DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Representación de la información

Números enteros

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Representando información.

Queremos representar una **magnitud** a través de un **sistema de representación**:

Finito soporte fijo, cantidad de elementos acotados

Composicional diversas magnitudes podrán representarse con un conjunto de elementos atómicos que deben ser fáciles de implementar y componer

Posicional la posición de cada `dígito` determina unívocamente en qué proporción modifica su valor a la magnitud total del número

El soporte formal lo encontraremos en las **bases de representación numérica**.

Bases.

En términos prácticos una base determina la **cantidad de símbolos distintos que podemos encontrar en un dígito dado** dentro de nuestra representación.

Una misma magnitud puede tener distintas representaciones en distintas bases. Por ejemplo la magnitud asociada al cuatro puede representarse como:

Base	**Valor**	**Notación**
2	100	$(100)_{(2)}$
3	11	$(11)_{(3)}$
10	4	$(4)_{(10)}$

Bases.

- En base 2, usamos los símbolos 0 y 1 y escribimos los naturales: 0, 1, 10, 11, 100, 101, 110...
- En base 3, usamos los símbolos 0, 1 y 2 y escribimos los naturales: 0, 1, 2, 10, 11, 12, 20...
- ...y así...

Bases más comunes

Base	Símbolos usados
2 (binario)	0, 1
8 (octal)	0, 1, 2, 3, 4, 5, 6, 7
10 (decimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16 (hexadecimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Cambios de bases

Recordemos que un **cambio de base** es una operación que transforma un número $n = [1, 1, 0, 1](1101_{(2)})$ representado como lista de símbolos para una base dada, por ejemplo 2 (binario) y lo representa en otra base, por ejemplo 10 (decimal) $[1, 3](13_{(10)})$.

$$1101_{(2)} \rightarrow 13_{(10)}$$

Podemos pensar que el cambio de base es una traducción entre dos formas de representar una misma magnitud.

¿Para qué queremos el cambio de base?

Para convertir una magnitud de una representación que nos resulta natural (base 10) a la forma en que representan y almacenan los datos en la computadora (base 2).

Cambios de bases: Teorema de la división.

La **división euclídea**, llamada también **teorema o algoritmo de la división** va a vincular una magnitud a y una base b diciendo que hay un *único cociente y resto* que permiten escribir la magnitud a en base al valor b.

Nos valemos de esto para aplicar operaciones sucesivas que descompongan a en base a valores $b^n, b^{n-1}, \ldots, b^1$, sabiendo que estos b^i se relacionan con el símbolo $\hat{a}_i$ que va en la posición i cuando queremos hacer el cambio de base: $a \rightarrow \hat{a}$.

Cambios de bases: Teorema de la división.

Teorema:
Sean $a, b \in \mathbb{Z}$ con $b \neq 0$.

Existen $q, r \in \mathbb{Z}$ con $0 \leq r < |b|$

tales que $a = b \times q + r$

Además, q y r son únicos *(de a pares)*.

¿Cómo lo usamos ?

$a = b \times q + r$

$a = (b \times q_1 + r_1) \times b + r$

$a = [(b \times q_2 + r_2) \times b + r_1] \times b + r$

Podemos continuar con la expansión hasta que $q_N < b$

$a = \{[(b \times q_N + r_N) \times b) + r_{N-1}] \times ...\} \times b + r \times 1$

Si distribuimos va a quedar:

$$a = q_N \times b^{N+1} + r_N \times b^N + ... + r_1 \times b + r \times b^0$$

Representación posicional:
Podemos ver que el primer elemento de cada producto es el que va a aparecer en los dígitos de nuestra representación posicional, donde:

$$a = q_N \times b^{N+1} + r_N \times b^N + ... + r_1 \times b + r \times b^0$$

Se puede escribir como:

q_N	r_N	$\ldots$	r_1	r

O de forma correcta, incluyendo la base:

$$(q_N r_N \ldots r_1 r)_{(b)}$$

Ejemplo:

$$27 = 2 \cdot 10^1 + 7 \cdot 10^0 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$27 = (27)_{(10)} = (11011)_{(2)}$$

Ejercitación
Escribir los siguientes números en binario, octal y hexadecimal.

- diez
- quinientos doce

Representación de la información

Representación finita

Cuando pensamos en los números y cuando los escribimos en la vida diaria, no contamos con una restricción evidente en la cantidad de dígitos con los que podemos operar. Pero en un soporte electrónico, como resulta ser el caso de la computadora, cada dato se representa con una cantidad finita de elementos. En el caso de los números podemos pensar que **lo que está acotado es la cantidad de dígitos que podemos emplear. Cada dígito va a poder tener tantos valores distintos como tenga la base**. En base 10 son 10 valores distintos, del 0 al 9, en base 2 son dos que van del 0 al 1.

Rango de representación

El rango de representación viene asociado al tipo de dato, hasta ahora vimos números naturales ($\mathbb{N}_0$), y a la cantidad de dígitos que podemos escribir. Por ahora la forma de computar el rango es aplicando el cálculo de combinaciones cruzadas, por ejemplo una tira de 4 dígitos:

$$(a_3 a_2 a_1 a_0)_{(b)}$$

Donde cada dígito puede tener valores entre 0 y $b - 1$, tiene un rango igual a:

$$b \times b \times b \times b = b^4$$

Rango de representación

$$b \times b \times b \times b = b^4$$

Por ejemplo, si representamos nuestros datos como naturales de ocho dígitos en base 2 tendremos:

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$$

Esto luego se conocerá como el rango de un entero sin signo en 8 bits, y va del 0 al 255.

Overflow

Si una magnitud a representar cae fuera del rango de representación, tenemos una situación que se conoce como **overflow** (desborde), ya que no hay forma de representarla en formato actual. Por ejemplo, para los ocho dígitos de base dos del ejemplo, la magnitud 27 es representable:

$$27 = (\mathbf{00011011})_{(2)}$$

Noten que completamos con 0 los dígitos a izquierda para escribir los ocho elementos de nuestra representación finita. Ahora, la magnitud 770 no es representable por quedar fuera del rango **overflow**:

$$770 = (11\mathbf{00000010})_{(2)}$$

Precisamos 10 dígitos como mínimo para representar la magnitud en base 2.

Tipos de datos

Hasta este punto vimos cómo

- Representar números naturales ($\mathbb{N}_0$)
- Interpretar una base númerica y realizar cambios de base

Los datos tienen su **información asociada** y su **tipo de dato**. En el caso que se presentó **la información asociada son los valores de cada dígito** y **el tipo de dato serían los naturales acotados**. El tipo nos indica cómo interpretar la información, en este caso **cómo vincular el dato con una magnitud** y qué operaciones podemos realizar con ella.

Tipos numéricos

Vamos a utilizar los siguientes tipos de datos para representar números naturales y enteros, todos a partir de la representación en base 2 (binaria), a saber:

- **Sin signo**
  representa únicamente números positivos
- **Signo + Magnitud**
  se usa el primer dígito (bit) para indicar el signo de la magnitud
- **Exceso** m
  Represento n como $m + n$.
  De esta manera, estamos desplazando la ubicación de la magnitud asociada al cero del comienzo del rango de representación a la posición m. Los valores a izquierda de m serán interpretados como negativos

Veamos ejemplos con rangos de representación para datos de 3 bits. El rango es:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Los datos del rango son siempre iguales, lo que va a cambiar van a ser las magnitudes asociadas a cada elemento (cómo los interpretamos). Los datos son:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
000	001	010	011	100	101	110	111

Tipos numéricos

Las magnitudes asociadas al rango serán:

Posición	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Dato	000	001	010	011	100	101	110	111
Sin signo	0	1	2	3	4	5	6	7
Signo+magnitud	0	1	2	3	−0	−1	−2	−3
Exceso m(m=2)	−2	−1	0	1	2	3	4	5

Noten que para signo+magnitud hay dos datos asociados al cero (el valor está desnormalizado).

- **Complemento a 2**
  - Los positivos se representan igual.
  - Para los números negativos **la forma práctica de saber la forma en que vamos a representarlos es la siguiente:** Si n es la magnitud a representar y $d_3 d_2 d_1 d_0$ los cuatro dígitos para representarlo en 4 bits, lo que haremos es invertir los bits de a uno (cambiar unos por ceros y ceros por unos) y sumar uno.
  - Por ejemplo -4 se representa como $inv(4) + 1$, en 5 bits sería: $inv(01000) + 1 = 10111 + 1 = 11000$.

- **Complemento a 2**
  - Los positivos se representan igual.
  - Otra forma de pensarlo, para los números negativos lo que se almacena es un complemento $\dot{n}$, a partir de:

$$\dot{n} = 2^k_{(10)} - n$$

Donde n es la magnitud interpretada, $\dot{n}$ el dato almacenado y k la cantidad de dígitos (o bits) utilizados para representar al número. O sea lo que guardamos es la resta entre el primer número que se sale del rango y el valor absoluto de n. Esto también equivale a restar la magnitud a cero y descartar el acarreo.

$$\boxed{\dot{n} = 2^k_{(10)} - n}$$

- **Ejemplo**

  Quiero representar el número $-2_{(10)}$ con $k = 3$ dígitos binarios.
  Entonces, $2^k_{(10)} \rightarrow 2^{3_{(10)}}_{(10)} = 8_{(10)} = 1000_{(2)}$
  Luego $\dot{2} = 1000_{(2)} - 2_{(10)}$ donde $\dot{2}$ es el complemento a 2 del número.
  Escribimos el 2 en la base correspondiente:
  $\dot{2} = 1000_{(2)} - 2_{(10)} \rightarrow \dot{2} = 1000_{(2)} - 10_{(2)}$.
  Finalmente $\boxed{\dot{2} = 110_{(2)}}$

Codificando...

En base 2, datos de 4 bits

	Signo + Magnitud	Complemento a 2	Exceso a 15
3	0011	0011	OVERFLOW
-2	1010	1110	1101
-8	OVERFLOW	1000	0111

Codificando...(con más bits)

En base 2, datos de 8 bits

	Signo + Magnitud	Complemento a 2	Exceso a 15
3	0000 0011	0000 0011	0001 0010
-2	1000 0010	1111 1110	0000 1101
-8	1000 1000	1111 1000	0000 0111

Codificando...(con más bits)

Similitudes entre 4 y 8 bits

	Signo + Magnitud	Complemento a 2	Exceso a 15
3	0000 0011	0000 0011	0001 0010
-2	1000 0010	1111 1110	0000 1101
-8	1000 1000	1111 1000	0000 0111

Extendiendo la cantidad de bits de precisión:

- Signo + Magnitud: Se extiende con 0's, pero el bit más significativo se mantiene indicando el signo.
- Complemento a 2: Se extiende con el valor del bit más significativo.
- Exceso a m: Se extiende siempre con 0's.

Resumiendo: Enteros como numerales binarios

Sin Signo

Solo sirve para los positivos.

Numeral(dato) $\rightarrow$ número que representa

$1111 \rightarrow 15_{(10)}$
$1110 \rightarrow 14_{(10)}$
$1101 \rightarrow 13_{(10)}$
$1100 \rightarrow 12_{(10)}$
$1011 \rightarrow 11_{(10)}$
$1010 \rightarrow 10_{(10)}$
$1001 \rightarrow 9_{(10)}$
$1000 \rightarrow 8_{(10)}$

$0111 \rightarrow 7_{(10)}$
$0110 \rightarrow 6_{(10)}$
$0101 \rightarrow 5_{(10)}$
$0100 \rightarrow 4_{(10)}$
$0011 \rightarrow 3_{(10)}$
$0010 \rightarrow 2_{(10)}$
$0001 \rightarrow 1_{(10)}$
$0000 \rightarrow 0_{(10)}$

Para los numerales de 4 *bits*.

Resumiendo: Enteros como numerales binarios

Signo+Magnitud

El primer bit es el signo, los demás son el *significado* (la magnitud del número en valor absoluto).

numeral $\rightarrow$ número que representa

$1111 \rightarrow -7_{(10)}$
$1110 \rightarrow -6_{(10)}$
$1101 \rightarrow -5_{(10)}$
$1100 \rightarrow -4_{(10)}$
$1011 \rightarrow -3_{(10)}$
$1010 \rightarrow -2_{(10)}$
$1001 \rightarrow -1_{(10)}$
$1000 \rightarrow -0_{(10)}$

$0111 \rightarrow 7_{(10)}$
$0110 \rightarrow 6_{(10)}$
$0101 \rightarrow 5_{(10)}$
$0100 \rightarrow 4_{(10)}$
$0011 \rightarrow 3_{(10)}$
$0010 \rightarrow 2_{(10)}$
$0001 \rightarrow 1_{(10)}$
$0000 \rightarrow 0_{(10)}$

Para los numerales de 4 *bits*.

Resumiendo: Enteros como numerales binarios

Complemento a dos

Los numerales que representa positivos son iguales a los anteriores

Para los negativos, dado un $\dot{n}$ negativo se representan escribiendo

$2^k - n$ en notación sin signo

cuentas $\rightarrow$ numeral $\rightarrow$ número que representa

$2^4 + (-1) = 15 \rightarrow$	$1111 \rightarrow -1_{(10)}$	$0111 \rightarrow 7_{(10)}$
$2^4 + (-2) = 14 \rightarrow$	$1110 \rightarrow -2_{(10)}$	$0110 \rightarrow 6_{(10)}$
$2^4 + (-3) = 13 \rightarrow$	$1101 \rightarrow -3_{(10)}$	$0101 \rightarrow 5_{(10)}$
$2^4 + (-4) = 12 \rightarrow$	$1100 \rightarrow -4_{(10)}$	$0100 \rightarrow 4_{(10)}$
$2^4 + (-5) = 11 \rightarrow$	$1011 \rightarrow -5_{(10)}$	$0011 \rightarrow 3_{(10)}$
$2^4 + (-6) = 10 \rightarrow$	$1010 \rightarrow -6_{(10)}$	$0010 \rightarrow 2_{(10)}$
$2^4 + (-7) = 9 \rightarrow$	$1001 \rightarrow -7_{(10)}$	$0001 \rightarrow 1_{(10)}$
$2^4 + (-8) = 8 \rightarrow$	$1000 \rightarrow -8_{(10)}$	$0000 \rightarrow 0_{(10)}$

Para los numerales de 4 *bits*.

Resumiendo: Enteros como numerales binarios

Exceso a m

El número n se representa como $m + n$

cuentas $\rightarrow$ numeral $\rightarrow$ número que representa

$5 + (10) = 15 \rightarrow 1111 \rightarrow 10_{(10)}$
$5 + (9) = 14 \rightarrow 1110 \rightarrow 9_{(10)}$
$5 + (8) = 13 \rightarrow 1101 \rightarrow 8_{(10)}$
$5 + (7) = 12 \rightarrow 1100 \rightarrow 7_{(10)}$
$5 + (6) = 11 \rightarrow 1011 \rightarrow 6_{(10)}$
$5 + (5) = 10 \rightarrow 1010 \rightarrow 5_{(10)}$
$5 + (4) = 9 \rightarrow 1001 \rightarrow 4_{(10)}$
$5 + (3) = 8 \rightarrow 1000 \rightarrow 3_{(10)}$

$5 + (2) = 7 \rightarrow 0111 \rightarrow 2_{(10)}$
$5 + (1) = 6 \rightarrow 0110 \rightarrow 1_{(10)}$
$5 + (0) = 5 \rightarrow 0101 \rightarrow 0_{(10)}$
$5 + (-1) = 4 \rightarrow 0100 \rightarrow -1_{(10)}$
$5 + (-2) = 3 \rightarrow 0011 \rightarrow -2_{(10)}$
$5 + (-3) = 2 \rightarrow 0010 \rightarrow -3_{(10)}$
$5 + (-4) = 1 \rightarrow 0001 \rightarrow -4_{(10)}$
$5 + (-5) = 0 \rightarrow 0000 \rightarrow -5_{(10)}$

Para los numerales de 4 *bits* en exceso 5.

Leyendo datos

Para interpretar un valor, o sea una tira de valores binarios o bits, es necesario conocer su tipo. Tipos distintos para un mismo valor determinan (potencialmente) distintas magnitudes.

Transformando los datos

Operaciones aritmético-lógicas

Vamos a presentar algunas operaciones aritméticas y lógicas en base 2, en particular **o lógico, y lógico, xor lógico, negación lógica, desplazamientos, suma, resta y multiplicación**. Todas las operaciones van a estar asociadas a un algoritmo de resolución, del mismo modo que utilizamos uno para realizar cuentas en base 10 en la vida cotidiana.

Propiedades de los datos y las operaciones

Es importante comprender que hay atributos que nos interesa observar tanto de **los datos como de las operaciones**. Por ejemplo:

- **De los datos numéricos** si son negativos o pares
- **De las operaciones** si los resultados se mantienen dentro del rango de representación

Propiedades de los datos

Veamos como observar el dato nos permite determinar propiedades del valor representado. Por ejemplo, en complemento a 2 de un dato de 4 bits:

Posición	v_3	v_2	v_1	v_0
Interpretación	*signo*	x	x	*paridad*
Negativo	1	x	x	x
Par	x	x	x	0

Operaciones lógicas

Las operaciones lógicas que veremos pueden involucrar a uno o dos operandos, **se aplican sobre el dato almacenado, o sea los bits del valor representado**.

Las presentamos rápidamente y una por una.

O lógico

El **o logico** o disyunción se aplica bit a bit. La operación $a \vee b = c$ se puede describir atómicamente (por cada elemento indivisible) como $c_i = a_i \vee b_i$. Veamos un ejemplo de 4 bits, noten que la aplicación no depende del tipo de dato, lo trata indistintamente:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \vee b$	1	0	1	1

Nota

Podemos notar que las operaciones lógicas se aplican **bit a bit**:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
	$+$	$+$	$+$	$+$
b	0	0	1	1
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$c = a \vee b$	1	0	1	1

Es posible encontrar al signo $+$ para representar la disyunción, porque equivale a una suma sin acarreo en bits. Lo mismo sucede entre la conjunción y el signo $*$.

Y lógico

El **y lógico** se aplica bit a bit. La operación $a \wedge b = c$ se puede describir atómicamente (por cada elemento indivisible) como $c_i = a_i \wedge b_i$. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \wedge b$	0	0	1	0

Xor lógico

El **xor lógico** se aplica bit a bit. La operación $a \veebar b = c$ se puede describir atómicamente (por cada elemento indivisible) como $c_i = (a_i \wedge \neg b_i) \vee (\neg a_i \wedge b_i)$, exactamente uno de los bits vale 1. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \veebar b$	1	0	0	1

Negación lógica

La **negación lógica** se aplica bit a bit. La operación $\neg a = c$ se puede describir atómicamente (por cada elemento indivisible) como $c_i = \neg a_i$, dando vuelta los valores de cada elemnto. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = \neg a$	0	1	0	1

Desplazamiento a izquierda

El **desplazamiento a izquierda** se aplica desplazando los bits del dato tantas posiciones como se indiquen a izquierda. La operación $a \ll n = c$ para un dato de k bits se puede describir atómicamente como $c_i = a_{i-n}$ si $i \leq k - n - 1$ y 0 en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \ll 2$	1	0	0	0

Desplazamiento lógico a derecha

El **desplazamiento lógico a derecha** se aplica desplazando los bits del dato tantas posiciones como se indiquen a derecha. La operación $a \gg_l n = c$ para un dato de k bits se puede describir atómicamente como $c_i = a_{i+n}$ si $i \geq n$ y 0 en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \gg_l 2$	0	0	1	0

Desplazamiento aritmético a derecha

El **desplazamiento aritmético a derecha** se aplica desplazando los bits del dato tantas posiciones como se indiquen a derecha, pero copiando el valor del bit más significativo de origen en los valores vacantes del resultado. La operación $a \gg_a n = c$ para un dato de k bits se puede describir atómicamente como $c_i = a_{i+n}$ si $i \geq n$ y a_{k-1} en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \gg_a 2$	1	1	1	0

Desplazamiento a derecha, observaciones

¿Por qué existe una diferencia entre el desplazamiento a derecha lógico y aritmético? La aplicación de una operación lógica puede tener muchos motivos, pero en particular el desplazamiento a derecha o izquierda en n posiciones **tiene el efecto de multiplicar o dividir por 2^n el valor representado**, siempre que se trate de un entero sin signo o en complemento a dos.

Desplazamiento a derecha, ejercicio

Si a es un valor representado en complemento a dos, ¿cuándo valen las siguientes propiedades?

$$a \gg_l n = a/2^n$$

$$a \gg_a n = a/2^n$$

Ejercicio para la casa.

Adición Binaria

Para realizar operaciones aritméticas como la suma, la resta o la multiplicación, podemos intentar utilizar dos conceptos importantes, por un lado el **razonamiento composicional** y por el otro un **análisis extensivos de los casos**. El primero tiene que ver con tratar de resolver la suma representada en base 2 a partir de cada dígito (descomposición) y el segundo con construir una tabla que calcule todos los resultados posibles para sumas de un dígito en base 2 (análisis extensivo).

Adición Binaria

Veamos que puede suceder cuando sumamos dos dígitos cualesquiera en base dos:

a_i	b_i	$a_i + b_i$
0	0	0
0	1	1
1	0	1
1	1	**1**0

Podemos notar que la última fila produce un resultado que no se puede representar con un sólo dígito. Para el caso atómico (de un único dígito) ese 1 va conocerse como **carry** o acarreo porque va a afectar la suma del dígito inmediato a izquierda.

Adición Binaria

Redefinimos el resultado, diviendo $a_i + b_i$ entre el resto r_i y su acarreo c_i:

a_i	b_i	c_i	r_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adición Binaria

Podemos extender la suma, para tener en cuenta el hecho de que quizás el dígito inmediato a derecha produjo acarreo en su suma, donde c_i es el carry (acarreo) del dígito anterior:

a_i	b_i	c_{i-1}	c_i	r_i
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Adición Binaria

Con esto ya podemos definir la adición binaria para dos números a y b sin signo de n dígitos de la siguiente forma:

$carry:$	c_{n-1}	c_{n-2}	$\ldots$	c_0	
$a:$	$-$	a_{n-1}	$\ldots$	a_1	a_0
$b:$	$-$	b_{n-1}	$\ldots$	b_1	b_0
$a+b:$	c_{n-1}	r_{n-1}	$\ldots$	r_1	r_0

Aquí c_n es lo que consideramos el acarreo de la suma, y se puede interpretar como un indicador o **flag** de la propiedad de desborde de la operación, ya que el resultado no es representable en n bits.

Ejemplo de la suma

En decimal	En base 3
$8\,4\,5$	$2\,1\,2$
$+\,3\,4\,2$	$+\,1\,0\,1$
$1\,1\,8\,7$	$1\,0\,2\,0$

Podemos definir equivalentemente la resta binaria para dos números a y b sin signo de n dígitos de la siguiente forma:

$borrow$:	b_{n-1}	b_{n-2}	$\ldots$	b_0	
a :	$-$	a_{n-1}	$\ldots$	a_1	a_0
b :	$-$	b_{n-1}	$\ldots$	b_1	b_0
$a - b$:	b_{n-1}	r_{n-1}	$\ldots$	r_1	r_0

Aquí b_n es lo que consideramos el préstamo o *borrow* de la resta. En este caso el *borrow* se produce cuando el sustraendo es mayor que el minuendo, por lo tanto, se debe "pedir" al dígito adyacente.

Resta y multiplicación

Como **tarea para el hogar**, intenten describir primero la operación de resta, aplicando descomposición y análisis extensivo de los casos a partir de la resta de a un dígito de números enteros sin signo.
Y luego expliquen cómo debería resolverse la multiplicación entre dos números de n bits sin signo, no es necesario que se restrinjan a la descomposición y al análisis extensivo, pueden suponer que se puede computar un resultado final a partir del cómputo de resultados parciales.

Técnicas de uso corriente

¿Cómo pasamos rápido de decimal a binario?

Pasar el número 28 y a binario.

Si hacemos divisiones sucesivas:

$28 \;/\; 2 \quad = 14 \quad \text{Resto} = 0$

$14 \;/\; 2 \quad = 7 \quad \text{Resto} = 0$

$7 \;/\; 2 \quad = 3 \quad \text{Resto} = 1$

$3 \;/\; 2 \quad = 1 \quad \text{Resto} = 1$

Quedando así el número 11100.

Otra aproximación.

Busco $2^x \geq 28 \rightarrow 5$, ya que $2^5 = 32$. Necesitaré 5 dígitos

Sabemos:

$x_1 * 2^4 + x_2 * 2^3 + x_3 * 2^2 + x_4 * 2^1 + x_5 * 2^0$

1	1	1	0	0

¿Cómo pasamos rápido de binario a hexa?

Pasar el número 10101100

- ¿Cuánto elementos puedo representar con un dígito hexadecimal?
- 16
- Luego es potencia de 2 directo, 2^4
- o sea puedo separarlos de a 4:

1010	1100
A	C

¿Y para pasar de hexadecimal a binario?

¿Y para pasar a complemento a 2? Hay que hacer la guía...

Sobre el Overflow y el Carry...

El truco para detectar un overflow es observar que si el bit de signo es igual en ambos operandos (ambos positivos o negativos) el resultado de la suma debería preservar el signo (suma de positivos produce un positivo, suma de negativos produce un negativo).

$$overflow \iff ((a_{n-1} = b_{n-1}) \wedge (a_{n-1} \neq c_{n-1}))$$

Sobre el Overflow y el Carry...

Algunos ejemplos para pensar **en C2** (complemento a 2)...

$$5 - 3 = 5 + (-3) = 2$$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 1\ 1\ 0\ 1 \quad (C \leftarrow C) \\ \hline 1\ 0\ 0\ 1\ 0 \end{array}$$

OK!

$$-5 - 3 = (-5) + (-3) = (-8)$$

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\ 1\ 1\ 0\ 1 \quad (C \leftarrow C) \\ \hline 1\ 1\ 0\ 0\ 0 \end{array}$$

OK!

$$-5 - 4 = (-5) + (-4) = -9$$

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1 \end{array}$$

$(C \leftarrow \overline{C})$

OVERFLOW!

$$5 + 4 = 9$$

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 1\ 0\ 0 \quad (\overline{C} \leftarrow C) \\ \hline 0\ 1\ 0\ 0\ 1 \end{array}$$

OVERFLOW!

Sobre el Overflow y el Carry...

Noten que el **overflow** y el **carry** son propiedades de una operación, a diferencia de la paridad o el signo, que son propiedades de un dato aislado. Para determinar si se cumple una propiedad de una operación, puede ser necesario observar tanto los operandos como el resultado (caso del overflow).

Cierre

Repaso de la clase

Resumen de la clase práctica del día de la fecha.

- Motivación, características y necesidad de contar con un sistema de representación de la información sobre un soporte dado. **Representación numérica en base 2**.
- Formas de representar tipos numéricos, **naturales (sin signo), enteros (signo+magnitud, exceso m, complemento a 2) y vistazo de racionales(IEEE 754)**.
- Operaciones lógico-aritméticas utilizadas para transformar la información (**o lógico, y lógico, xor lógico, negación lógica, desplazamientos, suma, resta y multiplicación**)y sus propiedades asociadas (**carry y overflow**).

Próximas clases

- ¡Ya se puede hacer la práctica 1 completa!.
- Circuitos Combinatorios.
- Circuitos Secuenciales.

Cierre

Pasamos a las preguntas.

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Lógica Digital - Circuitos Combinatorios

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Repaso: Algebra de Boole

Axiomas

Partimos de las siguientes proposiciones (axiomas):

Axiomas

Partimos de las siguientes proposiciones (axiomas):

(A1) Existen dos elementos: $X = 1$ *si* $X \neq 0$ *ó* $X = 0$ *si* $X \neq 1$

Axiomas

Partimos de las siguientes proposiciones (axiomas):

(A1) Existen dos elementos: $X = 1$ *si* $X \neq 0$ *ó* $X = 0$ *si* $X \neq 1$

(A2) Existe el operador negación $\overline{()}$ tal que: *Si* $X = 1 \Rightarrow \overline{X} = 0$

Axiomas

Partimos de las siguientes proposiciones (axiomas):

(A1) Existen dos elementos: $X = 1$ *si* $X \neq 0$ *ó* $X = 0$ *si* $X \neq 1$

(A2) Existe el operador negación $\overline{()}$ tal que: *Si* $X = 1 \Rightarrow \overline{X} = 0$

(A3) $0 \cdot 0 = 0 \qquad 1 + 1 = 1$

Axiomas

Partimos de las siguientes proposiciones (axiomas):

(A1) Existen dos elementos: $X = 1$ *si* $X \neq 0$ *ó* $X = 0$ *si* $X \neq 1$

(A2) Existe el operador negación $\overline{()}$ tal que: *Si* $X = 1 \Rightarrow \overline{X} = 0$

(A3) $0 \cdot 0 = 0 \qquad 1 + 1 = 1$

(A4) $1 \cdot 1 = 1 \qquad 0 + 0 = 0$

Axiomas

Partimos de las siguientes proposiciones (axiomas):

(A1) Existen dos elementos: $X = 1$ *si* $X \neq 0$ *ó* $X = 0$ *si* $X \neq 1$

(A2) Existe el operador negación $\overline{()}$ tal que: *Si* $X = 1 \Rightarrow \overline{X} = 0$

(A3) $0 \cdot 0 = 0 \qquad 1 + 1 = 1$

(A4) $1 \cdot 1 = 1 \qquad 0 + 0 = 0$

(A5) $0 \cdot 1 = 1 \cdot 0 = 0 \qquad 0 + 1 = 1 + 0 = 1$

Propiedades

De los axiomas anteriores se derivan las siguientes propiedades:

Propiedad	AND	OR
Identidad	$1.A = A$	$0 + A = A$
Nulo	$0.A = 0$	$1 + A = 1$
Idempotencia	$A.A = A$	$A + A = A$
Inverso	$A.\overline{A} = 0$	$A + \overline{A} = 1$
Conmutatividad	$A.B = B.A$	$A + B = B + A$
Asociatividad	$(A.B).C = A.(B.C)$	$(A + B) + C = A + (B + C)$
Distributividad	$A + (B.C) = (A + B).(A + C)$	$A.(B + C) = A.B + A.C$
Absorción	$A.(A + B) = A$	$A + A.B = A$
De Morgan	$\overline{A.B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}.\overline{B}$

Propiedades

De los axiomas anteriores se derivan las siguientes propiedades:

Propiedad	AND	OR
Identidad	$1.A = A$	$0 + A = A$
Nulo	$0.A = 0$	$1 + A = 1$
Idempotencia	$A.A = A$	$A + A = A$
Inverso	$A.\overline{A} = 0$	$A + \overline{A} = 1$
Conmutatividad	$A.B = B.A$	$A + B = B + A$
Asociatividad	$(A.B).C = A.(B.C)$	$(A + B) + C = A + (B + C)$
Distributividad	$A + (B.C) = (A + B).(A + C)$	$A.(B + C) = A.B + A.C$
Absorción	$A.(A + B) = A$	$A + A.B = A$
De Morgan	$\overline{A.B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}.\overline{B}$

Tarea: ¡Demostrarlas!

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}$$

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}$$

Solución:

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)} \longleftarrow$ De Morgan

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}$$

Solución:

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)} \longleftarrow$ De Morgan

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z} \longleftarrow$ Distributiva

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}}$$

Solución:

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}} \longleftarrow$ De Morgan

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z}} \longleftarrow$ Distributiva

$\overline{(\overline{X} \cdot Y)} \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ De Morgan

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}}$$

Solución:

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}} \longleftarrow$ De Morgan

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z}} \longleftarrow$ Distributiva

$\overline{(\overline{X} \cdot Y)} \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ De Morgan

$(X + \overline{Y}) \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ Distributiva

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}$$

Solución:

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)} \longleftarrow$ De Morgan

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z} \longleftarrow$ Distributiva

$\overline{(\overline{X} \cdot Y)} \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ De Morgan

$(X + \overline{Y}) \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ Distributiva

$(X + \overline{Y}) \cdot (Z + \overline{Z}) \longleftarrow$ Inverso

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}}$$

Solución:

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}} \longleftarrow$ De Morgan

$\overline{\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z}} \longleftarrow$ Distributiva

$\overline{(\overline{X} \cdot Y)} \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ De Morgan

$(X + \overline{Y}) \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ Distributiva

$(X + \overline{Y}) \cdot (Z + \overline{Z}) \longleftarrow$ Inverso

$(X + \overline{Y}) \cdot 1 \longleftarrow$ Identidad

Ejercicio 0

Demostrar si la siguiente igualdad entre funciones booleanas es verdadera o falsa:

$$(X + \overline{Y}) = \overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)}$$

Solución:

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{(Y + Z)} \longleftarrow$ De Morgan

$\overline{(\overline{X} \cdot Y)} \cdot Z + X \cdot \overline{Z} + \overline{Y} \cdot \overline{Z} \longleftarrow$ Distributiva

$\overline{(\overline{X} \cdot Y)} \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ De Morgan

$(X + \overline{Y}) \cdot Z + (X + \overline{Y}) \cdot \overline{Z} \longleftarrow$ Distributiva

$(X + \overline{Y}) \cdot (Z + \overline{Z}) \longleftarrow$ Inverso

$(X + \overline{Y}) \cdot 1 \longleftarrow$ Identidad

$X + \overline{Y}$ *Lo que queríamos demostrar*.

Notación

En el lenguaje coloquial vamos a llamar a las operaciones indistintamente de la siguiente forma:

$$A + B \equiv A \ OR \ B$$

Notación

En el lenguaje coloquial vamos a llamar a las operaciones indistintamente de la siguiente forma:

$$A + B \equiv A \ OR \ B$$

$$AB \equiv A.B \equiv A \ AND \ B$$

Notación

En el lenguaje coloquial vamos a llamar a las operaciones indistintamente de la siguiente forma:

$$A + B \equiv A \ OR \ B$$

$$AB \equiv A.B \equiv A \ AND \ B$$

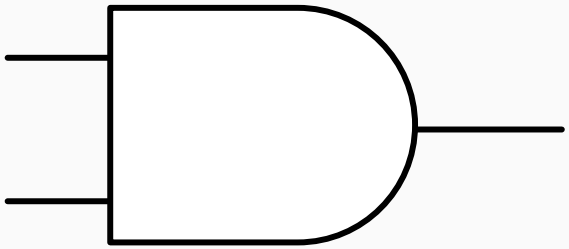
$$\overline{A} \equiv NOT \ A$$

Compuertas, señales y tablas de verdad

Compuertas

Son modelos idealizados de dispositivos electrónicos o de computo, que realizan operaciones booleanas.

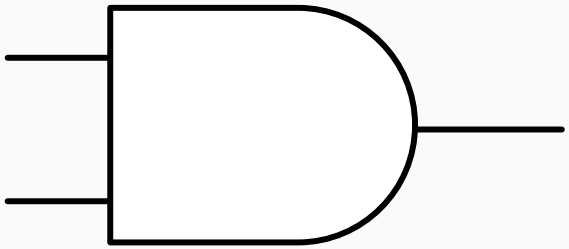
Las podemos representar gráficamente:



Compuertas

Son modelos idealizados de dispositivos electrónicos o de computo, que realizan operaciones booleanas.

Las podemos representar gráficamente:



O describir mediante un lenguaje de descripción de hardware (HDL), por ejemplo en `SystemVerilog`:

```
assign o = a & b;
```

Son representaciones que nos permiten observar todas las salidas para todas las combinaciones de entradas[1].

Por ejemplo, la función del ejercicio ($F = X + \overline{Y}$) se representa:

X	Y	F
0	0	
0	1	
1	0	
1	1	

[1] Como resulta esperable, esta representación puede volverse muy compleja cuando el número de variables y salidas crece.

Son representaciones que nos permiten observar todas las salidas para todas las combinaciones de entradas[1].

Por ejemplo, la función del ejercicio ($F = X + \overline{Y}$) se representa:

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

[1] Como resulta esperable, esta representación puede volverse muy compleja cuando el número de variables y salidas crece.

Son representaciones que nos permiten observar todas las salidas para todas las combinaciones de entradas[1].

Por ejemplo, la función del ejercicio ($F = X + \overline{Y}$) se representa:

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

[1] Como resulta esperable, esta representación puede volverse muy compleja cuando el número de variables y salidas crece.

Son representaciones que nos permiten observar todas las salidas para todas las combinaciones de entradas[1].

Por ejemplo, la función del ejercicio ($F = X + \overline{Y}$) se representa:

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

[1] Como resulta esperable, esta representación puede volverse muy compleja cuando el número de variables y salidas crece.

Tablas de verdad

Son representaciones que nos permiten observar todas las salidas para todas las combinaciones de entradas[1].

Por ejemplo, la función del ejercicio ($F = X + \overline{Y}$) se representa:

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

[1] Como resulta esperable, esta representación puede volverse muy compleja cuando el número de variables y salidas crece.

