0	0	0
B	0	1	1	1	0	0	1
C	1	0	0	0	0	0	0
C	1	0	0	1	0	1	1
C	1	0	1	0	0	1	1
C	1	0	1	1	0	1	1
-	1	1	0	0	\times	\times	\times
-	1	1	0	1	\times	\times	\times
-	1	1	1	0	\times	\times	\times
-	1	1	1	1	\times	\times	\times

Al igual que q_1q_0 , d_1d_0 representa un estado, pero en este caso el de llegada. x_1x_0 son las transiciones. y representa la salida.

Creamos los mapas de Karnaugh:

$q_{1,0} \setminus x_{1,0}$	00	01	11	10
00	1	1	1	0
01	0	1	0	0
11	×	×	×	×
10	0	0	0	0

$$d_1 = \neg q_1 \neg q_0 x_0 \vee \neg q_1 \neg q_0 \neg x_1 \vee \neg q_1 \neg x_1 x_0$$

$q_{1,0} \setminus x_{1,0}$	00	01	11	10
00	0	0	0	1
01	0	0	0	0
11	×	×	×	×
10	0	1	1	1

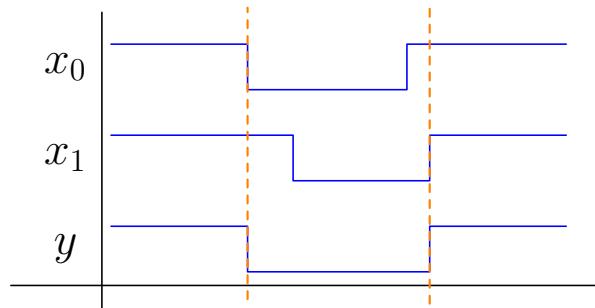
$$d_0 = q_1 x_0 \vee \neg q_0 x_1 \neg x_0$$

$q_{1,0} \setminus x_{1,0}$	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	×	×	×	×
10	0	1	1	1

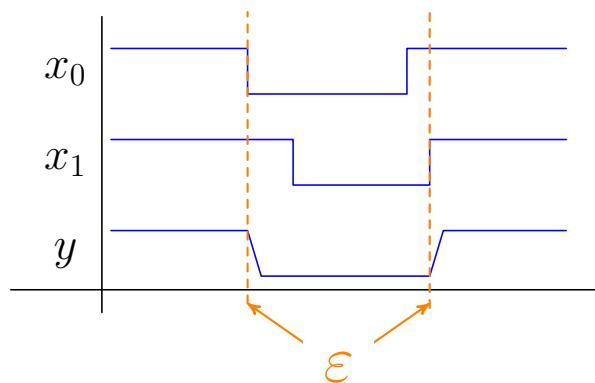
$$y = x_1 x_0 \vee q_1 x_1 \vee q_1 x_0$$

1.3.8. Tiempo de retardo

Hemos supuesto que las compuertas no introducen retardo.



En realidad hay un retardo ε



El retardo varía pero lo asumiremos constante $T = n\varepsilon$

1.3.9. Elementos biestables

Tienen 2 salidas:

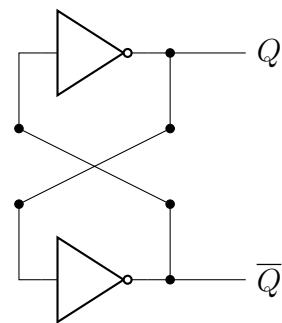
- q
- $\neg q$



Tienen 2 estados estables:

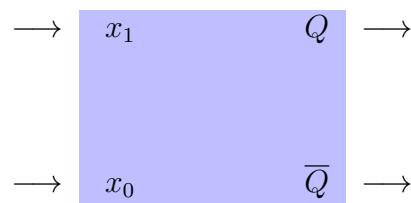
- $q = 0$ ($\neg q = 1$)
- $q = 1$ ($\neg q = 0$)

Implementación



Elementos biestables con entradas

Las entradas permiten controlar el estado.

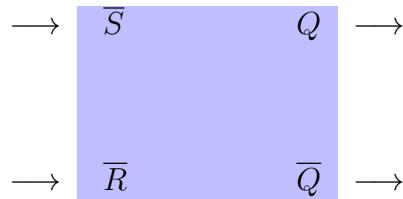


Hay 2 tipos:

- Latch (asíncronos): las salidas pueden cambiar en cualquier momento.
- Flip-flop (síncronos): las salidas solo pueden cambiar en los pulsos de bajada del reloj.

Latch SR

Permite controlar una salida q y su negado $\neg q$

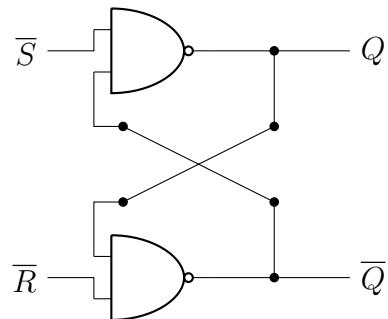


- $S = 1, R = 0$ (set) $\Rightarrow Q \leftarrow 1$
- $S = 0, R = 1$ (reset) $\Rightarrow Q \leftarrow 0$
- $S, R = 1 \Rightarrow Q, \bar{Q} \leftarrow 1$
- $S, R = 0$ (hold) $\Rightarrow Q \leftarrow Q$

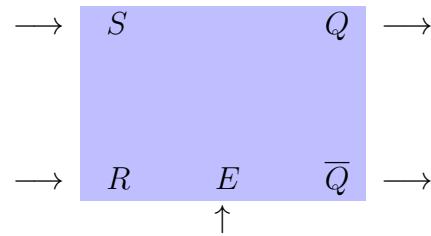
\bar{S}	\bar{R}	Q_{next}	\bar{Q}_{next}	Desc.
0	0	1	1	-
0	1	1	0	set
1	0	0	1	reset
1	1	Q	Q	hold

Implementación

Implementación con NANDs



Latch SR con compuerta

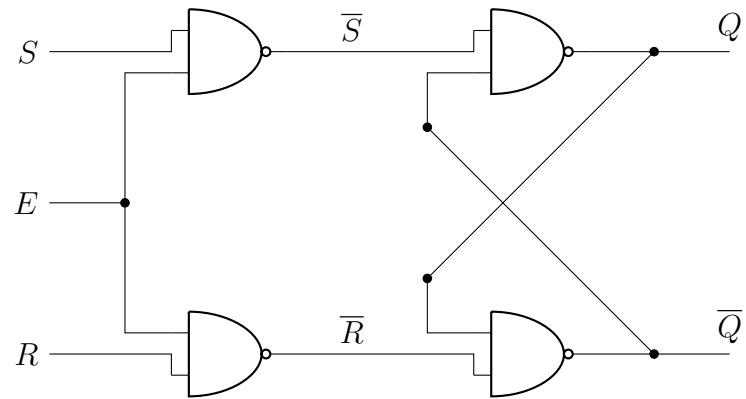


- $E = 0 \implies Q \leftarrow Q$
- $E = 1 \implies$ Comportamiento de SR

E	S	R	Q_{next}	\bar{Q}_{next}	Desc.
0	\times	\times	Q	\bar{Q}	disabled
1	0	0	Q	\bar{Q}	hold
1	0	1	0	1	reset
1	1	0	1	0	set
1	1	1	1	1	-

Implementación

Implementación con NANDs



Latch data

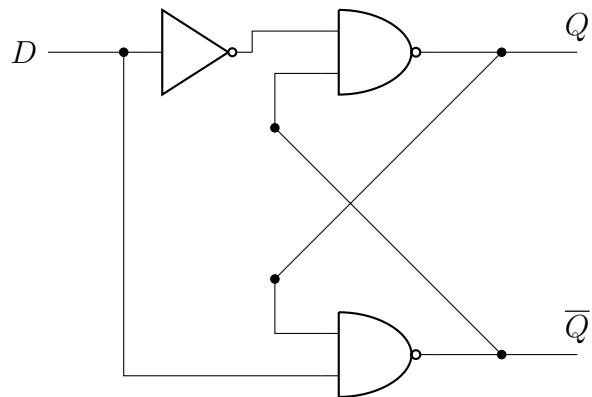
Permite olvidarse del problema de la simultaneidad de S y R coordinando ambas entradas.



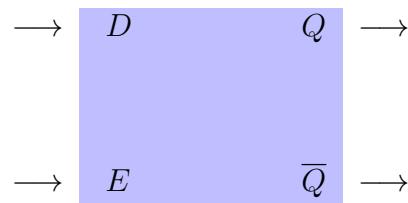
D	Q_{next}	\bar{Q}_{next}
0	0	1
1	1	0

Implementación

Implementación con NANDs



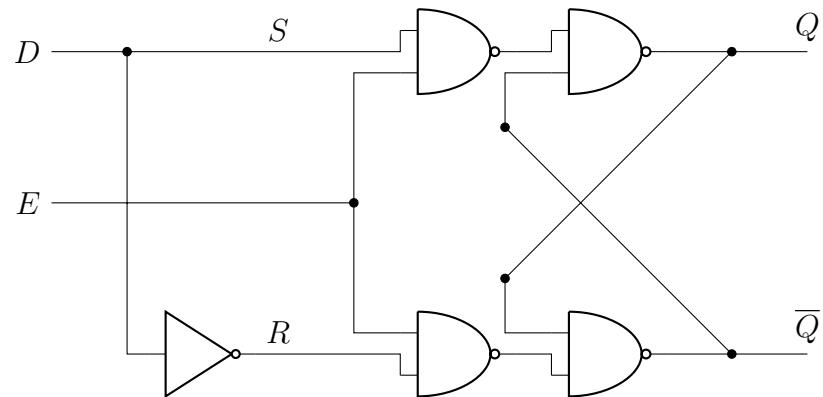
Latch data con compuerta



E	D	Q_{next}	\bar{Q}_{next}
0	X	Q	Q
1	0	0	1
1	1	1	0

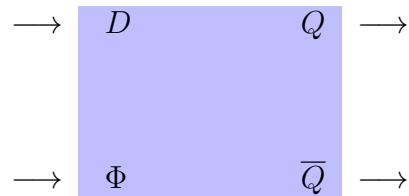
- Mientras $E = 1$, $Q = D$
- Cuando $E = 0$, Q no cambia

Implementacion

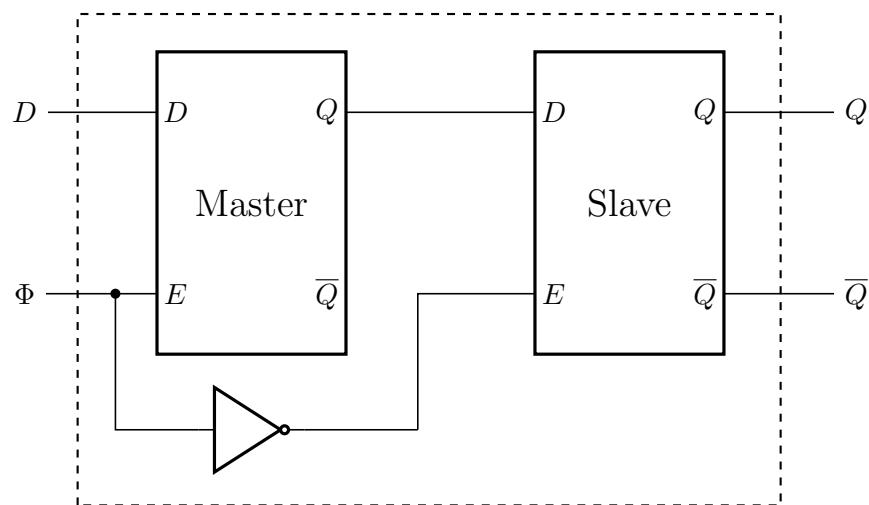


Flip-flop data

Copia D en el ciclo de bajada del reloj.

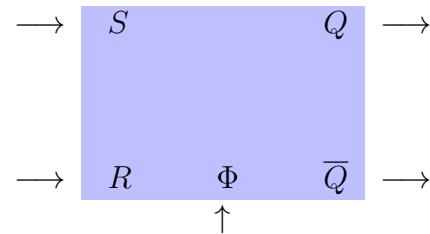
**Implementación**

Implementación en base a latch data (maestro-esclavo).



Flip-flop SR

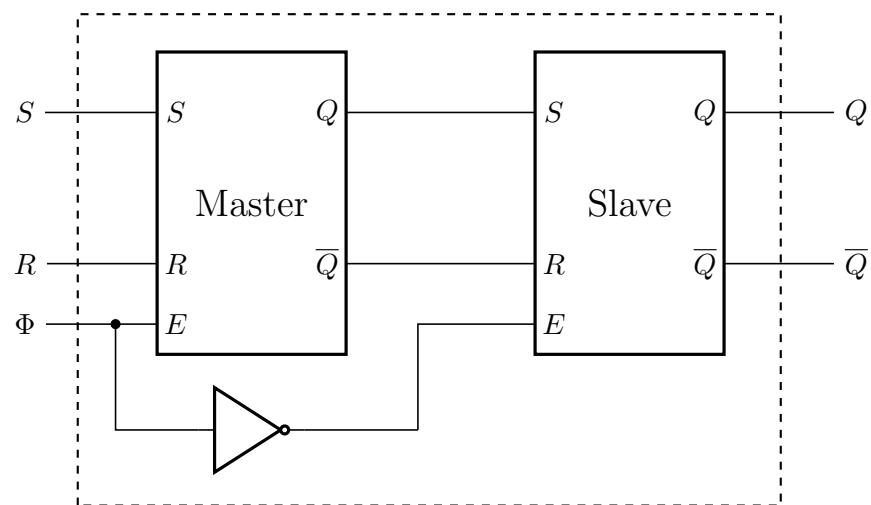
Solo toma en cuenta S y R en los ciclos de bajada del reloj.



S	R	Q_{next}	\bar{Q}_{next}	Desc.
0	0	Q	\bar{Q}	hold
0	1	0	1	reset
1	0	1	0	set
1	1	X	X	-

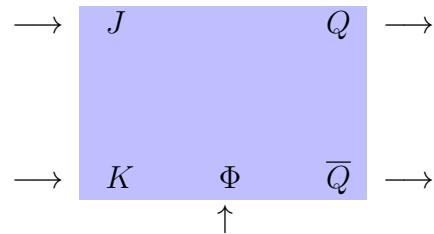
Implementación

Implementación en base a latch SR (maestro-esclavo)



Flip-flop JK

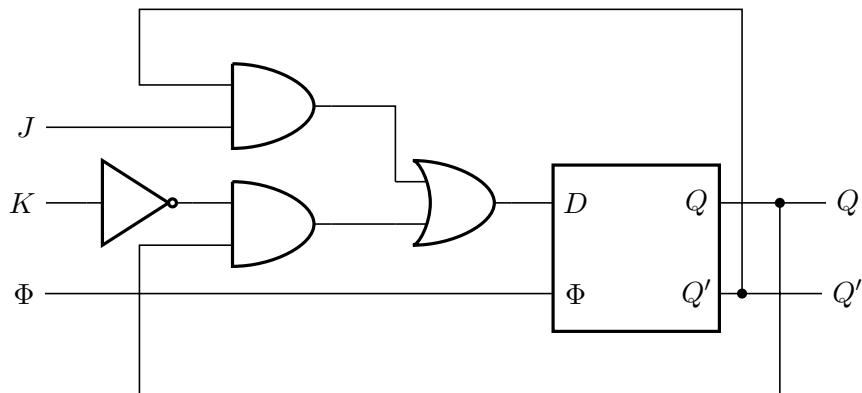
Muy similares a flip-flop SR pero permiten hacer toggle con ambas entradas en 1.



J	K	Q_{next}	\bar{Q}_{next}	Desc.
0	0	Q	\bar{Q}	hold
0	1	0	1	reset
1	0	1	0	set
1	1	\bar{Q}	Q	toggle

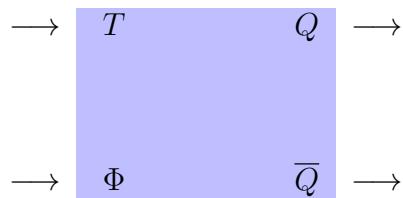
Implementación

Implementación en base a flip-flop D



Flip-flop T

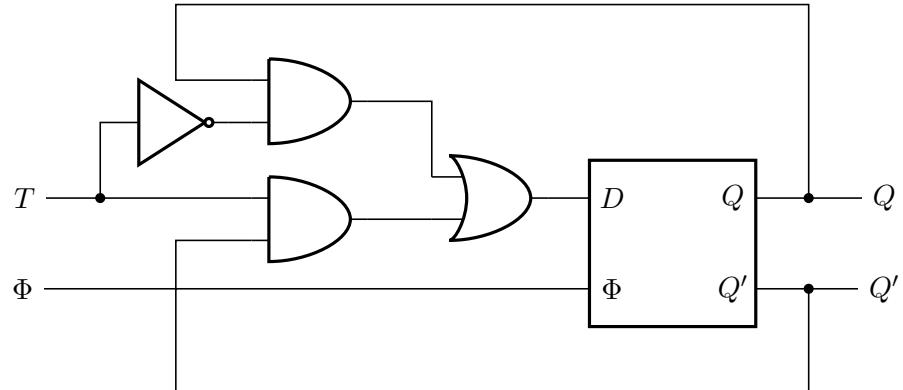
Cambia (toggle) el valor de q cuando $t=1$



T	Q_{next}	\bar{Q}_{next}	Desc.
0	Q	\bar{Q}	hold
1	\bar{Q}	Q	toggle

Implementación

Implementación en base a flip-flop data.



1.4. Diseño modular

1.4.1. Motivación

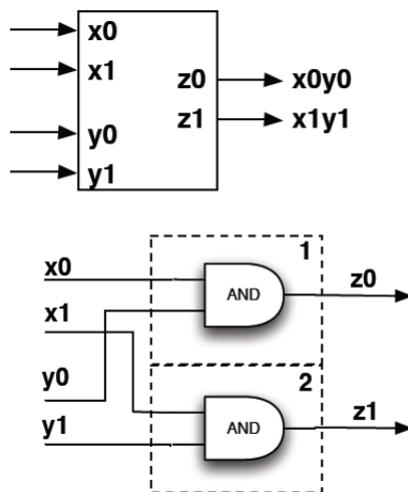
Tabla de verdad: filas= 2^n

$$2^n \left\{ \begin{array}{|c|c|c|c|c|c|} \hline x_2 & x_1 & x_0 & d_1 & d_0 & y \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ \hline \end{array} \right.$$

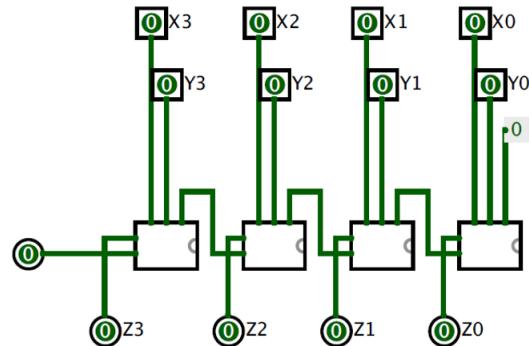
¡Metodología no sirve para muchas entradas!
Mejor acoplar circuitos mas simples \Rightarrow diseño modular

1.4.2. Configuraciones Modulares

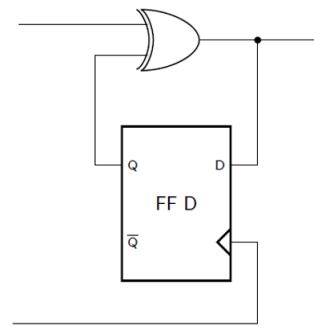
- Paralelo



- Cascada

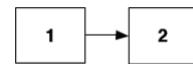


- Serial o secuencial

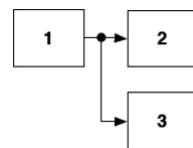


1.4.3. Estrategias de comunicación

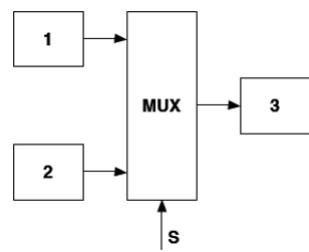
- Punto a punto



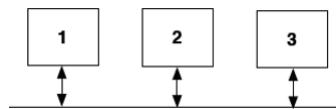
- 1 a N



- N a 1

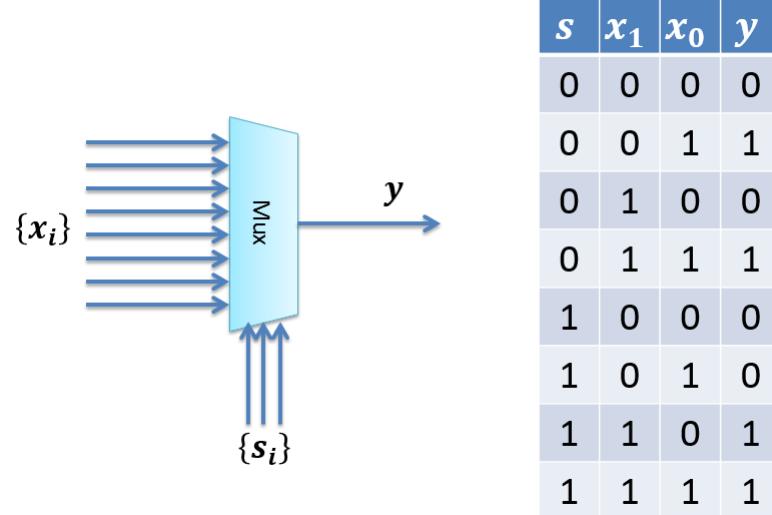


■ Bus



1.4.4. Selección de Entradas o Salidas

Multiplexor (mux)

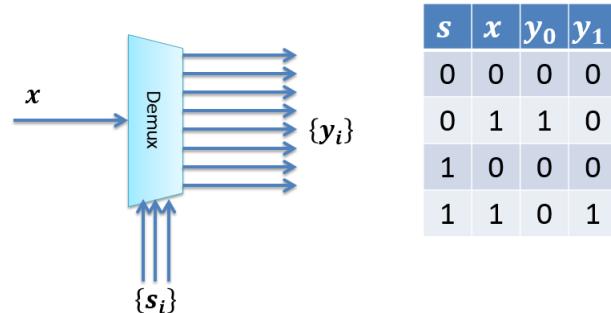


1.4. DISEÑO MODULAR

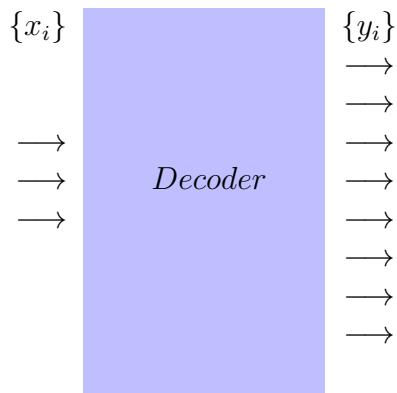
51

La entrada S determina cual de todas las entradas x_i sera usada en la salida:

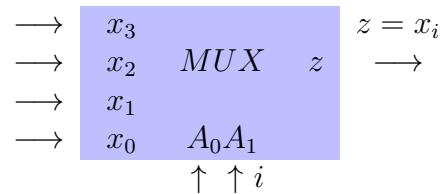
Demultiplexor (demux)



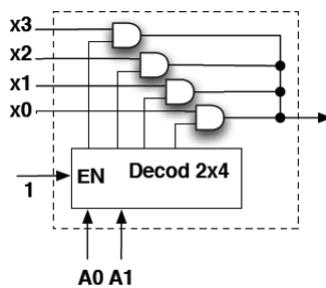
S determina por cual de todas las salidas se retornara la entrada x

Decodificador (decoder)

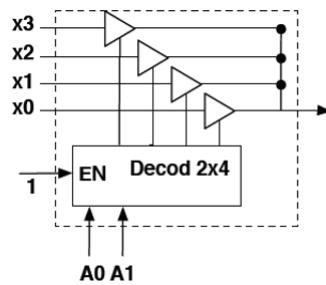
Las entradas determinan en qué posición se ubicará el 1

Implementación multiplexor

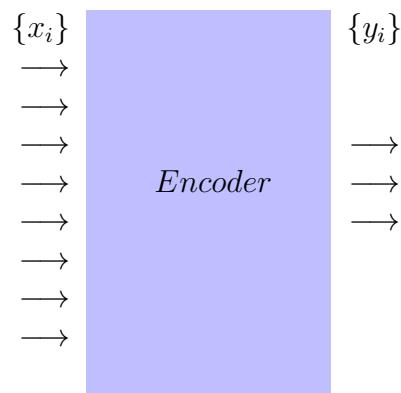
- Usando AND



- Usando compuertas tristate

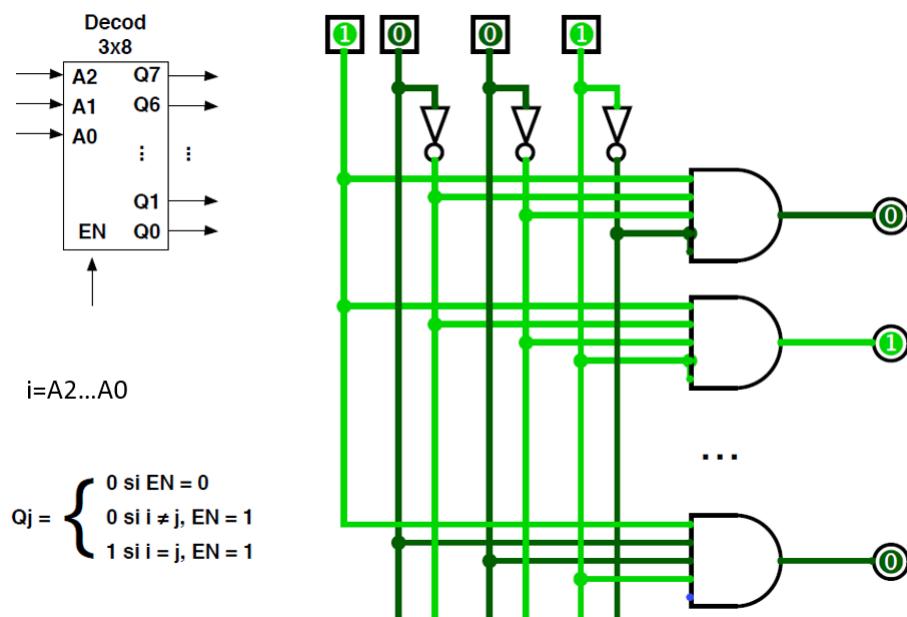


Codificador (encoder)



