INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Unidad de Aprendizaje

Arquitectura de Computadoras

Memoria de programa, datos y pila

Abr 2025

Memoria

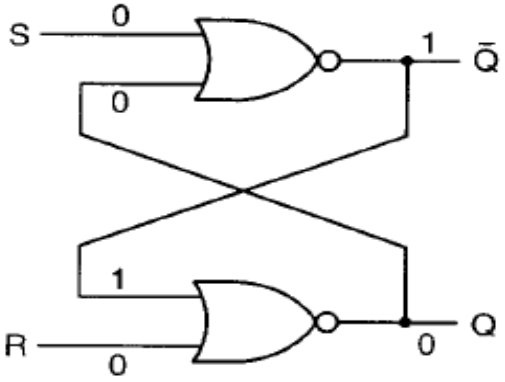
Es un componente indispensable de toda computadora, que sirve para almacenar los datos y las instrucciones que se van ejecutar.

La memoria se puede analizar desde sus componentes básicos al nivel de compuertas, como los latches.

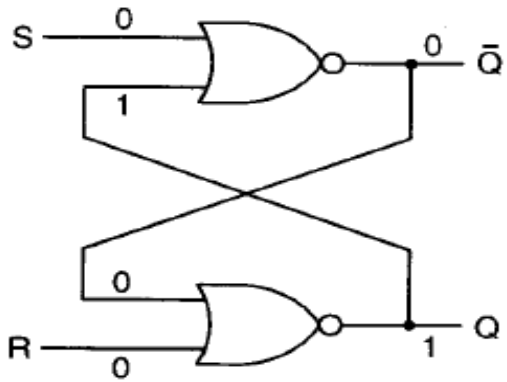
Latches

Una memoria de un bit requiere un circuito que “recuerde” los valores de entrada anteriores.

Latch SR



Latch NOR en el estado 0



Latch NOR en el estado 1

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Tabla de verdad para la NOR

Flip-Flops

En algunos circuitos se requiere muestrear el valor que hay en una línea dada en un determinado instante de tiempo y almacenar ese valor, se utilizan los flip-flops, donde la transición de estado no ocurre cuando el reloj está en 1, sino durante el cambio de 0 a 1 (flanco ascendente o de subida) o de 1 a 0 (flanco descendente o de bajada).

La diferencia entre un flip-flop y un latch es que el primero se dispara por flanco y el segundo por nivel.

Los flip-flops se pueden tener en diversas configuraciones, una de las más sencillas tiene dos flip-flops tipo D independientes con señales de preestablecer y borrar.

Memoria de datos

La memoria de datos se utiliza para almacenar temporalmente la información con la que el procesador está trabajando, como la RAM.

La memoria de datos guarda los valores que manipulan las instrucciones, como variables, resultados intermedios y otros datos necesarios durante la ejecución de un programa.

Memoria de programa

Almacena las instrucciones que el procesador debe ejecutar y conforman el programa que controla el funcionamiento del sistema.

Existen diferentes tipos de memorias de programa, como la ROM donde se almacena código, o la memoria flash, que permite reprogramar y actualizar el contenido.

Memorias RAM

Memoria

•**RAM (Memoria de Acceso Aleatorio):** (Se puede leer y escribirse) Almacena temporalmente los datos y las instrucciones que la CPU necesita en tiempo real. Es volátil, lo que significa que pierde su contenido al apagarse la computadora.

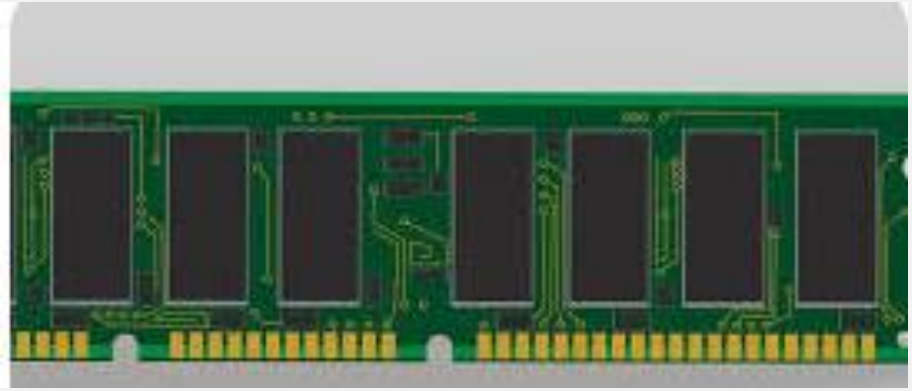
•RAM estática (SRAM, Static RAM) se construye empleando circuitos similares al flip-flop tipo D básico. Su contenido se conserva si está alimentado el circuito. Son memoria muy rápidas, su acceso es de nanosegundos, por eso son consideradas como caché de nivel 2. Requiere de seis transistores por bit.

Memoria RAM

•RAM dinámica (DRAM, Dinamic RAM) no usa flip-flops, es una matriz de celdas, cada una de las cuales contiene un transistor y un condensador que puede cargarse y descargarse progresivamente, lo que permite almacenar ceros y unos.

Nota: Debido a que la carga eléctrica tiende a fugarse, cada bit de este tipo de memoria debe refrescarse (cargarse de nuevo) en pocos milisegundos para evitar que los datos se pierdan.

Las RAM dinámicas tienen una densidad muy alta(muchos bits- chip) por eso se utilizan como memoria principal, aunque la desventaja es la velocidad pues son lentas (decenas de ns).



Memoria RAM

RAM estática (SRAM, Static Random Access Memory-Memoria de acceso aleatorio estática) no usa flip-flops, es un tipo de memoria volátil, lo que significa que pierde su contenido cuando se apaga la alimentación eléctrica. Sin embargo, mientras esté alimentada, los datos son retenidos de manera estable sin necesidad de refrescarse, a diferencia de la DRAM.

La SRAM es más rápida que la DRAM, esto se debe a que no requiere el proceso de refresco y puede acceder a los datos de manera casi instantánea.