Compuertas - NOT

Gráficamente:

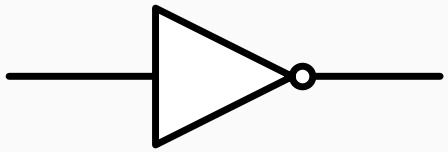


Tabla de verdad:

A	NOT A
0	1
1	0

En `SystemVerilog`:

```
assign o = ~a;
```

Compuertas - AND

Gráficamente:

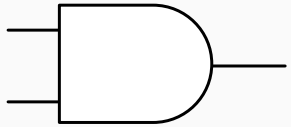


Tabla de verdad:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

En `SystemVerilog`:

```
assign o = a & b;
```

Compuertas - OR

Gráficamente:

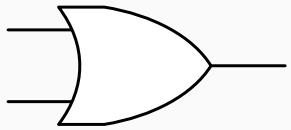


Tabla de verdad:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

En `SystemVerilog`:

```
assign o = a | b;
```

Compuertas - XOR u OR-EXCLUSIVA

Gráficamente:

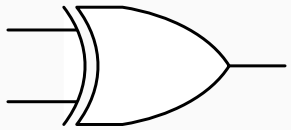


Tabla de verdad:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

En `SystemVerilog`:

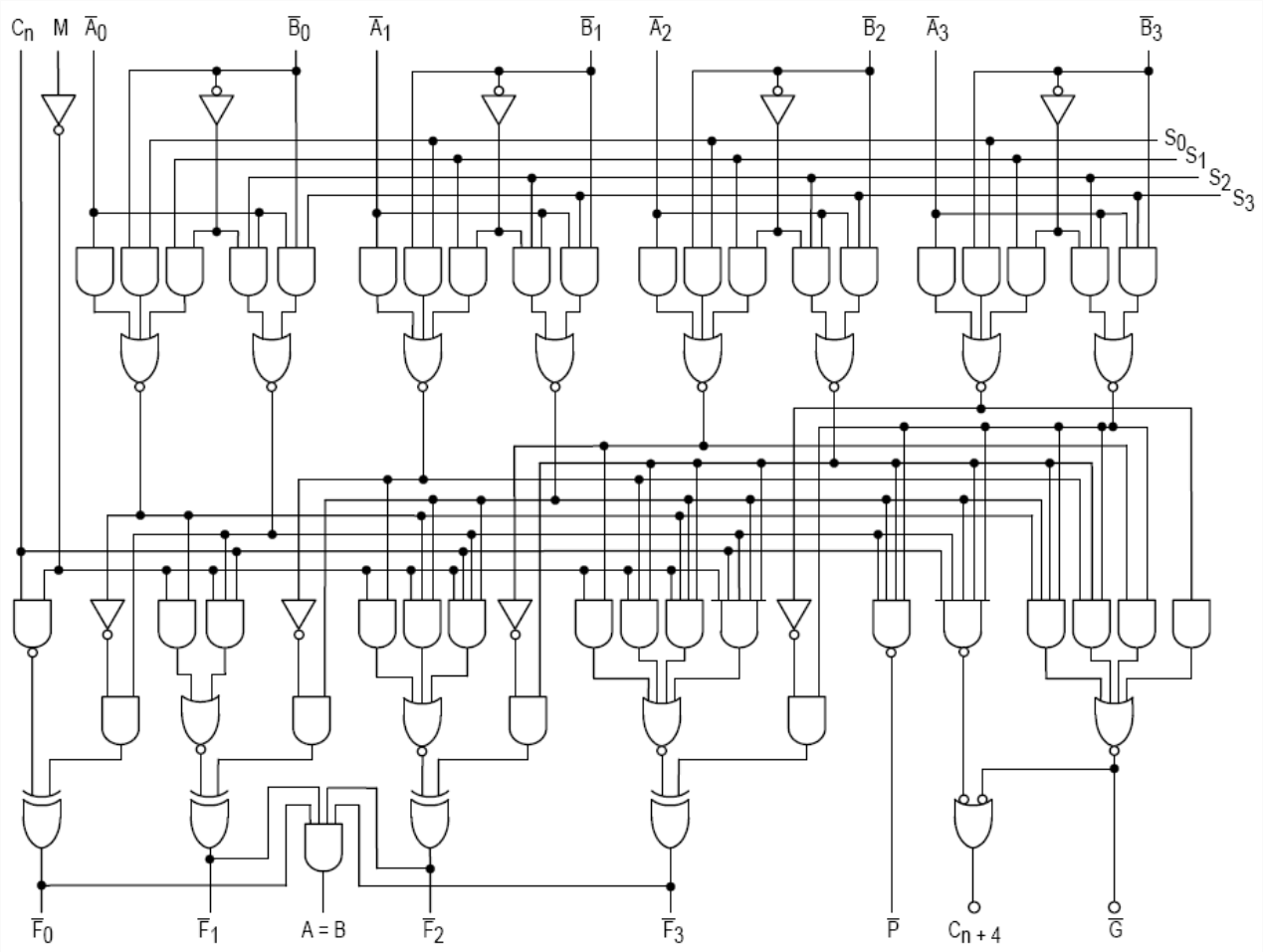
```
assign o = a ^ b;
```

Entradas y salidas - Categorización

Distintas vistas de un circuito

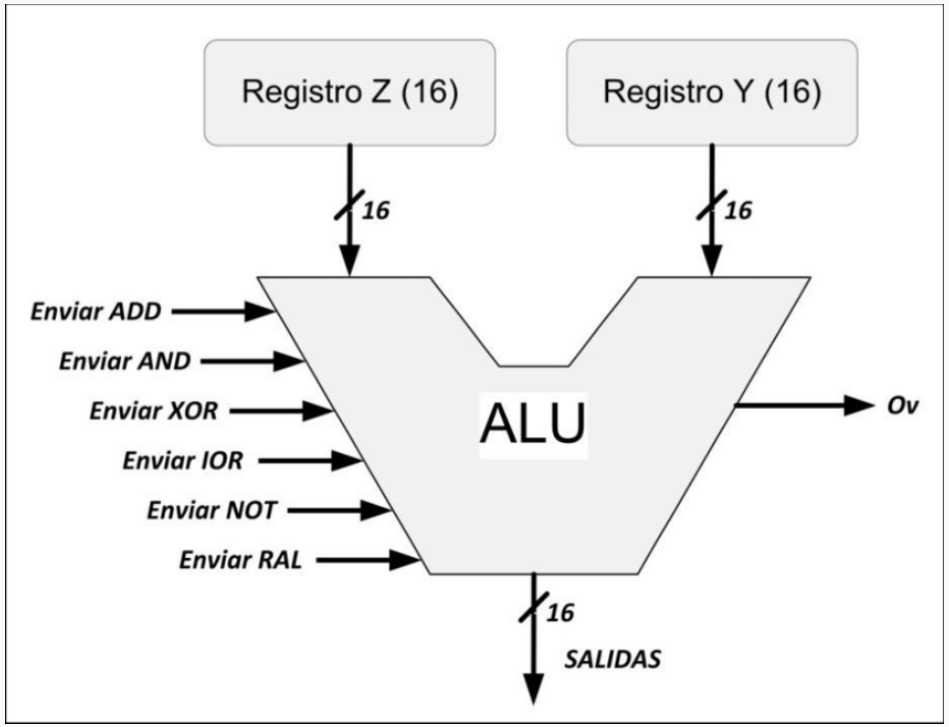
Por momentos vamos a querer abstraer nuestros circuitos en módulos de los cuales observaremos solamente sus entradas y salidas. Veamos un ejemplo donde ocultamos parte de la complejidad pasando de una vista interna del circuito (caja blanca) a una externa (caja negra).

Desde el museo: ALU 74181



Ejemplo con una ALU

Aplicando lo anterior, podemos trabajar con la ALU viéndola de la siguiente manera:



Establecen el sentido de la información:

En la ALU anterior se representan con las flechas...

Establecen el sentido de la información:

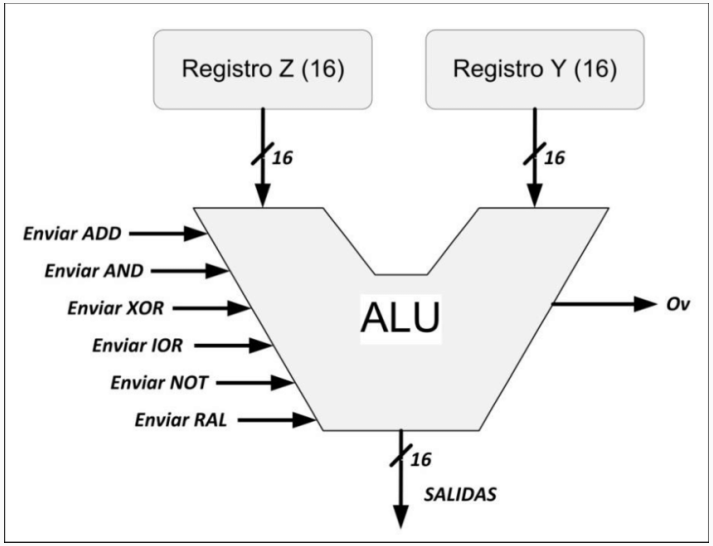
En la ALU anterior se representan con las flechas...

En `SystemVerilog`:

```
module ALU #(parameter DATA_WIDTH = 16)
            (input [DATA_WIDTH-1:0] operandoZ,
            input [DATA_WIDTH-1:0] operandoY,
            input [2:0] opcode,
            output [DATA_WIDTH-1:0] salidas,
            output overflow);
end module;
```

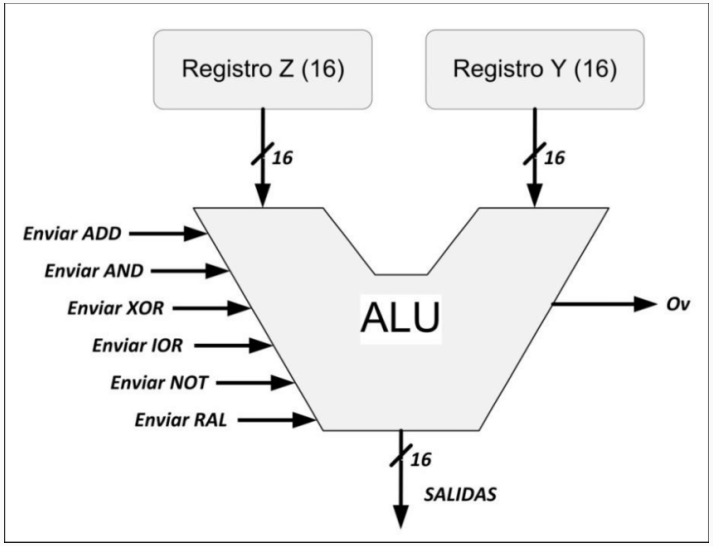
Entradas/Salidas: Tipos

En la ALU, ¿son funcionalmente todas iguales las entradas y salidas?



Entradas/Salidas: Tipos

En la ALU, ¿son funcionalmente todas iguales las entradas y salidas?



NO

Datos vs. Control

Lógica proposicional a circuitos combinatorios

Motivación

El estudio de la lógica proposicional y del álgebra de Boole tiene que ver con que vamos a querer implementar funciones lógicas en nuestro soporte electrónico con circuitos combinatorios.

Traduciendo una fórmula

Sabemos que se puede describir el comportamiento de un circuito combinatorio construyendo una tabla de verdad que determine las salidas que corresponden a cada combinación de los valores de entrada. Vamos a utilizar esto para describir un procedimiento que nos permite construir un circuito combinatorio cuyo comportamiento implementa cualquier fórmula proposicional φ.

Traduciendo una fórmula

Habrá casos en los que nos resultará difícil derivar un circuito de la fórmula, ya sea porque no vemos un vínculo directo entre la expresión y las compuertas básicas, o porque es conveniente expresarlo con una tabla de verdad.

Mecanismo de traducción

El mecanismo es el siguiente:

Mecanismo de traducción

El mecanismo es el siguiente:

- Si tenemos una fórmula φ que se expresa en función de las variables $x_1, \ldots, x_n$(las entradas).

Mecanismo de traducción

El mecanismo es el siguiente:

- Si tenemos una fórmula φ que se expresa en función de las variables $x_1, \ldots, x_n$(las entradas).
- Construimos una tabla de verdad con una fila para cada combinación posible de las entradas (por ej. $x_1 \to 1, x_2 \to 0, \ldots, x_n \to 1$) y en la columna de la salida y ingresamos el valor de la fórmula evaluada en esos valores $\varphi(1, 0, \ldots, 1)$.

Mecanismo de traducción

El mecanismo es el siguiente:

Mecanismo de traducción

El mecanismo es el siguiente:

- Vamos a utilizar solamente las filas en las que la función vale 1.

Mecanismo de traducción

El mecanismo es el siguiente:

- Vamos a utilizar solamente las filas en las que la función vale 1.
- Para cada fila i en la que φ es verdadera (vale 1) vamos a construir un término t_i como conjunción (y lógico o `AND`) de todas las entradas, donde cada variable aparece negada si su valor era 0 en la fila y sin negar en caso contrario.

Mecanismo de traducción

El mecanismo es el siguiente:

- Vamos a utilizar solamente las filas en las que la función vale 1.
- Para cada fila i en la que φ es verdadera (vale 1) vamos a construir un término t_i como conjunción (y lógico o `AND`) de todas las entradas, donde cada variable aparece negada si su valor era 0 en la fila y sin negar en caso contrario.
- Por ejemplo, si en la fila 4 la asignación (valuación) de las variables era $x_1 \to 1, x_2 \to 0, \ldots, x_n \to 1$, t_4 va a ser $x_1 \wedge \neg x_1 \wedge \ldots \wedge x_n$.

Mecanismo de traducción

El mecanismo es el siguiente:

Mecanismo de traducción

El mecanismo es el siguiente:

- Una vez que tenemos los términos $t_i, t_j, \ldots$ para cada fila en la que la función vale 1, vamos a hacer una disyunción (o lógico u `OR`) de todos los términos $\varphi' = t_i \vee t_j \vee \ldots$.

Mecanismo de traducción

El mecanismo es el siguiente:

- Una vez que tenemos los términos $t_i, t_j, \ldots$ para cada fila en la que la función vale 1, vamos a hacer una disyunción (o lógico u `OR`) de todos los términos $\varphi' = t_i \vee t_j \vee \ldots$.
- A este mecanismo se lo conoce como suma de productos y nos da una expresión de φ o de la tabla de verdad que puede traducirse fácilmente a un circuito combinatorio.

Volviendo al ejemplo

La fórmula $(F = X + \overline{Y})$ se representa:

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

Volviendo al ejemplo

La fórmula $(F = X + \overline{Y})$ se representa:

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

En este caso los términos serían $t_1 = \neg x \wedge \neg y, t_3 = x \wedge \neg y$ y $t_4 = x \wedge y$ y la expresión $\varphi' = (\neg x \wedge \neg y) \vee (x \wedge \neg y) \vee (x \wedge y)$.
A esta expresión se conoce como suma de productos.

Circuitos básicos

Ejercicio I - Sumador Simple

Armar un **sumador de 1 bit**. Tiene que tener dos entradas de un bit y dos salidas, una para el resultado y otra para indicar si hubo o no acarreo.

Ejercicio I - Sumador Simple

Armar un **sumador de 1 bit**. Tiene que tener dos entradas de un bit y dos salidas, una para el resultado y otra para indicar si hubo o no acarreo.

Solución:

Ejercicio I - Sumador Simple

Armar un **sumador de 1 bit**. Tiene que tener dos entradas de un bit y dos salidas, una para el resultado y otra para indicar si hubo o no acarreo.

Solución:

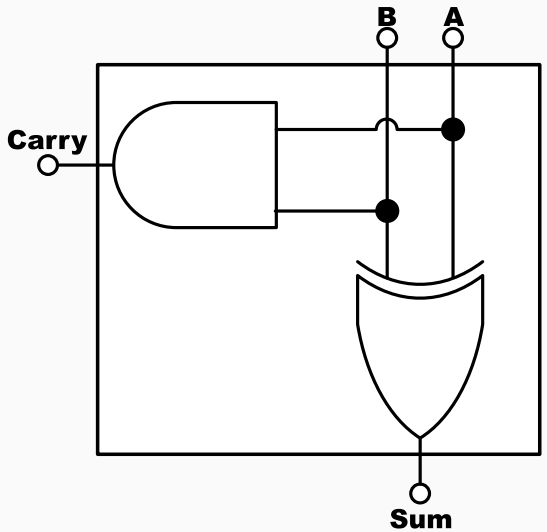
A	B	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Ejercicio I - Sumador Simple

Armar un **sumador de 1 bit**. Tiene que tener dos entradas de un bit y dos salidas, una para el resultado y otra para indicar si hubo o no acarreo.

Solución:

A	B	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Ejercicio II - Sumador Completo

Teniendo dos sumadores simples (de 1 bit) y sólo una compuerta a elección, arme un **sumador completo**. El mismo tiene 2 entradas de 1 bit y una tercer entrada interpretada como C_{In}, tiene como salida C_{Out} y S.

Ejercicio II - Sumador Completo

Teniendo dos sumadores simples (de 1 bit) y sólo una compuerta a elección, arme un **sumador completo**. El mismo tiene 2 entradas de 1 bit y una tercer entrada interpretada como C_{In}, tiene como salida C_{Out} y S. **Solución**:

Ejercicio II - Sumador Completo

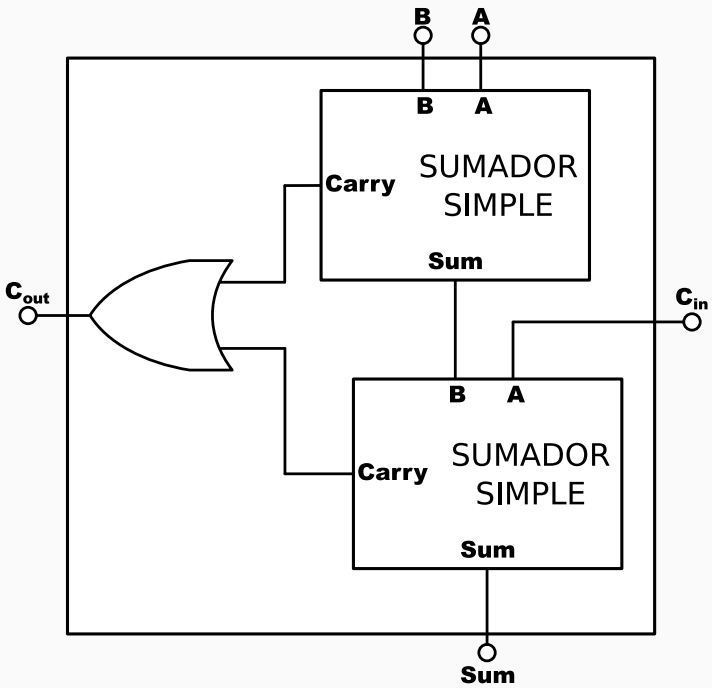
Teniendo dos sumadores simples (de 1 bit) y sólo una compuerta a elección, arme un **sumador completo**. El mismo tiene 2 entradas de 1 bit y una tercer entrada interpretada como C_{In}, tiene como salida C_{Out} y S. **Solución**:

C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ejercicio II - Sumador Completo

Teniendo dos sumadores simples (de 1 bit) y sólo una compuerta a elección, arme un **sumador completo**. El mismo tiene 2 entradas de 1 bit y una tercer entrada interpretada como C_{In}, tiene como salida C_{Out} y S. **Solución**:

C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Ejercicio III - Shift

Armar un circuito de 3 *bits*. Este deberá mover a izquierda o a derecha los bits de entrada de acuerdo al valor de una entrada extra que actúa como control. En otras palabras, un shift *izq-der* de *k-bits* es un circuito de $k+1$ entradas $(e_k, ..., e_0)$ y k salidas $(s_{k-1}, ..., s_0)$ que funciona del siguiente modo:

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Ejercicio III - Shift

Armar un circuito de 3 *bits*. Este deberá mover a izquierda o a derecha los bits de entrada de acuerdo al valor de una entrada extra que actúa como control. En otras palabras, un shift *izq-der* de *k-bits* es un circuito de $k+1$ entradas $(e_k, ..., e_0)$ y k salidas $(s_{k-1}, ..., s_0)$ que funciona del siguiente modo:

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k-1$ y $s_{k-1} = 0$

Ejemplos:

$$\text{shift\_lr}(1,011) = 110$$

Ejercicio III - Shift

Armar un circuito de 3 *bits*. Este deberá mover a izquierda o a derecha los bits de entrada de acuerdo al valor de una entrada extra que actúa como control. En otras palabras, un shift *izq-der* de *k-bits* es un circuito de $k+1$ entradas $(e_k, ..., e_0)$ y k salidas $(s_{k-1}, ..., s_0)$ que funciona del siguiente modo:

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k-1$ y $s_{k-1} = 0$

Ejemplos:

$$\text{shift\_lr}(1,011) = 110 \quad \text{shift\_lr}(0,011) = 001$$

Ejercicio III - Shift

Armar un circuito de 3 *bits*. Este deberá mover a izquierda o a derecha los bits de entrada de acuerdo al valor de una entrada extra que actúa como control. En otras palabras, un shift *izq-der* de *k-bits* es un circuito de $k+1$ entradas $(e_k, ..., e_0)$ y k salidas $(s_{k-1}, ..., s_0)$ que funciona del siguiente modo:

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Ejemplos:

$$\text{shift\_lr}(1,011) = 110 \quad \text{shift\_lr}(0,011) = 001$$
$$\text{shift\_lr}(1,100) = 000$$

Ejercicio III - Shift

Armar un circuito de 3 *bits*. Este deberá mover a izquierda o a derecha los bits de entrada de acuerdo al valor de una entrada extra que actúa como control. En otras palabras, un shift *izq-der* de *k-bits* es un circuito de $k+1$ entradas $(e_k, ..., e_0)$ y k salidas $(s_{k-1}, ..., s_0)$ que funciona del siguiente modo:

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Ejemplos:

$$\text{shift\_lr}(1{,}011) = 110 \quad \text{shift\_lr}(0{,}011) = 001$$
$$\text{shift\_lr}(1{,}100) = 000 \quad \text{shift\_lr}(1{,}101) = 010$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix}$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix}$$

$$e_3 . e_1$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix} \qquad s_0 = \begin{bmatrix} 0 & \text{si } e_3 = 1 \\ e_1 & \text{si } e_3 = 0 \end{bmatrix}$$

$$e_3 . e_1$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix} \qquad s_0 = \begin{bmatrix} 0 & \text{si } e_3 = 1 \\ e_1 & \text{si } e_3 = 0 \end{bmatrix}$$

$$e_3 . e_1 \qquad \overline{e_3} . e_1$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix} \quad s_0 = \begin{bmatrix} 0 & \text{si } e_3 = 1 \\ e_1 & \text{si } e_3 = 0 \end{bmatrix} \quad s_1 = \begin{bmatrix} e_0 & \text{si } e_3 = 1 \\ e_2 & \text{si } e_3 = 0 \end{bmatrix}$$

$$e_3 . e_1 \qquad \overline{e_3} . e_1$$

Ejercicio III - Shift

- Si $e_k = 1$, entonces $s_i = e_{i-1}$ para todo $0 < i < k$ y $s_0 = 0$
- Si $e_k = 0$, entonces $s_i = e_{i+1}$ para todo $0 \leq i < k - 1$ y $s_{k-1} = 0$

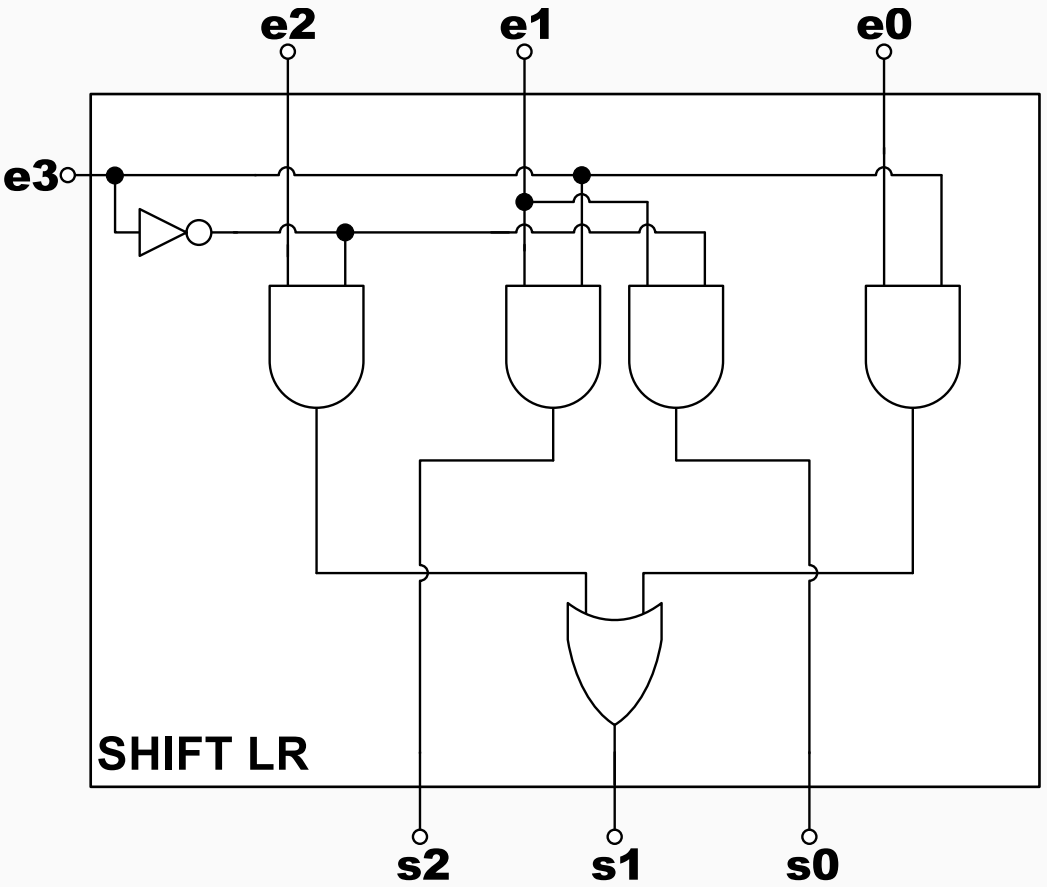
Solución:

$$s_2 = \begin{bmatrix} 0 & \text{si } e_3 = 0 \\ e_1 & \text{si } e_3 = 1 \end{bmatrix} \qquad s_0 = \begin{bmatrix} 0 & \text{si } e_3 = 1 \\ e_1 & \text{si } e_3 = 0 \end{bmatrix} \qquad s_1 = \begin{bmatrix} e_0 & \text{si } e_3 = 1 \\ e_2 & \text{si } e_3 = 0 \end{bmatrix}$$

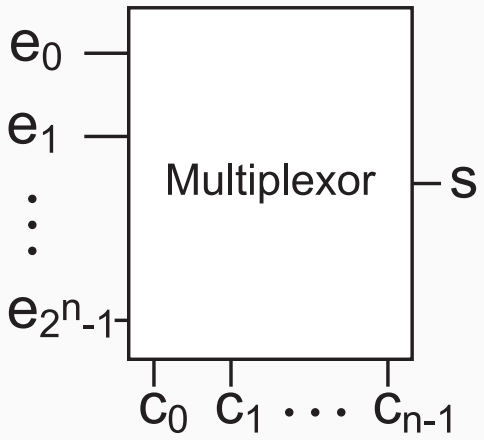
$$e_3.e_1 \qquad \overline{e_3}.e_1 \qquad e_3.e_0 + \overline{e_3}.e_2$$

Ejercicio III - Shift

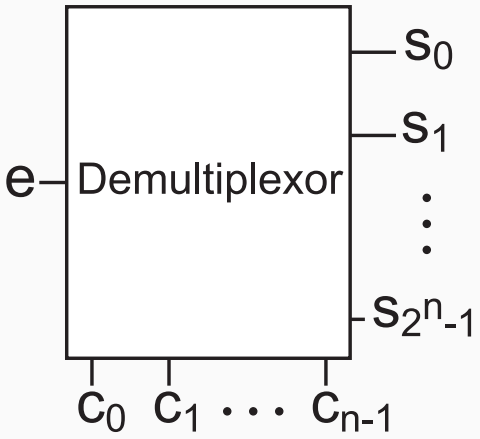
Solución:



Más combinatorios: Multiplexor y Demultiplexor



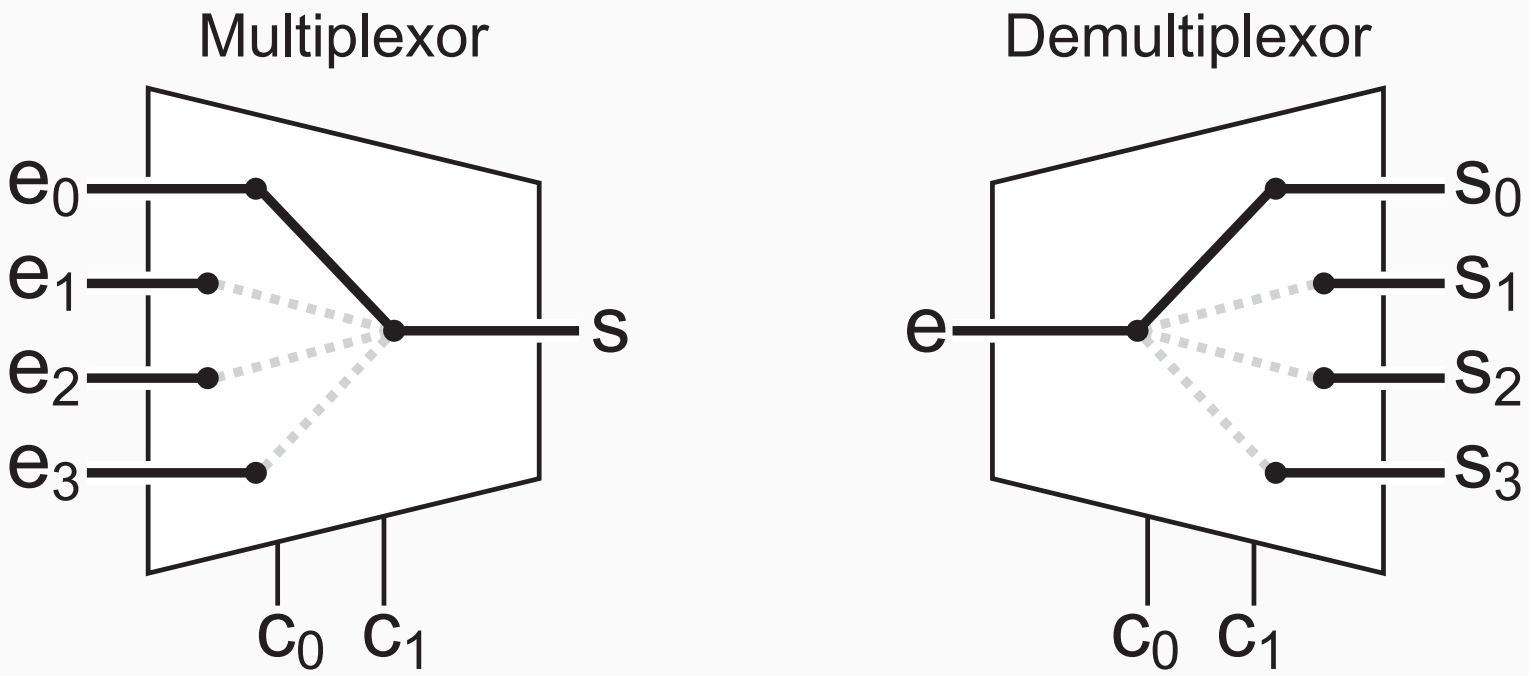
Las líneas de control c permiten seleccionar una de las entradas e, la que corresponderá a la salida s.



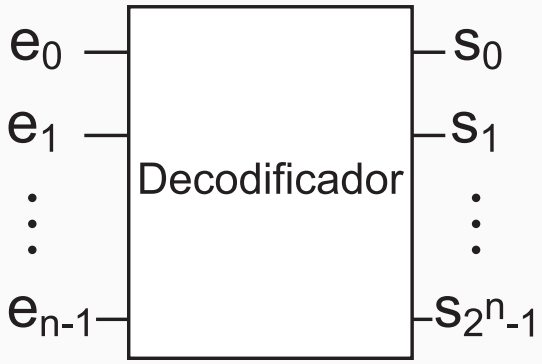
Las líneas de control c permiten seleccionar cual de las salidas s tendrá el valor de e.

Multiplexor y Demultiplexor

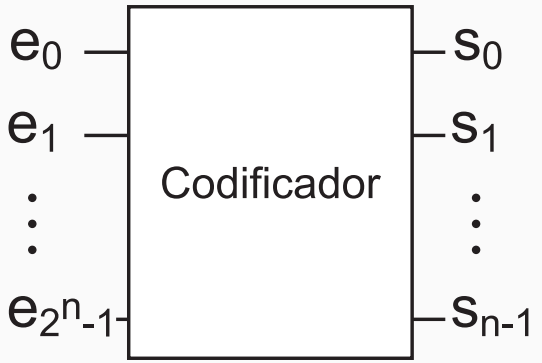
- Ejemplo,



Más combinatorios: Codificador y Decodificador



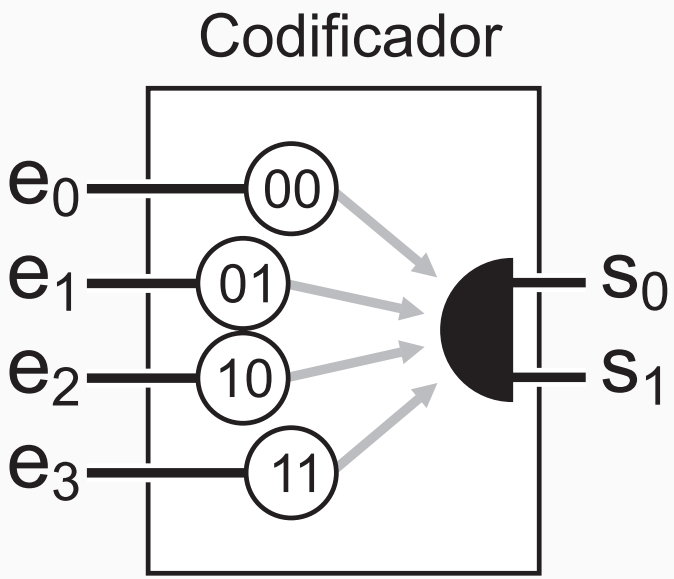
Cada combinación de las líneas e corresponderá a una sola línea en alto de la salida s.

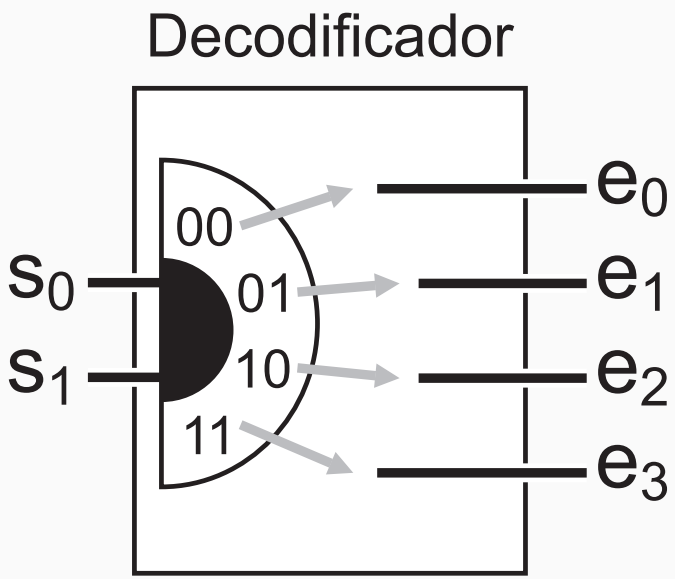


Una y sólo una línea en alto de e corresponderá a una combinación en la salida s.

Codificador y Decodificador

- Ejemplo,

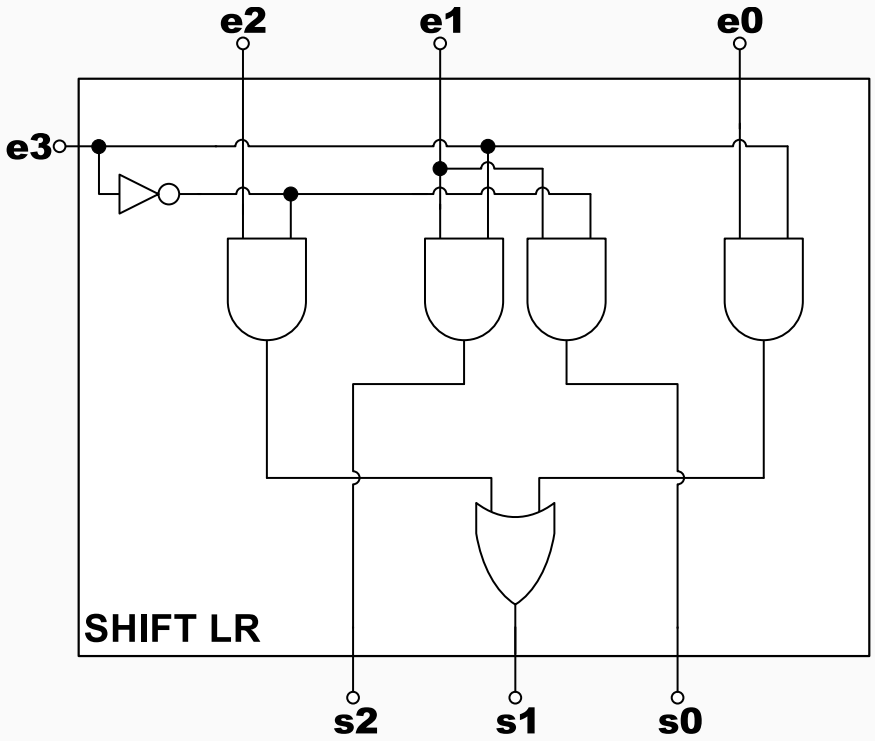




Timing

¡Las compuertas no son instantáneas!

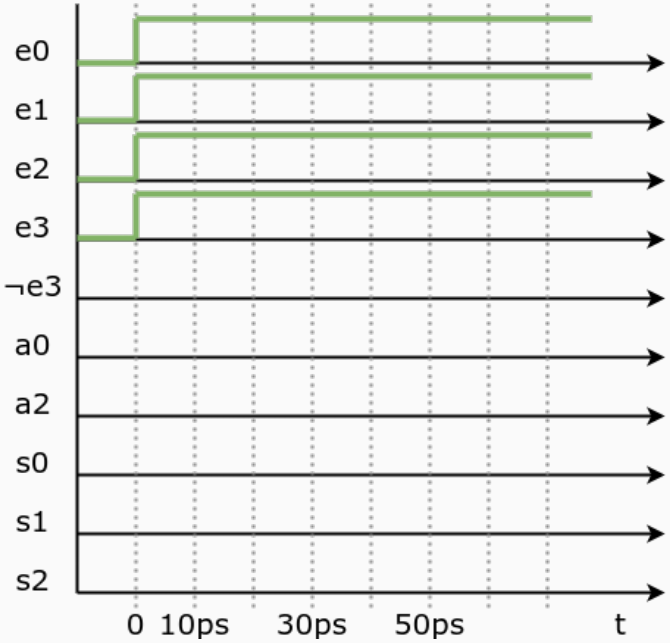
Revisitemos nuestro Shift LR:

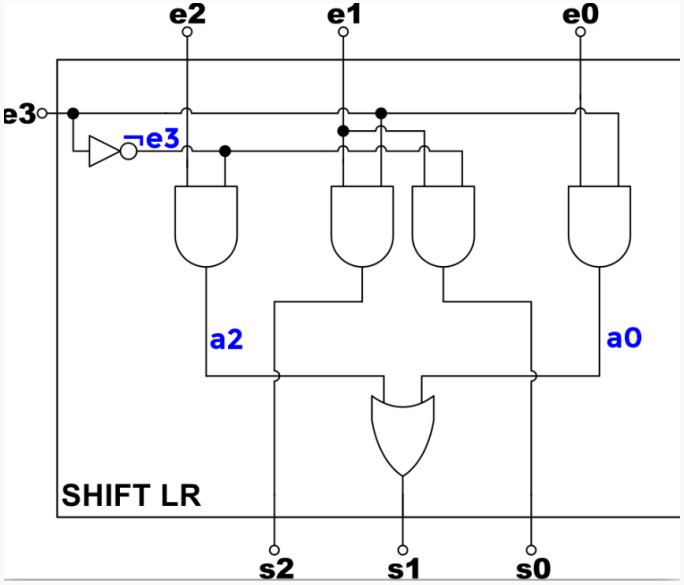


Timing

Para el circuito Shift LR anterior, supongamos (de forma optimista) que todas las compuertas tardan 10ps en poner un resultado válido en sus salidas. A partir de ello, dibujemos el diagrama de tiempos para cuando todas las entradas cambian simultáneamente de '0' a '1'.

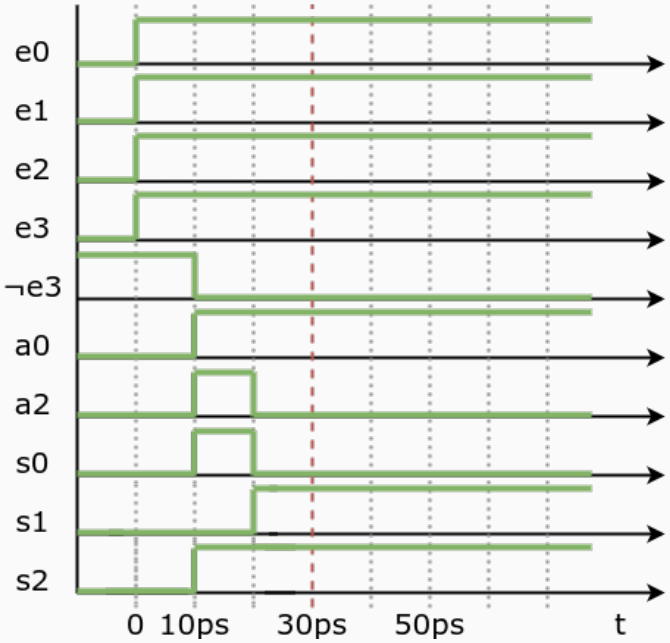
Hagamos un diagrama de tiempos[2]:

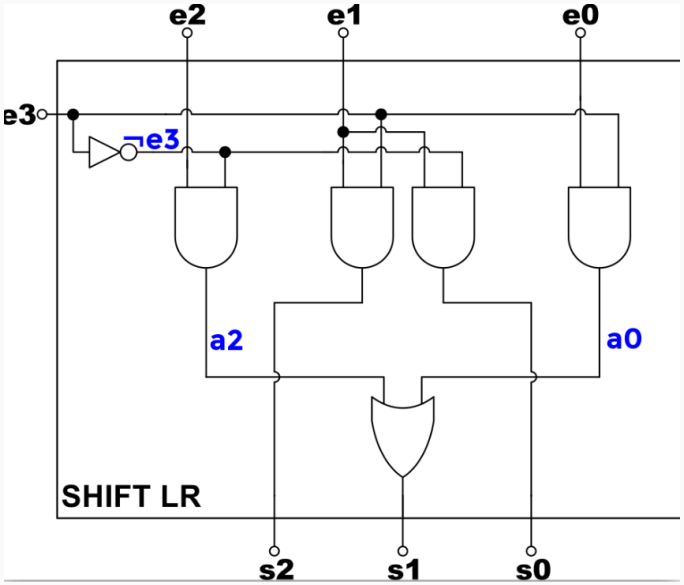




[2]Y nombremos a las señales que no tienen nombre

Diagrama de tiempos





Timing

¿Cuál es el mínimo tiempo que se debe esperar para leer un resultado válido de su salida?

