



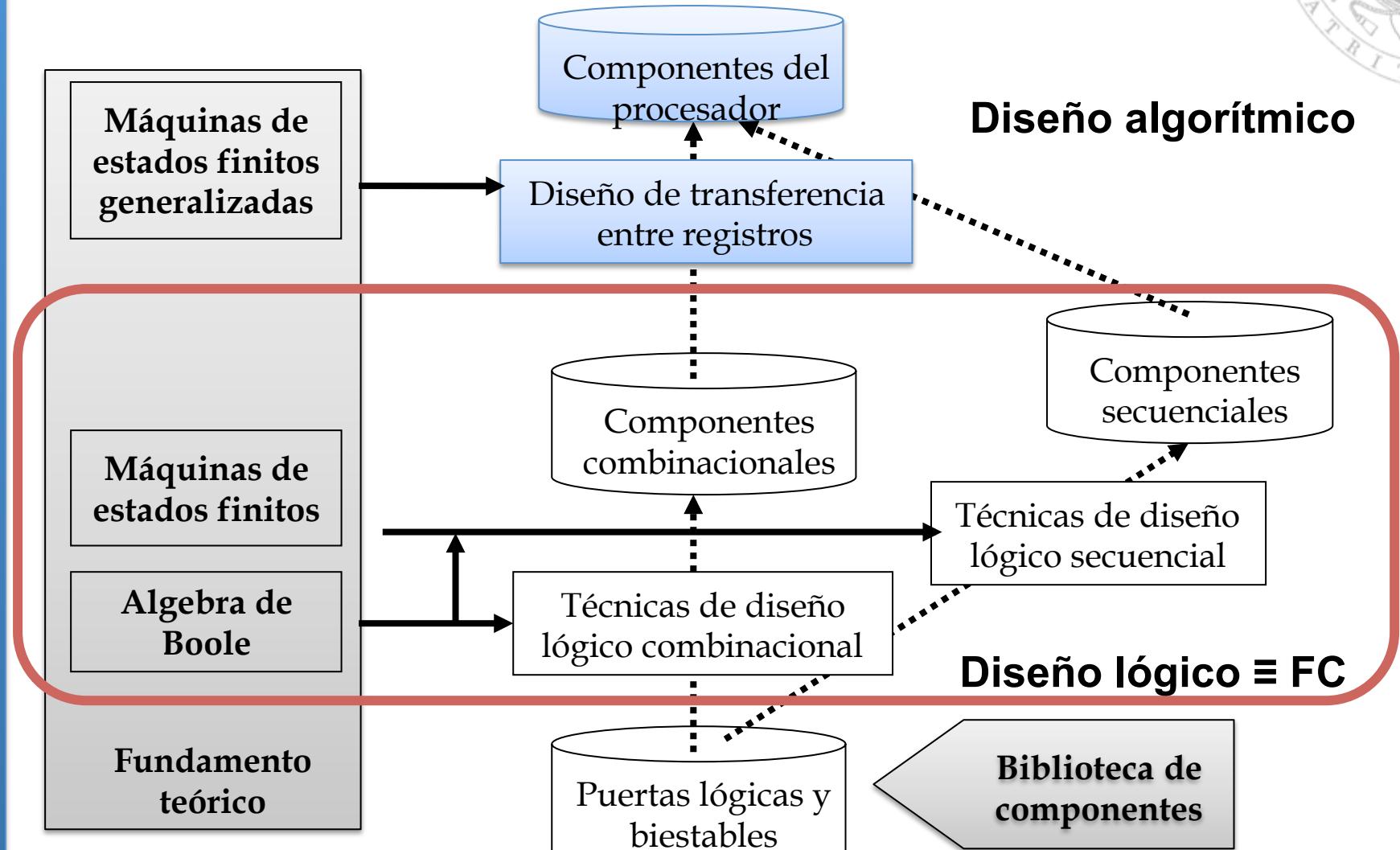
Tema 4: Diseño algorítmico



Índice

1. Introducción
2. Conocimientos previos
3. Diseño algorítmico: diagrama ASM, unidad de control y ruta de datos
4. Optimización de ASM
5. Principios de diseño: top-down/bottom-up, divide y vencerás, iterativo
6. Diseño RTL

Flujo de diseño



Conocimientos previos



- Especificación de sistemas secuenciales
 - FSM
 - ¿Dónde? [Fundamentos de computadores](#)
- Implementación de sistemas secuenciales
 - Optimización de estados. Biestables, Flip-flops
 - ¿Dónde? [Fundamentos de computadores](#)

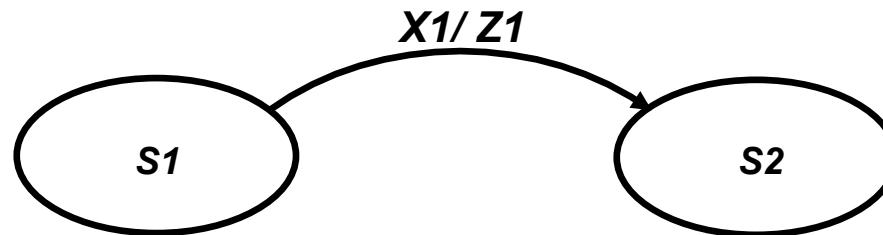


Máquinas de estados finitos (FSM)

- **Mealy:** Un cambio en la entrada en cualquier instante influye inmediatamente en la salida.

$$Z(t) = H(X(t), S(t))$$

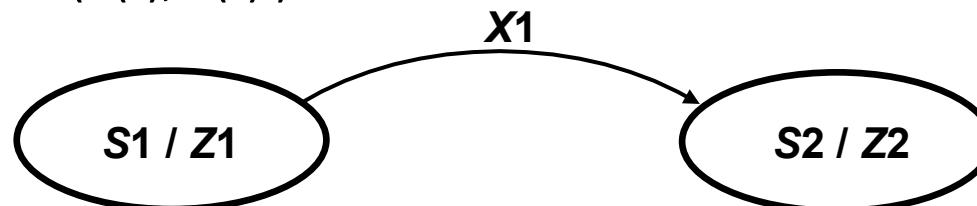
$$S(t+1) = G(X(t), S(t))$$



- **Moore:** Sólo el cambio del estado influye en la salida.

$$Z(t) = H(S(t))$$

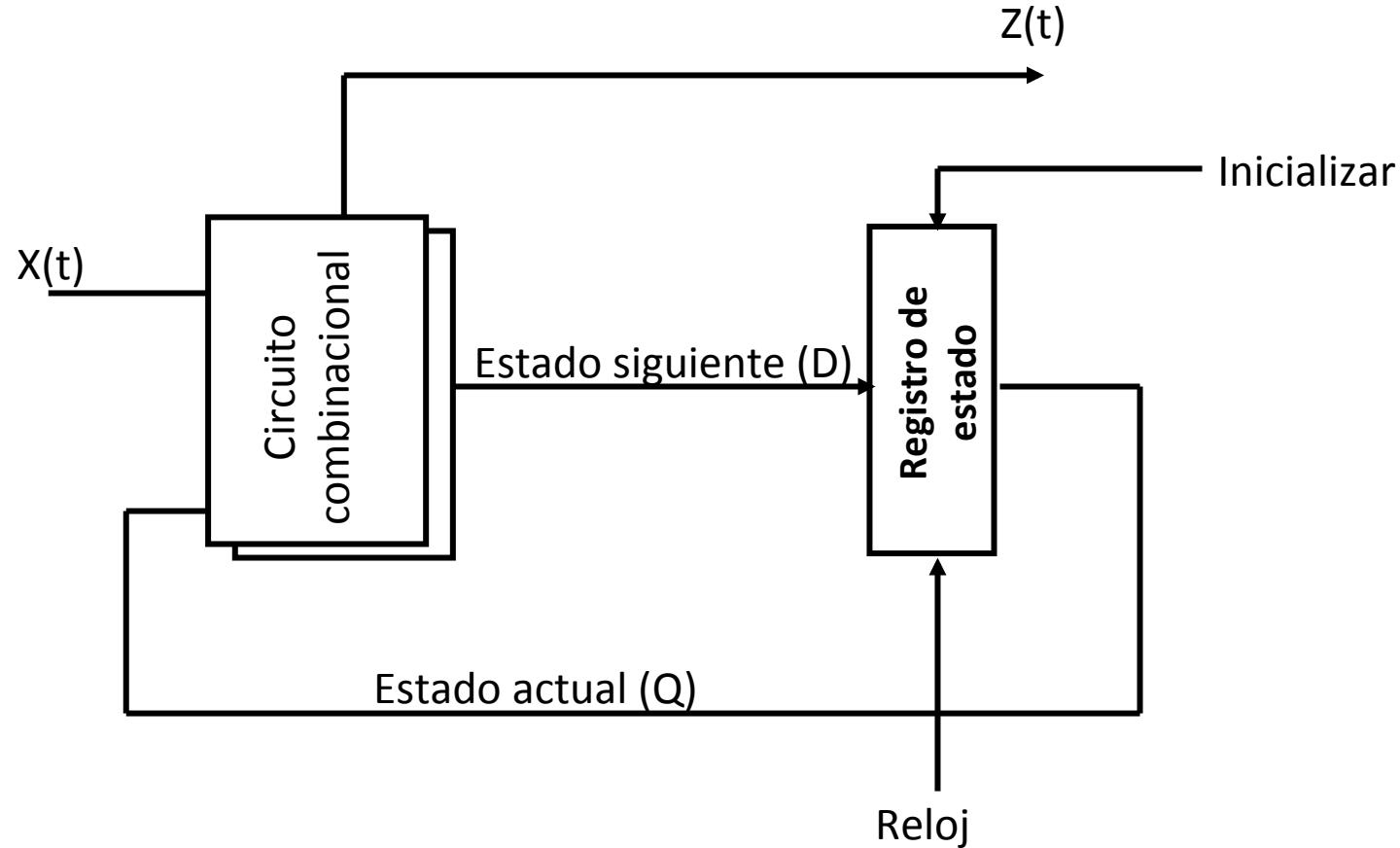
$$S(t+1) = G(X(t), S(t))$$



2. Conocimientos previos

toc

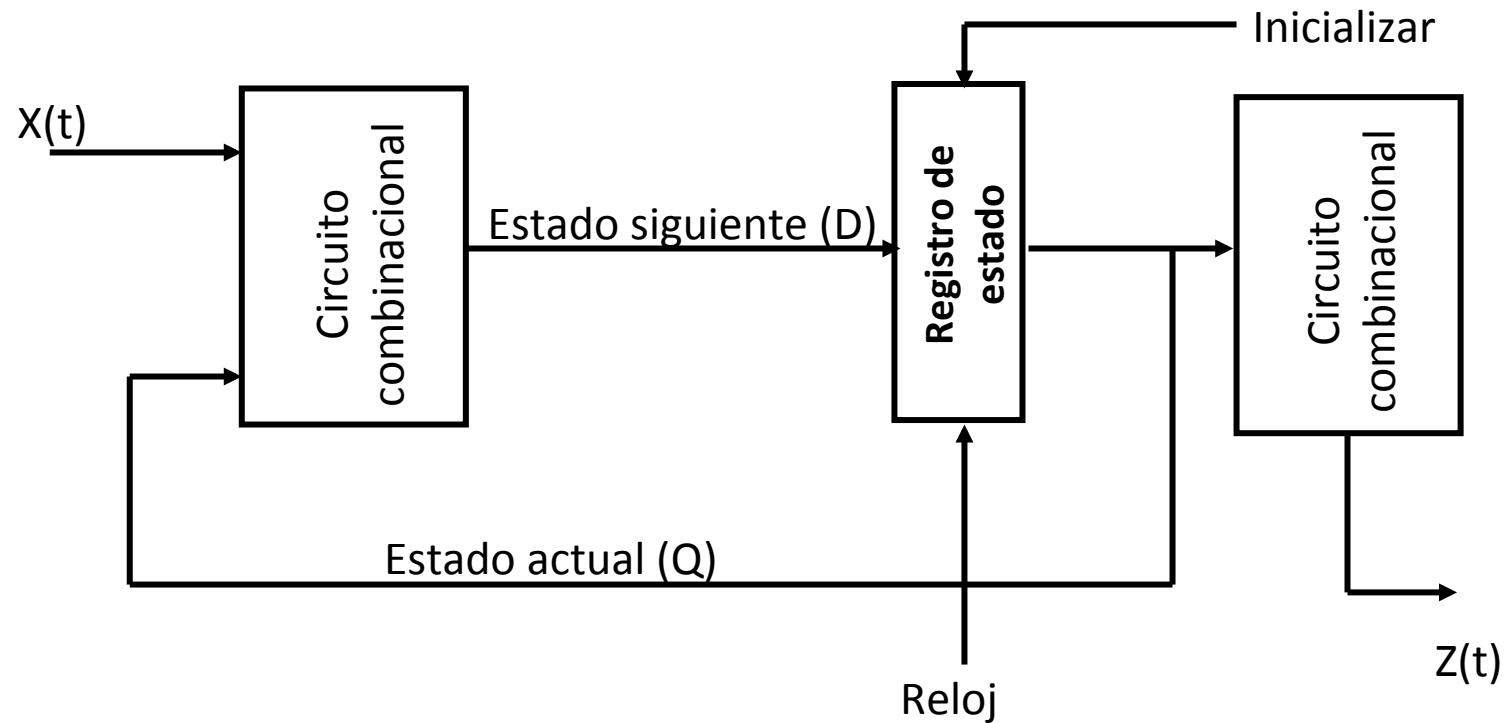
Implementación canónica. Mealy



2. Conocimientos previos

toc

Implementación canónica. Moore



Hipótesis funcionamiento síncrono



- Presentada en el tema 2.

Elementos de memoria



- Unos de los **elementos HW más importantes** de un sistema secuencial son los **elementos de almacenamiento**:
 - Almacenan el estado actual del sistema (máquina de control).
 - Almacenan valores intermedios (registros de datos).
- El elemento de memoria más sencillo es el **biestable**:
 - Se denomina biestable porque presenta dos únicos estados estables:
 - Salida 0
 - Salida 1
 - Sirve para almacenar un bit de información.



Tipos de biestables

- Según su comportamiento lógico:

- S-R
- D
- J-K
- T

S	R	Q+	D	Q+	J	K	Q+	T	Q+
0	0	Q	0	0	0	0	Q	0	Q
0	1	0	1	1	0	1	0	1	\overline{Q}
1	0	1			1	0	1	1	
1	1	proh.			1	1	\overline{Q}	1	\overline{Q}

- Según su comportamiento temporal:

- Latch
- Latch síncrono (sensible a nivel)
- Flip-flop disparado por flanco
- Flip-flop maestro-esclavo

Ecuaciones Características

$$R-S: \quad Q+ = S + \bar{R} Q$$

$$D: \quad Q+ = D$$

$$J-K: \quad Q+ = J \bar{Q} + \bar{K} Q$$

$$T: \quad Q+ = T \bar{Q} + \bar{T} Q$$

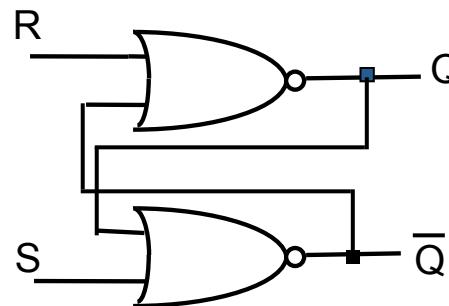
Deduccidas a partir de diagramas de K
 para $Q(t+1) = Q+ = f(\text{Entradas}, Q)$

Biestable asíncrono: Latch



- La salida cambia cuando cambian las entradas.
- Ejemplo 1: S-R (con entradas activas a nivel alto).

