DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 3 - Almacenamiento de Datos

Profesor: Germán Leandro Contreras Sagredo

Objetivos de la clase

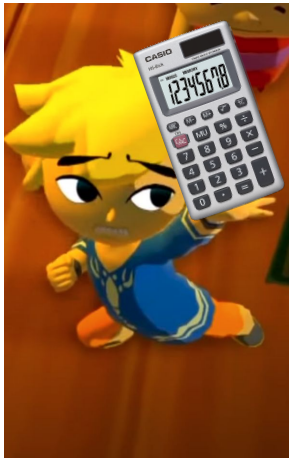
- Entender la necesidad de almacenar datos en nuestra máquina programable.
- Conocer distintos circuitos digitales y componentes que permiten almacenar datos.
- Realizar ejercicios que consoliden los conocimientos anteriores.

Hasta ahora...

Ya tenemos el mecanismo y los medios para poder operar con números en nuestra máquina programable.

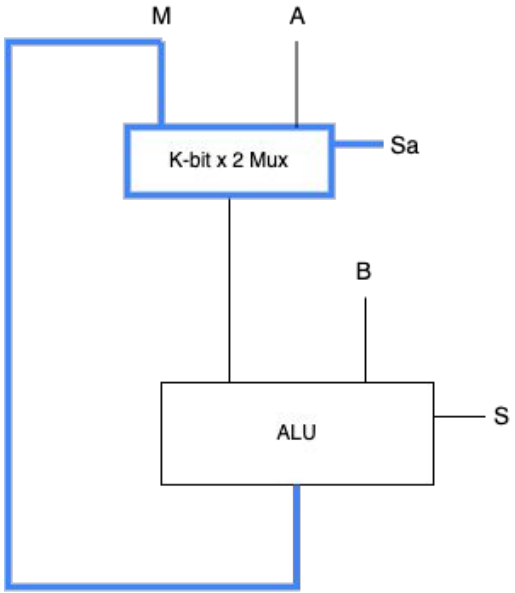
Pero, ¿qué podemos hacer con nuestra ALU actualmente?

- Realizar 8 operaciones distintas entre dos números *A* y *B*.
- Observar el resultado. ¿Podemos hacer algo con el resultado?



Creando una calculadora con la ALU

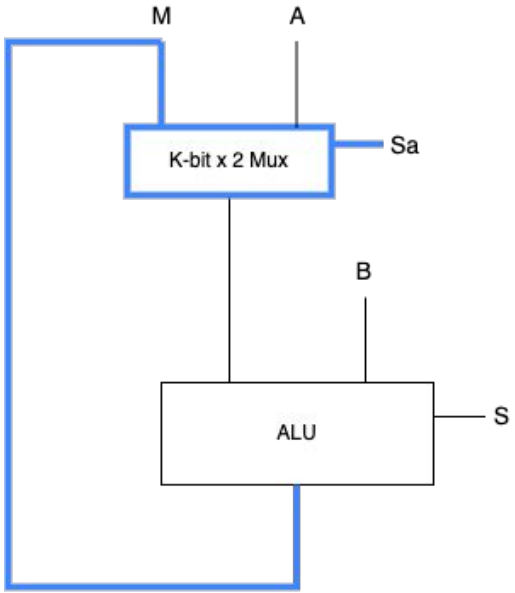
- Queremos usar el resultado de nuestra ALU. Una primera opción es conectar el resultado con la **entrada *A*** de la unidad mediante un Mux de dos entradas con un nuevo bit de selección para decidir qué valor operar.



- ¿Existe algún problema con esto?
  - ¿Cuánto tiempo toma la operación?
  - ¿Qué tan rápido debo cambiar la señal Sa?
  - ¿Cuándo o cómo me detengo?
  - ¿Cómo retengo el resultado final?

Creando una calculadora con la ALU

- Esta alternativa no es óptima. Requerimos de una componente que permita:
  - Almacenar un estado (bit o secuencia de bits).
  - Realizar el cambio de estado en **un instante determinado**.
- Los componentes que necesitamos son los ***latches*** y ***flip-flops***.



Latch

- Componente que permite **guardar el estado anterior de una señal**. El término proviene del verbo en inglés: “encerrar” (con pestillo). Estos se conocen como **circuitos secuenciales**.

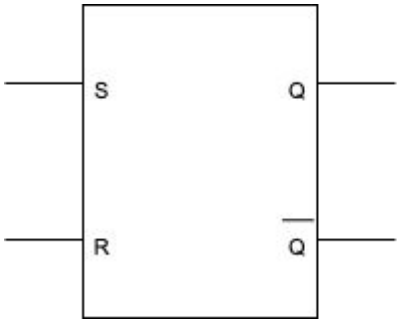
- Se construyen con compuertas **que ya conocemos** (`NAND`, `NOR`, `NOT`, etc.).

- Existen dos tipos principalmente: RS, D.

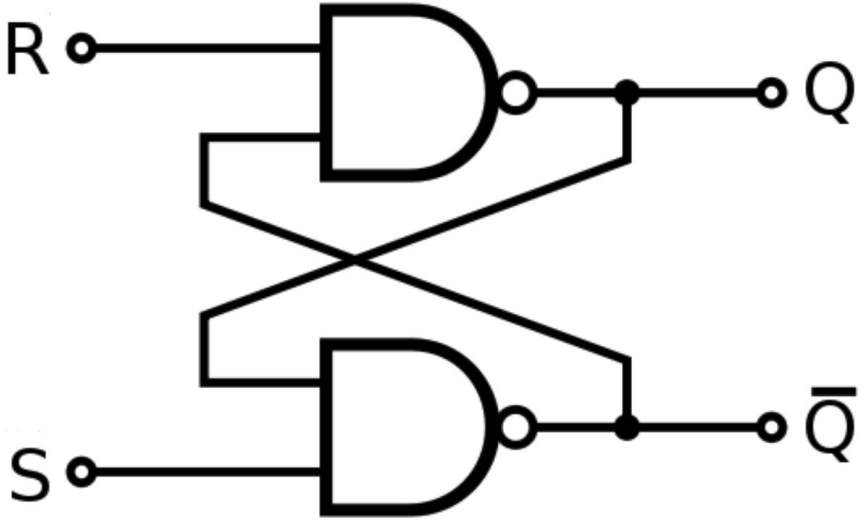
- Distintas combinaciones de estos nos permitirán crear distintos tipos de memoria.

Latch RS

Componente que realiza *set* ($Q = 1$), *reset* ($Q = 0$) o mantiene un estado previo: $Q(t+1) = Q(t)$. A continuación veremos cómo funciona.



Abstracción



S	*R*	*Q(t+1)*
0	0	-
0	1	0
1	0	1
1	1	*Q(t)*

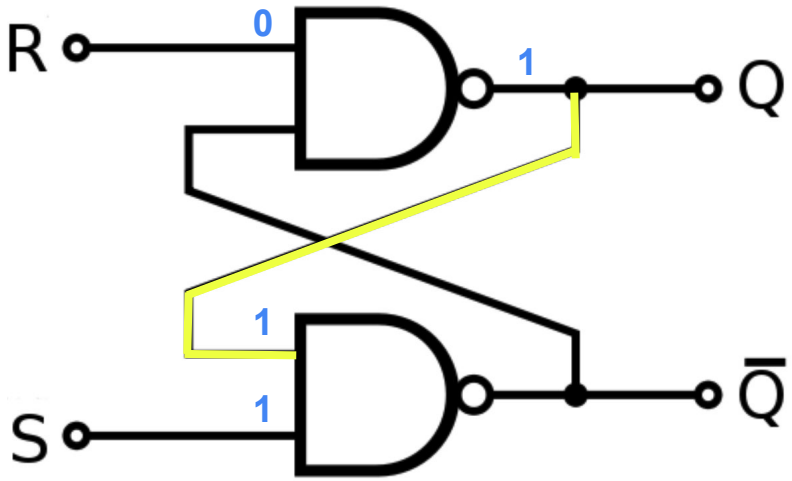
Latch RS - Caso a caso

$R = 0, S = 1$

Partamos el flujo por R. Como su valor es 0, sabemos que la compuerta NAND entregará como salida 1.

Luego, como la compuerta NAND inferior recibe dos *input* 1, su salida será un 0.

De esta forma, $Q = 1$ y $\neg Q = 0$.



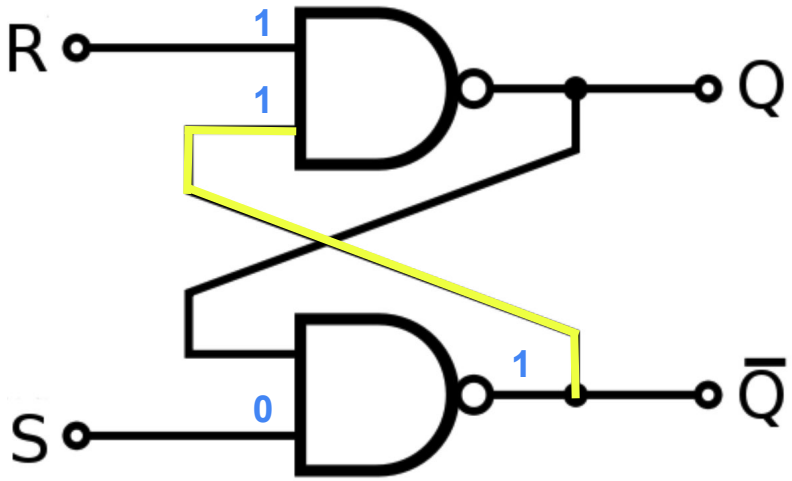
Latch RS - Caso a caso

$R = 1, S = 0$

Partamos el flujo por S. Como su valor es 0, sabemos que la compuerta `NAND` entregará como salida 1.

Luego, como la compuerta `NAND` superior recibe dos *input* 1, su salida será un 0.

De esta forma, $Q = 0$ y $\neg Q = 1$.