Timing

¿Cuál es el mínimo tiempo que se debe esperar para leer un resultado válido de su salida?

- En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de *capas* de compuertas (*latencia*)

Timing

¿Cuál es el mínimo tiempo que se debe esperar para leer un resultado válido de su salida?

- En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de *capas* de compuertas (*latencia*)
- En este caso debemos esperar al menos $3 \cdot 10ps = 30ps$ para poder leer la salida.

Timing

¿Cuál es el mínimo tiempo que se debe esperar para leer un resultado válido de su salida?

- En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de *capas* de compuertas (*latencia*)
- En este caso debemos esperar al menos $3 \cdot 10ps = 30ps$ para poder leer la salida.

¿Cómo enfrentamos este problema?

Secuenciales...

Intervalo

Lógica Digital - Circuitos Secuenciales

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Sobre la clase de hoy

Hoy vamos a ver los principios de diseño, práctica y ejemplos de circuitos secuenciales, la estructura de la clase va ser la siguiente:

- **Repaso de circuitos combinatorios**
- **Retroalimentación y cambio de modelo**
- **Circuitos secuenciales asincrónicos**
- **Circuitos secuenciales sincrónicos**
- **Latchs - Flip-flops, registros y memorias**

Sobre la clase de hoy

Hoy vamos a ver los principios de diseño, práctica y ejemplos de circuitos secuenciales, la estructura de la clase va ser la siguiente:

- **Repaso de circuitos combinatorios**
- **Retroalimentación y cambio de modelo**
- **Circuitos secuenciales asincrónicos**
- **Circuitos secuenciales sincrónicos**
- **Latchs - Flip-flops, registros y memorias**

Latchs - Flip-flops

Latchs

Son circuitos que permiten *trabar o asegurar* el valor de su salida

Latchs

Son circuitos que permiten *trabar o asegurar* el valor de su salida

- Permiten el cambio de sus salidas según el **nivel** de las entradas.

Latchs

Son circuitos que permiten *trabar o asegurar* el valor de su salida

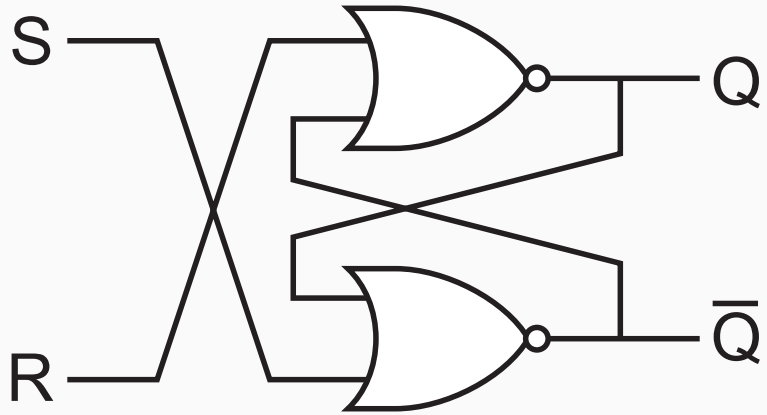
- Permiten el cambio de sus salidas según el **nivel** de las entradas.
- Utilizan **realimentación**

Latchs

Son circuitos que permiten *trabar o asegurar* el valor de su salida

- Permiten el cambio de sus salidas según el **nivel** de las entradas.
- Utilizan **realimentación**

Ejemplo:



Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

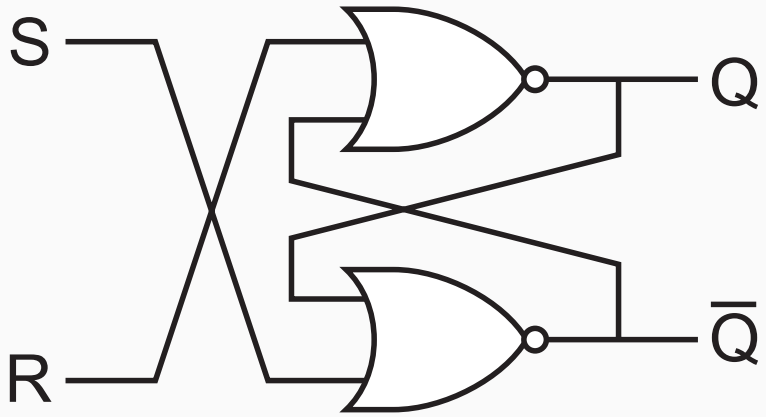


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0		
0	1		
0	0		
1	1		

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

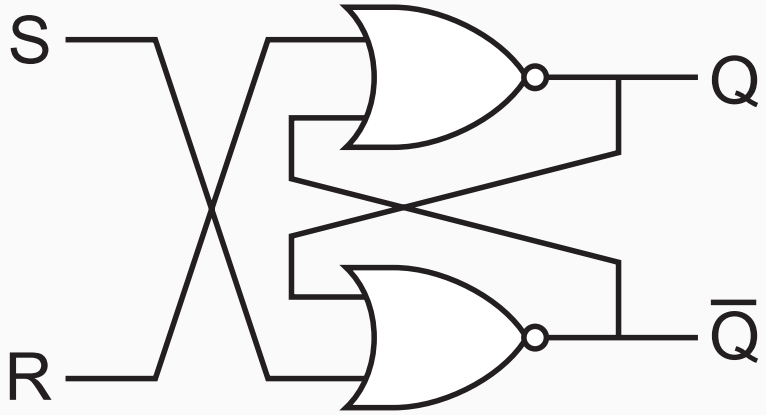


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0		0
0	1		
0	0		
1	1		

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

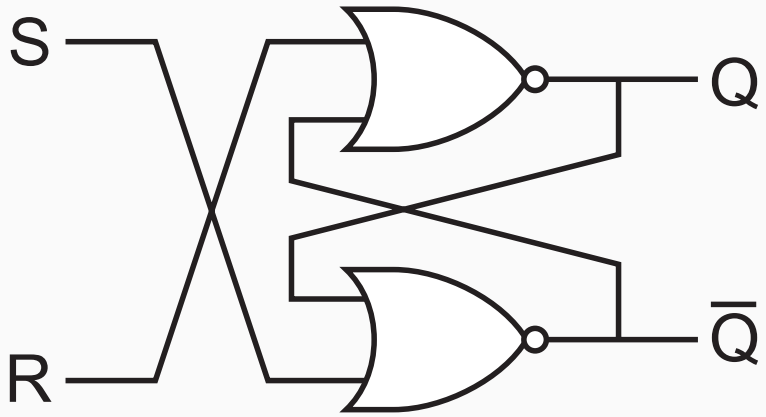


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1		
0	0		
1	1		

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

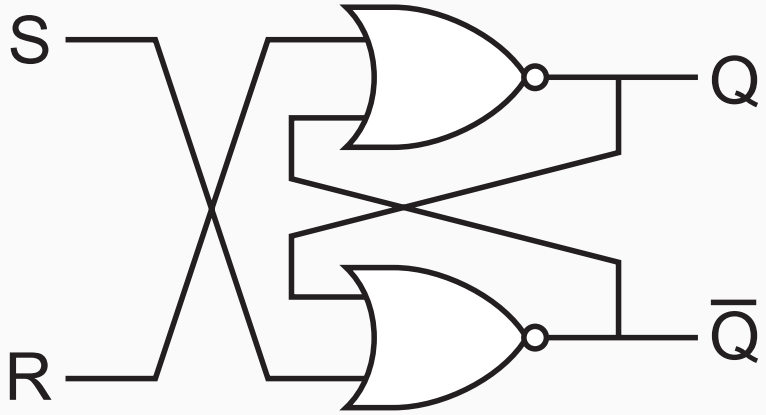


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0		
1	1		

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

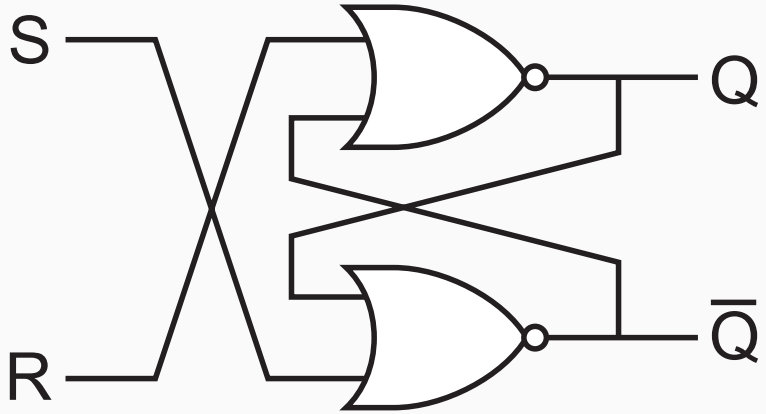


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
1	1		

[1] $Q*$ o $\overline{Q}*$ refiere al *estado* anterior de la salida

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

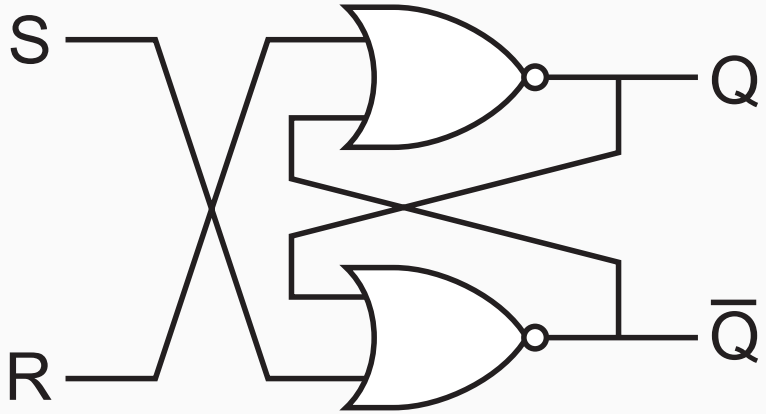


Tabla de verdad:

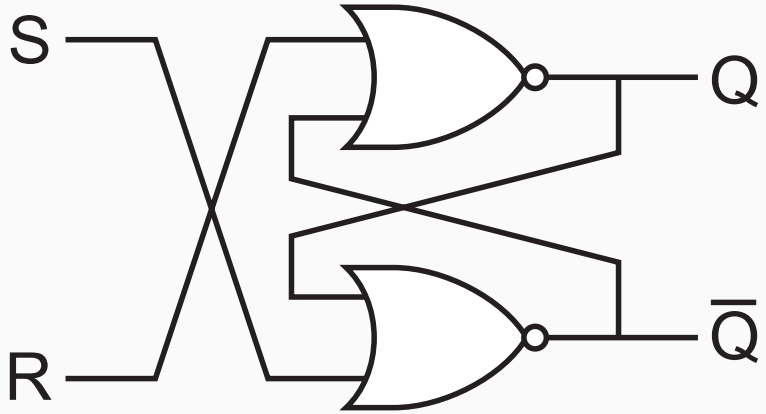
S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
1	1	0	0

[1] $Q*$ o $\overline{Q}*$ refiere al *estado* anterior de la salida

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:



Con $S, R = (1, 1)$:

Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
1	1	0	0

[1] $Q*$ o $\overline{Q}*$ refiere al *estado* anterior de la salida

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

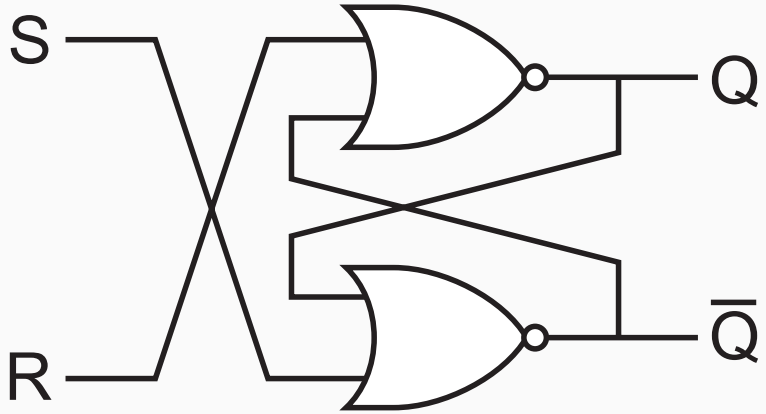


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
1	1	0	0

Con $S, R = (1, 1)$:

- El valor de las salidas es inconsistente con la especificación

[1] $Q*$ o $\overline{Q}*$ refiere al *estado* anterior de la salida

Latch RS (Reset-Set)

Analicemos el ejemplo anterior:

Latch RS implementado con NOR:

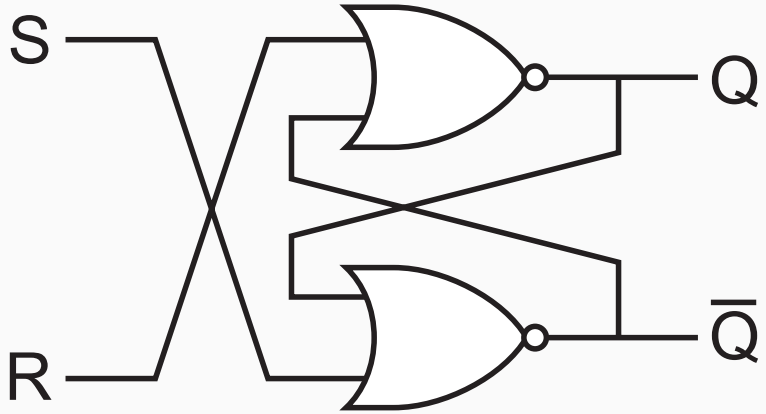


Tabla de verdad:

S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
1	1	0	0

Con $S, R = (1, 1)$:

- El valor de las salidas es inconsistente con la especificación
- El valor de las salidas depende de la implementación. **Tarea:** implementar con NANDs

[1] $Q*$ o $\overline{Q}*$ refiere al *estado* anterior de la salida

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

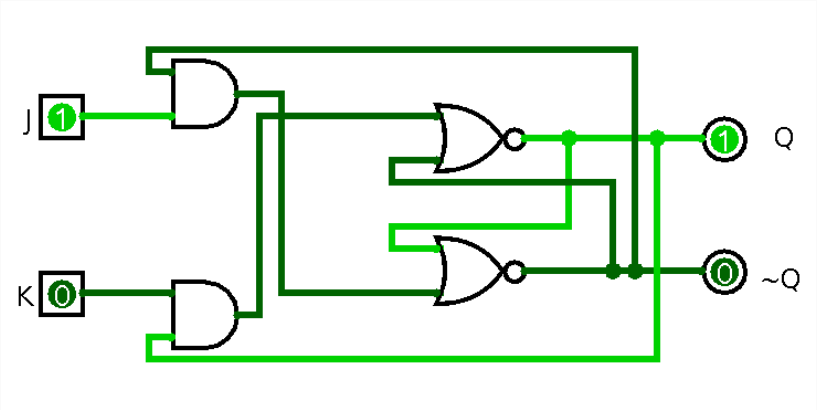


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0		
0	1		
0	0		
1	1		

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

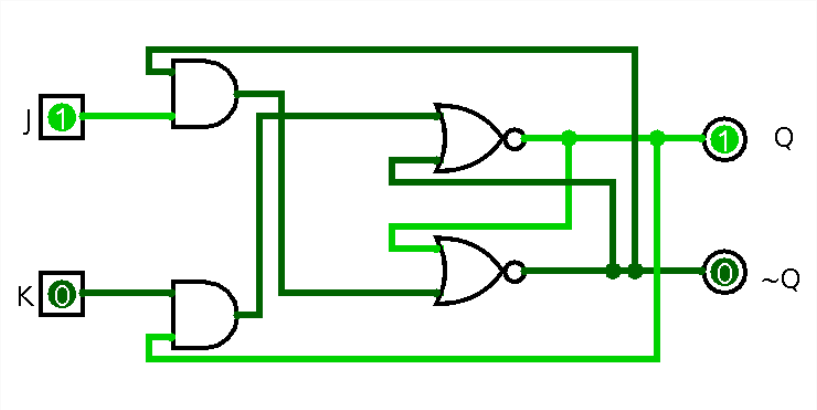


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0		0
0	1		
0	0		
1	1		

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

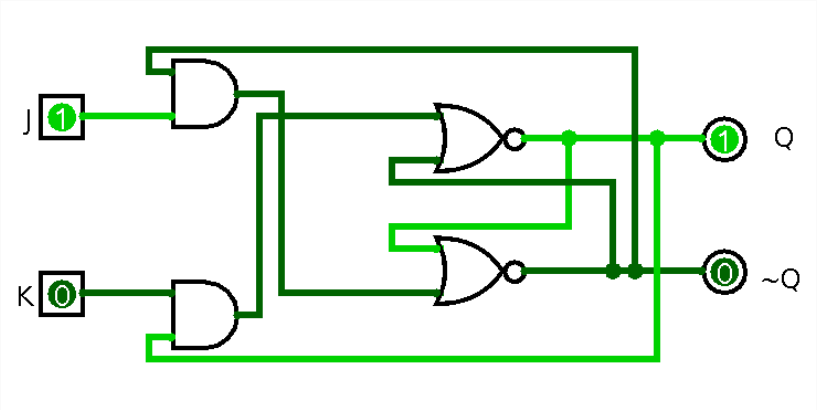


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1		
0	0		
1	1		

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

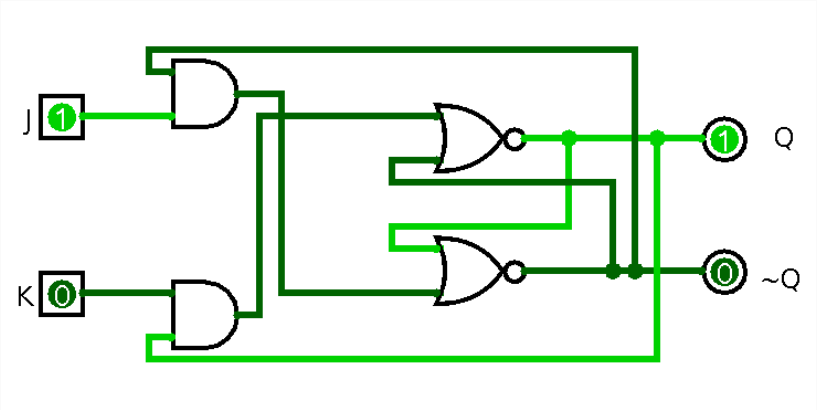


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0		
1	1		

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

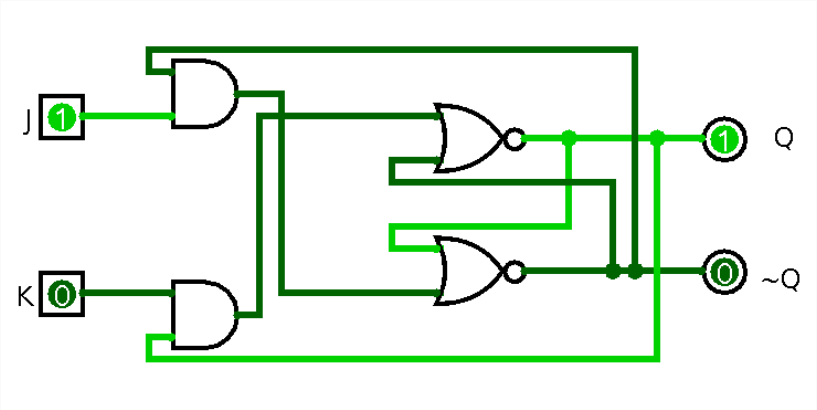


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1		

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

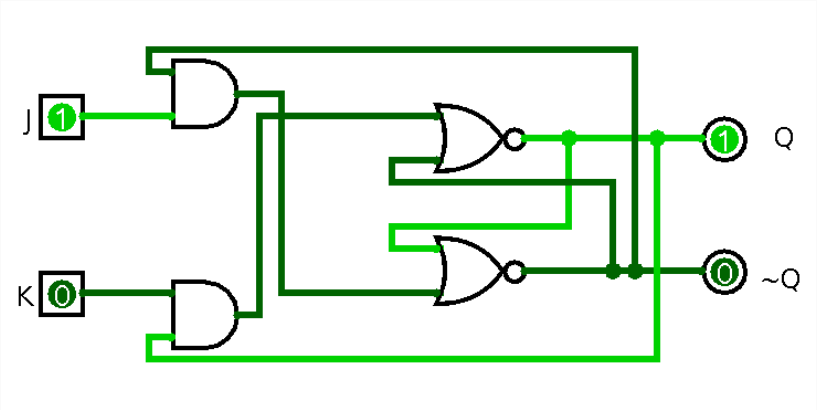


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	$\overline{Q}*$	$Q*$

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

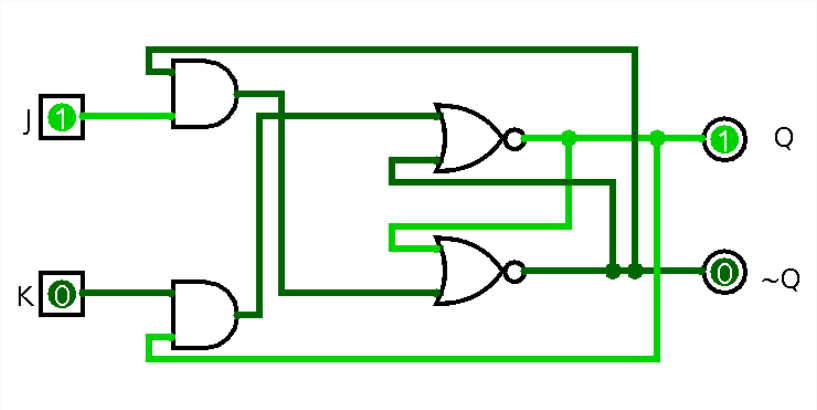


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	$\overline{Q}*$	$Q*$

Con $S, R = (1, 1)$:

- El valor de las salidas está ahora definido.

Latch JK

Tratemos de modificar el comportamiento para el caso cuando las entradas son $(1, 1)$:

Latch JK:

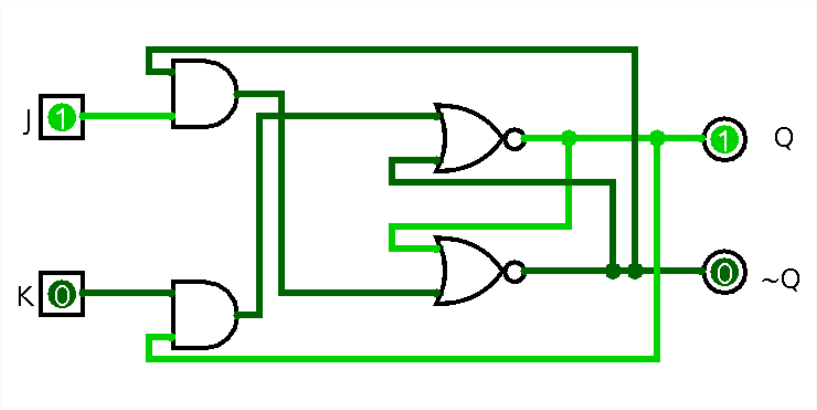


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	$\overline{Q}*$	$Q*$

Con $S, R = (1, 1)$:

- El valor de las salidas está ahora definido.
- El circuito oscila (estado inestable).

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

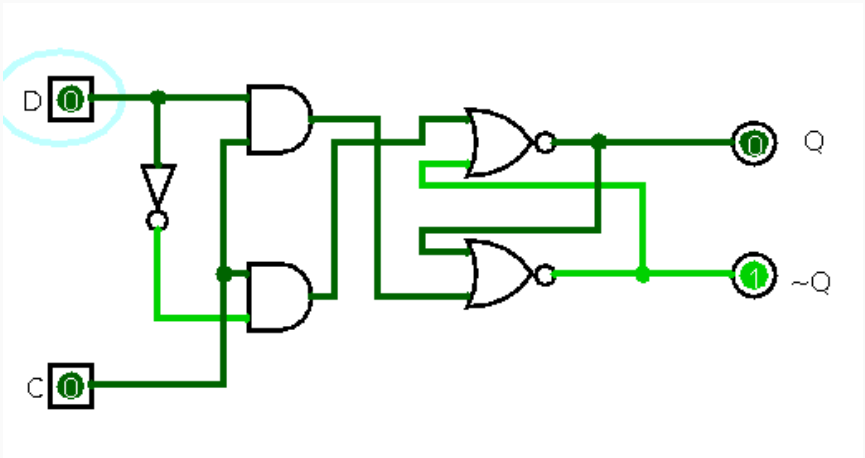


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0		
0	1		
0	0		
1	1		

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

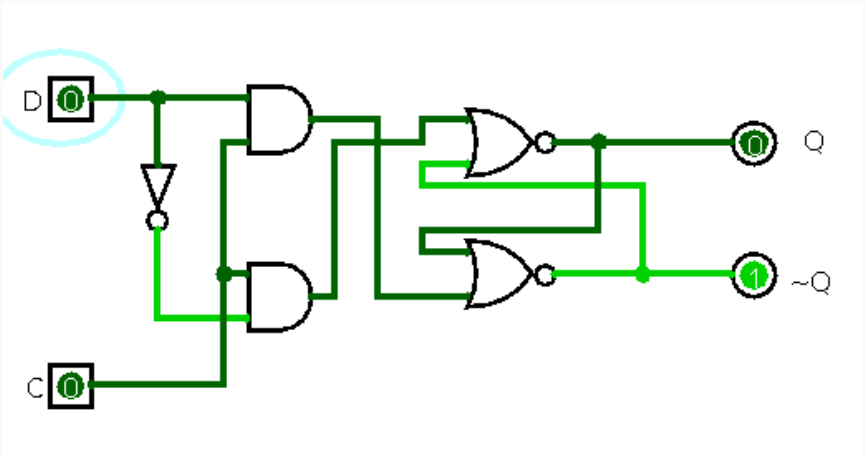


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0		$\overline{Q}*$
0	1		
0	0		
1	1		

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

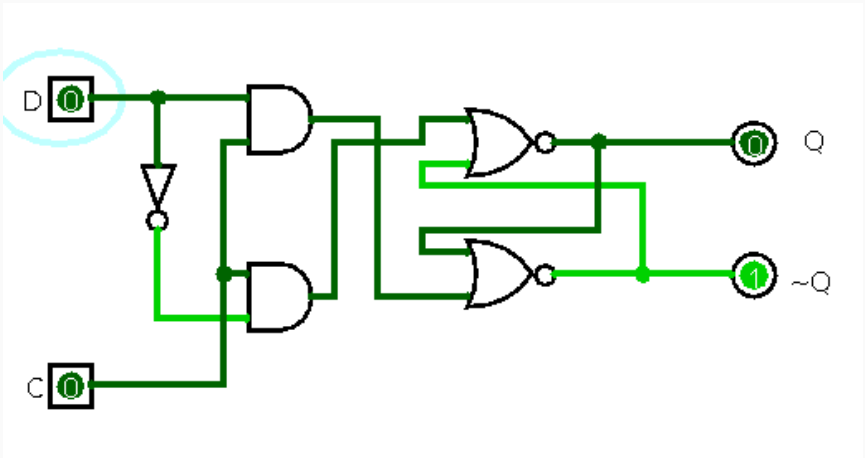


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1		
0	0		
1	1		

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

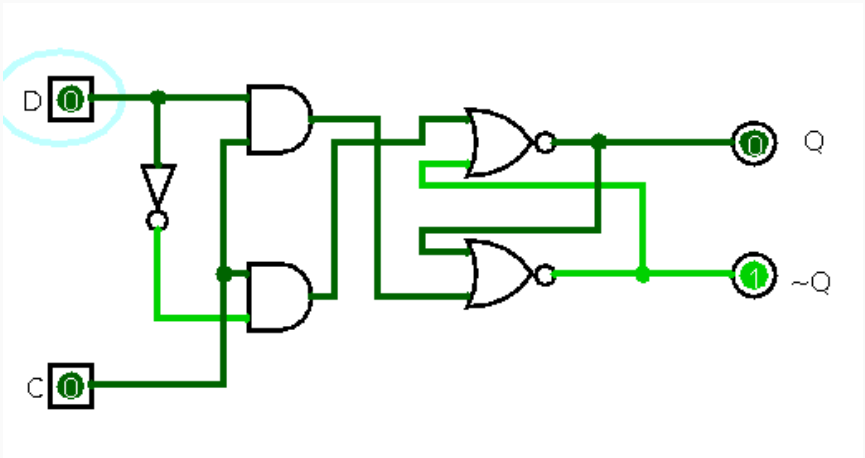


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0		
1	1		

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

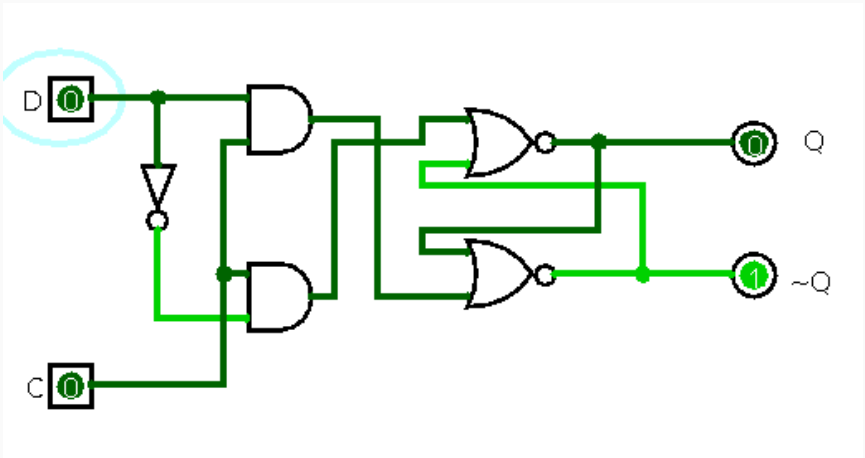


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1		

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

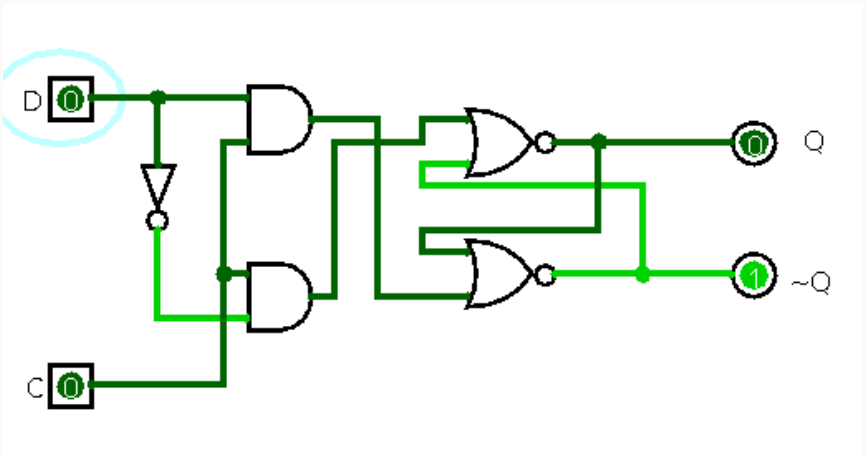


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

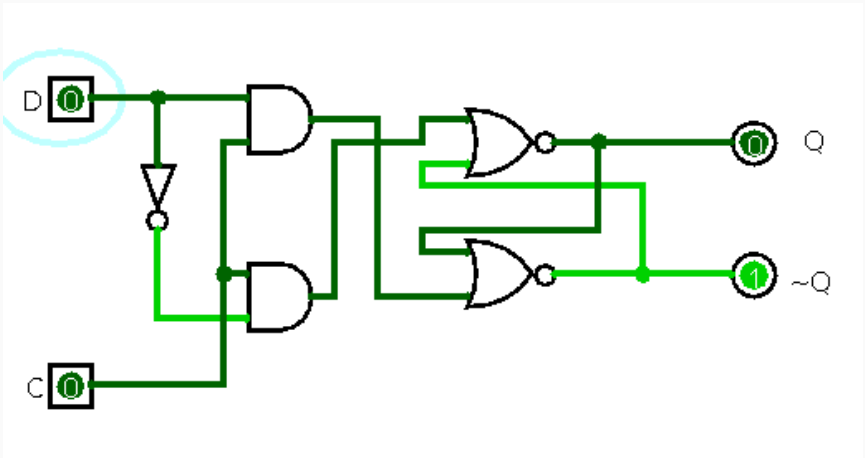


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

En este caso el circuito es estable en todos los estados. Sin embargo:

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

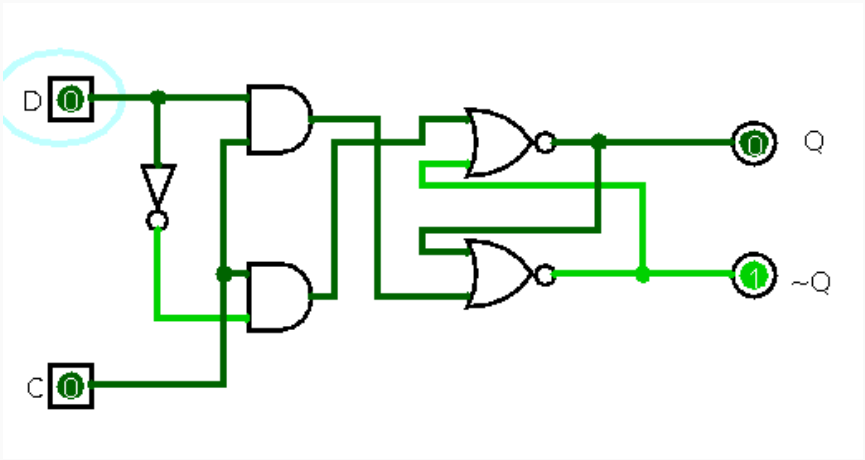


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

En este caso el circuito es estable en todos los estados. Sin embargo:

- Los tiempos no se pueden predecir (dependen de D)

Latch D

- Nos permite almacenar 1 bit
- Tiene una entrada de **datos** y una de **control**

Latch D:

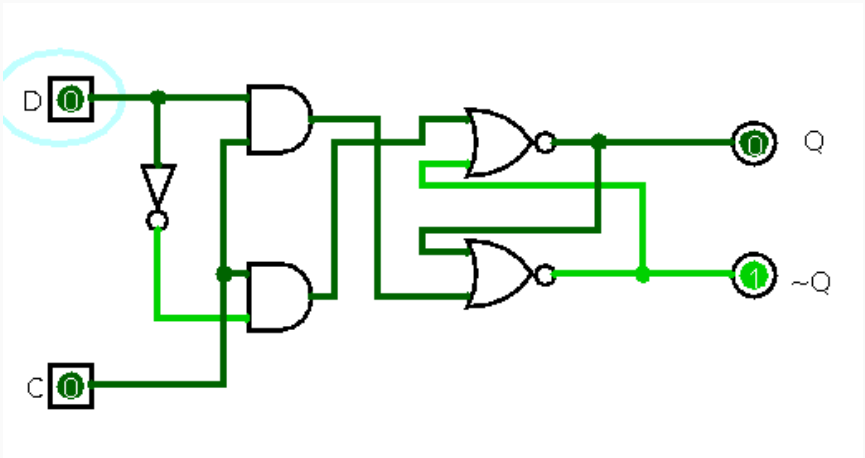


Tabla de verdad:

D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

En este caso el circuito es estable en todos los estados. Sin embargo:

- Los tiempos no se pueden predecir (dependen de D)
- Puede causar carreras si existe un lazo en el circuito externo.

Sincronizando...

Como vimos en la primer parte, nos interesa poder tener un control de los momentos de transición de estados $\Rightarrow$ **CLOCK**

Sincronizando...

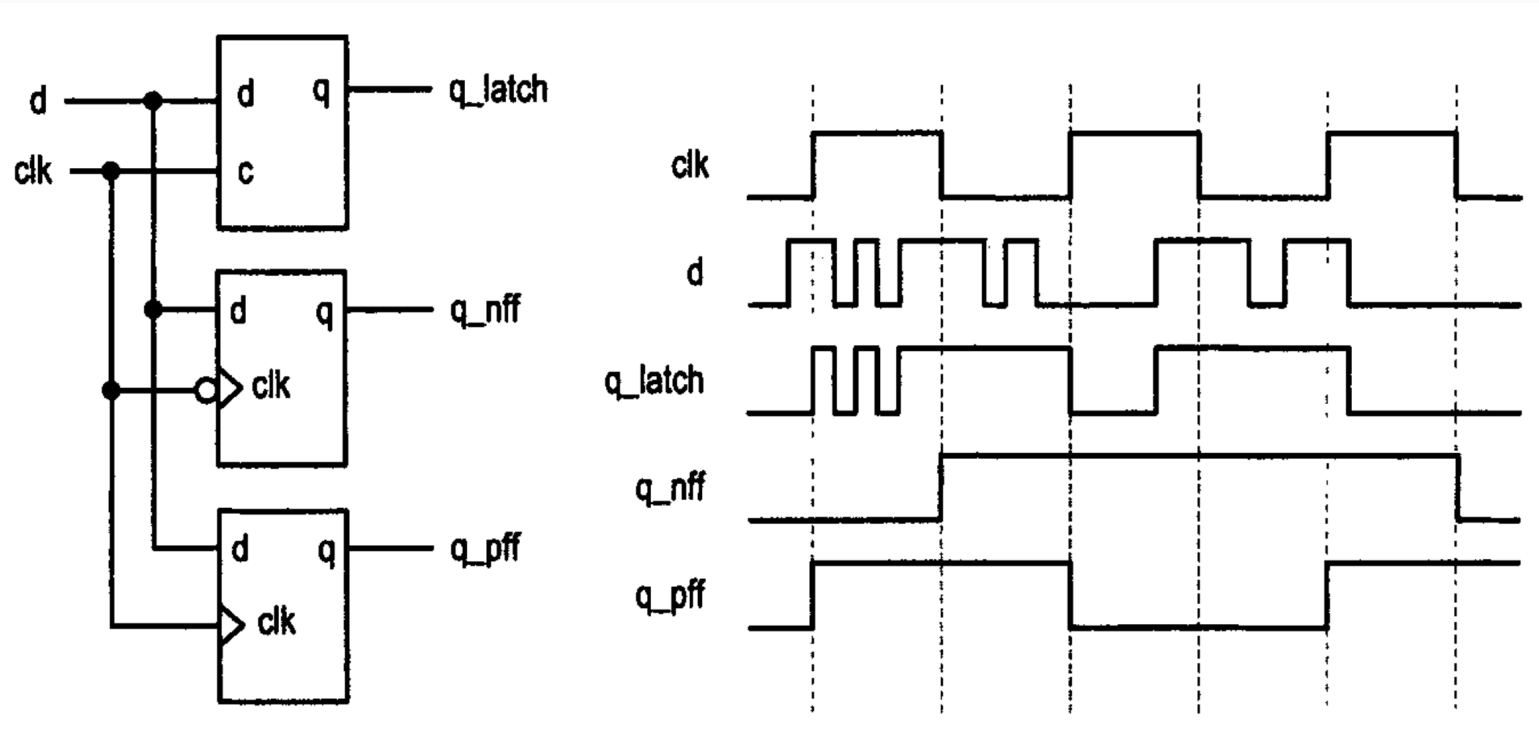
Como vimos en la primer parte, nos interesa poder tener un control de los momentos de transición de estados $\Rightarrow$ **CLOCK**

Vimos también que ser reactivo al nivel de una señal no es conveniente $\Rightarrow$ **Sensibilidad al flanco**

Sincronizando...