En lugar de utilizar un condensador para almacenar información como en la DRAM, la SRAM utiliza un circuito flip-flop (por lo general compuesto por 4 o 6 transistores) para almacenar un bit de información. Este circuito mantiene el estado "1" o "0" de forma estable mientras haya energía.

La SRAM tiende a consumir más energía, debido a que los transistores que componen cada celda requieren estar activos para mantener los datos.

La SRAM tiene una densidad de almacenamiento más baja en comparación con la DRAM, ocupa más espacio en el chip debido al mayor número de transistores que requiere para cada bit.

SRAM Asíncrona: No está sincronizada con un reloj. La lectura y escritura en la memoria se producen de manera asíncrona, lo que significa que los datos pueden ser leídos o escritos en cualquier momento, sin necesidad de una señal de reloj.

SRAM Sincrónica: Aquí, las operaciones de lectura y escritura están sincronizadas con una señal de reloj. Esto permite integrarla más fácilmente en sistemas con controladores de memoria que requieren sincronización con un reloj.

organización de una memoria SRAM

Se refiere a la forma en que se estructuran sus celdas, cómo se organizan las direcciones de memoria y cómo se manejan las operaciones de lectura y escritura.

La celda básica de memoria en una SRAM se construye utilizando un conjunto de transistores. Cada celda de memoria en la SRAM generalmente está compuesta por 4 a 6 transistores. La celda SRAM puede almacenar un bit de información (ya sea 0 o 1). La información se almacena de forma estática en el circuito flip-flop formado por los transistores.

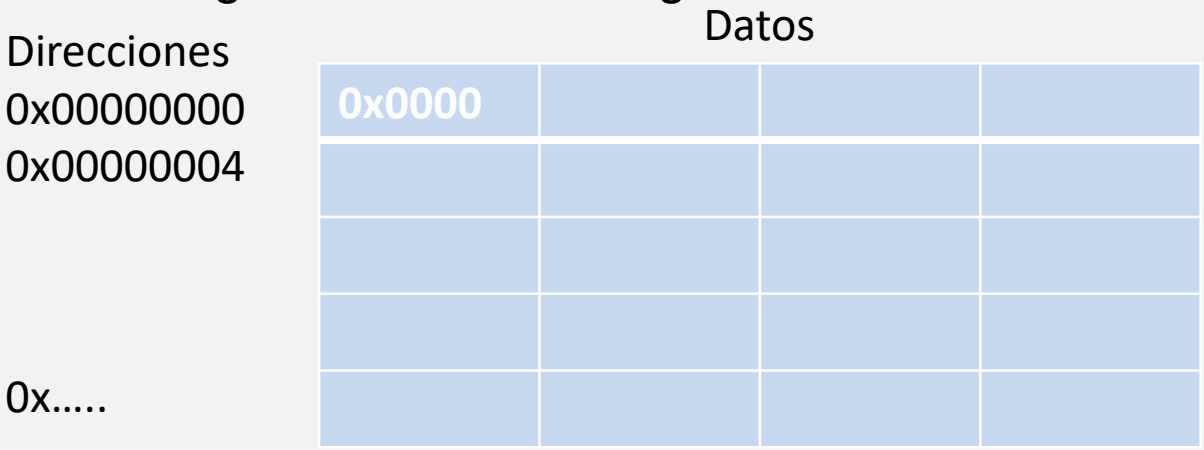
organización de una memoria SRAM

En una memoria SRAM, las celdas de memoria se organizan en una matriz bidimensional o arreglo (array) de celdas, donde cada celda representa un bit de datos.

Las filas corresponden a las **direcciones** de memoria, es decir, el conjunto de posiciones en las cuales se pueden almacenar datos.

Las columnas están formadas por las **líneas de lectura y escritura** de los bits, es decir, los bits que forman el dato (generalmente 8, 16 o 32 bits por dirección, dependiendo del tamaño de la palabra de memoria).

Por ejemplo, si tienes una memoria de 256 palabras de 8 bits cada una, la organización sería algo como:



La memoria SRAM está organizada en una matriz de celdas, pero para facilitar el acceso a estas celdas, se usa un esquema de decodificación de direcciones y líneas de control.

Elementos de memoria:

Dirección (Address): Para acceder a una celda en la memoria, se proporciona una dirección específica. La dirección se decodifica en dos partes:

- Fila (Row): Selecciona qué parte de la memoria se va a acceder (cuál de las filas de celdas se seleccionará).

- Columna (Column): Se utiliza para identificar qué bit dentro de una palabra se va a leer o escribir.

Datos:

- Entrada de datos (Data_in): Los datos que se escriben en la memoria.
- Salida de datos (Data_out): Los datos que se leen de la memoria.

Señales de Control

Señal de escritura (Write Enable o WE): Controla si los datos se escriben o no en la memoria.

Señal de lectura (Read Enable o RE): Controla si los datos se leen de la memoria.

Reloj (Clock): El reloj se usa para controlar las operaciones de lectura y escritura.

Reset: Puede usarse para reiniciar la memoria.

Las operaciones básicas que se pueden realizar en una memoria SRAM son la lectura y la escritura.

Ambas operaciones dependen de las señales de control y de la dirección proporcionada.

Escritura: Cuando se quiere escribir datos en la memoria, se proporciona una dirección y los datos que se desean almacenar. Además, se activa la señal de escritura (WE), lo que indica que el sistema debe almacenar los datos en la celda correspondiente a la dirección especificada.

Proceso de escritura: Aquí se recibe una dirección de memoria. Los datos a escribir se colocan en la entrada de datos (Data_in). La señal de escritura (WE) se activa, permitiendo que los datos sean almacenados en la dirección indicada.

Lectura: Para leer datos de la memoria, se proporciona la dirección de memoria deseada y se activa la señal de lectura (RE). Los datos almacenados en esa dirección se colocan en la salida de datos (Data_out).