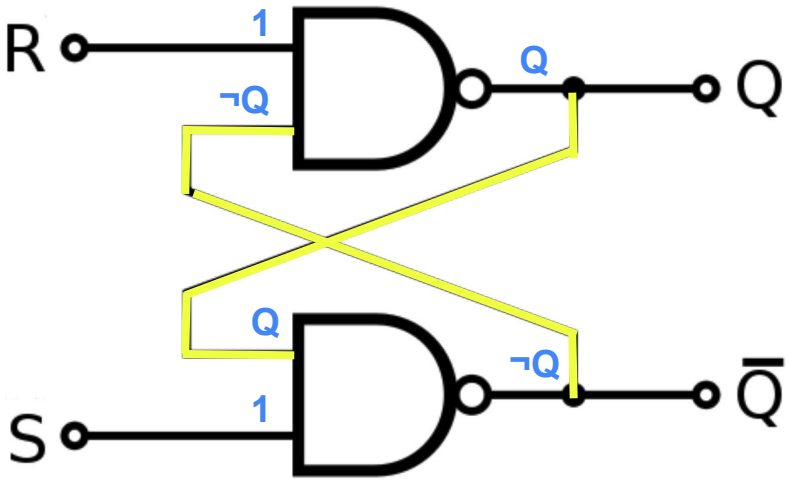


Latch RS - Caso a caso

$R = 1, S = 1$

Independiente del flujo que sigamos, vemos que ambas salidas dependen de Q y $\neg Q$ por leyes de dominación.

A continuación, realizaremos las operaciones lógicas para verificar la permanencia de los estados.



Latch RS - Caso a caso

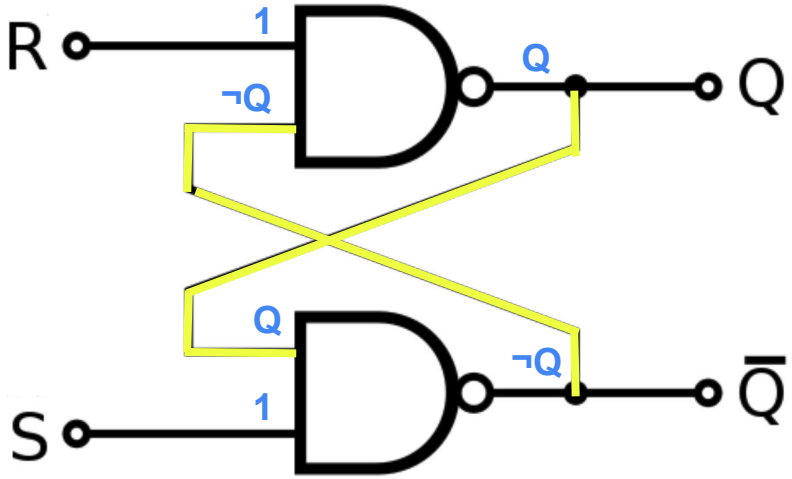
$R = 1, S = 1$

Por dominación.

R NAND ¬Q
= NOT(1 AND ¬Q)
= NOT(¬Q) = Q

S NAND Q
= NOT(1 AND Q)
= NOT(Q) = ¬Q

¡Retenemos el estado!



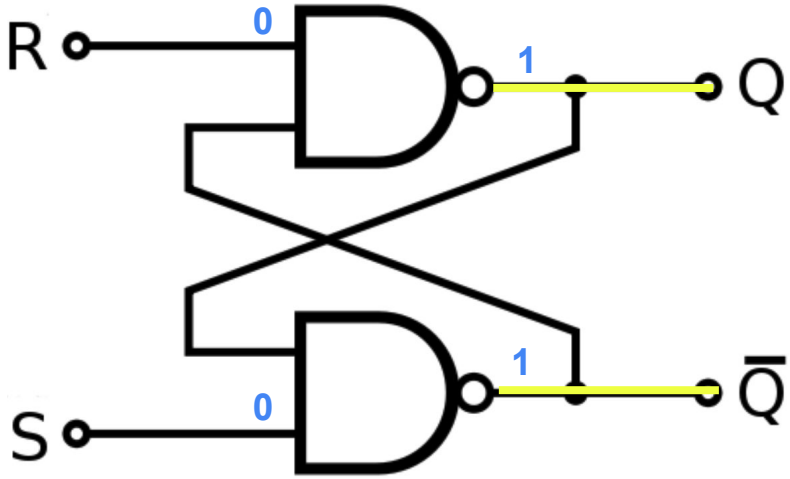
Latch RS - Caso a caso

$R = 0, S = 0$

En ambos flujos, la salida de la compuerta `NAND` será 1.

De esta forma, $Q = \neg Q = 1$.

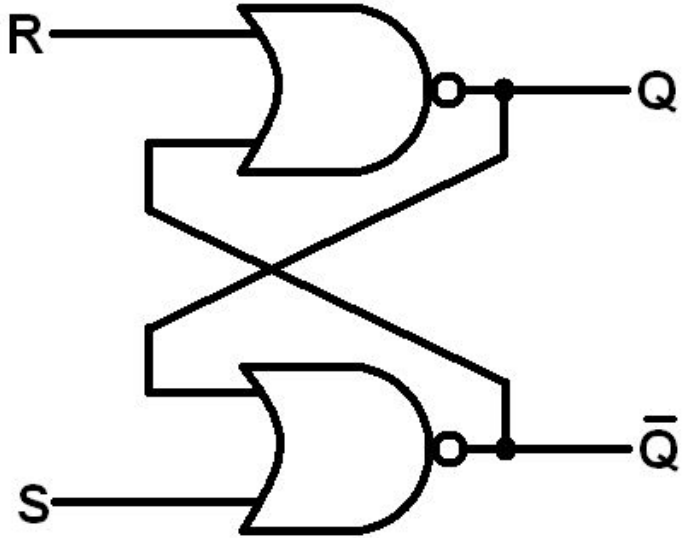
Esto se considera un estado **inválido**, ya que no se respeta el inverso del estado final.



Latch RS - Otra construcción

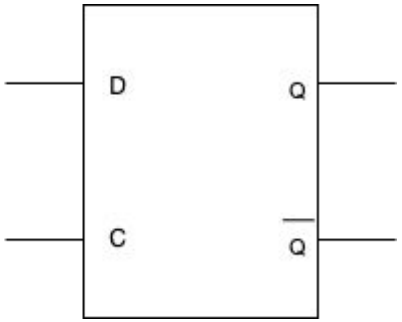
Recordando que tanto la compuerta `NAND` como la compuerta `NOR` se consideran universales, también existe la construcción del latch RS con esta última.

Por temas de comodidad y consistencia, seguiremos utilizando compuertas `NAND` para el resto de componentes a construir.

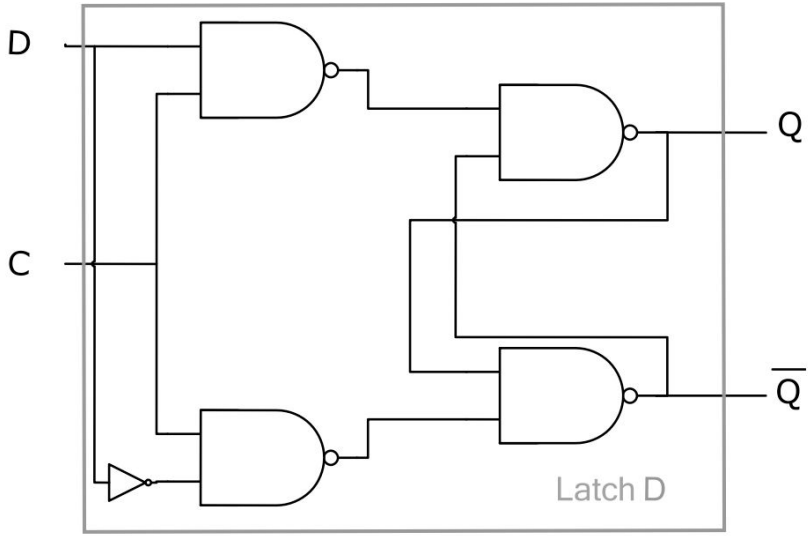


Latch D

Componente que actualiza el valor de un estado *Q* con una señal *D* si, y solo si la señal de control está activa (*C* = 1). *D* = ***D****ata*.



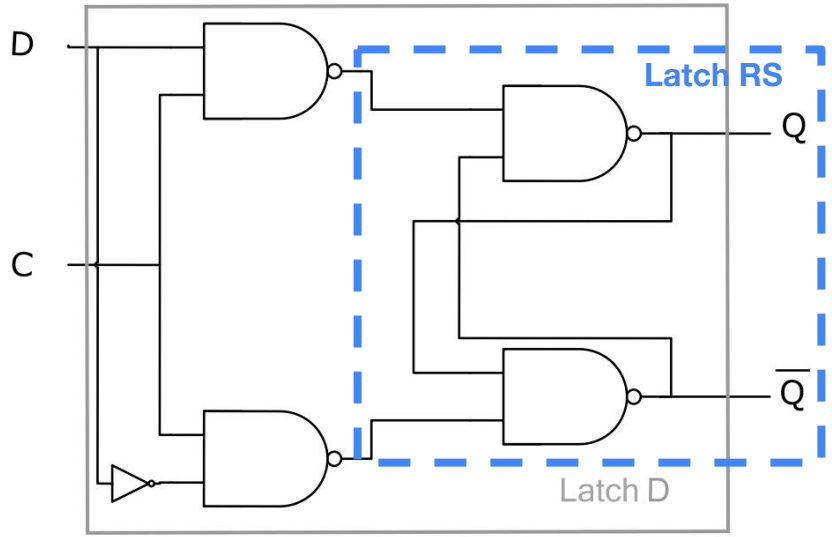
Abstracción



C	*D*	*Q(t+1)*
0	0	*Q(t)*
0	1	*Q(t)*
1	0	0
1	1	1

Latch D

Si observamos bien, es una versión del latch RS **sin estados inválidos**.