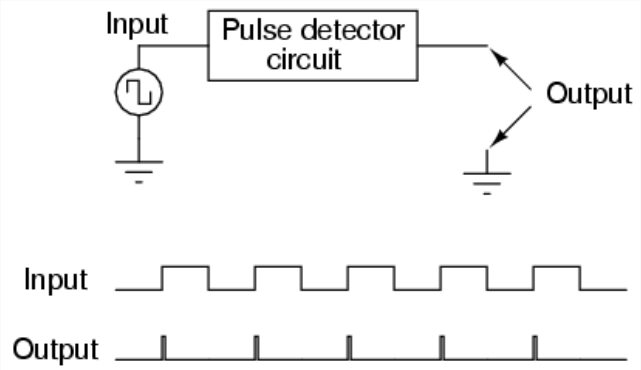
Como vimos en la primer parte, nos interesa poder tener un control de los momentos de transición de estados ⇒ **CLOCK**

Vimos también que ser reactivo al nivel de una señal no es conveniente ⇒ **Sensibilidad al flanco**



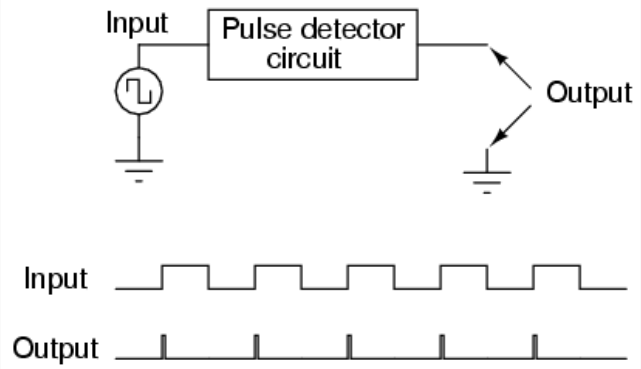
Detector de flanco

Necesitamos un circuito que se comporte de la siguiente manera:

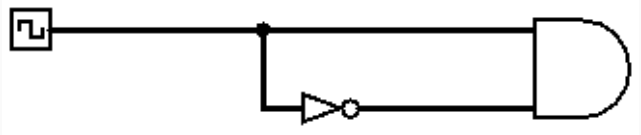


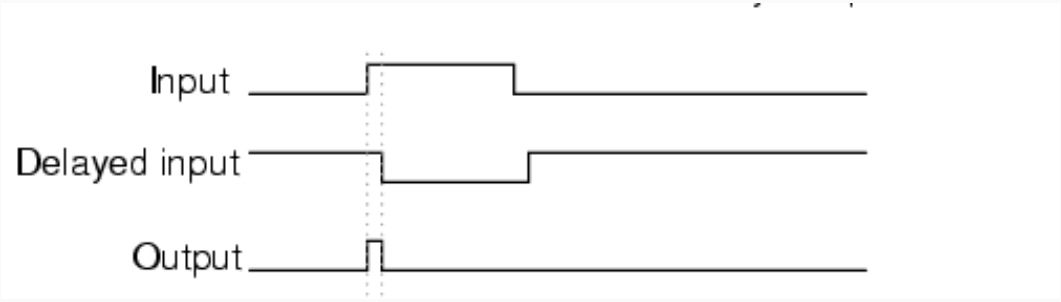
Detector de flanco

Necesitamos un circuito que se comporte de la siguiente manera:



Entonces, aprovechando los tiempos de propagación:





Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

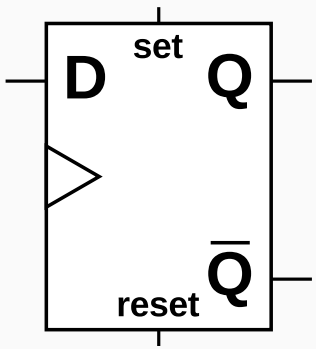


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0		
0	$1\uparrow$		
0	0		
1	$1\uparrow$		

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

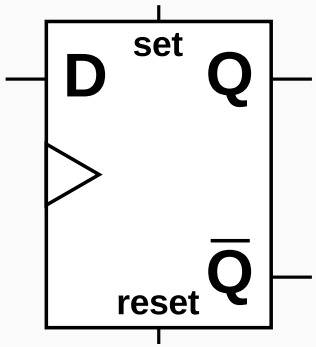


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0		$\overline{Q_T}$
0	$1\uparrow$		
0	0		
1	$1\uparrow$		

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

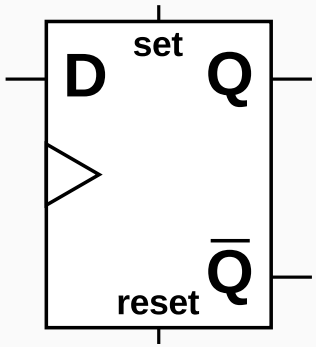


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0	Q_T	$\overline{Q_T}$
0	$1\uparrow$		
0	0		
1	$1\uparrow$		

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

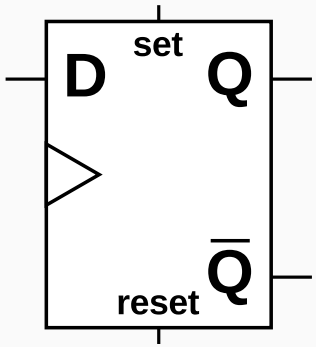


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0	Q_T	$\overline{Q_T}$
0	$1\uparrow$	0	1
0	0		
1	$1\uparrow$		

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

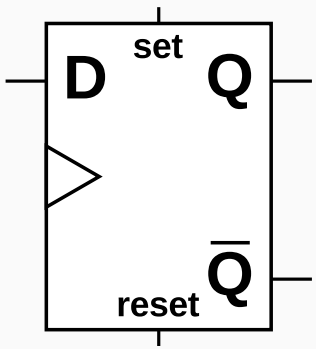


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0	Q_T	$\overline{Q_T}$
0	$1\uparrow$	0	1
0	0	Q_T	$\overline{Q_T}$
1	$1\uparrow$		

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

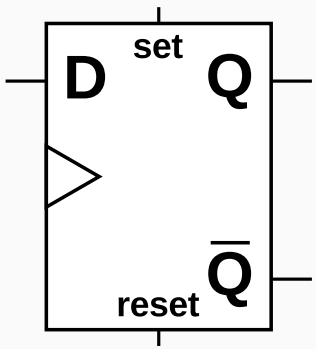


Tabla de verdad:

D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0	Q_T	$\overline{Q_T}$
0	$1\uparrow$	0	1
0	0	Q_T	$\overline{Q_T}$
1	$1\uparrow$	1	0

Flip-Flop D (Delay)

Ahora nuestro latch es sólo sensible a los flancos ascendentes de clock, entonces:

Lo podemos representar:

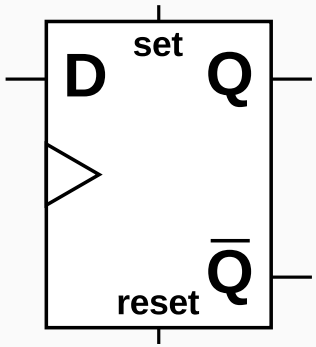


Tabla de verdad:

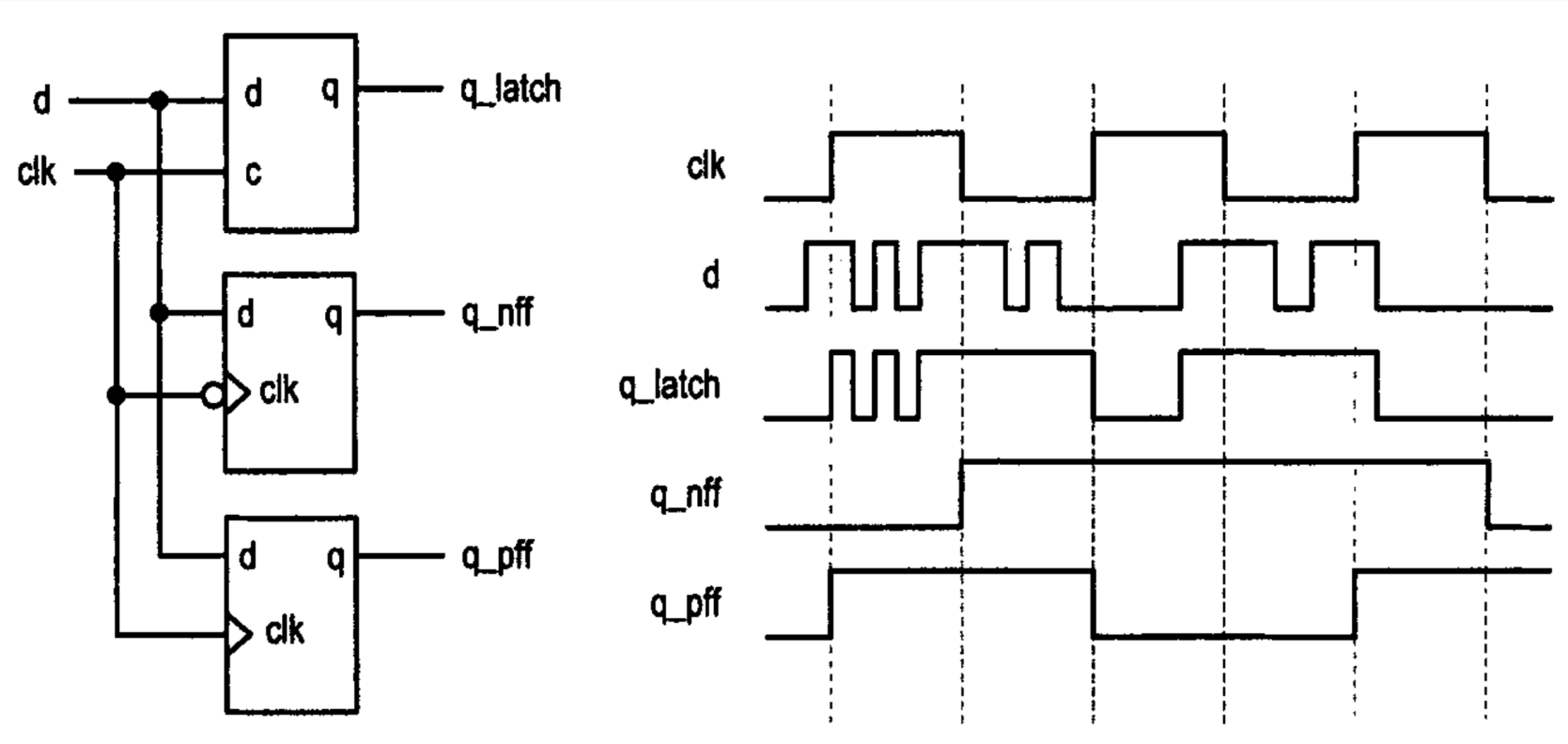
D	clk	Q_{T+1}	$\overline{Q_{T+1}}$
1	0	Q_T	$\overline{Q_T}$
0	1↑	0	1
0	0	Q_T	$\overline{Q_T}$
1	1↑	1	0

Siendo $T = n.T_{clock}$ y $T + 1 = (n + 1)T_{clock}$, donde:

- T_clock es el período del clock (tiempo que dura un ciclo)
- n es una cierta cantidad de pulsos de clock

Flip-Flop D (Delay)

Ahora podemos entender bien las diferencias:



Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

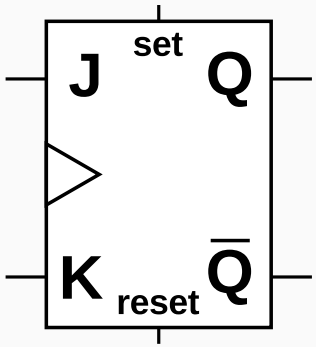


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$		
0	1	$1\uparrow$		
0	0	$1\uparrow$		
1	1	$1\uparrow$		
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

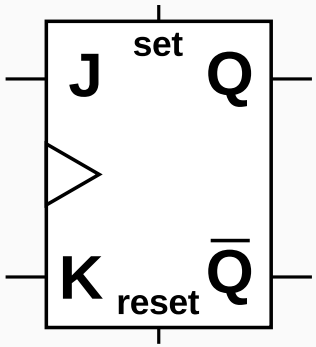


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$		0
0	1	$1\uparrow$		
0	0	$1\uparrow$		
1	1	$1\uparrow$		
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

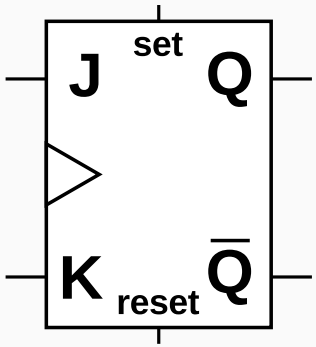


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$		
0	0	$1\uparrow$		
1	1	$1\uparrow$		
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

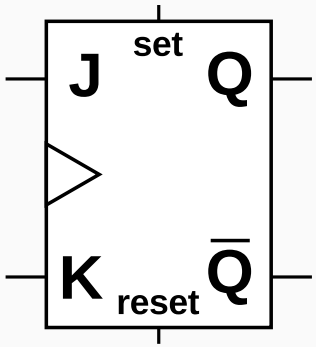


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$	0	1
0	0	$1\uparrow$		
1	1	$1\uparrow$		
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

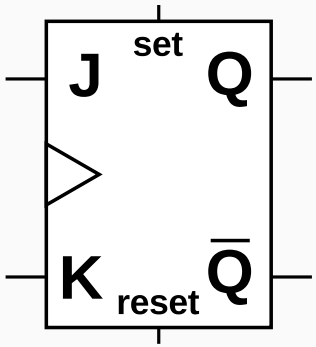


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$	0	1
0	0	$1\uparrow$	Q_T	$\overline{Q}_T$
1	1	$1\uparrow$		
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

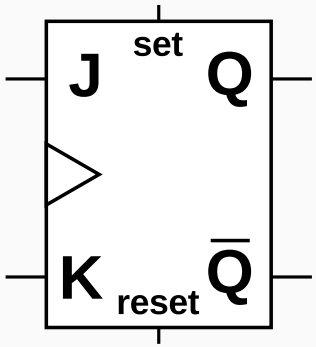


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$	0	1
0	0	$1\uparrow$	Q_T	$\overline{Q}_T$
1	1	$1\uparrow$	$\overline{Q}_T$	Q_T
x	x	0		

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

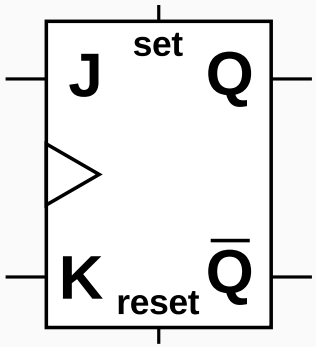


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$	0	1
0	0	$1\uparrow$	Q_T	$\overline{Q}_T$
1	1	$1\uparrow$	$\overline{Q}_T$	Q_T
x	x	0	Q_T	Q_T

Flip-Flop J-K

Volviendo al latch J-K, ahora con detección de flanco podemos obtener un comportamiento más *adecuado*:

Ahora lo podemos representar como:

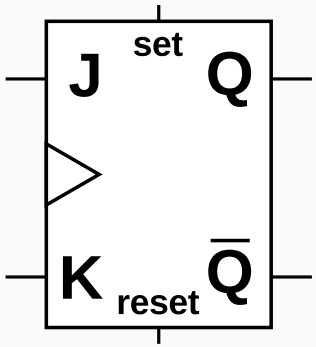


Tabla de verdad:

J	K	clk	Q_{T+1}	$\overline{Q}_{T+1}$
1	0	$1\uparrow$	1	0
0	1	$1\uparrow$	0	1
0	0	$1\uparrow$	Q_T	$\overline{Q}_T$
1	1	$1\uparrow$	$\overline{Q}_T$	Q_T
x	x	0	Q_T	Q_T

Ahora en el caso crítico donde $J, K = (1, 1)$ la salida tiene un estado y un tiempo de cambio bien definido:

Se niega el valor anterior cada 1 colck

Registros y memorias

Registros

Ya vimos como un FF D puede almacenar un bit... ¡pero sólo durante un clock!

Registros

Ya vimos como un FF D puede almacenar un bit... ¡pero sólo durante un clock!

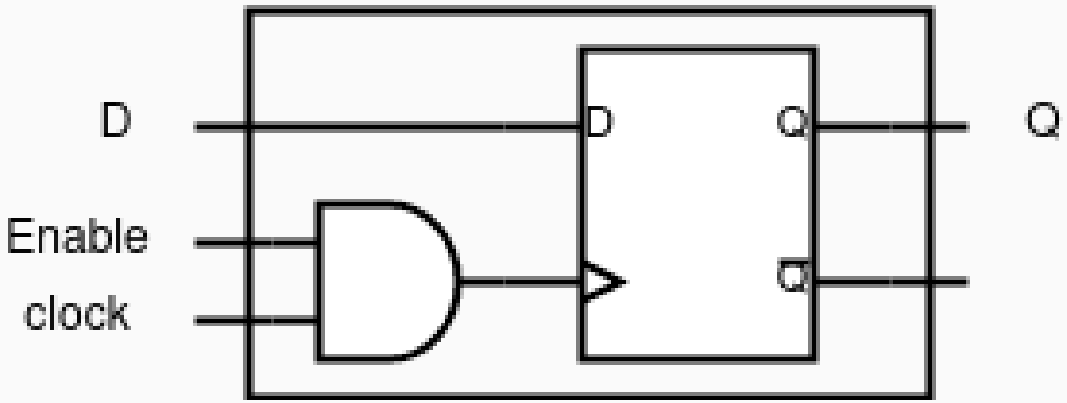
- Debemos poder elegir con una entrada adicional de control por cuanto tiempo queremos almacenar $\Rightarrow$ **enable**.

Registros

Ya vimos como un FF D puede almacenar un bit... ¡pero sólo durante un clock!

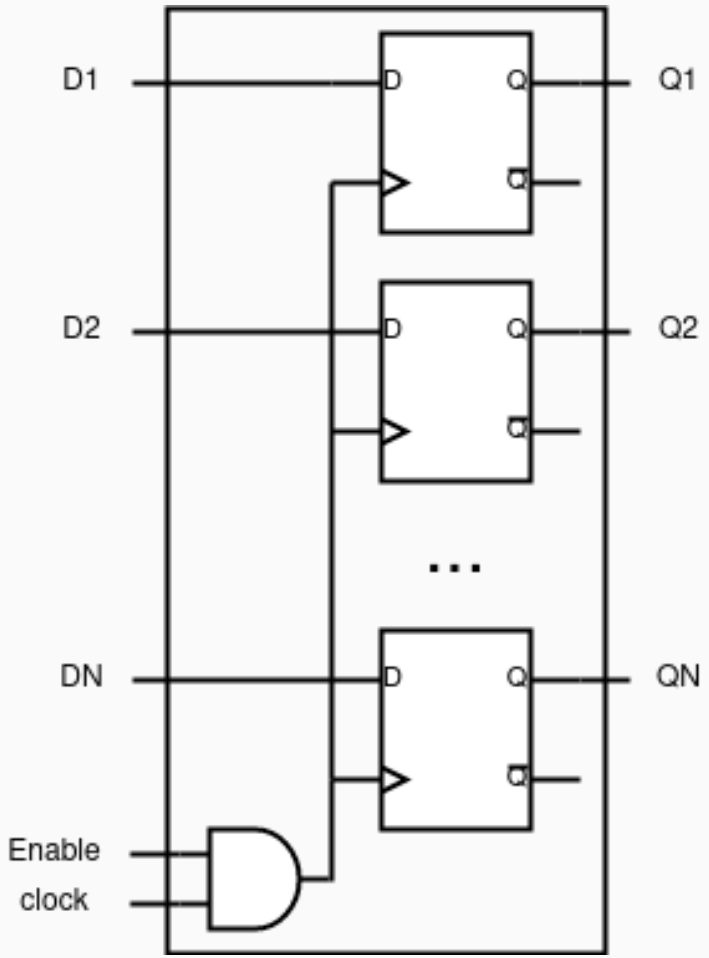
- Debemos poder elegir con una entrada adicional de control por cuanto tiempo queremos almacenar $\Rightarrow$ **enable**.

¡Sencillo!:



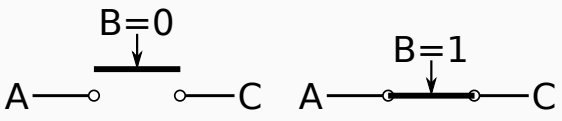
Registro de N-bits

Podemos componer la solución anterior para poder almacenar N bits:



Componentes de Tres Estados

Noción Eléctrica



Símbolo

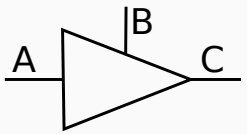
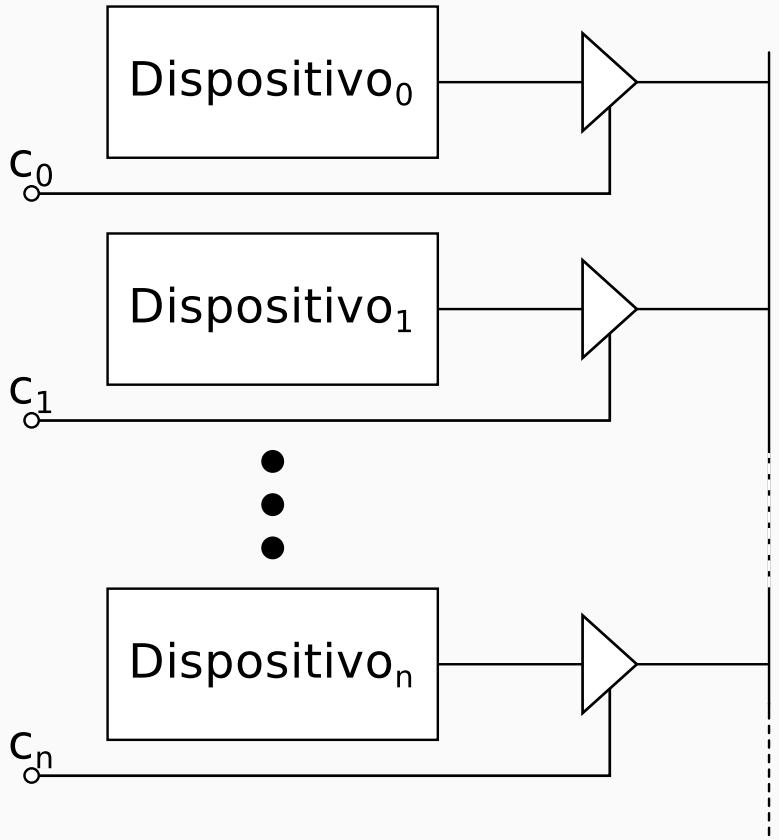


Tabla de Verdad

A	B	C
0	1	0
1	1	1
_	0	Hi-Z

Hi-Z significa "alta impedancia", es decir, que tiene una resistencia alta al pasaje de corriente. Como consecuencia de esto, podemos considerar al pin C como desconectado del circuito.

Componentes de Tres Estados



IMPORTANTE: Sólo deben ser usados a la salida de componentes para permitirles conectarse a un medio compartido (bus).

Ejercicio 0

a) Diseñar un registro de 3 *bits*. El mismo debe contar con 3 entradas $e_0, \ldots, e_2$ para ingresar el dato a almacenar, 3 salidas $s_0, \ldots, s_2$ para ver el dato almacenado y las señales de control CLK, RESET y WRITEENABLE.

Ejercicio 0

a) Diseñar un registro de 3 *bits*. El mismo debe contar con 3 entradas $e_0, \ldots, e_2$ para ingresar el dato a almacenar, 3 salidas $s_0, \ldots, s_2$ para ver el dato almacenado y las señales de control CLK, RESET y WRITEENABLE.

b) Modificar el diseño anterior agregándole componentes de 3 estados para que sólo cuando se active la señal de control ENABLEOUT muestre el dato almacenado.

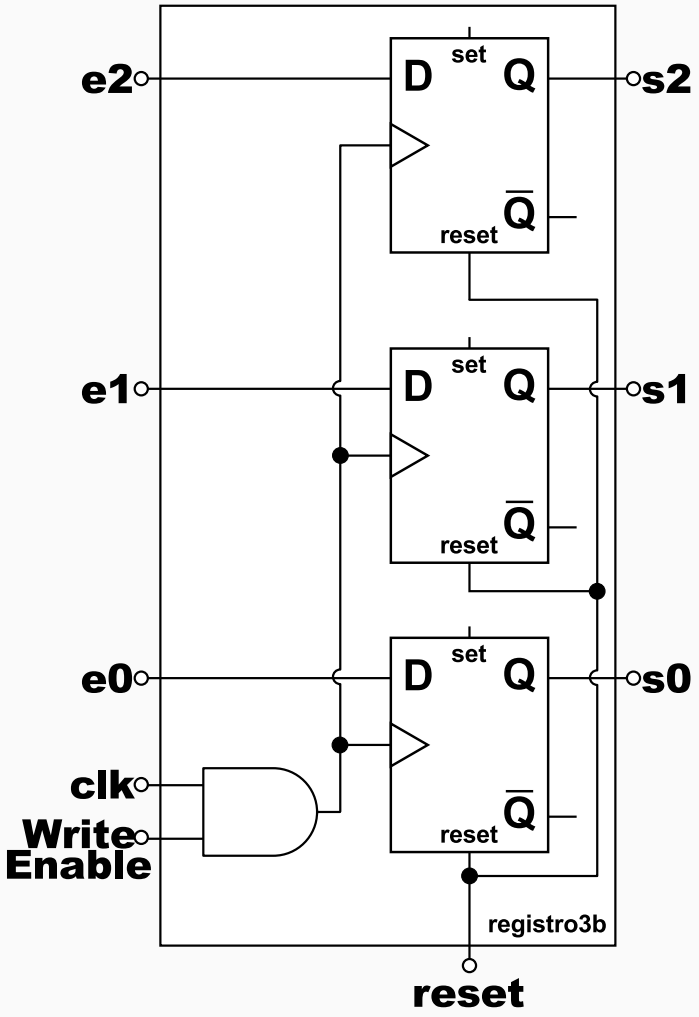
Ejercicio 0

a) Diseñar un registro de 3 *bits*. El mismo debe contar con 3 entradas $e_0, \ldots, e_2$ para ingresar el dato a almacenar, 3 salidas $s_0, \ldots, s_2$ para ver el dato almacenado y las señales de control CLK, RESET y WRITEENABLE.

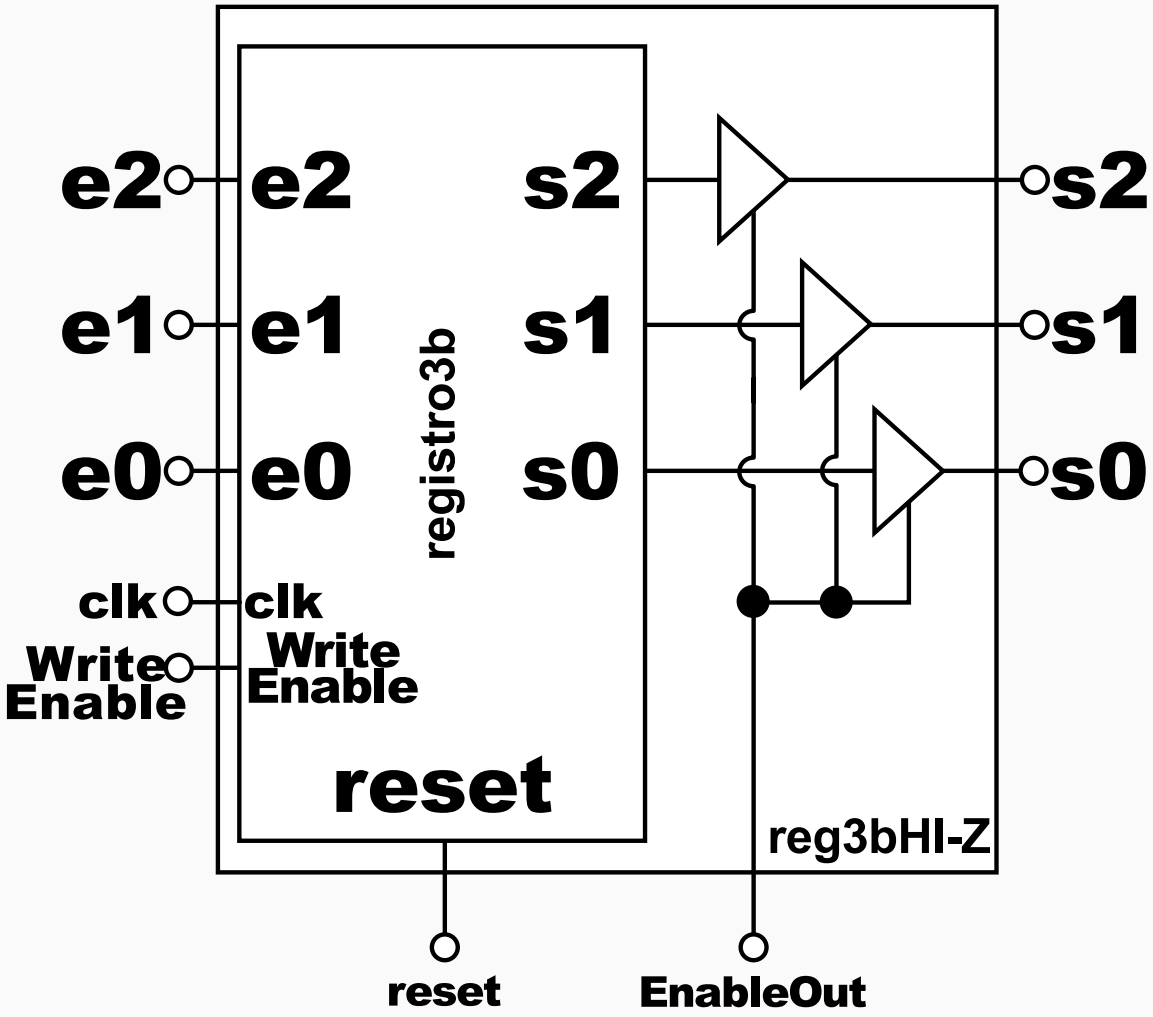
b) Modificar el diseño anterior agregándole componentes de 3 estados para que sólo cuando se active la señal de control ENABLEOUT muestre el dato almacenado.

c) Modificar nuevamente el diseño para que e_i y s_i estén conectadas entre sí al mismo tiempo teniendo en lugar de 3 entradas y 3 salidas, 3 entrada-salidas

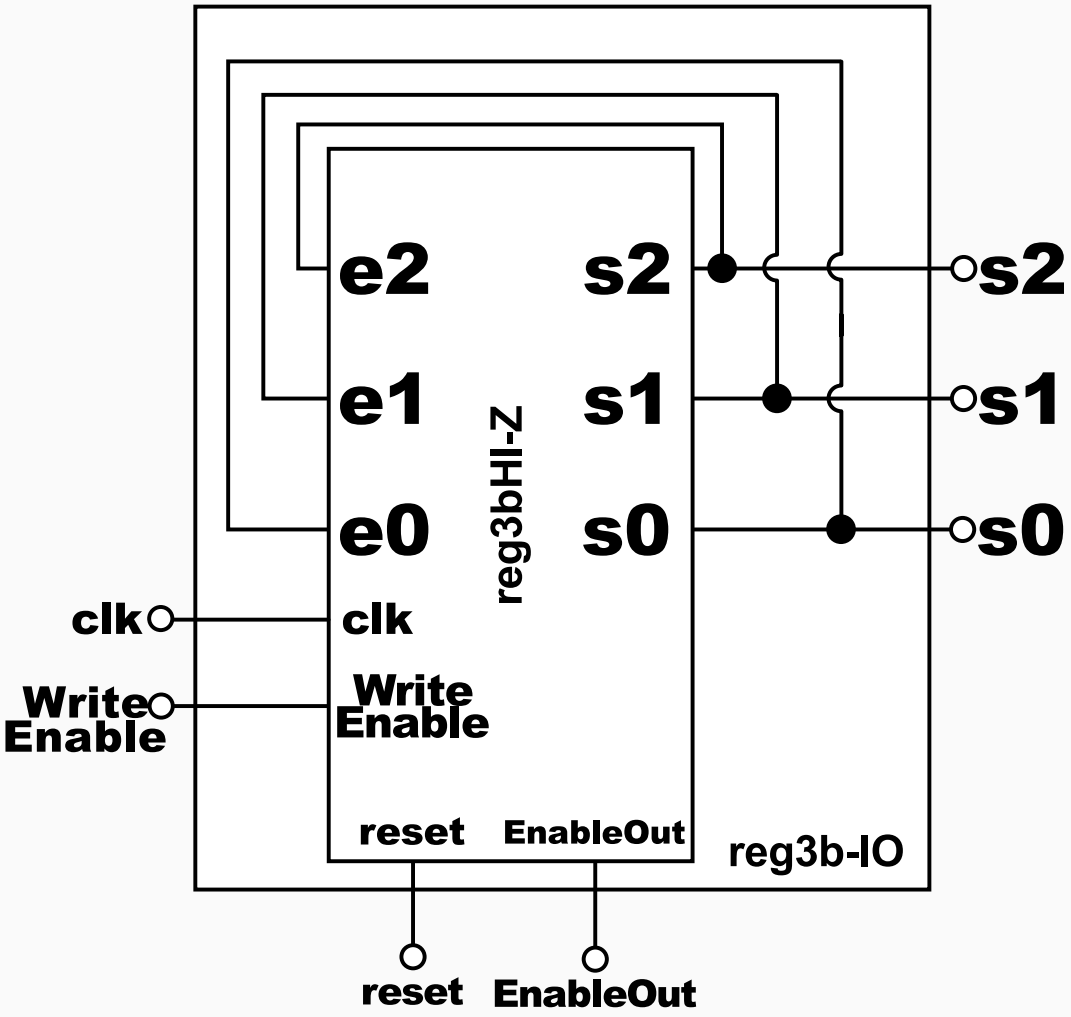
Solución - Ejercicio 0.a



Solución - Ejercicio 0.b



Solución - Ejercicio 0.c



Ejercicio 1

a) Realizar el esquema de interconexión de n registros como el diseñado

Ejercicio 1

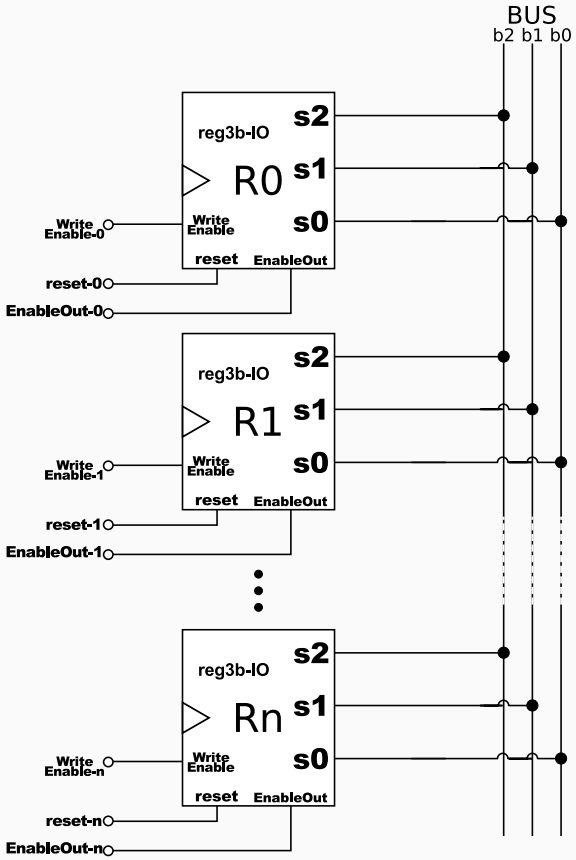
a) Realizar el esquema de interconexión de n registros como el diseñado

b) Dar una secuencia de valores de las señales de control para que se copie el dato del R1 al R0

Ejercicio 1

a) Realizar el esquema de interconexión de n registros como el diseñado

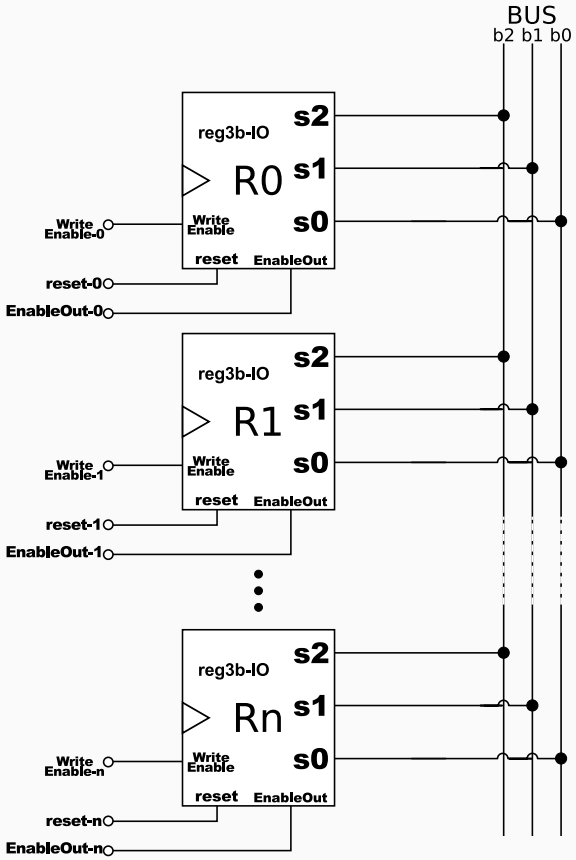
b) Dar una secuencia de valores de las señales de control para que se copie el dato del R1 al R0



Ejercicio 1

a) Realizar el esquema de interconexión de n registros como el diseñado

b) Dar una secuencia de valores de las señales de control para que se copie el dato del R1 al R0



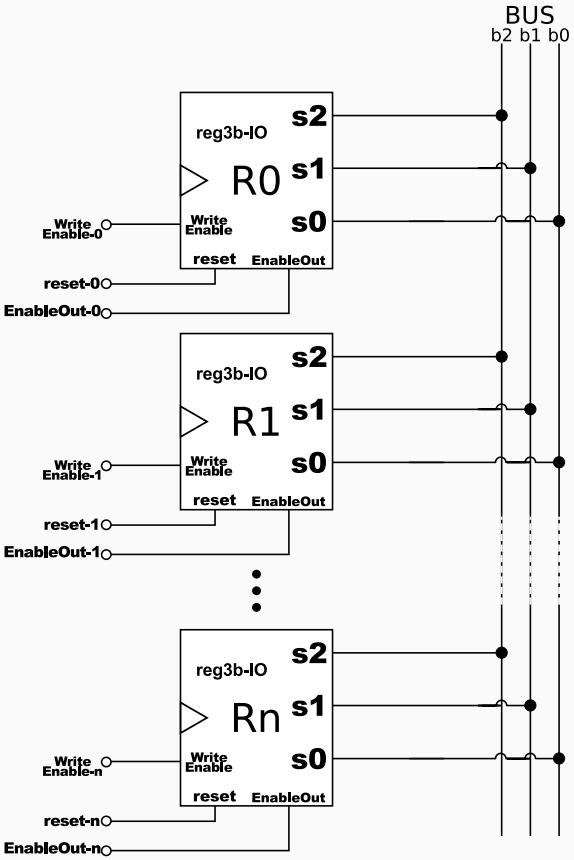
Señales de control:

R0	R1	...	Rn
WriteEnable-0	WriteEnable-1	...	WriteEnable-n
reset-0	reset-1	...	reset-n
EnableOut-0	EnableOut-1	...	EnableOut-n

Ejercicio 1

a) Realizar el esquema de interconexión de n registros como el diseñado

b) Dar una secuencia de valores de las señales de control para que se copie el dato del R1 al R0



Señales de control:

R0	R1	...	Rn
WriteEnable-0	WriteEnable-1	...	WriteEnable-n
reset-0	reset-1	...	reset-n
EnableOut-0	EnableOut-1	...	EnableOut-n

Inician todas las señales en 0. Luego se sigue la siguiente secuencia:

- EnableOut-1 $\leftarrow$ 1
- WriteEnable-0 $\leftarrow$ 1
- ...clk....
- WriteEnable-0 $\leftarrow$ 0
- EnableOut-1 $\leftarrow$ 0

Memorias (intro)

Conceptualmente podemos pensar una memoria como M posiciones de almacenamiento de N bits cada una.