Proceso de lectura: Se recibe una dirección de memoria.La señal de lectura (RE) se activa.Los datos almacenados en la celda correspondiente a esa dirección se colocan en la salida de datos (Data_out).

Decodificación de Direcciones

La decodificación de direcciones es un proceso clave en la organización de una memoria SRAM. Dado que la memoria está organizada en un arreglo de celdas, se requiere un decodificador de direcciones para seleccionar la fila y la columna correspondiente.

Si la memoria tiene 2^n direcciones y cada dirección contiene m bits de datos, el decodificador de direcciones se divide en dos partes

Parte de la dirección de la fila: Se selecciona cuál de las filas debe accederse.

Parte de la dirección de la columna: Se selecciona el bit dentro de la palabra.

Por ejemplo, en una memoria de 256 direcciones (direcciones de 8 bits cada una) se necesitaría un decodificador de 8 bits de dirección ($256 = 2^8$) y se organizaría de tal forma que cada dirección acceda a una fila específica y a una columna que contiene los bits de datos.

El tamaño de la memoria y la estructura de sus direcciones dependen de la cantidad de palabras que la memoria puede almacenar y del tamaño de la palabra (el número de bits por palabra). Algunos ejemplos típicos incluyen:

Memoria SRAM de 32K x 8

Esto significa que la memoria tiene 32,768 direcciones, y cada dirección puede almacenar 8 bits de datos (1 byte).

Memoria SRAM de 64K x 16

Esto significa que la memoria tiene 65,536 direcciones, y cada dirección puede almacenar 16 bits de datos (2 bytes).

En FPGAs, hay dos formas principales de implementar memoria RAM:

1. Lógica distribuida (Distributed RAM)

Utiliza las **LUTs (Look-Up Tables)** del FPGA como memoria, es muy útil para pequeñas memorias (por ejemplo, registros, buffers) y ocupa recursos lógicos (LUTs) y lógica distribuida, lo cual puede afectar el resto del diseño si se usa un gran número de éstos. Cuando la memoria es pequeña se infiere por defecto este tipo de lógica, si no se indica otra cosa.

2. Bloques de RAM embebida (Block RAM o EBR en Lattice)

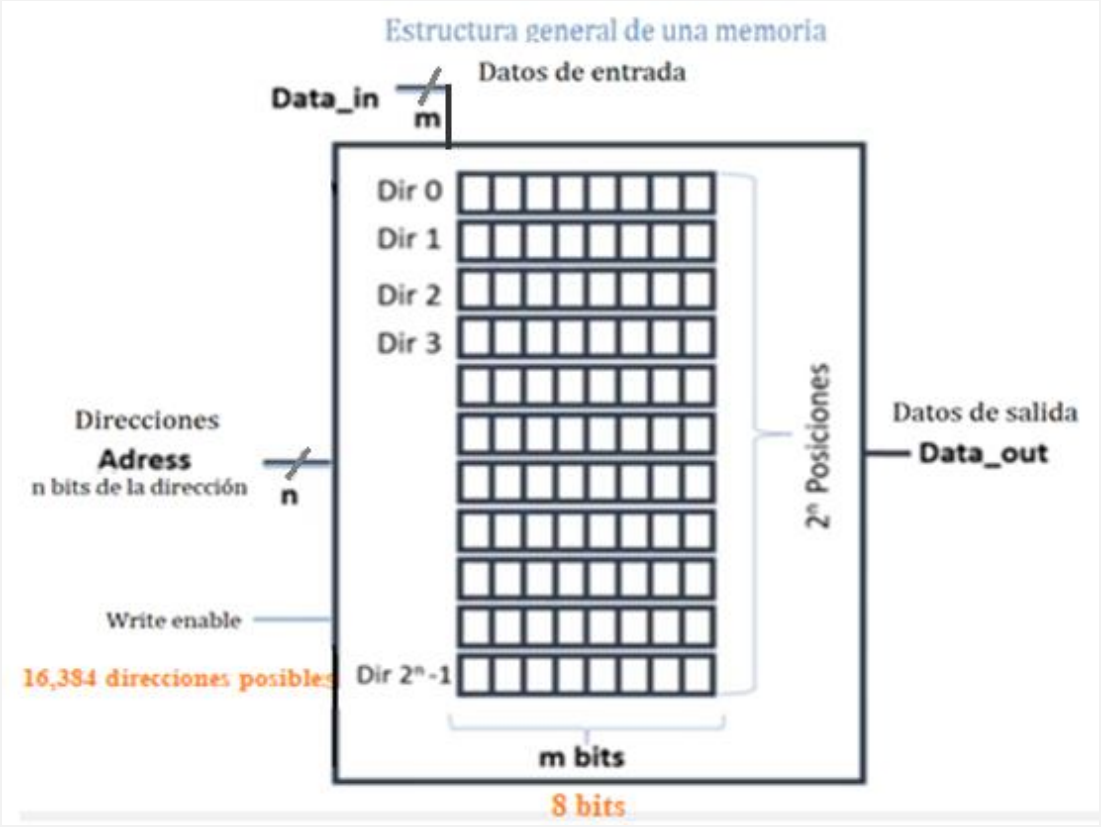
Son bloques dedicados de RAM dentro del FPGA (RAM embebida), no utilizan LUTs. Estos bloques son más eficientes para memorias grandes como por ejemplo una 16K x 8, el acceso es más rápido y menor el consumo de lógica.

Ejemplo:

En la **estructura** de una **memoria SRAM de 16K x 8** se tienen:

- **16K** significa que tenemos **16,384 direcciones posibles**.
- **8 bits por dirección** significa que cada dirección puede almacenar **8 bits** (1 byte).

Por lo que, la memoria tiene **16,384 ubicaciones de memoria**, y cada una de esas ubicaciones puede almacenar **8 bits**.



En teoría, para acceder a una celda o casilla de memoria se requiere una dirección única.