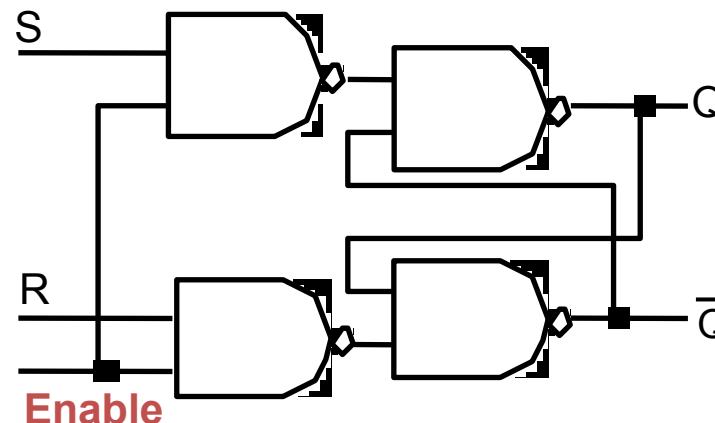
S	R	$Q(t+)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	proh.



Biestable síncrono sensible a nivel: Latch síncrono



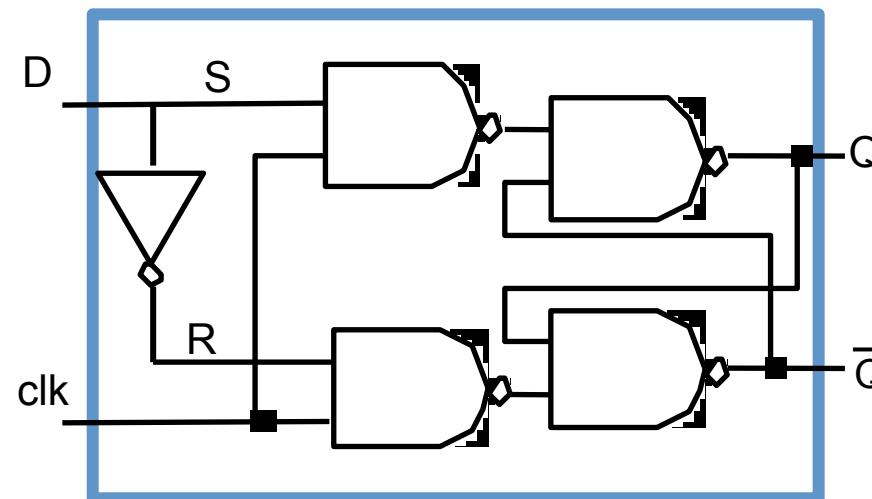
- La salida cambia cuando está activa la señal de capacitación (reloj).
- Ejemplo: S-R con entrada de capacitación.



Biestable síncrono sensible a nivel: Latch síncrono



- ¿Cómo conseguimos que sea un biestable tipo D?



clk	D	
0	0	
0	1	
1	0	
1	1	



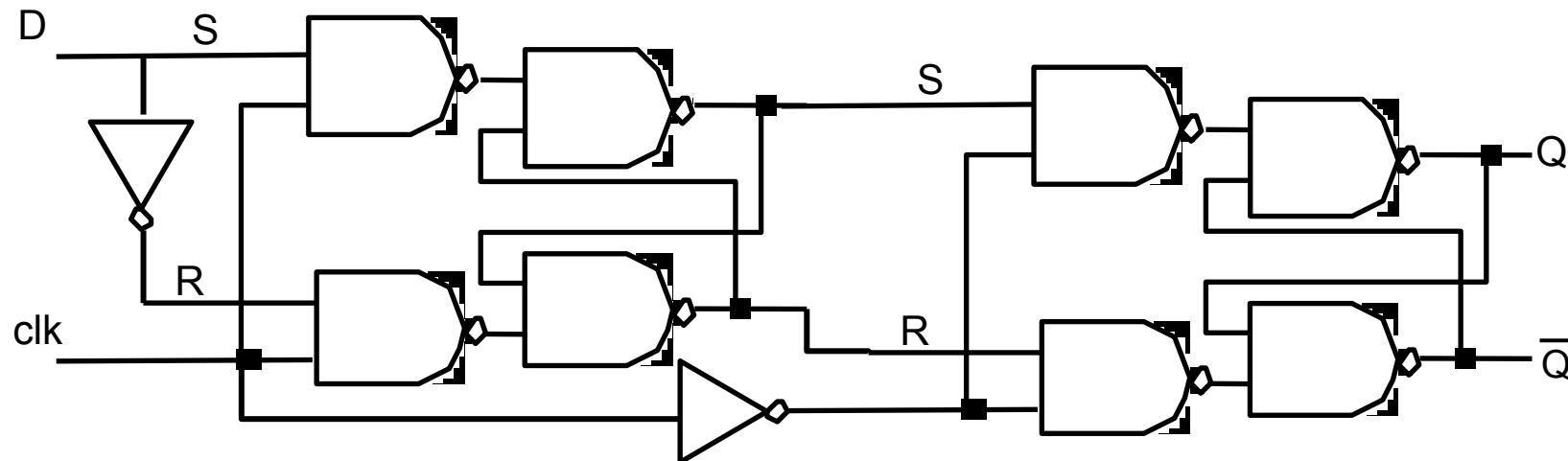
Problemas

- Si usamos latches debemos garantizar que:
 - El pulso de reloj es más corto que el retardo del latch.
 - Las entradas se mantienen constantes durante el pulso de reloj.
- Una alternativa es usar FLIP-FLOPs: más fiables.
 - Disparados por flanco: la salida sólo varía durante la transición del reloj (que es la entrada dinámica).
 - Maestro-esclavo.

Biestable maestro-esclavo: FF



- Se lee la entrada en un nivel y se modifica la salida en el contrario.



Comparación del comportamiento temporal de los biestables



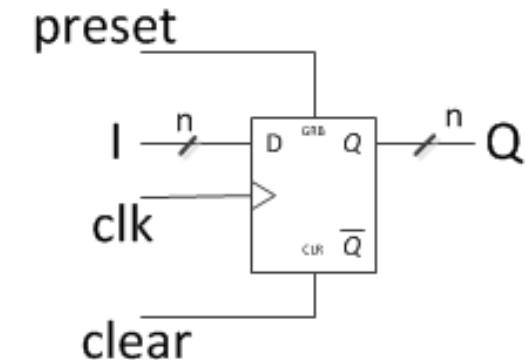
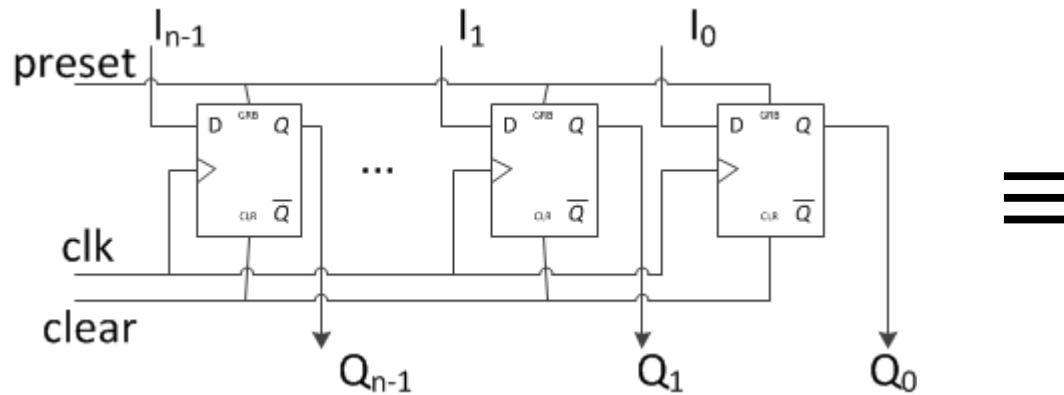
TIPO	¿Cuándo se muestran las entradas?	¿Cuándo son válidas las salidas?
Latch sin reloj	siempre	retardo de propagación desde el cambio en la entrada
Latch sensible a nivel	reloj en alta (T_{setup} y T_{hold} a cada lado del eje de bajada)	retardo de propagación desde el cambio en la entrada
Flipflop flanco de subida	transición de reloj de baja a alta (T_{setup} y T_{hold} a cada lado del eje de subida)	retardo de propagación desde flanco de subida del reloj
Flipflop flanco de bajada	transición de reloj de alta a baja (T_{setup} y T_{hold} a cada lado del eje de bajada)	retardo de propagación desde flanco de bajada del reloj
Flipflop Maestro-esclavo	transición de reloj de alta a baja (T_{setup} y T_{hold} a cada lado del eje de bajada)	retardo de propagación desde flanco de bajada del reloj

2. Conocimientos previos

toc

Registros

- Biestable de n-bits



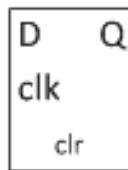


Registros

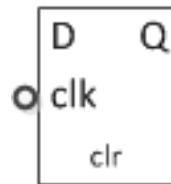
- Tipos:

- Según temporización

- Disparado por nivel

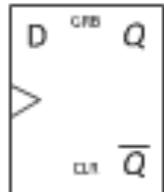


alto

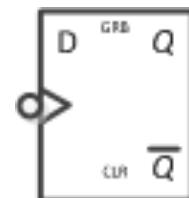


bajo

- Disparado por flanco



subida



bajada

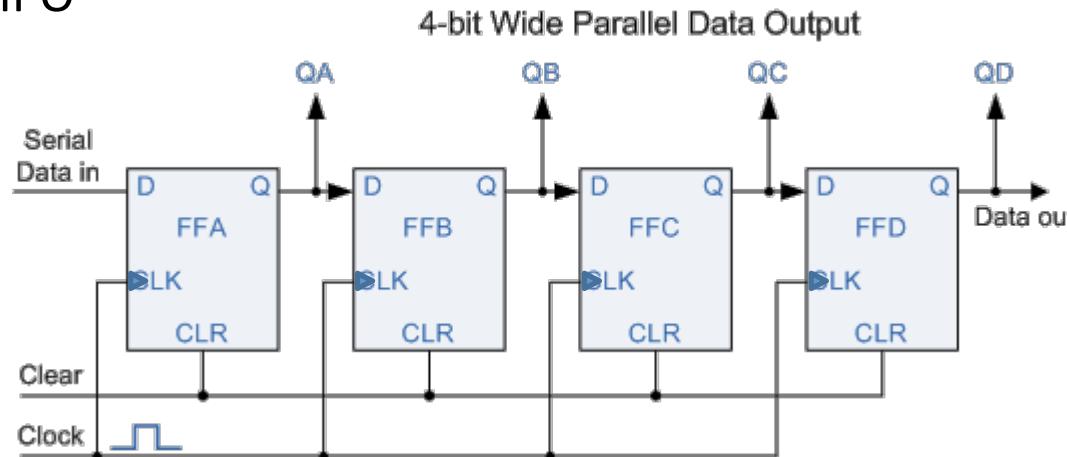
- Según funcionalidad

- Carga paralelo/Salida paralela-PIPO
 - Carga serie/Salida paralela-SIPO
 - Carga paralelo/Salida serie-PISO
 - Carga serie/Salida serie-SISO



Registros

- Tipos:
 - PIPO. Visto con anterioridad
 - SIPO

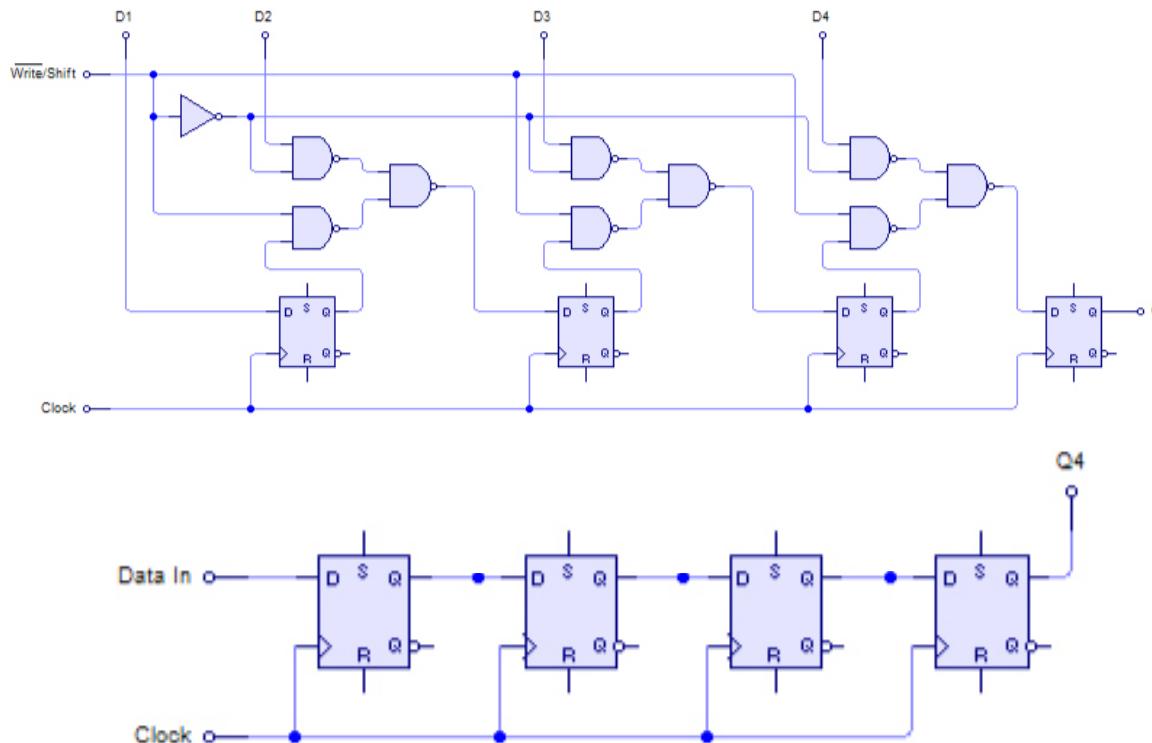


2. Conocimientos previos



Registros

- Tipos:
 - PISO
 - SISO



toc

VHDL



```
entity ff is
    port (clk: in std_logic;
          rst: in std_logic;
          dff: in std_logic;
          qff: out std_logic);
end ff;

architecture rtl of ff is
begin
p_ff: process(clk, rst)
begin
    if rst='1' then
        dff<='0';
    elsif rising_edge(clk) then
        q<=d;
    end if;
end process p_ff;
end ff;
```

VHDL



```
entity reg is
    port (clk: in std_logic;
          rst: in std_logic;
          d:   in std_logic_vector(7 downto 0);
          q:   out std_logic_vector(7 downto 0));
end reg;

architecture struct of reg is
component ff
    port(clk: in std_logic;
          rst: in std_logic;
          dff: in std_logic;
          qff: out std_logic);
end component ff;
begin
    gen_ff: for i in 0 to 7 generate
        i: ff port map(
            clk => clk,
            rst => rst,
            dff => d(i),
            qff => q(i)
        );
    end generate gen_ff;
end struct;

architecture rtl of reg is
begin
    p_reg: process(rst, clk)
    begin
        if rst='1' then
            q <= (others => '0');
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process p_reg;
end rtl;
```

¿Qué es el diseño algorítmico?