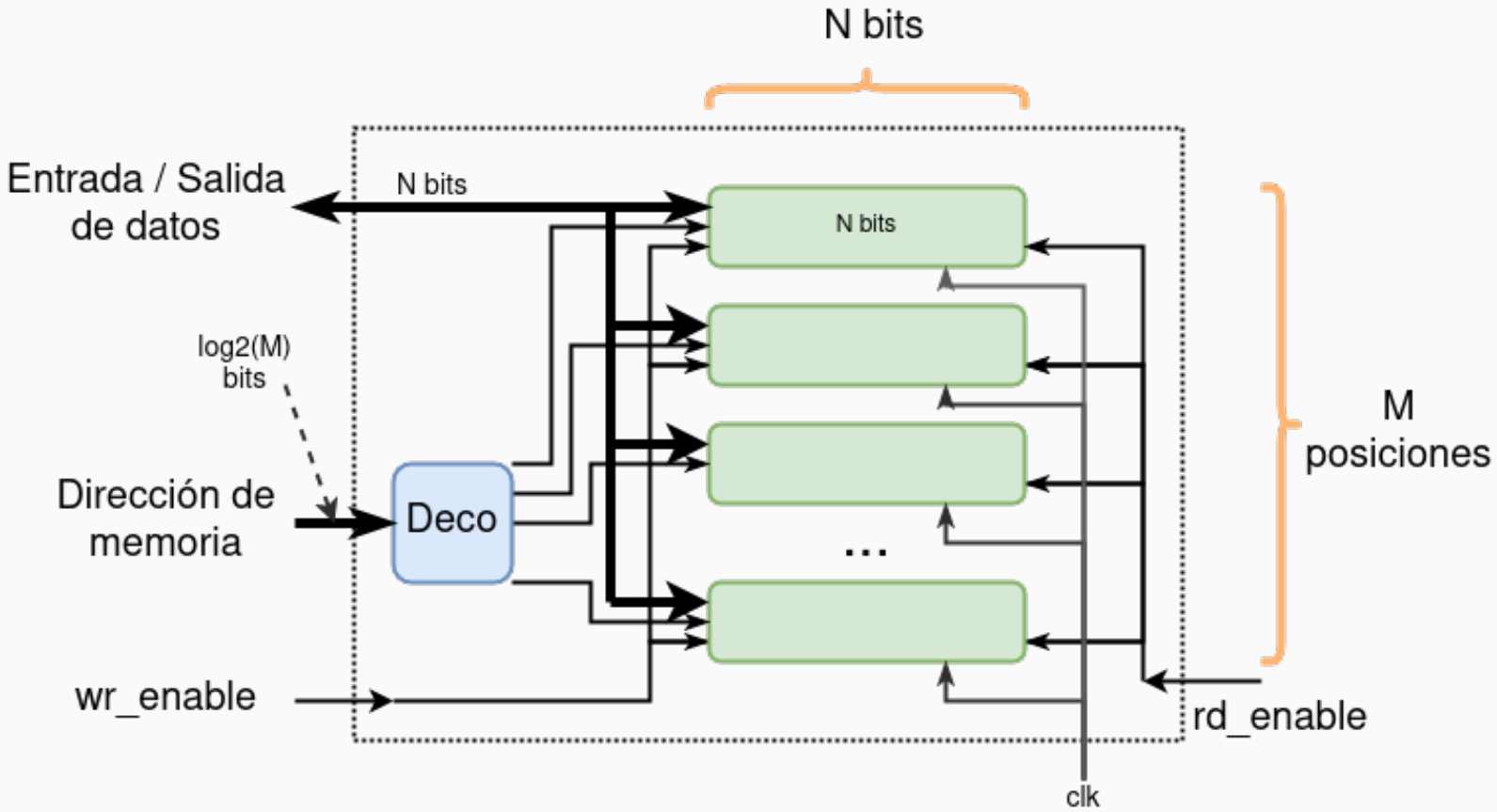
Memorias (intro)

Conceptualmente podemos pensar una memoria como M posiciones de almacenamiento de N bits cada una.

Debemos poder seleccionar a cuál queremos acceder

Memorias (intro)

Conceptualmente podemos pensar una memoria como M posiciones de almacenamiento de N bits cada una.
Debemos poder seleccionar a cuál queremos acceder



Conclusiones

Conclusiones

- Estudiamos la realimentación en circuitos para mantener un dato en el tiempo y vimos implementaciones de *latchs*
- Estudiamos los problemas asociados a los tiempos de propagación de las señales
- Analizamos el uso de un *clock* para limitar la cantidad de estados, controlar las transiciones y evitar carreras
- Vimos algunas implementaciones de flip-flops, registros y memorias

Conclusiones

- Estudiamos la realimentación en circuitos para mantener un dato en el tiempo y vimos implementaciones de *latchs*
- Estudiamos los problemas asociados a los tiempos de propagación de las señales
- Analizamos el uso de un *clock* para limitar la cantidad de estados, controlar las transiciones y evitar carreras
- Vimos algunas implementaciones de flip-flops, registros y memorias

Conclusiones

- Estudiamos la realimentación en circuitos para mantener un dato en el tiempo y vimos implementaciones de *latchs*
- Estudiamos los problemas asociados a los tiempos de propagación de las señales
- Analizamos el uso de un *clock* para limitar la cantidad de estados, controlar las transiciones y evitar carreras
- Vimos algunas implementaciones de flip-flops, registros y memorias

Conclusiones

- Estudiamos la realimentación en circuitos para mantener un dato en el tiempo y vimos implementaciones de *latchs*
- Estudiamos los problemas asociados a los tiempos de propagación de las señales
- Analizamos el uso de un *clock* para limitar la cantidad de estados, controlar las transiciones y evitar carreras
- Vimos algunas implementaciones de flip-flops, registros y memorias

La práctica...

- Con lo visto hoy pueden realizar la **parte A de la práctica 2**.
- Pueden usar el purpleLogisim evolution (Requiere Java 16 o superior. Para ejecutarlo, teclear en una consola `java -jar logisim-evolution-3.8.0-all.jar` desde la carpeta donde se encuentra el archivo descargado.)
- El próximo jueves deberíamos el **primer taller** de la materia, el cual es **obligatorio**. Será en los laboratorios del pabellón **Cero+Infinito** (ver cuales en el cronograma que está en el campus).
- Bibliografía recomendada: *The Essentials of Computer Organization and Architecture - Linda Null - Capítulo 3*

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Arquitectura 1/2

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Clase de hoy

Hoy vamos a ver:

- Definición de arquitecturas.

Clase de hoy

Hoy vamos a ver:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.

Clase de hoy

Hoy vamos a ver:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.

Clase de hoy

Hoy vamos a ver:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Codificación de instrucciones, compilación y ensamblado.

¿Qué es la arquitectura, o mejor dicho, la arquitectura de un procesador? La arquitectura de un procesador se refiere a aquello con lo que podemos trabajar cuando escribimos un programa. Son las instrucciones, los registros y la forma de acceder a memoria, definiendo así la estructura lógica y comportamental del procesador.

¿Cómo interactuamos con la arquitectura de un procesador?
Escribiendo un programa en un lenguaje ensamblador, o sea, el lenguaje que el procesador entiende.

¿Qué cosa no es la arquitectura de un procesador? La implementación física específica del procesador que le permite ejecutar estos programas. Puede haber varias implementaciones distinas de una misma arquitectura pertenecientes a una o varias empresas, para el programa, siempre y cuando respeten lo que la arquitectura define, van a ser intercambiables.

¿Qué elementos expuestos a quién programa constituyen una arquitectura?

¿Qué elementos expuestos a quién programa constituyen una arquitectura?

- El conjunto de instrucciones.

¿Qué elementos expuestos a quién programa constituyen una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.

¿Qué elementos expuestos a quién programa constituyen una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.
- La forma de acceder a la memoria.

¿Qué elementos expuestos a quién programa constituyen una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.
- La forma de acceder a la memoria.

¿Qué es una instrucción, un registro o una memoria?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

- ¿Qué comportamiento tiene este programa?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?
- ¿Cómo se representan y almacenan las variables `a` y `largo`?

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Preguntémonos:

- ¿Qué comportamiento tiene este programa?
- ¿Cómo interpreta el procesador la línea 5? ¿Esto se realiza en una o varias instrucciones de lenguaje máquina?
- ¿Cómo se representan y almacenan las variables `i` y `acumulador`?
- ¿Cómo se representan y almacenan las variables `a` y `largo`?
- ¿Cómo se decide cuál es la próxima instrucción a ejecutar?

Programa de ejemplo en C

Al final de la clase vamos a poder responder todas estas preguntas, en el contexto de una arquitectura en particular. A continuación, un pequeño adelanto.

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li    t0, 0          # acumulador = 0
6      li    t1, 0          # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge   t1, a1, fin    # Si i >= largo, sale del ciclo
9      slli  t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10     add   t2, a0, t2     # Actualiza la dir. de memoria
11     lw    t2, 0(t2)      # De-referencia la dir,
12     add   t0, t0, t2     # Agrega el valor al acumulador
13     addi  t1, t1, 1      # Incrementa el iterador
14     j     ciclo          # Vuelve a comenzar el ciclo
15 fin:
16     mv    a0, t0         # Mueve t0 (acumulador) a a0
17     ret                  # Devuelve valor por a0
```

Lenguaje ensamblador

Lenguajes de alto y bajo nivel

Al programar solemos utilizar lenguajes de alto nivel. Estos lenguajes se expresan en un dominio independiente de la arquitectura del procesador donde se vaya a correr el programa.

Proveen un nivel de abstracción basado en:

Proveen un nivel de abstracción basado en:

- Variables que preservan valores (`int a, b = 3;`).

Proveen un nivel de abstracción basado en:

- Variables que preservan valores (`int a, b = 3;`).
- Estructuras de control que permiten modificar la ejecución secuencial del programa (`if`, `switch`, `for`).

Proveen un nivel de abstracción basado en:

- Variables que preservan valores (`int a, b = 3;`).
- Estructuras de control que permiten modificar la ejecución secuencial del programa (`if`, `switch`, `for`).
- Un mecanismo que nos permite realizar una invocación o llamada a una función desde cualquier punto del programa, pasando y recibiendo parámetros (`int foo(int bar)`).

Lenguajes de alto y bajo nivel

Los procesadores pueden ejecutar instrucciones escritas en un lenguaje en particular, que conoce su arquitectura y se expresa estrictamente en términos de sus componentes (instrucciones, registros y memoria). Este es el lenguaje ensamblador de esta arquitectura ()RISC V en nuestro caso).

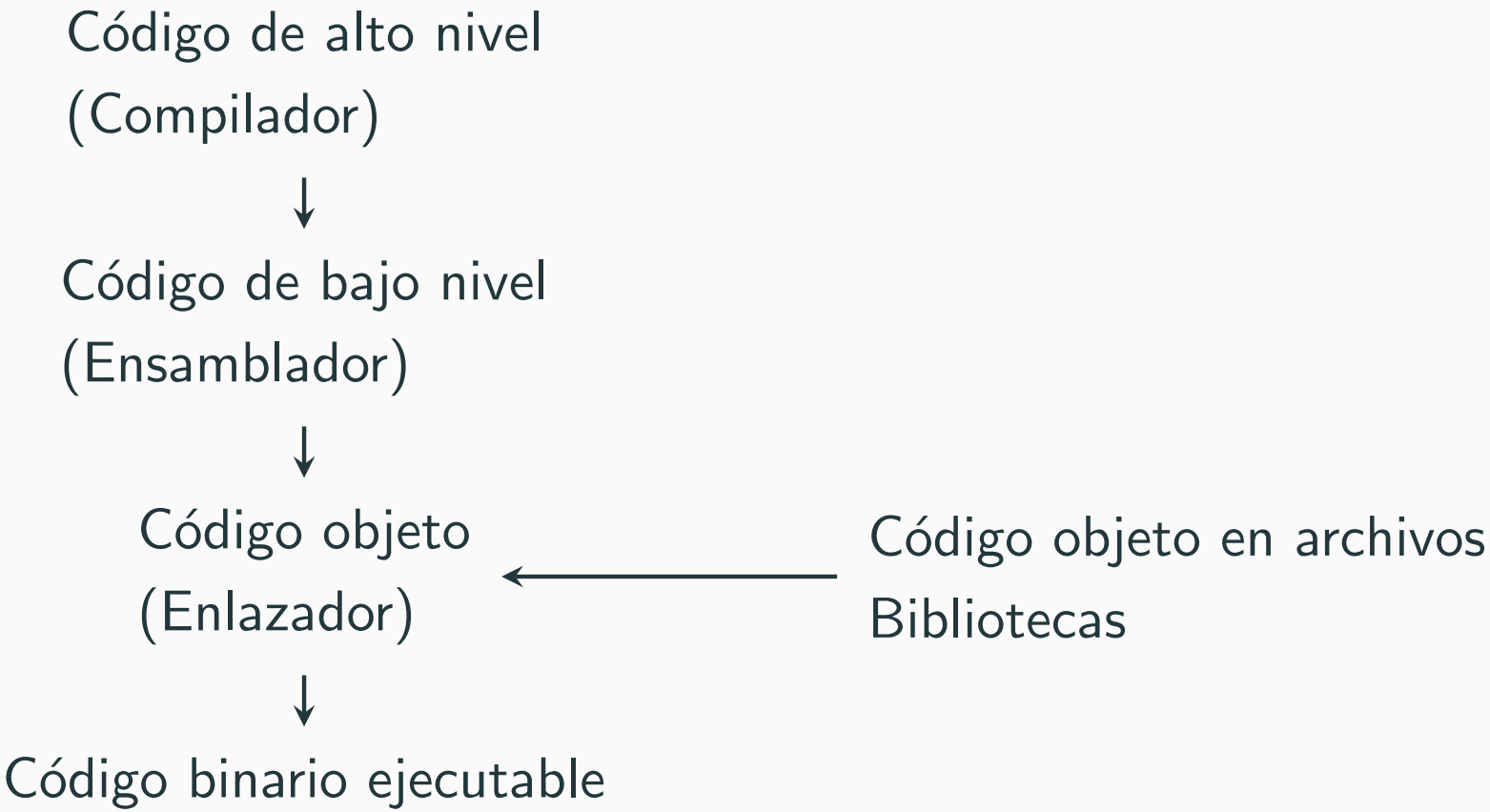
Lenguajes de alto y bajo nivel

Cada arquitectura cuenta con un lenguaje ensamblador particular, en nuestro caso cuando decimos lenguaje ensamblador, nos estamos refiriendo al lenguaje ensamblador de RISC V.
Los lenguajes ensambladores son, en realidad, una familia de lenguajes de bajo nivel.

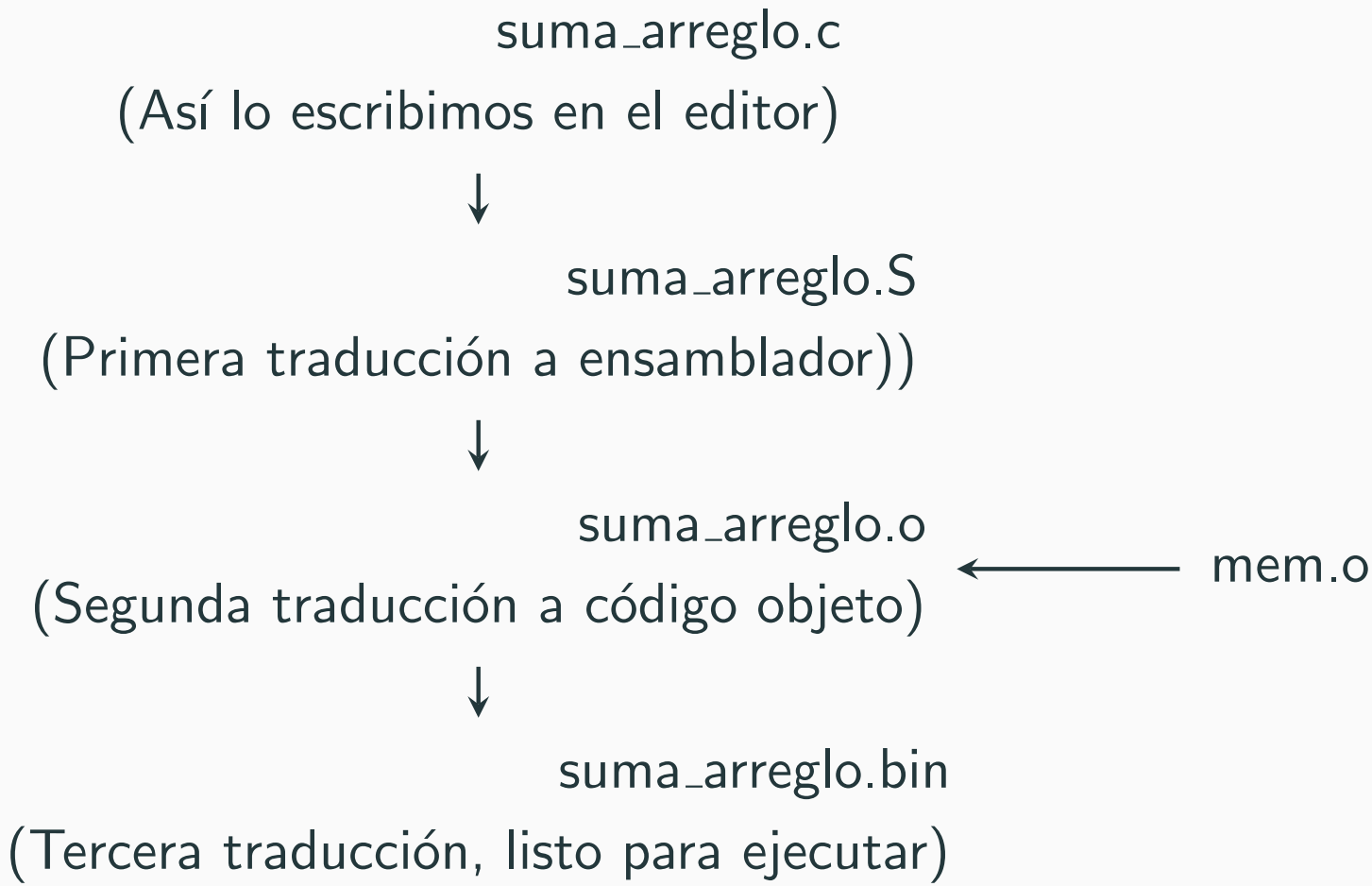
Lenguajes de alto y bajo nivel

Los procesadores implementan una arquitectura y necesitan ser acompañados por programas de compilado, ensamblado y enlazado que permiten escribir código en alto nivel y conseguir que éste se traduzca, en una serie de pasos, en código binario ejecutable. En caso contrario solamente podríamos programar en lenguaje ensamblador.

Lenguajes de alto y bajo nivel



Lenguajes de alto y bajo nivel



La arquitectura `RISC V` es una arquitectura abierta, modular, de uso industrial y que está ganando rápidamente adopción en varios dominios estratégicos.

Una suma en el lenguaje ensamblador de `RISC V` se escribe de la siguiente manera:

C	RISC V
`a = b + c;`	`add a, b, c`

Una suma en el lenguaje ensamblador de `RISC V` se escribe de la siguiente manera:

C	RISC V
`a = b + c;`	`add a, b, c`

La primera parte, `add`, recibe el nombre de mnemónico, e indica el tipo de operación que queremos realizar, en este caso una suma. Los operandos `b` y `c` son los operandos de fuente y `a` el operando destino ya que será el que almacene el valor del resultado de la operación.

Instrucciones compuestas

C	RISC V
`// operaciones compuestas` `a = b + c - d;`	`add t, b, c # t = b + c` `sub a, t, d # a = t - d`

Instrucciones compuestas

C	RISC V
`// operaciones compuestas` `a = b + c - d;`	`add t, b, c # t = b + c` `sub a, t, d # a = t - d`

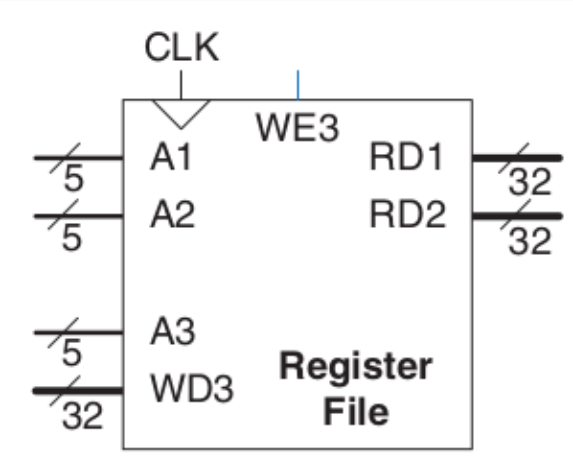
El lenguaje ensamblador no permite la composición de operaciones del modo en que lo hace, por ejemplo, C, por lo que debemos descomponer la operaciones en instrucciones atómicas (una suma y una resta).

La operaciones lógico aritméticas modifican el estado del procesador según su semántica, dichas modificaciones deben realizarse rápidamente debido a que constituyen el grueso del cómputo que ocurre en nuestros procesadores. Es por esto que los operandos de fuente y destino son registros y no direcciones de memoria.

Si queremos realizar operaciones aritméticas o lógicas con datos que se encuentran en memoria, debemos primero mover esos datos de la memoria principal a los registros.

Register file

`RISC V` cuenta con 32 registros que suelen implementarse como un arreglo de memoria estática de 32 bits con varios puertos. A este arreglo se lo suele referir como banco de registros o archivo de registros (register file).



Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.
- Los registros `s0` a `s11` y los `t0` a `t6` se utilizan para almacenar variables.

Nombres de los registros

Los registros pueden nombrarse por su índice, desde `x0` a `x31` o según su uso habitual, que indica el propósito que suele cumplir el registro en el funcionamiento de un programa.

- El registro `zero` (`x0`) almacena siempre el valor 0, y no puede ser escrito. Cualquier operación que lo tenga como operando de destino, descarta la escritura del mismo.
- Los registros `s0` a `s11` y los `t0` a `t6` se utilizan para almacenar variables.
- `ra` y de `a0` a `a7` tienen usos relacionados con las llamadas a función.

Nombres de los registros según su uso

31 … 0	
x0 / zero	Alambrado a cero
x1 / ra	Dirección de retorno
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporal
x6 / t1	Temporal
x7 / t2	Temporal
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Argumento de función, valor de retorno
x11 / a1	Argumento de función, valor de retorno
x12 / a2	Argumento de función
x13 / a3	Argumento de función
x14 / a4	Argumento de función
x15 / a5	Argumento de función
x16 / a6	Argumento de función
x17 / a7	Argumento de función
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporal
x29 / t4	Temporal
x30 / t5	Temporal
x31 / t6	Temporal

32

31 … 0
pc

32

En el lenguaje ensamblador no nos referimos a un conjunto de variables no acotadas y cuyo nombre podemos definir según convenga para la interpretación del programa, sino que contamos con un conjunto fijo de 32 elementos con los que operar.

Por eso, cuando traducimos un programa de un lenguaje de alto nivel a ensamblador debemos decidir en qué registros almacenar los valores de nuestras variables.

Instrucciones compuestas sobre registros

C	RISC V

`// operaciones compuestas`
`a = b + c - d;`
	`# s0 = a, s1 = b`
	`# s2 = c, s3 = d, t0 = t`
	`add t0, s1, s2  # t = b + c`
	`sub s0, t0, s3  # a = t - d`

Instrucciones compuestas sobre registros

C	RISC V

`// operaciones compuestas`
`a = b + c - d;`
	`# s0 = a, s1 = b`
	`# s2 = c, s3 = d, t0 = t`
	`add t0, s1, s2  # t = b + c`
	`sub s0, t0, s3  # a = t - d`

Volvemos al ejemplo anterior utilizando los nombres reales de los registros sobre los que podemos operar.

Valores inmediatos

Las instrucciones de lenguaje ensamblador pueden tener valores constantes como operandos, suelen llamarse valores inmediatos ya que se encuentran disponibles en la misma instrucción (no hace falta recuperar su valor a partir de un registro o desde la memoria).

Valores inmediatos

El valor puede escribirse en decimal, hexadecimal (prefijo `0x`) o binario (prefijo `0b`). Los valores inmediatos son de 12 bits y se extiende su signo a 32 bits antes de operar.

Operando con constantes

C	RISC V

`a = a + 4;`
`b = a -12;`
	`#s0=a, s1=b`
	`addi s0, s0, 4 # a = a + 4`
	`addi s1, s0, -12 # b = a - 12`

Operando con constantes

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`a = a + 4;`
`b = a -12;`
	`#s0=a, s1=b`
	`addi s0, s0, 4 # a = a + 4`
	`addi s1, s0, -12 # b = a - 12`

Podemos definir constantes positivas y negativas como operandos utilizando la operación `addi` (`add inmediate`).

Asignando constantes a registros

C	RISC V

`i = 0;`
`x = 2032;`
`y = -78;`
	`#s4=i, s5=x, s6=y`
	`addi s4, zero, 0 # i = 0`
	`addi s5, zero, 2032 # i = 0`
	`addi s6, zero, -78 # i = 0`

Asignando constantes a registros

C	RISC V

`i = 0;`
`x = 2032;`
`y = -78;`
	`#s4=i, s5=x, s6=y`
	`addi s4, zero, 0 # i = 0`
	`addi s5, zero, 2032 # i = 0`
	`addi s6, zero, -78 # i = 0`

Podemos definir constantes positivas y negativas como operandos.

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE #s2=0xABCDE000` `addi s2, s2, 0x123 #s2=0xABCDE123`

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE #s2=0xABCDE000` `addi s2, s2, 0x123 #s2=0xABCDE123`

Como los valores inmediatos son de 12 bits y se los extiende respetando el signo a 32 bits cuando realizamos una operación, cargar una constante de 32 bits requiere que hagamos dos operaciones.

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE #s2=0xABCDE000` `addi s2, s2, 0x123 #s2=0xABCDE123`

Primero cargamos los veinte bits más altos con la instrucción `lui`(load upper inmediate) y luego los 12 bits más bajos con un `addi` como veníamos haciendo.

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

Si la parte baja se expresa como un número negativo (bit más alto en 1), al extender el signo va a cargar con unos la parte alta. Por eso tenemos que tener esto en cuenta.

Valores inmediatos de 32 bits

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

La parte alta con todos unos equivale a un menos uno en complemento a dos, por lo cual, para compensar el efecto de la extensión del signo en la suma, se incrementa en uno la parte alta que vamos a cargar. En el ejemplo hacemos `lui s2, 0xFEEDB` en lugar de `lui s2, 0xFEEDA`.

Memoria

El tipo de operando que resta presentar es el de memoria. La memoria se estructura y accede como si fuera un arreglo de elementos de 32 bits (4 bytes).
El acceso a memoria es significativamente más lento que el acceso a registros pero nos permite acceder a mucha más información que si tuviésemos que operar solamente con registros.

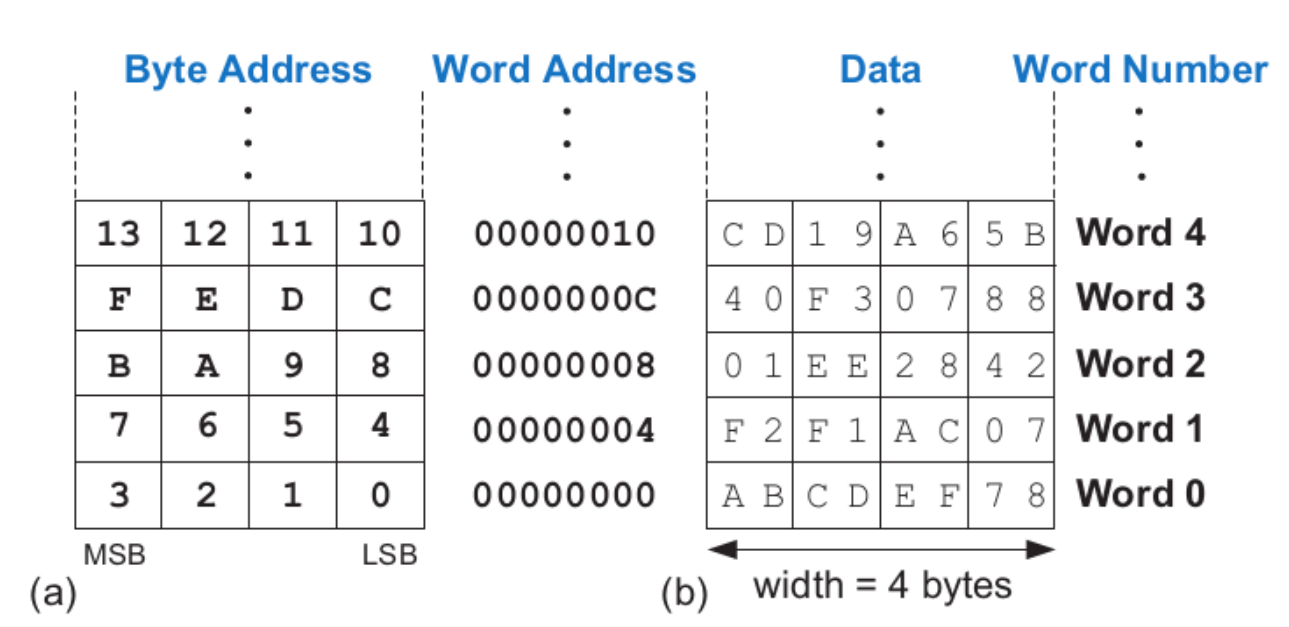
Direcciones de memoria

`RISC V` permite acceder a la memoria con índices (direcciones) de 32 bits, o sea 4.294.967.296 índices posibles.

Pero cabe notar que el índice apunta a un byte en particular, o sea, a uno de los cuatro bytes de la palabra, de modo que entre una palabra de 32 bits y otra, los índices avanzan en cuatro unidades.

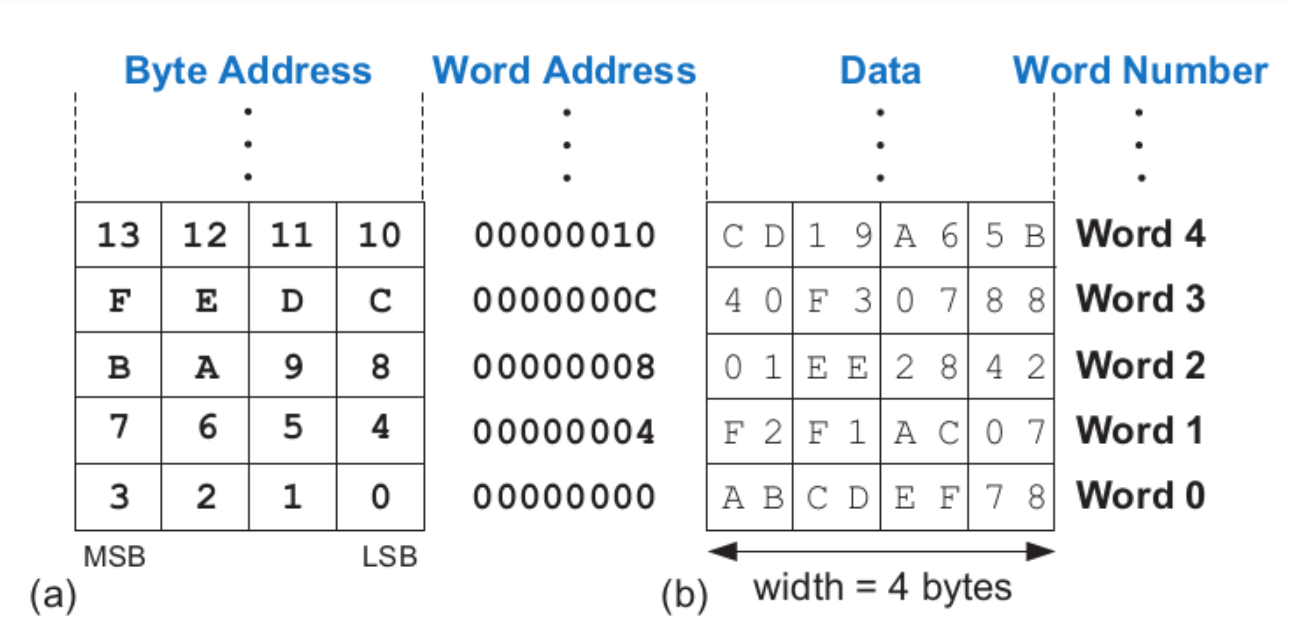
Podemos indicar que la lectura o escritura se hará a partir de un byte en particular.

Accesos a memoria (direcciones)



A la izquierda (a), vemos los índices de memoria (`byte address`) representados de derecha a izquierda, donde a la derecha vemos el byte menos significativo (LSB) y a la derecha el byte más significativo de la palabra (MSB). La dirección de palabra (word address) corresponde al índice del byte menos significativo de ésta.

Accesos a memoria (datos)



A la derecha (b) vemos los datos ordenados según palabras de 32 bits (4 bytes) y el número de palabra (word number). La relación entre número de palabra y dirección de palabra es:

$$\text{word address} * 4 = \text{word number}$$

Leyendo y escribiendo datos

Para operar con la memoria utilizamos las instrucciones `lw` (load word) para leer una palabra de memoria en un registro y `sw` (store word) para escribir una palabra desde un registro a la memoria. Las direcciones se definen como:

$$\text{dirección} = \text{base} + \text{desplazamiento}$$

Donde la base será el valor de un registro y el desplazamiento una constante con signo de 12 bits.

Leyendo un arreglo

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V
`int a = mem[2];`	`#s7 = a, s3 = mem` `lw, s7, 8(s3)`

Leyendo un arreglo

C	RISC V
`int a = mem[2];`	`#s7 = a, s3 = mem` `lw, s7, 8(s3)`

Si suponemos que los datos del arreglo `mem` son palabras de 4 bytes, y que la posición de memoria en la que comienza el arreglo está almacenada en `s3`, la forma de leer el tercer dato del arreglo (recordemos que el primer dato se encuentra en `mem[0]`) es indicando `s3` como la base y 8 como el desplazamiento, ya que la memoria se accede con índices que apuntan de a byte y cada dato tiene 4 bytes (4 * 2 = 8).

Escribiendo en un arreglo

C	RISC V

`mem[5] = 33;`	`#s3 = mem`
	`addi t3, zero, 33`
	`sw, t3, 20(s3)`

Escribiendo en un arreglo

C	RISC V

`mem[5] = 33;`	`#s3 = mem`
	`addi t3, zero, 33`
	`sw, t3, 20(s3)`

Si suponemos que los datos del arreglo `mem` son palabras de 4 bytes, y que la posición de memoria en la que comienza el arreglo está almacenada en `s3`, la forma de escribir el quinto dato del arreglo (recordemos que el primer dato se encuentra en `mem[0]`) es indicando `s3` como la base y 20 como el desplazamiento ya que la memoria se accede con índices que apuntan de a byte y cada dato tiene 4 bytes (4 * 5 = 20).

Hasta este punto se presentó lo siguiente:

Hasta este punto se presentó lo siguiente:

- Definición de arquitectura.

Hasta este punto se presentó lo siguiente:

- Definición de arquitectura.
- Definición de lenguajes de alto y bajo nivel.

Hasta este punto se presentó lo siguiente:

- Definición de arquitectura.
- Definición de lenguajes de alto y bajo nivel.
- Lenguaje ensamblador de RISCV.

Hasta este punto se presentó lo siguiente:

- Definición de arquitectura.
- Definición de lenguajes de alto y bajo nivel.
- Lenguaje ensamblador de RISCV.
- Operaciones, operandos, uso de registros, constantes y memoria.

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li    t0, 0          # acumulador = 0
6      li    t1, 0          # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge   t1, a1, fin    # Si i >= largo, sale del ciclo
9      slli  t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10     add   t2, a0, t2     # Actualiza la dir. de memoria
11     lw    t2, 0(t2)      # De-referencia la dir,
12     add   t0, t0, t2     # Agrega el valor al acumulador
13     addi  t1, t1, 1      # Incrementa el iterador
14     j     ciclo          # Vuelve a comenzar el ciclo
15 fin:
16     mv    a0, t0         # Mueve t0 (acumulador) a a0
17     ret                  # Devuelve valor por a0
```

Intervalo

Programando con RISCV

Uno de los principios fundamentales de los procesadores es el de programa almacenado en memoria, eso significa que las instrucciones que describen el comportamiento de un programa se almacenan (siguiendo un formato particular) en la memoria del procesador, la misma que se accede en las operaciones de lectura y escritura (`sw`,`lw`).

Tamaño de la instrucción

Cada instrucción ocupa 32 bits (una palabra), por lo cual sus direcciones se incrementan en múltiplos de 4, recordemos que la arquitectura RISC V permite acceder a la memoria con direcciones que refieren al byte menos significativo a partir del cual leer o escribir la palabra.

Fetch-decode-execute

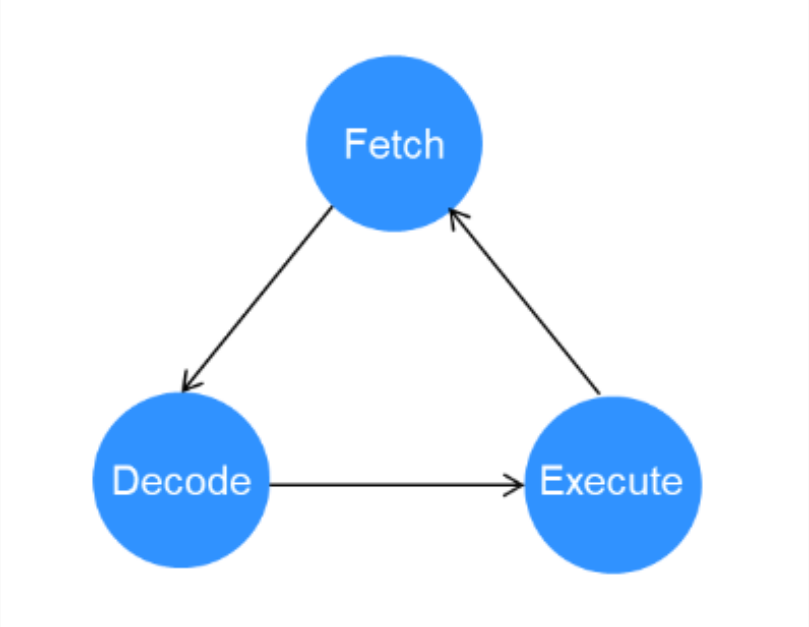
Dirección	Instrucción almacenada
0x538	`addi s1, s2, 3`
0x53C	`lw t2, 8(s1)`
0x540	`sw s3, 3(t6)`

Fetch-decode-execute

Dirección	Instrucción almacenada
0x538	`addi s1, s2, 3`
0x53C	`lw t2, 8(s1)`
0x540	`sw s3, 3(t6)`

El procesador ejecuta el programa almacenando la posición de memoria de la instrucción que se está ejecutando en un registro de 32 bits conocido como el program counter (PC). Va a cargar el contenido de la instrucción de memoria (fetch), ejecutarla (execute) y luego incrementar el PC en 4 posiciones para repetir el ciclo. Al comenzar este programa se carga la instrucción de la posición `0x538`, se la ejecuta, se incrementa el `PC` a `0x53C` y se vuelve a repetir el ciclo.