El número total de direcciones (**16K**) es **16,384**. Esta cantidad se expresa como $\mathbf{2^{14}}$, lo que significa que se necesitan **14 bits** para representar todas esas direcciones.

Esto quiere decir, que la **dirección completa** que se utiliza para acceder a una celda en esta **memoria** de **16 K** será de **14 bits**, los cuales van a determinar a qué fila o columna accesar.

Para memorias grandes (tamaño en Ks) su implementación en FPGAs, es posible usar lógica distribuida.

Para hacer accesos más eficientes, se puede trabajar con lógica distribuida, donde la clave está en cómo se usa esa dirección de 14 bits para acceder a las celdas de memoria.

En lugar de tener una dirección única que apunte directamente a una celda en la memoria, la dirección se divide en dos partes:

Fila: Una parte de los 14 bits se usa para seleccionar qué fila de memoria queremos acceder.

Columna: La otra parte de los 14 bits se usa para seleccionar qué columna dentro de esa fila.

La memoria está organizada en una **matriz bidimensional**. Si visualizamos una matriz de celdas, esta tiene filas y columnas.

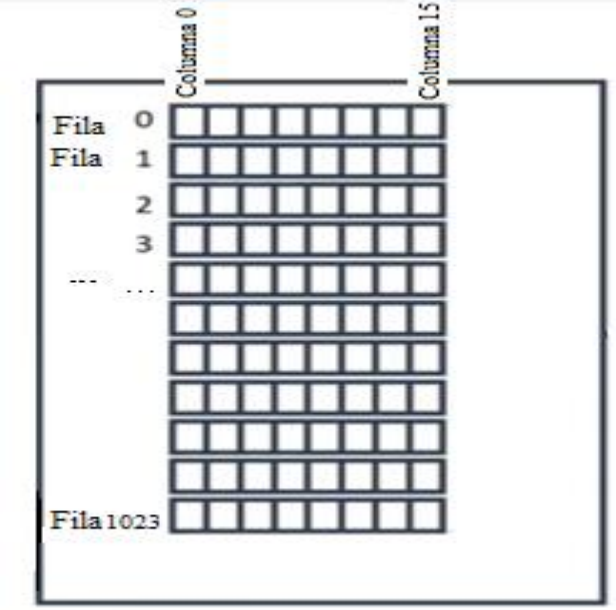
Para dividir los bits de dirección, en este caso, se dividen los 14 bits de dirección en dos partes:

Parte de la fila: Se usan 10 bits para seleccionar qué fila se quiere. 2^{10}=1024 filas.

Parte de la columna: Se usan 4 bits para seleccionar la columna dentro de esa fila, 2^4 =16 columnas.

Entonces, con 10 bits para las filas, se tienen **1024 filas**, y con 4 bits para las columnas, tenemos **16 columnas** dentro de cada fila.

Esto da una matriz de **1024 filas** y **16 columnas**, por lo que se tiene un arreglo de celdas de memoria con 1024 x 16 celdas, y cada celda puede almacenar 8 bits.



Cuando se proporciona una **dirección de 14 bits**, por ejemplo, **10110011001101**, los primeros 10 bits de esta dirección se usan para seleccionar la fila y los últimos 4 bits se usan para seleccionar la columna dentro de esa fila.

Fila: Se seleccionan los primeros 10 bits para elegir cuál de las 1024 filas de memoria usar. **(1011001100)**

Columna: Los últimos 4 bits de la dirección seleccionan cuál de las 16 columnas de esa fila acceder **(1101)**

La dirección binaria 10110011001101 accede a:
Fila 716 y Columna 13

Dividir la dirección en fila y columna permite que el acceso a la memoria sea más eficiente, utilizando decodificadores de fila y columna para acceder a las celdas rápidamente sin tener que acceder a cada celda de manera secuencial.

Ejemplo: Implementación RAM dividida en fila –columna (usando lógica distribuida)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RAM_Distribuida is
    Port (
        clk                 : in  STD_LOGIC;
        we                  : in  STD_LOGIC;
        fila      : in  STD_LOGIC_VECTOR(9 downto 0);  -- 10 bits = 1024 filas
        columna             : in  STD_LOGIC_VECTOR(3 downto 0);  -- 4 bits = 16 columnas
        data_in             : in  STD_LOGIC_VECTOR(7 downto 0);
        data_out            : out STD_LOGIC_VECTOR(7 downto 0)
    );
end RAM_Distribuida;

architecture Behavioral of RAM_Distribuida is

    -- Tipos de la RAM: 1024 filas × 16 columnas (8 bits por posición) → Total 16384 bytes
    type matriz_col is array (15 downto 0) of  STD_LOGIC_VECTOR(7 downto 0); -- 16 columnas
    type matriz_ram is array (0 to 1023) of matriz_col;                      -- 1024 filas
```

```
 --Declaración de la RAM
   signal memoria : matriz_ram := (others => (others => (others => '0'))); -- Inicialización

 --  Aquí se coloca el atributo sis equiere explícitamente indicar al sintetizador que
 --  implemente la RAM en LUTs (no en BRAM)
   attribute ram_style : string;
   attribute ram_style of memoria : signal is "distributed";

   signal dato_leido : STD_LOGIC_VECTOR(7 downto 0);

begin
   process(clk)
   begin
      if rising_edge(clk) then
         if we = '1' then
            memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna))) <= data_in;
         end if;
         dato_leido <=
memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna)));
      end if;
   end process;

   data_out <= dato_leido;
end Behavioral;
```

El programa crear una memoria RAM de 1024 × 16 posiciones, donde cada celda almacena 8 bits, y no utiliza bloques de BRAM del FPGA, sino que se implementa en LUTs (lógica distribuida).

La dirección de memoria se divide en dos partes: Fila: 10 bits → 2^{10} = 1024 filas y Columna: 4 bits → 2^4 = 16 columnas por fila.

El programa incluye las **bibliotecas estándar** para manejar señales lógicas (STD_LOGIC) y conversiones numéricas (NUMERIC_STD).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

En la **entidad** se definen los puertos

