# Microarquitectura 02: Sistemas secuenciales, FSM

## Organización del computador - FIUBA

2.$^{do}$ cuatrimestre de 2023

Última modificación: Sun Sep 17 18:36:10 2023 -0300

# Créditos

Para armar las presentaciones del curso utilizamos:


R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.


D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.


J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

# Tabla de contenidos

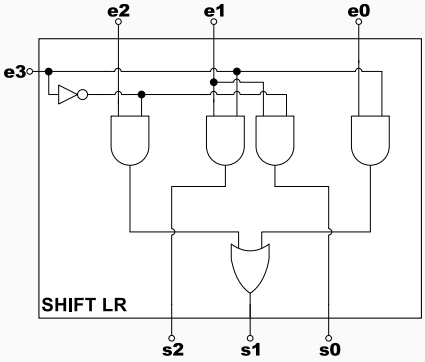
# Sobre la clase de hoy

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

- Repaso de sistemas combinatorios
- Retroalimentación y cambio de modelo
- Sistemas secuenciales asincrónicos
- Sistemas secuenciales sincrónicos
- Flip-flops, registros y memorias
- Máquinas de estado

# Timing

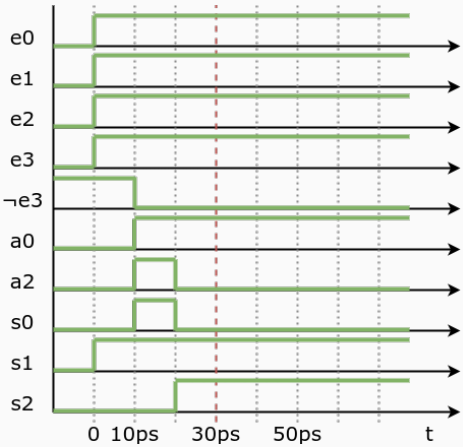
Recordemos que **las compuertas no son instantáneas.**



# Timing

Cada componente discreto va a tener un retardo característico que representa **la cantidad tiempo transcurrido entre que llegaron a estabilizarse sus salidas respecto del primer instante en que se estabilizaron las entradas**.

# Timing



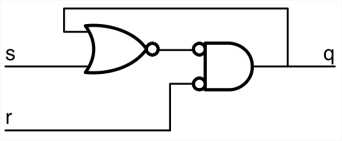
# Sistemas combinatorios

Ahora volviendo un poco a los sistemas combinatorios, hay dos motivos fundamentales por los cuales podemos representar su comportamiento como una tabla que asigna valores a las salidas en base a los valores de las entradas, y tienen que ver con que:

1. **los tiempos de propagación** quedan fuera de nuestro modelo bajo el principio de abstracción.
2. no hay *retroalimentación de señales*.

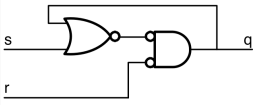
# Retroalimentación

Ahora supongamos que queremos construir un sistema que permita conservar un bit de información. Básicamente queremos poder indicar cuando el bit vale 1 (activando la señal **s**et) y cuando vale 0 (activando la señal **r**eset), y proponemos la siguiente configuración:




¿Qué valor tiene $q$ para cada par $(s, r)$, cuándo se estabiliza?

# Retroalimentación

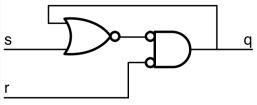



Intentemos armar su tabla de verdad.

| $s$ | $r$ | $q$ |
|---|---|---|
| 0 | 0 | ? |
| 0 | 1 | ? |
| 1 | 0 | ? |
| 1 | 1 | ? |

¿El valor de $q$ es el actual o el estabilizado?

# Retroalimentación




Definamos cómo se estabilizará ($q'$) en base a su valor actual ($q$).

| $s$ | $r$ | $q$ | $q'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Retroalimentación

Para poder razonar sobre el comportamiento del sistema debemos representar todas las señales, **de entrada ($s$,$r$), de salida ($q'$) e internas o retroalimentadas ($q$)**.

| $s$ | $r$ | $q$ | $q'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Retroalimentación

Vamos a llamar **valuación** a cada asignación de valores de verdad a las variables del sistema y representan todos los estados posibles que puede tener el sistema. En nuestro caso las variables del sistema son $s$, $r$ y $q$, ya que $q'$ está indicando los valores que puede tomar $q$ en el próximo estado.

| $s$ | $r$ | $q$ | $q'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Retroalimentación

Al introducir la retroalimentación y junto con ella la dependencia de señales internas, el sistema no puede pasar de una valuación a cualquiera otra, por ejemplo:

| | Estado | | | |
|---|---|---|---|---|
| | $s$ | $r$ | $q$ | $q'$ |
| 1: | 0 | 0 | 0 | 0 |
| 2: | 0 | 0 | 1 | 1 |

No podemos pasar del estado indicado en la primera fila $\langle 000 \rangle$ al estado indicado en la segunda $\langle 001 \rangle$ porque el valor actual de la salida ($q$) depende del valor que tenía en el estado anterior ($q'$).