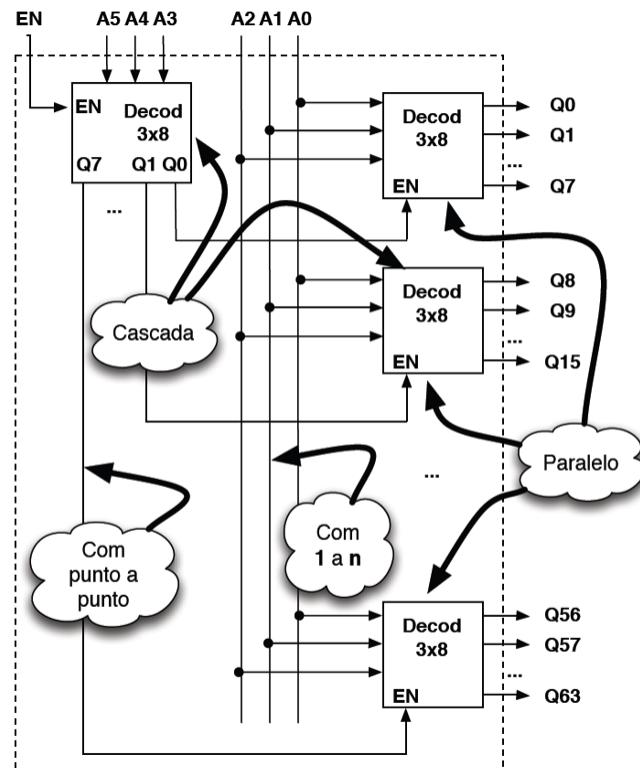
La salida se determina de acuerdo en que posición se encuentra el 1 más significativo (en la entrada)

Implementación decodificador



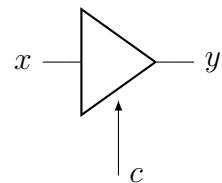
Decodificador 6x64

9 decod 3x8 en cascada y/o paralelo

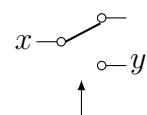


1.4.5. Compuerta tristate

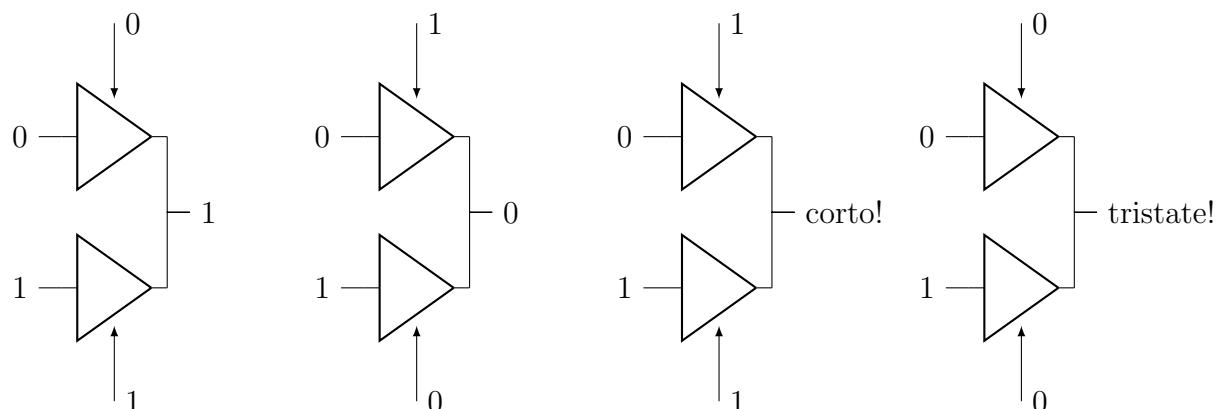
Actúa como un interruptor



Si $c=0$, y se desconecta y no influencia al circuito



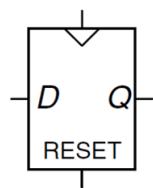
De esta forma se evitan cortocircuitos. Aplicación: buses



1.4.6. Elementos Reseteables

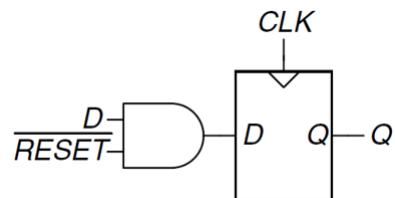
Flip-flop reseteable

Permiten llevar su valor a 0 con una señal de reset



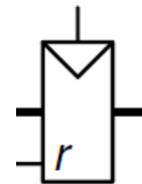
La señal de reset puede ser síncrona o asíncrona

Implementación síncrona



Registro reseteable

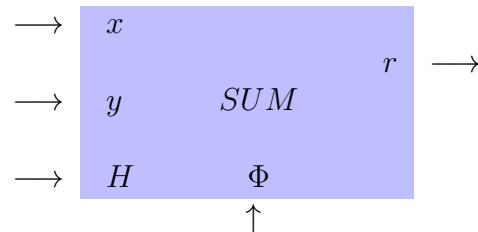
Análogo al flip-flop reseteable pero con n bits



La señal de reset lleva todos los bits a cero.
¿Cómo se puede implementar?

1.4.7. Sumador 1 bit 3 entradas

Sumador serial

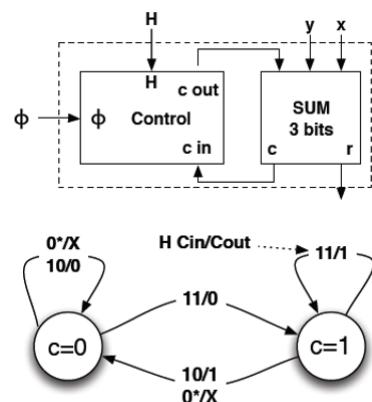
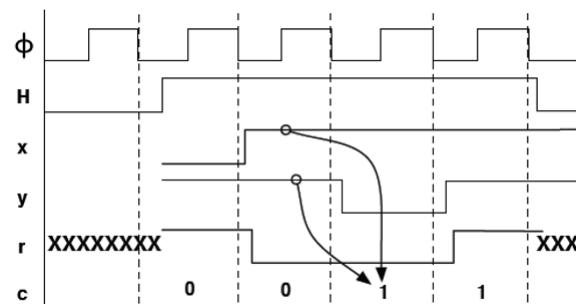


Suma números binario de menor a mayor significancia.

Implementación modular

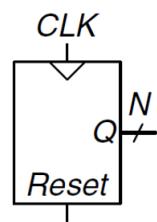
Configuración serial, usa un sumador de 3 bits.

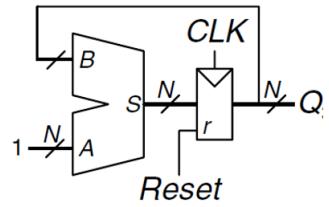
El circuito secuencial recuerda el carry anterior y reinicia con h .



1.4.8. Contador

Cuenta cuantos pulsos se han recibido desde el último reset.

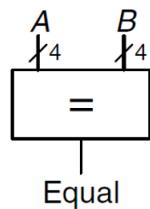


Implementación

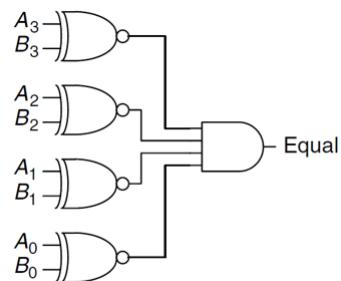
1.4.9. Comparadores

Comparador de igualdad

Compara si dos entradas son iguales bit a bit

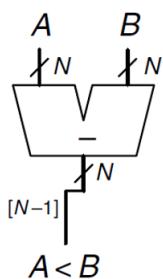


Implementación (4 bits)



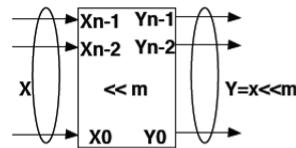
Comparador de magnitud

Compara si una entrada es mayor que otra.



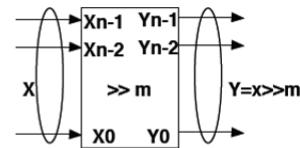
1.4.10. Shifters

Left Shifter:

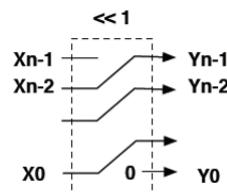


$$Y_j = \begin{cases} 0 & \text{si } j < m \\ X_{j-m} & \text{si } j \geq m \end{cases}$$

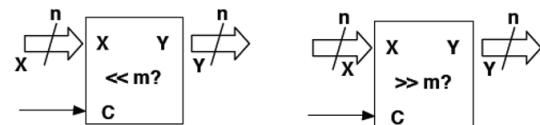
Right Shifter:



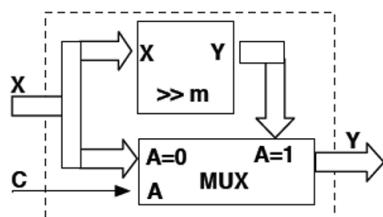
$$Y_j = \begin{cases} 0 & \text{si } j \geq n-m \\ X_{j-m} & \text{si } j < n-m \end{cases}$$



Shifters condicionales



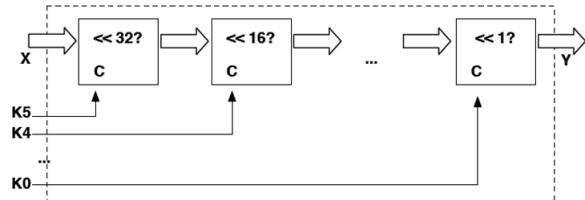
$$Y = \begin{cases} X & \text{si } c = 0 \\ X \ll m & \text{si } c = 1 \end{cases} \quad Y = \begin{cases} X & \text{si } c = 0 \\ X \gg m & \text{si } c = 1 \end{cases}$$



Barrel shifter

Permite seleccionar el número de bits de shifting(m)

Se puede implementar usando conditional shifters con $m = 2^k$



1.4.11. Multiplicadores

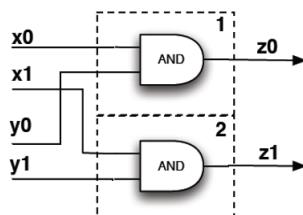
Elementos con múltiples bits

Generalizaciones a varios bits de elementos de un bit.

Implementación: paralelo

Ejemplo:

compuertas, cod-decod, mux-demux.



Observación

Notar que en cualquier base(n):

$$x \ll m = xn^m$$

En decimal:

$$2 \ll 3 = 2000 = 2 * 10^3$$

En binario:

$$11 \ll 2 = 1100 = 11_2 * 2^2$$

Por lo tanto:

$$[x_k \dots x_0] = \sum_{i=1}^n x_i N^i$$

- En decimal: $502 = 5 \times 10^2 + 0 \times 10^1 + 2 \times 10^0$
- En binario: $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

Algoritmo de colegio

Para decimales:

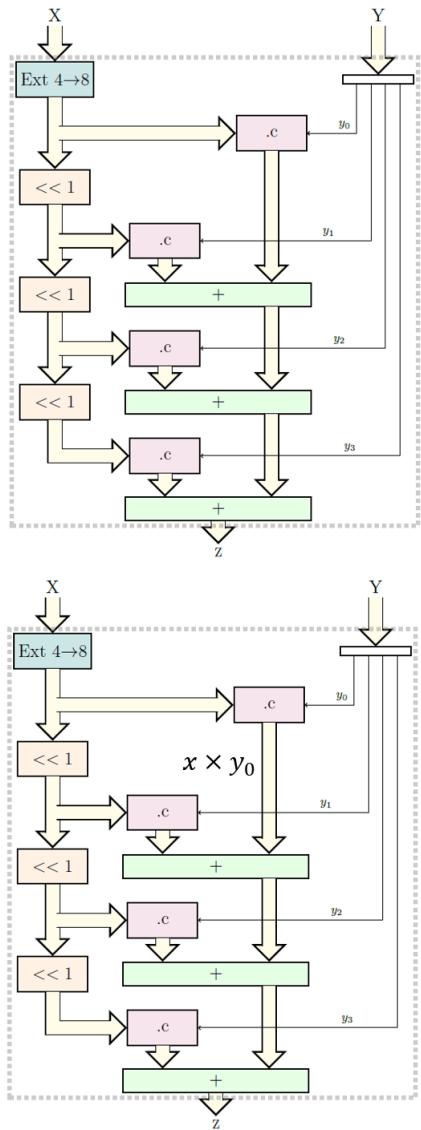
$$\begin{array}{r}
 & 3 & 5 & 4 & 6 & \leftarrow & x \\
 \times & 1 & 4 & 5 & 7 & \leftarrow & y \\
 \hline
 & 2 & 4 & 8 & 2 & 2 & \leftarrow & x \times y_0 \\
 & 1 & 7 & 7 & 3 & 0 & \leftarrow & (x \ll 1) \times y_1 \\
 & 1 & 4 & 1 & 8 & 4 & \leftarrow & (x \ll 2) \times y_2 \\
 & 3 & 5 & 4 & 6 & & \leftarrow & (x \ll 3) \times y_3 \\
 \hline
 & 5 & 1 & 6 & 6 & 5 & 2 & 2
 \end{array}$$

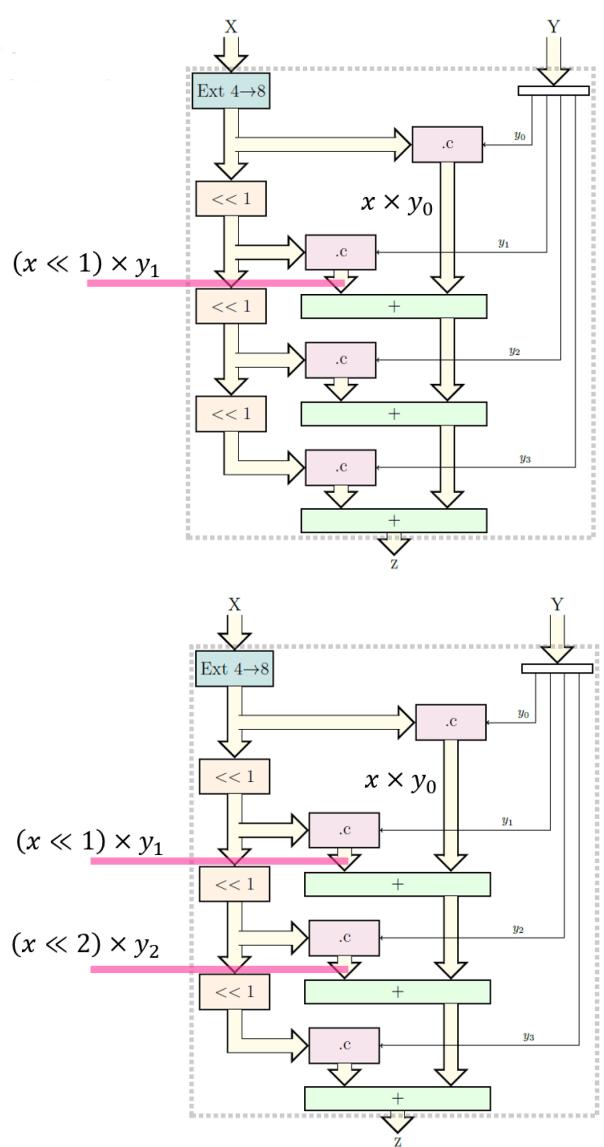
Para binarios:

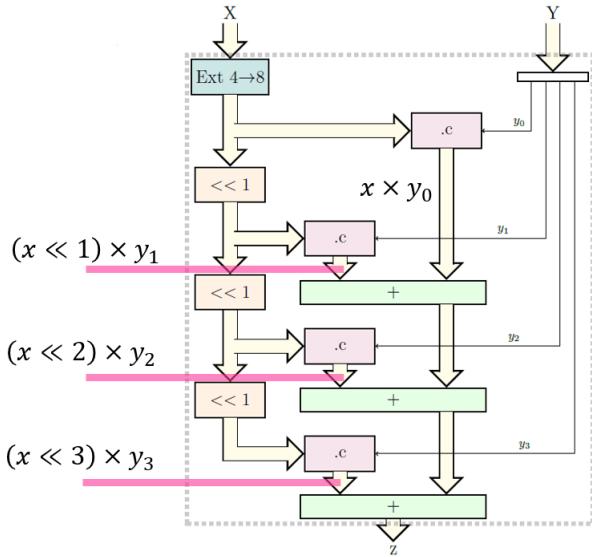
$$\begin{array}{r}
 & 1 & 1 & 1 & 0 & \leftarrow & x \\
 \times & 1 & 0 & 1 & 1 & \leftarrow & y \\
 \hline
 & 1 & 1 & 1 & 0 & \leftarrow & x \times y_0 \\
 & 1 & 1 & 1 & 0 & \leftarrow & (x \ll 1) \times y_1 \\
 & 0 & 0 & 0 & 0 & \leftarrow & (x \ll 2) \times y_2 \\
 & 1 & 1 & 1 & 0 & \leftarrow & (x \ll 3) \times y_3 \\
 \hline
 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0
 \end{array}$$

En binario, el multiplicar por un bit es lo mismo que hacer AND bit a bit

Multiplicador 4 bits







Implementación en serie

```

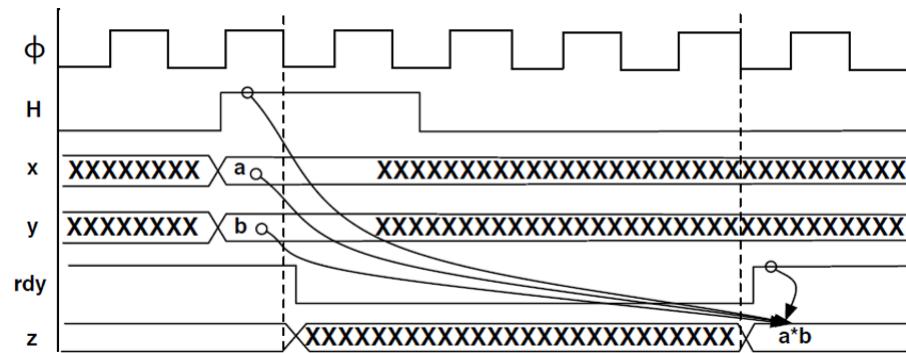
Mult (x,y de 32 bits) {
    sea Rx de 64 bits = Ext 32->64 (x);
    sea Ry de 32 bits = y;
    sea Rz de 64 bits = 0;

    while (Ry != 0) {
        if(Ry[0] != 0)
            Rz += Rx;
        Rx := Rx << 1;
        Ry := Ry >> 1;
    }

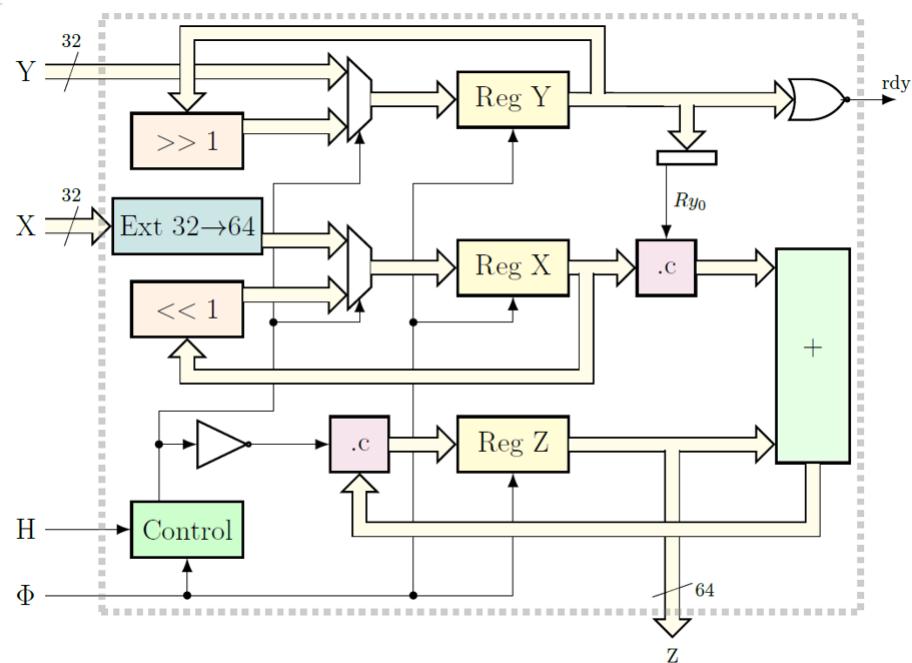
    return Rz;
}

```

Diagrama de tiempo



Implementación



Capítulo 2

Arquitectura lógica de un computador

2.1. Representación numérica

Palabras (words) de 8, 16, 32 o 64 bits

Representan:

- Enteros (con y sin signo)
- Carácteres
- Números reales

Notación:

$$x = |x_{n-1}x_{n-2}\dots x_1x_0|$$

2.1.1. Notación hexadecimal

Usa base 16 (4 bits por dígito). Normalmente se agrega el prefijo 0x

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

Ejemplo:

$$\begin{aligned}
 45_{10} &= \textcolor{red}{101101}_2 \\
 &= 2D_{16} \\
 &\text{o } 0x2D
 \end{aligned}$$

2.1.2. Enteros

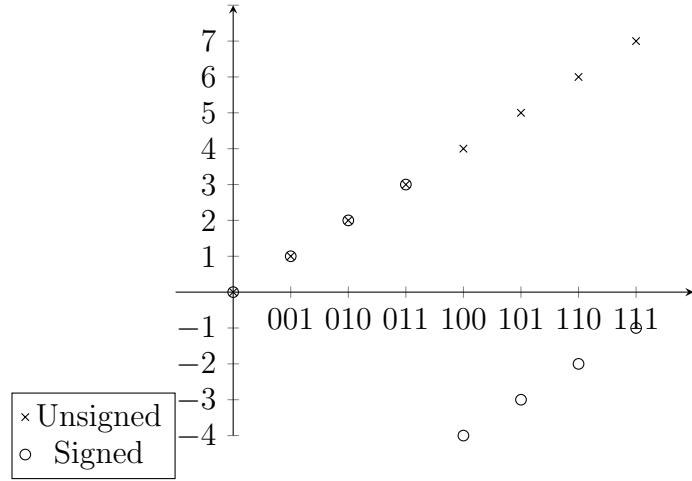
- Sin signo (unsigned)

$$[[x]]_u = \sum_{i=0}^{n-1} x_i 2^i$$

- Con signo (signed)

$$[[x]]_s = \begin{cases} [[x]]_u & \neg x_{n-1} \\ [[x]]_u - 2^n & x_{n-1} \end{cases}$$

2.1.3. Comparación entre signed y unsigned



2.1.4. Obtención de $[[x]]_s$

Queremos representar un número d con la siguiente notación:

$$[[x^s]]_s$$

Tenemos que:

$$\begin{aligned} |d| &= [[x^u]]_u \\ \text{Si } d \geq 0, \quad d &= [[x^u]]_u \\ x^s &= x^u \end{aligned}$$

Ejemplo:

Ej (4 bits) :

Para representar $d = 5$:

$x^s = x^u = 0101_2$

$$\begin{aligned} \text{Si } d < 0, -d &= [[x^u]]_u \\ x^s &= \overline{x^u} + 1 \end{aligned}$$

Ejemplo:

$$\begin{aligned} \text{Ej (4 bits):} \\ \text{Para representar } d = -5: \\ x^s &= \overline{x^u} + 1 \\ &= \overline{0101} + 1 \\ &= 1010_2 + 1 \\ &= 1011_2 \end{aligned}$$

2.1.5. Interpretación de $[[x]]_s$

Queremos saber cuál número d está escrito en notación $[[x]]_s$.