```
entity RAM_Distribuida is
    Port (
        clk             : in  STD_LOGIC;
        we              : in  STD_LOGIC;
        fila            : in  STD_LOGIC_VECTOR(9 downto 0);
        columna         : in  STD_LOGIC_VECTOR(3 downto 0);
        data_in         : in  STD_LOGIC_VECTOR(7 downto 0);
        data_out        : out STD_LOGIC_VECTOR(7 downto 0)
    );
end RAM_Distribuida;
```

Señal	Dirección	Descripción
clk	Entrada	Reloj
we	Entrada	Habilita escritura (write enable).
fila	Entrada	Dirección de fila (10 bits).
columna	Entrada	Dirección de columna dentro de la fila (4 bits).
data_in	Entrada	Dato a escribir (8 bits).
Data_out	Salida	Dato leído de la memoria.

En la **Arquitectura** se realiza la declaración de la memoria en dos niveles (fila-columna), donde la memoria comienza con ceros en la inicialización.

```
type matriz_col is array (15 downto 0) of STD_LOGIC_VECTOR(7 downto 0); -- 16 columnas
type matriz_ram is array (0 to 1023) of matriz_col;                     -- 1024 filas

signal memoria : matriz_ram := (others => (others => (others => '0')));
```

Se utiliza una señal para mantener el valor leído antes de asignarlo a data_out (señal de salida interna)

```
signal dato_leido : STD_LOGIC_VECTOR(7 downto 0);
```

El proceso principal se ejecuta ene el flanco de subida del reloj con rising_edge(clk), cuando we=1, se escribe data_in en la dirección dada por fila y columna. Después se lee el dato de esa misma posición y se guarda en dato_leído.
Para finalizar, el data_out recibe el valor leído.
La lectura se realiza en el mismo ciclo que la escritura pero se ve reflejada en la salida en el siguiente flanco de reloj.

```
process(clk)
begin
   if rising_edge(clk) then
      if we = '1' then

memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna))) <=
data_in;
      end if;

      dato_leido <=
memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna)));
   end if;
end process;

data_out <= dato_leido;
```

El programa del ejemplo evita la inferencia de BRAM, por lo que no utiliza arreglos planos grandes como:

ram array (0 to 16383) of std_logic_vector(7 downto 0);
puesto que esto, tiende a BRAM.

Usa estructuras anidadas (fila y columna) con matrices pequeñas, por lo que el sintetizador lo implementará con LUTs distribuidas.

Si se usa FPGAs de Xilinx, se pueden aplicar atributos como:
attribute ram_style : string;
attribute ram_style of memoria : signal is "distributed";

Con éstos atributos se indica de forma explícita que no debe usar BRAM.

Trabajar un **testbench** en VHDL permite simular y verificar que la memoria RAM diseñada está funcionando correctamente, sin necesidad de usar hardware real. Se puede usar ModelSim, Vivado Simulador, GHDL.

Testbench

1. Escribir un valor en una dirección específica
2. Leer ese valor para verificar que se escribió correctamente
3. Realizar otras pruebas con múltiples ubicaciones (Fila-columna)
4. Verificar que ocurre cuando no se escribe, es decir cuando we=0.

Recordar que:

En los ciclos donde se escribe, el dato en data_out se visualiza hasta el próximo flanco de reloj.
En los ciclos donde we=0, el data_out debe reflejar el valor almacenado en esa posición de fila-columna.

Ejemplo de un Testbench para memoria RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TB_RAM_Distribuida is
end TB_RAM_Distribuida;

architecture Behavioral of TB_RAM_Distribuida is

   -- Component declaration (la unidad bajo prueba)
   component RAM_Distribuida
      Port (
         clk      : in  STD_LOGIC;
         we       : in  STD_LOGIC;
         fila     : in  STD_LOGIC_VECTOR(9 downto 0);
         columna  : in  STD_LOGIC_VECTOR(3 downto 0);
         data_in  : in  STD_LOGIC_VECTOR(7 downto 0);
         data_out : out STD_LOGIC_VECTOR(7 downto 0)
      );
   end component;
```

```
    -- Señales para conectar al DUT
    signal clk_tb      : STD_LOGIC := '0';
    signal we_tb       : STD_LOGIC := '0';
    signal fila_tb     : STD_LOGIC_VECTOR(9 downto 0) := (others => '0');
    signal columna_tb  : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
    signal data_in_tb  : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal data_out_tb : STD_LOGIC_VECTOR(7 downto 0);

    -- Clock period
    constant CLK_PERIOD : time := 10 ns;

begin

    -- Instancia de la memoria
    uut: RAM_Distribuida
        Port Map (
            clk     => clk_tb,
            we      => we_tb,
            fila    => fila_tb,
            columna => columna_tb,
            data_in => data_in_tb,
            data_out => data_out_tb
        );
```

```
-- Generador de reloj
clk_process : process
begin
   while now < 200 ns loop
      clk_tb <= '0';
      wait for CLK_PERIOD / 2;
      clk_tb <= '1';
      wait for CLK_PERIOD / 2;
   end loop;
   wait;
end process;

-- Proceso de estimulación
stim_proc : process
begin
   -- Primer ciclo: escribir en fila 5, columna 3
   fila_tb    <= std_logic_vector(to_unsigned(5, 10));
   columna_tb <= std_logic_vector(to_unsigned(3, 4));
   data_in_tb <= x"AB";
   we_tb      <= '1';
   wait for CLK_PERIOD;
```

```
        -- Segundo ciclo: desactivar escritura, leer lo que hay
        we_tb <= '0';
        wait for CLK_PERIOD;

        -- Tercer ciclo: escribir otro valor en otra posición
        fila_tb    <= std_logic_vector(to_unsigned(15, 10));
        columna_tb <= std_logic_vector(to_unsigned(7, 4));
        data_in_tb <= x"CD";
        we_tb      <= '1';
        wait for CLK_PERIOD;

        -- Leer el valor que escribimos primero (fila 5, col 3)
        we_tb      <= '0';
        fila_tb    <= std_logic_vector(to_unsigned(5, 10));
        columna_tb <= std_logic_vector(to_unsigned(3, 4));
        wait for CLK_PERIOD;
        -- Leer el valor que escribimos después (fila 15, col 7)
        fila_tb    <= std_logic_vector(to_unsigned(15, 10));
        columna_tb <= std_logic_vector(to_unsigned(7, 4));
        wait for CLK_PERIOD;
        -- Terminar simulación
        wait;
    end process;
end Behavioral;
```