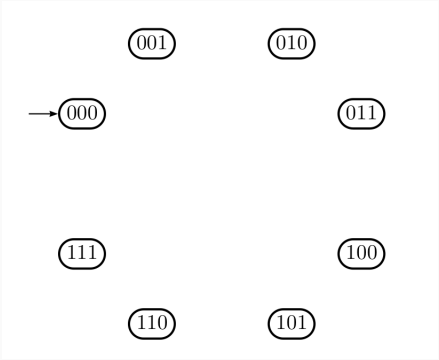
# Retroalimentación

Intentemos modelar el comportamiento del sistema como los posibles cambios en su conjunto de señales.

| $s$ | $r$ | $q$ | $q'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

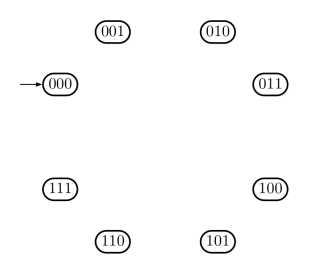
# Retroalimentación

Representemos las distintas valuaciones.



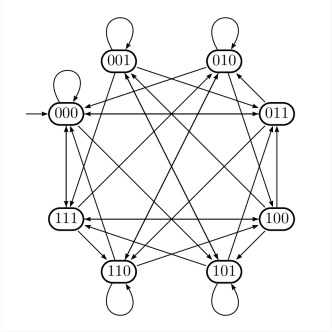
# Retroalimentación

¿Cuáles son los cambios posibles de valores en las variables del sistema?



# Retroalimentación

¿Cuáles son los cambios posibles de valores en las variables del sistema?




Esto es lo que se conoce como una **estructura de Kripke**.

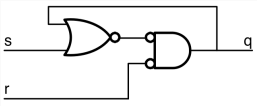
# Retroalimentación

Se trata de un tipo de modelo llamado **máquina de estados** que permite representar, razonar y probar propiedades sobre el comportamiento de objetos de dominio heterogéneo, entre ellos:

- circuitos digitales
- protocolos de comunicación
- interfaces de programación de aplicaciones
- ejecuciones de un programa

# Retroalimentación

Repasando, queríamos implementar un sistema que pudiese almacenar un bit de información.




| $s$ | $r$ | $q$ | $q'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Retroalimentación

Al retroalimentar una de las señales tuvimos que:

- abandonar el enfoque funcional para describir al sistema
- desdoblar los valores de la salida $q$ de su valor previo $q'$
- introducir la noción de **valuación, estado** y **comportamiento**
- presentar informalmente la idea de **máquinas de estados**

# Retroalimentación

Recordemos que había dos motivos por los cuales podíamos dar una visión funcional (que se describe completamente en base a sus valores de entrada y de salida) de los sistemas combinatorios:

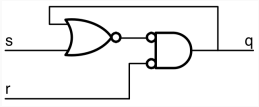
1. **los tiempos de propagación** quedan fuera de nuestro modelo bajo el principio de abstracción
2. y no hay *retroalimentación de señales* en nuestros sistemas

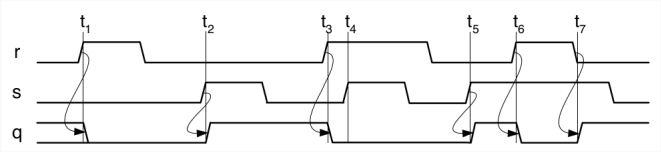
Pero hasta ahora sólo observamos los problemas relacionados con la retrolimentación de señales.

**¿Qué sucede cuando queremos razonar sobre sistemas secuenciales que se componen entre sí?**

# Sincronización

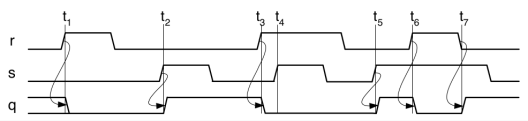
El ejemplo presentado anteriormente es suficientemente simple como para poder evitar razonar sobre el orden de las señales.



# Sincronización

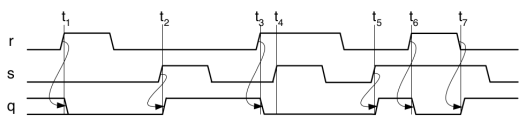
No hay garantías sobre el momento en el cuál las señales van a cambiar de valor.




Esto implica que no hay garantías sobre el momento en el cuál un componente cambia de estado (valuación).

# Sincronización

No hay garantías sobre el momento en el cuál las señales van a cambiar de valor.