Fórmula 1

$$[[x]]_s = \begin{cases} [[x]]_u & \neg x_{n-1} \\ [[x]]_u - 2^n & x_{n-1} \end{cases}$$

$$\begin{aligned} \text{Ej (4 bits):} \\ [[0111]]_s \\ = [[0111]]_u \quad (x_{n-1} = 0) \\ = 7_{10} \end{aligned}$$

$$\begin{aligned} \text{Ej (4 bits):} \\ [[1111]]_s \\ = [[1111]]_u - 16_{10} \quad (x_{n-1} = 1) \\ = 15_{10} - 16_{10} = -1_{10} \end{aligned}$$

Fórmula 2

$$[[x]]_s = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Ej (4 bits):

$$\begin{aligned} & [[0111]]_s \\ & = [[111]]_u - 0 \times 8_{10} \quad (x_{n-1} = 0) \\ & = 7_{10} \end{aligned}$$

Ej (4 bits):

$$\begin{aligned} & [[1111]]_s \\ & = [[111]]_u - 8_{10} \quad (x_{n-1} = 1) \\ & = 7_{10} - 8_{10} = -1_{10} \end{aligned}$$

Complemento de 2

Caso 1:

$$\begin{aligned} \text{Si } x_{n-1} = 0 \Rightarrow d \geq 0 \\ \Rightarrow [[x]]_s = [[x]]_u \end{aligned}$$

Ej (4 bits):

$$\begin{aligned} & [[0111]]_s \\ & = [[0111]]_u \quad (x_{n-1} = 0) \\ & = 7_{10} \end{aligned}$$

Caso 2:

$$\begin{aligned} \text{Si } x_{n-1} = 1 &\Rightarrow d < 0 \\ \Rightarrow [[x]]_s &= -([[\bar{x}]]_u + 1) \end{aligned}$$

Ej (4 bits):

$$\begin{aligned} &[[1111]]_s \\ &= -([[1111]]_u + 1) \quad (x_{n-1} = 1) \\ &= -([[0000]]_u + 1) = -1_{10} \end{aligned}$$

2.1.6. Suma de enteros

$$\text{¿} [[x \oplus y]]_u = [[x]]_u + [[y]]_u ?$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \rightarrow_u 5 \\ \oplus \ 0 \ 0 \ 1 \ 1 \rightarrow_u + 3 \\ \hline 1 \ 0 \ 0 \ 0 \rightarrow_u \checkmark \ 8 \end{array}$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \rightarrow_u 5 \\ \oplus \ 1 \ 1 \ 1 \ 1 \rightarrow_u + 15 \\ \hline [1] \ 0 \ 1 \ 0 \ 0 \rightarrow_u \times \times \ 4 \end{array}$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \rightarrow_s 5 \\ \oplus \ 0 \ 0 \ 1 \ 1 \rightarrow_s + 3 \\ \hline 1 \ 0 \ 0 \ 0 \rightarrow_s \times \times \ -4 \end{array}$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \rightarrow_s 5 \\ \oplus \ 1 \ 1 \ 1 \ 1 \rightarrow_s + -1 \\ \hline 0 \ 1 \ 0 \ 0 \rightarrow_s \checkmark \ 4 \end{array}$$

$$\begin{aligned} [[x \oplus y]]_u &= \text{mód } 2^n = [[x]]_u + [[y]]_u \text{ y } [[x \oplus y]]_s = \text{mód } 2^n = [[x]]_s + [[y]]_s \text{ con} \\ a &= \text{mód } m b \Leftrightarrow a \text{ mód } m = b \text{ mód } m \end{aligned}$$

2.1.7. Resta de enteros

$$x \ominus y = x \oplus \neg y \oplus 1$$

$$[[x \ominus y]]_u = \text{mód } 2^n [[x]]_u - [[y]]_u \text{ y } [[x \ominus y]]_s = \text{mód } 2^n [[x]]_s - [[y]]_s$$

$$\begin{array}{r} \begin{array}{rrrrrr} 1 & 1 & 0 & 1 & \rightarrow_u & 13 \\ \ominus & 0 & 1 & 0 & \rightarrow_u & - \\ \hline 1 & 0 & 0 & 0 & \rightarrow_u & \checkmark \end{array} & \begin{array}{rrrrrr} 1 & 1 & 0 & 1 & \rightarrow_s & -3 \\ \ominus & 0 & 1 & 0 & \rightarrow_s & - \\ \hline 1 & 0 & 0 & 0 & \rightarrow_s & \checkmark \end{array} & -8 \end{array}$$

				1
	1	1	0	1
⊕	1	0	1	0
	1	0	0	0

2.1.8. Punto flotante

Usa el mismo principio de la notación científica.

$$\begin{aligned} [[x]]_f &= \sigma \times m \times 2^e \\ -\sigma (\pm 1) &\text{ indica el signo} \\ -m (1, \dots) &\text{ indica la mantisa (dígitos más significativos)} \\ -e &\text{ indica el exponente} \end{aligned}$$

La fracción es la parte fraccional de la mantisa.

$$f = m - 1$$

Para codificar en binario

- σ es codificado en un bit, s :

$$\sigma = (-1)^s$$

- m es codificado en N_f bits, $\{f_i\}$:

$$f = \sum_{i=0}^{N_f-1} f_i 2^{i-N_f}$$

- e es codificado en N_e bits, $\{e_i\}$:

$$e = e_{\text{offset}} + \sum_{i=0}^{N_e-1} e_i 2^i$$

Precisión simple ($e_{\text{offset}} = -127$)

x_{31}	x_{30}, \dots, x_{23}	x_{22}, \dots, x_0
s	e_7, \dots, e_0	f_{22}, \dots, f_0
Signo (1)	Exponente (8)	Fracción (23)

Precisión doble ($e_{\text{offset}} = -1023$)

x_{63}	x_{62}, \dots, x_{52}	x_{51}, \dots, x_0
s	e_{10}, \dots, e_0	f_{51}, \dots, f_0
Signo (1)	Exponente (11)	Fracción (52)

2.1.9. Casos especiales

2.1.10. Codificacion BCD

Binary coded decimal.

Cada cuatro bits representan un dígito decimal.

$$\begin{aligned} 416_{10} &= 4 \times 10^2 + 1 \times 10^1 + 6 \times 10^0 \\ &= [[0100\textcolor{red}{0001}\textcolor{blue}{0110}]]_{BCD} \end{aligned}$$

2.1.11. Reducción

Se eliminan los bits mas significativos

$$Trunc^{n \rightarrow m}(|X_{n-1} \dots X_{m-1} \dots X_0|) = |X_{m-1} \dots X_0|$$

Si $[[x]]_{u/s}$ puede ser representado en m bits

$$[[x]]_{u/s} = [[Trunc^{n \rightarrow m}(x)]]_{u/s}$$

Ejemplo:

$$\begin{array}{rcl} 0 & 0 & 1 & 0 & 1 \rightarrow_u 5 & 0 & 0 & 1 & 0 & 1 \rightarrow_s 5 \\ & & & & \downarrow Trunc_{u^5 \rightarrow 4} & & & & & \downarrow Trunc_{u^5 \rightarrow 4} \\ 0 & 1 & 0 & 1 & \rightarrow_u 5 & 0 & 1 & 0 & 1 \rightarrow_s 5 & & \end{array}$$

$$\begin{array}{rcl} 1 & 1 & 0 & 1 & 0 \rightarrow_u 26 & 1 & 1 & 0 & 1 & 0 \rightarrow_s -6 \\ & & & & \downarrow Trunc_{u^5 \rightarrow 4} & & & & & \downarrow Trunc_{u^5 \rightarrow 4} \\ 1 & 0 & 1 & 0 & \rightarrow_u \not{\emptyset} & 1 & 0 & 1 & 0 \rightarrow_s -6 & & \end{array}$$

2.1.12. Extensión sin signo

Se agregan 0s a la izquierda

$$Ext_{u^{m \rightarrow n}}(|X_{m-1} \dots X_0|) = |0_1 \dots 0_{n-m} x_{m-1} \dots x_0|$$

Se cumple:

$$[[Ext_{u^{m \rightarrow n}}(x)]]_u = [[x]]_u$$

Ej:

$$\begin{array}{rcl} 1 & 1 & 0 & 1 \rightarrow_u 13 & 1 & 1 & 0 & 1 \rightarrow_s -3 \\ & & & \downarrow Ext_{u^{4 \rightarrow 5}} & & & \downarrow Ext_{u^{4 \rightarrow 5}} & \\ 0 & 1 & 1 & 0 & 1 \rightarrow_u 13 & 0 & 1 & 1 & 0 & 1 \rightarrow_s \not{\exists} \end{array}$$

2.1.13. Extensión con signo

Se repite el bit de signo ($n - m$ veces)

$$Ext_{s^{m \rightarrow n}}(|x_{m-1} \dots x_0|) = |x_{m-1} \dots x_{m-1} x_{m-1} \dots x_0|$$

Se cumple:

$$[[Ext_{s^{m \rightarrow n}}(x)]]_s = [[x]]_s$$

Ej:

$$\begin{array}{rcl} 1 & 1 & 0 & 1 \rightarrow_u 13 & 1 & 1 & 0 & 1 \rightarrow_s -3 \\ & & & \downarrow Ext_{u^{4 \rightarrow 5}}^s & & & \downarrow Ext_{u^{4 \rightarrow 5}}^s & \\ 1 & 1 & 1 & 0 & 1 \rightarrow_u \not{\exists} & 1 & 1 & 1 & 0 & 1 \rightarrow_s -3 \end{array}$$

2.2. Arquitectura lógica

Es la interfaz entre los compiladores y el *hardware*. También se le conoce como Instruction set architecture (ISA).

Especifica:

- El conjunto de instrucciones
- Las direcciones de memoria, registros y modos de direccionamiento
- Otros, como tipos de datos, I/O, interrupciones, manejo de excepciones, etc.

2.2.1. Tipos de ISA

- CISC: complex instruction set computer
- RISC: reduce instruction set computer
- VLIW: very long instruction word

CISC

- Complex instruction set computer
- Implementaciones: IBM System/360, Motorola 68k, x86, etc

RISC

- Reduced instruction set computer
- Set de instrucciones minimalista
- Instrucciones ortogonales
- Meta: 1 instrucción por ciclo de reloj
- Implementaciones: SPARC, MIPS, ARM, etc

VLIW

- Very long instruction word
- Se usa una instrucción muy larga
- Se pueden ejecutar muchas operaciones por instrucción
- El compilador tiene más trabajo
- El hardware es más simple
- Principales implementaciones actuales: DSP

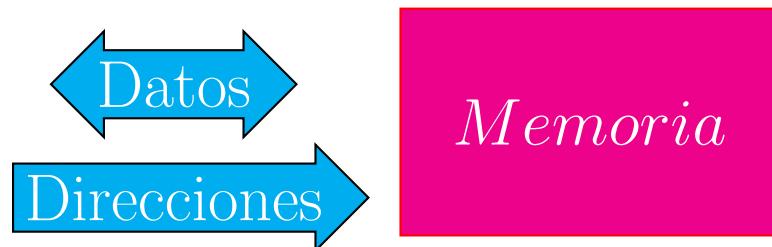
2.2.2. CPU

- Procesa instrucciones
- Recibe datos como parámetros
- Entrega datos como resultados
- Contiene registros

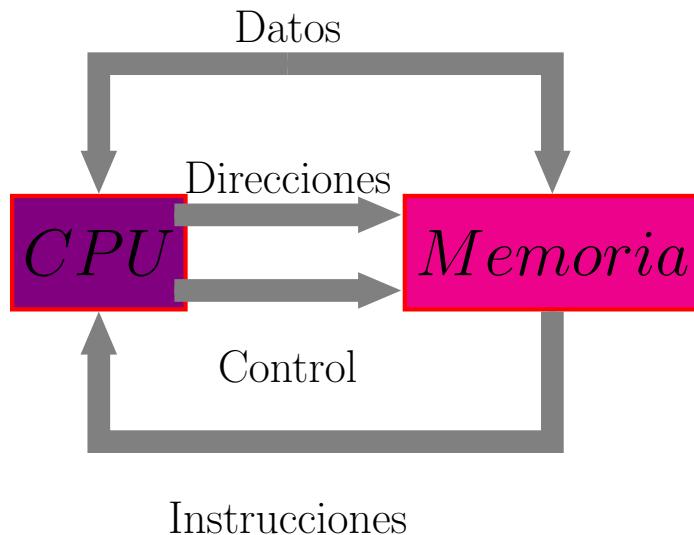


2.2.3. Memoria

- Contiene datos guardados en una tabla
- Cada dato tiene una dirección asociada
- Datos pueden ser leídos o escritos
- Los datos pueden ser en particular instrucciones



2.2.4. Principio funcionamiento computador



2.2.5. Lenguaje de máquina vs Assembler

La CPU interpreta las instrucciones, las cuales están en lenguaje de máquina:

- Binario
- OPCode + Argumentos

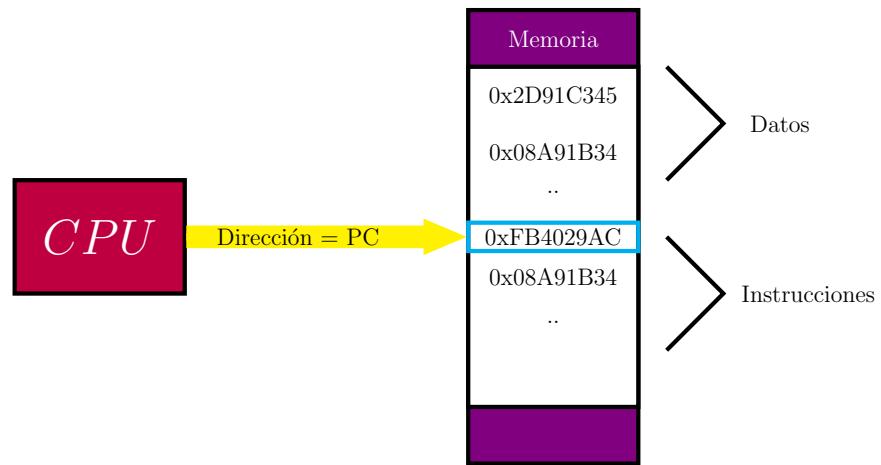
Para generar el lenguaje de máquina se utiliza Assembler:

- Legible por un ser humano
- Instrucciones y argumentos nemotécnicos

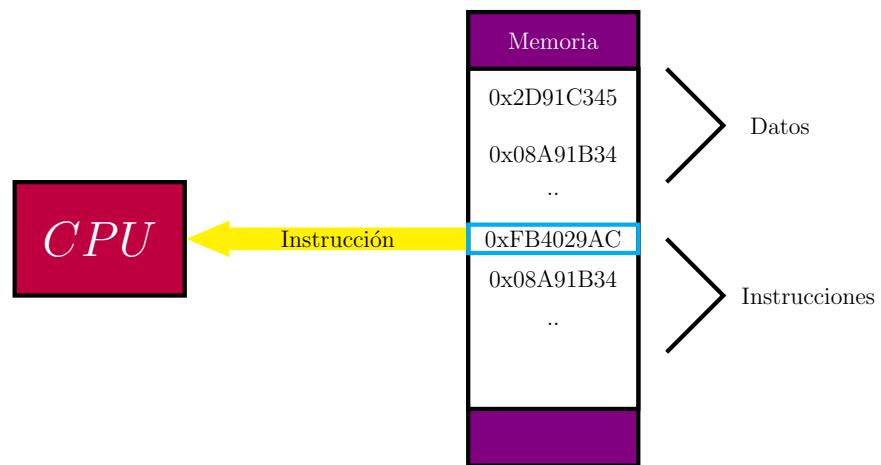
2.2.6. Programa almacenado en memoria

Los programas se guardan en memoria.

La CPU lee las instrucciones (fetch).



Para saber qué dirección leer, la CPU guarda el program counter (PC).



2.2.7. Tipos de instrucciones

- Lógicas/aritméticas
- Saltos (mueven PC)
- Lectura/escritura

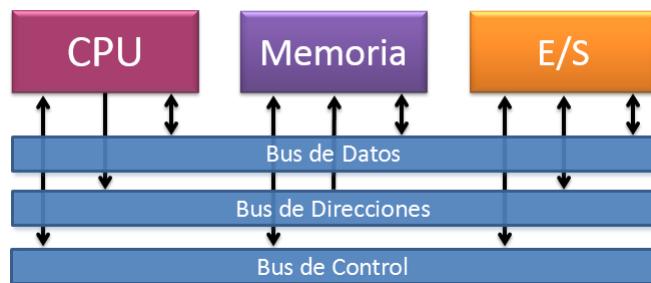
2.2.8. Tipos de argumentos

- Registros
- Inmediatos
- Direcciones de memoria (CISC)

2.2.9. Arquitectura Von Neumann

Los datos y las instrucciones usan el mismo BUS (de datos).

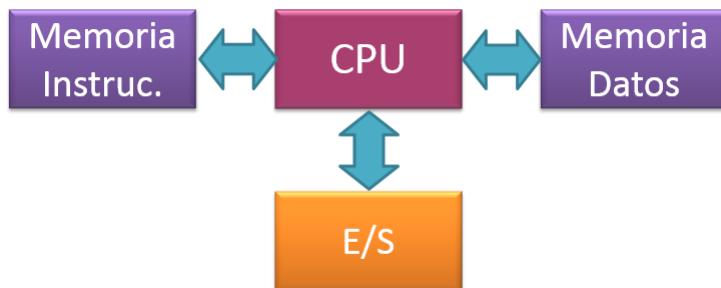
¡Potencial cuello de botella!



2.2.10. Arquitectura Harvard

Datos e instrucciones físicamente separadas.

- Almacenamiento
- Bus



2.3. Assembler x86

x86 es una arquitectura (no una CPU).

Historia:

Año	Bits	Nombre	Instrucciones
1978	16	Intel 8086	IA-16
1985	32	Intel 80386	IA-32
2003	64	AMD Athlon 64	AMD64 (compatible con IA-32 pero no con IA-64)

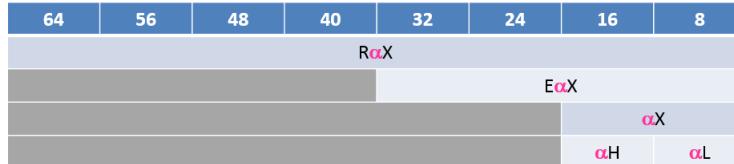
- CISC
- Little endian

2.3.1. Registros

- Memoria interna de la CPU
- Muy rápido acceso
- Casi todas las operaciones los usan

2.3.2. Registros de propósito general

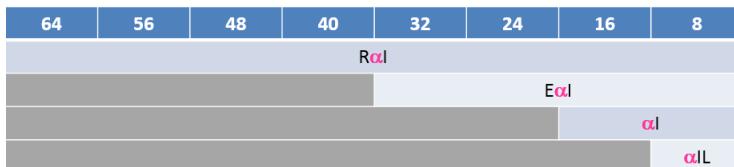
- $\alpha \in \{A, B, C, D\}$



- AX/EAX/RAX: Acumulador
- BX/EBX/RBX: Índice Base (Arreglos)
- CX/ECX/RCX: Contador
- DX/EDX/RDX: Datos/general

2.3.3. Registros de índices

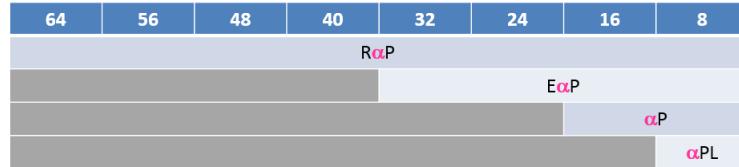
- $\alpha \in \{S, D\}$



- SI/ESI/RSI: Índice de Origen para operaciones de strings
- DI/EDI/RDI: Índice de Destino para operaciones de strings

2.3.4. Registros de punteros

- $\alpha \in \{S, B\}$



- SP/ESP/RSP: Puntero a la parte superior de la pila (pila invertida).
- BP/EBP/RBP: Puntero al frame actual de la pila, su base (pila invertida).

2.3.5. Registros de puntero de instrucción



- IP/EIP/RIP: Instruction pointer. Tiene la dirección a la instrucción ejecutada actualmente, llamada ***program counter***.

2.3.6. Assembler

Diferentes sintaxis:

- GNU Assembler (GAS) = sintaxis AT&T
- Sintaxis Intel

Al operar direcciones de memoria, se debe especificar su tamaño con un sufijo:

Sufijo	Nombre	Número de bits
b	byte	8
w	word	16
l	long	32

2.3.7. Sintaxis

Inmediatos

Llevan el prefijo \$

Ejemplo: \$65 \$0xD4

Registros

Llevan el prefijo %

Ejemplo %exa %esi

Comentarios

Empiezan con #

2.3.8. Instrucciones aritméticas

<code>add src, dest</code>	Suma/resta contenidos de src a/de dest
<code>adc src, dest</code>	Suma, seteando un bit carry
<code>sbb src, dest</code>	Resta usando bit carry
<code>inc dest</code>	<code>add 1, dest</code> más rápido
<code>dec dest</code>	<code>sub 1, dest</code> más rápido
<code>neg dest</code>	$-1 * \text{dest}$

Ejemplo:

```

inc  %eax      # agrega 1 al valor en el registro eax
incl %eax      # idem
add  $42, %bh   # agrega 42 al valor en el registro bh
addb $0x2A, %bh # idem
sbbw %ax, %bx   # sustraen ax de bx, se guarda en bx

```

2.3.9. Multiplicación-división

<code>mul src</code>	Multiplicación sin/con signo de <code>src</code> por el valor en el registro correspondiente (ver abajo)
<code>imul src</code>	
<code>div src</code>	División con/sin signo del valor en el registro correspondiente por <code>src</code> (ver abajo). Si <code>quot</code> no cabe, interrupción de overflow aritmético
<code>idiv src</code>	

<code>mul arg / imul arg</code>			
<code>op_size</code>	<code>arg2</code>	<code>dest_high</code>	<code>dest_low</code>
8 bits	AL	AH	AL
16 bits	AX	DX	AX
32 bits	EAX	EDX	EAX

<code>div arg / idiv arg</code>			
<code>op_size</code>	<code>dividend</code>	<code>remain</code>	<code>quot</code>
8 bits	AX	AH	AL
16 bits	DX, AX	DX	AX
32 bits	EDX, EAX	EDX	EAX

2.3.10. Copia información (mov)

<code>mov src, dest</code>	Copia data desde <code>src</code> a <code>dest</code>
----------------------------	---

Ejemplo:

```
    movl $42, %eax  
    movb %ah, %bl
```

2.3.11. Copiar dirección de memoria (lea)

<code>lea mem, reg</code>	Carga la dirección de la ubicación de la memoria <code>mem</code> a <code>reg</code>
---------------------------	--

Ejemplo:

```
    lea foo, %eax      # copia la dirección de foo en eax
```

2.3.12. Direccionamiento indirecto

La versión más simple es usar la dirección de memoria almacenada en un registro. Para ello, se encierra el registro entre paréntesis.

Ejemplo:

```
    mov %ebx, (%eax)      # guarda el valor de ebx en la  
                           # dirección de memoria apuntada  
                           # por eax
```

2.3.13. Extensión

movs src, dest	Extiende considerando el signo
movz src, dest	Extiende con 0s

Reciben dos sufijos de tamaño: origen y destino.

Ejemplo:

```
movb    $-204, %al
movsbw  %al, %bx
```

2.3.14. Comparaciones

comp arg1, arg2	Compara arg1 con arg2, determinando si:
	<ul style="list-style-type: none"> ▪ arg1 > arg2 (greater) ▪ arg1 == arg2 (equals) ▪ arg1 < arg2 (lesser)
test arg1, arg2	AND bit a bit (bitwise)

Setean el registro EFLAGS en función del resultado de la comparación.

2.3.15. Saltos condicionales

Hacen saltar la ejecución a otro punto del programa en función del resultado de una comparación anterior.

je loc	Jump on equal	jnz loc	Jump on not zero
jne loc	Jump on not equal	jz loc	Jump on zero
jg loc	Jump on greater	ja	jg for signed numbers
jge loc	Jump on greater or equal	jae	jge for signed numbers
jl loc	Jump on lesser	jb	jl for signed numbers
jle	Jump on lesser or equal	jbe	jle for signed numbers
jo loc	Jump on overflow	jmp	Jump always

2.3.16. Implementación if

Usando comparaciones y saltos condicionales se puede implementar if.
Ejemplo:

```

cmp %ax, %bx
jg label1      # salta a label1 si ax > bx
test $1, %ebx
jnz sig_u:     # salta a sig_u si bit menos significativo de ebx es 1
label1:
test %ah, %ah
jz label3      # salta a label3 si ah = 0 (ah && ah = ah)
sig_u:
cmp %al, %ah
jz l_42        # salta a l_42 si ah = al

```

2.3.17. Operaciones lógicas

Comando	Descripción
and src, dest	AND bit a bit
or src, dest	OR bit a bit
xor src, dest	XOR bit a bit
not dest	Inversión bit a bit

2.3.18. Shift y rotate

Comando	Descripción
ror dest	rotar a la derecha 1 bit
rol dest	rotar a la izquierda 1 bit
shr dest	shift de 1 bit a la derecha, llenar con 0
shl dest	shift de 1 bit a la izquierda, llenar con 0
sar dest	shift de 1 bit a la derecha, llenar con el bit signo
sal dest	shift de 1 bit a la izquierda, llenar con 0
scr arg	shr con el bit shifteado a flag carry
scl arg	shl con el bit shifteado a flag carry

2.3.19. Direccionamiento directo (o desplazamiento)

Se da como argumento directamente la dirección de memoria.

Ejemplo:

```
.data          #sección data, para declarar variables globales
foo:          #variables llamada 'foo'
    .int 2      #es un entero con valor '2'

[ . . . ]

.text          #sección de texto, el código va aquí
.global _main  #una función global llamada '_main'
_main:
[ . . . ]

    movl $6, foo  #copia '6' a la ubicación
                    #de la memoria de 'foo'
```

2.3.20. Direccionamiento indirecto

Permite acceder a un valor a través de su dirección de memoria (guardada en un registro).

disp(base, index, scale)			
keyword	meaning	examples	dereferencing what
disp	displacement	foo(,1)	foo pointer (not in a register)
base	base register	(%eax) -4(%ebp)	address in eax address 4 bytes above stack base
index	index register	foo(,%eax)	foo array index eax, if foo contains 1-byte sized data
scale	constant 1, 2, 4 or 8	foo(,%eax,4) 0(%eax,%ebx,2)	foo array index eax, if foo contains 4-byte sized data index ebx of array starting in eax, if contains 2-byte sized data

2.3.21. Ejemplos

Dado los enteros:

Nombre	Posición
i	ebp - 4
*a	ebp - 8
r	ebp - 12

Calcular: $r = (-a[i]) \cdot 10$

```

movl -4(%ebp), %eax      #get i
movl -8(%ebp), %ebx      #get a
movl 0(%ebx,%eax,4), %eax #get a[i]
negl %eax                #-a[i]
imull $10, %eax          #(-a[i]) *10, truncate to 32 bits
movl %eax,-12(%ebp)      #copy to r

```

También se puede usar en operaciones aritméticas

```
inc -4(%ebp)      #i++
```

Otro ejemplo:

- Dado: int *a, n , s = 0, i = 0
- escriba while (i<n) s+= a[i++]
- asuma *a en eax, s en ecx, i en esi

```

movl $0, %ecx      #s = 0
movl $0, %esi      #i = 0
L1:
cmp %ebx, %esi      #i ? n
jge L2              #if (i >= n), salir
movl (%eax, %esi, 4), %edx #get a[i] (en %edx)
addl %edx, %ecx      #s += a[i]
jmp L1              #volver a iniciar el loop
L2:
[ . . . ]

```

2.4. Pila

La CPU mantiene una pila (stack) en memoria.