Fetch-decode-execute



A esto lo llamamos ciclo de fetch decode execute o ciclo de instrucción. Ya que al ejecutar un programa se carga una instrucción de memoria (fetch), se la decodifica para configurar el procesador según su tipo (decode) y luego se actualiza el estado del procesador (registros y memoria), de acuerdo al a semántica de la instrucción (execute).

En la sección de control de ejecución condicional veremos la importancia que tiene el valor del program counter.

Instrucciones

Instrucciones lógicas

El set de instrucciones de RISC V cuenta con instrucciones lógicas como la conjunción (`and`), disyunción (`or`) y la disyunción excluyente (`xor`).

En el diagrama vemos los valores de los registros `s1` y `s2`, representados en formato binario, y luego los resultados de aplicar las operaciones lógicas con distintos operandos de destino utilizando los anteriores como fuente.

	Source registers			
s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`and s3, s1, s2`	**s3**	0100 0110	1010 0001	0000 0000	0000 0000
`or s4, s1, s2`	**s4**	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	**s5**	1011 1001	0101 1110	1111 0001	1011 0111

Algunos usos típicos de las instrucciones lógicas son:

Algunos usos típicos de las instrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.

Algunos usos típicos de las instrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.
- `and`: Nos permite limpiar partes de un registro, si quisiéramos preservar solamente la parte baja de `0xBABAC0C0` podemos hacer un `and` con `0x0000FFFF` consiguiendo `0x0000C0C0`.

Algunos usos típicos de las instrucciones lógicas son:

- `or`: Combinar dos registros que sólo tienen asignada la parte alta y baja respectivamente, un `or` entre `0xFEED0000` y `0x0000F0CA` resulta en `0xFEEDF0CA`.
- `and`: Nos permite limpiar partes de un registro, si quisiéramos preservar solamente la parte baja de `0xBABAC0C0` podemos hacer un `and` con `0x0000FFFF` consiguiendo `0x0000C0C0`.
- `xor`: Conseguir la negación lógica al aplicar la operación a -1, recordemos que -1 se codifica con todos 1, por lo que `xori s1, s2, -1` va a aplicar un `xor` entre `s2` y -1 que se codifica como `0xFFF` en 12 bits y se extiende a `0xFFFFFFFF` al ejecutar, consiguiendo un `xor` contra todos unos, que efectivamente niega el valor.

Las instrucciones de desplazamiento permiten desplazar un valor a izquierda o derecha en una cantidad definida por el segundo operando fuente, si este segundo operando se trata de un inmediato, lo codifica en 5 bits (complemento a dos extendiendo el signo a 32 bits).

Hay tres operaciones posibles:

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.
- `sra` (shift right arithmetic): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con el valor del bit más significativo a izquierda (preserva signo).

Instrucciones de desplazamiento

Hay tres operaciones posibles:

- `sll` (shift left logical): desplaza a izquierda el valor tantas veces como especifique el segundo operando fuente, completando con ceros a derecha.
- `srl` (shift right logical): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con ceros a izquierda.
- `sra` (shift right arithmetic): desplaza a derecha el valor tantas veces como especifique el segundo operando fuente, completando con el valor del bit más significativo a izquierda (preserva signo).

Existen versiones donde el segundo operando fuente es un inmediato (`slli, srli, srai`).

En el diagrama vemos los valores del registro s5, representado en formato binario, y luego los resultados de aplicar las operaciones de desplazamiento.

		Source register			
	s5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly code		Result			
slli t0, s5, 7	t0	1000 1110	0000 1000	0111 0011	1000 0000
srli s1, s5, 17	s1	0000 0000	0000 0000	0111 1111	1000 1110
srai t2, s5, 3	t2	1111 1111	1110 0011	1000 0010	0001 1100

Consiguiendo un byte en particular

Utilizando desplazamientos y máscaras podemos acceder a un byte en particular dentro de una palabra, si tenemos el valor `0xABCDEF00` en el registro `s1` y queremos conseguir el segundo byte (desde el menos significativo) y almacenarlo en `s2` podemos hacer lo siguiente:

```
1 srli t0, s1, 8
2 andi s2, t0, 0xFF
```

La primera instrucción desplaza el valor un byte a la derecha y la segunda preserva solamente el byte menos significativo, que luego almacena en `s2`.

Para poder ejecutar programas que no tengan un flujo secuencial (donde todas las instrucciones se suceden en orden), necesitamos poder saltear instrucciones en nuestro programa o volver a una instrucción anterior, como suele suceder en los lenguajes de alto nivel con las estructuras de `if`, `while`, `for`, `case`.

El mecanismo para conseguir esto en el lenguaje ensamblador de `RISC V` es modificar el valor del registro `PC` (program counter) de modo que la próxima instrucción no sea la siguiente en la memoria sino la que se defina en una instrucción específica.

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.
- `blt`(branch if less than): que reemplaza el valor del `PC` si el primer operando es menor que el segundo.

Las instrucciones de control de flujo van a comparar el valor de los dos primeros operandos, y en función del resultado van reemplazar el valor del `PC` con el del tercer operando. Las instrucciones son:

- `beq`(branch if equal): que reemplaza el valor del `PC` si los dos primeros operandos son iguales.
- `bne`(branch if not equal): que reemplaza el valor del `PC` si los dos primeros operandos son distintos.
- `blt`(branch if less than): que reemplaza el valor del `PC` si el primer operando es menor que el segundo.
- `bge`(branch if greater than or equal): que reemplaza el valor del `PC` si el primer operando es mayor o igual que el segundo.

Existen variantes que interpretan a los operandos como enteros sin signo a la hora de realizar las comparaciones. Sus mnemónicos son `bltu, bgeu`.

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7  target:
8  add s1, s1, s0
```

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7 target:
8  add s1, s1, s0
```

Este ejemplo carga un 4 en `s0` y un 1 en `s1` (`addi`), luego desplaza a `s1` dos posiciones a la izquierda (`slli`), lo cual equivale a multiplicar por 4 y compara sin ambos registros son iguales (`beq`). El último operando es de tipo etiqueta.

Ejemplo y etiquetas


```
1          addi s0, zero, 4
2          addi s1, zero, 1
3          slli s1, s1, 2
4          beq s0, s1, target
5          addi s1, s1, 1
6          sub s1, s1, s0
7          target:
8          add s1, s1, s0
```

Las etiquetas se definen como `nombre:` donde `nombre` es la referencia que podemos usar en otras instrucciones y será interpretada como la dirección de memoria donde se almacena la instrucción inmediatamente siguiente a su definición.

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7 target:
8  add s1, s1, s0 #dir: 0xB400
```

Ejemplo y etiquetas


```
1  addi s0, zero, 4
2  addi s1, zero, 1
3  slli s1, s1, 2
4  beq s0, s1, target
5  addi s1, s1, 1
6  sub s1, s1, s0
7 target:
8  add s1, s1, s0 #dir: 0xB400
```

Si la instrucción `add s1, s1, s0` se encuentra almacenada en la dirección `0xB400`, al evaluar la condición en `beq s0, s1, target` y determinar que los valores de los operandos son iguales, el `PC` será actualizado con el valor `0xB400` y la próxima instrucción a ejecutar será `add s1, s1, s0` en lugar de `addi s1, s1, 1`.

Saltos incondicionales

En los casos anteriores el valor del PC se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del PC se utilizan las instrucciones:

En los casos anteriores el valor del `PC` se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del `PC` se utilizan las instrucciones:

- `j` (jump): que simplemente actualiza el valor del `PC` con el del operando provisto (inmediato de 20 bits extendidos en signo a 32).

En los casos anteriores el valor del `PC` se actualizaba solamente cuando se cumplía una condición luego de comparar el valor de dos operandos. Para realizar una actualización (salto) incondicional del valor del `PC` se utilizan las instrucciones:

- `j` (jump): que simplemente actualiza el valor del `PC` con el del operando provisto (inmediato de 20 bits extendidos en signo a 32).
- `jal` (jump and link): que almacena el valor actual del `PC` en el registro indicado en el primer operando y actualiza el valor del `PC` con el del segundo operando (inmediato de 20 bits extendidos en signo a 32).

Saltos incondicionales


```
1  j target
2  srai s1, s1, 2
3  addi s1, s1, 1
4  sub s1, s1, s0
5 target:
6  add s1, s1, s0
```

Saltos incondicionales


```
1  j target
2  srai s1, s1, 2
3  addi s1, s1, 1
4  sub s1, s1, s0
5 target:
6  add s1, s1, s0
```

En este ejemplo la segunda, tercera y cuarta instrucción no se ejecutan, ya que el salto incondicional de la primera instrucción contínua la ejecución en `add s1, s1, s0`.

Escribiendo un loop

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`addi s0, zero, 1`
	`add s1, zero, zero`
	`#t0=128`
	`addi t0, zero, 128`
	`while:`
	`beq s0, t0, fin`
	`slli s0, s0, 1 #pow=pow*2`
	`addi s1, s1, 1 #x+=1`
	`j while`
	`fin:`

Escribiendo un loop

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`addi s0, zero, 1`
	`add s1, zero, zero`
	`#t0=128`
	`addi t0, zero, 128`
	`while:`
	`beq s0, t0, fin`
	`slli s0, s0, 1 #pow=pow*2`
	`addi s1, s1, 1 #x+=1`
	`j while`
	`fin:`

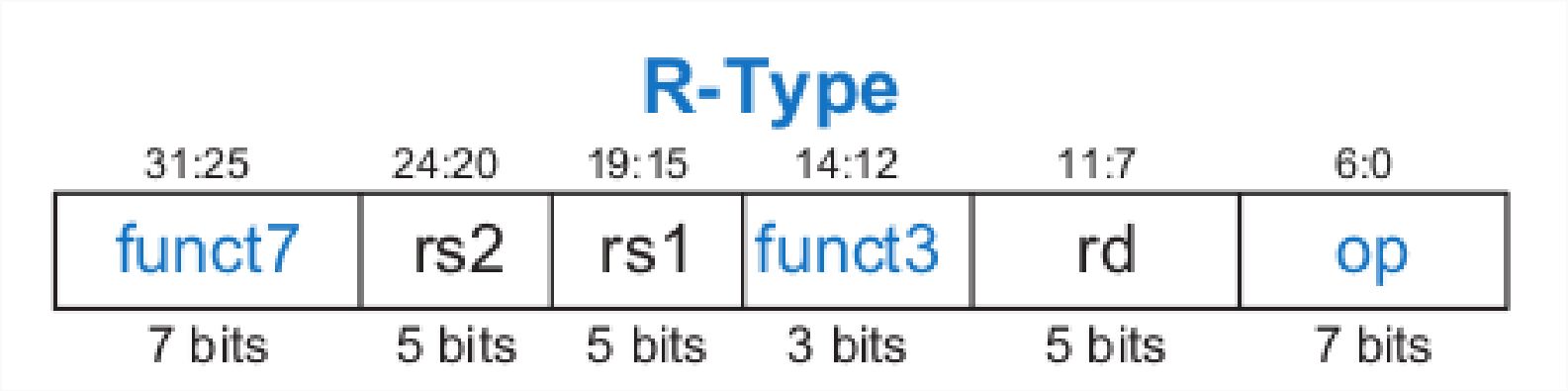
Esta traducción indica como podemos implementar un ciclo `while` con un salto condicional y uno incondicional.

Intervalo

Lenguaje de máquina

El lenguaje ensamblador es un lenguaje de bajo nivel pero los programas escritos en este lenguaje no pueden ser ejecutados por el procesador, es por eso que el código fuente debe ser ensamblado para producir el archivo binario cuyos contenidos pueden ser cargados en memoria y ejecutados.

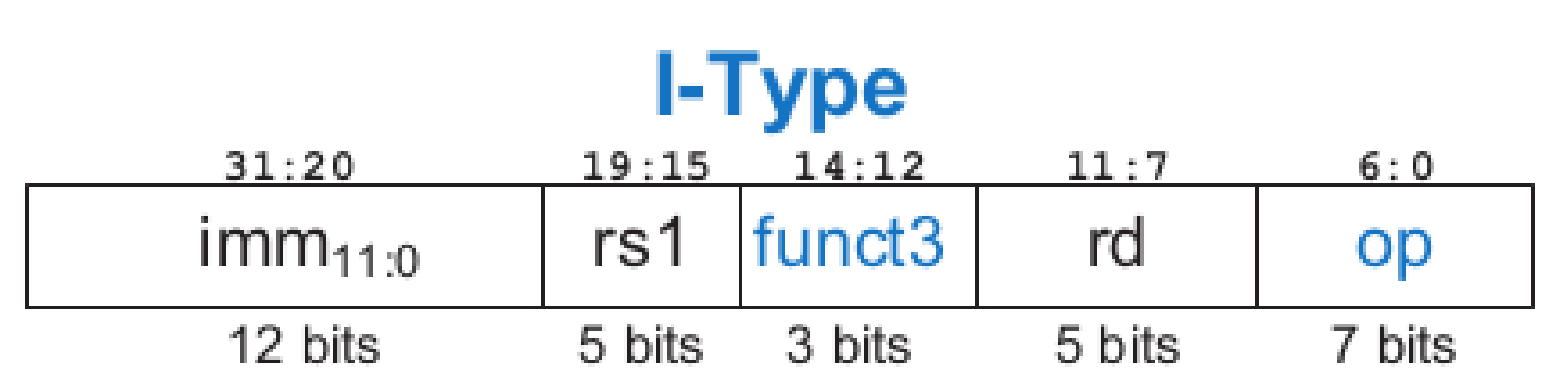
Las instrucciones de tipo `R` utilizan dos registros como operandos fuente (`rs1, rs2`) y uno como operando destino `rd`. El campo `op` junto con `funct7` y `funct3` determinan el tipo de instrucción codificada.



Instrucciones R, ejemplos

Assembly	Field Values					
	funct7	rs2	rs1	funct3	rd	op
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

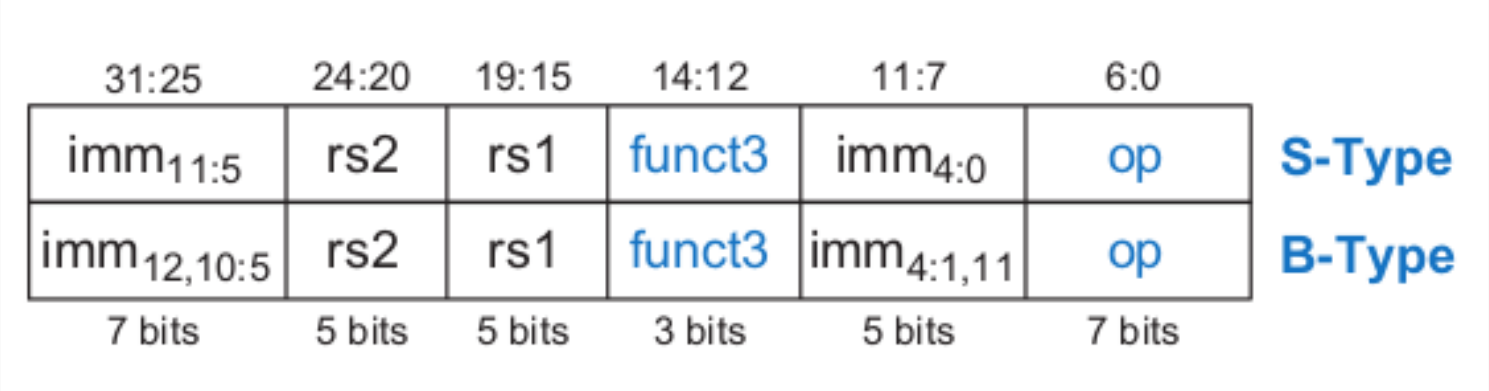
Las instrucciones de tipo `I` utilizan un registros como operando fuente (`rs1`), un inmediato de 12 bits (`imm`) y uno como operando destino `rd`. El campo `op` junto con `funct3` determinan el tipo de instrucción codificada.



Instrucciones I, ejemplos

Assembly	$imm_{11:0}$	rs1	funct3	rd	op
`addi s0, s1, 12` `addi x8, x9, 12`	12	9	0	8	19
`addi s2, t1, -14` `addi x18,x6, -14`	-14	6	0	18	19
`lw t2, -6(s3)` `lw x7, -6(x19)`	-6	19	2	7	3
`lb s4, 0x1F(s4)` `lb x20,0x1F(x20)`	0x1F	20	0	20	3
`slli s2, s7, 5` `slli x18, x23, 5`	5	23	1	18	19
`srai t1, t2, 29` `srai x6, x7, 29`	(upper 7 bits = 32) 29	7	5	6	19
	12 bits	5 bits	3 bits	5 bits	7 bits

Las instrucciones de carga (`S`) y de saltos condicionales (`B`) se codifican como se indica a continuación. Ambos formatos codifican un inmediato en la instrucción, en el caso de las isntrucciones de carga es de 12 bits, en los saltos condicionales es de 13 bits y expresa el desplazamiento en complemento a 2 al que se debe saltar en relación al valor actual del `PC`. Este desplazamiento (`offset`) siempre se desplaza una posición a izquierda antes de sumarlo al `PC` ya que se encuentra siempre en posiciones pares.



Instrucciones S, ejemplos

Assembly	Field Values					
	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
`sw t2, -6(s3)` `sw x7, -6(x19)`	1111 111	7	19	2	11010	35
`sh s4, 23(t0)` `sh x20,23(x5)`	0000 000	20	5	1	10111	35
`sb t5, 0x2D(zero)` `sb x30,0x2D(x0)`	0000 001	30	0	0	01101	35
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Instrucciones B, ejemplos


```
#Address   # RISC-V Assembly
0x70       beq  s0, t5, L1   1
0x74       add  s1, s2, s3   2
0x78       sub  s5, s6, s7   3
0x7C       lw   t0, 0(s1)    4
0x80  L1:  addi s1, s1, -15
```

`L1` is 4 instructions (i.e., **16 bytes**) past `beq`

$imm_{12:0}$ = 16	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly

```
beq s0, t5,  L1
beq x8, x30, 16
```

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

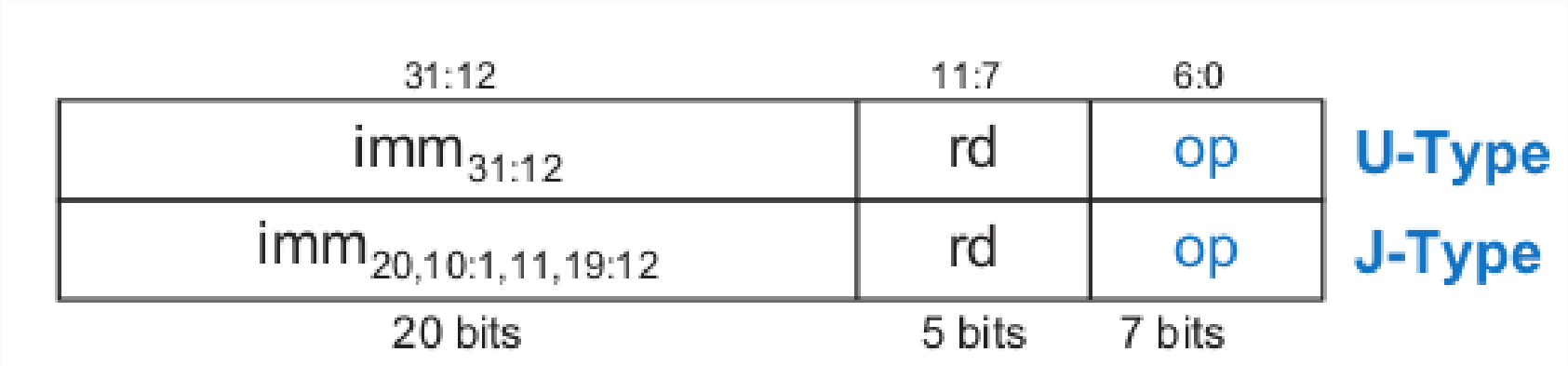
Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x01E40863)

Instrucciones U/J

Las instrucciones de inmediato superior (`U`) y de saltos incondicionales (`J`) se codifican como se indica a continuación. Ambos formatos codifican un inmediato en la instrucción, en el caso de las instrucciones de inmediato superior es de 20 bits, en los saltos incondicionales es de 21 bits y expresa el valor de los 21 bits más altos de la dirección a la que se debe saltar en relación al valor actual del `PC`. Este desplazamiento (`offset`) siempre se desplaza una posición a izquierda antes de sumarlo al `PC` ya que se encuentra siempre en posiciones pares.



Instrucciones U, ejemplos

Assembly	Field Values		
	$imm_{31:12}$	rd	op
`lui s5, 0x8CDEF` `lui x21,0x8CDEF`	0x8CDEF	21	55
	20 bits	5 bits	7 bits

Instrucciones J, ejemplos


```
# Address          RISC-V Assembly
0x0000540C         jal ra, func1
0x00005410         add s1, s2, s3
...                ...

0x000ABC04  func1: add s4, s5, s8
...                ...
```

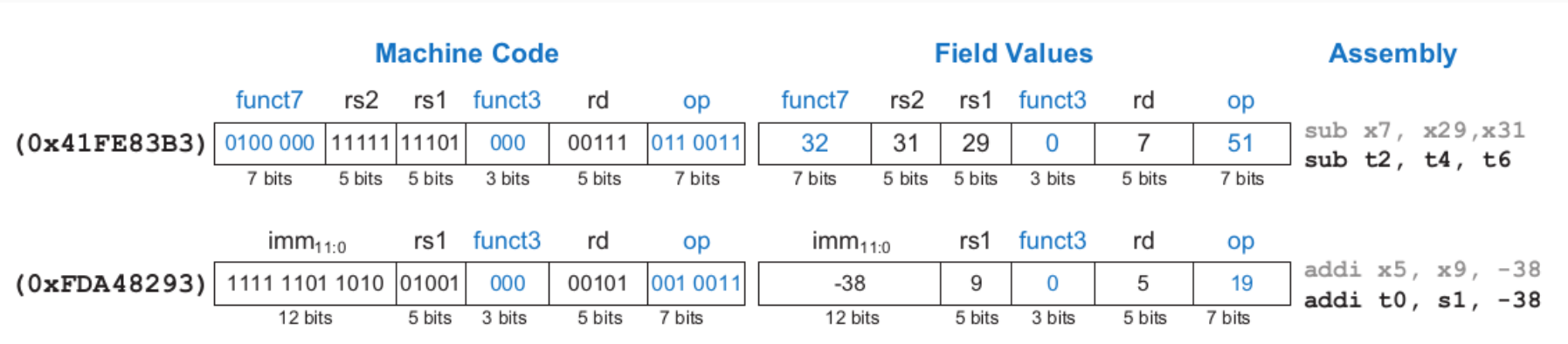
`func1` is **0xA67F8 bytes** past `jal`

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values			Machine Code			
	$imm_{20,10:1,11,19:12}$	rd	op	$imm_{20,10:1,11,19:12}$	rd	op	
`jal ra, func1` `jal x1, 0xA67F8`	0111 1111 1000 1010 0110	1	111	0111 1111 1000 1010 0110	00001	110 1111	(0x7F8A60EF)
	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

Es importante comprender el formato con el que se codifican las instrucciones al traducirlas al lenguaje máquina para poder realizar tanto la codificación como la decodificación de las mismas en caso de ser necesario.

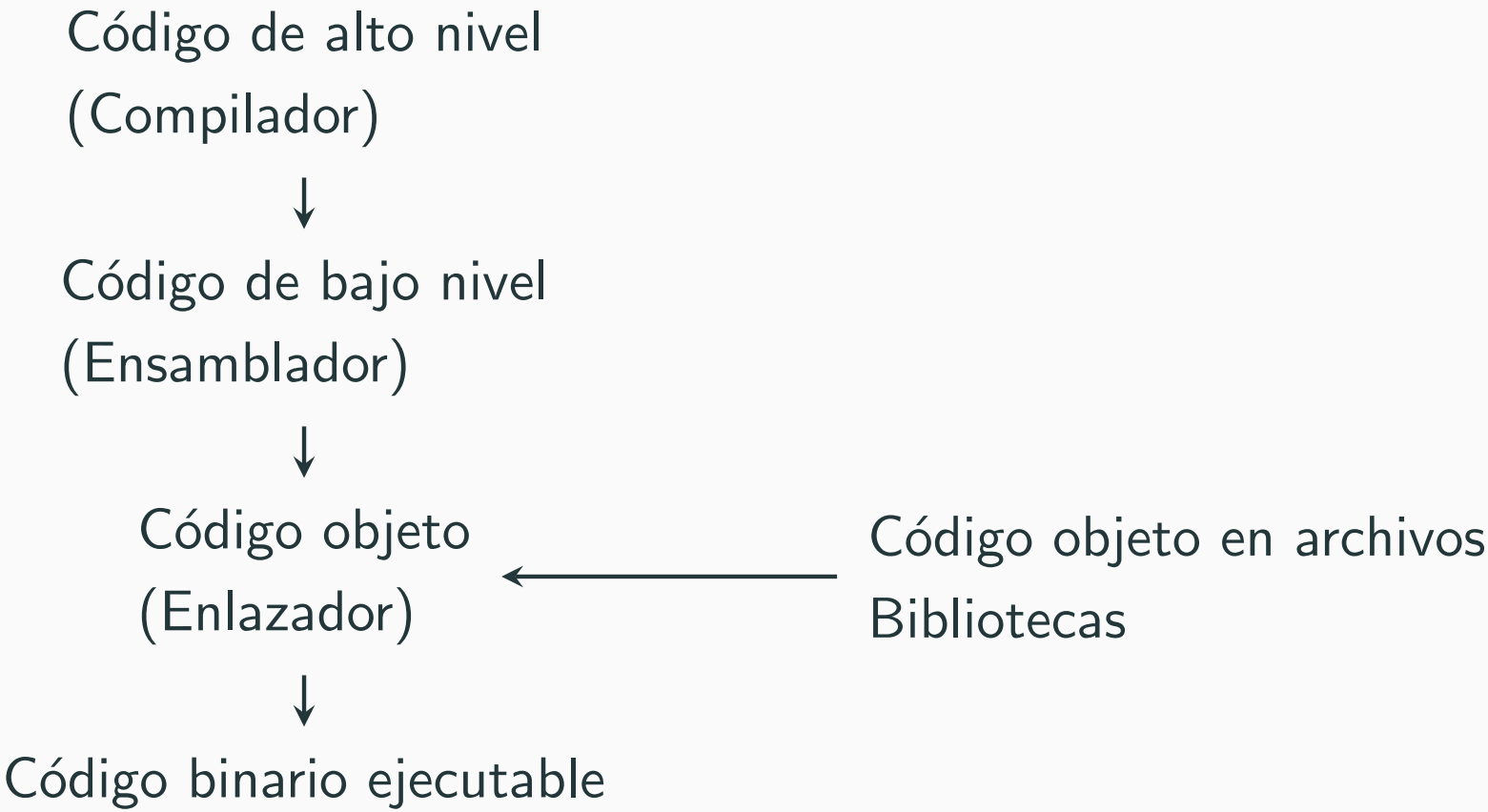
Ejemplo de decodificación



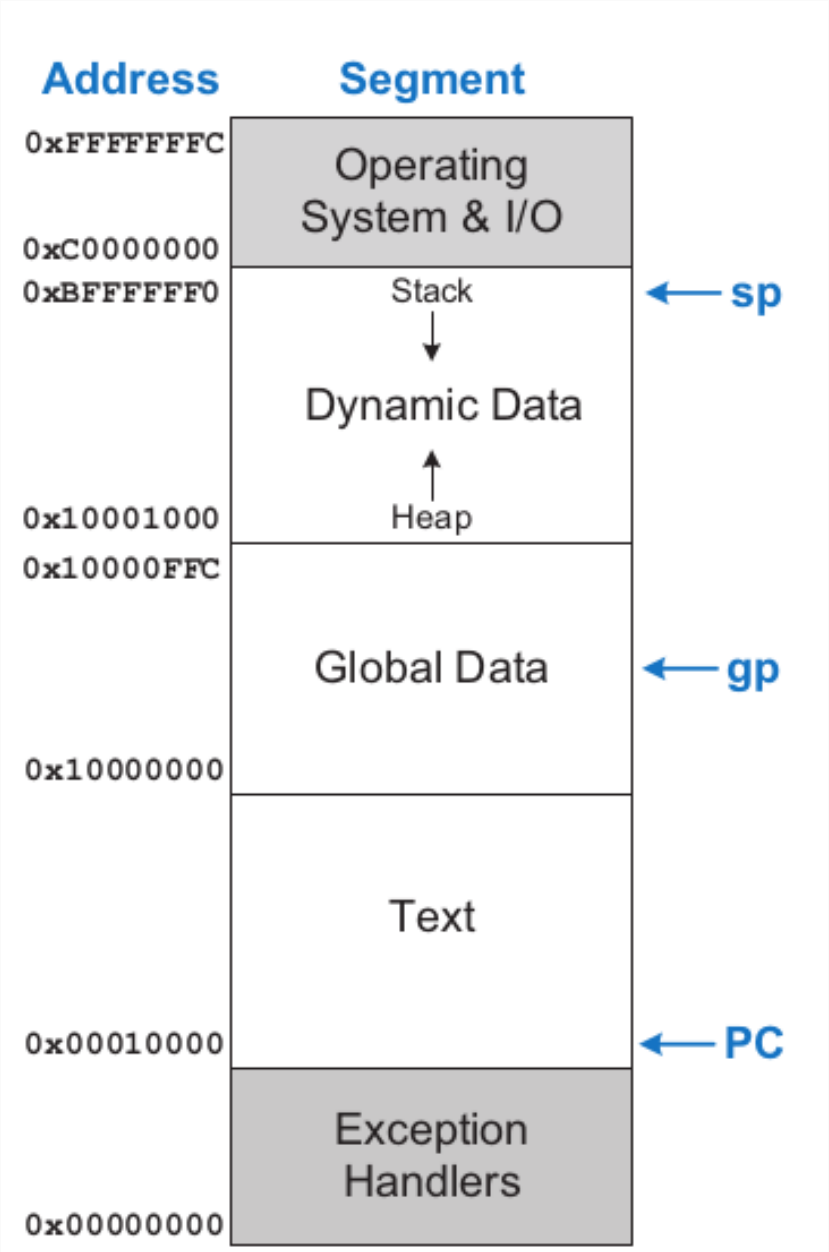
Compilación, ensamblado y ejecución

Habíamos presentado anteriormente el esquema de traducciones que nos permite llegar de código de alto nivel a un formato binario que pueda cargarse en la memoria principal para poder ejecutar, vamos a repasarlo y a presentar el mapa de memoria.

Repaso de la cadena de compilación



El mapa de memoria



El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de entrada y salida.

El mapa de memoria

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de entrada y salida.
- Luego se encuentra la región de datos dinámicos donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).

El mapa de memoria

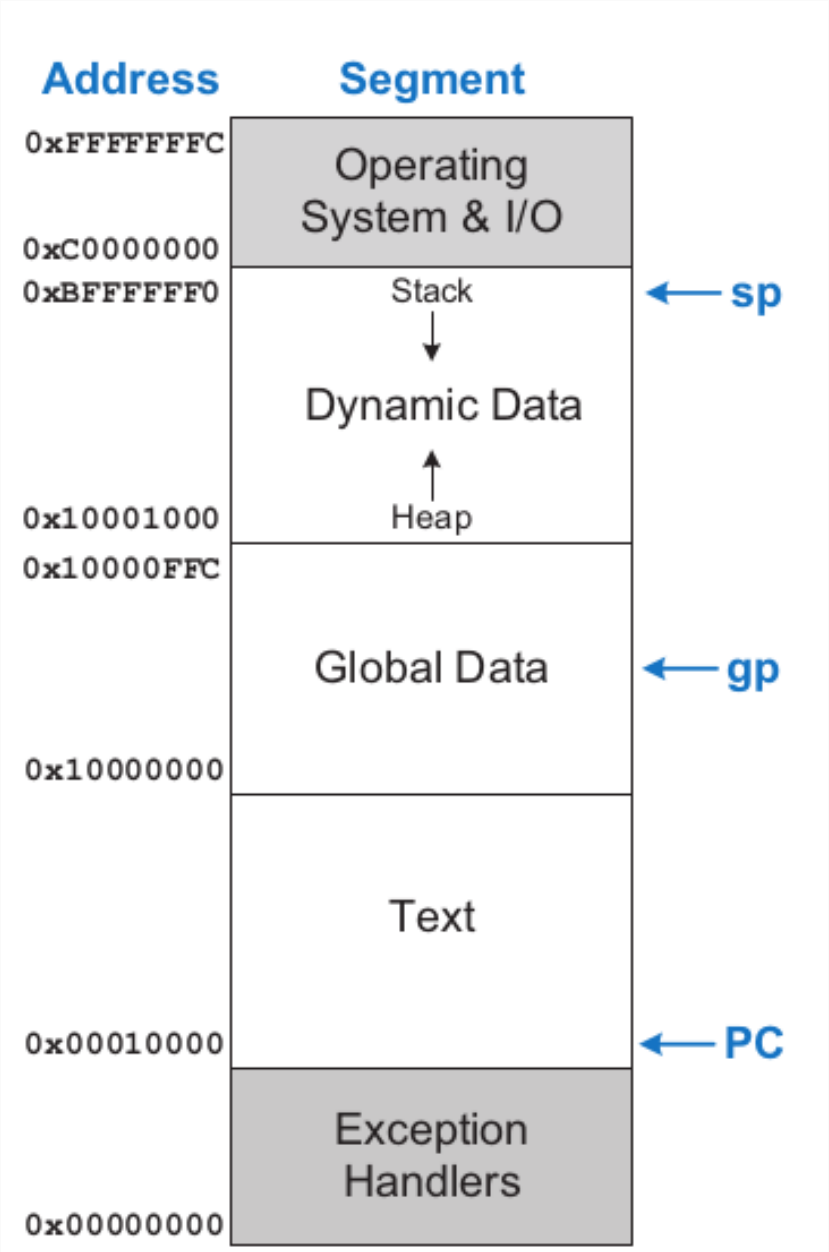
El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de entrada y salida.
- Luego se encuentra la región de datos dinámicos donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).
- Luego se encuentran los datos globales(`.global`), donde se almacenan variables y constantes globales.

El mapa de memoria divide a la memoria principal según su uso:

- La región más alta se reserva para comunicacíon de entrada y salida.
- Luego se encuentra la región de datos dinámicos donde en las direcciones altas vamos a encontrar la pila (`stack`)y en las direcciones bajas el `heap` que es la estructura que permite a un programa hacer un pedido explícito de memoria (`malloc`, `free`, sin usar el `stack`).
- Luego se encuentran los datos globales(`.global`), donde se almacenan variables y constantes globales.
- Y luego el texto(`.text`), que es donde se encuentra el contenido binario de nuestro programa.

El mapa de memoria



Existen algunas directivas, que no son realmente instrucciones, sino indicaciones para que el progama ensamblador puede reservar memoria, definir constantes y ubicar el programa y los datos según las secciones definidas en el mapa de memoria, a continuación presentamos algunas.

Directivas de ensamblado

Assembler Directive	Description
`.text`	Text section
`.data`	Global data section
`.bss`	Global data initialized to 0
`.section .foo`	Section named `.foo`
`.align N`	Align next data/instruction on 2^N-byte boundary
`.balign N`	Align next data/instruction on N-byte boundary
`.globl sym`	Label `sym` is global
`.string "str"`	Store string "`str`" in memory
`.word w1, w2, ..., wN`	Store N 32-bit values in successive memory words
`.byte b1, b2, ..., bN`	Store N 8-bit values in successive memory bytes
`.space N`	Reserve N bytes to store variable
`.equ name, constant`	Define symbol `name` with value `constant`
`.end`	End of assembly code

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).
- Una constante `caracter` de 8 bits (un `byte`).

Veamos por ejemplo cómo se inicializan los datos en la sección de `.data` que va a ubicar la información en lo que el mapa se muestra como `Global Data`, arriba del código (`.text`), mostramos:

- Una constante `largo` de 32 bits (una palabra o `word`).
- Una constante `caracter` de 8 bits (un `byte`).
- Un arreglo `arreglo` de palabras de 32 bits.

Inicializando datos