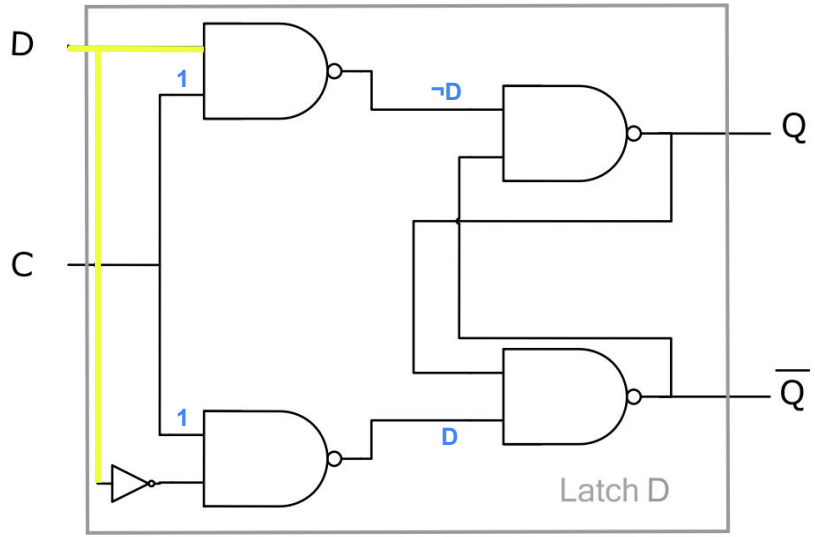


Si *C* = 0, las dos entradas del latch RS "interno" se vuelven 1, siendo esta la combinación que **retiene el estado**.

Por otra parte, dada la negación de *D* en la secuencia inferior, nunca se tendrá la combinación inválida del latch RS.

Latch D

¿Y si $C = 1$?



Las salidas de ambas compuertas NAND de entrada serán el inverso de la señal ingresada. Luego, como una de las señales *D* o ¬*D* será 0, una de las compuertas NAND de salida tendrá valor 1, lo que permite el cambio de estado de la componente.

Latch D

- Al ser su manejo de señales más intuitivo, lo utilizaremos por sobre el latch RS.

- Ahora, ¿es este el componente que necesitamos? Podemos almacenar el valor de entrada con la señal de control, no obstante, si la dejamos activa el estado final del componente podría seguir cambiando si cambia el dato de ingreso.

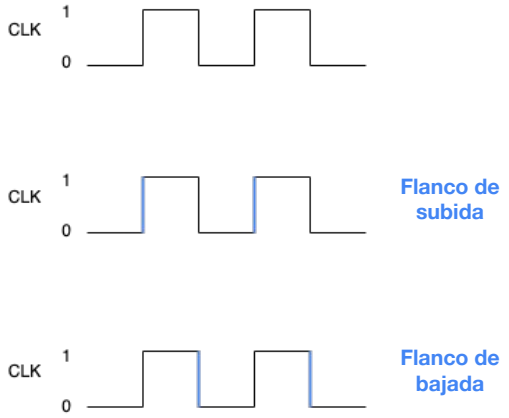
- **Necesitamos que la actualización ocurra una única vez en un instante dado**.

Flanco de subida y de bajada

- Para poder actualizar un estado en un instante dado, necesitamos entender el concepto de flanco.

- Supongamos que tenemos una señal CLK que cambia su valor de 0 a 1 y de 1 a 0 con una frecuencia constante.

- El flanco de subida corresponderá al instante en que la señal cambia de 0 a 1, mientras que el flanco de bajada será el instante en que la señal cambia de 1 a 0.

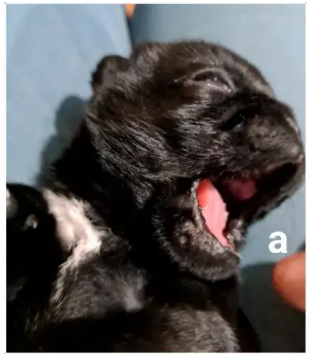


Flanco de subida y de bajada

Queremos, entonces, que nuestro componente de almacenamiento actualice su estado si, y solo si, **existe un flanco de subida en la señal de control**.

Así es como se construye el componente **flip-flop**.

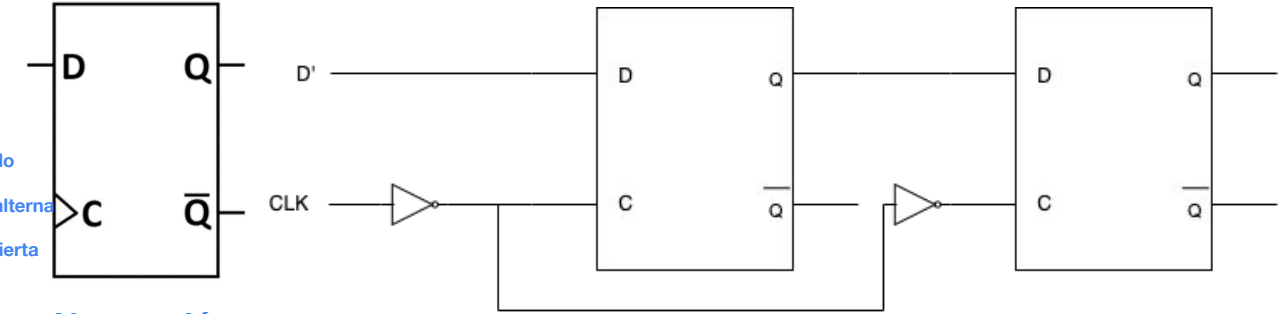
Su nombre no viene del término para “sandalias” en inglés, sino que su uso data de los 1900s, haciendo referencia a la “inversión completa de dirección”.



Flip-flop D

- Componente que permite **guardar el estado anterior de una señal en un instante dado**.

- El instante de almacenamiento corresponde al flanco de subida de la señal de control. Veremos a continuación el caso a caso.



* El triángulo indica que la señal C alterna su valor con cierta frecuencia.

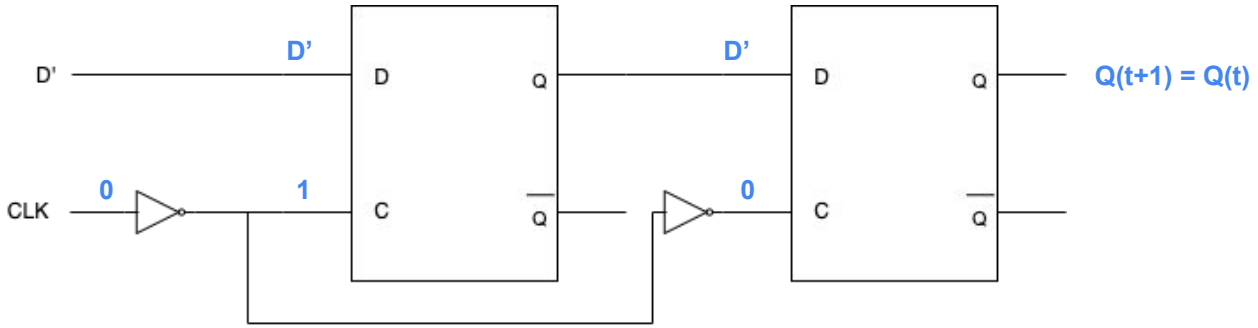
Abstracción

CLK	*D*	*Q(t+1)*
0/1/↓	0/1	*Q(t)*
↑	0	0
↑	1	1

Flip-flop D - Caso a caso

CLK = 0

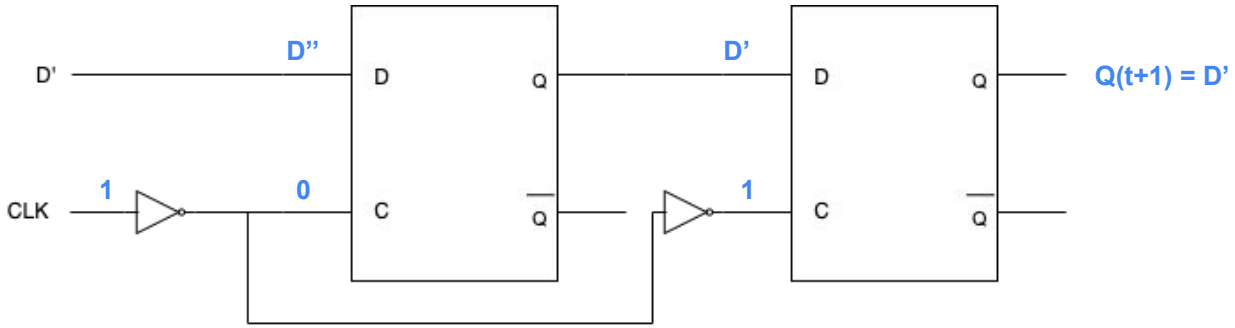
Se habilita el almacenamiento en el primer latch pero no en el segundo, la salida del segundo latch (estado final) sigue siendo el estado anterior.



Flip-flop D - Caso a caso

CLK = 0 → CLK = 1

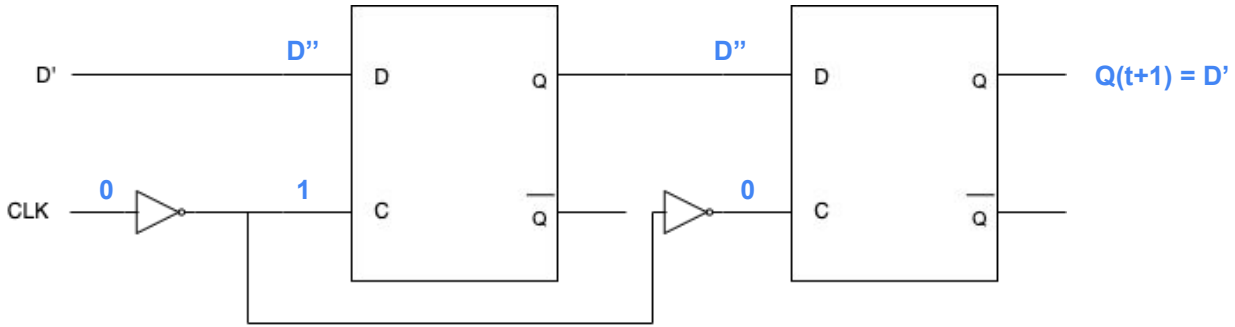
Se habilita el almacenamiento en el segundo latch pero no en el primero, por lo que aunque ingresara un nuevo valor *D*'' este no se almacenará preliminarmente. Por otra parte, el estado final ahora se convierte en *D*'.