El registro `esp` apunta al tope de la pila.

La pila crece hacia abajo.

Se pueden agregar (apilar) o sacar (desapilar) valores de la pila

<code>push src</code>	Push valor a la pila, decrementa <code>esp</code>
<code>pop dest</code>	Pop stack a <code>dest</code> , incrementa <code>esp</code>

2.4.1. Llamadas a subrutinas

El registro instruction pointer (IP) apunta a la instrucción que se está ejecutando. Se puede manipular IP para hacer llamadas a subrutinas:

<code>call lab</code>	IP then <code>jmp lab</code>
<code>ret</code>	<code>pop a IP</code>

2.4.2. Convención del llamador

El llamado debe preservar: `ebx`, `esi`, `edi`, `esp`, `ebp`.

Puede cambiar: `eax`, `ecx`, `edx`.

2.4.3. Manejo stack en llamadas

1. El llamador apila argumentos en la pila en orden inverso y llama con call (lo que apila el Instruction Pointer)
2. Se encadenan los registros de activación `ebp`
3. Se resguardan registros que no deben ser modificados según la convención del llamador
4. Se agranda el registro de activación para variables locales
5. Se calcula el valor a retornar
6. El valor retornado queda en `eax`
7. Se botan las variables locales y se restituyen los registros guardados
8. El llamado retorna usando `ret` (lo que desapila el Instruction Pointer)
9. El llamador desapila los argumentos

Ejemplo:

<code>res = proc(arg1, arg2)</code>	<code>int proc(int p1, int p2){</code> <code>int local</code> <code>[. . .]</code> <code>return local;</code> <code>}</code>
<code>pushl arg2 # paso 1</code> <code>pushl arg1 # paso 1</code> <code>call proc # paso 1</code> <code>movl %eax, res</code> <code>addl \$8, %esp #paso 9</code> <code>#leave =</code> <code>#movl %ebp, %esp</code> <code>#popl %ebp</code>	<code>.global proc</code> <code>proc:</code> <code>pushl %ebp # paso 2</code> <code>movl %esp, %ebp # paso 2</code> <code>pushl %edi # paso 3</code> <code>pushl %esi # paso 3</code> <code>pushl %ebx # paso 3</code> <code>subl \$4, %esp # paso 4</code> <code>[. . .] # paso 5</code> <code>movl -16(%ebp), %eax # paso 6</code> <code>addl \$4, %esp # paso 7</code> <code>popl %ebx # paso 7</code> <code>popl %esi # paso 7</code> <code>popl %edi # paso 7</code> <code>leave # paso 7</code> <code>ret # paso 8</code>

2.4.4. Búffer overflow

IP de retorno esta guardado en `ebp+4`. Si se sobreescribe ese valor la función no retornará a la función llamadora.

<code>ebp-16</code>	Variables locales
<code>ebp-12</code>	<code>ebx</code> llamador
<code>ebp-8</code>	<code>esi</code> llamador
<code>ebp-4</code>	<code>edi</code> llamador
<code>ebp</code>	<code>ebp</code> llamador
<code>ebp+4</code>	IP (dirección de retorno)
	<code>arg1</code>
	...
	<code>argn</code>

Ejemplos

Ejemplo 1:

Las siguientes funciones pueden causar búffer overflow:

```
int getUserId() { .global getUserId
    char BUF(7); getUserId:
    [ get user id as string,     pushl %ebp      #p2
    put in BUF ]               movl %esp, %ebp  #p2
    return [convert BUF];      subl $8, %esp   #p4
}                                [ get user input ]
                                  [ convert to int, put in eax ]
                                  addl $12, %esp    #p7
                                  ret           #p6
```

Ejemplo 2:

Si el usuario escribe “12345678....***”

Stack	Contiene	Debería contener
esp+12	"****" = 0x2A2A2A2A	ebp+4= IP del llamador (jmp)
esp+8	"...." = 0x2E2E2E2E	ebp del llamador (paso 2)
esp+4	"5678"	BUF [4] -> BUF [6], \0
esp	"1234"	BUF [8] -> BUF [3]

(El `char` tiene código ASCII 42 = 0x2A y `.` tiene código ASCII 56 = 0x2E)

Resultado: Con el `ret` la CPU salta a la dirección de memoria 0x2A2A2A2A y ejecuta cualquier código que se encuentre ahí!

2.4.5. Generación de ejecutable



2.4.6. Llamadas a sistema en Linux

#	Nombre	eax	ebx	ecx	edx
0	<code>sys_restart_syscall</code>	0x00	-	-	-
1	<code>sys_exit</code>	0x01	int error_code	-	-
2	<code>sys_fork</code>	0x02	struct_pt_regs*	-	-
3	<code>sys_read</code>	0x03	unsigned int fd	char __user *buf	size_t count
4	<code>sys_write</code>	0x04	unsigned int fd	const char __user *buf	size_t count
5	<code>sys_open</code>	0x05	const char __user *filename	int flags	int mode
6	<code>sys_close</code>	0x06	unsigned int fd	-	-
7	<code>sys_waitpid</code>	0x07	pid_t pid	int __user *stat_addr	int options
8	<code>sys_creat</code>	0x08	const char __user *pathname	int mode	-
9	<code>sys_link</code>	0x09	const char __user *oldname	const char __user *newname	-

2.4.7. Comparación de sintaxis

NASM	AT&T (GAS)
<pre>nasm -f elf -o program.o program.asm</pre>	<pre>as -o program.o program.s</pre>
<pre>; Empieza el segmento de texto section .text global_start ; Punto de entrada del programa _start: ; Poner el número para ; llamada a sistema mov eax, 1 ; Retornar valor mov ebx, 2 ; Llamar al SO int 80h</pre>	<pre># Empieza el segmento de texto .section .text .globl_start # Punto de entrada del programa _start: # Poner el número para # llamada a sistema movl \$1, %eax /* Retornar valor */ movl \$2, %ebx # Llamar al SO int 80h</pre>

2.4.8. Hello world

Hello world(GAS) v1:

```
.text                                # section declaration
.global_start                         # we must export the entry point to
                                      # the ELF linker or loader. They
                                      # conventionally recognize _start
                                      # as their entry point. Use ld -e foo
                                      # to override the default

_start:                               # write our string to stdout
    movl $len, %edx                  # third argument: message length
    movl $msg, %ecx                 # second argument: pointer to message
                                      # to write
    movl $1, %ebx                   # first argument: file handle (stdout)
    movl $4, %eax                   # system call number (sys_write)
    int $0x80                        # call kernel

                                      # and exit

    movl $0, %ebx                   # first argument: exit code
    movl $1, %eax                   # system call number (sys_exit)
    int $0x80                        # call kernel

.data                                # section declaration

msg:                                 # our dear string
    .ascii "Hello world\n"          # length of our dear string
    len = .-msg
```

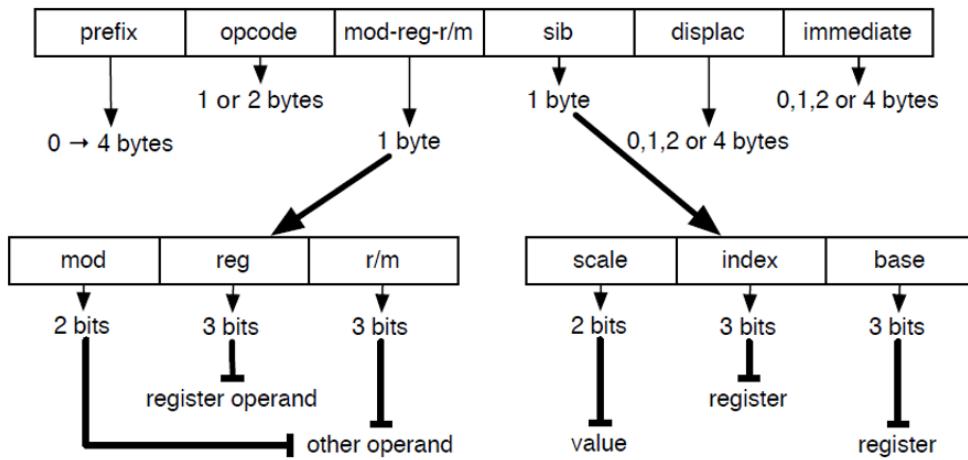
Hello world(GAS) v2 (compilada):

```
#include <stdio.h>

int main(){
    printf("Hello world!");
}
```

```
.file "helloWorld.cpp"
      .section      .rodata
.LC0:
      .string "Hello world!"
      .text
      .globl  main
      .type   main, @function
main:
.LFB0:
      .cfi_startproc
      pushq   %rbp
      .cfi_def_cfa_offset 16
      .cfi_offset 6,-16
      movq    %rsp, %rbp
      .cfi_def_cfa_register 6
      movl    $.LC0, %edi
      movl    $0, %eax
      call    printf
      movl    $0, %eax
      popq    %rbp
      .cfi_def_cfa 7, 8
      ret
      .cfi_endproc
.LFE0:
      .size   main,.-main
      .ident "GCC:(Gentoo 4.6.3 p1.11, pie-0.5.2) 4.6.3"
      .section      .note.GNU-stack,"",@progbits
```

2.4.9. Formato de instrucción



2.4.10. Ejemplos

```

addl %ebx, %esp = | 0000 0011 1110 0011 |
                    = | opcode | mode-reg-r/m |
                    = | 0000 0011 | 11 | 100 | 011 |
                    = | addl | reg | esp | ebx |
addl -8(%ebp), %esi = | 0000 0011 0111 0101 1111 1000 |
                      = | opcode | mod-reg-r/m | immediate |
                      = | 0000 0011 | 01 | 110 | 101 | 1111 1000 |
                      = | addl | esi p1 | ebp | esi p2 | -8 |

```

2.4.11. Variaciones opcode

El opcode puede variar para la misma instrucción

Instrucción	opcode	motivo
addl -8(%ebp), %esi	0x03	”Normal”
addl %esi, -8(%ebp)	0x01	Invertido
addb %dh, -8(%ebp)	0x00	Bytes + invertido
addb \$1, %esi	0x83	Inmediato

2.4.12. Prefijos

El prefijo puede indicar el tamaño de los operadores.

Por ejemplo:

```
addw %si, -8(%ebp) = | 0x66 | 0x01 | ...
```

El prefijo 0x66 indica que los operandos son de 16 bits.

2.5. ARM Assembler

2.5.1. Modos del Procesador

Modo	Ab.	Descripción
User	usr	Ejecución normal de los programas
FIQ	fiq	Interrupción rápida para transferencia de datos de alta velocidad
IRQ	irq	Manejo de interrupciones de propósito general
Supervisor	svc	Modo protegido para el sistema operativo
Abort	abt	Implementa memoria virtual y/o memoria protegida
Undefined	und	Soporta emulación por software de coprocesadores de HW
System	sys	Corre tareas privilegiadas del sistema operativo

Los modos del procesador pueden cambiar por:

- Software
- Interrupciones
- Procesamiento de excepciones

2.5.2. Modo User

- La gran mayoría de las aplicaciones corren en este modo.
- Mientras se ejecuta el programa no se puede acceder a recursos protegidos y cambiar de modo (a menos que se lance una excepción).
- De esta manera el Sistema Operativo puede controlar el uso de recursos.

2.5.3. Modos Privilegiados

- Todos los otros modos (no User) entran en esta clasificación, por ende pueden: acceder a recursos del sistema y cambiar de modo.
- Estos también se pueden subclasificar en: Modos de Excepción y Modo System.

2.5.4. Modos de Excepción

- Se entra a ellos cuando ocurre una excepción.
- Estos son: FIQ, IRQ, Supervisor, Abort y Undefined.
- Tienen registros adicionales para evitar corromper el estado del modo User.

2.5.5. Modo System

- Es el único modo privilegiado que no es de excepción.
- No tiene registros adicionales, es decir ocupa los mismos que el modo User.
- Puede corromper el estado del modo User.

2.5.6. Registros

Los procesadores ARM tienen 37 registros. Los registros están dispuestos en banks que se superponen parcialmente. Hay diferentes banks de registros para cada modo del procesador. Los banks de registros dan un rápido cambio de contexto para tratar las excepciones de procesador y operaciones privilegiadas.

usr	sys	svc	abt	und	irq	fiq
			R0			
			R1			
			R2			
			R3			
			R4			
			R5			
			R6			
			R7			
			R8		R8_fiq	
			R9		R9_fiq	
			R10		R10_fiq	
			R11		R11_fiq	
			R12		R12_fiq	
R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	
R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	
		R15				
		CPSR				
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	

Estos son los registros disponibles en la arquitectura ARM:

- 30 para propósitos generales, registros de 32 bits.
- 1 para Program Counter (PC).
- 1 para Current Program Status Register (CPSR)
- 5 para Saved Program Status Register (SPSR)

Registros de propósito general

- 15 registros de propósito genral son visibles en cualquier momento, en función del modo actual en que se encuentra el procesador.
- En el lenguaje ARM assembly por convención el registro R13 es usado como Stack Pointer (SP).
- En el modo User el registro R14 es usado como Link Register (LR) para guardar la dirección de retorno cuando se realiza una llamada a una subrutina. También

se puede utilizar como un registro de propósito general, si la dirección de retorno se almacena en el Stack.

- En los Modos de Excepción el registro R14 contiene la dirección de retorno de la excepción, o una dirección de retorno de una subrutina si las llamadas a subrutinas se ejecutan dentro de una excepción. R14 se puede utilizar como un registro de propósito general, si la dirección de retorno se almacena en el Stack.

Program Counter (PC)

- Program Counter se accede como R15 (o PC).
- Se incrementa en 4 bytes por cada instrucción en el estado.
- Las instrucciones de salto cargan la dirección de destino en el contador de programa.

También se puede cargar el Program Counter directamente mediante instrucciones de uso de datos. Por ejemplo, para volver de una subrutina, se puede copiar el registro de enlace en el Program Counter usando:

```
|| MOV PC, LR
```

Durante la ejecución, R15 no contiene la dirección de la instrucción que se está ejecutando, esta dirección es típicamente PC-8.

Current Processor Status Register (CPSR)

El CPSR contiene:

- El modo del procesador actual, el cual está representado por 4 bits.
- Flags que habilitan interrupciones, I y F.
- Copias de los flags de condición de la Unidad lógica de aritmética (ALU):

Flag	Nombre	Significado
C	Carry	Operation caused a carry
V	Overflow	Operation caused an overflow
Z	Zero	Operation resulted in 0
N	Negative	Operation resulted in a negative value

Saved Program Status Register (SPSR)

- Los SPSRs se utilizan para almacenar la CPSR cuando se toma una excepción.
- Una SPSR es accesible en cada uno de los modos de Excepción.
- Modos User y System no tienen un SPSR porque estos no son los modos que manejan las excepciones.

2.5.7. Asignación

Move (MOV)

La instrucción MOV copia el valor de Operand2 en Rd.

```
||MOV{S} Rd, Operand2
```

Move Not (MVN)

La instrucción MVN toma el valor de Operando2, realiza una operación lógica NOT bit a bit en el valor, y coloca el resultado en Rd.

```
||MVN{S} Rd, Operand2
```

Ejemplos

```
||MOV R3, #5 \tab @ R3 = 5  
MOV R2, R3 \tab @ R5 = R3 = 5  
MOV PC, LR  
  
MVN R2, #3 \tab @ R2 = -4
```

2.5.8. Operaciones aritméticas

Adición (ADD)

La instrucción ADD guarda en Rd la suma de los valores Rn y Operand2.

```
|| ADD{S} {Rd}, Rn, Operand2
|| Rd := Rn + Operand2
```

Adición con carry (ADC)

La instrucción ADC guarda en Rd la suma de los valores Rn, Operand2 y el Carry.

```
|| ADC{S} {Rd}, Rn, Operand2
|| Rd := Rn + Operand2 + Carry
```

Sustracción (SUB)

La instrucción SUB guarda en Rd la resta de los valores Rn y Operand2.

```
|| SUB{S} {Rd}, Rn, Operand2
|| Rd := Rn - Operand2
```

Sustracción con carry (SBC)

La instrucción SBC guarda en Rd la resta de los valores Rn, Operand2 y la negación del Carry.

```
|| SBC{S} {Rd}, Rn, Operand2
|| Rd := Rn - Operand2 - NOT(Carry)
```

Ejemplos

```
|| MOV R2, #1
|| ADD R2, R2, #2  @R2 = 1 + 2 = 3
|| SUB R2, R2, #1  @R2 = 3 - 1 = 2
|| ADC R2, R2, #3  @R2 = 2 + 3 + 0 (carry) = 5
|| SUB R2, R2, #4  @R2 = 5 - 4 - NOT(0) = 0
|| MOV R3, #3
|| ADD R1, R3, R2  @R1 = 0 + 3 = 3
```

2.5.9. Comparaciones

Estas instrucciones comparan el valor que hay en un registro con Operand2. Se actualizan los Flags de condición con el resultado de la comparación, pero no guarda el resultado en ningún registro.

Compare (CMP)

La instrucción CMP resta el valor de Operand2 del valor en el registro Rn.

```
|| CMP Rd, Operand2
```

Compare Negative (CMN)

La instrucción CMN suma el valor de Operand2 del valor en el registro Rn.

```
|| CMN Rd, Operand2
```

2.5.10. Operaciones lógicas

AND, OR, Exclusive OR, AND NOT y OR NOT

- Las instrucciones AND, ORR y EOR realizan operaciones bit a bit de AND, OR ,y OR exclusivo sobre los valores Rn y Operand2.
- La instrucción BIC (Bit Clear) realiza una operación AND de los bits de Rn con los complementos de los correspondientes bits de Operand2.
- La instrucción ORN realiza una operación OR de los bits de Rn con los complementos de los correspondientes bits de Operand2.

```
|| OP = AND, ORR, EOR, BIC y ORN
|| OP{S} Rd, Rn, <Operand2>
|| Rd := Rn OP Operand2
```

Ejemplos:

```

|| MOV R5, #6
|| AND R5, #1
|| ORR R5, #0
|| EOR R5, #0
|| BIC R5, #15
|| ORN R5, #7

```

Test bits (TST)

La instrucción TST realiza una operación AND bit a bit en el valor Rn y el valor de Operand2, guardando el resultado en Rn y no en los Flags de CPSR.

```

|| TST Rn, Operand2

```

Test de equivalencia (TEQ)

La instrucción TST realiza una operación OR bit a bit en el valor Rn y el valor de Operand2, guardando el resultado en Rn y no en los Flags de CPSR.

```

|| TEQ Rn, Operand2

```

2.5.11. Saltos

Las instrucciones de salto son usadas para:

- Saltos hacia atrás para formar loops.
- Saltos hacia adelante para crear estructuras condicionales.
- Saltos a subrutinas.

```

|| OP = B, BL,BX
|| OP <label>

```

La instrucción B realiza un salto hacia *label*.

La instrucción BL realiza un salto hacia *label* (cambia PC) y además copia la dirección de la próxima instrucción en R14 (LR). Se usan los registros R0-R3 como argumentos (pueden ser corruptos) y R4-R11 como variables locales (deben ser preservados).

La instrucción BX realiza un salto a una función definida en el programa.

2.5.12. Ejecución condicional

Las instrucciones pueden ser condicionadas con sufijo, como los que se muestran a continuación:

Mn	Significado	Mn	Significado
CS	Carry Set	CC	Carry Clear
EQ	Equal (Zero Set)	NE	Not Equal (Zero Clear)
VS	Overflow Set	VC	Overflow Clear
GT	Greater Than	LT	Less Than
GE	Greater Than or Equal	LE	Less Than or Equal
PL	Plus (Positive)	MI	Minus (Negative)
HI	Higher Than	LO	Lower Than (aka CC)
HS	Higher or Same (aka CS)	LS	Lower or Same

En general la gran mayoría de las instrucciones mostradas pueden ser condicionadas. Ejemplos:

```
CMP R0,R1
BEQ ends @Salta al label ends si R0 es igual a R1

CMP R0, R1
SUBGT R0, R0, R1 @resta R1 a R0 si R0 es mayor a R1
```

2.5.13. Load - Store

#of es el desplazamiento en bytes y debe ser en múltiplos de 4.

Load Register (LDR)

La instrucción LDR carga una palabra desde la memoria.

```

|| LDR Rd, [Rn {, #of}]    @Rd := [Rn + #of]
|| LDR Rd, [Rn {, #of}]!   @Rd := [Rn + #of] \tab Rn := Rn + #of
|| LDR Rd, [Rn], #of      @Rd := [Rn] \tab Rn := Rn + #of

```

Store Register (STR)

La instrucción STR guarda una palabra en la memoria.

```

|| STR Rd, [Rn {, #of}]    @[Rn + #of] := Rd
|| STR Rd, [Rn {, #of}]!   @[Rn + #of] := Rd \tab Rn := Rn + #of
|| STR Rd, [Rn], #of      @[Rn] := Rd \tab Rn := Rn + #of

```

Ejemplos

```

|| LDR R3, [R2,#15]
|| LDR R0, [R1,#7]!
|| STR r1,[sp,#20]

```

2.5.14. Stack

Los Stacks son muy flexibles en la arquitectura ARM, ya que la aplicación se dejó completamente al software.

El conjunto de instrucciones ARM no contiene instrucciones como Push y Pop. Todas las operaciones son realizadas por instrucciones de acceso a memoria, con modos de direccionamiento automático.

El Stack Pointer (SP) es un registro que apunta a la parte superior de la pila. Por convención el registro R13 es usado como Stack Pointer.

Existen diferentes opciones de implementación produciendo distintos tipos de pila:

- Según como crece el Stack: Ascendente y descendente. En el primero al realizar un Push el SP se incrementa, mientras que en el segundo se decrementa.
- Según lo que apunta el puntero: Empty Stack y Full Stack. En el primero se apunta a la ubicación en donde se almacenará, un push guardará el valor e incrementará el valor del SP; mientras que en el segundo se apunta a la ubicación del último elemento que se almacenó, un Push incrementará el valor del SP y guardará el valor.

Con lo anterior es posible usar 4 tipos de Stacks diferentes: Full-Ascending, Full-Descending, Empty-Ascending y Empty-Descending. Todos pueden ser usados para cargar y guardar instrucciones. Se puede realizar Push y Pop múltiples.

Load-Multiple (LDM)

La instrucción LDM carga múltiples valores desde un registro.

```
|| LDM{type-stack} Rn!, RegList
```

Store-Multiple (STM)

La instrucción STM guarda múltiples valores en un registro.

```
|| STM{type-stack} Rn!, RegList
```

Ejemplos

```
LDMFD R13!,{R2,R3} ; cargamos 2 valores desde el Stack en los
registros R2 y R3, FD = Full Descending
ADD R2,R3,R2 ; guardamos en R2 la suma de R3 y R2
STMFD R13!,{R2} ; guardamos en el Stack el resultado obtenido
```

2.5.15. Ejemplo

```
J0:  
    STMFD R13!, {R4} ;guarda R4 en el Stack  
    LDMFD R13!, {R2} ;carga en R2 el primer elemento del Stack  
    CMP R2, #0 ;compara R2 con #0  
    BNE J2 ;Salta a J2 si R2 es distinto de #0  
    MOV R2, #5 ;guarda #5 en R2  
    STMFD R13!, {R2} ;push de R2 en el Stack  
    B J3 ;salto a J3 (if-else)  
  
J2:  
    MOV R2, #7 ;guarda #7 en R2  
    STMFD R13!, {R2} ;push de R2 en el Stack  
  
J3:  
    B J1 ;salto a J1, es decir seguimos con el main  
  
main:  
    MOV R2, #2 ; se guarda #2 en R2  
    STMFD R13!, {R2} ;push de R2 en el Stack  
    LDR R5, =J0 ;guardamos en R5 la direccion de la funcion J0  
    STMFD R13!, {R5} ;push de R5 en el Stack  
    MOV R2, #0 ;se guarda #0 en R2  
    STMFD R13!, {R2} ;push de R2 en el Stack  
    LDMFD R13!, {R4,R5} ;guardamos los 2 prim elem del Stack en R4  
    y R5  
    BX R5 ;R5 es una direccion de funcion, ejecuta la funcion  
  
J1:  
    LDMFD R13!, {R2,R3} ;guardamos los 2 prim elem del Stack en R2  
    y R3  
    ADD R2, R3, R2 ;se guarda la suma de R2 y R3 en R2  
    STMFD R13!, {R2} ;push de R2 en el Stack 0  
    LDMFD R13!, {R1} ;guardamos en R1 el primer elemento del Stack  
    MOV R0, #1 ;se guarda #0 en R1  
    swi 0x6b ;se imprime en pantalla el valor de R1  
    swi 0x11 ;salida del programa
```


Capítulo 3

Arquitectura física de un computador

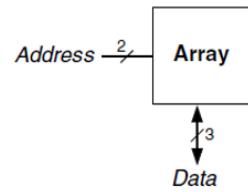
3.1. Memoria

Varios tipos:

- RAM:
 - SRAM
 - DRAM
 - SDRAM
- ROM
 - EPROM
 - Flash
 - etc

Arreglos

Normalmente las memorias están organizadas en arreglos bidimensionales de celdas.



En cada celda se almacena 1 bit.