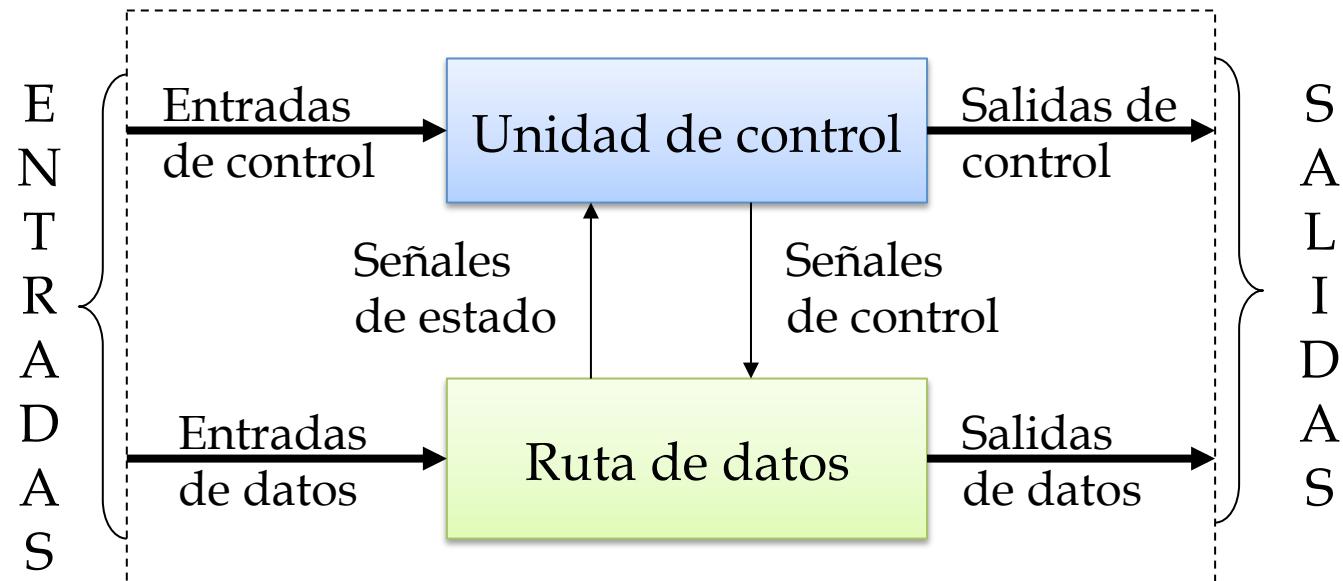


- Modo de especificación e implementación de sistemas digitales, que permite sistematizar y automatizar en gran medida su construcción.
- Parte siempre de una especificación en la que el comportamiento del sistema se describe en forma de un algoritmo:
 - Cómo calcular la salida en función de la entrada.
- Implementación:
 - Unidad de Control y Ruta de Datos.



¿Qué son los sistemas algorítmicos?

- Sistemas secuenciales síncronos.
- Comportamiento definido IMPLÍCITAMENTE
 - No se especifica el valor de Z sino el modo de calcularlo: el algoritmo.
- Modelo:





Proceso de diseño: resumen

1. Estudio de la especificación:
 - Esquema de pasos secuenciales a seguir (algoritmo)
 - Posible HW a utilizar (módulos específicos complejos)
2. Creación de un diagrama ASM: partiendo de las conclusiones del apartado 1, crear un diagrama que cumpla las especificaciones.
3. Diseño de la unidad de control:
 - Codificar cada uno de los estados del ASM
 - Codificar el cambio de estado
 - Mediante señales internas de control
 - Mediante señales externas de control
 - Codificar las señales de control hacia la ruta de datos: cada estado tendrá asociado unos valores de **TODAS** las señales que controlan los módulos complejos (por ejemplo: señal load de los registros).
4. Diseño de la ruta de datos:
 - Interconectar las señales de datos externas e internas a los módulos.
 - Interconectar las señales de control (obtenidas en el apartado 3) a los módulos de la ruta de datos.



Especificación

1. Estudio de la especificación
 - ¿Entiendo completamente el enunciado del problema
 - 1. Esquema de pasos secuenciales a seguir (algoritmo)
 - ¿Cumple las especificaciones?
 - ¿Puede simplificarse?
 - 2. Posible HW a utilizar (módulos específicos complejos)
 - ¿Añadir este HW me obliga a replantear el punto anterior?

Diagrama ASM



2. Creación de un diagrama ASM: partiendo de las conclusiones del apartado anterior:
 1. Número de estados
 2. Necesito más o menos estados
 3. Cuando se hacen efectivas las señales control
 1. Cuándo se carga un dato en un registro
 2. Cuándo deja de contar un contador
 3. Cuándo es correcta una comparación
 4. ...
 4. Estoy realizando un diseño Moore, Mealy o una mezcla

Diagrama de máquina de estados algorítmica (ASM)



- Forma gráfica de representar el algoritmo.
- Elementos:
 - Caja de estado: asignaciones y operaciones simultáneas.
 - Caja de decisión: bifurcación condicional con 2 posibles salidas.
 - Caja de salida condicional: asignaciones que se realizan cuando se cumple una condición (Mealy).
 - Bloque ASM: una caja de estado con red de cajas de decisión y de salida condicional.

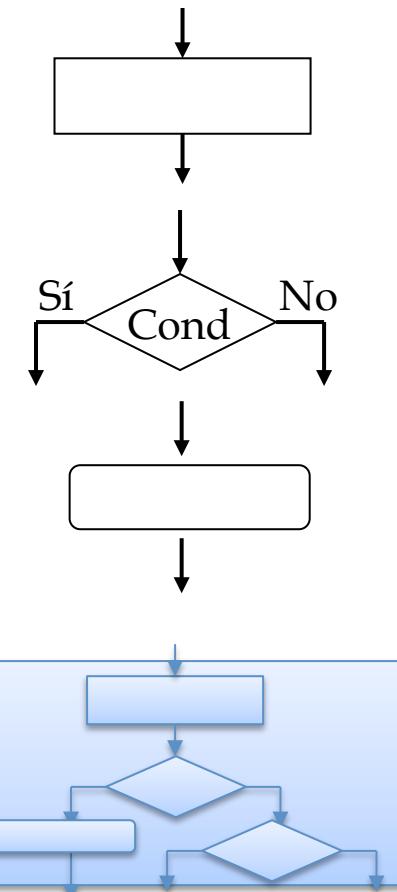
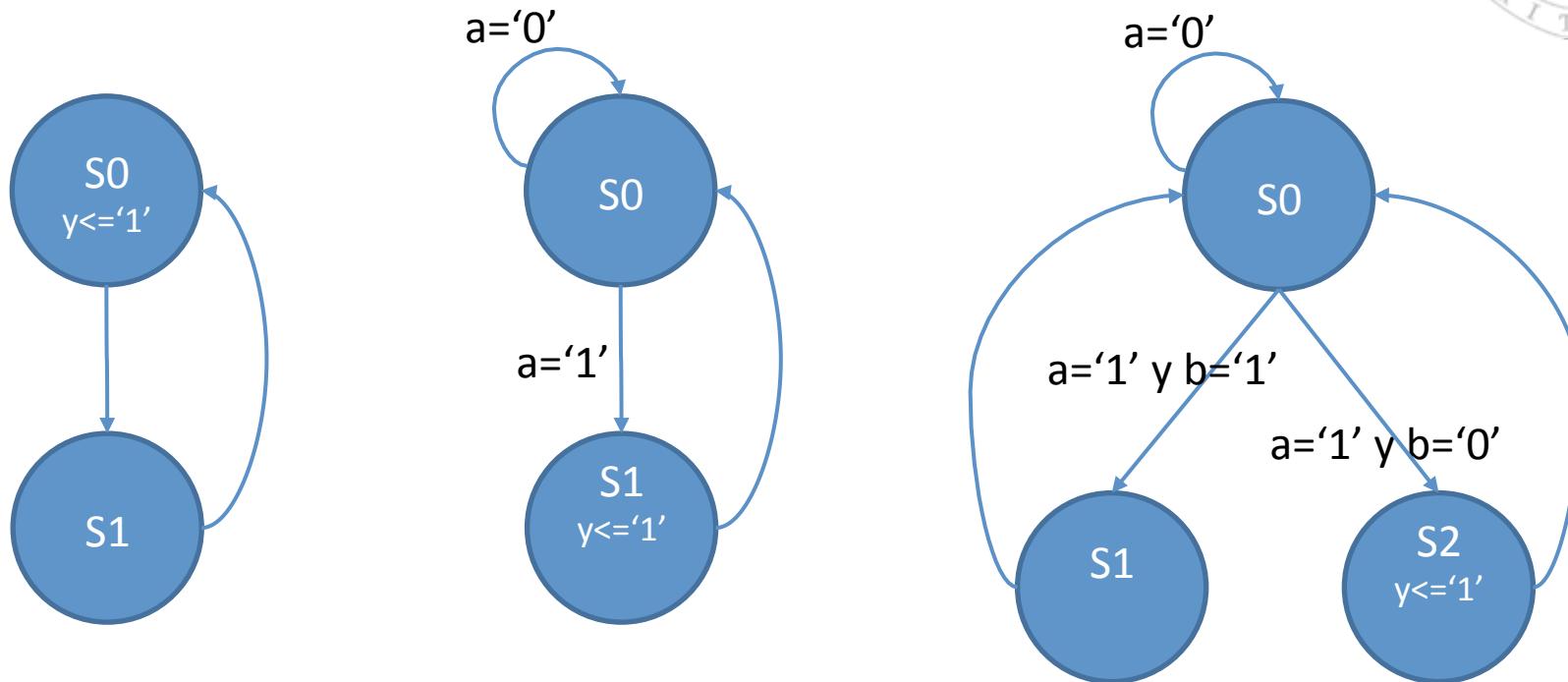
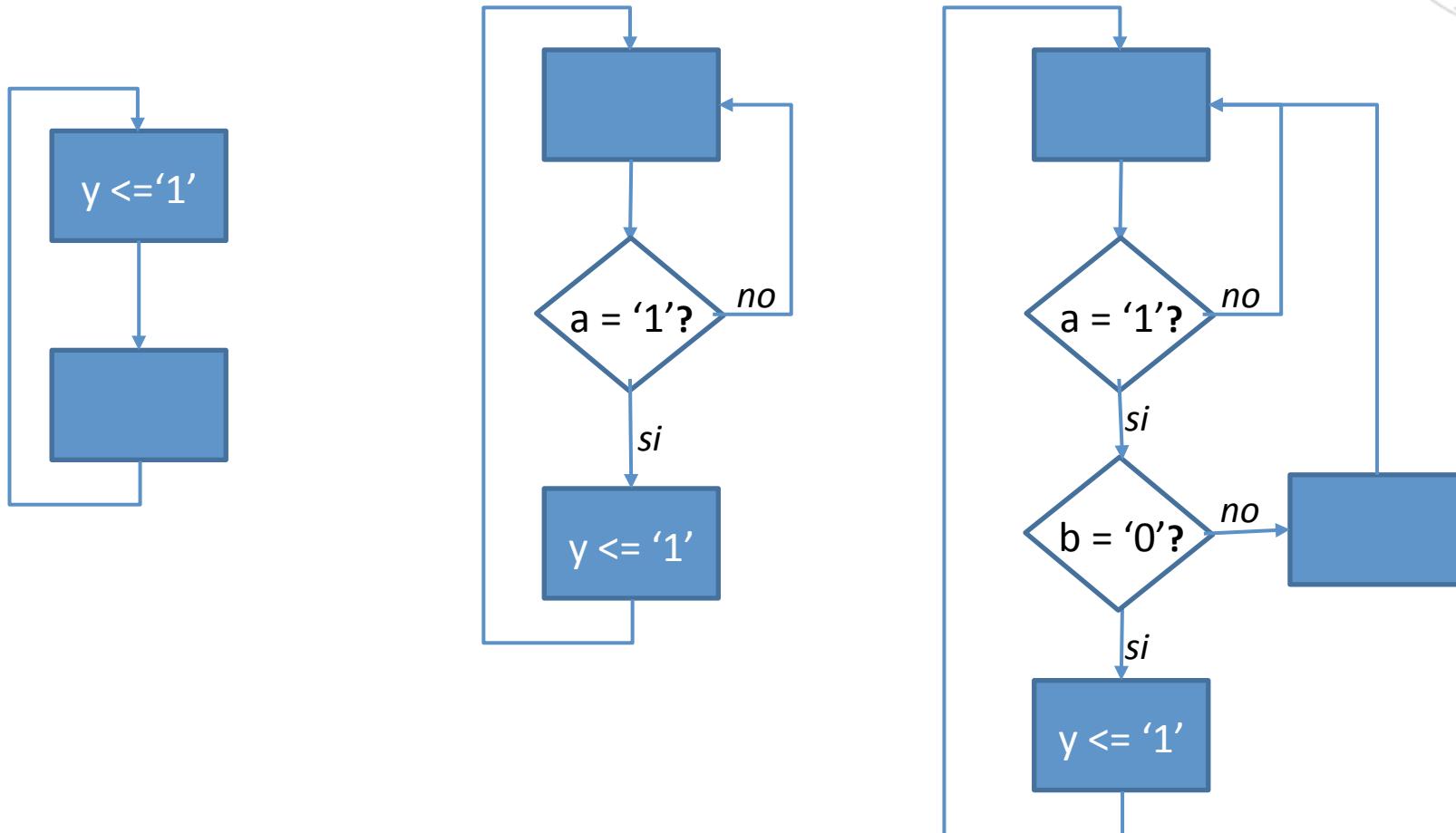


Diagrama de Máquina de Estados Algorítmica (ASM)



3. Diseño algorítmico

Diagrama de Máquina de Estados Algorítmica (ASM)