Ejemplo de código para programar la BRAM del FPGA

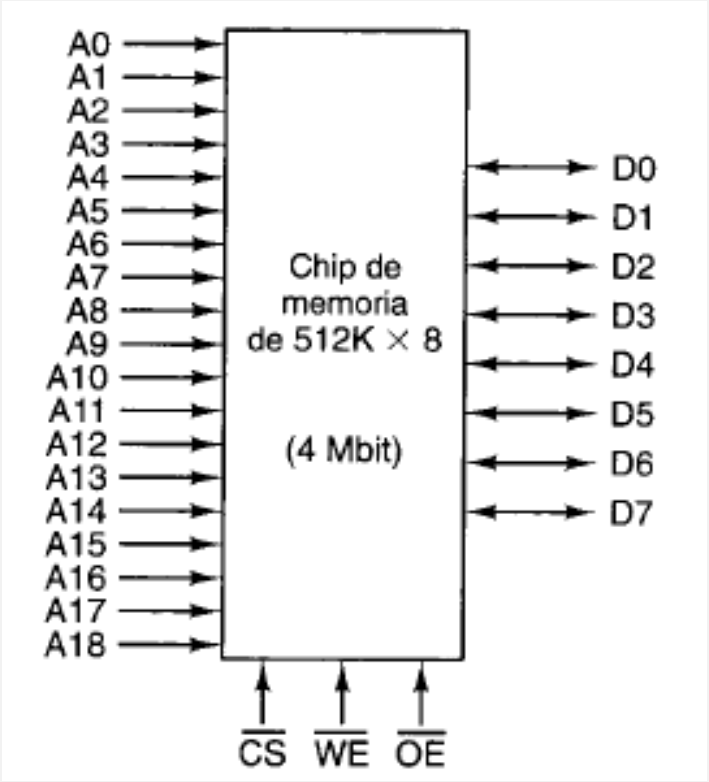
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Memoria_RAM_FilaColumna is
    Port (
        clk       : in  STD_LOGIC;
        we        : in  STD_LOGIC;  -- Señal de escritura
        fila      : in  STD_LOGIC_VECTOR(9 downto 0);         -- Parte fila: 10 bits
        columna   : in  STD_LOGIC_VECTOR(3 downto 0);   -- Parte columna: 4 bits
        data_in   : in  STD_LOGIC_VECTOR(7 downto 0);     -- Dato a escribir
        data_out  : out STD_LOGIC_VECTOR(7 downto 0)    -- Dato leído
    );
end Memoria_RAM_FilaColumna;

architecture Behavioral of Memoria_RAM_FilaColumna is

    -- Definición de la memoria: 1024 filas × 16 columnas (cada posición de 8 bits)
    type fila_type is array (15 downto 0) of STD_LOGIC_VECTOR(7 downto 0); -- 16
columnas
    type ram_type  is array (0 to 1023) of fila_type;                  -- 1024 filas
```

```
signal memoria : ram_type;

begin

   process(clk)
   begin
      if rising_edge(clk) then
         if we = '1' then
            memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna))) <=
data_in;
         end if;
         data_out <=
memoria(to_integer(unsigned(fila)))(to_integer(unsigned(columna)));
      end if;
   end process;