Necesitamos asegurar (idealmente) que el tiempo entre cambios de estado sea mayor que el peor tiempo de propagación de nuestros componentes.

# Sincronización

Ya no podemos pensar que eventualmente las señales se estabilizan porque **deseamos que puedan cambiar a lo largo del tiempo**.

La retroalimentación así como la vimos introduce el siguiente problema:

- Tenemos que razonar sobre el orden y tiempo de propagación de las señales para poder saber cuándo una señal cambia de valor en la salida.
- Estos tiempos de propagación dependen de la cantidad de componentes y configuración interna de cada componente.

# Sincronización

Ya no podemos pensar que eventualmente las señales se estabilizan porque **deseamos que puedan cambiar a lo largo del tiempo**.

La retroalimentación así como la vimos introduce el siguiente problema:

- Se rompe el **principio de encapsulamiento** que nos permitía razonar sobre un componente sólo a través de su interfaz (entradas/salidas).
- ¿Por qué? Por que sin conocer su configuración **no podemos conocer su estado** (señales internas).

# Sincronización

La necesidad que tenemos ahora es la de:

**Diseñar un mecanismo que nos permita poder razonar con certeza sobre el estado de todos los componentes del sistema.**

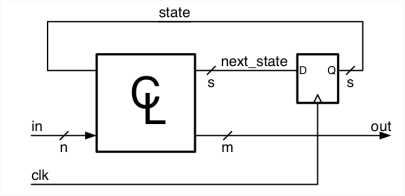
# Sincronización

La propuesta es:

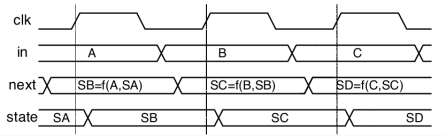
Introducir una señal especial (**clock**) que determina el intervalo de tiempo dentro del cuál los valores de salida (**siguiente estado**) se actualizan en base a los valores de entrada (**estado actual**).

# Sincronización

Veamos un ejemplo:




El clock determina en qué momento el estado siguiente pasa a ser el actual:



# Sincronización

A partir de ahora las nociones de **estado** y de sincronización a través de una señal de **clock** van a ser elementos fundamentales de los diseños de nuestros sistemas secuenciales.

A continuación vamos a ver en detalle y en orden de complejidad creciente, **los elementos fundacionales que nos permiten construir y componer sistemas secuenciales sincrónicos**.

# Sincronización

Estos sistemas son de la mayor relevancia para nuestor objetivo, que era diseñar un procesador, ya que precisamos que la información pueda transformarse a lo largo del tiempo.

Para poder razonar y especificar el comportamiento de estos componentes **vamos a estudiar los formalismos de máquinas de estado más usados**, ya que nos permiten modelar el problema con un grado de abstracción suficientemente adecuado.

# Conclusión de la primera parte

Hasta ahora vimos:

- Repaso de sistemas combinatorios
- Retroalimentación y cambio de modelo
- Sistemas secuenciales asincrónicos (introducción)
- Sistemas secuenciales sincrónicos (introducción)

Falta ver:

- Flip-flops, registros y memorias
- Máquinas de estado (en detalle)

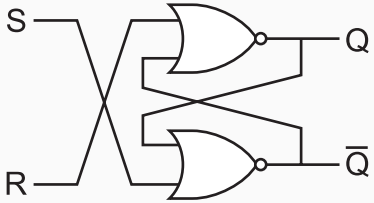
¿Preguntas hasta acá?

# Latchs

Son sistemas 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:

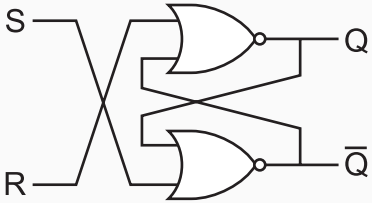



Tabla de verdad:

| $S$ | $R$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|
| 1 | 0 | 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:

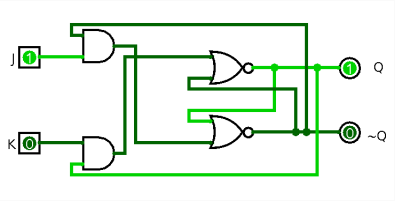



Tabla de verdad:

| $J$ | $K$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|
| 1 | 0 | 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 sistema oscila (estado inestable).