Flip-flop D - Caso a caso

CLK = 1 → CLK = 0

Se vuelve a habilitar el almacenamiento en el primer latch y se bloquea el segundo. Esto hace que ahora la nueva señal *D''* sea almacenada en el primer latch **sin alterar nuestro estado final**. Así termina un ciclo de la señal CLK.



Flip-flop D

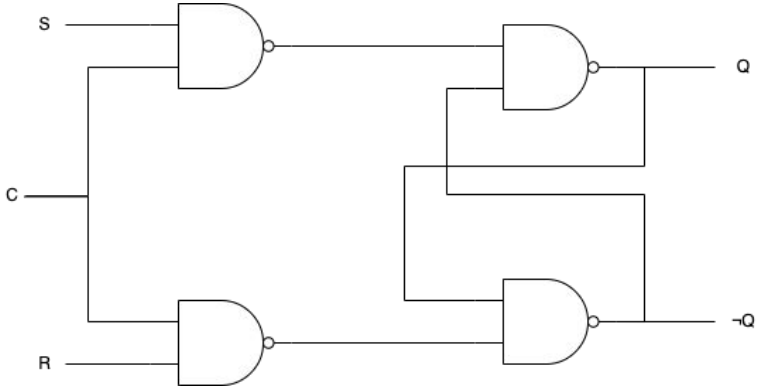
- Los flip-flop D (construidos con latches D) serán los componentes a utilizar para crear los registros de nuestra máquina programable.

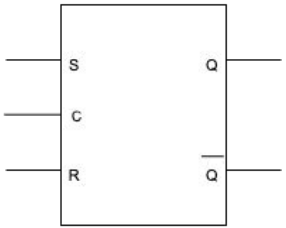
- No obstante lo anterior, cabe señalar la existencia de otros flip-flop:

  - Flip-flop RS
  - Flip-flop JK
  - Flip-flop T

Flip-flop RS

Similar al latch RS, pero con cambio de estado solo en flanco de subida. Para su construcción, primero se define el latch RS con señal de control C. Notar que por las compuertas `NAND` conectadas a la señal *C*, los *input* del latch RS se vuelven ¬*S* y ¬*R*. Esto invierte las combinaciones que almacenan o invalidan el estado final.



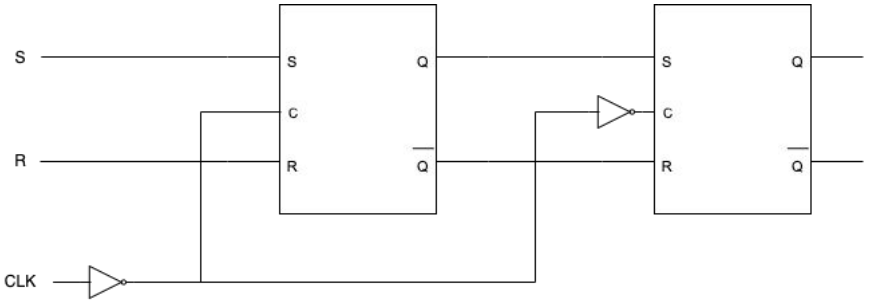


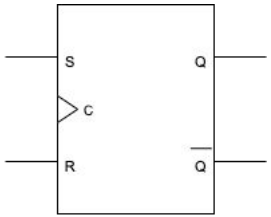
Abstracción

C	*S*	*R*	*Q(t+1)*
0	0/1	0/1	*Q(t)*
1	0	0	*Q(t)*
1	0	1	0
1	0	1	1
1	1	1	-

Flip-flop RS

Al igual que con el flip-flop D, se crea el flip-flop RS con una secuencia de latches, en este caso, del tipo RS con control *C*.



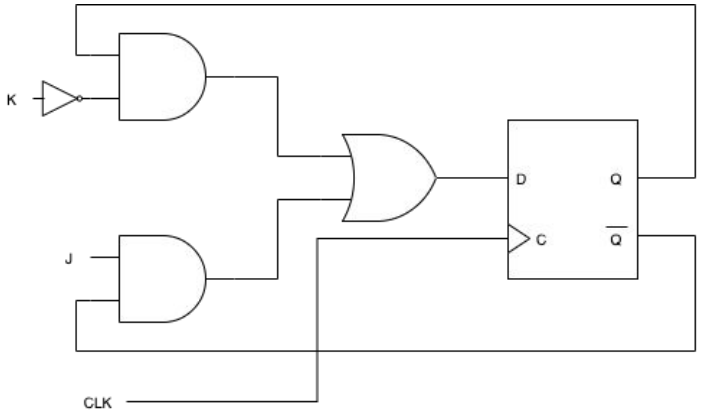


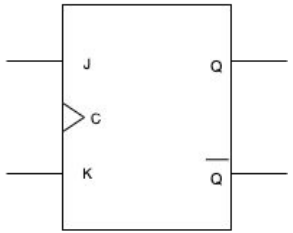
Abstracción

CLK	*S*	*R*	*Q(t+1)*
0/1/↓	0/1	0/1	*Q(t)*
↑	0	0	*Q(t)*
↑	0	1	0
↑	0	1	1
↑	1	1	-

Flip-flop JK

Llamados así por su inventor, **J**ack **K**ilby. Funcionan de la misma forma que un flip-flop RS, pero en su caso no existe combinación inválida de señales de entrada, sino que esta combinación **invierte** el estado final.



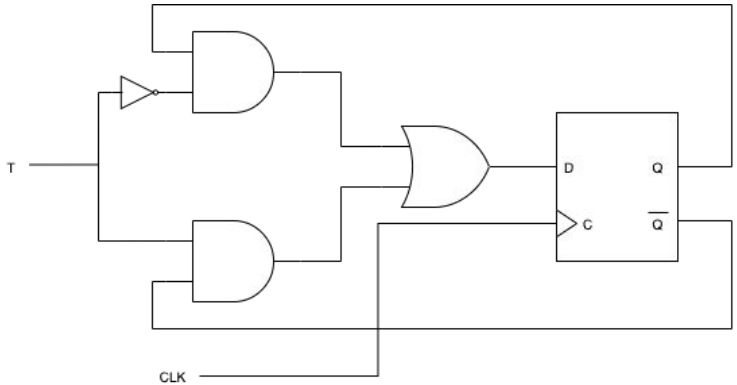


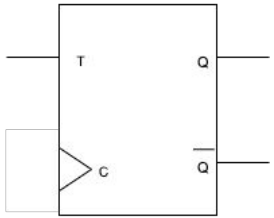
Abstracción

CLK	*J*	*K*	*Q(t+1)*
0/1/↓	0/1	0/1	*Q(t)*
↑	0	0	*Q(t)*
↑	0	1	0
↑	1	0	1
↑	1	1	¬*Q(t)*

Flip-flop T

Llamados así por el verbo ***T**oggle* (“alternar”). Cuando su señal *T* se activa, el estado del componente se alterna a su valor inverso.





Abstracción

CLK	*T*	*Q(t+1)*
0/1/↓	0/1	*Q(t)*
↑	0	*Q(t)*
↑	1	¬*Q(t)*

Algunos comentarios adicionales antes de seguir

- Los flip-flop JK son los más utilizados por la flexibilidad que otorgan con la cantidad de combinaciones que poseen. No obstante, usaremos los flip-flop D por comodidad.

- En teoría, se pueden definir latches de tipo JK o T, pero no se usan en la práctica.

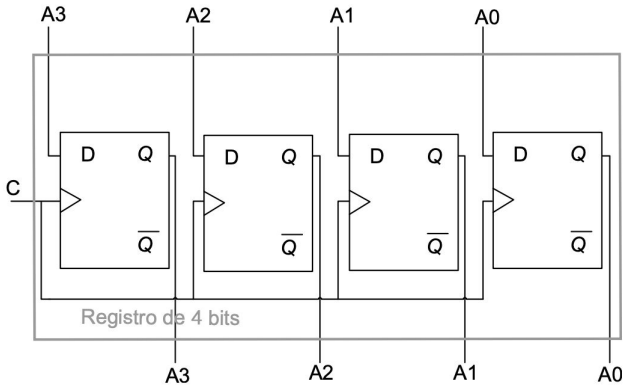
- Al tener dos latches en secuencia, estos suelen llamarse latch “maestro” y latch “esclavo” respectivamente.

“Interesante”

Registro de memoria

Volviendo al flip-flop D: Si deseamos almacenar números de 4 bits, por ejemplo, simplemente creamos un componente con 4 flip-flops D y una única señal *C* que alterna su valor con cierta frecuencia.

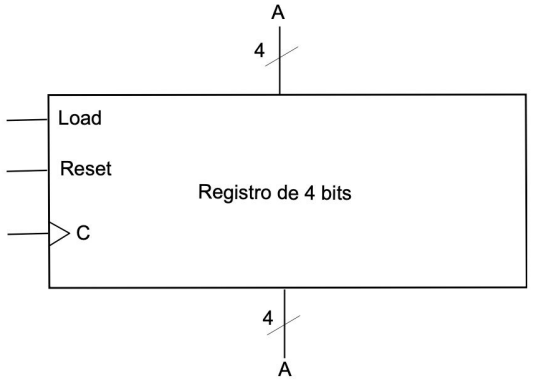
Esto corresponde a un **registro de memoria de 4 bits**.