end Behavioral;
```

Chips Memorias RAM

Ejemplo: dos posibles organizaciones de un chip de 4 Mbits:
Un chip de 512 K x 8 y otro chip de 4096 K x 1

Para ello se requieren 19 líneas de dirección para direccionar uno de los 2^{19} bytes, y se necesitan 8 líneas de datos para cargar o almacenar el byte seleccionado.



$\overline{CS}$ *(Chip select)* para habilitar la selección del chip

$\overline{WE}$ $(Write\ Enable)$para habilitar escritura

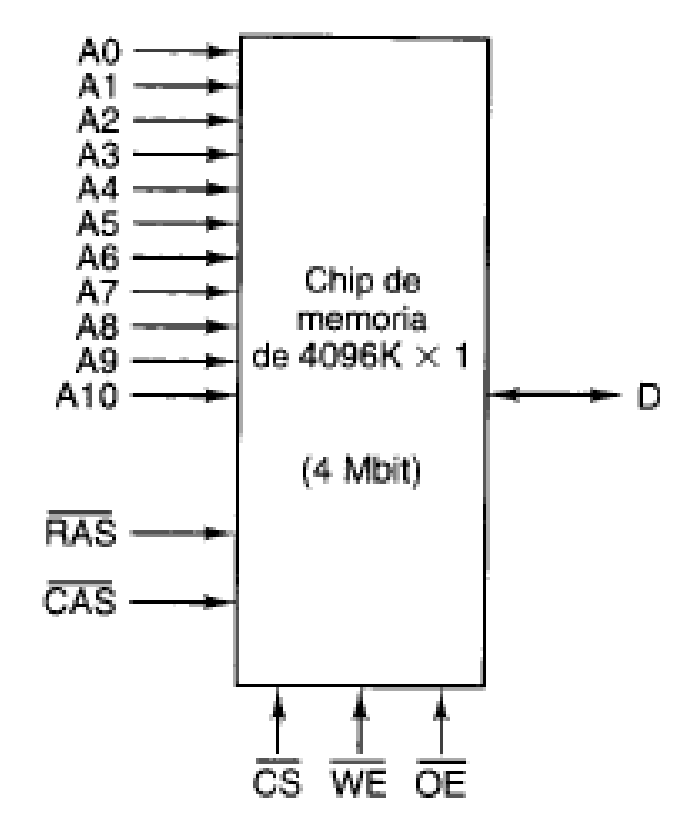
$\overline{OE}$ *(Output Enable)* para habilitar la alimentación de las señales de salida, si no se habilita, la salida del chip se desconecta del circuito (no envía datos a las salidas. No interfiere con otros components del Sistema, cuando se comparten buses de datos). Esto es alta impedancia o tri-state.

Nota: tamaño de memoria se da en bits, no en bytes

Chips Memorias RAM

Para la organización de 4096 x 1, se tiene una matriz de 2048 x 2048 celdas de 1 bit lo que da 4 Mbits.

Primero se selecciona una localidad (renglón) colocando su número de 11 bits en las terminales de dirección.



$\overline{RAS}$ *(Row Address Strobe)* para habilitar dirección de fila.

$\overline{CAS}$ *(Colum Address Strobe)*para habilitar dirección de columna.

Nota: tamaño de memoria se da en bits, no en bytes

Capacidad de una Memoria RAM

Nº de palabras o localidades: 2^n

Bits por palabra: m

} Organización 2^n x m bits

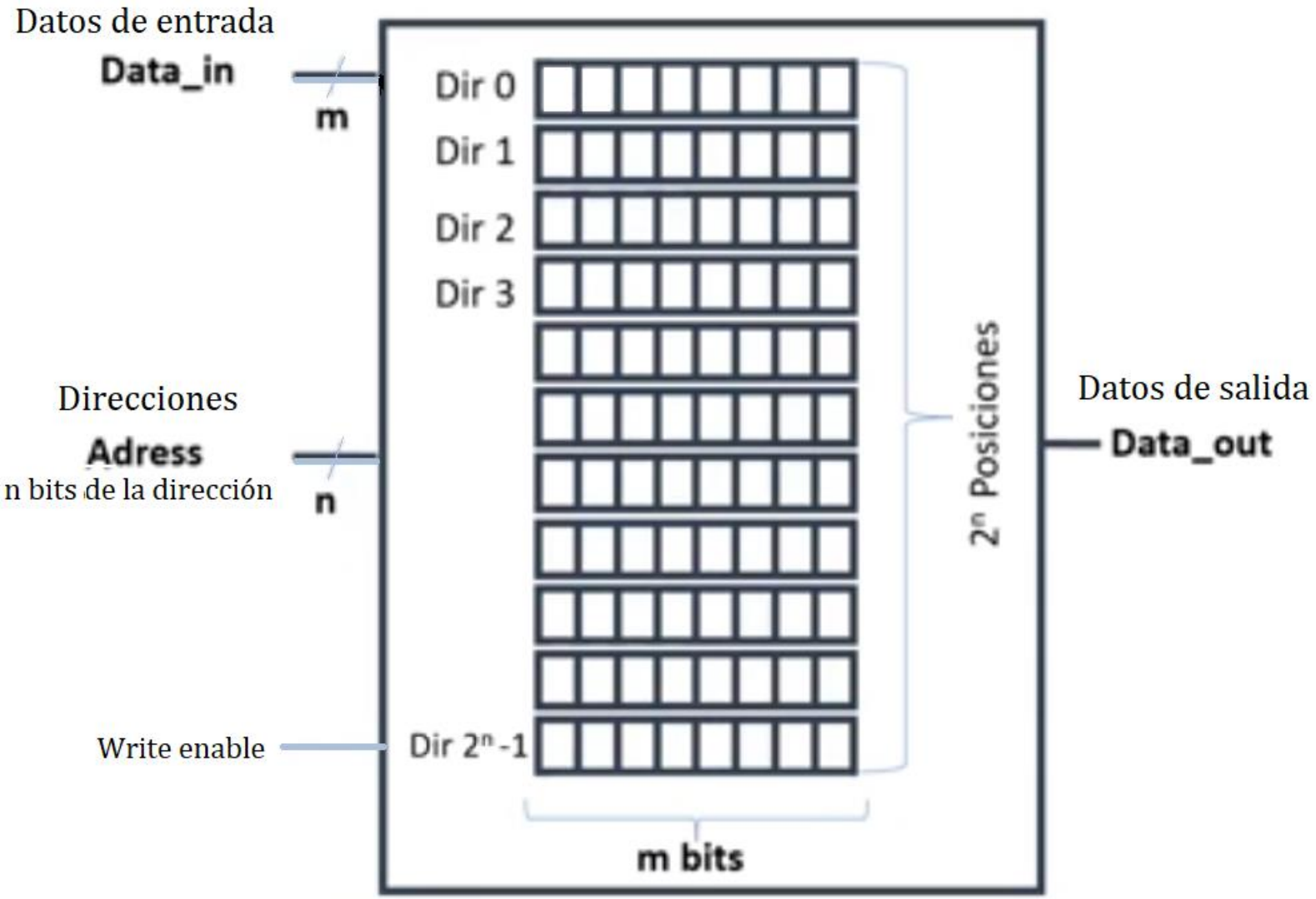
Ejemplo: n=11, m=8