# Latch D

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

Latch D:

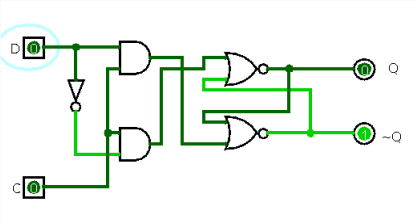



Tabla de verdad:

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

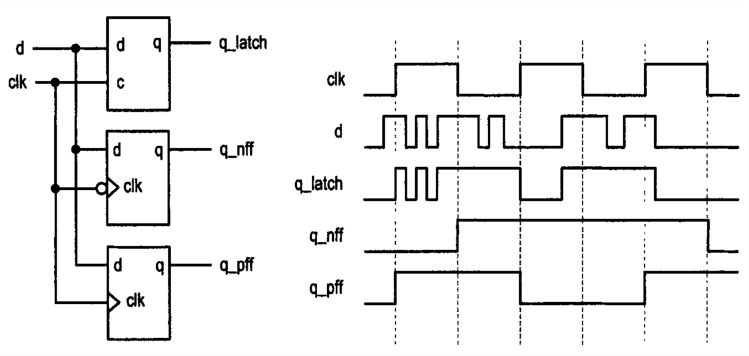
En este caso el sistema es estable en todos los estados. Sin embargo:

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

# Sincronizando...

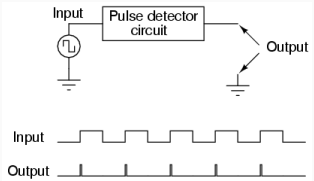
Como vimos en la primer parte, nos interesa poder tener un control de los momentos de transición de estados ⇒ **CLOCK**

Ser reactivo al nivel de una señal no es conveniente, ¿por qué? ⇒ **Sensibilidad al flanco**

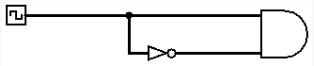


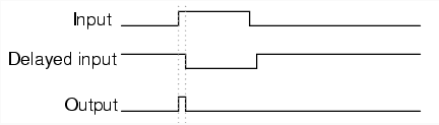
# Detector de flanco

Necesitamos un sistema 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:

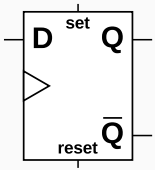



Tabla de verdad:

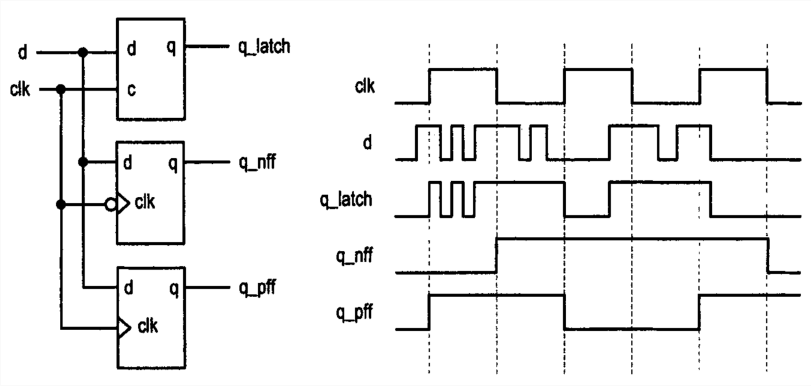
| $D$ | $clk$ | $Q_{T+1}$ | $\overline{Q_{T+1}}$ |
|---|---|---|---|
| 1 | 0 | $Q_T$ | $\overline{Q_T}$ |
| 0 | 1↑ | 0 | 1 |
| 0 | 0 | $Q_T$ | $\overline{Q_T}$ |
| 1 | 1↑ | 1 | 0 |

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:

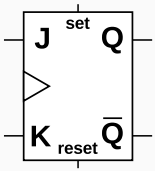



Tabla de verdad:

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

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

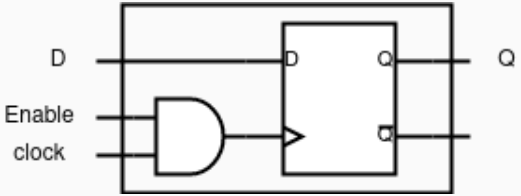
*Se niega el valor anterior cada 1 colck*

# Registros

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

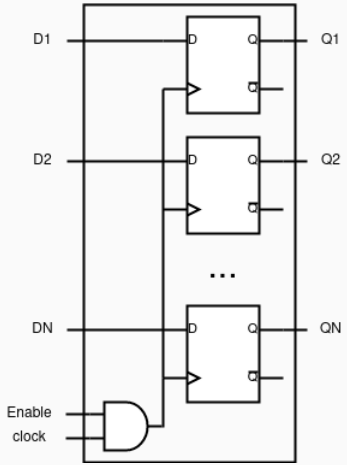
- Debemos poder elegir con una entrada adicional de control por cuanto tiempo queremos almacenar $\Rightarrow$ **enable**.