Una dirección de memoria hace referencia a una fila completa de la matriz (llamada palabra)

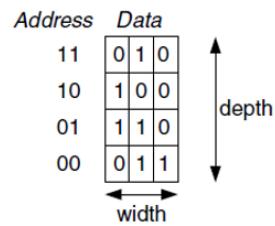
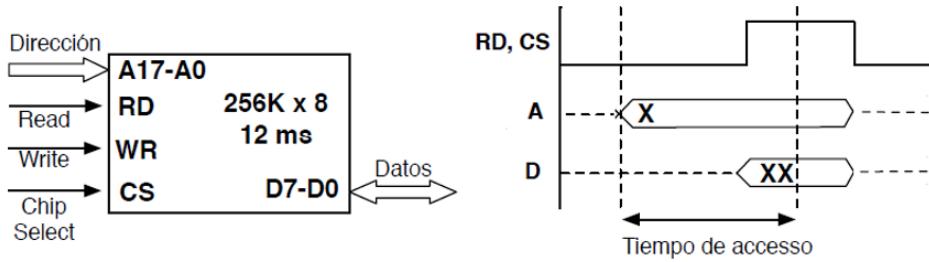


Diagrama de tiempo



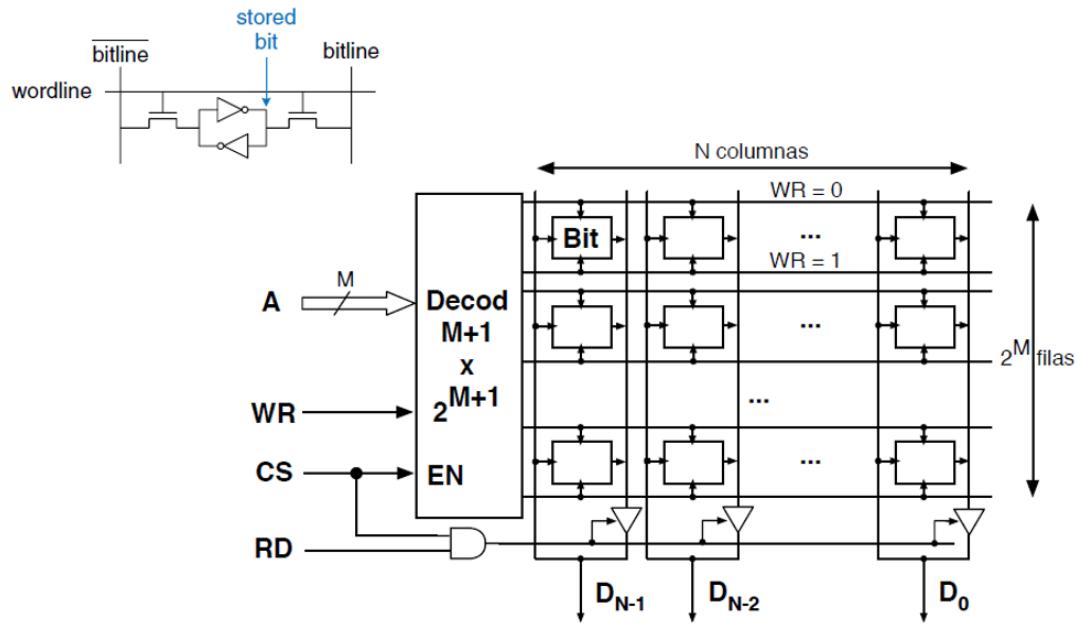
Espera

Si el tiempo de acceso es muy extenso, la memoria setea WAIT. La CPU repite el segundo ciclo hasta que WAIT vaya a cero.

SRAM

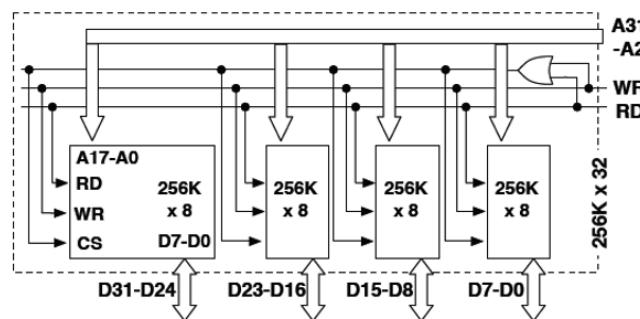
- Static Random Memory
- Rápida
- Al apagarse se borran
- 4-6 transistores por bit

Implementación:



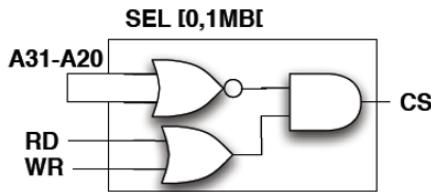
Memorias en paralelo

El ancho de la palabra de las memorias es menor al del bus de datos \Rightarrow Ponemos chips en paralelo:



Selection chip

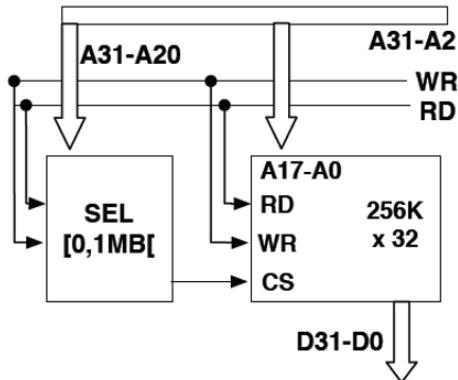
$$a \in [0, 1MB] \Leftrightarrow a = [a_{31} a_{30} \dots a_{20} a_{21} \dots a_2 a_1 a_0] = [0 0 \dots 0 ? ? \dots ? 0 0] \quad (1MB = 2^{20})$$



Escalamiento memoria

SEL [0,1MB] = 1
 Si $(RD+WR)$ y $A \in [0, 1MB]$
 i.e. $a = [a_{31} a_{30} \dots a_{20} a_{21} \dots a_2 a_1 a_0] = [0 0 \dots 0 ? ? \dots ? 0 0]$

SEL [1MB, 2MB] = 1
 Si $(RD+WR)$ y $A \in [1 MB, 2MB]$
 i.e. $a = [a_{31} a_{30} \dots a_{20} a_{21} \dots a_2 a_1 a_0] = [0 0 \dots 1 ? ? \dots ? 0 0]$

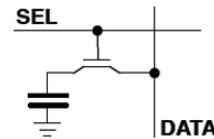


Se puede usar un decodificador para seleccionar el chip

DRAM

- Dynamic RAM
- Barata pero lenta
- 1 bit = 1 condensador y 1 transistor
- SEL = 1 \Rightarrow Escribir, cargando o descargando condensador
- SEL = 0 \Rightarrow Se mantiene por unos milisegundos
- Para leer, SEL = 1

- Lectura destructiva



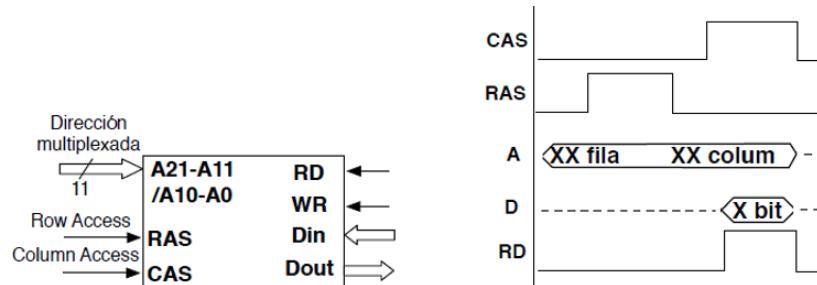
- Lento: tiempo de acceso: 60-70 ms (vs 10-20 ms SRAM)
- Cada 4 ms reescribir (descarga)
- Formatos:
 - SIMM (Single In-line Memory Module)
 - DIMM (Dual In-line Memory Module)

Paridad:

- Bit adicional redundante
- XOR de 8 bits de datos
- Permite detectar errores de almacenamiento (ej: descarga condensador)
- Al leer, se verifica paridad
- Si no coincide \Rightarrow parity error
- Error-Correcting Memory usa Hamming Codes

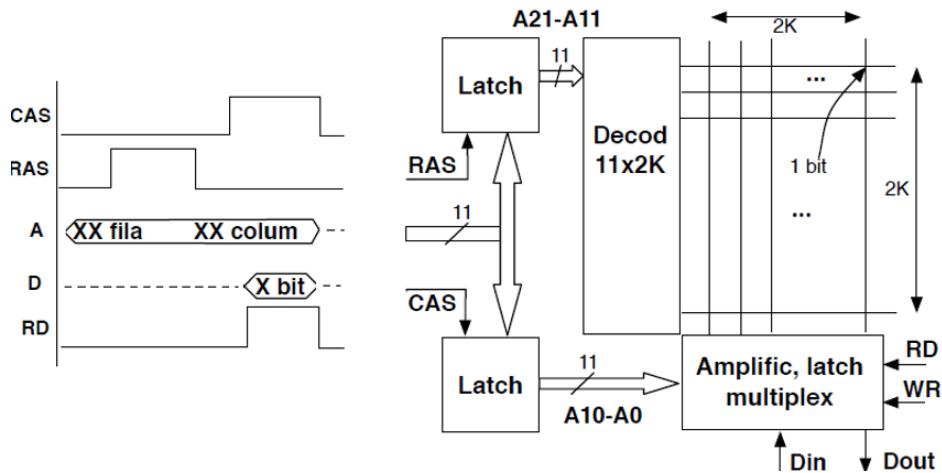
Dirección multiplexada:

Los bits se organizan como una matriz



Implementación:

Se lee toda una fila. Lectura destructiva \Rightarrow reescritura



Otros detalles:

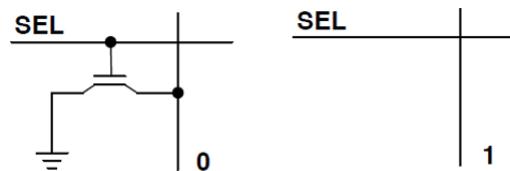
- Reescritura forzada cada 4ms
- Se usan bloques de 8 bits por fila
- Tambien existe el "Page Mode"

SDRAM

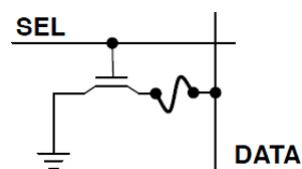
- Synchronous DRAM
- Esta controlado por un reloj
- Espera x ciclos de reloj (CAS Latency)
- Uso mas complicado de BUS de datos
- Burst mode
- DDR duplica velocidad de burst mode

ROM

- Read Only Memory
- Grabadas de fabrica
- No se pueden borrar
- No se borran al apagarlas
- Lentas
- Uso: bootstrap

**PROM**

- Programable ROM
- Se graba cada bit quemando un fusible
- No se pueden regrabar



EEPROM

- Erasable PROM
- El fusible puede ser restituido con luz ultravioleta aplicada sobre una ventana de cuarzo
- ¡Hay que tapar la ventana!



EEPROM

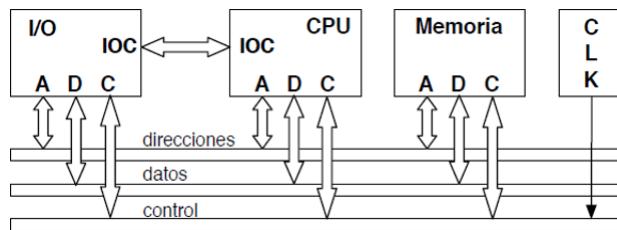
- Electrically Erasable PROM
- Tiene circuitería interna para poder borrar
- Se pueden borrar bits individualmente

Flash

- No volátil
- Puede ser reescrita por el PC
- Más lenta que SRAM o DRAM
- Se borran filas completas
- Tiene un límite de escrituras (write-erase cycles)

3.2. CPU

3.2.1. M32



3 buses:

- Bus de datos: D31 - D0
- Bus de direcciones: A31 - A2, BE3 - BE0
- Bus de control:

RD	WR	CLK	WAIT
Lectura	Escritura	Reloj	Prolongar acceso a memoria

3.2.2. CPU

- Interpreta el lenguaje de máquina
- Realiza internamente proceso de instrucción
- Contiene:
 - los registros
 - Contador programa
 - Registro de estado

3.2.3. Ciclo Instrucción

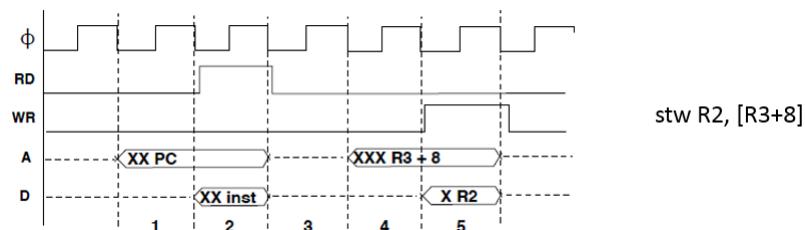
1. Fetch ($1 \text{ PC} \Rightarrow A \text{ y } 2 \text{ RD}$):

1 Se guarda el *PC* en *AR*.

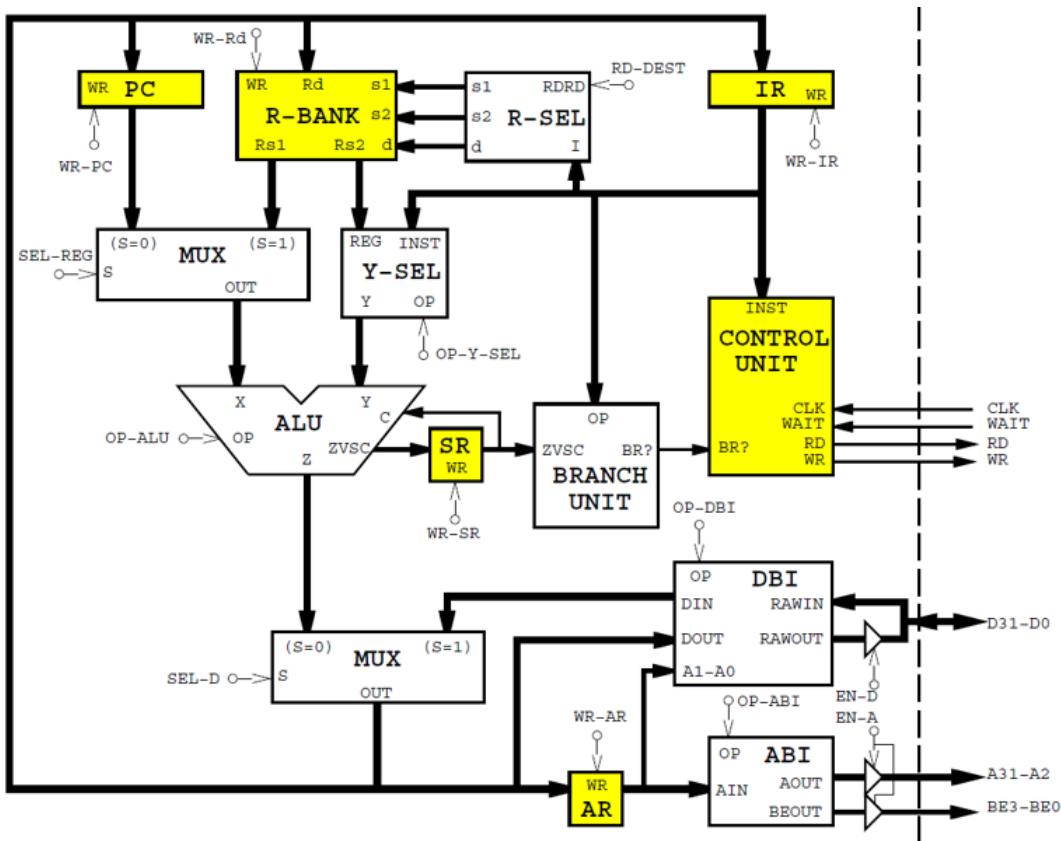
2 Espera instrucción, cuando exista la carga a *IR*.

2. Decod (3)

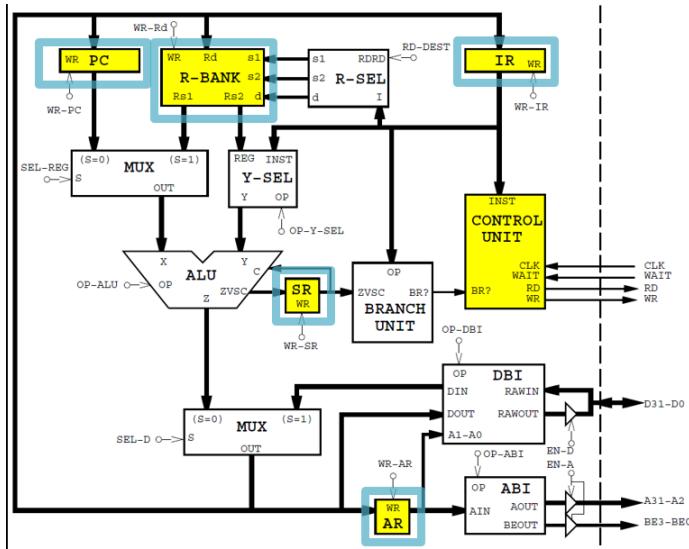
3. Exec: Depende de la instrucción (ej: 4 y 5).



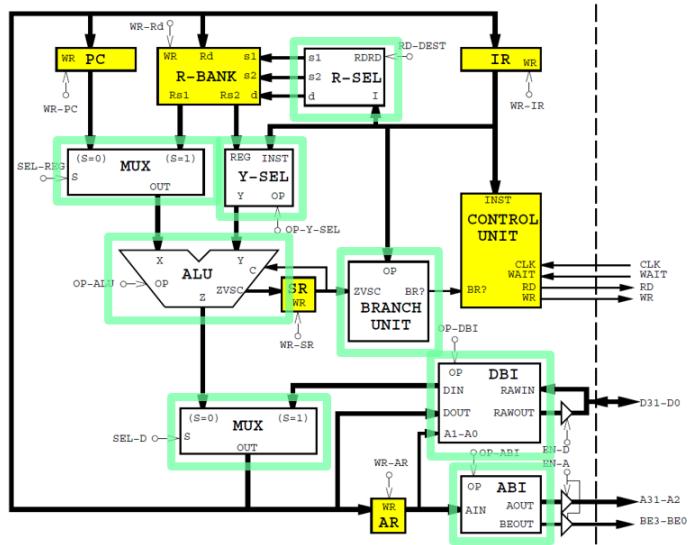
3.2.4. CPU-M32



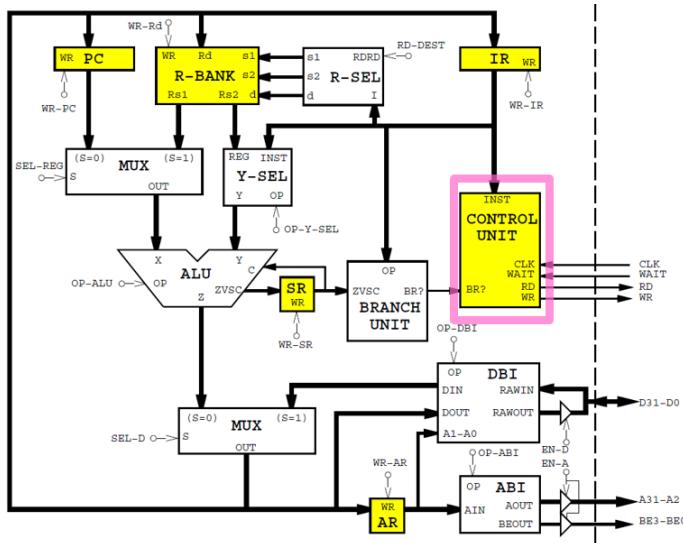
Unidades de almacenamiento:



Unidades combinacionales:



Unidad de control:



ALU (arithmetic/logic unit)

Realiza todos los cálculos del tipo $z=x \ op - alu$ y en donde $op - alu$ es la operación que indique la unidad de control.

Además indica las características del resultado en ZVSC.

OP-ALU configura la operación que realizará la *ALU* con el valor que provenga de *R-Bank o PC* (según lo que contenga *SEL-REG*) y el valor de *Y-SEL*.

Control Unit

Circuito secuencial muy complejo que genera las señales de control. Éstas hacen fluir los datos y direcciones por los buses que interconectan las unidades de almacenamiento y las de cálculo.

R-Bank (register bank)

Almacena los registros R0-R31. En **s1** y **s2** se indica el número de los registros que aparecen constantemente por **RS1** y **RS2**.

WR-Rd Permite escribir en el Bus o leer de él. Si está en 1, el Registro a escribir se selecciona de acuerdo al índice **d**.

Registros síncronos: PC y SR

Mantienen el contador de programa y el registro de estado ZVSC.

WR-PC y *WR-SR* permiten escribir en el registro correspondiente si están en 1.

Registros asíncronos: IR y AR

Mantienen la instrucción en curso y la dirección que se debe colocar en el bus de direcciones

WR-IR y *WR-AR* permiten escribir en el registro correspondiente si están en 1.

R-SEL

Extrae de la instrucción los números de los registros que intervienen en una operación. *d* corresponde al índice del registro de destino, y son los bits 23-19. *S₁* puede tener el valor de *d* o el que aparece en 18-14, esto es decidido por *RD-DEST*. *S₂* contiene los bits 19-23.

RD-DEST Si es 1, significa que el destino será utilizado como primer argumento, por ejemplo en $R1 = R1 + R2$.

Y-SEL

Elige el segundo operador de una instrucción, el que puede ser 0, 4 o lo que diga la instrucción (*RS2* o *imm*) o un desplazamiento de 24 bits en caso de un salto.

OP-Y-SEL Puede ser @0, @4, @INST, @DISP según lo antes mencionado.

Branch unit

Calcula la condición de salto durante instrucción del tipo b <cond> <label>

DBI (Data Bus Interface)

Interfaz con el bus de datos. Durante lecturas y escrituras la memoria espera bytes y halfwords en posiciones distintas de donde se encuentra la palabra original. DBI desplaza los datos y extiende el signo si es necesario.

OP-DBI puede ser @W, @H o @B, correspondiente a una palabra, media o un byte.

ABI (Address Bus Interface)

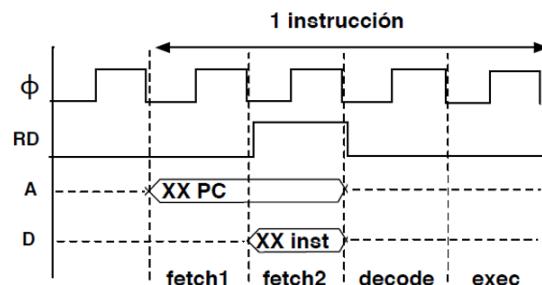
Genera los valores de A31-A2 y BE3-BE0 a partir de una dirección y el tipo de operación que le indique la unidad de control

OP-DBI Se puede escribir o leer una palabra completa, media o un Byte.

3.2.5. Sincronicidad

En cada ciclo de reloj (pulso de bajada):

- La unidad de control genera las señales de control
- Datos de los registros \Rightarrow buses externos y unidades de cálculo
- Las unidades de cálculo realizan un cálculo
- Se actualizan PC, SR y Rd



3.2.6. Operaciones

Transferencias entre registros	Señales de control
<pre> fetch1: AR := PC goto fetch2 </pre>	OP-Y-SEL := \$0\$0 OP-ALU := \$0\$OR WR-AR, EN-A OP-ABI := \$0\$W
<pre> fetch2: IR := Memw[AR] if WAIT goto fetch2 else goto decode </pre>	OP-DBI := \$0\$LDW SEL-D, WI-IR, EN-A, RD OP-ABI := \$0\$W
<pre> decode: PC := PC ⊕ 4 goto execute1 </pre>	OP-Y-SEL := \$0\$4 OP-ALU := \$0\$ADD WR-PC

Instrucción: add R4, -103, R11

Transferencia entre registros	Señales de control
<pre> exec1: R11 := R4 ⊕ -103 goto fetch1 </pre>	SEL-REG, WR-RD, WR-SR, RD-DEST OP-Y-SEL := \$0\$INST OP-ALU := \$0\$ADD

Instrucción: `stb R3, [R5 + R0]`

Transferencias entre registros	Señales de control
<pre> exec1: AR := R5 ⊕ R0 goto exec2 </pre>	SEL-REG, WR-AR, EN-A OP-Y-SEL := @INST OP-ALU := @ADD OP-ABI := @W
<pre> exec2: Memb[AR] := Truncb[R3] if WAIT goto exec2 else goto fetch1 </pre>	SEL-REG, RD-DEST, EN-D, EN-A, WR OP-Y-SEL := @0 OP-ALU := @OR OP-DBI := @STB OP-ABI := @B

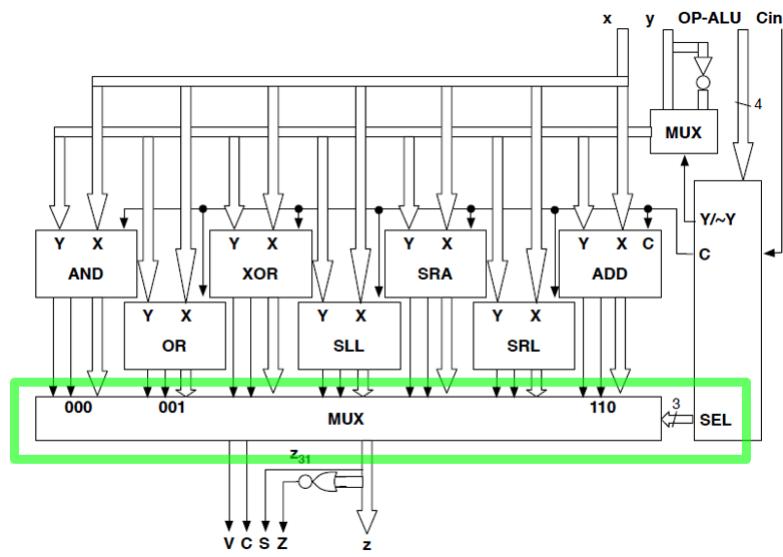
Instrucción: `bg <label>`

Transferencias entre registros	Señales de control
<pre> exec1: if br? goto exec2 else goto fetch1 </pre>	
<pre> exec2: PC := PC ⊕ ExtS(IR[23-0]) goto fetch1 </pre>	WR-PC OP-Y-SEL := @DISP OP-ALU := @ADD

3.2.7. Implementación ALU

Realiza en paralelo las 7 operaciones.