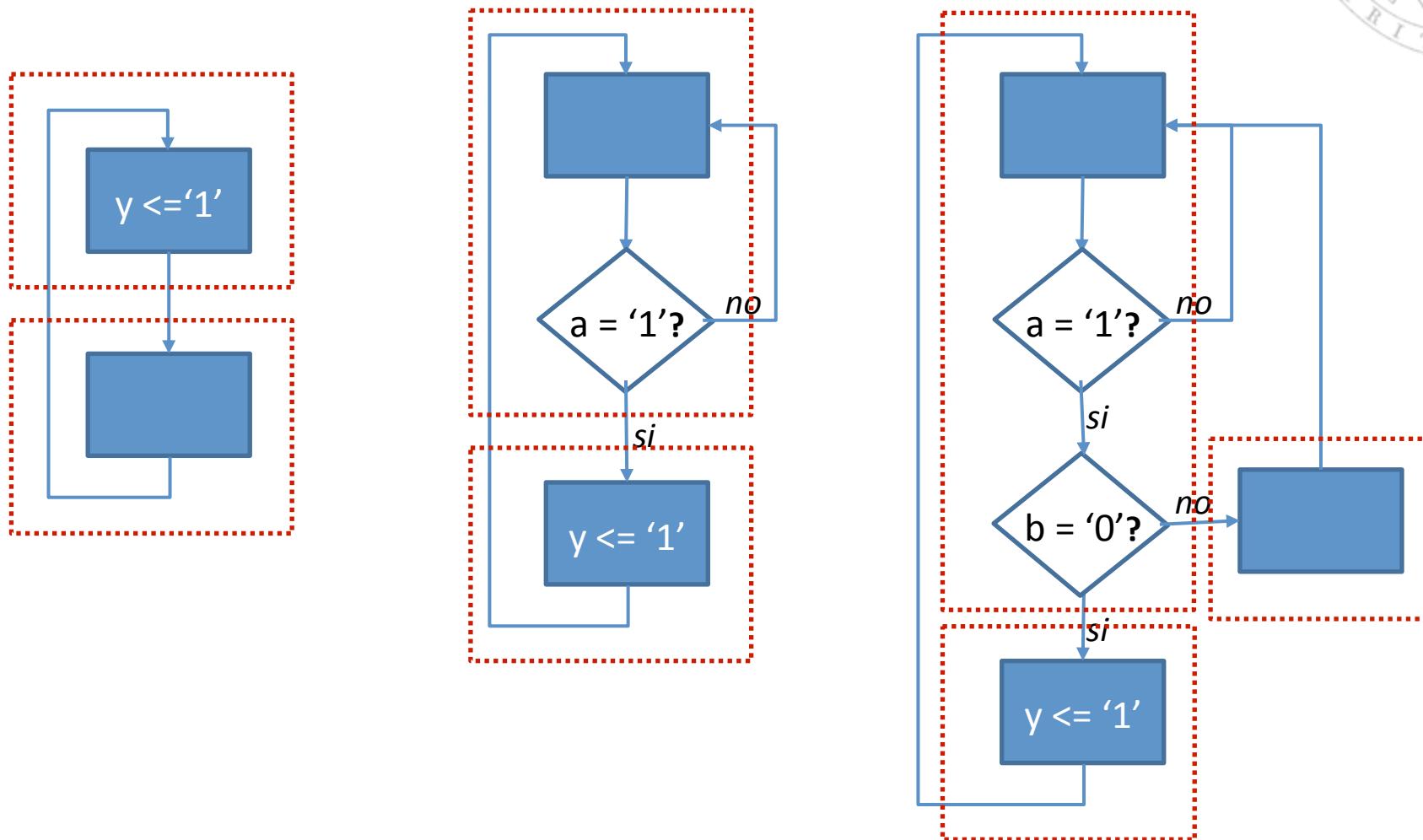


toc

3. Diseño algorítmico

toc

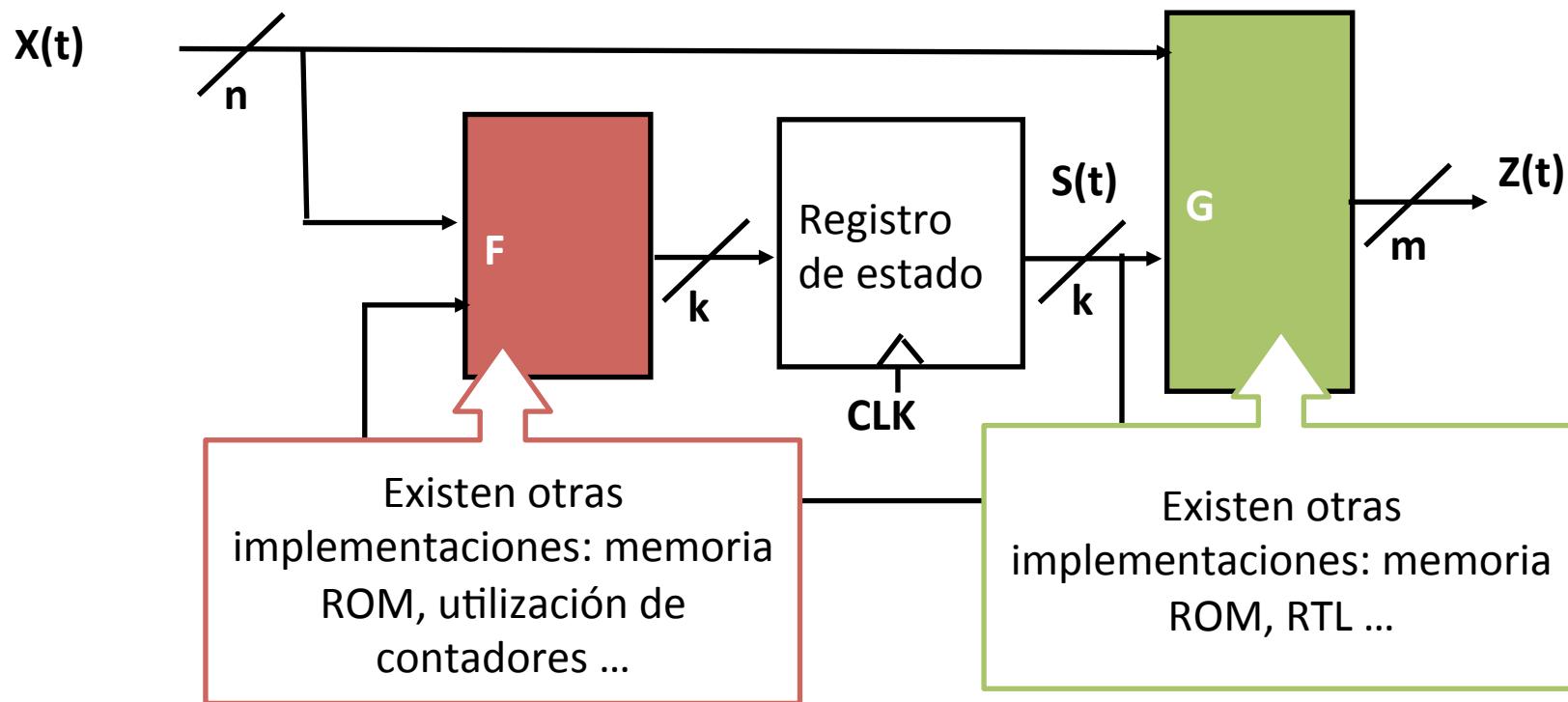
Diagrama de Máquina de Estados Algorítmica (ASM)



Diseño de la unidad de control



- Implementación canónica:
 - Fundamentos de Computadores (1er cuatrimestre)
 - Registro de estado + red combinacional.
 - Ejemplo con n entradas, k variables de estado y m salidas. Para implementar F y G se necesita un módulo de $n+k$ variables y $m+k$ salidas.



Diseño de la unidad de control



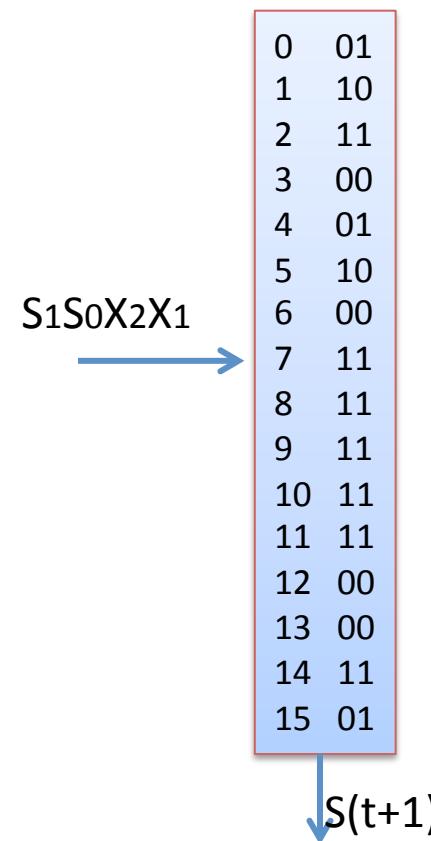
- Registro de estado y red combinacional
 - Implementación canónica.
 - Uso de ROMs y PLAs en redes secuenciales en las que
 - Estado siguiente y salida dependen de subconjuntos diferentes de las variables.
 - Estado siguiente sólo depende de una variable de entrada en cada estado actual.
- Contador y red combinacional
 - Si en muchos casos $S(t+1) = S(t)+1$
- Registro de desplazamiento y red combinacional
 - Registro de desplazamiento realimentado.
 - Registro de desplazamiento no realimentado.

Diseño de la unidad de control



- Diseño con ROM o PLA
- Ejemplo:

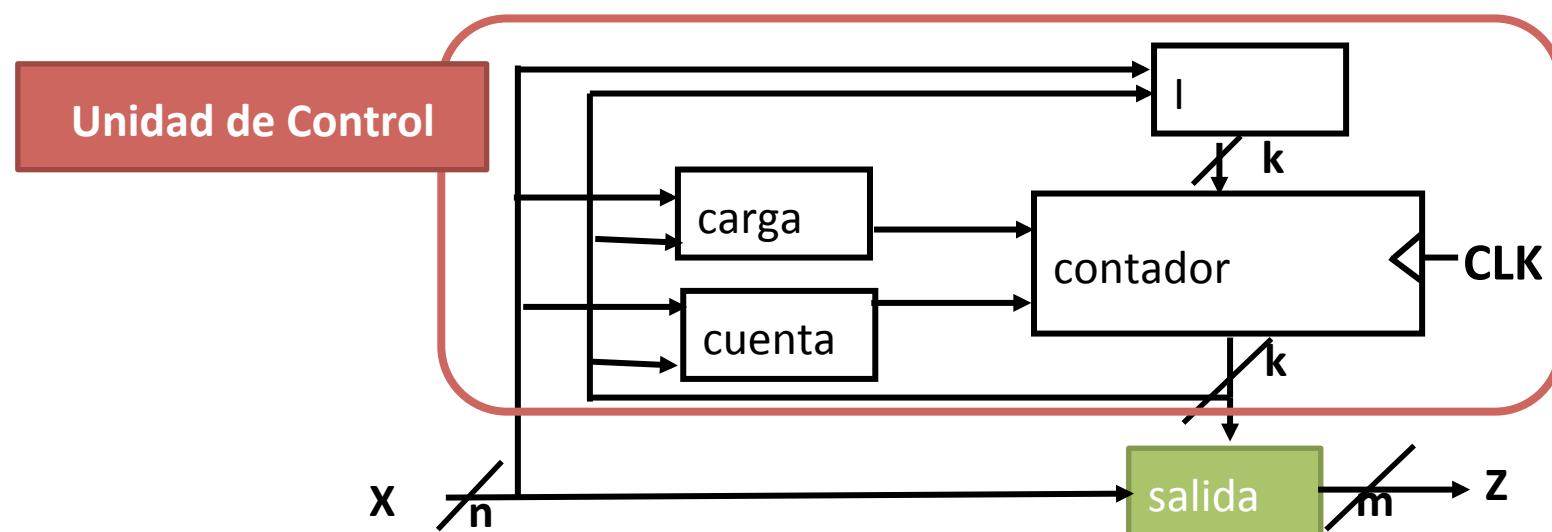
S(t)	X ₂ X ₁ =00	01	10	11	S(t+1)
00	01	10	11	00	
01	01	10	00	11	
10	11	11	11	11	
11	00	00	11	01	



Diseño de la unidad de control



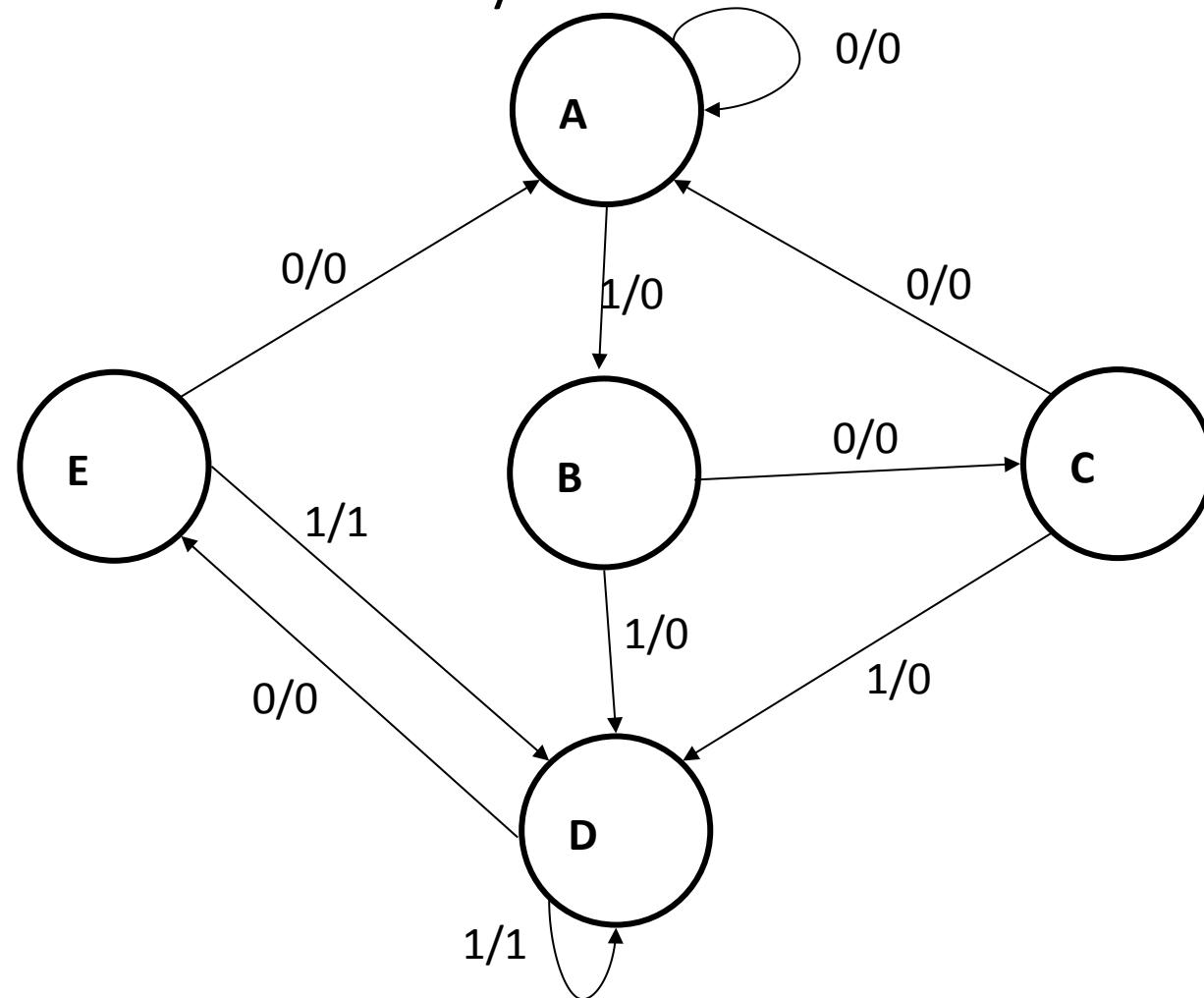
- Diseño con un contador y red combinacional
 - Estado= Número binario de k bits
 - Posibles transiciones de estado:
 - 1) $S(t+1) = (S(t)+1) \bmod p$ (o $(S(t)-1) \bmod p$)
 - 2) $S(t+1) = S(t)$
 - 3) $S(t+1) = I, I \neq S(t), I \neq (S(t)+1) \bmod p, I \neq (S(t)-1) \bmod p$



Diseño de la unidad de control



- Diseño con un contador y red combinacional



Diseño de la unidad de control



- Diseño con un registro de desplazamiento y red combinacional.
 - Estado siguiente = estado actual \gg o \ll
 - Ejemplo: Reconocedor de secuencia.

Además



- El diseño con ROMs también puede utilizarse para las salidas del sistema (circuito G).
- El diseño con ROM es el que utiliza la herramienta de Xilinx para implementar los diseños
 - Utiliza ROMs de 16x1 bits.



Diseño de la ruta de datos

1. Instanciar los módulos hardware deducidos del ASM.
2. Interconectar las señales de datos externas e internas a los módulos.
3. Interconectar las señales de control (obtenidas en el apartado 3) a los módulos de la ruta de datos.

Ejemplo sistema algorítmico



- Utilizando los conceptos de diseño algorítmico diseñar un sistema que sea capaz de reconocer una clave.
- El sistema tendrá dos modos de funcionamiento:
 - Cambiar clave: En este modo se puede introducir la nueva clave (4 bits) en un ciclo de reloj.
 - Introducir clave: en este modo se introduce la clave, un bit por cada ciclo y se compara con la clave guardada:
 - En caso de acierto se pone la señal acierto a 1 y se vuelve al estado inicial (decidir si se introduce o se cambia la clave).
 - En caso de fallo se pone la señal acierto a 0 y el sistema volvería a pedir de nuevo que se introduzca la clave (bit a bit).