¡Sencillo!:



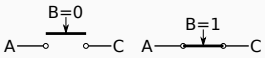
# Registro de N-bits

Podemos componer la solución anterior para poder almacenar N bits:

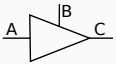


# Componentes de Tres Estados

## Noción Eléctrica




## Símbolo

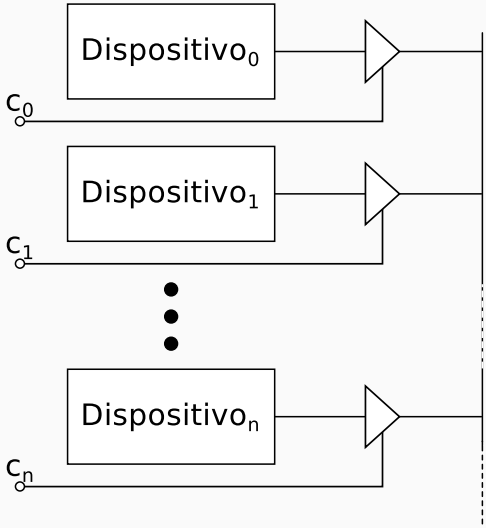



## Tabla de Verdad

| 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 sistema.

# Componentes de Tres Estados



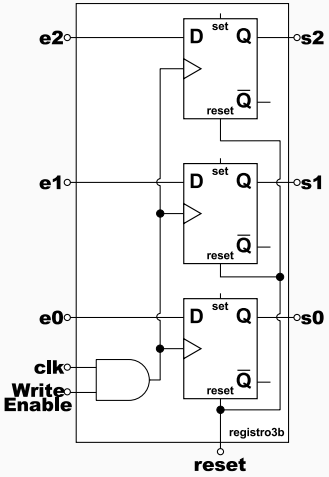
# Ejercicio 0

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.

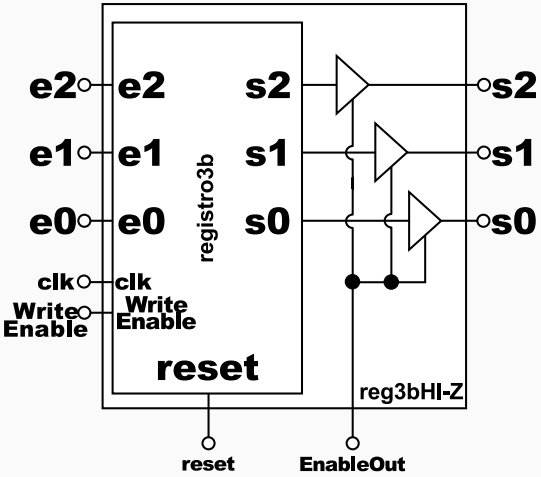
b) Modificar el diseño anterior agregándole componentes de 3 estados para que sólo cuando se active la señal de control ENABLEOUT muestre el dato almacenado.

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

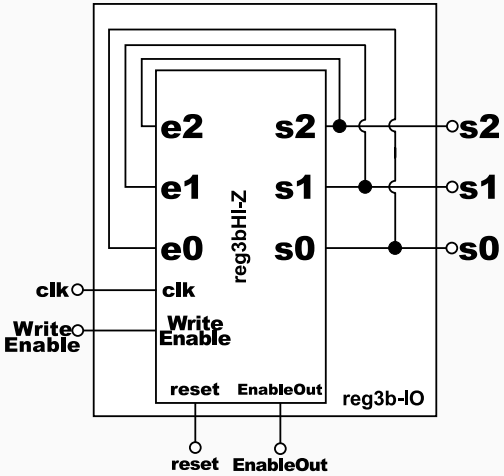
# Solución - Ejercicio 0.a



# Solución - Ejercicio 0.b

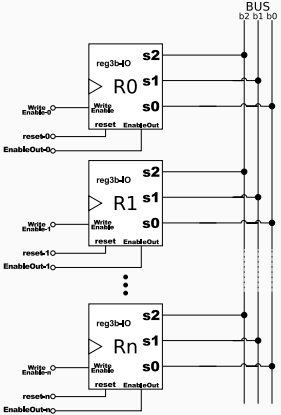


# Solución - Ejercicio 0.c



# Ejercicio 1

a) Realizar el esquema de interconexión de n registros como el diseñado
b) Dar una secuencia de valores de las señales de control para que se copie el dato del R1 al R0




Señales de control:

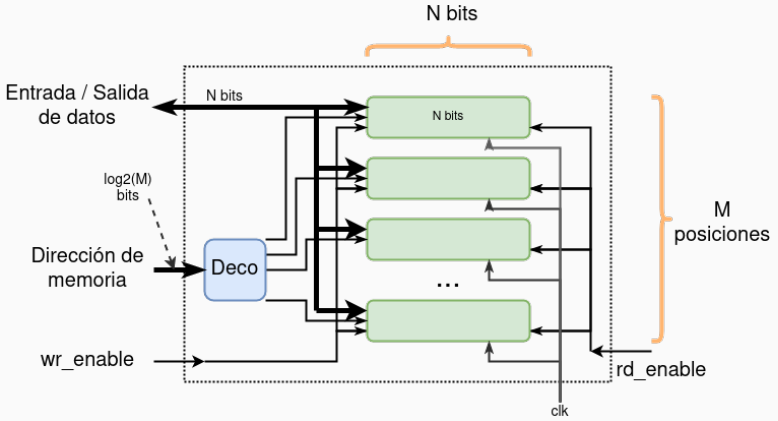
| R0 | R1 | ... | Rn |
|---|---|---|---|
| WriteEnable-0 | WriteEnable-1 | ... | WriteEnable-n |
| reset-0 | reset-1 | ... | reset-n |
| EnableOut-0 | EnableOut-1 | ... | EnableOut-n |