Selecciona la salida (V, C y Z).

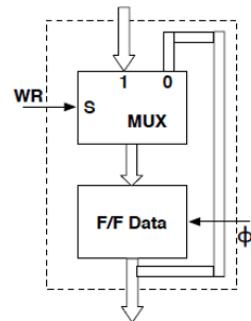


3.2.8. Circuito combinacional

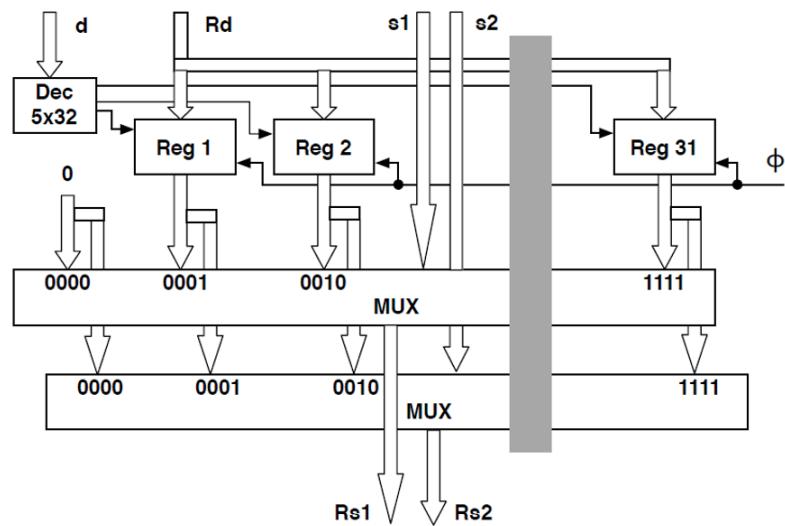
Operaciones realizadas por la ALU		
OP-ALU	Código binario	Valor de z
@ADD	000	$x \oplus y$
@ADDX	0001	$x \oplus y \oplus c$
@SUB	0010	$x \oplus \neg y \oplus 1$
@SUBX	0011	$x \oplus \neg y \oplus c$
@AND	0100	$x \& y$
@OR	0101	$x y$
@XOR	0110	$x \text{ xor } y$
@SLL	0111	$x << y$
@SRL	1000	$x >> y$
@SRA	1001	$x >>_s y$

OP-ALU	C _{in}	C	SEL	Y/~Y	Calcula
ADD	0000	X	0	110	$x \oplus y$
ADDX	0001	0	0	110	$x \oplus y \oplus c$
		1	1	110	$x \oplus y \oplus c$
SUB	0010	X	1	110	$x \oplus \neg y \oplus 1$

3.2.9. Registro síncrono

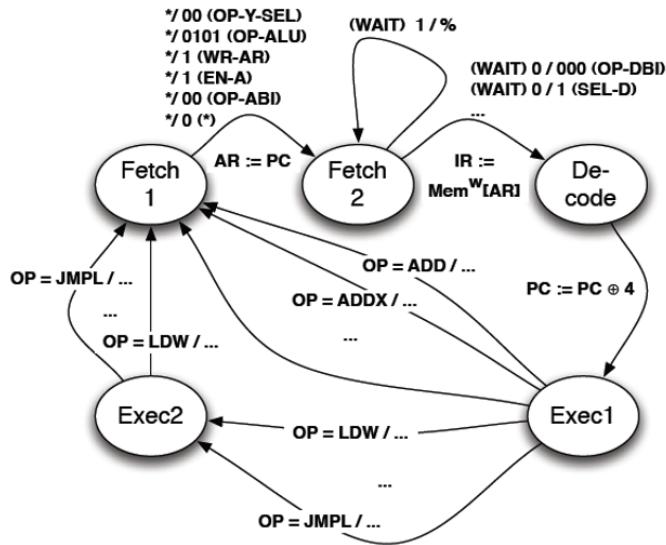


3.2.10. Banco de registros



3.2.11. Unidad de control

Circuito secuencial complejo.



3.2.12. Optimización

- Dentro del Ciclo
 - Unidad de Control calcula Señales de Control
 - Unidades de Cálculo efectúan sus cálculos
 - Unidades de Almacenamiento guardan resultados
- El ciclo de reloj queda acotado
- Para acelerar la CPU se puede:
 - Hacer transistores más rápidos
 - Mejorar tasa instrucciones/ciclo
 - Disminuir tiempos de cómputo (señales de control y cálculos)
- La existencia de decode permite precalcular señales de control

3.3. Apéndice

3.3.1. Preguntas

Resueltos

P3 C2 2008-1 A continuación se muestran 8 conjuntos de transferencias entre registros para el diseño actual de M32. Considerando cada uno de estos conjuntos de forma independiente señale cuáles de ellos se pueden llevar a cabo en 1 solo ciclo de reloj y cuáles no. Si un conjunto de transferencias es válido, indique las señales de control necesarias para realizarlas. Si un conjunto no lo es, explique por qué.

- 1. $AR = PC; PC = PC+4$
- 2. $PC = 4; AR = 4$
- 3. $R1 = PC$ cuando IR contiene la instrucción ADD R2, R3, R1 ($R1 = R2+R3$)
- 4. $PC = R1+R3$ cuando IR contiene la instrucción ADD R2, R3, R1
- 5. $R1 = PC+R2$ cuando IR contiene la instrucción ADD R2, R3, R1
- 6. $IR = R2+R3$ cuando IR contiene la instrucción ADD R2, R3, R1
- 7. $PC = \text{Mem}[AR]$
- 8. $R1 = R2+R3; IR = \text{Mem}[AR]$ cuando IR contiene la instrucción ADD R2, R3, R1

Observaciones: No intente encontrar un sentido a estas transferencias. Recuerde que AR e IR son latches.

Solución

- 1. La única forma de cargar el latch AR es haciendo una operación en la ALU y poniendo su resultado en el bus (El cable que “rodea” la CPU). Entonces no se pueden hacer las operaciones referidas en el mismo ciclo de reloj, porque en ambos casos hay que pasar por la ALU.
- 2. Sí, con un truco sucio: La arquitectura de M32 dispone de 32 registros, y el registro R0 siempre tiene valor cero. Entonces, si en IR hay una instrucción XXX R0, R1, R2, por Rs1 sale un cero, y se ponen las siguientes señales de control: $OP-Y-SEL \leftarrow @4$

SEL-REG \leftarrow 1

WR-AR \leftarrow 1

OP- ALU \leftarrow @ADD WR-PC \leftarrow 1.

Si no se dispusiera de R0 no se podría hacer lo pedido.

- 3. Sí. OP-Y-SEL \leftarrow @0

- 4. RD-DEST \leftarrow 1

WR-Rd \leftarrow 1

WR-PC1

OP-ALU \leftarrow @OR

OP-ALU \leftarrow @ADD

OP-Y-SEL \leftarrow 0

SEL-REG \leftarrow 1

- 5. No, porque el MUX superior no puede dejar pasar a PC y Rs2 al mismo tiempo.

- 6. OP-Y-SEL \leftarrow @INST, SEL-REG \leftarrow 1

OP-ALU \leftarrow @ADD

WR-IR \leftarrow 1

- 7. No, porque hay un acceso a memoria, lo que implica que en el mejor de los casos la memoria responde en el ciclo de reloj siguiente al de la petición. Sin embargo, si la memoria fuera tan rápida que pudiera responder en el mismo ciclo de reloj con un valor, sí se podría, hacer, encendiendo las señales SEL-D y WR-PC.

- 8. Un argumento fácil es que como hay acceso a memoria, entonces ya no se puede hacer la operación en un solo ciclo de reloj. Sin embargo, incluso si la memoria fuera rápida (como en el punto anterior) no se podría hacer lo pedido, porque habría conflicto en el MUX inferior (pasa R1+R3 para entrar en Rd, o pasa Mem[AR] para entrar en IR).

Propuestos

3.3.2. Referencias

- Clase 16: Memorias <https://www.youtube.com/watch?v=cKEowYtr0HM>
- Clase 17: CPU <https://www.youtube.com/watch?v=RHz57ufIyvg>
- Clase 18: M32 <https://www.youtube.com/watch?v=BaSt4Hz5qTM>

Capítulo 4

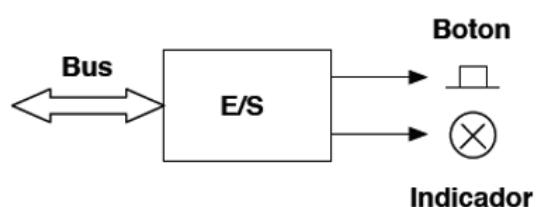
Entrada/Salida

Dispositivos de entrada/salida:

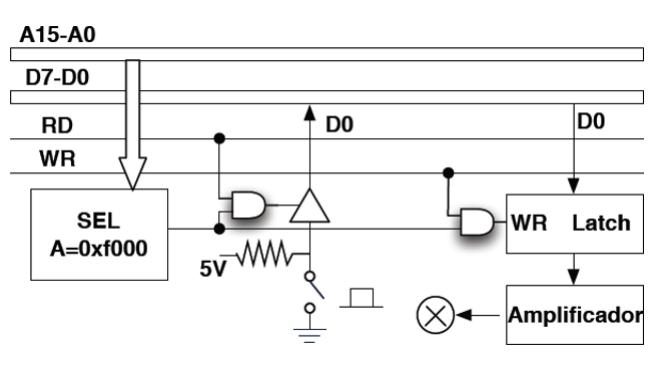
- Mouse, teclado
- Monitor, impresora
- Disco duro, Lector de DVD
- Módem, tarjetas de red

4.1. Lectura / Escritura en Dispositivo E/S

¿Cómo examinar y comandar a los dispositivos de entrada/salida?



Ejemplo:



- Para saber si el botón fue presionado se lee en la dirección 0xF000
- Para encender o apagar el LED se escribe en la dirección 0xF000

Supongamos que R1=0xF000

```

ldw [R1 + 0], R2
⇒ R2 = |?...?b|
      b = 0 boton presionado
      b = 1 boton arriba
      (en C: R = *((char*) 0xf000))

```

```

stw R3, [R1 + 0]
⇒ R3 = |?...?1| apaga la luz
⇒ R3 = |?...?0| enciende la luz
      (en C: *((char*) 0xf000) = R)

```

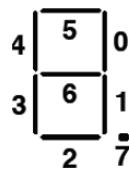
Obs: si se escribe un valor en la dirección de un dispositivo y luego se lee su contenido, en general no se obtiene lo mismo que se escribió ⇒ un dispositivo se disfraza de memoria, ¡pero no se comporta como memoria!

4.2. Mapeo en memoria

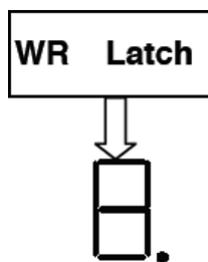
- Los dispositivos se disfrazan de memoria
- Leen-escriben en un rango de direcciones
- Un programa:
 - Examina un dispositivo leyendo en una dirección de memoria (asignada al dispositivo)
 - Comanda un dispositivo escribiendo en una dirección de memoria (asignada al dispositivo)

4.2.1. Ejemplo: visor de calculadora

- Hay 8 displays de 7 segmentos
- En cada display, se desea mostrar un dígito



Ideas solución:



- Se puede guardar en un byte la información necesaria para prender un display
- Cada segmento está asociado a un bit dentro del byte.
- No se necesita leer el valor de los latches, por lo que basta con escribirlos (aunque se podrían leer, pero es más caro y no tiene sentido)

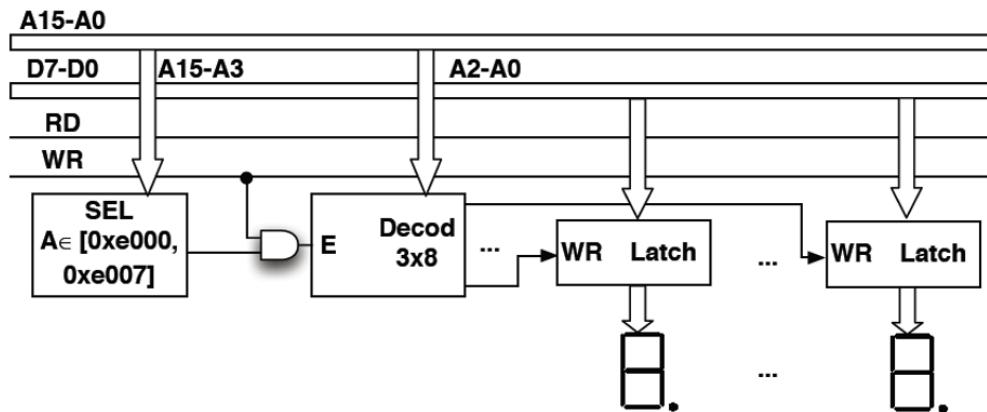
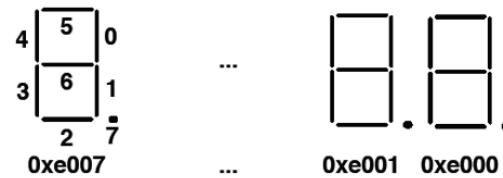


Figura 4.1: *

Solución cara: A cada dígito se le asigna 1 byte

Solución 1 (cara)

Asignar una dirección de 1 byte a cada dígito.

Si se escribe |01011110| en 0xe007 entonces aparecerá "b" en el octavo dígito

Solución 2 (económica)

- Hacer una interfaz que permita encender no más de un dígito a la vez.
- Hacer un programa que encienda secuencialmente cada dígito más de 60 veces por segundo.
- Esto crea la ilusión óptica de que el visor esta permanentemente encendido.
- Mapeo del visor en memoria:
 - 0xe000 indica que dígito esta encendido
 - 0xe001 el bitmap de ese dígito

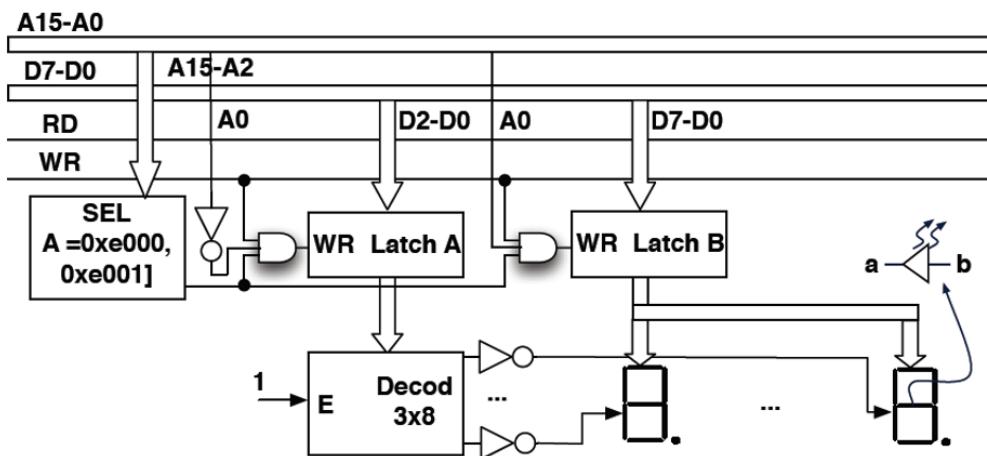
Código en C:

```
char bm_digs[8] /* bitmap de cada dígito */
display(){
    char *port_ndig = (char*) 0xe000;
    char *port_bmdig = (char*) 0xe001;

    for(int num_digit = 0 ; ; num_digit_digit = (num_digit+i)%8){
        *port_ndig = num_digit;           //cuál dígito
        *port_bmdig = bm_digs[num_digit]; //dibuja dígito
        usleep(1000/(60*8));
    }
}
```

Implementación:

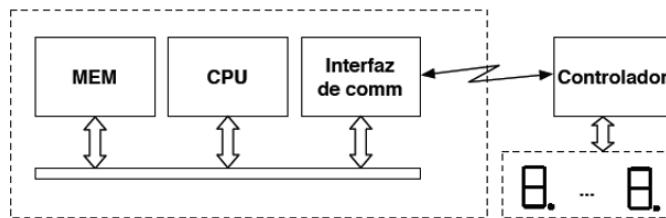
La salida del decodificador está negada por lo que entrega casi todas sus líneas en 1. Una sola línea está en 0 la que corresponde al dígito iluminado. Cada uno de sus segmentos está conectado a un bit del latch B que controla si enciende o no.



Problema: Si se quiere mostrar el visor, el programa gasta todo su tiempo en el ciclo `display()`. Si se dedica a otro cálculo, ¡el visor se apaga! Solución: Usar un controlador para desplegar el visor.

4.3. Controlador de entrada y salida

- Microprocesador dedicado a una función específica requerida para el funcionamiento de algún dispositivo.



- La CPU y el controlador intercambian información a través de una línea de comunicación.

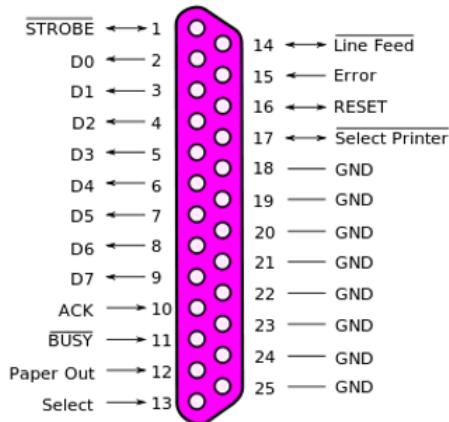
4.3.1. Típicas líneas de comunicación

	Velocidad	Distancia	Uso
Serial RS232	300 a 115 200 bit/s	hasta 10 m	módem, teclado (obsoleto), mouse (obsoleto)
SATA	1.5 Gbit/s (SATA 150) 3 Gbit/s (SATA 300) 6Gbit/s (SATA 6 Gbit/s)	hasta 1 metro	discos duros, DVD, ...
USB	1.5 o 12 Mbit/s (1.0) 480 Mbit/s (2.0) 5 Gbit/s (3.0, 3.2 Gbits/s real)	oficial hasta 5 m (1.0, 2.0), 3 m (3.0)	Universal (?)
FireWire	400 Mbit/s (FW 400) 800 Mbit/s (FW800)	Hasta 4,5 m (FW 400) o 100 m (FW 800 sobre Cat5e UTP)	A/V, ej: cámaras

4.4. Comunicación

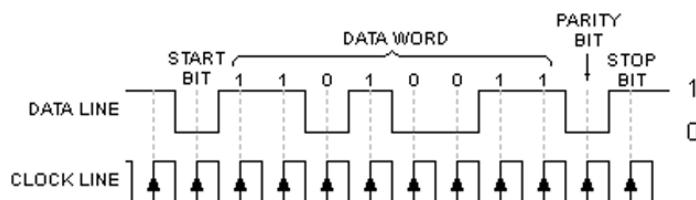
4.4.1. Comunicación paralela

- Se transmiten 8 bits simultáneamente
- Circuitería simplificada
- Crosstalk \Rightarrow Cable corto
- Skew \Rightarrow transmisión lenta
- Desplazada por comunicación serial.



4.4.2. Comunicación serial

- Cada byte es transmitido en un paquete.
- Los bits de parada/partida permiten saber cuando empieza un byte.
- El bit de paridad permite detectar errores.



Configuración

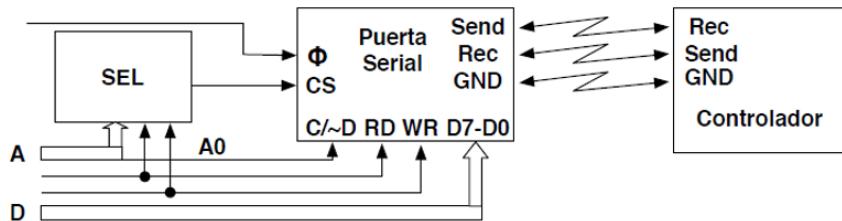
Ambos puertos en la comunicación deben tener la misma configuración:

- Velocidad (300,2400, 9600,..., 115200 bits/s)
 - Bits de datos (5, 6, 7, 8)
 - Paridad (sin, par, impar)

Interfaz SW \Leftrightarrow Serial

El puerto serial se opera a través de:

- Puerto de datos ($C/\sim D = 0$)
 - Puerto de control ($C/\sim D = 1$):
 - Se escribe el número de bits, paridad, etc.
 - Se lee el estado de envío/recepción de los datos.



Código

Configuración/estado:

Escritura:

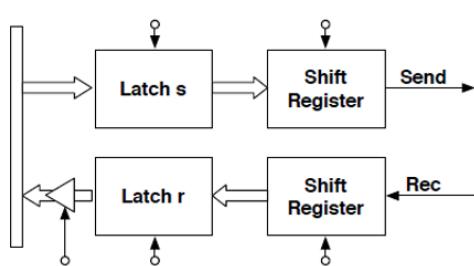
```
void enviar(char *buff, int n){
    while(n--){
        while(!*port_ctrl&1)      /* ¿se puede enviar? */
        ;                         /* busy wait */
        *port_data = *buff++;
    }
}
```

La transmisión de 1 byte toma ± 1 ms a 9600 bps, por lo que el ciclo de busy-waiting se ejecuta 1 000 a 10 000 veces antes de poder volver a transmitir.

Lectura:

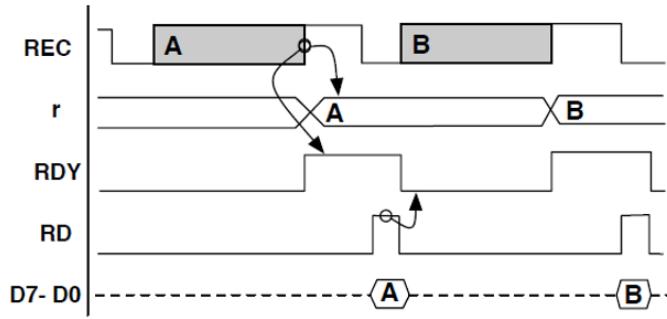
```
void recibir(char *buff, int n){
    while(n--){
        while(!*port_ctrl&2)      /* ¿se puede recibir? */
        ;                         /* busy wait */
        *buff++ = *port_data;     /* recibe 1 byte */
    }
}
```

Implementación



Cuando se completa la recepción de 1 , se almacena en el , esperando a que sea leído por la .

Si se recibe un nuevo byte antes de que la CPU lea el previo, se pierde 1 byte. Esto da un tiempo de 1 ms a 9600 bps para leer el byte. Para evitar ese problema se puede reemplazar el por un buffer de 16 bytes (16550 UART), lo que multiplica por 16 el plazo de lectura.



4.5. Ciclo busy-waiting

Ej: Escritura en puerto serial :

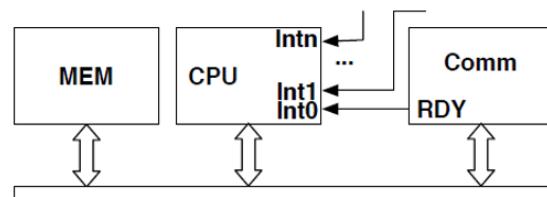
```
void enviar(char *buff, int n){
    while(n--){
        while(!*port_ctrl&1)      /* ¿se puede enviar? */
            ;                      /* busy wait */
        *port_data = *buff++;
    }
}
```

Mientras espera, ¡consume CPU!
Esta CPU no puede hacer nada más.

4.6. Interrupciones

4.6.1. Interrupciones (por HW)

La CPU NO está esperando activamente al dispositivo.



Cuando el dispositivo necesita la atención de la CPU:

- Setea RDY (la entrada INTx de la CPU) en 1.
- La CPU interrumpe el procedimiento en curso y guarda su estado.
- Invoca una rutina de atención de interrupciones, la cual interactúa con el dispositivo.
- La CPU retorna al procedimiento suspendido en el punto en que estaba en forma transparente.

4.6.2. Habilitación

Las interrupciones por hardware pueden ser habilitadas/inhibidas con instrucciones de máquina:

- enable_int
- disable_int

Un bit del registro de estado indica si las interrupciones están habilitadas.

4.6.3. Implementación

- En M32 se agrega un bit I al registro de estado ZVSC \Rightarrow IZVSC
- I = 1 \Rightarrow interrupciones por hardware habilitadas
- I = 0 \Rightarrow interrupciones por hardware inhibidas

4.6.4. Interrupciones por SW

- Son gatilladas por eventos ocurridos a partir de la ejecución de un programa.
- Ej:
 - Llamadas a instrucciones con opcode inexistente
 - División por cero
- El bit I no puede inhibir las interrupciones por software

4.6.5. Dirección en memoria

- La dirección de la rutina de atención de interrupciones (por software y hardware) se obtiene de un vector de interrupciones.

0	div por 0
+4	cód de op no existente
+8	int por hw 1
+12	int por hw 2
...	...

- El vector de interrupciones se ubica típicamente:
 - En una dirección fija dependiente de la arquitectura (ej: 80888 dir 0) o
 - En una dirección apuntada desde un registro especial en la CPU (ej: 80386).

4.6.6. Ciclo de ejecución

- La unidad de control chequea durante el ciclo fetch el estado de la línea Int
- Si Int = 1 la unidad de control invoca la interrupción.