Registro de memoria

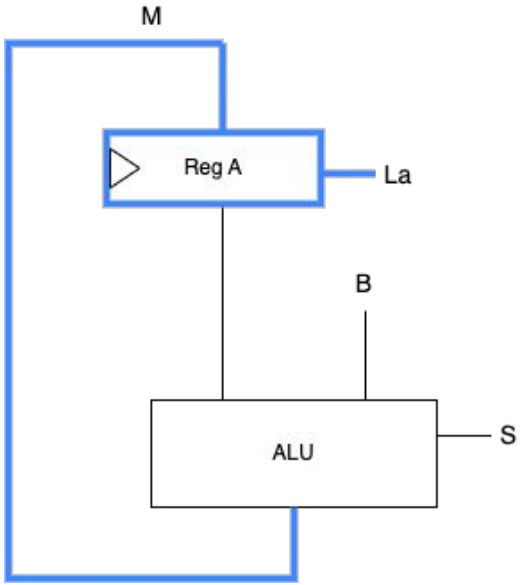
Ahora, si lo dejamos así como está, el valor almacenado cambiará siempre que exista un flanco de subida. Queremos que esto ocurra **solo cuando deseemos hacerlo**. Para esto, agregamos dos señales: Una para cargar o no el valor de entrada (*Load*) y otra para limpiar el valor almacenado (*Reset*).

Ejercicio: Modifique el diagrama interno del registro de memoria de 4 bits para que incluya las señales *Load* y *Reset*.



Creando una calculadora con la ALU

Ahora que contamos con un registro de memoria que puede almacenar estados y cambiarlos en un instante dado, podemos tener una solución óptima para nuestra calculadora.



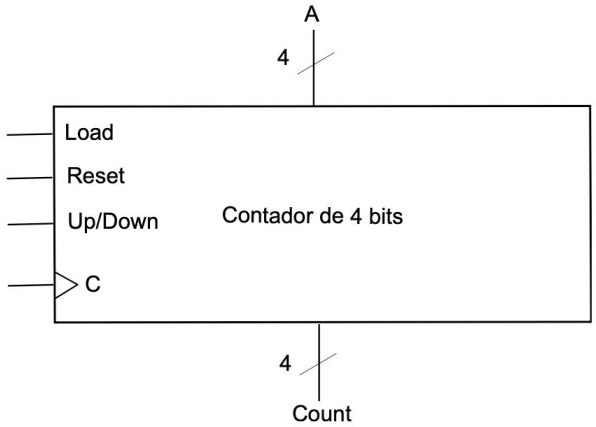
Con la señal L_a (*Load A*) controlamos la sobreescritura o no del registro de *A* con el resultado de la ALU.

Si bien ya tenemos una primera calculadora, debemos conocer otros tipos de memoria a utilizar en nuestra máquina.

Contador

Podemos convertir nuestro registro de memoria en un **contador**, añadiendo señales que incrementan (*Up*) o decrementan (*Down*) el valor almacenado.

Ejercicio: Modifique el diagrama interno del registro de memoria con *Load* y *Reset* de 4 bits para que incluya las señales *Up* y *Down*.



Memorias

Si bien los registros nos permiten almacenar unidades de información o **palabras** (cadena finita de bits), queremos también un componente en el que podamos acceder o modificar distintos datos según una **dirección de memoria**.

Este proceso de acceso (ya sea de lectura o escritura) se conoce como **direccionamiento**.

Dirección en decimal	Dirección en binario	Palabra
0	000	$Palabra_0$
1	001	$Palabra_1$
2	010	$Palabra_2$
3	011	$Palabra_3$
4	100	$Palabra_4$
5	101	$Palabra_5$
6	110	$Palabra_6$
7	111	$Palabra_7$

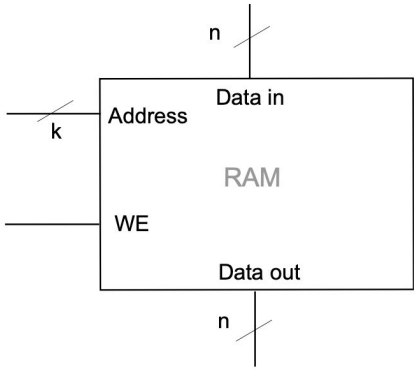
Tipos de memoria

En nuestra máquina programable, usaremos principalmente dos tipos de memoria:

- Memoria de lectura y escritura: ***Random access memory*** o **RAM**.
- Memoria de solo lectura: ***Read only memory*** o **ROM**.

Tipos de memoria - RAM

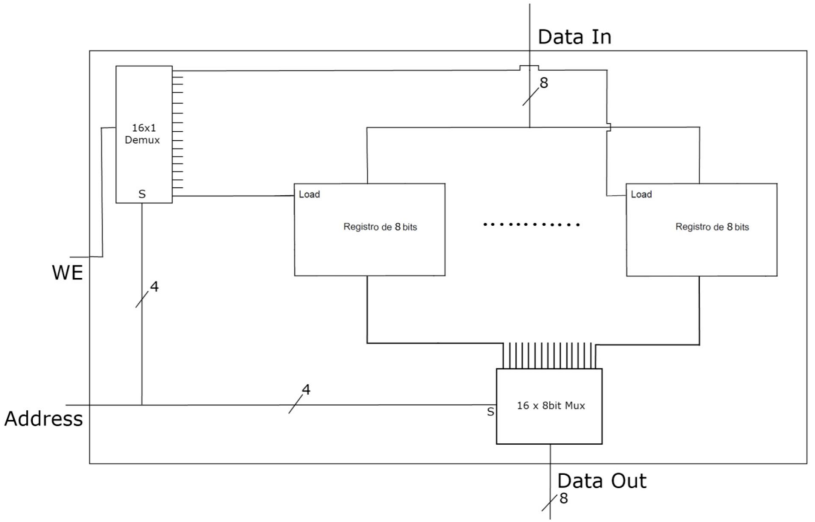
Extensión de los registros. Ocupan flip-flops pero extienden sus señales de control para poder acceder a **una sola palabra** y habilitar o no la escritura de ella. Se les llama *random access memory* porque el acceso a cualquier dirección de memoria es directo (no es necesario recorrer las direcciones anteriores).



Con *k* bits para el bus de direccionamiento y buses de entrada/salida de *n* bits, la RAM almacena 2^k palabras de memoria de *n* bits. Por otra parte, tenemos una señal de un bit `WE` (***W**rite **E**nable*) que habilita o no la escritura sobre la palabra seleccionada.

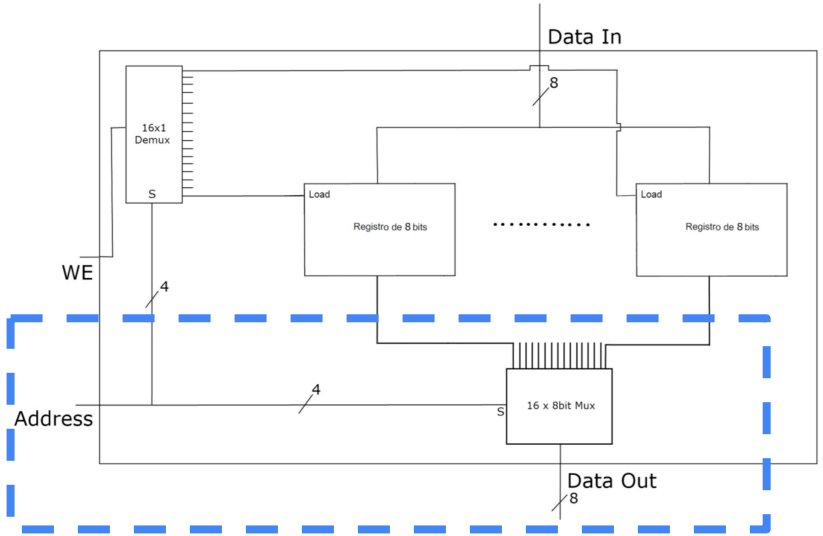
Tipos de memoria - RAM

Haciendo uso solo de los componentes vistos en clases hasta ahora, podemos construir una RAM de palabras de 8 bits (1 byte) y 16 direcciones de memoria de la siguiente manera.



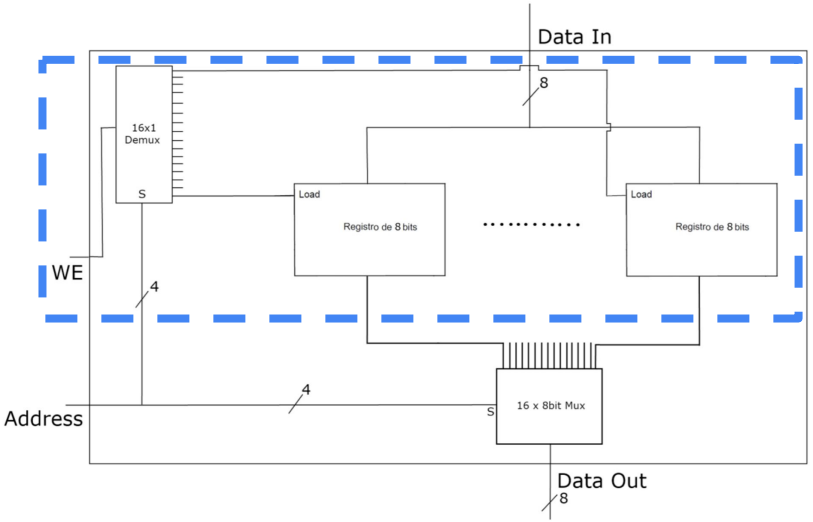
Tipos de memoria - RAM

Si estamos realizando una lectura, con el componente **16x8-bit Mux** seleccionamos **solo uno** de los valores de los registros de memoria internos a través del bus *Address* de 4 bits. Este resultado corresponderá al bus de salida *Data Out*.