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

- EnableOut-1 ⟵ 1
- WriteEnable-0 ⟵ 1
- ...clk....
- WriteEnable-0 ⟵ 0
- EnableOut-1 ⟵ 0

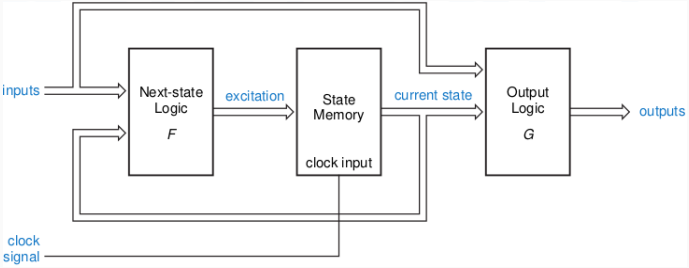
# Memorias (intro)

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



# Modelo general de un sistema secuencial

Conceptualmente podemos pensar a un secuencial como:




Compuesto por tres bloques principales:

- **Lógica de próximo estado**: $f(estado_{actual}, entradas)$
- **Registro (o memoria) de estado**: $f(estado_{próximo}, clk)$
- **Lógica de salida**: $f(estado_{actual}, entradas)$

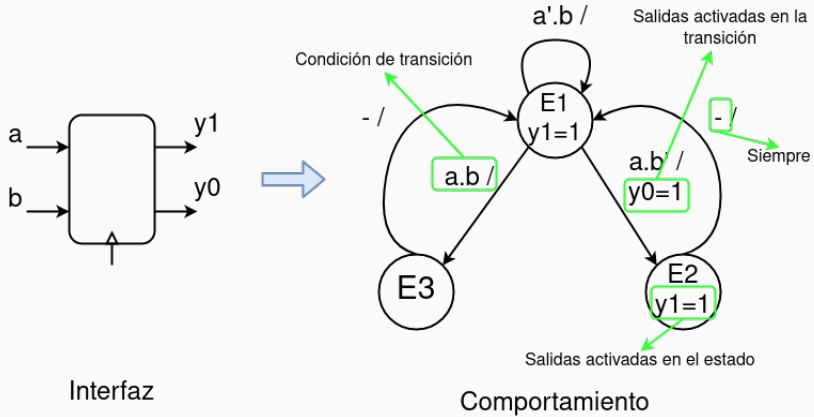
# Máquinas de estado

- Los sistemas secuenciales pueden ser pensados formalmente como una *Maquina de Estados Finitos* o *FSM*
- Las FSM son el siguiente nivel en cuanto a capacidad de computo luego de la lógica combinacional.

Una máquina de estados queda definida por:

- Una lista de estados
- Un estado inicial
- Una lista de funciones disparan las transiciones en función de las entradas

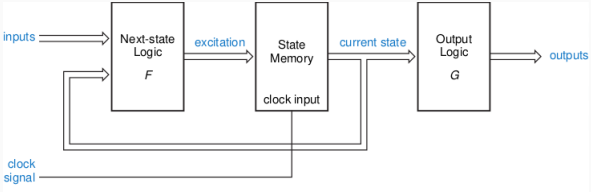
# Diagramas de estado




Todas las salidas en ’0’ salvo que se explicite lo contrario

# FSM - Moore

Si la salida depende sólo del estado actual la llameremos como *FSM de Moore*:

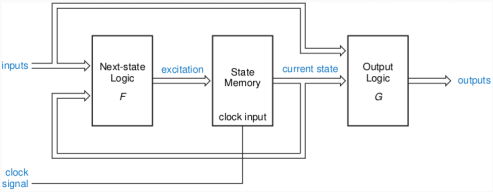



Características:

- La salida siempre cambia un clock después que se dispara la condición de transición
- No produce *glitches* a la salida
- La cantidad de estados para reproducir cierto comportamiento puede ser más grande que con otro tipo de FSM.