4.6.7. Resguardo registros

- Se deben resguardar los registros usados por la rutina de atención y por el procedimiento:
 - La unidad de control se encarga de PC y SR
 - La rutina de atención se encarga de Rx
- Para retornar al procedimiento interrumpido, se debe invocar RETI, que restaura SR y PC

4.6.8. Deshabilitación interrupciones

- El dispositivo continúa seteando INT en 1 hasta que se suprima la causa de la interrupción.
- Por ello se necesita inhibir las interrupciones antes de invocar la rutina de atención.
- Si no, después de ejecutar la primera instrucción, se volvería a provocar una nueva interrupción, cayéndose en un ciclo infinito
- Al restaurar SR se reabilitan las interrupciones nuevamente

4.6.9. Código de resguardo

Realizado por la unidad de control:

```
R31 := R31 - 8
Memw[R31 + 4] := PC          !Push PC
Memw[R31 + 0] := Extw(SR)    !Push SR
SR.I = 0                      !inhibe las interrupciones
PC := Memw[IVR + 4 + x]       !dir del vec ints
                               !+despl de la int
```

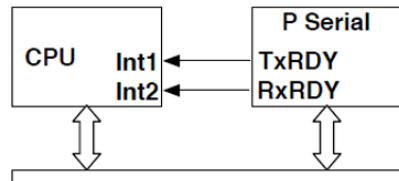
RETI (llamado al terminar la rutina de atención de interrupciones):

```
SR := Memw[R31 + 0]           !Pop SR (restaura SR.I)
PC := Memw[R31 + 4]           !Pop PC
R31 := R31 + 8
```

Observación:

El hecho que la rutina de atención deba resguardar los registros que usa y que retorne con RETI obliga a programarla en Assembler (al menor una parte), o a usar un que genera código especial.

4.6.10. Ejemplo



Transmisión serial usando interrupciones.

Rutina de atención de INT2:

```
<guardar registros>
while(n>0 && *port_ctrl & 2){
    n--;
    *buff++ = *port_data;
}
if(n==0)
    Procesar();
<restaurar registros>
RETI
```

4.6.11. Demora atención

- Desde el instante en que el dispositivo sube la interrupción hasta que se llama la rutina de atención, puede pasar bastante tiempo ya que:
 - las interrupciones pueden estar inhibidas
 - otros dispositivos pueden pedir interrumpir, pero se atienden de uno a la vez.
- Por ello se reciben todos los bytes que hayan llegado.
- Si el buffer interno de la puerta serial se llena antes de la interrupción ¡hay pérdida de bytes!

4.6.12. Evaluación

- Busy-waiting :
 - mínimo 10 instrucciones por byte transferido.
 - tasa máxima de transferencia = $\pm 1 \text{ MB/s}$ (1 millón de byte/s)
 - con 100 % ocupación de la CPU.
- Interrupciones:
 - ± 100 instrucciones por byte transferido.
 - tasa máxima de transferencia = $\pm 100 \text{ KB/s}$ (100 000 byte/s) si hay 1 byte/interrupción.
 - Ocupación de la CPU para una puerta de 2000 byte/s: $100 \text{ instr/byte} \cdot 2000 \text{ byte/s} = 200.000 \text{ instr/s} = 2\% \text{ de 10 MIPS} = 2\% \text{ de la CPU}$
 - ¡El 98 % restante se ocupa para otras tareas!

4.6.13. Problema

- Ej: un dispositivo USB2 lento que transfiere:
 - $16 \text{ Mbit/s} (\text{USB2 max} = 480 \text{ Mbit/s}) = 2 \text{ MB/s} (= \pm 2\,000\,000 \text{ bytes / s})$
 - Con busy-waiting: $2\,000\,000 \text{ bytes/s} \cdot 10 \text{ inst/byte} = 20\,000\,000 \text{ inst/s} = 20 \text{ MIPS} = 200\% \text{ de la CPU}$
 - Con interrupciones
 - Ciertos dispositivos pierden datos si no se les atiende dentro de un plazo prudente
 - Puede no cumplirse si se inhiben las interrupciones.

4.7. Direct Memory Access (DMA)

- Objetivos:
 - Desligar a la CPU de la transferencia de bloques de datos entre memoria y dispositivos
 - Permitir velocidades de transferencia acotadas por la velocidad de la memoria.
- Un controlador de DMA es un co-procesador dedicado a transferir datos entre memoria y dispositivos para liberar la CPU

4.7.1. Interfaz DMA

- Un dispositivo que usa se conecta con:
 - la línea ()
 - DACK (Data Acknowledge)
- El controlador de DMA tiene un numero fijo de pares DREQ/DACK.
- Solo se puede conectar un dispositivo por par.

4.7.2. Configuración DMA

- Cuando un programa (ej: un driver) necesita hacer una transferencia, configura el DMA con la siguiente información:
 - dirección de un búfer en memoria
 - numero n de bytes a transferir
 - numero del par DREQ/DACK que usa el dispositivo en cuestión
- El controlador de DMA posee varias puertas de control por donde recibe esta información.

4.7.3. Procedimiento DMA

Para transferir los datos ocurre lo siguiente:

1. El dispositivo coloca DERQ en 1.
2. El controlador DMA pide el bus a la CPU, usando una línea HOLD.
3. La CPU concluye cualquier operación que esté usando el bus, lleva todas sus líneas que lo conectan a un y coloca una línea HOLDA (HOLD ACK) en 1.
4. El DMA transfiere datos: coloca la dirección en el bus de direcciones, RD o WR en 1 y señala al dispositivo que escriba o lea el dato en el bus de datos (= un acceso simultáneo a la memoria y a una puerta de datos del dispositivo).
5. Si el dispositivo lleva la línea DREQ a 0, el DMA lleva HOLD a 0 y lleva sus líneas a tristate, la CPU lleva HOLDA a 0 y puede volver a ocupar el bus.
6. Si no, se trata de una transferencia en ráfaga (burst transfer), goto 4.

4.7.4. Observaciones

- La y comparten y .
- Mientras el DMA ocupa el bus, la CPU puede continuar ejecutando la instrucción de máquina en curso, pero si se necesita acceder al bus debe esperar a que el DMA suelte el bus.
- La transferencias que realiza el DMA son transparentes para el programa que ejecuta la .
- Al fin de la transferencia de n bytes, el DMA interrumpe la CPU usando una linea de interrupción.

4.7.5. Evaluación

- Costo de transferencia de un bloque de N bytes:
- Programacion del DMA (costo fijo)
 - 1 acceso a memoria y dispositivo por cada byte , por el controlador
 - 1 interrupción al final del bloque

4.8. Ejercicios propuestos

4.8.1. Problema 1

La figura muestra un conversor digital/análogo (D/A) y un comparador (CMP). El conversor digital/análogo recibe como entrada un número entero binario en D7-D0 (entre 0 y 255) y entrega en X una señal analógica consistente en un voltaje entre 0 y 12 V proporcional al valo de la entrada. El comparador recibe valores analógicos X e Y y los compara entregando un valor binario en 1 que indica que X es igual a Y, 0 en caso contrario.

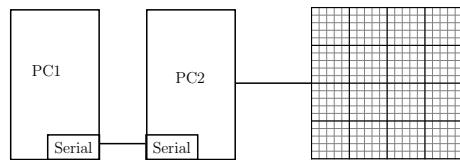
- a) Diseñe e implemente una interfaz de entrada/salida para ambas componentes. Considere un microprocesador con un bus de datos de 8 bits y un bus de direcciones de 16 bits. Haga que al escribir en la dirección 0xffff se establezca la entrada del conversor D/A (preocúpese de que esta entrada se mantenga constante hasta que se reescriba un nuevo valor) y al leer en la misma dirección se lea la salida GT de CMP. Las señales X e Y salen del sistema para ser conectadas por el usuario.
- b) Explique como un usuario puede usar estas componentes para convertir una señal analógica entre 0 y 12V en un valor binario entre 0 y 255. Escriba en C la rutina convertirAD() que hace la conversión (busque la simplicidad en esta rutina, no la eficiencia)

4.8.2. Problema 2

Se desea proveer una interfaz serial a un dispositivo consistente en una pantalla de 16×16 LEDs para poder conectarlo a cualquier computador. Para ello, se puede usar un computador con arquitectura M32 como controlador de entrada/salida (es un controlador sobredimensionado).

La pantalla permite prender un bit a la vez y por ello su entrada tiene 9 bits:

- 4 para especificar la fila
- 4 para especificar la columna
- 1 para el valor del LED (1= ON, 0=OFF)



- a) Diseñe la interfaz en el PC2 para conectar la pantalla a los buses. Su dirección de memoria debe ser **0x47A2**.
Además, conecte al PC2, empezando en la dirección **0x0000**, una memoria con el tamaño justo para guardar la imagen mostrada en la pantalla. Especifique las características de la memoria ¿Cuál es su dirección final?
- b) Implemente en el PC2 una función que lea los bytes recibidos en el puerto serial y los interprete de la siguiente manera: El primer byte es un comando 0x00 o 0x01 que significa respectivamente “apagar” o “encender”. El segundo byte indica la dirección del pixel a prender o apagar. Este cambio se hace en la memoria, en el lugar en que está almacenado el valor de dicho pixel (no directamente en la pantalla).

Capítulo 5

Arquitecturas avanzadas

5.1. Memoria caché

5.1.1. Problema

- La gasta mucho tiempo esperando el envío de datos desde y hacia la memoria.
- Aún cuando la CPU sea rápida, el acceso a memoria puede hacer mucho mas lenta la ejecución de un programa.
- ¿Cómo tener un acceso mas rápido a la memoria?
 - Memoria dinámica (DRAM): barata, pero lenta.
 - Memoria estática (SRAM): rápida, pero cara.

5.1.2. Experimento

- Descomponer la memoria requerida por un programa en líneas de 4, 16, 32 o 64 bytes de direcciones contiguas.
- Contabilizar el número de accesos a cada una de las líneas durante un intervalo de ejecución de un programa.
- Hacer un ranking de las líneas mas accesadas
- Hecho empírico 1: El 98 % de los accesos se concentran en el 3 % de las líneas.

5.1.3. Localidad espacial

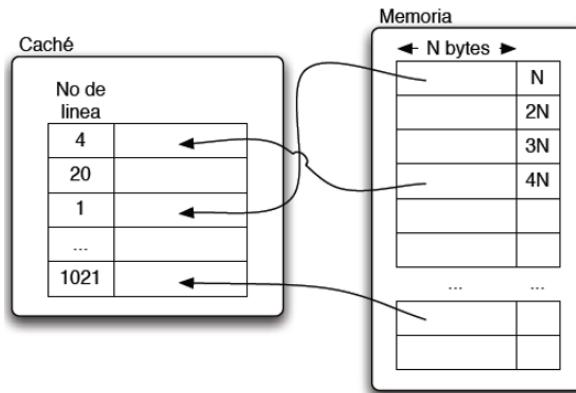
- Si se hace un acceso a un elemento de memoria, hay alta probabilidad de que se hará otro acceso a elementos que se encuentran cerca en el futuro próximo.
- Usando este fenómeno podemos tener accesos más rápidos a la memoria y por lo tanto una cpu más rápida.
- Idea 1
 - memoria estática (el caché) de pequeño tamaño (S_c) para guardar las líneas más accesadas ahora.
 - El resto se almacena en la memoria dinámica (Tamaño S_m , con $S_c \ll S_m$).

Problema

- ¿Cómo saber cuáles serán las líneas más accesadas durante el intervalo de ejecución?
- Hecho empírico 2:
 - un buen predictor de las líneas que serán más accesadas en el futuro cercano es el conjunto de las líneas que fueron accesadas en el pasado más reciente.

5.1.4. Localidad temporal

Si se hace un acceso a un elemento de memoria, hay alta probabilidad de que se haga otro acceso a este elemento en el futuro próximo.



Idea 2:

Guardar en el caché las ultimas líneas de la memoria que hayan sido accesadas.

Supongamos:

```
Tc = tiempo de acceso caché
! = +- 10 ms
Tm = Tiempo de acceso a memoria
! = 70 ms
% de éxito = hit rate = hr
! = 90 -> 99%
T acceso:
Tc <= hr*Tc+(1-hr)*Tm << Tm
```

El caché contiene una aproximación de las líneas más frecuentemente accesadas. Se requiere un campo adicional que almacene el número de línea que se guarda en una línea del cache. $Costo = Sc*Costo(SRAM) + Sm*Costo(DRAM) \Rightarrow Costo(DRAM)$ predomina

Problema

Problema: al accesar la memoria hay que “buscar” el número de linea en el caché. Para que sea eficiente hay que buscar en paralelo, lo cual es muy caro

5.1.5. Grados! de asociatividad

- full: una línea de la memoria puede ir en cualquier línea del caché
- 1: si el caché tiene L_c líneas, la línea l de la memoria solo puede ir en la linea $l \bmod L_c$ del cache
- $2n$: se usan $2n$ caches paralelos de 1 grado de asociatividad

5.1.6. Lectura

- Grado 1: Al leer la línea l se examina la etiqueta n de la línea $l \bmod L_c$ del caché.
 - Si $l = n$ éxito

- Si $l \neq n$ fracaso. Hay que recuperar la línea de la memoria dinámica y reemplazar la línea n por la l
- Grado $m > 1$: verificar $l = n$ en todos los cachés

5.1.7. Observaciones

- A igual tamaño de caché: mayor número de grados de asociatividad
 - mejor tasa de aciertos
 - mayor costo
- A igual grado de asociatividad: mayor tamaño del caché, mejor la tasa de aciertos
- En la práctica, a igual costo el óptimo en la tasa de aciertos se alcanza en 1, 2, 4 (a 8) grados de asociatividad.

5.1.8. Políticas para la escritura

- La estadística dice que se hacen 3 lecturas por una escritura
- Write-through: las escrituras se hacen en el caché y en la memoria simultáneamente, pero pueden hacerse sin bloquear a la CPU.
- Write-back: una escritura solo actualiza el caché. La memoria se actualiza cuando corresponde reemplazar esa línea por otra.

5.1.9. Ejemplo

- Un caché de 1 grado de asociatividad
- caché de 1024 bytes (1 KB)

```
char buff[128*1024];  
for(;;)  
    buff[0] + buff[64*1024]; /* línea mem buff[0] */  
                           = línea mem buff[64*1024]  
                           en el caché */
```

- Tasa de aciertos con respectos a los datos = 0 %

5.1.10. Eficiencia

- Normalmente, los programas no tienen comportamientos tan desfavorables.
- La eficiencia del cache dependerá de la localidad en los accesos del programa que se ejecuta.
- Esta localidad depende de las estructuras de datos que maneja el programa:
 - La pila: el acceso a variables locales tiene un hit rate que supera al 99 %
 - El código presenta muy buena localidad
 - El acceso secuencial de grandes arreglos o matrices (que superan el tamaño del caché) tienen mala localidad.

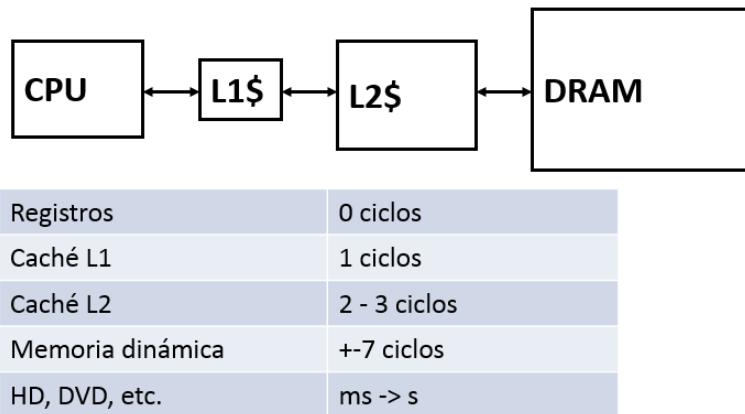
5.1.11. Problema

- Las variables locales y el código del ciclo podrían mapearse en el mismo lugar en el caché
- La velocidad de ejecución puede caer hasta un 50 %.
- Este problema no ocurre en un caché de 2 grados de asociatividad.
- Otra solución es tener un caché separado para:
 - código (instruction caché)
 - datos (data caché).

5.1.12. Jerarquía de memoria

Compromiso:

- Capacidad
- Desempeño



5.1.13. Jerarquía de Buses

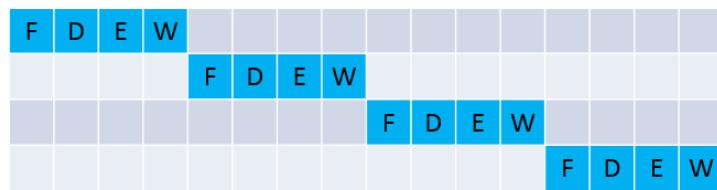
Una vez resuelto el problema de la velocidad de la memoria, el bus se transforma en el siguiente cuello de botella: Se pueden hacer buses económicos en donde es posible conectar CPU, memoria, DMA y todos los dispositivos. Sin embargo el resultado es un bus muy lento por que las señales tardan demasiado en propagarse. En efecto, para que las señales lleguen a todas las componentes es necesario agragar buffer y transciever que amplifican las señales, pero a su vez introducen un tiempo de retardo adicional. Por otro lado, se pueden hacer buses rápidos, pero que operan a distancias cortas (+- 10 → 20 cm) y con un número limitado de componentes conectadas al bus. Solución: Jerarquizar Las transferencias entre componentes se pueden ordenar:

5.2. Pipeling

5.2.1. Partes instrucción

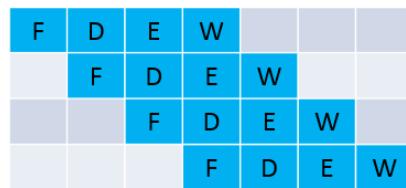
- Cada instrucción tiene distintas partes:

- F fetch
 - D decode
 - E execute
 - W write-back
- Normalmente, asumimos que cuando se empieza a ejecutar una instrucción, la anterior ya se terminó de ejecutar:



5.2.2. Pipeline

- Con la solución anterior, se puede ejecutar a lo mas 1 instrucción por cada N ciclos.
- Usando pipelining, las partes de distintas instrucciones se ejecutan en paralelo:



- En el caso ideal se podría tener 1 instrucción por ciclo

5.2.3. Hazards

- Son problemas que impiden llegar al paralelismo entre instrucciones.
- Hay 3 tipos:
 - Data Hazards
 - Structural Hazards
 - Control Hazards

5.2.4. Data Hazards

- Ocurren cuando distintas instrucciones modifican los mismos datos en distintas etapas del pipeline.
- Hay 3 subtipos:
 - Lectura después de escritura (RAW)
 - Escritura después de lectura (WAR)
 - Escritura después de escritura (WAW)

Lectura después de escritura (RAW)

- Una instrucción depende del resultado de una instrucción previa.
- Ej:

and R0, 0, R1	R1 = 0	F D E W
add R1, 4, R2	R2 = R1 + 4	F D E W
add R2, 10, R3	R3 = R2 + 10	F D E W

Escritura después de lectura (WAR)

- Un dato es cambiado después de ser utilizado
- Ej:

and R0, 0, R1	R1 = 0	F D E W
sub R1, 6, R2	R2 = R1 - 6	F D E W
add R2, 7, R3	R3 = R2 + 7	F D E W
add R1, 2, R2	R2 = R1 + 2	F D E W

Escritura después de escritura (WAW)

- Un dato es cambiado en dos instrucciones por lo que su valor final depende del orden de ellas
- Ej:

and R0, 0, R1	R1 = 0	F D E W
add R1, 12, R2	R2 = R1 + 12	F D E W
add R2, 5, R1	R1 = R2 + 5	F D E W

5.2.5. Structural Hazard

- Un mismo bus o componente de la CPU es requerida por dos instrucciones simultáneamente
- Ej:

ldw 0xFF43, R1	R1 = Mem(0xFF43)	F	D	E	W		
add R1, 4, R2	R2 = R1 + 4		F	D	E	W	
add R2, 10, R3	R3 = R2 + 10			F	D	E	W

5.2.6. Control Hazard

- Cuando hay un branch, no se sabe cual instrucción se ejecutara a continuación hasta después de evaluar la condición.
- Ej:

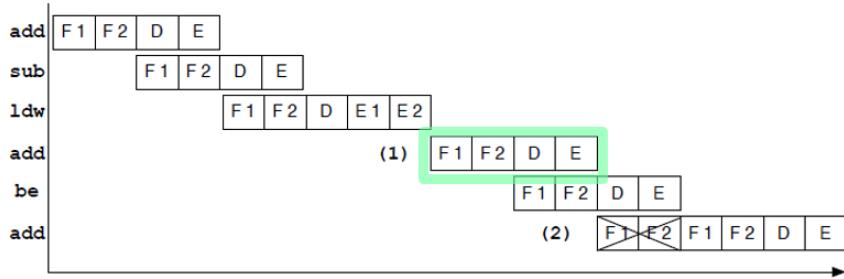
and R0, R2, R1	R1 = R2 \wedge R0	F	D	E	W		
be 64	If (z) PC=PC+64		F	D	E	W	
add R2, 10, R3	R3 = R2 + 10			F	D	E	W

5.2.7. Pipeling

Simultáneamente:

- Ejecución de la instrucción actual
- Carga de la siguiente instrucción

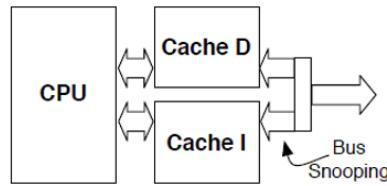
Solo se puede hacer un acceso a memoria a la vez. En este caso, load necesita accesar la memoria, por lo que la carga de la siguiente instrucción no puede hacerse en pipelining.



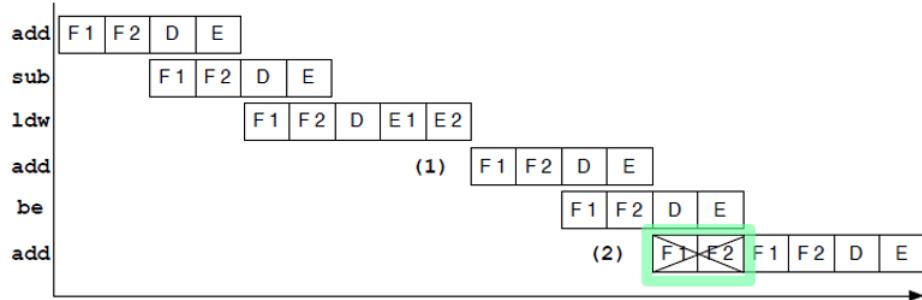
Los microprocesadores modernos incorporan dos buses (arquitectura de Harvard):

- uno para datos
- otro para instrucciones

Luego, si se puede cargar la siguiente instrucción en pipelining, siempre y cuando la instrucción este en el caché.



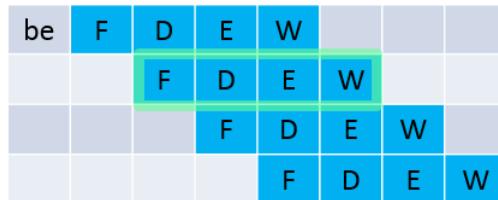
La siguiente instrucción se carga, pero no sirve, ya que el salto condicional previo se efectúa.



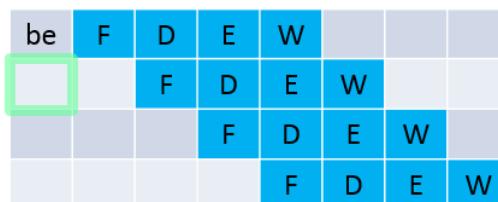
Para evitar que se tenga que botar instrucciones ya cargadas, en algunas arquitecturas se especifica que los saltos son retardados (delayed branch). La instrucción que sigue un salto se ejecuta aún cuando ocurra el salto.

5.2.8. Delayed Branches

- En los branches, la instrucción siguiente se ejecuta igual



- Es recomendable poner un nop después de un branch

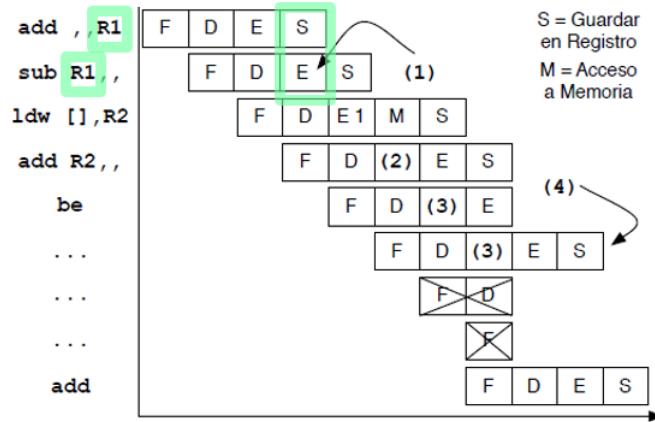


- Eventualmente, se puede poner otra instrucción (riesgoso), pero no otro branch ni set

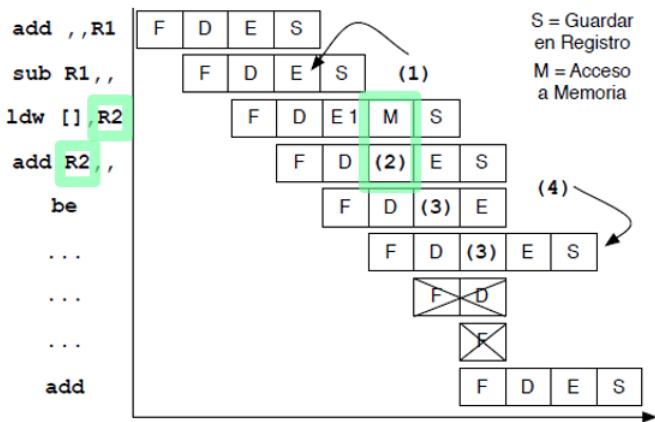
5.2.9. Niveles de pipelining

En la práctica se emplean más de 2 unidades para lograr ejecutar 1 instrucción por ciclo del reloj.