```
1          .section .data
2          # A partir de este punto comienzan los datos
3          largo: .word 0x4
4          caracter: .byte 10
5          arreglo: .word 0xc, 0x34d, 0x1, 0x0
6          .setion .text
7          # A partir de este punto comienzan las
               instrucciones
```

Al igual que con los saltos en el programa, las etiquetas que declaran constantes van a indicar la posición de memoria desde donde debe cargarse el dato.

Cierre

Clase de hoy

Hoy vimos:

- Definición de arquitecturas.

Clase de hoy

Hoy vimos:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.

Clase de hoy

Hoy vimos:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.

Clase de hoy

Hoy vimos:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.
- Codificación de instrucciones, compilación y ensamblado.

¡Eso es todo por hoy!

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Arquitectura 2/2

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Clase anterior

En la clase anterior vimos:

- Definición de arquitecturas.

Clase anterior

En la clase anterior vimos:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.

Clase anterior

En la clase anterior vimos:

- Definición de arquitecturas.
- El lenguaje ensamblador de `RISC V`.
- Lenguaje máquina y programa almacenado en memoria.

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.

Clase de hoy

Hoy vamos a ver:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.
- Uso de la pila.

¿Qué constituye una arquitectura?

¿Qué constituye una arquitectura?

- El conjunto de instrucciones.

¿Qué constituye una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.

¿Qué constituye una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.
- La forma de acceder a la memoria.

Repaso

¿Qué constituye una arquitectura?

- El conjunto de instrucciones.
- El conjunto de registros.
- La forma de acceder a la memoria.

¿Qué es una instrucción, un registro o una memoria?

Clase anterior

Volvamos a nuestro programa de referencia y al ejemplo de control de ejecución.

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Escribiendo un loop

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`  addi s0, zero, 1`
	`  add s1, zero, zero`
	`#t0=128`
	`  addi t0, zero, 128`
	`while:`
	`  beq s0, t0, fin`
	`  slli s0, s0, 1 #pow=pow*2`
	`  addi s1, s1, 1 #x+=1`
	`  j while`
	`fin:`

Escribiendo un loop

C	RISC V

`// calcula el valor de x`
`// tal que 2 a la x es 128`
`int pow = 1;`
`int x = 0;`
 	
`while(pow != 128){`
`  pow = pow * 2;`
`  x = x + 1;`
`}`
	`#s0=pow, s1=x`
	`  addi s0, zero, 1`
	`  add s1, zero, zero`
	`#t0=128`
	`  addi t0, zero, 128`
	`while:`
	`  beq s0, t0, fin`
	`  slli s0, s0, 1 #pow=pow*2`
	`  addi s1, s1, 1 #x+=1`
	`  j while`
	`fin:`

Esta traducción indica como podemos implementar un ciclo `while` con un salto condicional y uno incondicional.

Manejo de estructuras

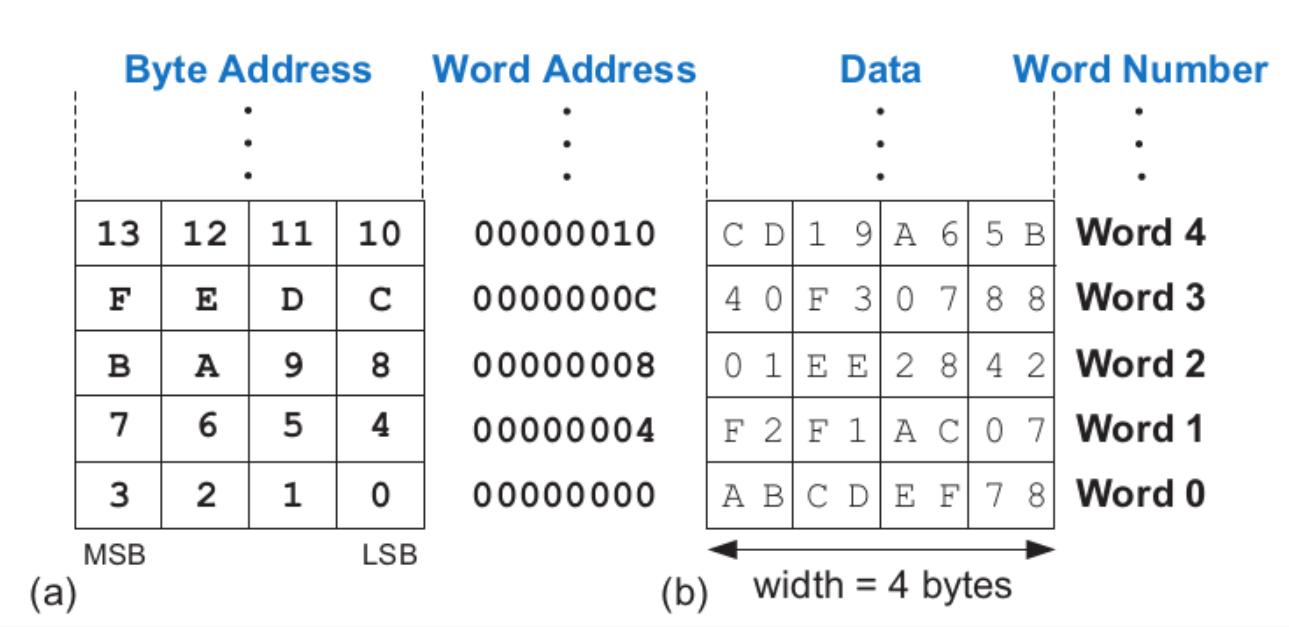
Direcciones de memoria

Recordemos cómo se realiza el acceso a datos en memoria.

Direcciones de memoria

`RISC V` permite acceder a la memoria con índices (direcciones) de 32 bits, o sea 4.294.967.296 índices posibles. Pero cabe notar que el índice apunta a un byte en particular, o sea, a uno de los cuatro bytes de la palabra, de modo que entre una palabra de 32 bits y otra, los índices avanzan en cuatro unidades. Podemos indicar que la lectura o escritura se hará en base a un byte en particular.

Accesos a memoria (direcciones)



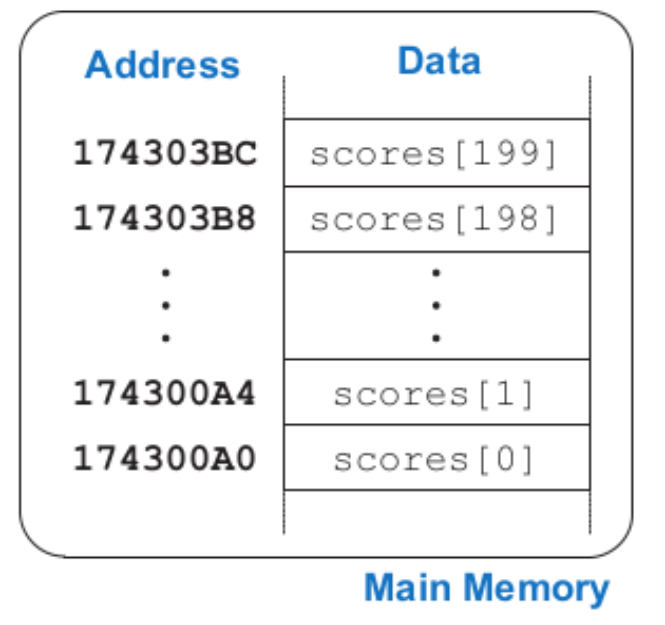
A la izquierda (a), vemos los índices de memoria (`byte address`) representados de derecha a izquierda, donde a la derecha vemos el byte menos significativo (LSB) y a la derecha el byte más significativo de la palabra (MSB). La dirección de palabra (word address) corresponde al índice del byte menos significativo de ésta.

Estructura de los arreglos

Los arreglos son estructuras que ubican elementos del mismo tamaño y tipo de forma consecutiva en la memoria del procesador. En un lenguaje de alto nivel, la forma de acceder a un elemento es a partir de una dirección base y la posición en el arreglo, a la que llamamos su índice. La forma de acceder en lenguaje ensamblador es calculando el desplazamiento desde la dirección del comienzo del arreglo hasta la dirección en la que se encuentra el elemento.

Acceso a un elemento del arreglo

En este ejemplo el arreglo `scores` contiene 200 elementos de 32 bits y comienza en la dirección `0x174300A0`. La forma de acceder al i-ésimo elemento es cargando el dato que se encuentra en `base + tamaño * índice`, en este caso, si queremos acceder al elemento 199 sería `0x174300A0 + 4 * 198 = 0x174303B8`.



Acceso a un elemento del arreglo

En el siguiente ejemplo se incrementa el valor de cada elemento del arreglo en 10.

Incrementando los valores de un arreglo

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

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

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

```
#s0=dir. scores, s1=i
  addi s1, zero, 0
  addi t2, zero, 200
for:
  bge s1, t2, fin
  slli t0, s1, 2
  add t0, t0, s0
  lw t1, 0(t0)
  addi t1, t1, 10
  sw t1, 0(t0)
  addi s1, s1, 1
  j for
fin:
```

Interfaz binaria de aplicación

Llamadas a función


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

¿Cómo escribimos funciones en `RISC V` pasando un número arbitrario de parámetros y devolviendo un parámetro de retorno?

Definiendo un contrato


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

Recordemos que contamos con una memoria principal direccionable y un número acotado de registros y con esto vamos a tener que definir un contrato que indique de qué forma se realizan las llamadas a función.

Interfaz binaria de aplicación

A este contrato que indica de qué forma vamos a realizar las llamadas a función para una arquitectura en particular lo llamamos interfaz binaria de aplicación. Define un conjunto de reglas que tanto quienes programan en ensamblador `RISC V` como el programa de compilación de un lenguaje de alto nivel a ensamblador `RISC V` deben respetar para poder interactuar con otros programas, llamadas a sistema y bibliotecas compartidas.

Ejemplo con argumentos

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C

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

RISC V

```
main:#s7=y
    addi a0, zero, 2
    addi a1, zero, 3
    addi a2, zero, 4
    addi a3, zero, 5
    jal dif_sumas
    add s7, a0, zero
dif_sums:#s3=result
    add t0, a0, a1
    add t1, a2, a3
    sub s3, t0, t1
    add a0, s3, zero
    jr ra
```

Llamadas a función, argumentos

En un lenguaje de alto nivel los programas se dividen en funciones que pueden llamarse unas o otras. Para implementar esta funcionalidad se debe decidir de qué manera una función puede identificar a otra y cómo se enviarán los parámetros de entrada y de salida. Los parámetros de entrada serán llamados argumentos y los de salida valor de retorno.

Llamadas a función, jal

En `RISC V` la función llamadora puede utilizar los registros `a0` hasta `a7` para enviar argumentos y luego la función llamada utiliza `a0` para copiar el valor de retorno. A la hora de invocar la ejecución de una función la función llamadora debe almacenar el `PC` en `ra`. Esto se consigue utilizando la instrucción `jal ra, foo`, donde `foo` es la función llamada.

Llamadas a función, preservando estado

La función llamada no debe interferir con el estado de la función llamadora, debido a esto debe respetar los valores de los registros guardados (`s0` a `s11`) y el registro de la dirección de retorno (`ra`), que indica cómo retornar la ejecución a la función llamadora. También debe mantenerse invariante la porción de memoria (stack) correspondiente a función llamadora.

Ejemplo de llamada

C	RISC V

`int main(){`
`  simple();`
`  ...`
`}`
 	
`void simple(){`
`  return;`
`}`
	`0x00000300 main: jal ra, simple`
	`0x00000304 ...`
	`... ...`
	`0x0000051c simple: jr ra`

Ejemplo de llamada

C	RISC V

`int main(){`
`  simple();`
`  ...`
`}`
 	
`void simple(){`
`  return;`
`}`
	`0x00000300 main: jal ra, simple`
	`0x00000304 ...`
	`... ...`
	`0x0000051c simple: jr ra`

Un ejemplo de llamada a `simple` y un retorno con un salto incondicional al registro de la direccón de retorno `jr ra`.

Ejemplo con argumentos

A continuación presentamos un ejemplo que involucra argumentos.

Ejemplo con argumentos

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

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

```
main:#s7=y
 addi a0, zero, 2
 addi a1, zero, 3
 addi a2, zero, 4
 addi a3, zero, 5
 jal dif_sumas
 add s7, a0, zero
dif_sums:#s3=result
 add t0, a0, a1
 add t1, a2, a3
 sub s3, t0, t1
 add a0, s3, zero
 jr ra
```

Uso de la pila y el stack pointer

¿Qué es la pila?

La pila es:

¿Qué es la pila?

La pila es:

- Una región de la memoria definida entre una dirección de memoria alta y la dirección indicada en el registro stack pointer (sp).

¿Qué es la pila?

La pila es:

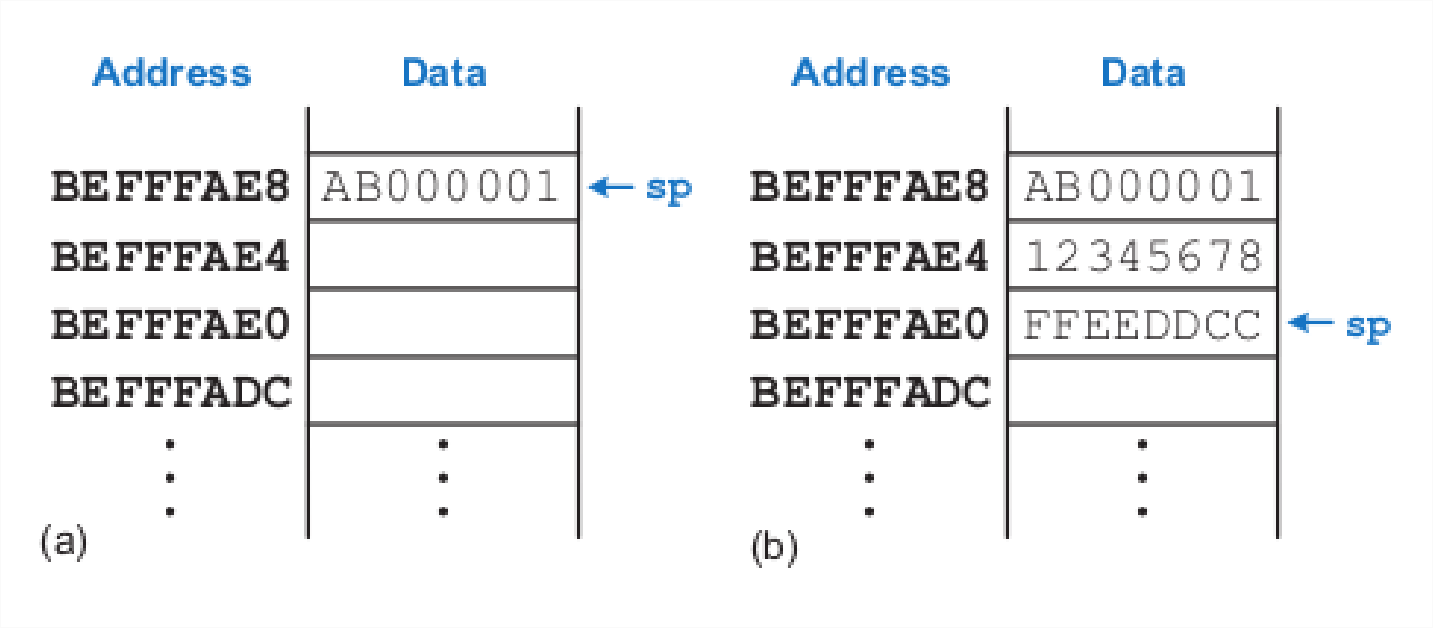
- Una región de la memoria definida entre una dirección de memoria alta y la dirección indicada en el registro stack pointer (sp).
- Un mecanismo para almacenar valores temporarios con una semántica de LIFO (el último elemento almacenado es el primero al que accedemos).

La pila (stack)

La semántica de uso es a través de operación de agregado (`push`) y retiro (`pop`) de un elemento siempre al tope de la pila. La pila suele comenzar en las direcciones altas de la memoria y va tomando (con cada push) las direcciones inmediatamente más bajas. Por eso se suele decir que la pila crece hacia abajo.

Stack pointer (sp)

Al igual que en muchas otras arquitecturas, `RISC V` propone el uso de uno de sus registros, `sp` (stack pointer), para indicar la dirección de tope de pila. En este ejemplo vemos como se actualiza la pila (y el stack pointer) luego de agegar dos palabras de 32 bits (`0x12345678` y `0xFFEEDDCC`) cambiando el `sp` de `0xBEFFFAE8` a `0xBEFFFAE0` (`sp` apunta al último elemento cargado).



La pila (stack)

Parte de la convención de `RISC V` (interfaz binaria de aplicación) indica que el stack pointer debe siempre estar alineado a 16 bytes, esto significa que su valor debe siempre cumplir con la congruencia

$$sp \,\%16 == 0$$

.

La pila (stack)

Veamos un breve ejemplo del uso de la pila y la alineación del `stack pointer`.

```
# apilando (push) en una llamada a funcion
foo: addi sp, sp, -16 # restamos 16 aunque
    precisemos 8 bytes
sw a0, 4(sp) #guarda a0
sw ra, 0(sp) #guarda ra
# cuerpo de la funcion
# desapilando (pop)
lw a0, 4(sp) #restaura a0
lw ra, 0(sp) #restaura ra
addi sp, sp, 16 # restaura el valor de stack
    pointer
```

Habíamos dicho que al llamar a una función había un acuerdo entre la función llamadora (la que inicia la llamada) y la función llamada (la que la recibe), donde se preservaba parte del estado del procesador entre el llamado y el retorno.

Reglas para llamar funciones

Vamos a presentar una serie de reglas que deberían asegurar que cada función llamadora entrega y cada función llamada recibe a los elementos de memoria del procesador (registros, memoria general y pila) en un estado conocido.

Reglas de llamada

Preserved (*callee*-saved)	Nonpreserved (*caller*-saved)
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp`	
Stack above the stack pointer	Stack below the stack pointer

Estado del procesador entre llamadas

Reglas de preservación de estado:

Reglas de preservación de estado:

- Regla para la llamadora: Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).

Estado del procesador entre llamadas

Reglas de preservación de estado:

- Regla para la llamadora: Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).
- Regla para la llamada: Si va a utilizar los registros permanentes (`s0-s11, ra`) debe guardarlos al comenzar y restaurarlos antes de retornar.

Estado del procesador entre llamadas

Reglas de preservación de estado:

- Regla para la llamadora: Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (`t0-t6, a0-a7`).
- Regla para la llamada: Si va a utilizar los registros permanentes (`s0-s11, ra`) debe guardarlos al comenzar y restaurarlos antes de retornar.

Para esto podemos utilizar la pila.

Ejemplo recursivo

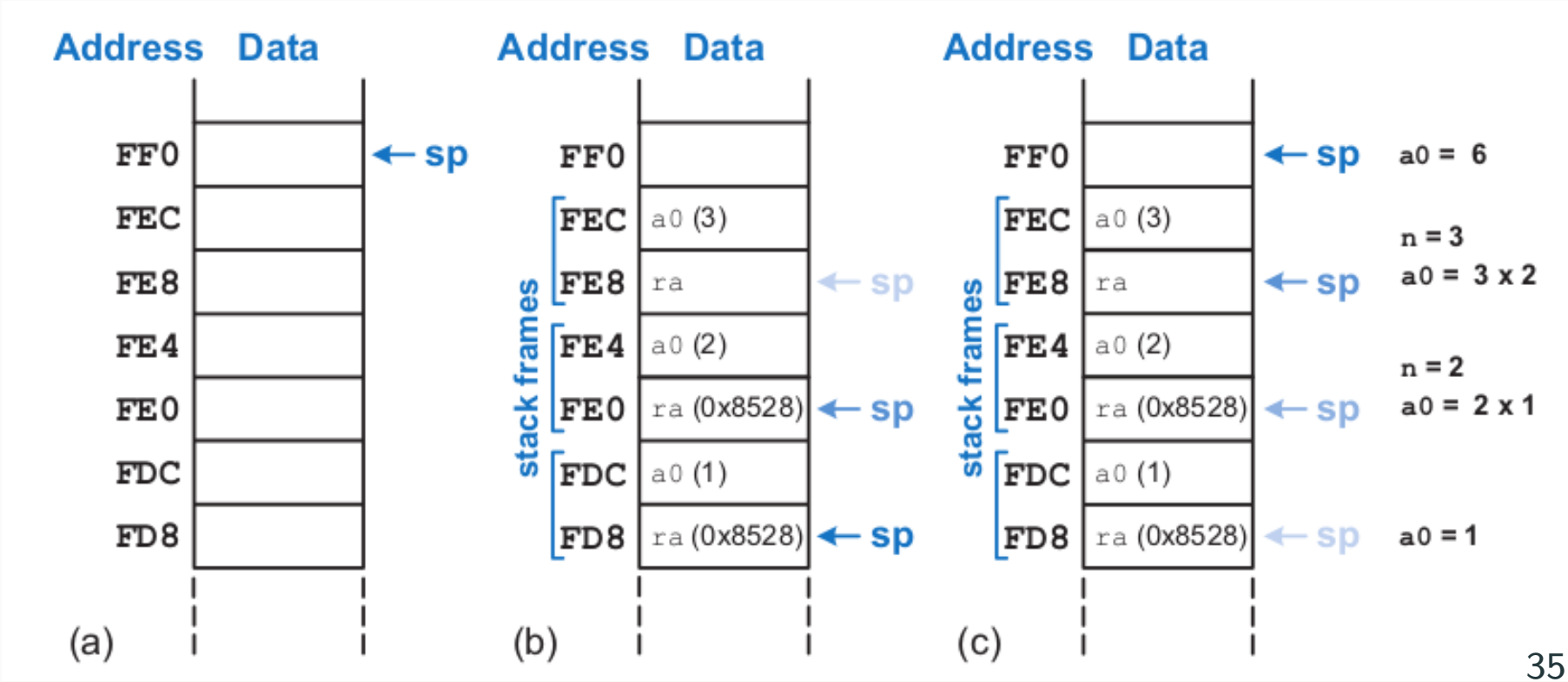
DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

C	RISC V

`int factorial(int n){`
`  if(n <= 1){`
`  return 1;`
`  }else{`
`  return`
`  (n*factorial(n-1));`
`  }`
`}`
	`factorial:addi sp, sp, -16`
	`  sw a0, 4(sp) #guarda a0`
	`  sw ra, 0(sp) #guarda ra`
	`  addi t0, zero, 1`
	`  bgt a0, t0, else`
	`  addi a0, zero, 1`
	`  addi sp, sp, 16`
	`  jr ra`
	`else: addi a0, a0, -1`
	`  jal factorial`
	`  lw t1, 4(sp)`
	`  lw ra, 0(sp)`
	`  addi sp, sp, 16`
	`  mul a0, t1, a0`
	`  jr ra`

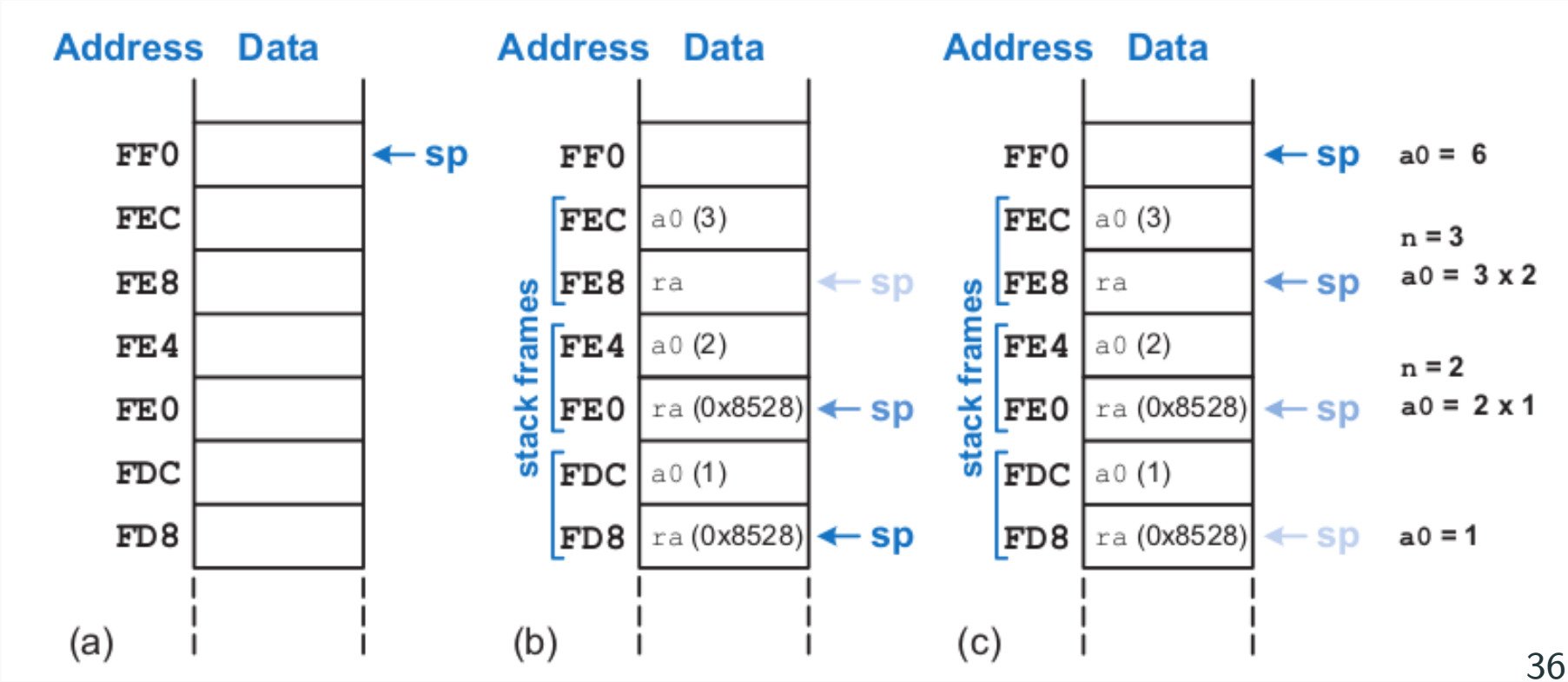
Stack pointer (sp)

Podemos ver como cada llamada recursiva utiliza una porción de la pila para preservar su estado y así cumplir con las reglas antes mencionadas, al espacio de la pila utilizado por la llamada en cuestión lo llamamos marco de pila o stack frame.



Stack pointer (sp)

Aquí la columna a muestra las posiciones altas de la memoria antes de la primer llamada, la columna b muestra el estado luego de tres llamadas recursivas y la columna c indica cómo se actualizan los valores de a0 al ir regresando de cada llamada (`jr ra`).



Pseudoinstrucciones

Pseudoinstrucciones

Algunas de las instrucciones empleadas en el lenguaje ensamblador no son verdaderamente instrucciones, en el sentido de que el procesador no sabe interpretarlas, sino que es el compilador el que se encarga de traducir una de estas así llamadas pseudointstrucción en una instrucción propiamente dicha. El uso de las pseudoinstrucciones se debe a que encapsulan operaciones comunes y convenientes pero que no justifican su inclusión en el set de instrucciones de la arquitectura si queremos mantenerlo acotado.

Ejemplos de pseudoinstrucciones

`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	`jal L1`
`call L5`	`auipc ra,` $imm_{31:12}$ `jalr ra, ra,` $imm_{11:0}$
`ret`	`jalr zero, ra, 0`

Interfaz binaria de aplicación en la práctica

Recomendación a la hora de programar

No intenten memorizar los nombres de todas las instrucciones y su semántica, tengan la documentación mientras escriben o hacen seguimiento de sus programas de lenguaje ensamblador:

- Hoja con lista de registros e instrucciones.
- Reglas de llamada a función.
- Estructura de la memoria.

Recomendación a la hora de programar

Vuelvan a revisar el material de lectura (manuales, clases y apuntes) tantas veces como haga falta. Hacer repetidas lecturas de la documentación es parte de la práctica de la ingeniería.

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li    t0, 0          # acumulador = 0
6      li    t1, 0          # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge   t1, a1, fin    # Si i >= largo, sale del ciclo
9      slli  t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10     add   t2, a0, t2     # Actualiza la dir. de memoria
11     lw    t2, 0(t2)      # De-referencia la dir,
12     add   t0, t0, t2     # Agrega el valor al acumulador
13     addi  t1, t1, 1      # Incrementa el iterador
14     j     ciclo          # Vuelve a comenzar el ciclo
15 fin:
16     mv    a0, t0         # Mueve t0 (acumulador) a a0
17     ret                  # Devuelve valor por a0
```

Revisión del programa de ejemplo

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

Programa de ejemplo en C


```
1 int sumar_arreglo(int a[], int largo) {
2     int acumulador = 0;
3     int i;
4     for (i = 0;i < largo;i++) {
5         acumulador = acumulador + a[i];
6     }
7     return acumulador;
8 }
```

¿Qué podemos entender de la traducción que presentamos antes?

Programa de ejemplo en ASM (RISC V)


```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
       = i
5      li    t0, 0          # acumulador = 0
6      li    t1, 0          # i = 0
7  ciclo:  # Comienzo de ciclo
8      bge   t1, a1, fin    # Si i >= largo, sale del ciclo
9      slli  t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10     add   t2, a0, t2     # Actualiza la dir. de memoria
11     lw    t2, 0(t2)      # De-referencia la dir,
12     add   t0, t0, t2     # Agrega el valor al acumulador
13     addi  t1, t1, 1      # Incrementa el iterador
14     j     ciclo          # Vuelve a comenzar el ciclo
15 fin:
16     mv    a0, t0         # Mueve t0 (acumulador) a a0
17     ret                  # Devuelve valor por a0
```

Cierre

Clase de hoy

Hoy vimos:

- Acceso a memoria y estructuras.

Clase de hoy

Hoy vimos:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.

Clase de hoy

Hoy vimos:

- Acceso a memoria y estructuras.
- Interfaz binaria de aplicación.
- Uso de la pila.

Fin

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Digitales

Microarquitectura

Primer Cuatrimestre 2025

Sistemas Digitales
DC - UBA

Introducción

Clase de hoy

Hoy vamos a ver:

- Definición de **microarquitecturas**.

Clase de hoy

Hoy vamos a ver:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.

Clase de hoy

Hoy vamos a ver:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.
- Procesador de ciclo simple.

Clase de hoy

Hoy vamos a ver:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.
- Procesador de ciclo simple.
- Instrucciones de memoria, registros y saltos condicionales.

Microarquitectura

La microarquitectura se ubica conceptualmente entre la **arquitectura** (aquello que se expone a la persona que programa el sistema) y la lógica combinatoria y secuencial. Implementa el soporte de estado arquitectónico y la lógica de control para actualizar el estado según lo indique la semántica de las instrucciones de la **ISA**.

Estado de la arquitectura

La microarquitectura va encargarse de actualizar el **estado de la arquitectura**, o sea, los **registros de propósito general y el program counter**. Recordemos que nos referimos como estado a los valores almacenados en los elementos de memoria y por ende el estado de la arquitectura se refiere a los elementos de memoria expuestos a la persona que programa el sistema.

Estado de la arquitectura

El procesador puede contener elementos de memoria que constituyen estado por fuera de la arquitectura, registros o banco de memoria utilizados para implementar mecanismos o funciones propios de la arquitectura pero que no son expuestos.

Instrucciones a evaluar

A la hora de justificar las decisiones de diseño e implementación de una microarquitectura para `RISCV`, vamos a enfocarnos en un subconjunto de las instrucciones básicas:

Instrucciones a evaluar

A la hora de justificar las decisiones de diseño e implementación de una microarquitectura para `RISCV`, vamos a enfocarnos en un subconjunto de las instrucciones básicas:

- **Registros:** `add`, `sub`, `and`, `or`, `slt`.

Instrucciones a evaluar

A la hora de justificar las decisiones de diseño e implementación de una microarquitectura para `RISCV`, vamos a enfocarnos en un subconjunto de las instrucciones básicas:

- **Registros:** `add`, `sub`, `and`, `or`, `slt`.
- **Memoria:** `sw`, `lw`.

Instrucciones a evaluar

A la hora de justificar las decisiones de diseño e implementación de una microarquitectura para `RISCV`, vamos a enfocarnos en un subconjunto de las instrucciones básicas:

- **Registros:** `add`, `sub`, `and`, `or`, `slt`.
- **Memoria:** `sw`, `lw`.
- **Salto:** `beq`.

Proceso de diseño - Datapath

Para comenzar con el diseño de un sistema complejo, como es el caso de nuestra microarquitectura, un enfoque posible es comenzar presentando y vinculando a los elementos que realizarán transformaciones con los datos, a esto lo llamaremos **el camino de datos o datapath**.

Proceso de diseño - Unidad de control

Luego decidiremos cómo implementar la unidad que se asegura de coordinar a los elementos del **datapath** para transformar a los datos a partir de la manipulación de sus señales de control, a esto lo llamaremos **unidad de control**.

Sobre los diagramas

En los diagramas se debe observar que:

Sobre los diagramas

En los diagramas se debe observar que:

- Las líneas **gruesas** indican datos de 32 bits.

Sobre los diagramas

En los diagramas se debe observar que:

- Las líneas **gruesas** indican datos de 32 bits.
- Las líneas *delgadas* indican datos de 1 bit.

Sobre los diagramas

En los diagramas se debe observar que:

- Las líneas **gruesas** indican datos de 32 bits.
- Las líneas *delgadas* indican datos de 1 bit.
- Las líneas `intermedias` indican datos de otro tamaño.

Sobre los diagramas

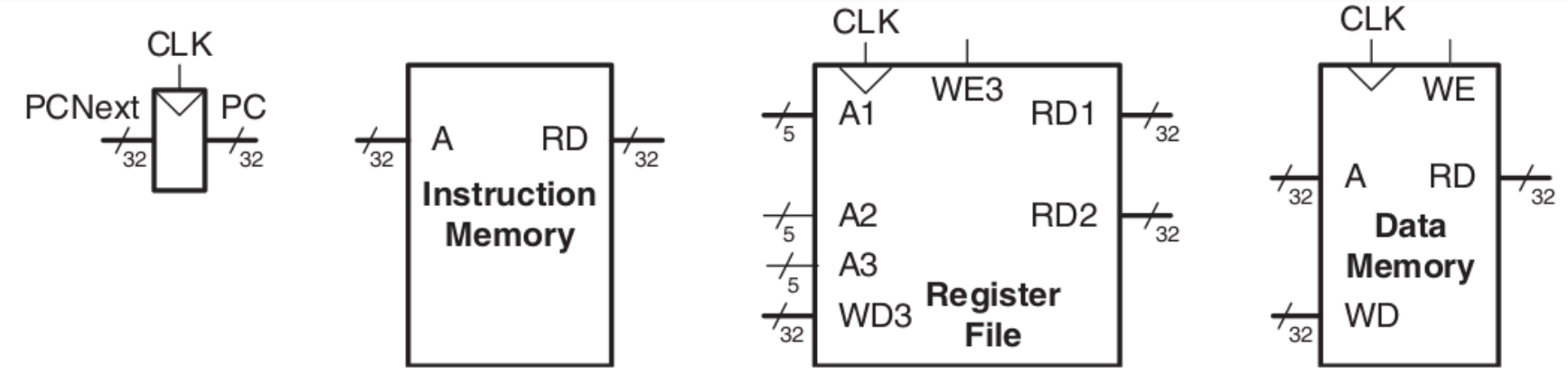
En los diagramas se debe observar que:

- Las líneas **gruesas** indican datos de 32 bits.
- Las líneas *delgadas* indican datos de 1 bit.
- Las líneas `intermedias` indican datos de otro tamaño.
- Las líneas azules indican señales de control.

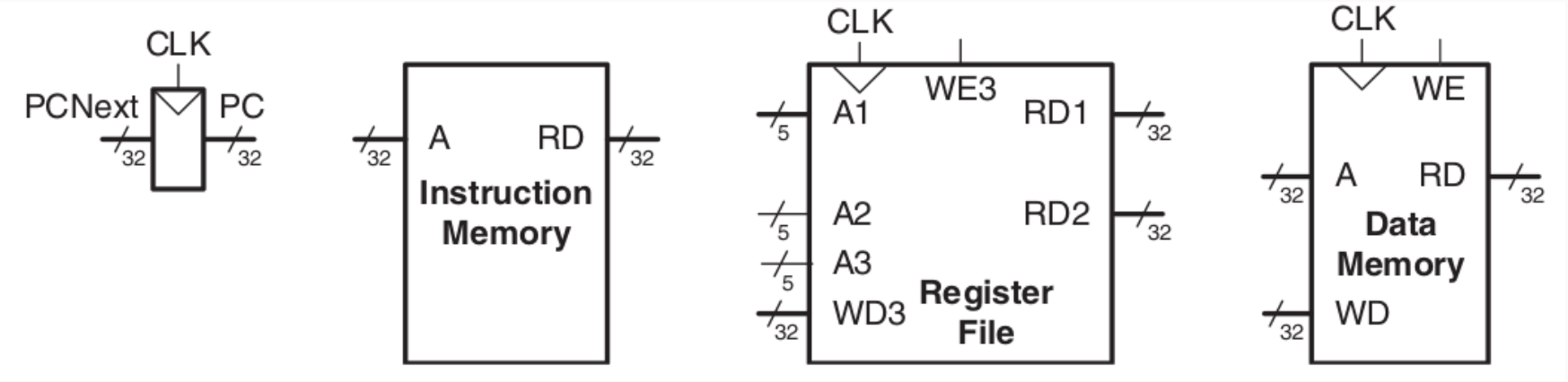
Elementos de memoria

Ahora vamos a presentar y estudiar los elementos de memoria del **datapath**.

Elementos de memoria

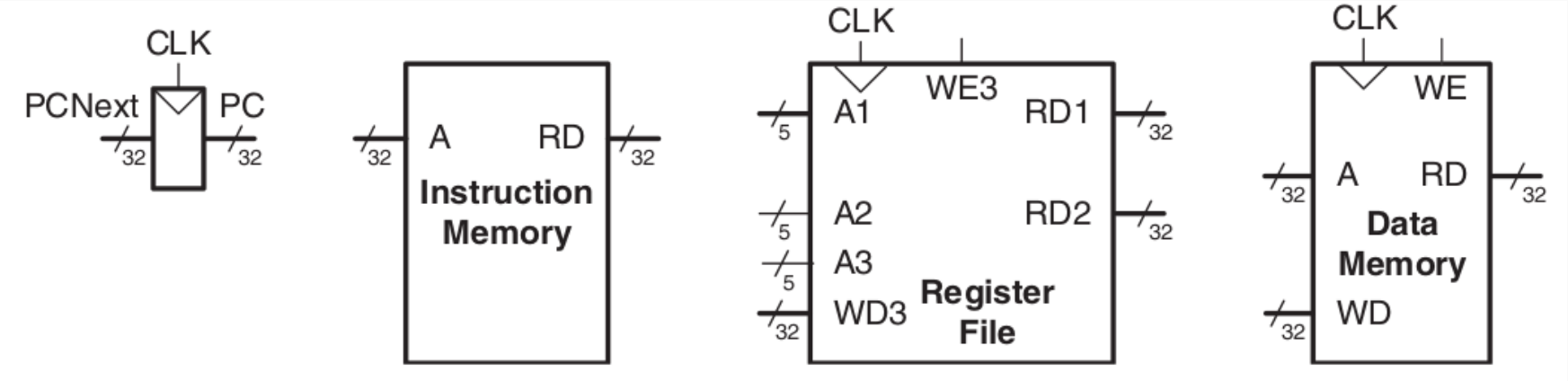


PC



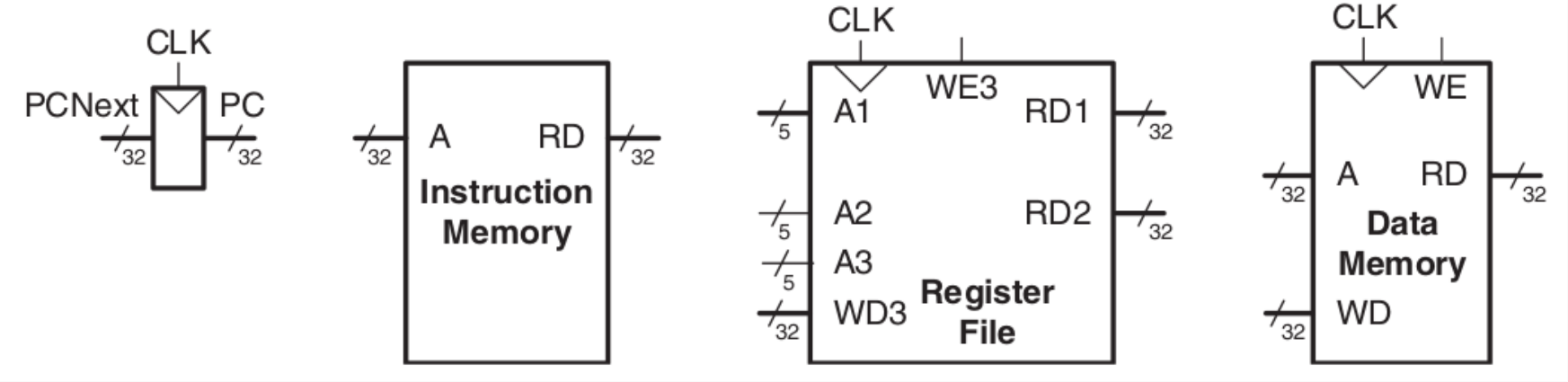
La salida **PC** indica la posición de la instrucción actual, **PCNext** es la entrada que indica la posición de la próxima instrucción.

Memoria de instrucciones



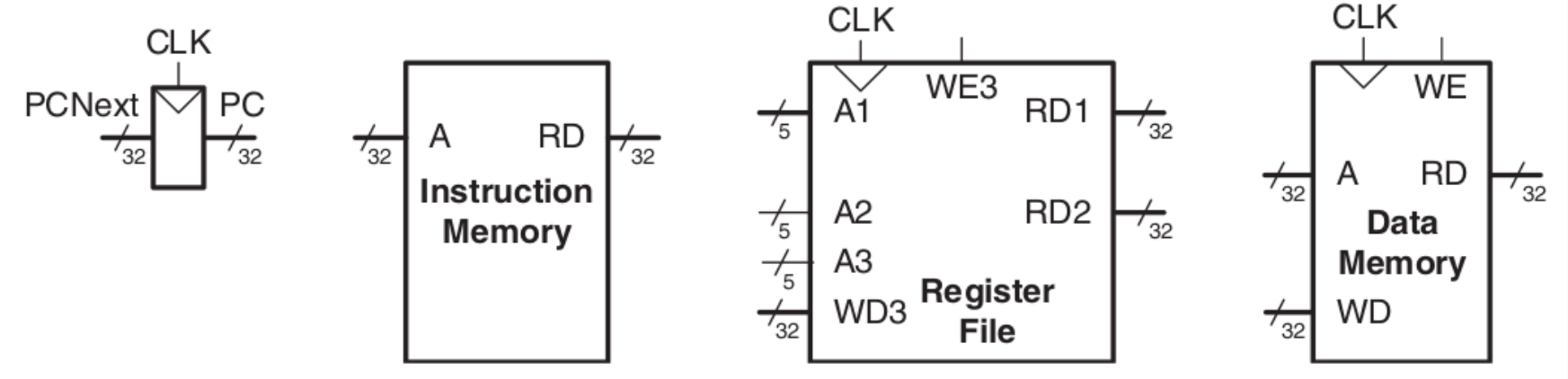
La memoria de instrucciones toma una dirección **A** de 32 bits y vuelca el valor de 32 bits que se encuentra en esa posición por la salida **RD**.