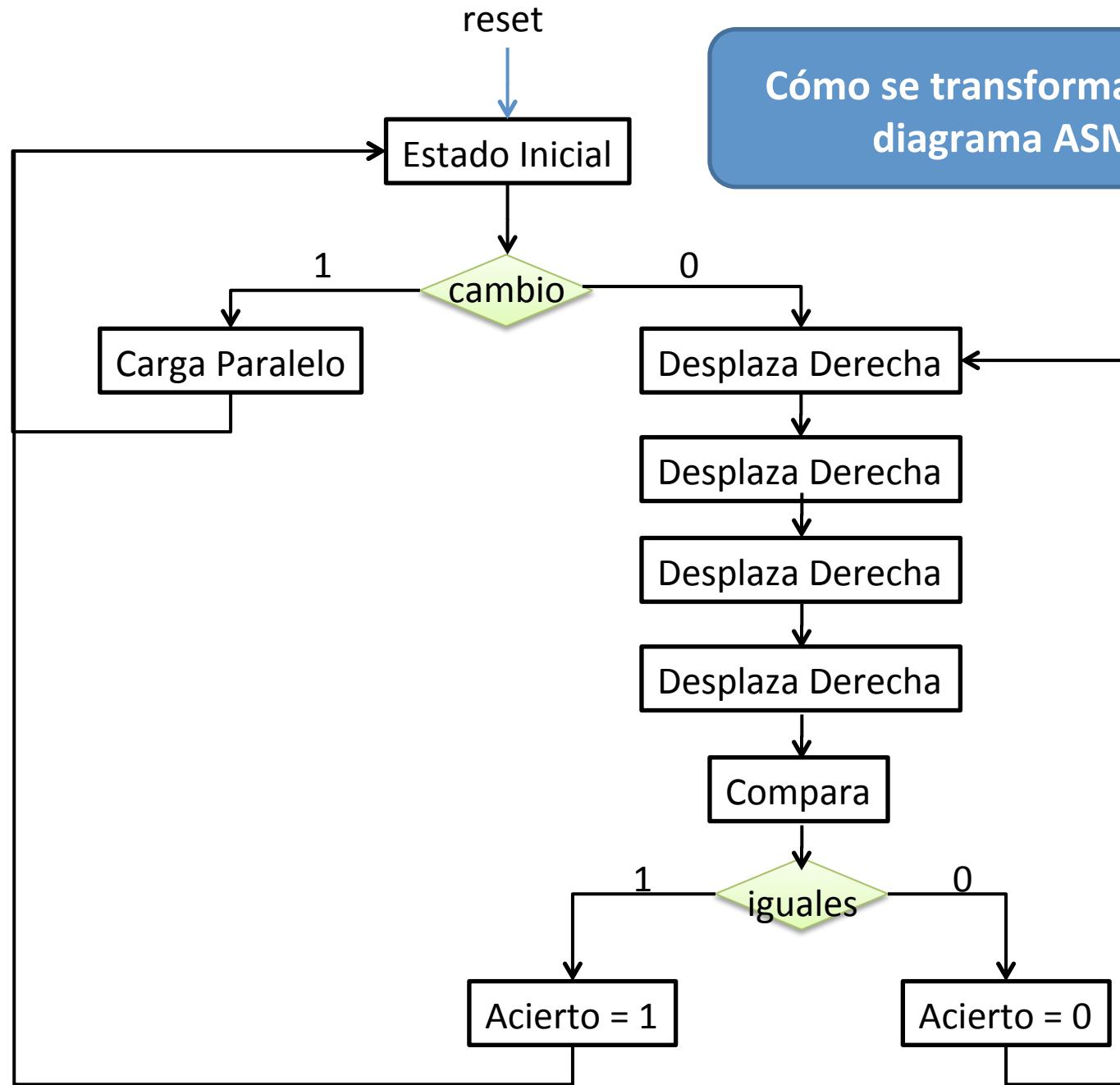


Ejemplo sistema algorítmico

- Estudio de la especificación:
 - Esquema de pasos secuenciales a seguir (algoritmo)
 - i. Estado inicial
 - ii. Si cambio es 1
 - Carga paralelo de la clave
 - Vuelta al estado inicial
 - iii. Si cambio es 0
 - Realizar 4 desplazamientos a la derecha (4 ciclos)
 - Si la clave es correcta: acierto = 1 y volver al punto i.
 - Si la clave es incorrecta: acierto = 0 y volver al punto iii.
- Possible HW a utilizar (módulos específicos complejos)
 - 2 registros, uno en modo carga paralelo y otro en modo carga serie.
 - Un comparador

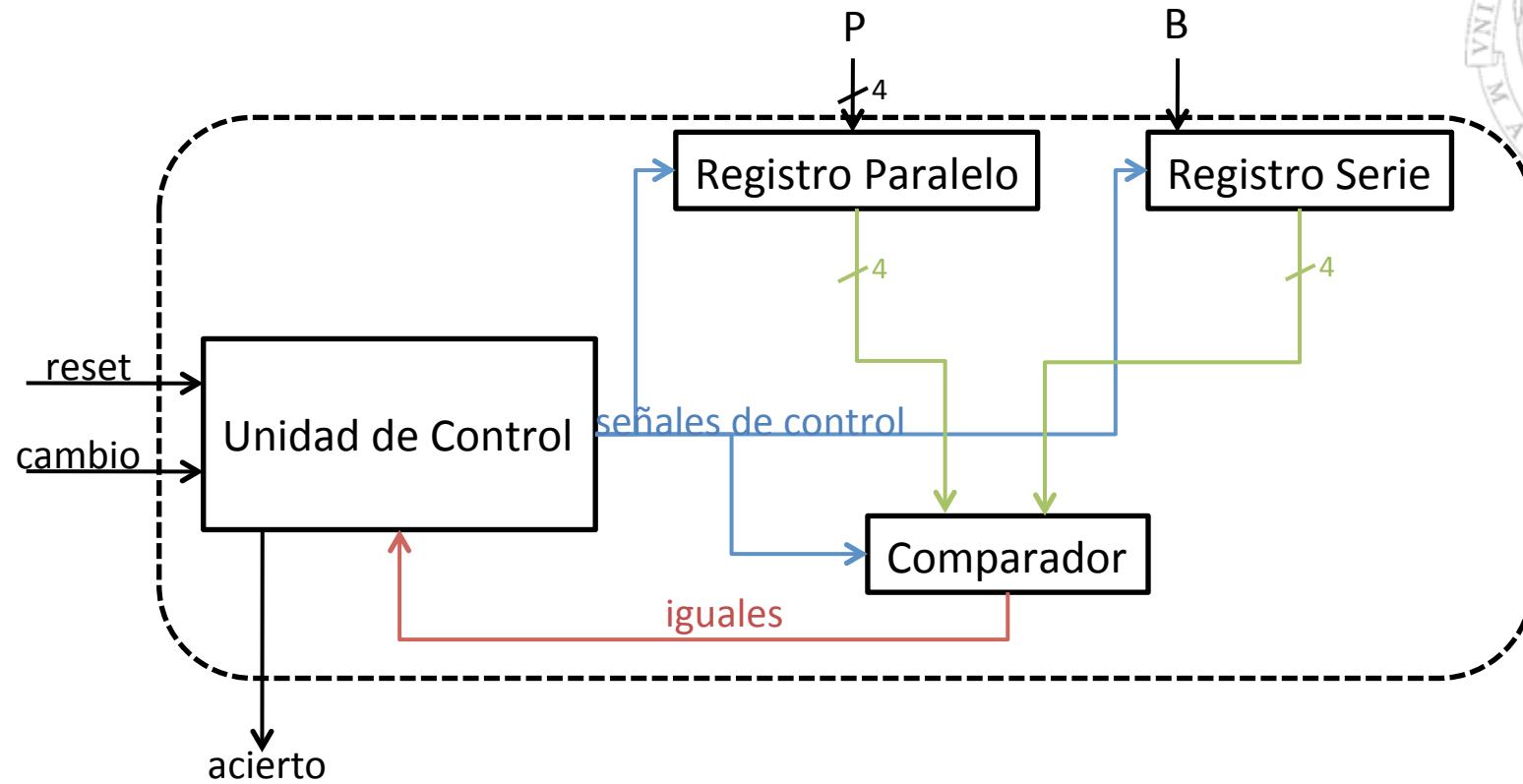
3. Diseño algorítmico

toc



3. Diseño algorítmico

toc



Ruta de Datos
+
Unidad de Control

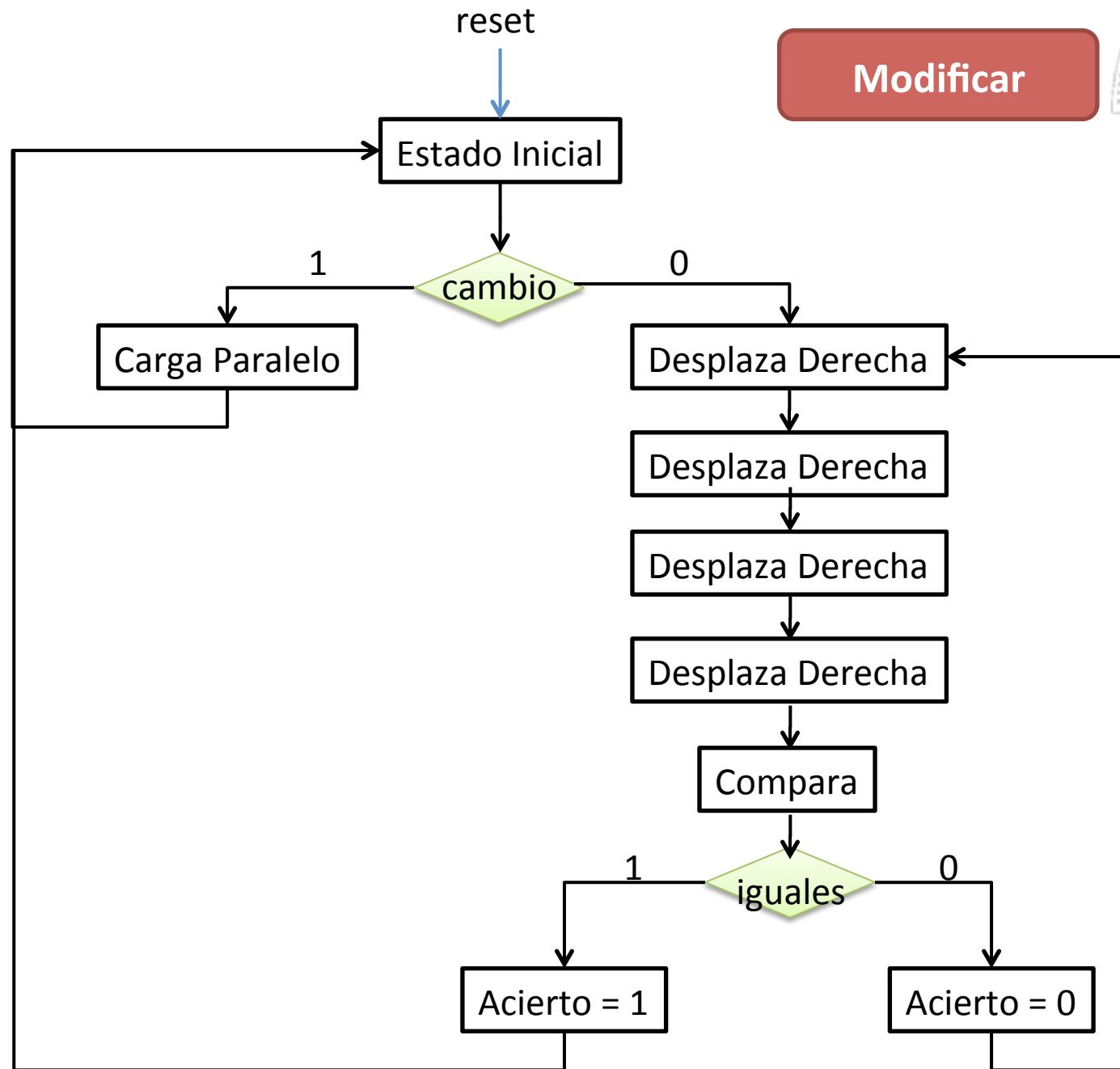


No funciona

¿Por qué?

3. Diseño algorítmico

toc



Optimización



- Compartición de unidades funcionales
 - Minimizar número de U.F. en la ruta de datos.
- Compartición de registros
 - Minimizar número de registros en la ruta de datos.
- Compartición de conexiones
 - Minimizar el número de conexiones en la ruta de datos.

Optimización



- Vamos a presentar las técnicas usando como ejemplo el ASM de la aproximación de la raíz cuadrada de dos enteros con signo:

$$\sqrt{a^2 + b^2} \simeq \max((0.875 \cdot x + 0.5 \cdot y), x)$$

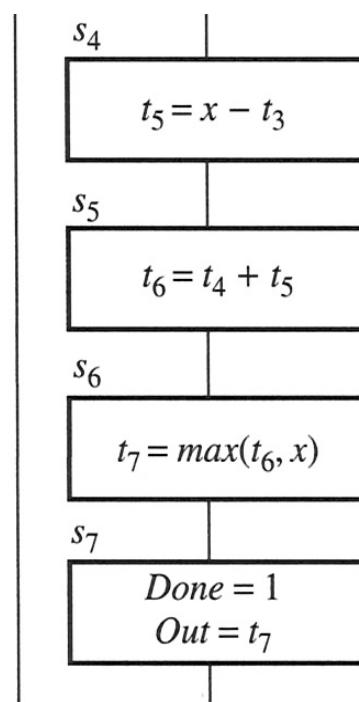
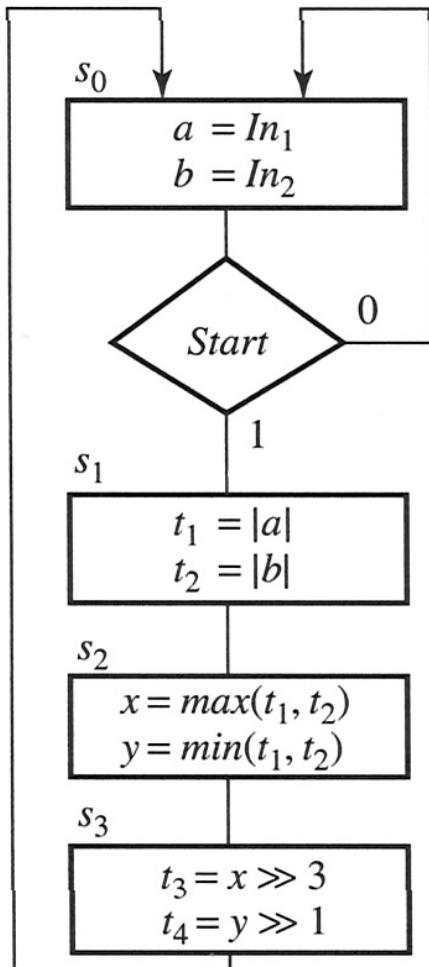
$$x = \max(|a|, |b|)$$

$$y = \min(|a|, |b|)$$

4. Optimización de ASM

toc

Optimización



Recursos

11 registros
2 abs
1 min
1 max
2 desplazadores
1 sumador
1 restador





Compartir Registro

- Algoritmo
 - Determinar tiempo de vida de la variable
 - Conjunto de estados en los que la variable está viva.
 - Agrupar variables con tiempos de vida disjuntos y asignarles un mismo registros
 - Agrupación con distintos criterios: p.e. minimizar el número de registros, minimizar número de registros y mux, ...