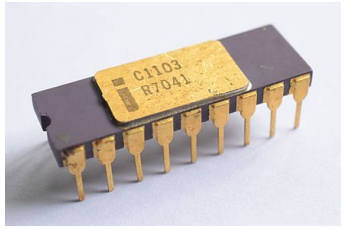
Tipos de memoria - RAM

Si realizamos una escritura, con el componente **16x1-bit Demux** transmitimos la señal WE como *Load* de **solo** el registro de memoria seleccionado a través de la señal *Address* de 4 bits. Esto realizará la carga del bus de entrada *Data In* sobre el registro de memoria deseado. El bus de salida *Data Out* tendrá el mismo valor al escoger el mismo registro sobreescrito.

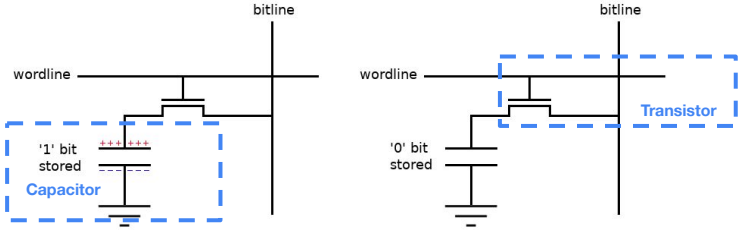


Tipos de memoria - Tipos de memoria RAM

***Dynamic* RAM o DRAM:** Memoria RAM "dinámica" que almacena un bit a partir de un condensador y un transistor (celda de memoria). Son más económicas y fáciles de manufacturar, pero requieren un "refrescado de memoria" o *memory refresh* (lectura y reescritura de las celdas de memoria cada ciertos milisegundos) ya que pierden su carga con facilidad.



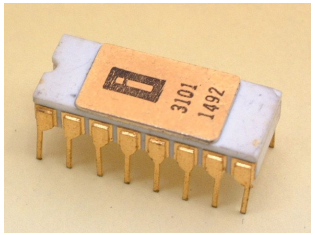
Intel 1103, primera DRAM de circuito integrado



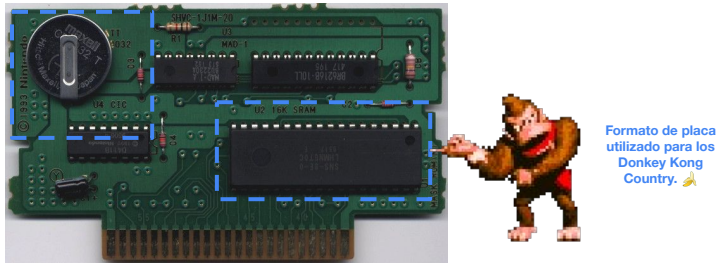
Esquema de almacenamiento de una celda de memoria con un transistor y capacitor.

Tipos de memoria - Tipos de memoria RAM

***Static* RAM o SRAM:** Memoria RAM “estática” que almacena estados con flip-flops.
Si bien necesitan más componentes internos y son más costosas y difíciles de manufacturar, poseen un acceso de memoria más rápido que las DRAM. Por otra parte, no requieren de *memory refreshing*.



Intel 3101, primer producto oficial de la empresa.



Placa de un cartucho de Super Nintendo. Posee una SRAM para guardar el progreso y se mantienen los datos con la alimentación de una batería.

Formato de placa utilizado para los Donkey Kong Country.

Tipos de memoria - Tipos de memoria RAM

- Ambos tipos (y las RAM en general) son volátiles, esto implica que **requieren de alimentación eléctrica para funcionar**. Si se acaba el suministro, se limpian los registros.

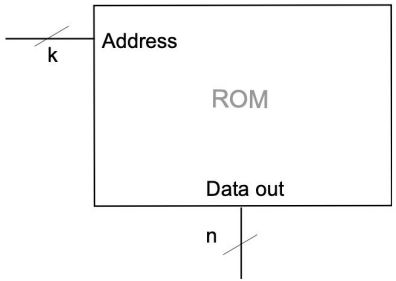
- Por temas de costo, las DRAM se usan como la memoria principal de cómputo mientras que las SRAM se usan más como memoria caché.

- La más utilizada hoy en día es la DDR SDRAM: *Double Data Rate* (almacenamiento doble al hacerse en flancos de subida y bajada) *Synchronous* (sincronizada con el *clock* del sistema) DRAM.

Tipos de memoria - ROM

Memoria de solo lectura. En la práctica, se les llama así porque se "graban una vez" sin permitir sobreescritura (o con un costo considerable). Además, son no volátiles, por lo que retienen la información a pesar de cortar su suministro eléctrico.

Con *k* bits para el bus de direccionamiento y bus de salida de *n* bits, la ROM almacena 2^k palabras de memoria de *n* bits (con valores ya establecidos).

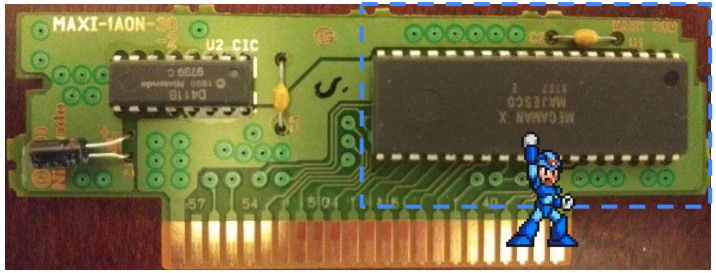


Tipos de memoria - ROM

Las ROM iniciales fueron las Mask ROM o MROM, cuyo nombre nace de la “máscara” de metal utilizada para definir el patrón de datos grabado en el chip de la componente.

Hoy en día otros tipos de ROM son más útiles al tener la conveniencia de poder modificar los datos escritos, tales como:

- PROM (*Programmable* ROM).
- EPROM (*Erasable* PROM).
- EEPROM (*Electrically Erasable* PROM).



Placa del cartucho de Super Nintendo del juego Mega Man X, incluyendo la MROM que contenían todas las placas de los juegos de esta consola.

Tipos de memoria - Memorias no volátiles

Otros tipos de memoria no volátiles:

- **HDD:** *Hard Drive Disk* o unidad de disco duro. Consiste en uno o varios discos de metal con recubrimiento magnético que hacen uso de un cabezal para modificar magnéticamente cada posición y, así, cada bit. Permite almacenar un mayor volumen de datos con menor costo monetario, pero con un considerable costo en términos de operaciones de lectura y escritura por sus movimientos mecánicos.

Tipos de memoria - Memorias no volátiles

Otros tipos de memoria no volátiles:

- **Memoria flash:** Tipo de memoria que almacena sus datos de forma eléctrica mediante compuertas `NAND` o `NOR` según la disposición de las celdas de memoria que contiene. Utilizada hoy en día para memorias USB, SD y SSD (*Solid State Drive*). Por su construcción, son más costosas que los HDD para un mismo tamaño, pero implican un menor costo en operaciones de lectura y escritura.

Tipos de memoria - Una última comparativa

Tipo de memoria	Tiempo de acceso promedio	Precio por GB promedio	Tamaño promedio
Registros	0.5 ns	$$$$$$$$$	1 MB
RAM	50 ns	5 USD (2015)	8 GB
Disco de estado sólido	25000 ns	0.05 USD (2023)	512 GB
Disco magnético	5000000 ns	0.015 USD (2023)	1 TB

Esta referencia sirve para entender por qué, por ejemplo, los computadores no funcionan solo a base de registros de memoria o RAM.

Memoria en la máquina programable - Variable

Hemos visto hasta ahora distintos componentes para almacenar datos, ¿cómo los usamos en nuestra máquina?

Cuando programamos, vamos definiendo **variables** que almacenan valores temporalmente y van siendo actualizadas según el flujo del código. Estas variables se irán almacenando en distintos componentes de memoria de nuestra máquina, pero en la próxima clase veremos cómo hacerlo.

Variables - Tipos de dato

Las variables, ya sea de forma explícita (como en TypeScript) o implícita (como en Python), poseen un **tipo de dato**. Este es un atributo que deriva en varias características del dato almacenado, tales como: codificación, tamaño, valores, operaciones posibles, etc.

Este atributo es de especial interés ya que **incide en la forma en que lo almacenamos en nuestra máquina**.

```
let variable: number = 27;
```

Identificador | Tipo de dato | Valor

Definición de variable en TypeScript

Variables - Tipos de dato de interés

Tipo de dato	Codificación	Interpretación	#Bits de representación
char **signed char**	base 2 sin signo **base 2 con signo en complemento de 2**	carácter o entero positivo **entero positivo o negativo**	8 **8**
short unsigned short	base 2 con signo en complemento de 2 base 2 sin signo	entero positivo o negativo entero positivo	16 16
int unsigned int	base 2 con signo en complemento de 2 base 2 sin signo	entero positivo o negativo entero positivo	32 32
long unsigned long	base 2 con signo en complemento de 2 base 2 sin signo	entero positivo o negativo entero positivo	64 64
long long unsigned long long	base 2 con signo en complemento de 2 base 2 sin signo	entero positivo o negativo entero positivo	128 128
float double long double	punto flotante de precisión simple punto flotante de precisión doble punto flotante de precisión cuádruple	Racionales y casos especiales Racionales y casos especiales Racionales y casos especiales	32 64 128

Este es el tipo de dato de la máquina programable del curso.

Más adelante veremos la representación de números de punto flotante.

Variables - Endianness

Supongamos que deseamos almacenar una variable cuya representación requiere de 16 bits, pero las palabras de memoria de nuestra unidad de almacenamiento son de 8 bits… ¿cómo lo almacenamos?

El ***endianness*** determina el orden en el que se almacena la secuencia de un dato.

Un ejemplo a continuación.