# FSM - Mealy

Si la salida depende tanto del estado actual cómo de las entradas, la llameremos como *FSM de Mealy*:



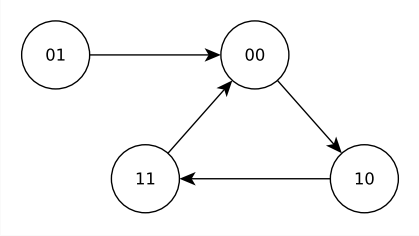

Características:

- La salida puede cambiar dentro del mismo clock en que se dispara la condición de transición
- Produce *glitches* a la salida
- La cantidad de estados para reproducir cierto comportamiento generalmente es más chica que en *Moore*

# Ejercicio 1

Implementar una FSM en base a un registro de dos bits que siga los siguientes estados y que cada cambio se produzca al apretar un pulsador. Usando flip-flops D y compuertas básicas a elección.

Nos piden además que el componente a desarrollar cuente con una entrada de `Reset`.



# Solución - Ejercicio 1

En este caso, dado un estado t definido por el valor de $Q_1$ y $Q_0$ podemos ver cuáles serán los próximos valores a almacenar:

| $Q_1(t)$ | $Q_0(t)$ | $Q_1(t+1)$ | $Q_0(t+1)$ |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

>Qué valores deberían tener $D_1$ y $D_0$ para obtener los valores deseados en el tiempo t+1, es decir, de $Q_1(t+1)$ y $Q_0(t+1)$?
Esto es, la **lógica de próximo estado**

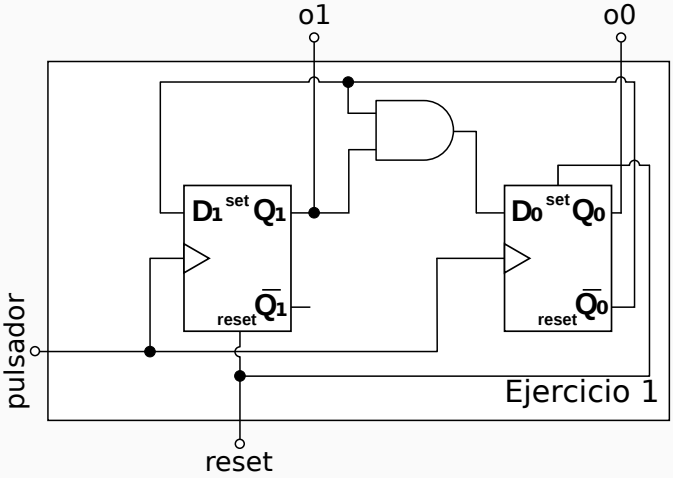
# Solución - Ejercicio 1

Usando que el flip-flop D define su próximo valor en referencia a lo que tiene en la entrada D, vemos que la suma de productos nos define los valores de D:

$$
\begin{aligned}
D_0 &= (Q_1.\bar{Q_0}) \\
D_1 &= (\bar{Q_1}.\bar{Q_0}) + (Q_1.\bar{Q_0}) \\
&= (\bar{Q_1} + Q_1).\bar{Q_0} \\
&= 1.\bar{Q_0} \\
&= \bar{Q_0}
\end{aligned}
$$

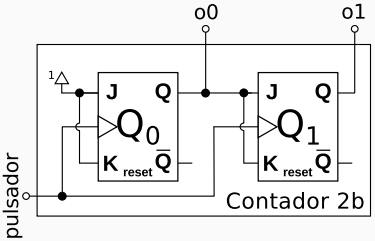
# Solución - Ejercicio 1

Así se obtiene el siguiente sistema:



# Ejercicio 2

Analizar los estados del siguiente componente:

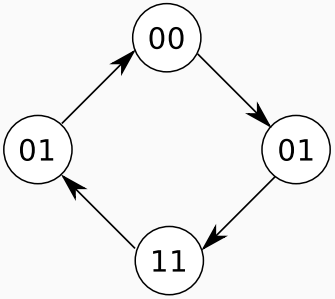



Solución:

| $Q_1(t)$ | $Q_0(t)$ | $Q_1(t+1)$ | $Q_0(t+1)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

# Ejercicio 3

Implementar una FSM en base a ub registro de dos bits que siga los siguientes estados y que cada cambio se produzca al apretar un pulsador.



# Solución - Ejercicio 3

Realizando un análisis análogo al del ejercicio anterior se obtiene:

| $Q_1(t)$ | $Q_0(t)$ | $Q_1(t+1)$ | $Q_0(t+1)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | ? | ? |
| 1 | 0 | - | - |
| 1 | 1 | 0 | 1 |

Lo cual no parece funcionar, ya que para el 01 no se puede determinar si es 11 ó 00 y para 10 no hay definido un próximo estado.

# Solución - Ejercicio 3

En este caso, necesitamos introducir la **lógica de salida**.