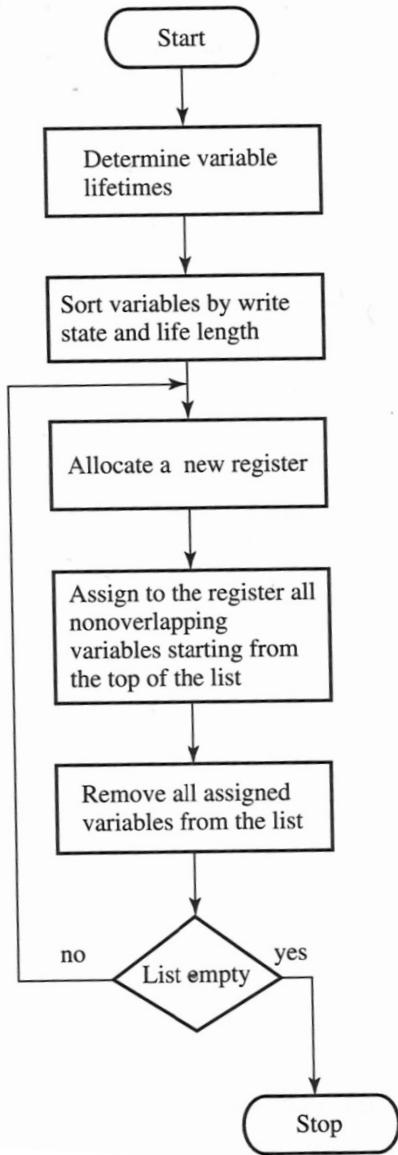


Algoritmo del vértice izquierdo (left-edge algorithm)

4. Optimización de ASM

toc

Compartir Registro



	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a	x						
b	x						
t_1		x					
t_2		x					
x			x	x	x	x	
y			x				
t_3				x			
t_4				x	x		
t_5					x		
t_6						x	
t_7							x

Number of live variables	s_1	s_2	s_3	s_4	s_5	s_6	s_7
2	2	2	2	3	3	2	1

$$R_1 = [a, t_1, x, t_7]$$

$$R_2 = [b, t_2, y, t_4, t_6] \rightarrow 3 \text{ registros!}$$

$$R_3 = [t_3, t_5]$$



Compartir Registro

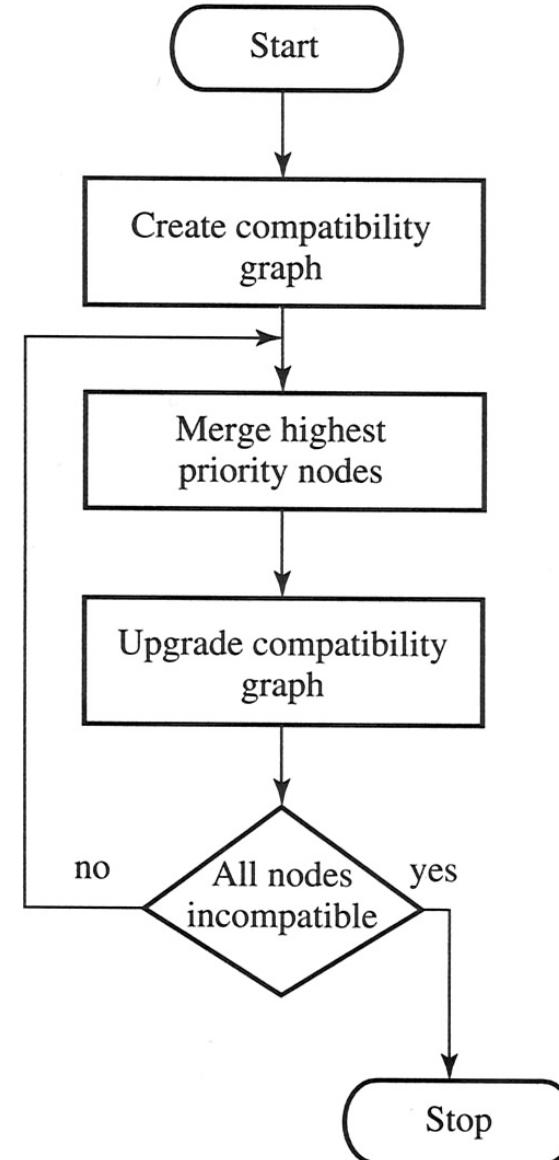
- Dependiendo de la asignación cambia el número de MUX en la ruta de datos.
- ¿Asignación óptima?
 - Algoritmo basado en grafo de compatibilidad
- Grafo de compatibilidad
 - Vértices = variables.
 - Arista: compatibilidad entre variables
 - Aristas de incompatibilidad
 - Aristas prioritarias (opcional). Incluye etiqueta s/d

UF que usan ambos nodos

UF que generan ambos nodos

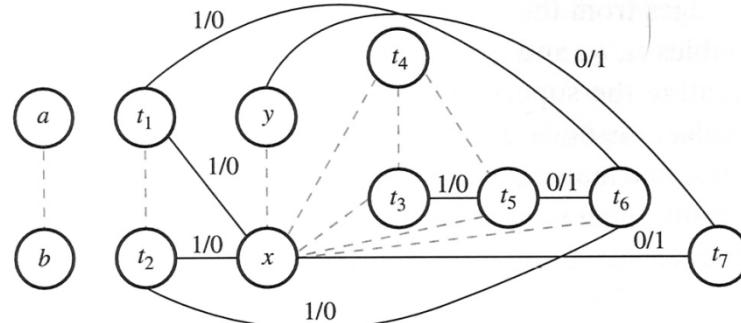
Optimización. Compartir Registro

- Partición del grafo:
agrupar vértices
conectados por aristas
prioritarias de mayor
peso y crear
supernodos.

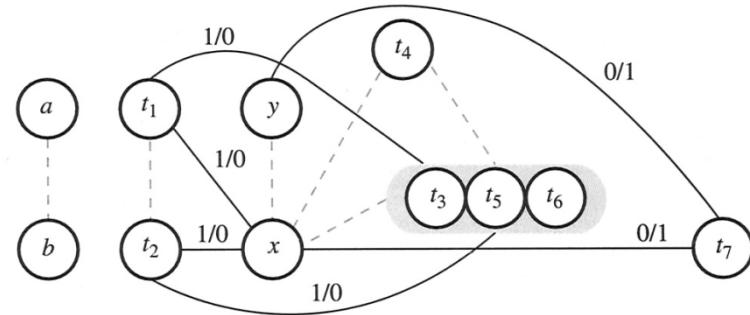


4. Optimización de ASM

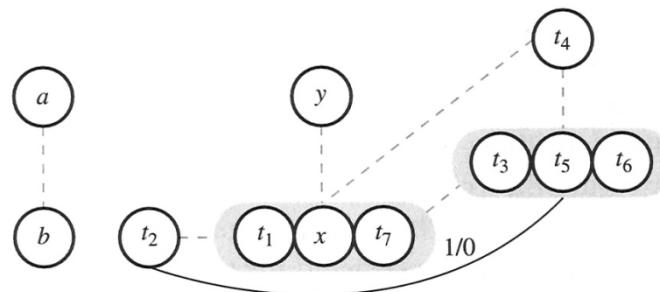
Optimización. Compartir Registro



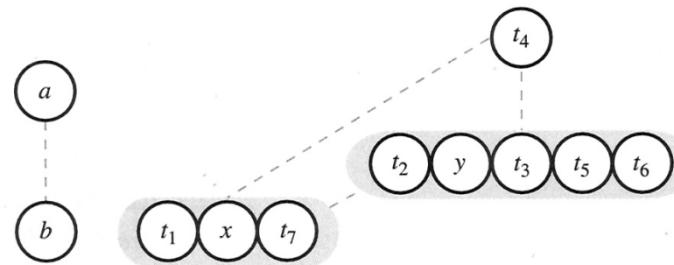
(a) Initial compatibility graph



(b) Compatibility graph after merging t_3 , t_5 , and t_6



(c) Compatibility graph after merging t_1 , x , and t_7



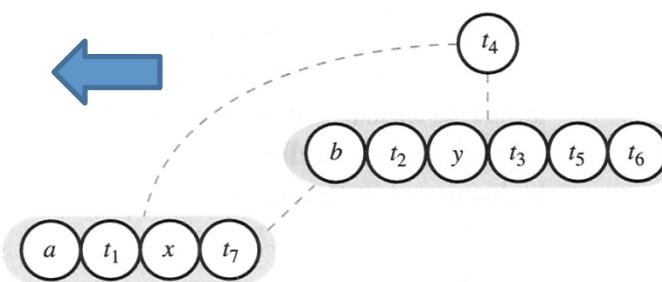
(d) Compatibility graph after merging t_2 and y

$$R_1 = [t_4]$$



$$R_2 = [b, t_2, y, t_3, t_5, t_6]$$

$$R_3 = [a, t_1, x, t_7]$$



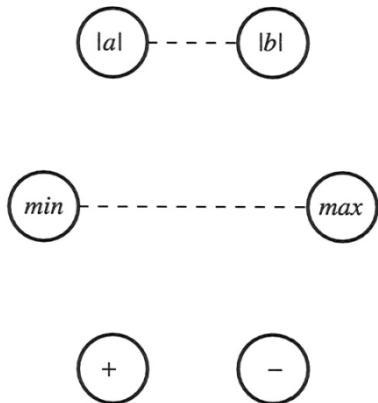
toc

Optimización. Compartir UF



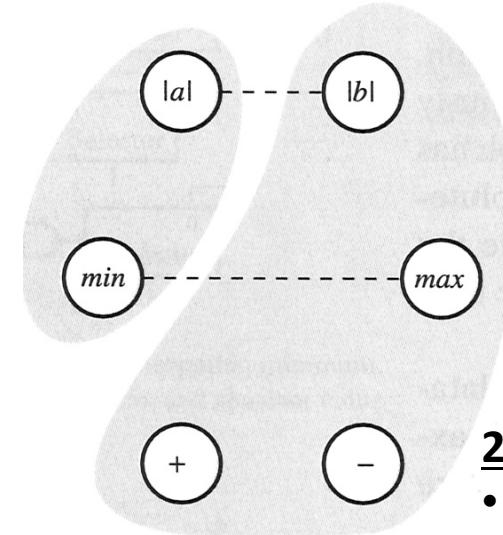
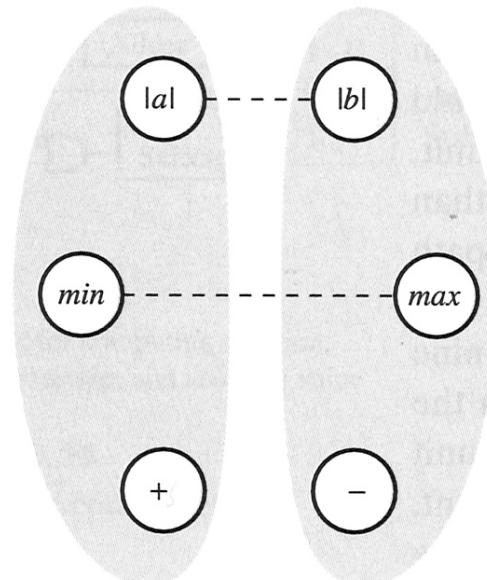
- Agrupar UF sencillas en UF más complejas: UF multifunción.
- ¿Cuándo? UF multifunción y el coste de conexión es menor que el coste de las UF sencillas.
- ¿Cómo? Algoritmo de particionamiento grafo de compatibilidad

Optimización. Compartir UF



2 UF:

- 2 ABS
- min
- max
- -
- +



2 UF:

- ABS, min
- ABS, max, -, +

2 UF:

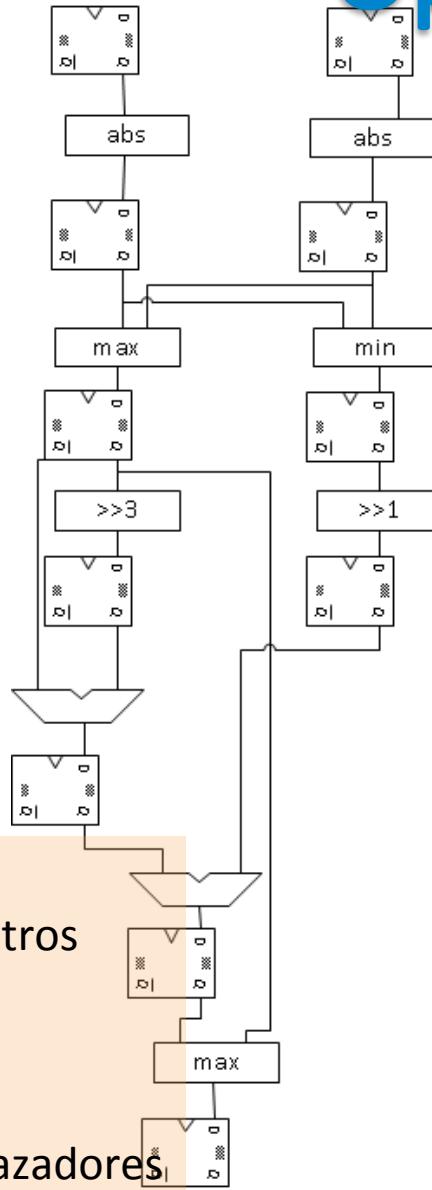
- ABS, min, +
- ABS, max, -

4. Optimización de ASM

toc

Original

- 11 registros
- 5 INV
- 6 +
- 4 MUX
- 2 desplazadores

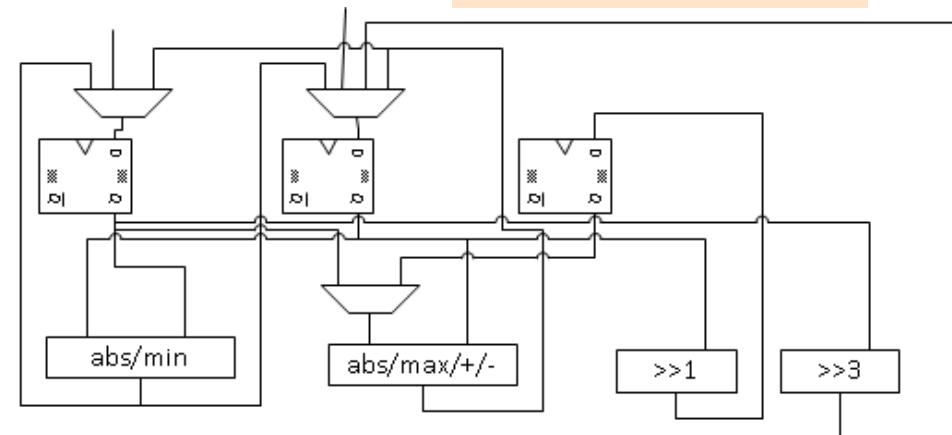


Optimización



Optimizada

- 3 registros
- 2 AND
- 1 INV
- 1 XOR
- 2 +
- 2 MUX
- 2 desplazadores

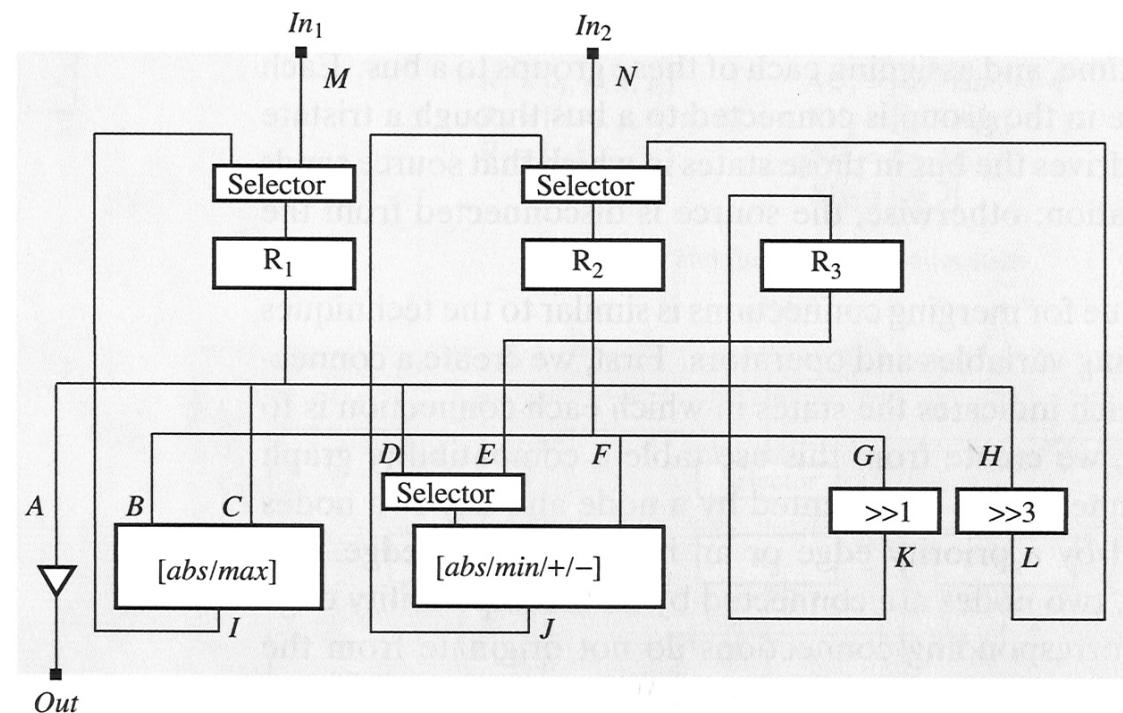


Optimización. Compartir conexión



- Basado en grafo de compatibilidad:
 - Tabla de uso de conexiones.
 - Grafo de compatibilidad:
 - Vértices son conexiones.
 - Arista:
 - Incompatibles: no tienen la misma fuente pero son usadas en el mismo ciclo.
 - Prioritarias: tienen una misma fuente o un mismo destino.