Archivo de registros



El archivo de registros contiene los 32 registros `x0`–`x31` y tiene dos puertos (salidas de datos) de lectura (**RD1** y **RD2**) que vuelcan el valor de los registros en la posiciones indicadas por las entradas **A1** y **A2**. También cuenta con un tercer puerto de escritura (entrada de datos) **WD3** que escribe el dato recibido en la posición indicada por **A3** durante el flanco ascendente de reloj si la señal de control **WE3** se encuentra alta.

Memoria de datos



La memoria de datos toma una dirección **A** de 32 bits y lee el valor de 32 bits que se encuentra en esa posición por la salida **RD** si el valor de la señal de control **WE** se encuentra bajo o escribe el contenido que ingresa por **WD** durante el flanco ascendente del ciclo de clock si se encuentra alto.

Procesador de ciclo simple

Procesador de ciclo simple

Vamos a estudiar una microarquitectura donde **las operaciones se completan durante un único cilo de reloj**, por lo que la duración del ciclo debe ser suficientemente larga como para permitir completar la operación más costosa (las que toma más tiempo). Esto significa que el rendimiento del procesador no será óptimo pero resulta conveniente como ejemplo introductorio a las microarquitecturas.

Programa de ejemplo

Utilizaremos el siguiente programa de ejemplo para justificar la interacción entre el **datapath** y la **unidad de control** e iremos conectando los elementos de memoria y agregando elementos y señales de control a medida que haga falta.

Programa de ejemplo

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Aquí podemos ver la posición de memoria en la que se encuentran las instrucciones codificadas, sus mnemónicos y la división de los distintas partes de cada palabra de 32 bits según su interpretación para la arquitectura.

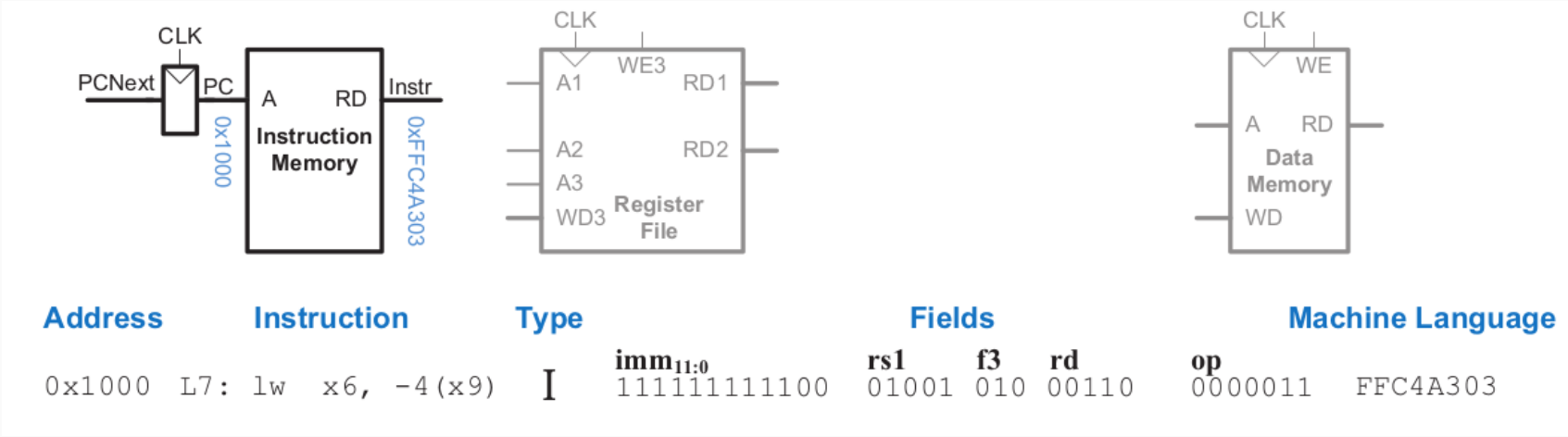
Instrucciones de memoria

Instrucciones de memoria

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

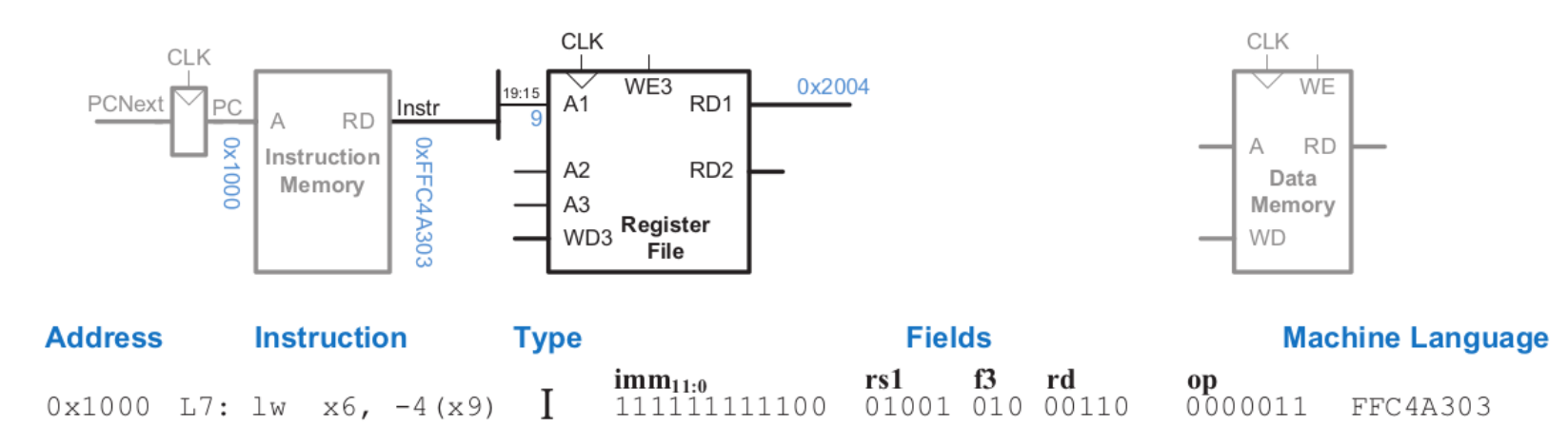
Comenzaremos estudiando los componentes involucrados con la lectura de la instrucción de memoria y la ejecución de la primera instrucción (**fetch** y **lw**).

Fetch



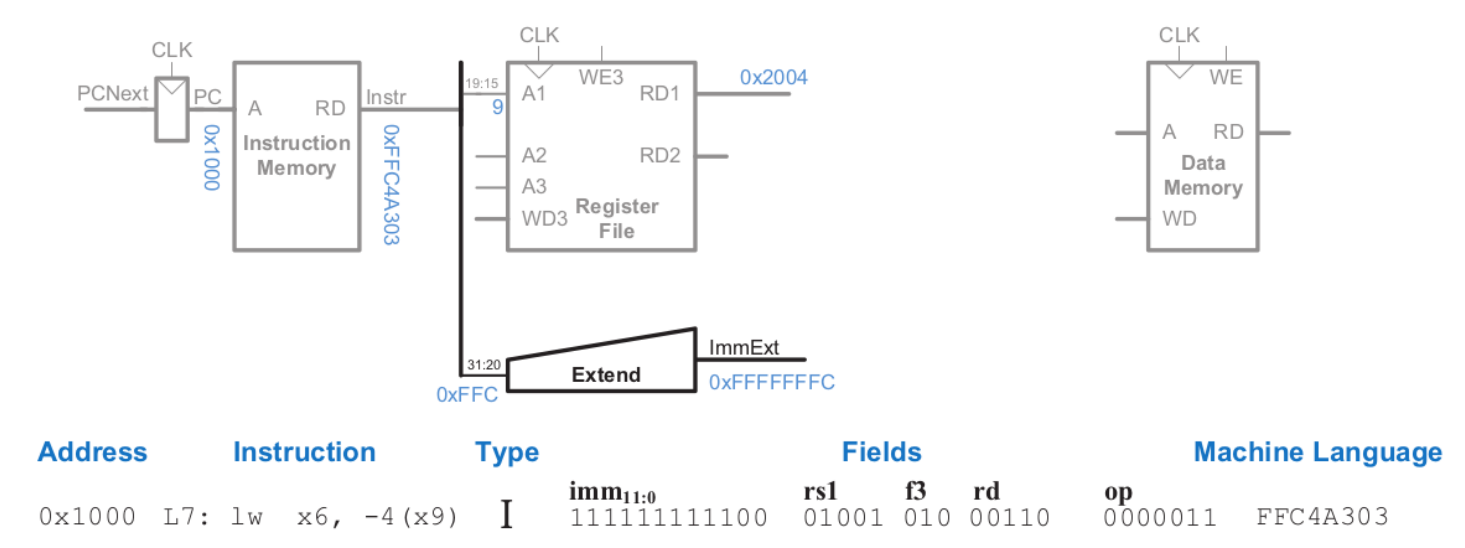
Vemos que la salida de **PC** indica la dirección **A** desde donde leer la instrucción actual de la memoria de instrucciones. La instrucción codificada es la que corresponde a una lectura de memoria (**lw**).

Lectura de memoria



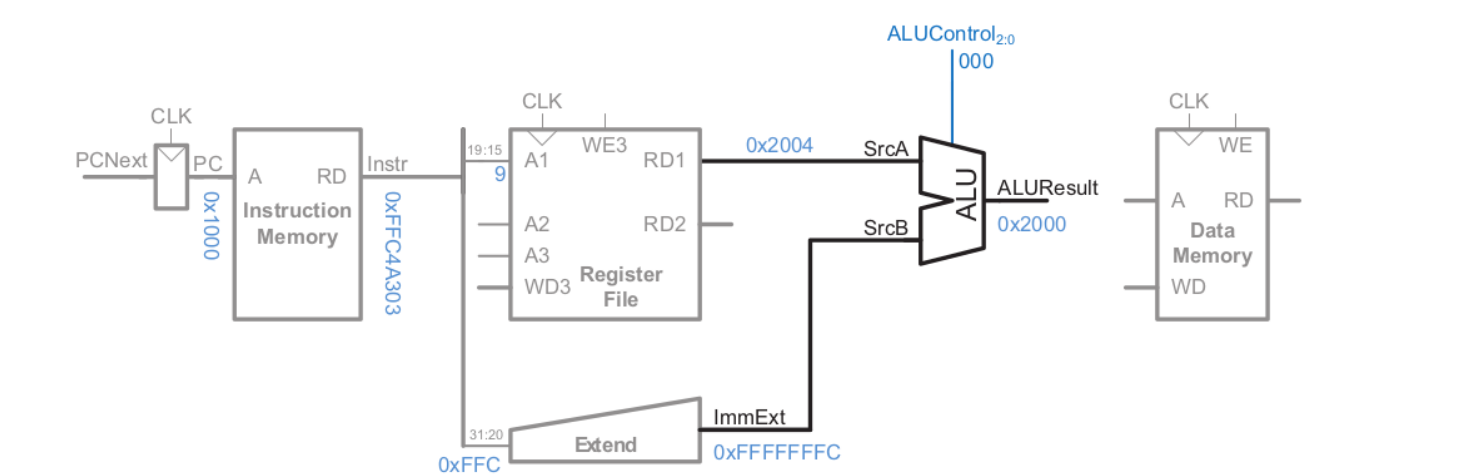
Los bits 19:15 de la instrucción indican el índice de 5 bits del operando fuente (registro) que contiene la base de la dirección a leer de memoria, por este motivo conectamos esta parte de la salida de datos de la memoria de instrucciones **RD** a la entrada de dirección de lectura **A1** del archivo de registros.

Extensión del desplazamiento



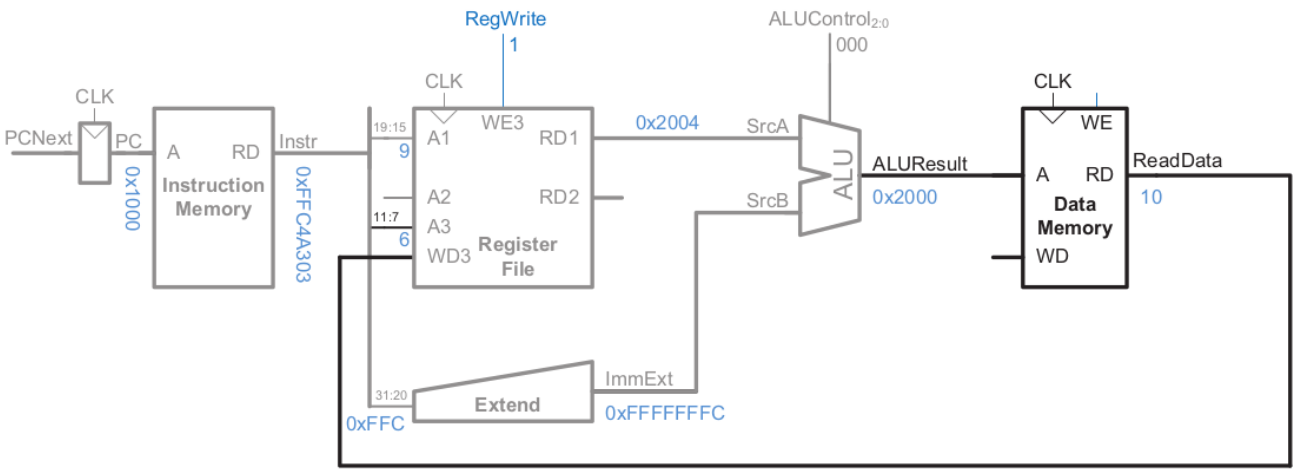
El cómputo de la dirección toma en cuenta también el desplazamiento, que en esta instrucción se codifica como los 12 bits que se encuentran en 31:20, como se trata de un valor en complemento a dos que debemos sumar a la base, es necesario extenderlo con un componente adicional.

Sumando la base



Para calcular la dirección de lectura utilizamos la ALU, ingresando base y desplazamiento como entradas e indicando que la opreación a realizar es una suma.

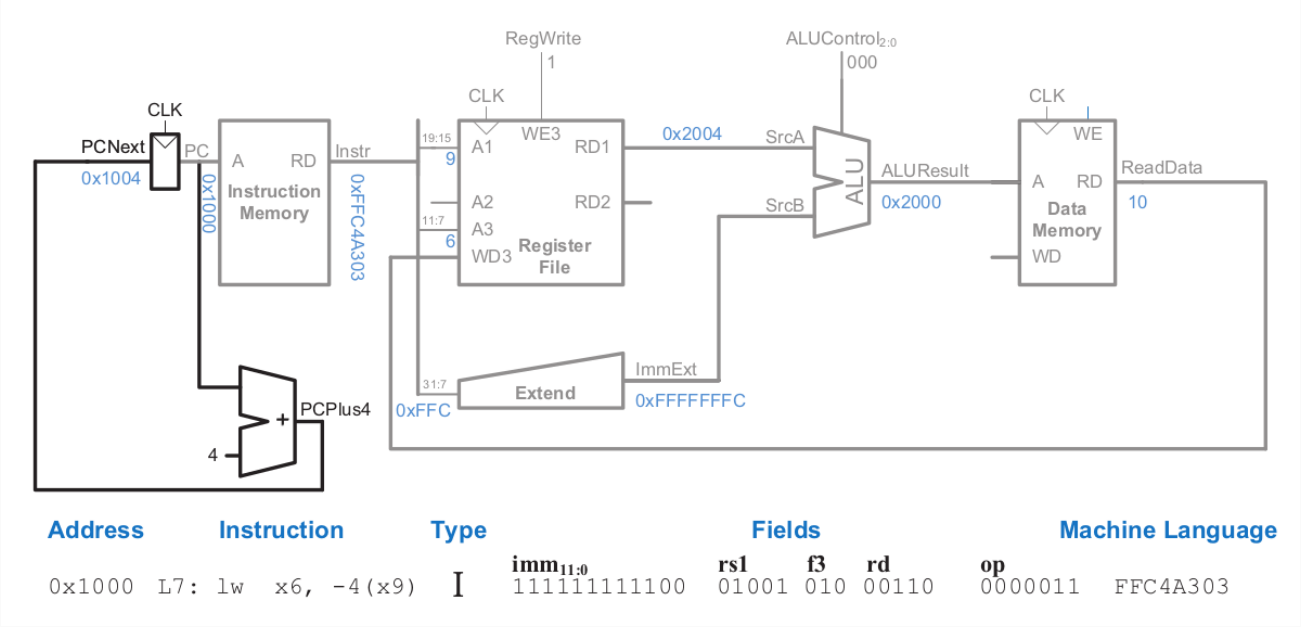
Escritura a registro



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

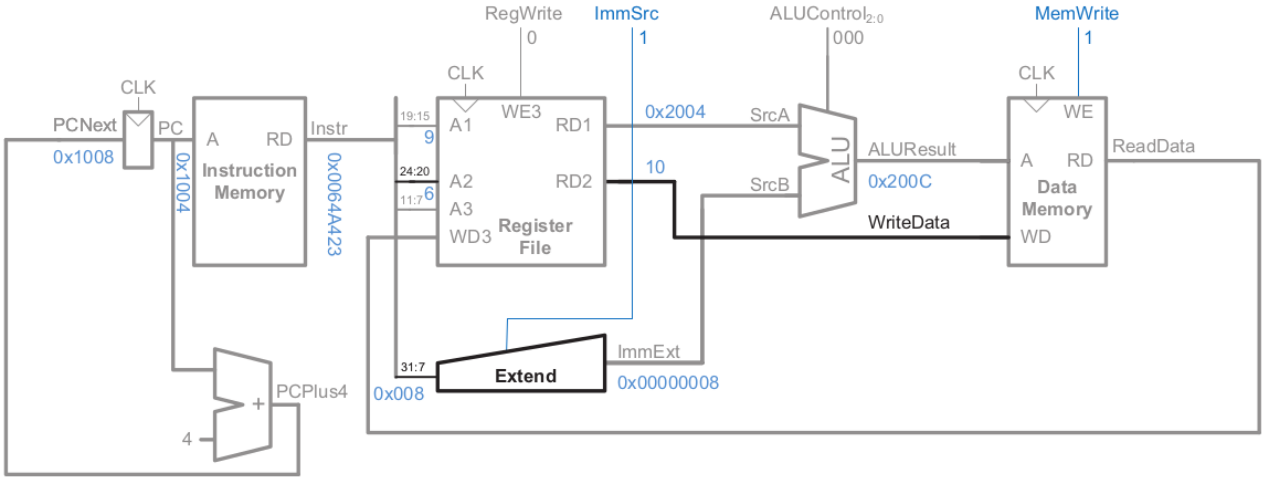
El valor resultante define la dirección **A** desde donde leer la memoria de datos, cuyo resultado **RD** es ingresado en el puerto de escritura **WD3** del archivo de registros a la vez que cargamos los bits 11:7 de la instrucción a la dirección de escritura y habilitamos la señal de control **WE3**.

Actualizando el PC



Mientras se está ejecutando esta instrucción debemos, a la par, calcular la posición desde donde leer la próxima instrucción, para esto utilizamos un sumador que incrementa en 4 el valor acual del PC y lo carga en **PCNext**.

Escritura a registro



Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

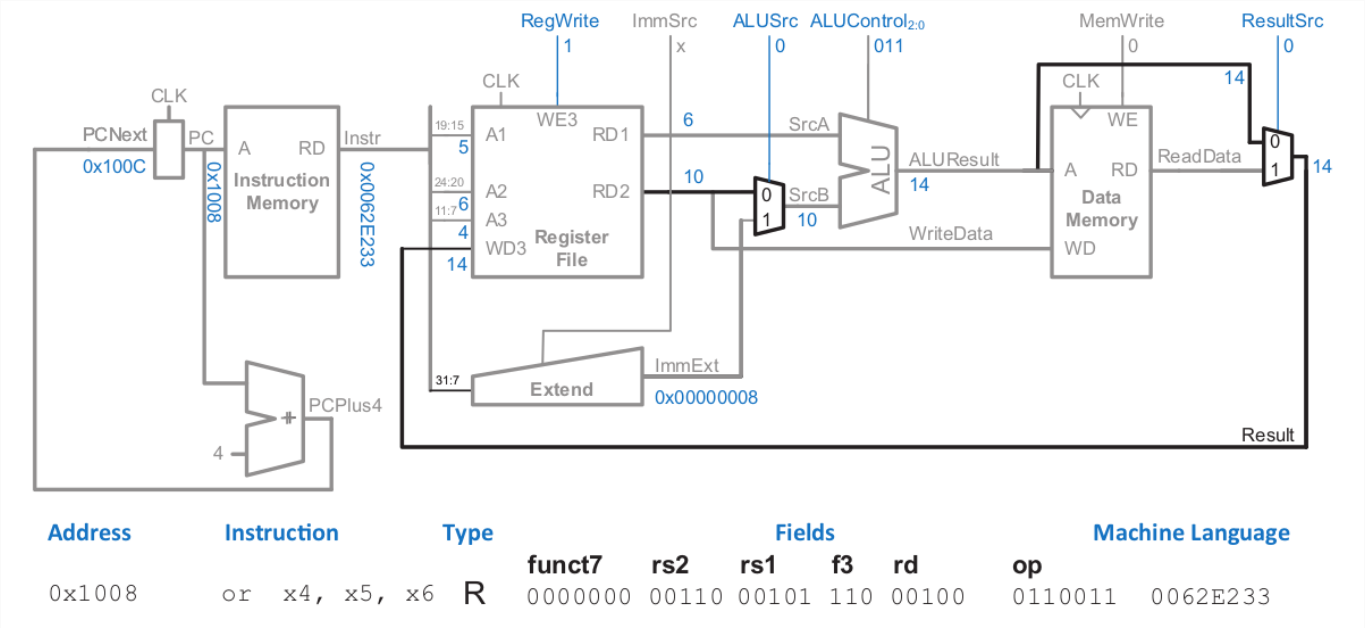
Para una escritura a memoria (**sw**), se utiliza el mismo mecanismo para determinar la dirección con una base y desplazamiento, pero se realiza una segunda lectura desde el banco de registros a través de los bits 24:20 de la instrucción indicando la posición en **A2** y asignando la salida **RD2** al puerto de escritura **WD** de la memoria de datos mientras se habilita **WE**.

Instrucciones con registros

Estructura de las instrucciones R

Las instrucciones con registros van a respetar un mismo esquema, donde tendremos dos registros de fuente, uno de destino y donde vamos a utilizar a la ALU para distintas operaciones según la semántica de cada caso.

Datapath actualizado



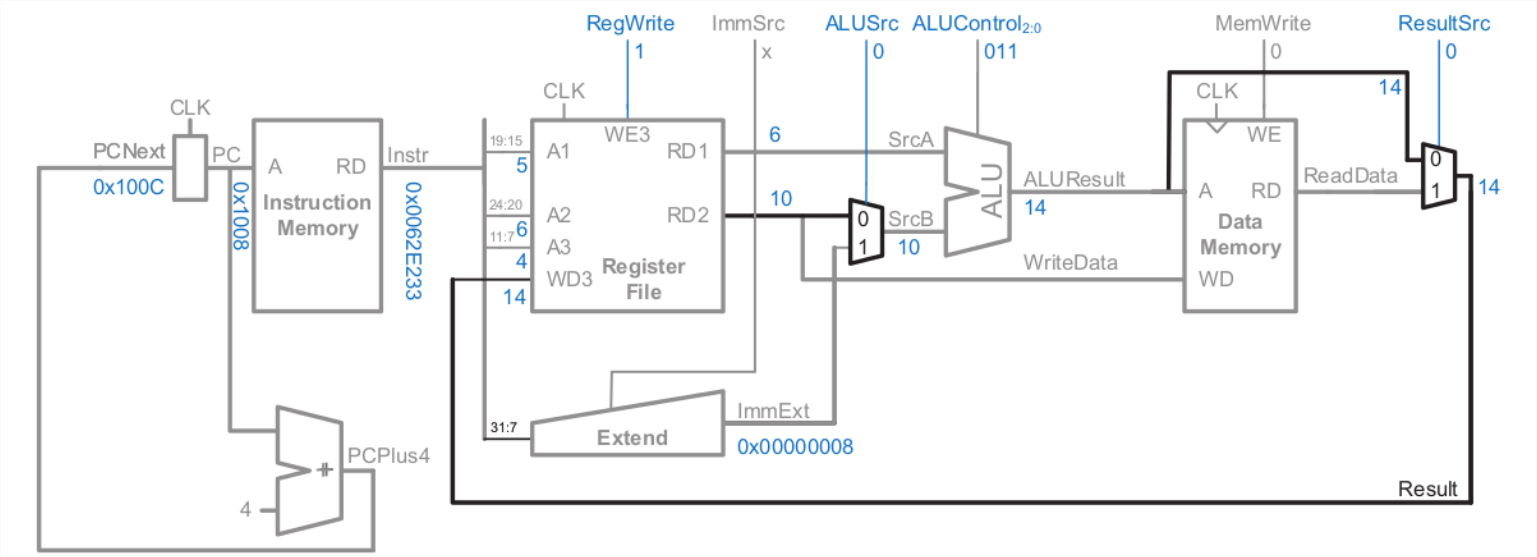
Address	Instruction	Type	Fields					Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

Agregamos dos multiplexores, uno para permitir usar el segundo puerto de lectura del archivo de registros **RD2** como segundo operando de la ALU, y otro para permitir usar la salida de la ALU como dato a escribir en el puerto de escritura del archivo de registros **WD3**. La entrada de operación de la ALU determina la semántica de la instrucción.

Diseño incremental

Podemos notar que el diseño de esta microarquitectura realiza las operaciones necesarias para computar todas la instrucciones presentadas hasta ahora y permite decidir cuáles salidas actualizan el estado del procesador en base a las señales de control de las memorias, el extensor de signo, la ALU y los multiplexores.

Datapath actualizado



Address	Instruction	Type	Fields					Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

Observemos las señales de control: **RegWrite, ImmSrc, ALUSrc, ALUControl, MemWrite, ResultSrc**. El manejo de estas señales será la responsabilidad de la **unidad de control**.

Instrucciones de salto

Salto condicional

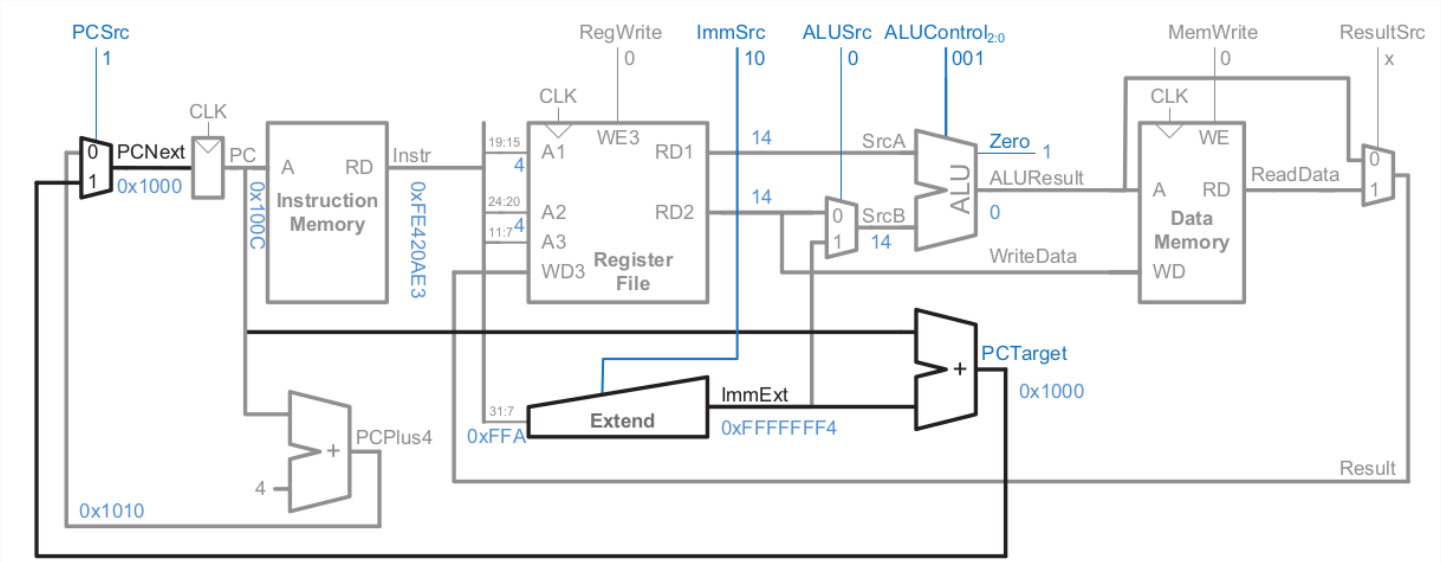
Las instrucciones de salto condicional definen un desplazamiento con respecto al PC de 13 bits codificado en 12 bits, donde el último bit se supone siempre en cero, es por esto que el extensor de signo debe tratar este caso por separado, con lo que su entrada de control pasa a tener dos bits.

Extensión de signo

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate

En esta tabla vemos cómo debe intepretar y extender las entradas el extensor para cada caso según el tipo de instrucción. Esto se le indica a través de la entrada de control **ImmSrc** de dos bits.

Instrucciones de salto

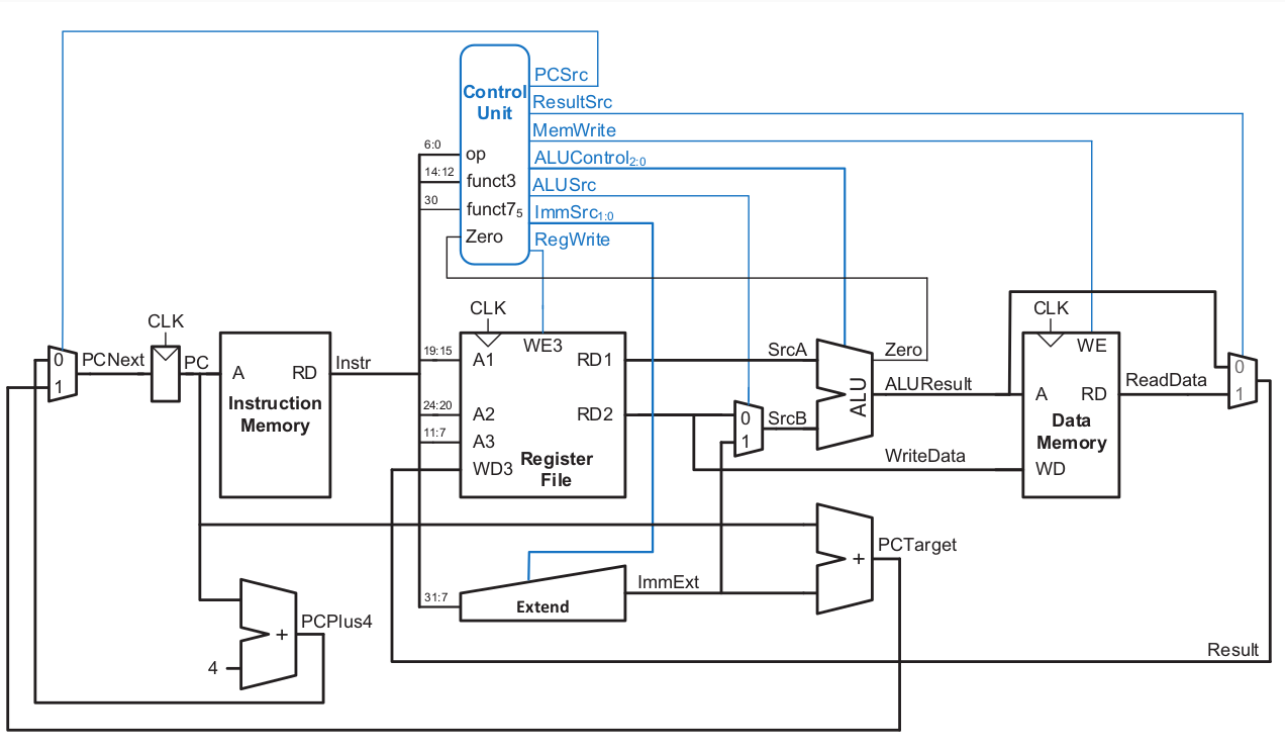


Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Agregando el caso necesario al extensor, un multiplexor y un sumador para actualizar la entrada de **PCNext**, podemos implementar soporte para saltos condicionales. El multiplexor selecciona la segunda entrada solamente si se cumple la condición de salto (en este caso si el flag Z está activado).

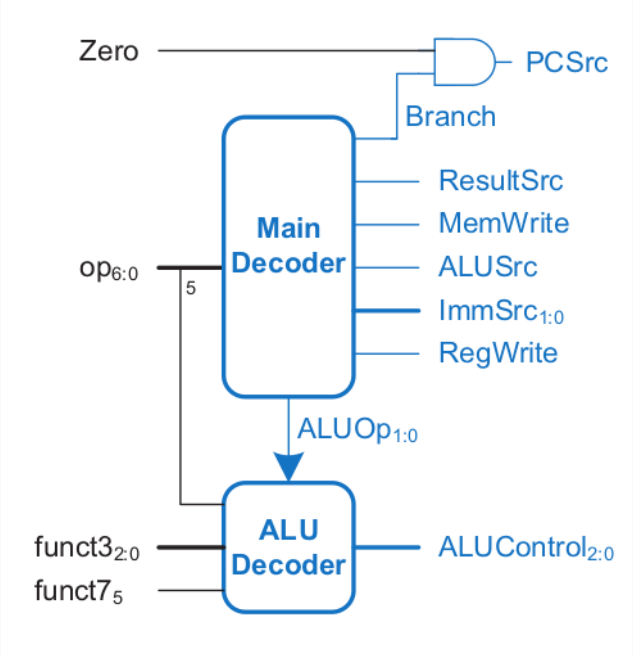
Lógica de control

Lógica de control



Aquí vemos el **datapath** y la **unidad de control** de un procesador de ciclo simple, con sus entradas, salidas y señales de control.

Unidad de control en cascada



La unidad de control se puede desacoplar de forma jerárquica entre el **controlador** y el **decodificador**, donde el decodificador decide qué operación realizar en la ALU, también se agrega una compuerta AND para decidir si se realiza el salto condicional en el caso de **beq**.

Unidad de control

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01

El **controlador** debe implementar esta función en la versión desacoplada, veamos que **ALUOp** indica simplemente si se debe realizar una operación de la ALU o no, el decoder será responsable de decidir cuál operación realizar.

Unidad de control

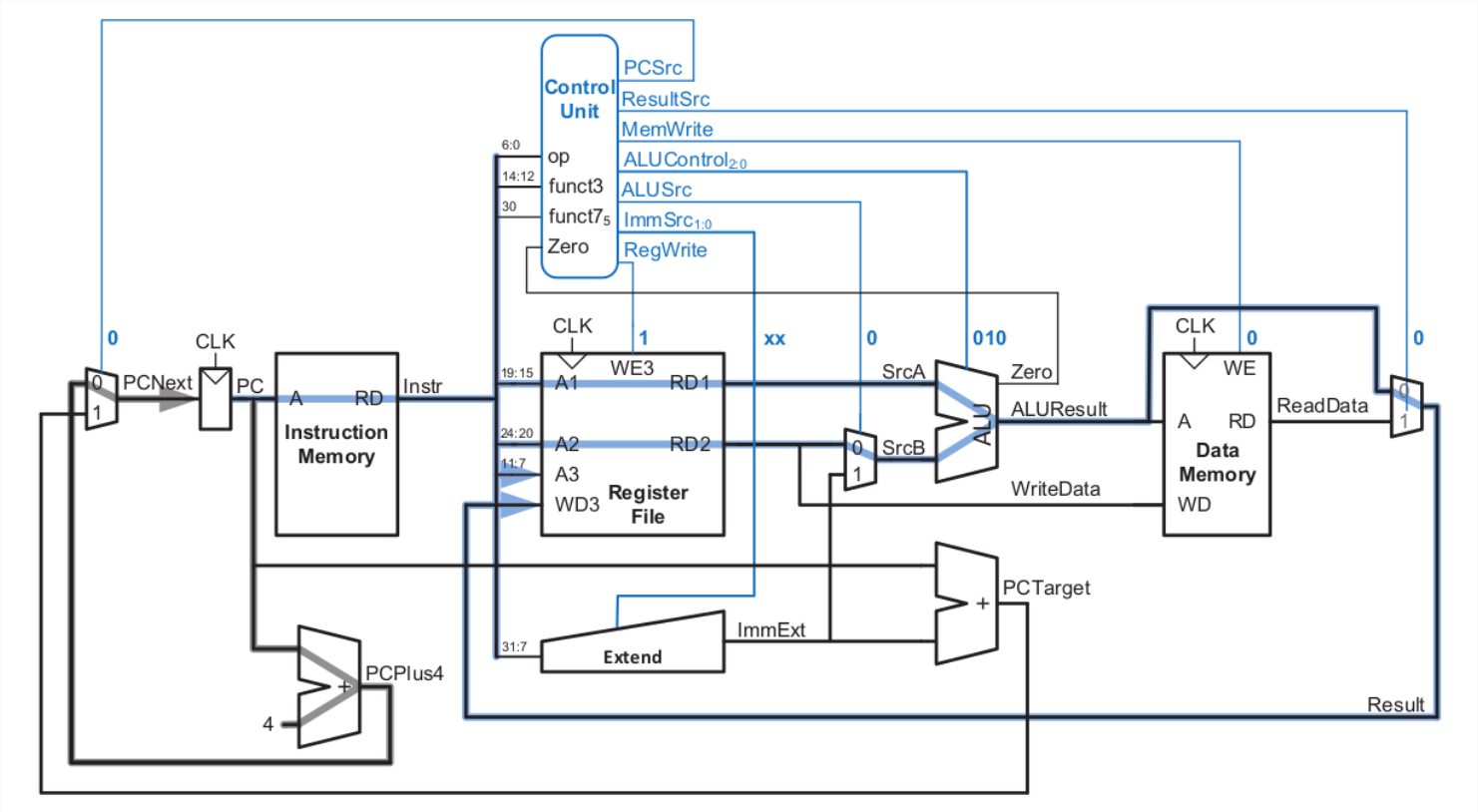
ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

El **decodificador** debe implementar esta función en la versión desacoplada.

Implementando ambas unidades

Con las técnicas vistas para los circuitos combinatorios podemos implementar tanto el **controlador** como el **decodificador**.

Ejemplo de ejecución (and)



Add inmediate

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01
addi	**0010011**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

Si quisiéramos agregar soporte para una suma con inmediato, sería suficiente agregar la siguiente línea a la función del **controlador**.

Cierre

Repaso

Hoy vimos:

- Definición de **microarquitecturas**.

Repaso

Hoy vimos:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.

Repaso

Hoy vimos:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.
- Procesador de ciclo simple.

Repaso

Hoy vimos:

- Definición de **microarquitecturas**.
- Estado de arquitectura, elementos de memoria, datapath y unidad de control.
- Procesador de ciclo simple.
- Instrucciones de memoria, registros y saltos condicionales.

Fin