Variables - Endianness

Supongamos que tenemos el dato “0000111101010101” y una memoria con palabras de 1 byte. Tenemos dos opciones:

Dirección de memoria (hexa)	Palabra almacenada (*big endian*)
0x00	00001111
0x01	01010101
0x02	-

Dirección de memoria (hexa)	Palabra almacenada (*little endian*)
0x00	01010101
0x01	00001111
0x02	-

Si la palabra más significativa se almacena en la dirección menor, el almacenamiento es del tipo ***big endian***; si la palabra menos significativa se almacena en la dirección menor, el almacenamiento es del tipo ***little endian***.

Memoria en la máquina programable - Arreglos

Un caso particular de las variables es que pueden ser definidas como **arreglos**. En este caso, ¿cómo podemos manejarlos en memoria?

Adicional a las características de una variable (tipo de dato, *endianness*), contamos con un **largo** y una **dirección de memoria de inicio**. Con estos atributos podemos **indexar** cada elemento para acceder a ellos en nuestra memoria.

A continuación, un ejemplo.

Arreglos - Almacenamiento

Supongamos que tenemos el siguiente arreglo definido (asumamos que cada número ocupa 1 byte).

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000011
0x02	00000101
0x03	00000111
0x04	-

```
let variables: number[] = [1, 3, 5, 7];
```

En este caso, la dirección de memoria de inicio sería **0x00** y debemos incrementar en una unidad este valor para acceder a cada elemento del arreglo.

La cantidad de unidades que sumamos a la dirección de memoria inicial **depende del tamaño del dato almacenado**.

Matrices - Almacenamiento

¿Y para arreglos de dos dimensiones, *i.e.* matrices?
Podemos almacenar por convención de **filas** o **columnas.**

```
let variablesMatrix: number[][] = [
  [1, 3, 5],
  [2, 4, 6]
];
```

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000011
0x02	00000101
0x03	00000010
0x04	00000100
0x05	00000110

Convención de filas

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000010
0x02	00000011
0x03	00000100
0x04	00000101
0x05	00000110

Convención de columnas

Matrices - Direccionamiento

Tenemos distintas fórmulas de direccionamiento según la convención utilizada:

Convención por filas

$dir(matriz[i,j]) = dir(\text{matriz}) + i * sizeof(matriz[i,j]) * \#columnas + j * sizeof(matriz[i,j])$

Convención por columnas

$dir(matriz[i,j]) = dir(\text{matriz}) + j * sizeof(matriz[i,j]) * \#filas + i * sizeof(matriz[i,j])$

La cantidad de columnas o de filas representa el largo de cada subarreglo de la matriz, por lo que necesitamos considerarlos para un direccionamiento efectivo.

Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.

Ejercicios

Construya el circuito del latch RS haciendo uso exclusivo de compuertas `OR` y `NOT`.

Ejercicios

Diseñe, utilizando todos los elementos de circuitos lógicos vistos en clases, un contador secuencial de 2 bits que se incrementa con cada flanco de subida de la señal de control.

Ejercicios

Implemente mediante compuertas lógicas, elementos de control y latches, un flip-flop tipo D que funcione con flanco de bajada.

Ejercicios

¿Cuántas direcciones tiene una memoria RAM de 4.5 KB que utiliza palabras de 3 bytes? (1 KB = 1024 bytes).

Ejercicios

Suponga que trabaja con un computador que utiliza datos de 2 bytes y que posee una memoria con palabras de 1 byte de almacenamiento. Describa, en este contexto, un mecanismo para determinar el *endianness* utilizado por el computador para el almacenamiento de datos.

Ejercicios

Suponga que se tiene una matriz almacenada en la dirección de memoria 0x0A. Esta posee un total de 4 filas y 5 columnas. Si se sabe que en una dirección de memoria se puede almacenar 1 byte, y la matriz almacena en cada celda un dato de 2 bytes, ¿cuál es la dirección del dato que se encuentra en la tercera columna de la segunda fila de la matriz? Asuma que se utiliza la convención de filas.

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 3 - Almacenamiento de Datos

Profesor: Germán Leandro Contreras Sagredo

Anexo - Videos complementarios

Por si desean profundizar más en los temas de esta clase.

- Video interactivo de almacenamiento de 1 byte con condensadores.
- Resumen de las DRAM; funcionamiento de las celdas de memoria en una DRAM; ciclos de lectura y escritura en una DRAM.
- Funcionamiento DDR.
- Funcionamiento HDD; funcionamiento memoria flash.
- Bonus: Videos que tocan conceptos del curso con NES y SNES.

Anexo - Resolución de ejercicios

¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.

Ejercicios - Respuesta

Modifique el diagrama interno del registro de memoria de 4 bits para que incluya las señales *Load* y *Reset*.

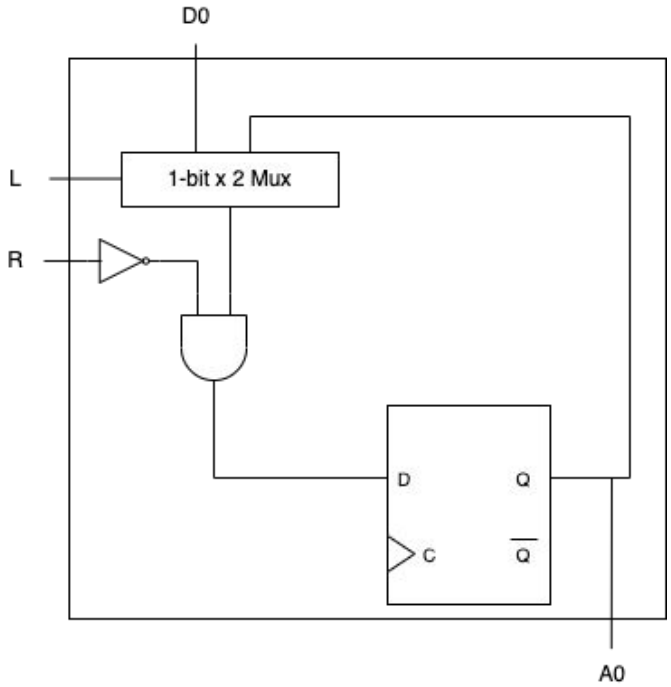
En este caso, solo realizamos la modificación del bit menos significativo dado que esta se expande de la misma forma para el resto de bits.

Circuito y explicación en la siguiente diapositiva.

Ejercicios - Respuesta

Si $R = 0$, entonces el valor a almacenar será el que provenga del Mux de dos entradas, pero si $R = 1$, se almacenará un 0 dado que será la salida estándar de la compuerta `AND`.

L se utiliza como el selector del Mux. Si $L = 0$, podemos definir que se seleccione el estado Q del bit del registro y que se escoja el bit de entrada si $L = 1$.



Ejercicios - Respuesta

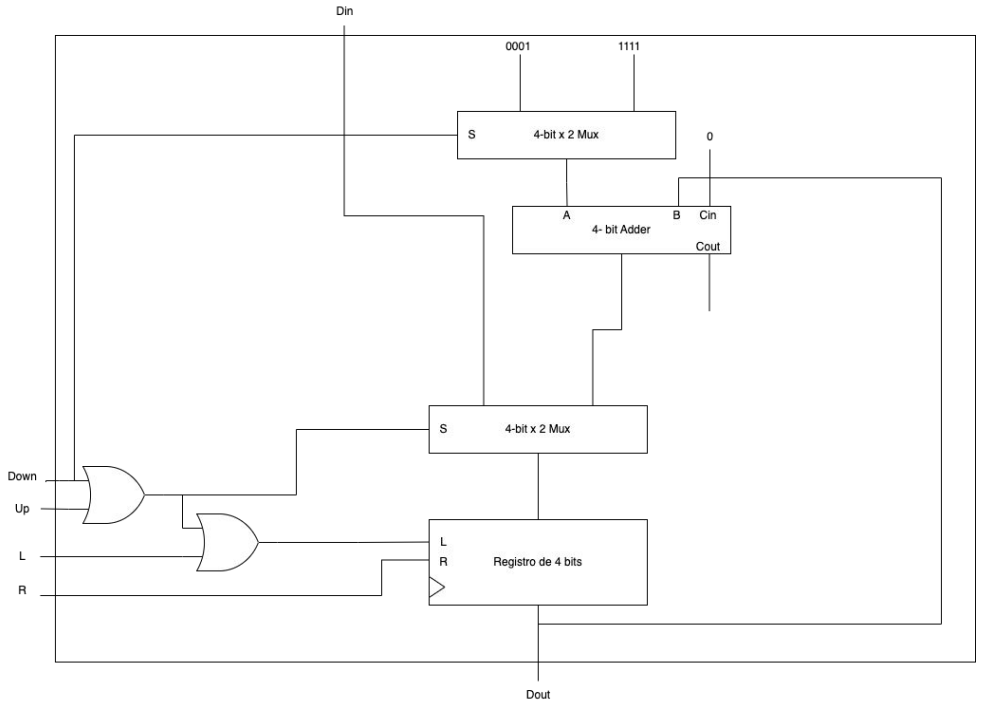
Modifique el diagrama interno del registro de memoria con *Load* y *Reset* de 4 bits para que incluya las señales *Up* y *Down*.

En este caso, nos basaremos plenamente en la construcción del registro de memoria con *Load* y *Reset* (y no a nivel de bit) para llevar a cabo la modificación.

Respuesta en la siguiente diapositiva.