Optimización. Compartir conexión

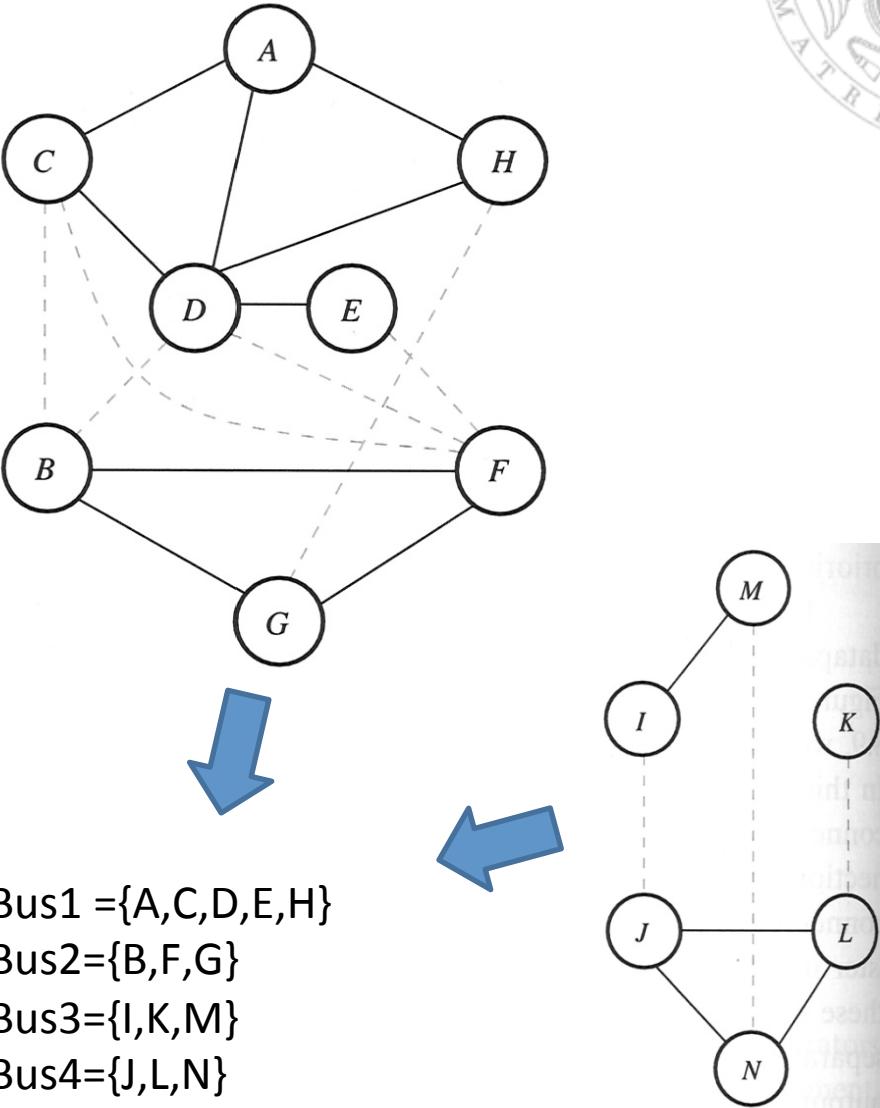


4. Optimización de ASM



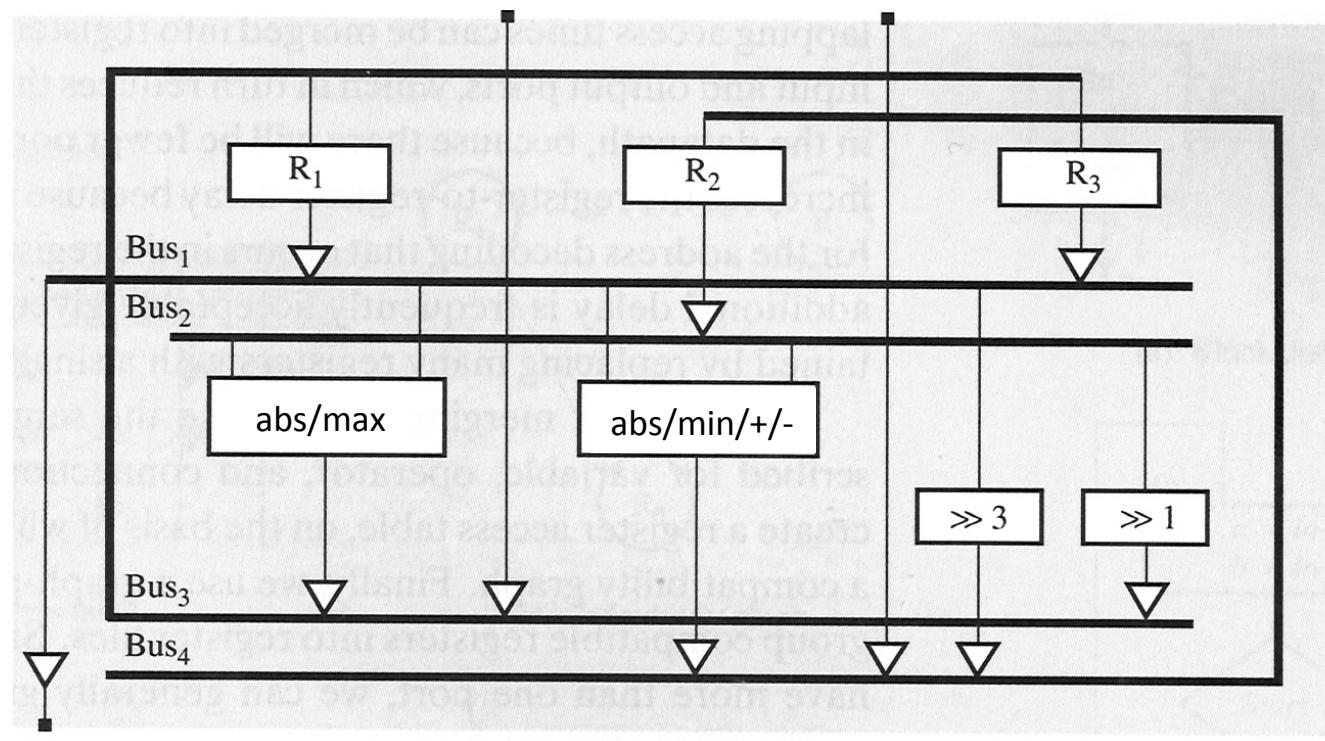
Optimización. Compartir conexión

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A								x
B		x				x		
C	x	x				x		
D		x		x				
E			x		x			
F	x	x		x	x			
G			x					
H			x					
I	x	x				x		
J	x	x		x	x			
K			x					
L			x					
M	x							
N	x							



4. Optimización de ASM

Optimización. Compartir conexión



toc

Up-down y Bottom-up



- ¿Cómo nos acercamos al problema?
 - Diseñamos primero los componentes más sencillos y los unimos para crear componentes más complicados (bottom-up)
 - Diseñamos el comportamiento del circuito a alto nivel y cada vez vamos acercándonos más al diseño hardware (up-down)
 - Diseño algorítmico.

Divide y vencerás



- En teoría de la programación el término divide y vencerás hace referencia a uno de los paradigmas de programación más importante.
 - Éste implica la resolución recursiva de un problema dividiéndolo en dos o más sub-problemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente.
- Podemos aplicar la misma metodología en hardware
 - Simplificando el problema original al dividirlo.
 - Es mucho más fácil y eficiente solucionar problemas pequeños.

Iterativo



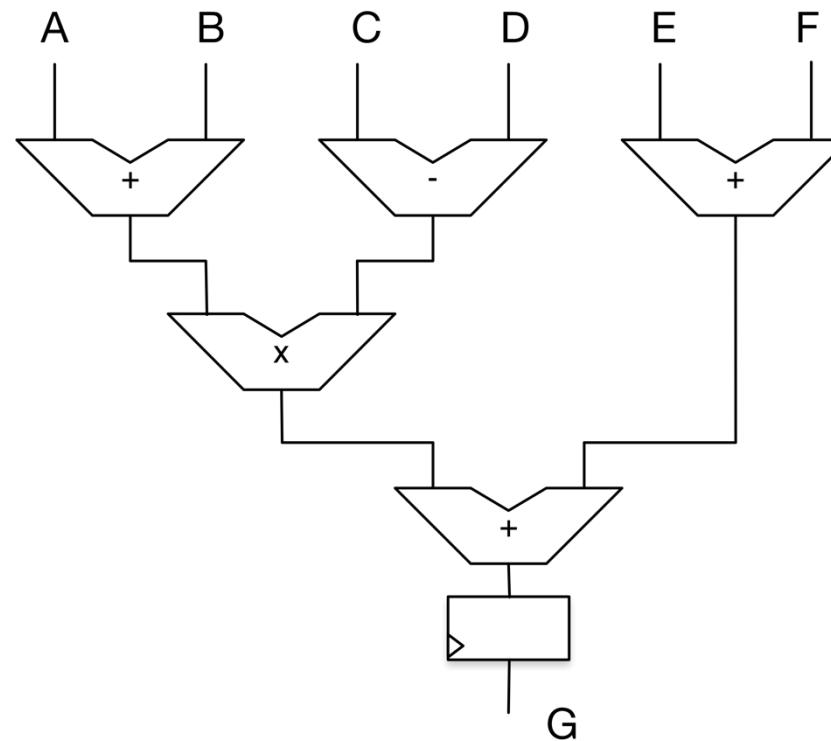
- Se trata de resolver un problema mediante aproximaciones sucesivas a la solución, empezando desde una estimación inicial
 - Deseamos obtener un circuito con un determinado tiempo de ciclo.
 - Deseamos obtener un circuito con una determinado consumo de potencia.
- Partiremos de una solución inicial obtenida según la sección anterior (diseño algorítmico) e iremos modificando los componentes y/o unidad de control para conseguir los objetivos de rendimiento.

Diseño RTL



- El diseño RTL es la solución natural al problema de implementar un sistema algorítmico.
- Se describe el circuito final como un conjunto de registros y unidades funcionales. Se indica en cada ciclo qué registros y que unidades funcionales se están utilizando.
 - Podríamos decir que así funciona la CPU
- En las siguientes ejemplos vamos a crear código VHDL a partir de un diseño RTL.

Diseño RTL. Ejemplo (I)



Diseño RTL. Ejemplo (I)

```
architecture rtl of example_DSP is
    signal s1, r, s2, m, s3 : unsigned(7 downto 0);
begin
    s1 <= a + b;
    r  <= c - d;
    s2 <= e + f;
    m  <= s1*r;
    p_reg : process(clk, rst)
begin
    if rst = '1' then
        f <= (others => '0');
    else
        f <= m + s2;
    end if;
end process p_reg;
end rtl;
```

```
architecture rtl of example_DSP is
begin
    p_reg : process(clk, rst)
begin
    if rst = '1' then
        f <= (others => '0');
    else
        f <= ((a+b) * (c-d)) + e + f;
    end if;
end process p_reg;
end rtl;
```



Diseño RTL. Ejemplo (II)



- ¿Qué obtenemos usando el siguiente código?

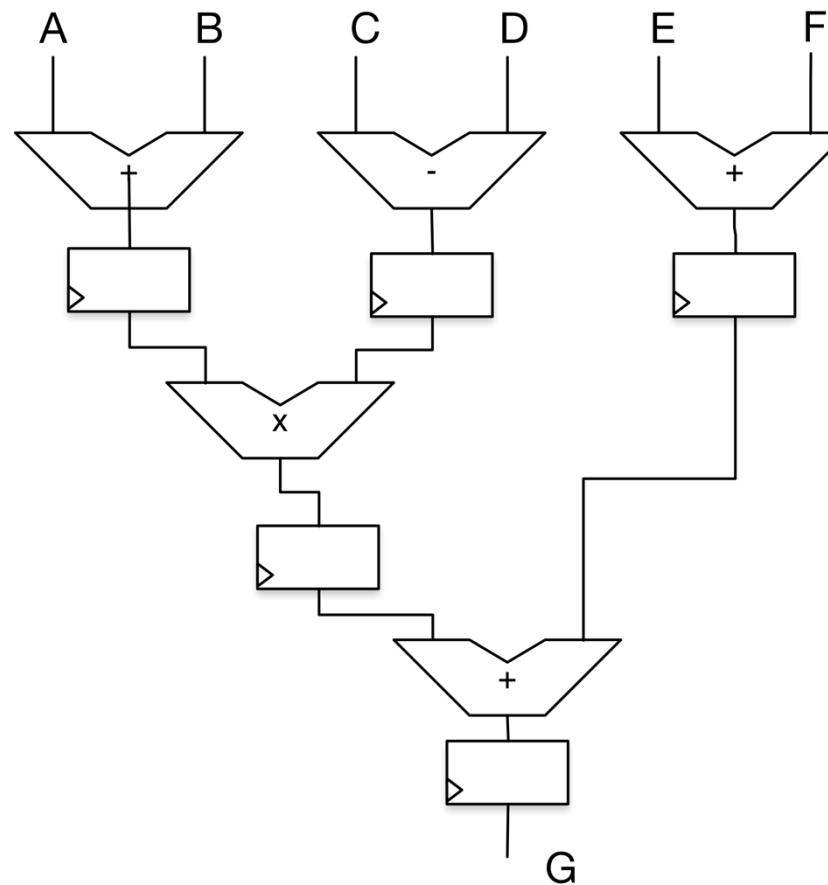
```
architecture rtl of example_DSP is
    signal s1, r, s2, m, s3 : unsigned(7 downto 0);
begin
    p_reg : process(clk,rst)
    begin
        if rst = '1' then
            s1<= (others => '0');
            r <= (others => '0');
            s2 <= (others => '0');
            m <= (others => '0');
            s3 <= (others => '0');

        elsif rising_edge(clk) then
            s1 <= a + b;
            r <= c - d;
            (continua ...)

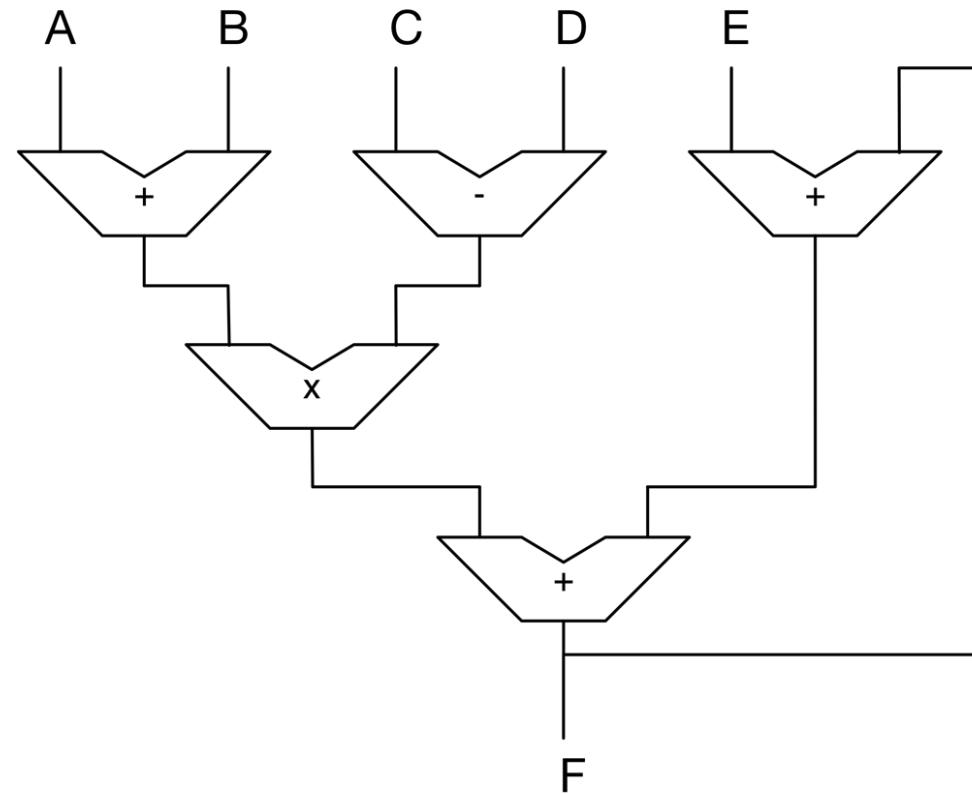
            ...
            s2 <= e + f;
            m <= s1*r;
            s3 <= m + s2;

        end if;
    end process p_reg;
    g <= s3;
end rtl;
```