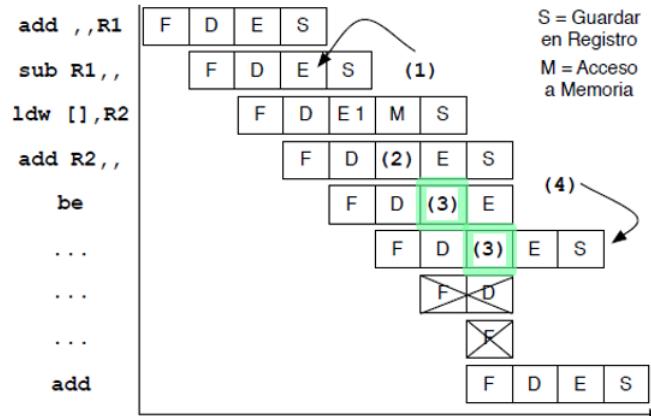
- Se usa el valor del registro calculado en la instrucción anterior, que todavía no termina de actualizarse. Register Bypassing es una técnica que lleva el resultado de una operación directamente a uno de los operadores de la siguiente instrucción.



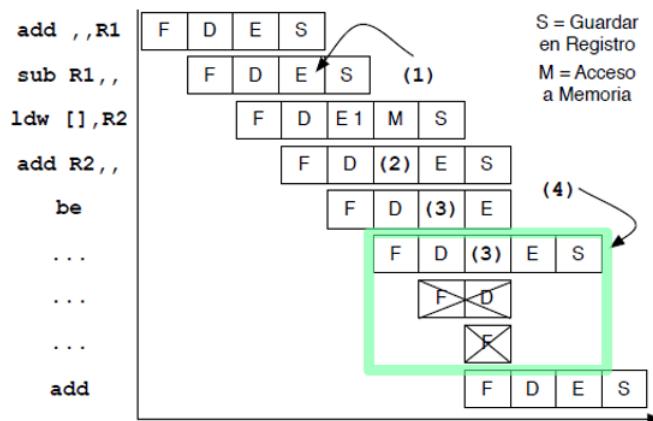
2. El valor de R2 se conocerá el final del ciclo M, se introduce un ciclo de relleno a la espera de R2. (Pipeline Stall, Data Hazard) Register Scoreboarding permite acordarse de los registros con valor desconocido.



3. Se debe esperar a que la unidad de ejecución se desocupe. (Pipeline Stall, Structural Hazard)



4. Se produce un salto por lo que se descarta el trabajo hecho con las 3 siguientes instrucciones. Los saltos son dañinos para la efectividad del pipeline (Branch Hazard)



5.2.10. Branch Prediction

- Con altos niveles de pipeline el trabajo que se descarta es inaceptable.
- Branch prediction es una técnica que intenta predecir saltos.
- Por ejemplo los saltos se almacenan en un Branch History Table.
- Al hacer la decodificación de una instrucción de salto almacenada se sigue cargando instrucciones a partir de la dirección de destino.
- Actualmente los CPU logran predecir correctamente mas del 95 % de los saltos.
- Todas estas técnicas permiten acercarse a la barrera de 1 instrucción por tick

5.3. Arquitecturas superescalares

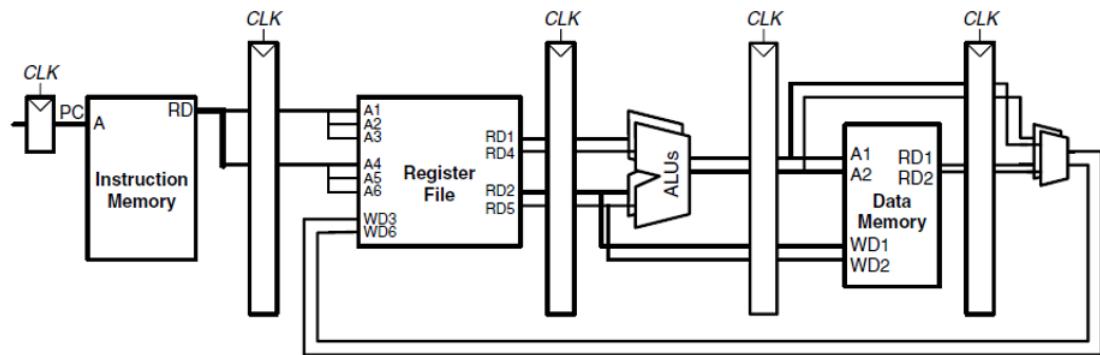
- Son aquellas donde se implementa dentro de la CPU al paralelismo al nivel de instrucciones.
- Pueden superar la barrera de 1 instrucción por ciclo.
- Para ello, hay unidades funcionales replicadas dentro del mismo core de CPU.

5.3.1. Grados de pipelining

- Es la cantidad de pipelines en paralelo que se pueden ejecutar.
- Ejemplo: superescalar de grado 2.
- Se colocan 2 pipelines en paralelo:
 - Se cargan 2 instrucciones a la vez
 - Se decodifican 2 instrucciones a la vez
 - Se ejecutan 2 instrucciones a la vez
 - Se actualizan 2 registros a la vez

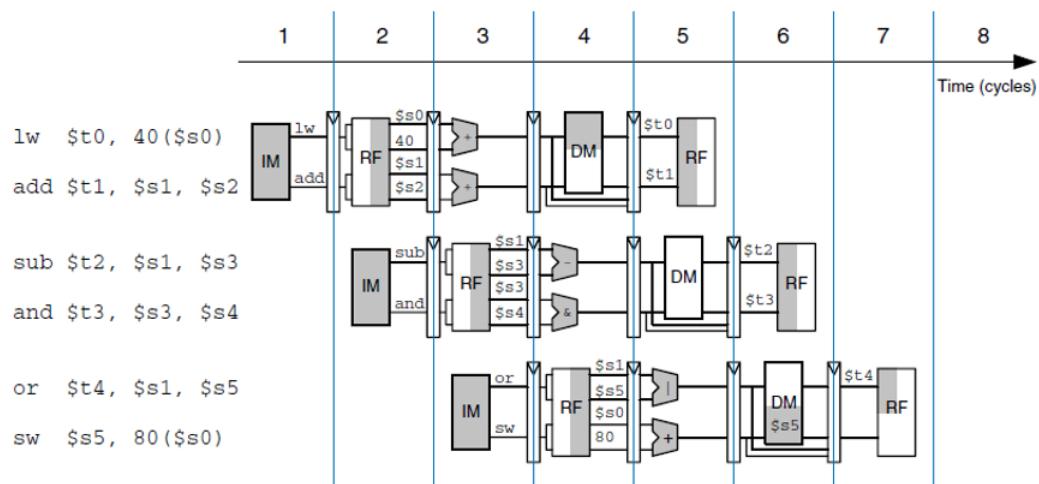
5.3.2. Implementación

Ruta de datos con un pipeline de grado 2:



5.3.3. Diagrama de pipeline

Ejemplo de diagrama de pipeline (grado 2):



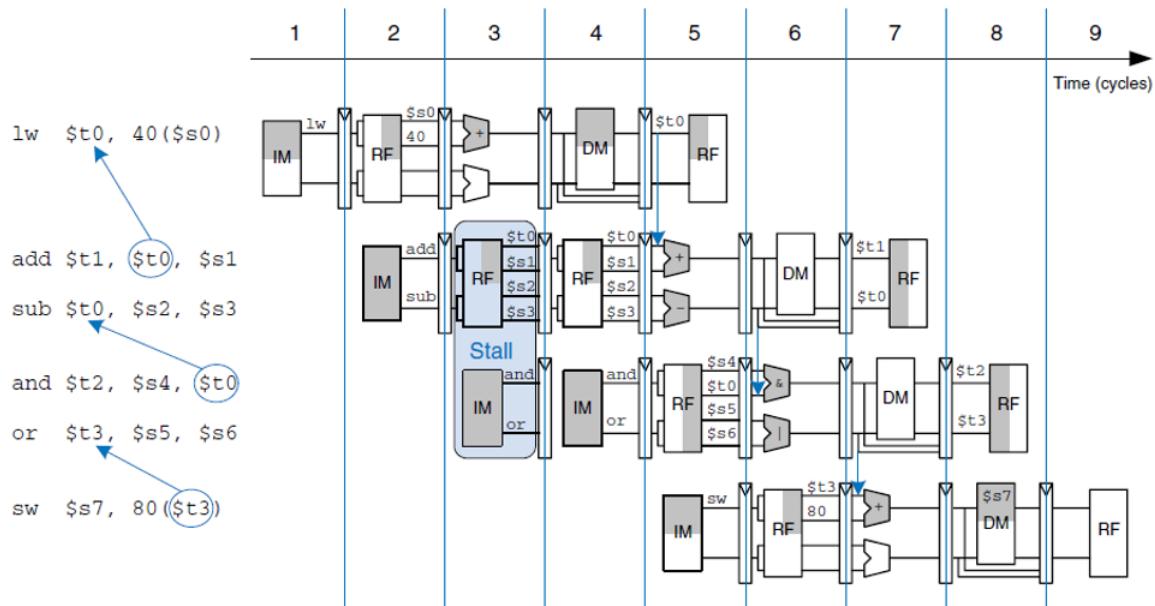
5.3.4. Dependencia

En ejecución, puede haber dependencia entre instrucciones que se desean ejecutar simultáneamente

```

lw $t0, 40($s0)
add $t1, $t0, %s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)

```



Solución

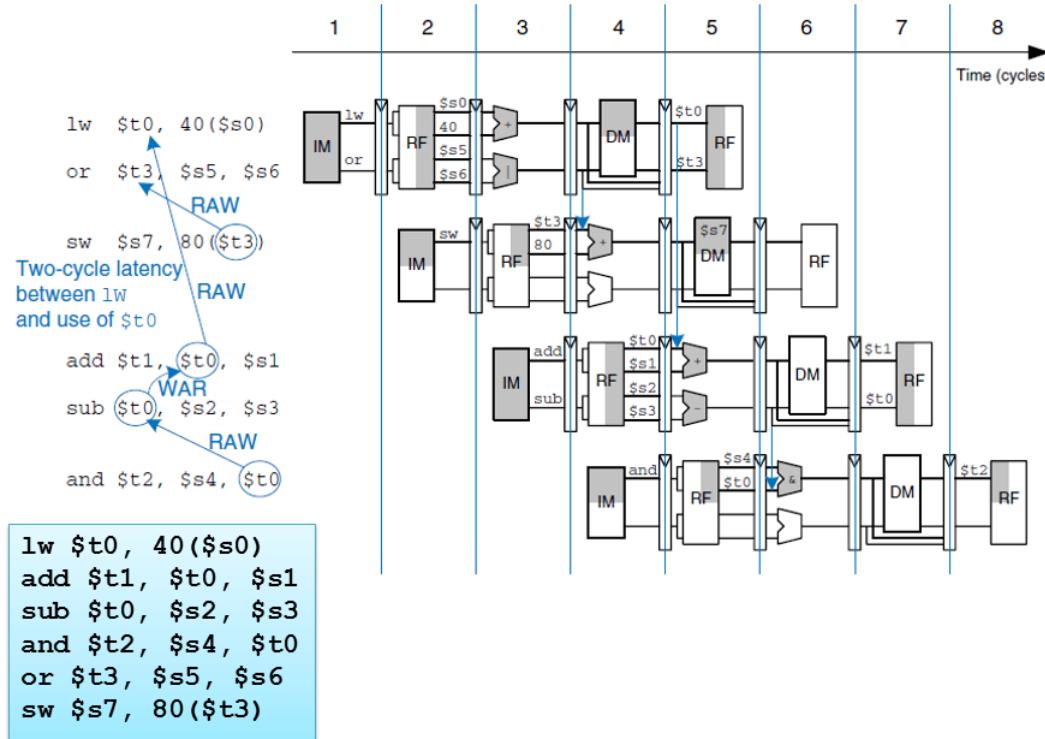
- En la etapa de ejecución:
 - Se analizan las dependencias de ambas instrucciones.
 - Si el resultado de la primera instrucción es un operando de la segunda (RAW): La ejecución de la segunda se descarta y se repite en el ciclo siguiente
- Los compiladores agregan una pasada que reordena las instrucciones de modo que no existan dependencias de datos.

Evaluación

- Ventajas: $1 < \text{IPC} \leq \text{numero de pipelines}$
- Desventajas:
 - Mucho mas alta complejidad de la CPU.
 - Hay que recompilar para limitar las dependencias.
- Ejemplos:
 - 486: 1 pipeline, 4 etapas (stages)
 - Pentium: 2 pipelines, 5 etapas
 - Pentium Pro, Pentium III: 3 pipelines, 10 etapas

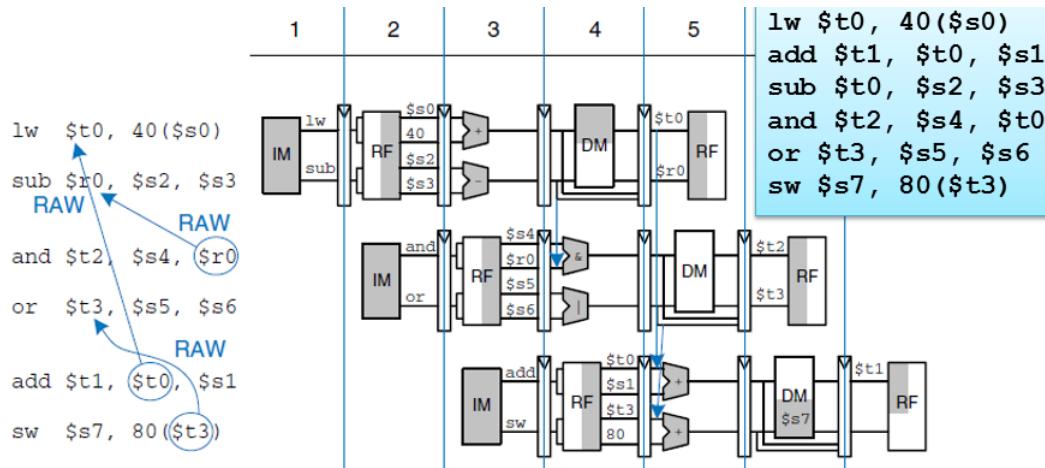
5.3.5. Ejecución fuera de orden

- Problema: Compiladores pueden reordenar las instrucciones, pero les falta conocimiento dinámico acerca de cuanto tomara un load, etc.
- Solución: Ejecución fuera de orden
- Una dependencia de datos bloquea la instrucción involucrada, pero no las que vienen a continuación si no existen dependencias.
- Se usa desde el Pentium Pro.



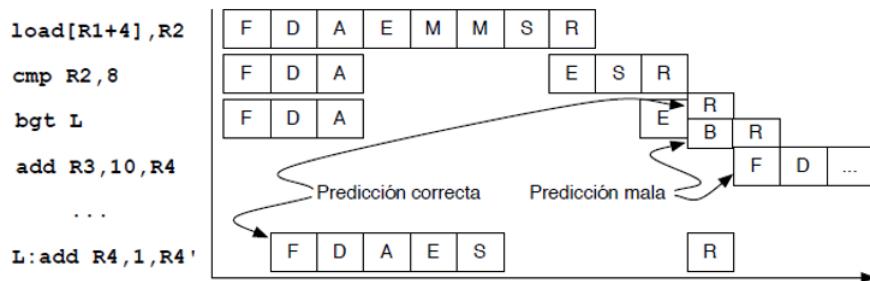
5.3.6. Register Renaming

- Ataca las dependencias de tipo WAR.
- Existen varios registros físicos por cada registro de la arquitectura



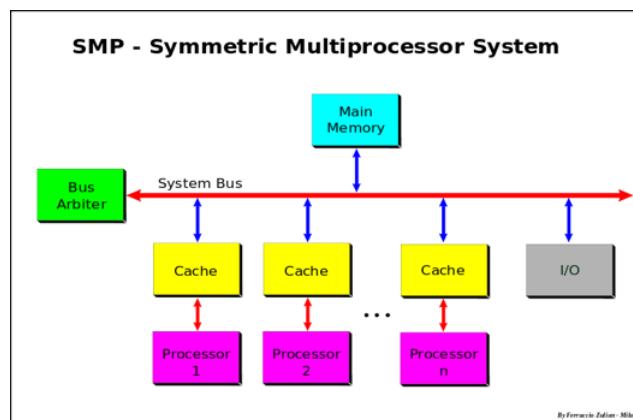
5.3.7. Ejecución especulativa

- Se ejecuta el código de un branch antes de saber si se llevará a cabo o no.
- En caso de no haber branch se restauran los nombres de los registros de modo que los registros modificados vuelven a su valor original.



5.4. Multi-core chips

5.4.1. Symmetric multiprocessing (SMP)



- Varias CPU en un computador
- Todas las CPU están conectados a una sola memoria central
- Cada CPU puede ejecutar un programa y trabajar con datos en cualquier lugar de memoria

5.4.2. Multi Core Chips

- Un chip multi-core contiene varios cores
- Cada core es casi un CPU completo, con incluso sus propios L1 y L2 caché (salvo primeras implementaciones).
- Un chip N-core puede correr N hilos de ejecución en paralelo, siempre que no usen recursos compartidos.
- Dado que el bus ya es lento para 1 CPU, ¡con varios CPU el problema es peor!

Problema

- Todos los cores tienen que ver la misma memoria
- Varios cores pueden tener la misma línea de memoria en sus propios cachés al mismo tiempo
 - ¿Qué pasa si un core escribe un valor en su caché (L1 o L2)?
- No es factible dar acceso directo y completo desde el caché de un CPU a otro CPU en sistemas SMP (por velocidad baja del bus).

5.4.3. Caché Coherence

- Mecanismo que permite que todos los cores operen con la misma memoria.
- La coherencia de cachés se obtiene si se cumplen las siguientes condiciones:
 - Si un procesador escribe en una dirección y luego lee en ella (sin que otro procesador haya escrito en el meantime) debe leer lo mismo que escribió
 - Si un procesador lee una dirección cuya última escritura fue hecha por otro procesador, debe leer el valor escrito por este.
 - Las escrituras deben ser secuenciadas, i.e. leídas en el orden que son escritas.

Implementación simple

Write-through cachés:

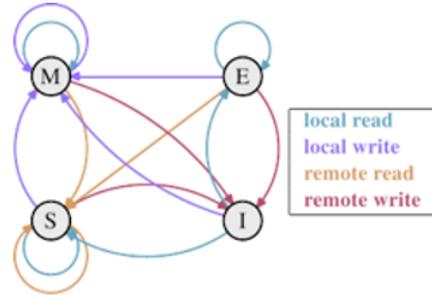
- Cada core C_i detecta cuando otro core C_o escribe un valor en su caché.
- Si esta línea también está en el caché de C_i , es marcada como sucia en C_i .
- Leer una línea sucia requiere cargar el nuevo valor desde la memoria central.

Implementación más sofisticada

- Usando bus snooping:
 - Leer una línea sucia significa que la memoria central no está up to date
 - El core C_o ve que el core C_i lee el valor cambiado usando bus snooping
 - Provee el nuevo valor directamente a C_i
 - En ese momento el memory controller también puede escribir este valor en memoria

5.4.4. MESI

- Protocolo ampliamente difundido
- Tiene 4 estados:
 - Modified: El procesador local ha modificado la línea. También implica que es la única copia dentro de los cachés.
 - Exclusive: La línea no ha sido modificada pero se sabe que es la única copia.
 - Shared: La línea no está modificada y podría existir en otros cachés.
 - Invalid: La línea tiene un contenido inválido (estado inicial)



	M	E	S	I
M	X	X	X	✓
E	X	X	X	✓
S	X	X	✓	✓
I	✓	✓	✓	✓

5.4.5. Request For Ownership (RFO)

- Operación especial
- Cuando otro core quiere escribir una línea con status (local) Modified, este core manda los datos al otro core y marca la línea (local) como Invalid
- Cuando este core quiere escribir una línea con status Shared, primero los otros core lo marcan como Invalid

5.4.6. Ejercicios resueltos

Problema 1

La figura muestra un extracto del contenido de un caché de 8 KB de 2 grados de asociatividad y líneas de 16 bytes. El cache se organiza en 2 bancos, cada uno con 256 líneas. Por ejemplo en la línea 3b (en hexadecimal) del banco izquierdo se almacena la línea de memoria que tiene como etiqueta 53b (es decir, la línea que va de la dirección 53b0 en hexadecimal a la dirección 53bf).

Linea	Banco 1		Banco 2		
	Cache	Etiqueta	Contenido	Etiqueta	Contenido
0a	b0a			80a	
3b	53b			13b	
81	481			681	

Cuadro 5.1: Diagrama del Caché

Un programa accede a las siguientes direcciones de memoria: b0a4, 13b0, 10a8, 4810, 6810, 4818, 0b0a, 080a. Indique qué accesos caen en la memoria caché y cuáles no. Además muestre un posible estado del caché después de aquellos accesos.

Solución:

Organizando los accesos a memoria quedaría:

	b0a4	13b0	10a8	4810	6810	4818	0b0a	080a
etiqueta cache	b0a	13b	10a	481	681	481	0b0	080
dir en cache	0a	3b	0a	81	81	81	b0	80
exito?	no	no	no	no	no	si	no	no

Para esta secuencia de accesos, una disposición de memoria caché seria:

Linea	Banco 1		Banco 2		
	Cache	Etiqueta	Contenido	Etiqueta	Contenido
0a	b0a			80a	
3b	53b			-	
80	080			-	
81	481			681	
b0	0b0			-	

Problema 2

La figura muestra a la izquierda un programa que se ejecuta en un procesador, con una implementación superescalar de grado 2 y 5 etapas en sus 2 pipelines: fetch, decode, analyze, execute y store. Acceder a la memoria toma 2 ciclos del reloj.

```
loop:
    shiftl R1, 2, R2
    load [R6+R2], R3
    add R1, 1, R1
    add R3, R4, R4
    cmp R1, R5
    blt loop
```

Instrucción	Ciclo				
	1	2	3	4	5
shiftl R1, 2, R2	F	D	A		
load [R6+R2], R3	F	D	A		
add R1, 1, R1		F	D		
add R3, R4, R4		F	D		

Cuadro 5.2: Proceso de Ejecución

Complete el diagrama de ejecución de la derecha, considerando 2 iteraciones del ciclo loop y una arquitectura superescalar con ejecución fuera de orden.

Luego indique en su diagrama (si es aplicable) en que momentos se usa (a) register bypassing, (b) register scoreboard, (c) register renaming, (d) predicción de saltos, y (e) ejecución especulativa.

Solución:

El diagrama de ejecución queda como sigue:

#	Instrucción	Ciclo												
		1	2	3	4	5	6	7	8	9	10	11	12	13
1	shiftl R1, 2, R2	F	D	A	S									
2	add R1, 1, R1	F	D	A	S									
3	load [R6+R2], R3		F	D	A	E	M	S						
4	add R3, R4, R4		F	D	A			E	S					
5	cmp R1, R5			F	D	A	E	S						
6	blt loop			F	D	A			E					
7	shiftl R1, 2, R2				F	F	D	A	E	S				
8	add R1, 1, R1				F	F	D	A		E	S			
9	load [R6+R2], R3						F	D	A	M	S			
10	add R3, R4, R4						F	D	A			E	S	
11	cmp R1, R5						F	D	A	E	S			
12	blt loop						F	D	A				E	

Cuadro 5.3: Proceso de ejecución de las instrucciones

La explicación del diagrama es la siguiente:

- De las instrucciones 1 a la 3 se realiza registry bypass para el registro R2.
- Entre las instrucciones 1 y 2 no es necesario el registry renaming.
- Entre las instrucciones 3 y 4 se realiza registry scoreboard del registro R3.
- en la instrucción 6 a la 7 y 8 se realiza una branch prediction. Las primeras fases fetch de las instrucciones 6 y 7 se eliminan por la predicción del branch.
- Entre las instrucciones 9 y 10 se realiza un registry scoreboard del registro R3.
- Observemos que las instrucciones de branch se ejecutan DESPUÉS de la fase de store de la instrucción de comparación, ya que es en esa fase en el cual se escribe el registro de flags, necesarios para determinar el branch.

Problema 3

La siguiente es una secuencia de direcciones de memoria (en hexadecimal) leídas por un procesador con una memoria cache de 4 KB (2^{12} bytes) de un grado de asociatividad:

5F30 6D18 5F30 7F30 6D10 7F30 5F30 6D10

El cache posee líneas de 16 bytes. Suponga que el cache está inicialmente vacío.

1. ¿Cuál es la porción de la dirección que se usa como etiqueta?
2. ¿Cuál la porción de la dirección que se usa para indexar el cache?
3. ¿Qué accesos a la memoria son aciertos en el cache y cuáles son desaciertos?

Problema 4

La tabla muestra ciclo por ciclo la ejecución de varias instrucciones en un procesador.

#	Instrucción	Ciclo														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	ADD R1, 4, R1	F	D	A	E	S										
b	LDW [R2+8], R3	F	D	A	E	M	M	M	M	M	S					
c	OR R1, 255, R4		F	D	A	E	S									
d	ADD R4, 1, R2		F	D	A		E	S								
e	CMP R3, 0			F	D	A						E				
f	BNE a.-			F	D	A							E			
a'	ADD R1, 4, R1				F	D	A							E	S	
b'	LDW [R2+8], R3				F	D	A							E	M	S

Cuadro 5.4: Proceso de ejecución de las instrucciones

1. Indique en qué momentos se recurre (si es que se recurre) a las siguientes técnicas: register bypassing, register scoreboard, ejecución superescalar y renombrado de registros.
2. Modifique la tabla de más arriba considerando que la predicción del salto f.- fue errónea. Invante sus propias instrucciones g.- y h.-. Indique solo las filas y columnas que hay que modificar en la tabla.

3. Rehaga la tabla de más arriba considerando un procesador de similares características pero con ejecución fuera de orden y ejecución especulativa. Explique además en qué momentos se recurre a renombre de registros y ejecución especulativa. No olvide agregar la fase de retiro en el pipeline.

5.5. Anexos

Los videos de las clases son los siguientes:

- Clase 22: Memoria caché
- Clase 23: Pipelining