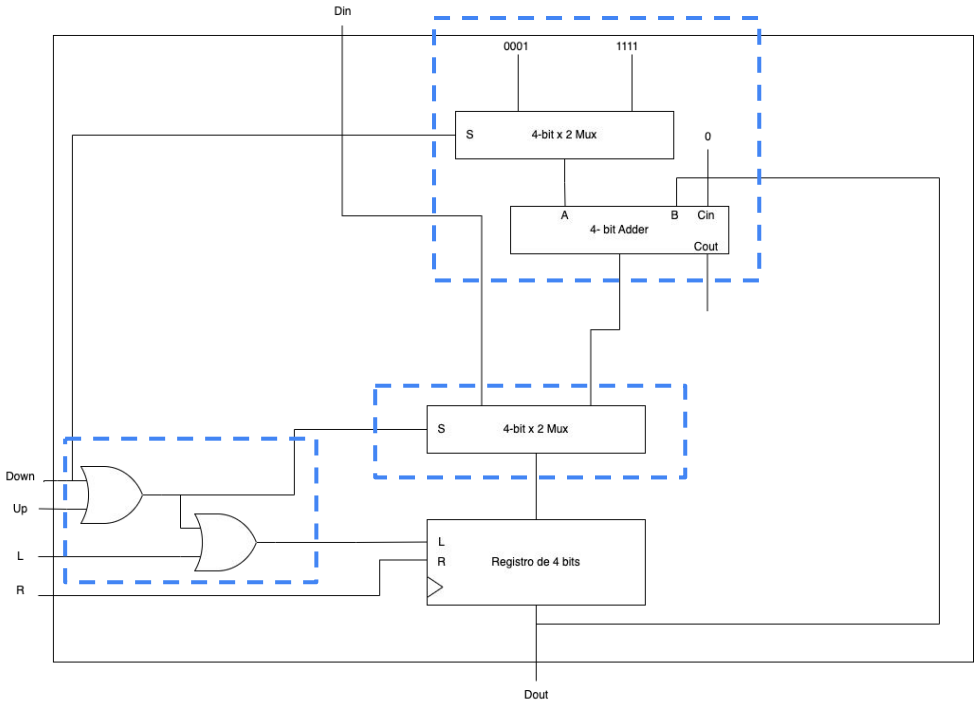
Ejercicios - Respuesta

En primer lugar, solo se lleva a cabo la carga sobre el registro si alguna de las siguientes señales está activa: *Load*, *Up*, *Down*. Por otra parte, la señal *R* sigue teniendo el mismo comportamiento.



Ejercicios - Respuesta

Si las señales *Up* o *Down* están activas, se seleccionará como ingreso de registro el resultado de la suma entre el estado previo y uno de dos valores: 0001 (1) o 1111 (complemento de 2 de 1). La selección de este número se realiza con la señal *Down*. Si ninguna de estas está activa, se selecciona el dato de entrada original (*Din*) como el ingreso del registro interno.



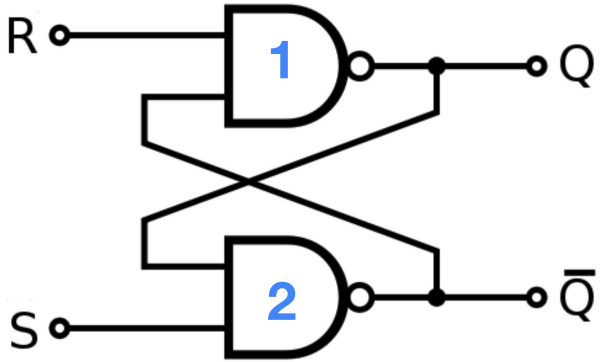
Ejercicios - Respuesta

Construya el circuito del latch RS haciendo uso exclusivo de compuertas OR y NOT.

La clave de este ejercicio es observar el circuito del latch RS con compuertas NAND y verlo según las fórmulas lógicas que lo definen.

NAND 1: R NAND ¬Q = NOT(R AND ¬Q)

NAND 2: S NAND Q = NOT(S AND Q)



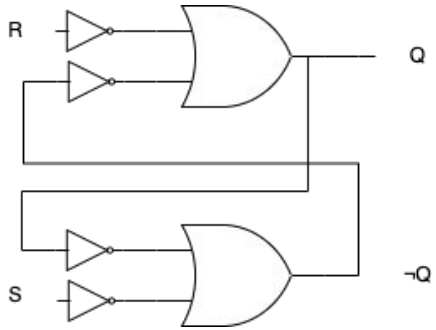
Ejercicios - Respuesta

Si aplicamos **Ley de Morgan:**

NAND 1: R NAND ¬Q = NOT(R AND ¬Q) = **¬R OR Q**

NAND 2: S NAND Q = NOT(S AND Q) = **¬S OR ¬Q**

Por lo tanto, basta con intercambiar las compuertas NAND por las combinaciones resultantes.



Ejercicios - Respuesta

Diseñe, utilizando todos los elementos de circuitos lógicos vistos en clases, un contador secuencial de 2 bits que se incrementa con cada flanco de subida de la señal de control.

Aquí lo esencial es ver cómo es la secuencia de números del contador y, en base a eso, definir la composición de cada bit.

La secuencia es: 00-01-10-11-00-01-10-11-00-… (hasta el infinito).

Ejercicios - Respuesta

Se puede observar que el bit menos significativo siempre alterna su valor entre 0 y 1 por cada iteración (*i.e.* por cada flanco de subida). Basta con que la entrada del flip-flop que almacena este dato sea su estado negado.

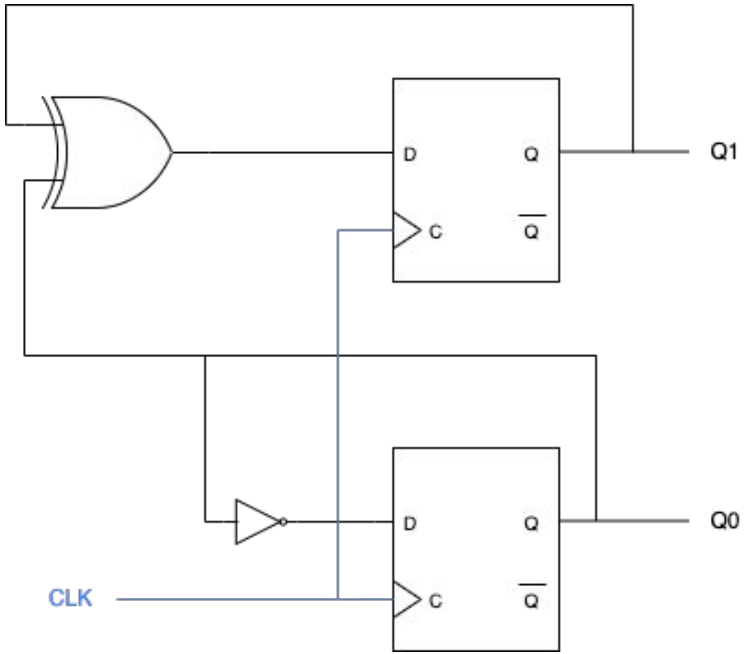
Para el bit más significativo, veamos la tabla de verdad respecto al número anterior de la secuencia.

$Q(t)\_1$	$Q(t)\_0$	$Q(t+1)\_1$
0	0	0
0	1	1
1	0	1
1	1	0

Vemos que hay una compuerta precisa para representar este comportamiento: **XOR**.

Ejercicios - Respuesta

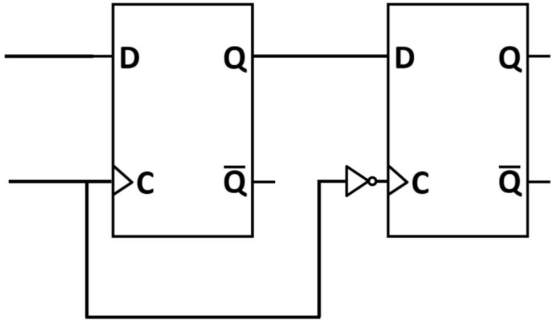
Finalmente, nuestro circuito quedará de la siguiente manera.



Ejercicios - Respuesta

Implemente mediante compuertas lógicas, elementos de control y latches, un flip-flop tipo D que funcione con flanco de bajada.

Simplemente tomamos como referencia la construcción con latches D en flanco de subida e invertimos el uso de la señal de control.



Ejercicios - Respuesta

¿Cuántas direcciones tiene una memoria RAM de 4.5 KB que utiliza palabras de 3 bytes? (1 KB = 1024 bytes).

Sabemos que: Tamaño RAM = #Direcciones * Tamaño de palabra

Entonces, simplemente reemplazamos.

4.5 KB = #Direcciones * 3B
4.5 * 1024 B = # Direcciones * 3B
 # Direcciones = 1536

Ejercicios - Respuesta

Suponga que trabaja con un computador que utiliza datos de 2 bytes y que posee una memoria con palabras de 1 byte de almacenamiento. Describa, en este contexto, un mecanismo para determinar el *endianness* utilizado por el computador para el almacenamiento de datos.

Basta con almacenar el número 1 en memoria. Este, en hexadecimal, es equivalente a 0x01. Por el tamaño de memoria, una de las direcciones tendrá el valor 0x0 mientras que la otra tendrá el valor 0x01. Al leer la primer dirección de memoria de nuestro dato, podremos determinar su *endianness* si es que leemos un 1 o un 0.

Ejercicios

Suponga que se tiene una matriz almacenada en la dirección de memoria 0x0A. Esta posee un total de 4 filas y 5 columnas. Si se sabe que en una dirección de memoria se puede almacenar 1 byte, y la matriz almacena en cada celda un dato de 2 bytes, ¿cuál es la dirección del dato que se encuentra en la tercera columna de la segunda fila de la matriz? Asuma que se utiliza la convención de filas.

Haremos uso directo de la fórmula en la siguiente diapositiva para determinarlo.

Ejercicios

dir(*matriz*[*i*,*j*]) = *dir*(matriz) + *i* * *sizeof*(*matriz*[*i*,*j*]) * #*columnas* + *j* * *sizeof*(*matriz*[*i*,*j*])

i = segunda fila = 1; *j* = tercera columna = 2; #*columnas* = 5; *sizeof*(*matriz*[*i*,*j*]) = 2

dir(*matriz*[1,2]) = 0x0A + 1 * 2 * 5 + 2 * 2 = 0x0A + 0x0E = 0x18

Como cada dato ocupa 2 bytes y tenemos una memoria de palabras de almacenamiento de 1 byte, las direcciones 0x18 y 0x19 corresponden al dato buscado.