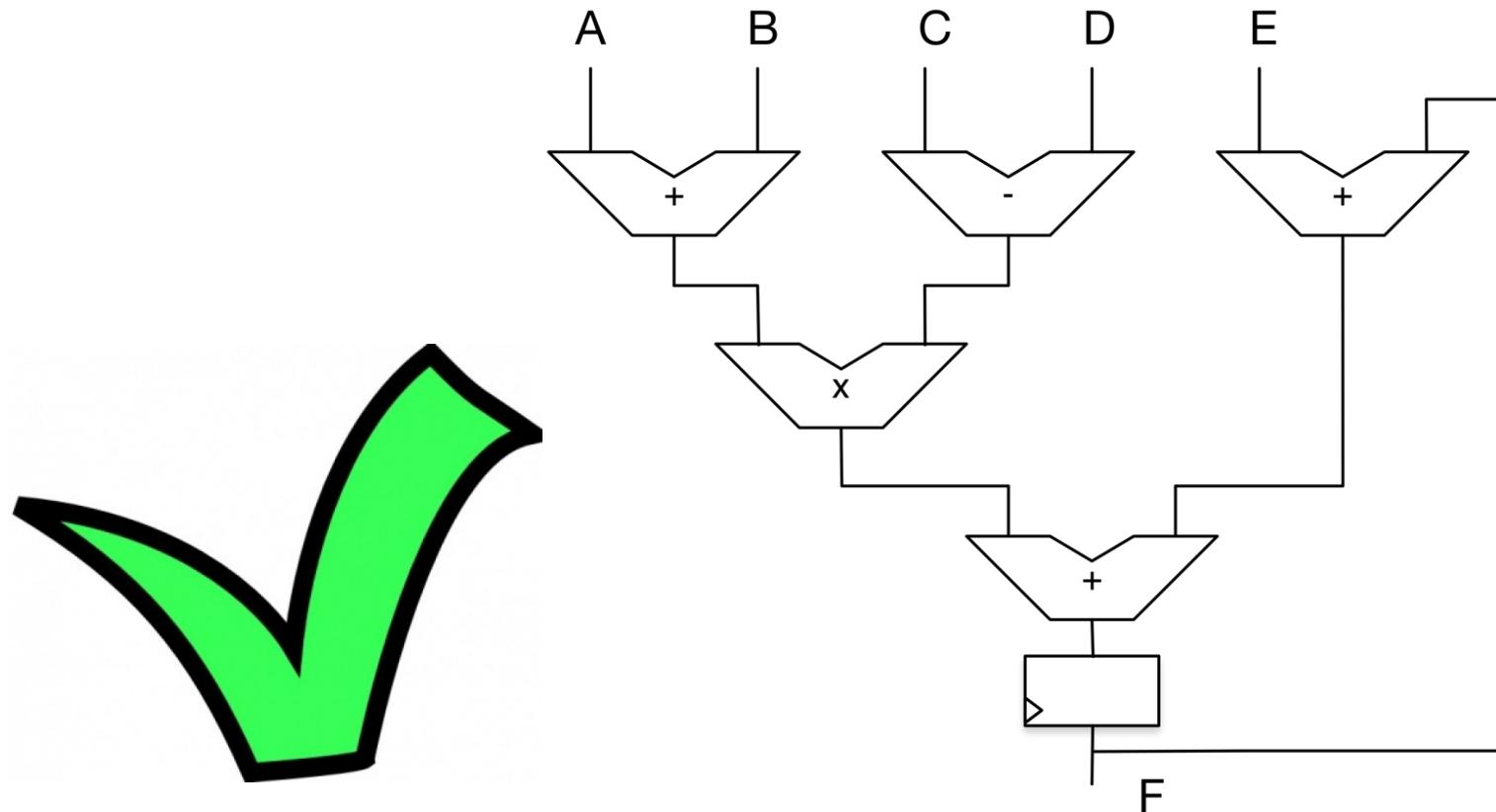
Diseño RTL. Ejemplo (II)



Diseño RTL. Ejemplo (III)



Diseño RTL. Ejemplo (III)



Diseño RTL. Ejemplo (III)

```
architecture rtl of example_DSP is
    signal s1, r, s2, m, s3 : unsigned(7 downto 0);
begin
    s1 <= a + b;
    r  <= c - d;
    s2 <= e + s3;
    m  <= s1*r;
    p_reg : process(clk, rst)
begin
    if rst = '1' then
        s3 <= (others => '0');
    else
        s3 <= m + s2;
    end if;
end process p_reg;
f<= s3;
end rtl;
```

```
architecture rtl of example_DSP is
    signal s3 : unsigned(7 downto 0);
begin
    p_reg : process(clk, rst)
begin
    if rst = '1' then
        s3 <= (others => '0');
    else
        s3 <= ((a+b) * (c-d)) + e + s3;
    end if;
end process p_reg;
f<= s3;
end rtl;
```



Diseño RTL. Ejemplo (IV)



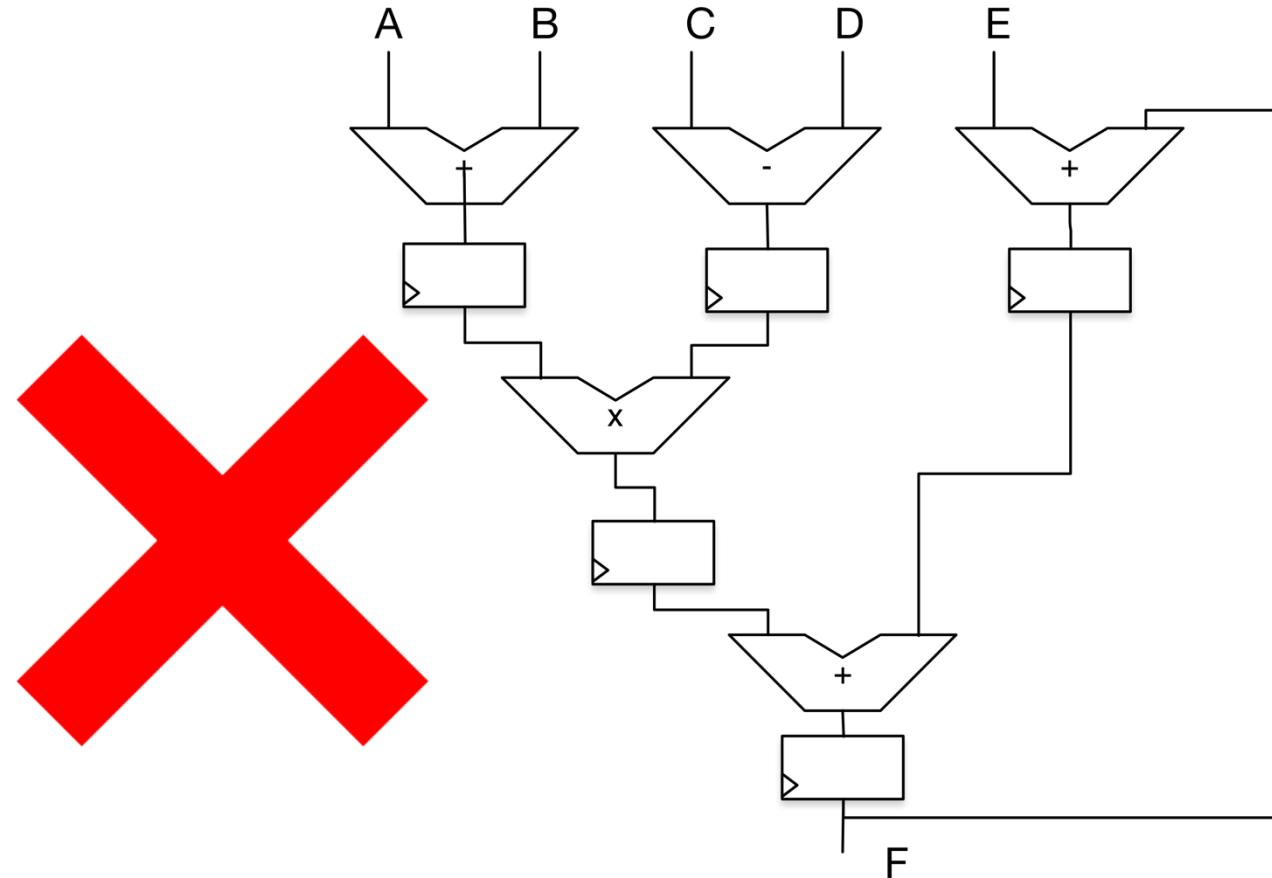
- ¿Qué obtenemos usando el siguiente código?

```
architecture rtl of example_DSP is
    signal s1, r, s2, m, s3 : unsigned(7 downto 0);
begin
    p_reg : process(clk,rst)
    begin
        if rst = '1' then
            s1<= (others => '0');
            r <= (others => '0');
            s2 <= (others => '0');
            m <= (others => '0');
            s3 <= (others => '0');

        elsif rising_edge(clk) then
            s1 <= a + b;
            r <= c - d;
            (continua ...)

            ...
            s2 <= e + s3;
            m <= s1*r;
            s3 <= m + s2;
        end if;
    end process p_reg;
    f<= s3;
end rtl;
```

Diseño RTL. Ejemplo (IV)



Diseño RTL. Ejemplo (V)



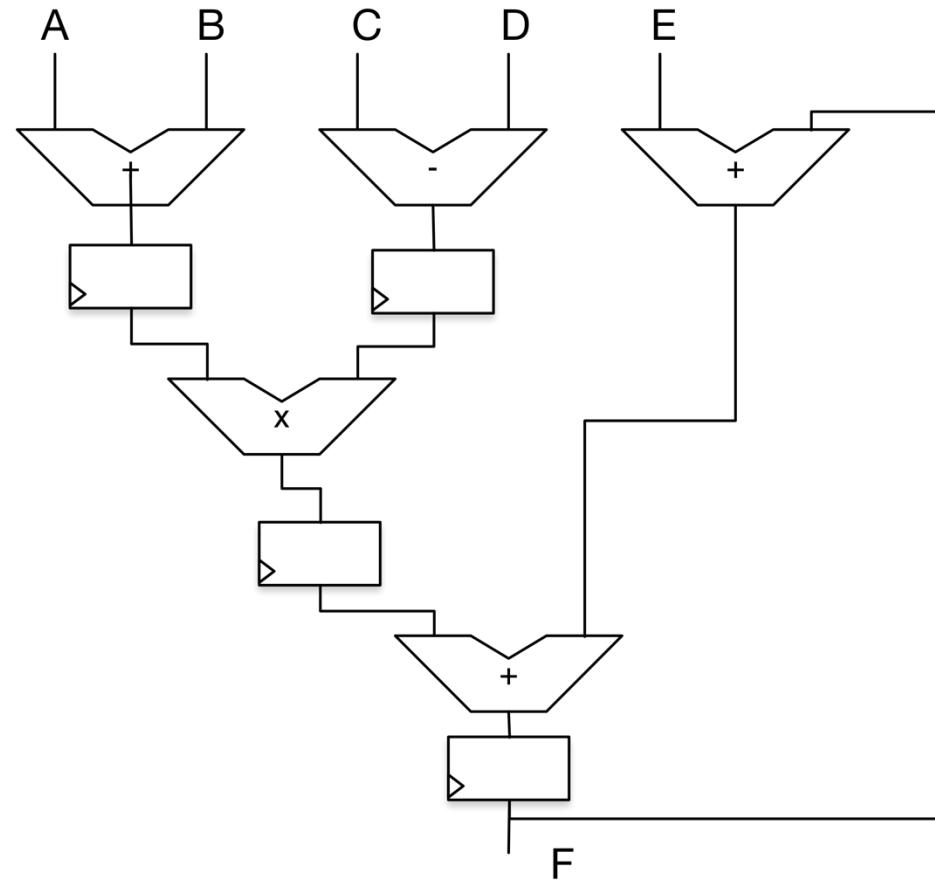
■ ¿Cómo arreglarlo?

```
architecture rtl of example_DSP is
    signal s1, r, s2, m, s3 : unsigned(7 downto 0);
begin
    s2 <= e + s3;
    p_reg : process(clk,rst)
    begin
        if rst = '1' then
            s1<= (others => '0');
            r <= (others => '0');
            m <= (others => '0');
            s3 <= (others => '0');

        elsif rising_edge(clk) then
            s1 <= a + b;
            r <= c - d;
            (continua ...)

            ...
            m <= s1*r;
            s3 <= m + s2;
        end if;
    end process p_reg;
    f<= s3;
end rtl;
```

Diseño RTL. Ejemplo (V)



Diseño RTL



- La solución anterior es rápida pero no permite:
 - Reutilizar hardware
 - Siempre está realizando cálculos => gran consumo de potencia

Diseño RTL



- Implementación ESTRUCTURAL en VHDL

```
process (cuenta)
begin
    case cuenta is
        when 0 =>
            load_RA<='1';
            load_RB<='1';
            load_RC<='1';
            load_RD<='1';
            load_RE<='1';
            load_RS3<='1';
            load_RS1<='0';
            load_RS2<='0';
            load_RR<='0';
            load_RM<='0';

            ...
end process;
-- + COMPONENTES SUMADORES, RESTADOR Y MULTIPLICADOR
-- + CONTADOR MÓDULO 4
```