**¡No debemos confundir la etiqueta del estado con el valor de las salidas!**

Renombremos a los estados con nombres únicos, por ejemplo:
$S_1 = 00, \quad S_2 = 01, \quad S_3 = 10, \quad S_4 = 11$

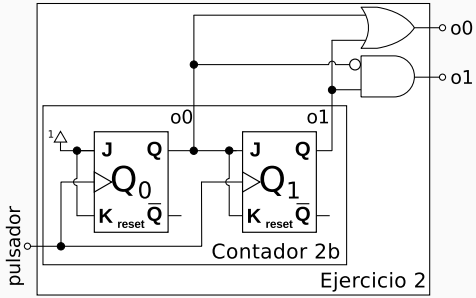
| $Q_1$ | $Q_0 \rightarrow o_1$ | $o_0$ |
|---|---|---|
| 0 | $0 \rightarrow 0$ | 0 |
| 0 | $1 \rightarrow 0$ | 1 |
| 1 | $0 \rightarrow 1$ | 1 |
| 1 | $1 \rightarrow 0$ | 1 |

# Solución - Ejercicio 3

Con lo cual podemos decir que:

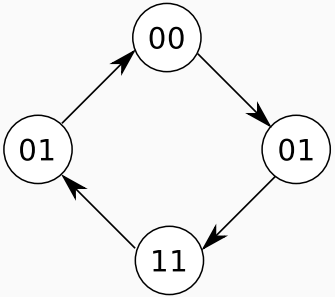
$$o_0 = Q_1 + Q_0 \quad \text{por producto de sumas}$$

$$o_1 = Q_1 \,.\, \bar{Q_0} \quad \text{por suma de productos}$$

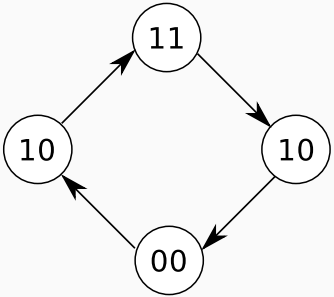


# Ejercicio 3- bis

Implementar una FSM en base a un registro de dos bits que siga los siguientes estados y que cada cambio se produzca al apretar un pulsador. Con el agregado de que tenga una entrada llamada NEG que genera los siguientes comportamientos:



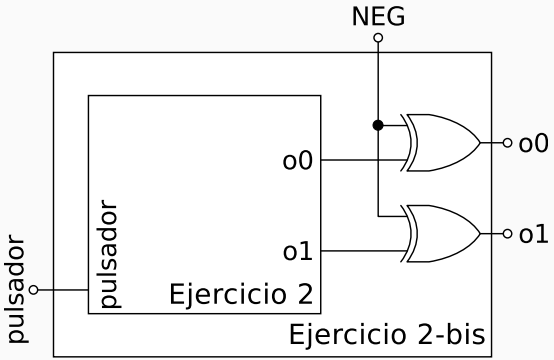

Si NEG vale 0




Si NEG vale 1

# Solución - Ejercicio 3

En este caso transformamos la máquina en una **FSM de Mealy**:



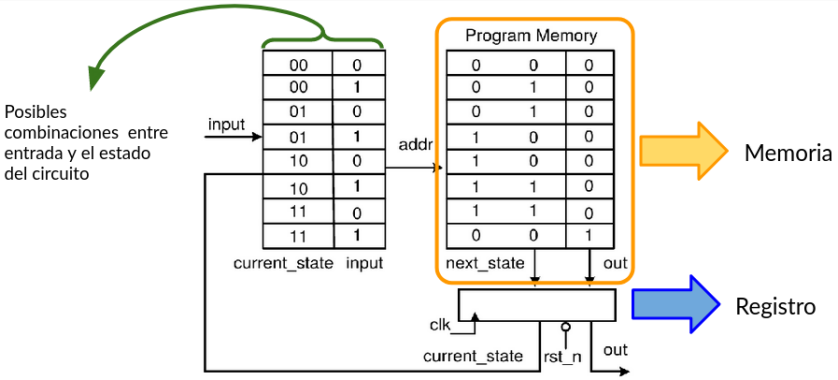
# FSM Microprogramada

Ejercicio para pensar:

- Diseñar un sistema sincrónico con una entrada de un bit y una salida de un bit que detecte la ocurrencia de 4 bits en ‘1’ a la entrada. El mismo pondrá un ‘1’ en su salida durante un ciclo de reloj luego de contar cuatro ‘1’s en su entrada. El ciclo se repite indefinidamente.
- Se puede utilizar un registro de 3 bits y una “Memoria” de 3 bits y 8 posiciones.
- ¿Cómo se cambiaría el comportamiento del sistema sin cambiar el hardware?

# FSM Microprogramada

## Resolución



# Conclusiones

- Estudiamos la realimentación en sistemas 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
- Estudiamos con cierta formalidad el comportamiento de sistemas secuenciales en general

# Preguntas

# ¡Gracias!

# Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`