Organización 2^{11} x 8=2k x 8

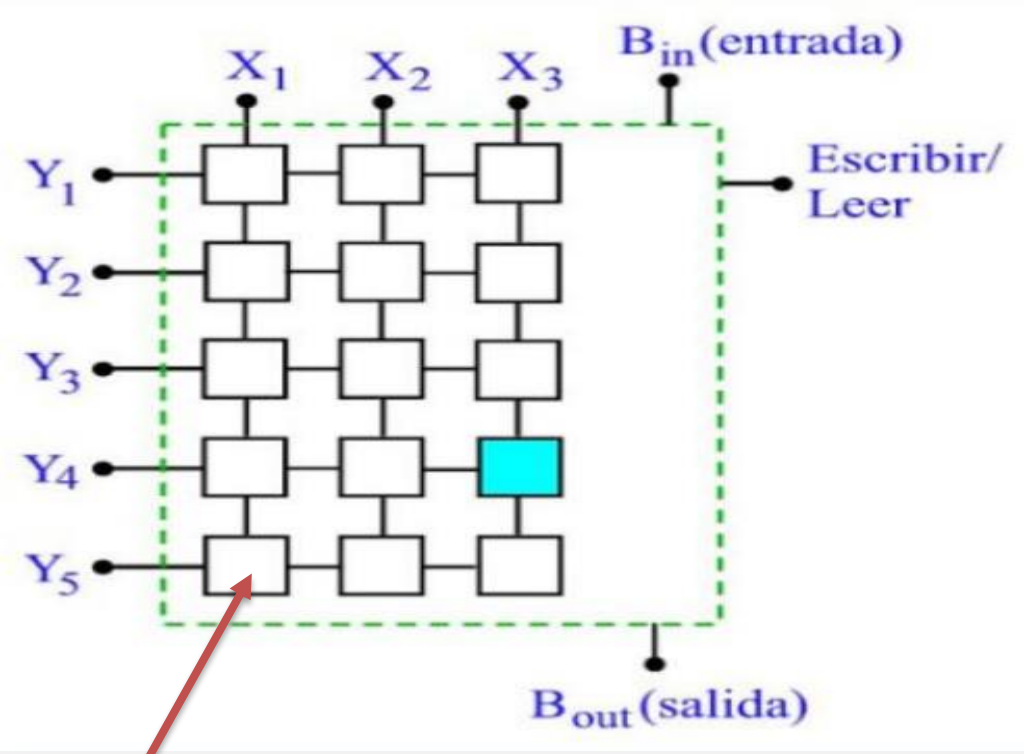
Capacidad 16 Kbits= 16384 bits

n= número de líneas de dirección en binario

Estructura general de una memoria



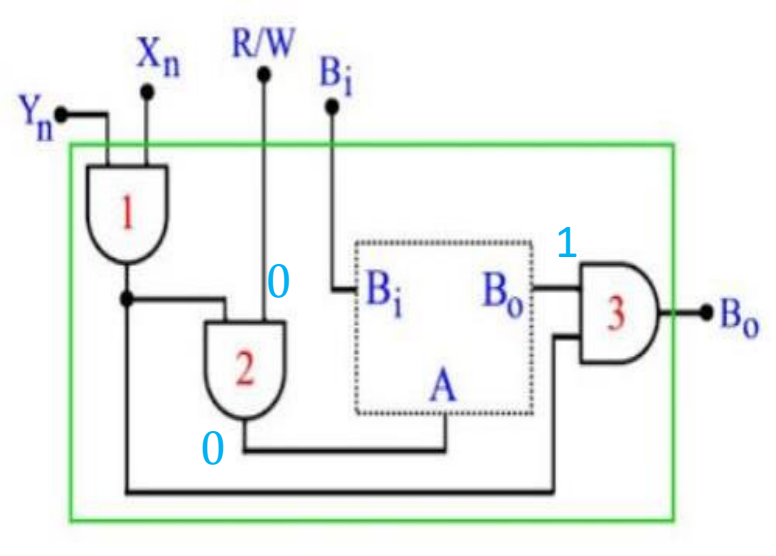
Considerando un arreglo rectangular de flip-flops ordenados en un acomodo rectangular de 5 por 3 cuadros, cada cuadro conteniendo un flip-flop. Cómo se puede localizar un flip-flop determinado en un momento dado. De qué forma se puede introducir y sacar información de ese flip-flop.



celda o
célula básica

En la figura, para encontrar el elemento en azul, se requiere activar (poner a “1” y dejando un “0” en todas las demás) las terminales X3 y Y4.

Bi es el bit que va a ser almacenado por el flip-flop seleccionado y Bo es el bit ya almacenado por el mismo flip-flop .



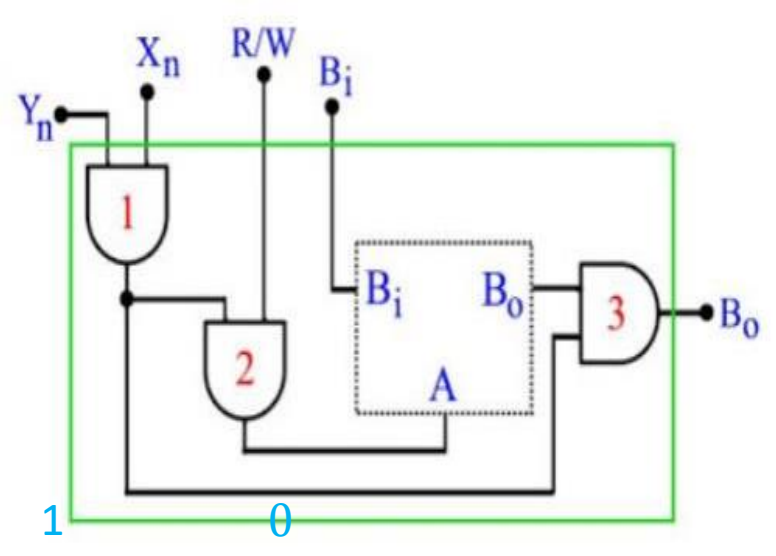
Función R (Read) de lectura

La terminal R/W (Read/Write, Leer/Escribir) está desactivada con un 0, por lo que, la salida del AND2 es 0, lo cual implica que la terminal de acceso A está desactivada y la información contenida por el flip-flop (Bo) permanece inalterada.

Al mismo tiempo, el AND3 está activado y su salida Bo dependerá de la información Bo previamente almacenada en el flip-flop.

Por lo tanto, se está leyendo información de la celda de interés.

Bi es el bit que va a ser almacenado por el flip-flop seleccionado y Bo es el bit ya almacenado por el mismo flip-flop .



Función W (Write) de lectura

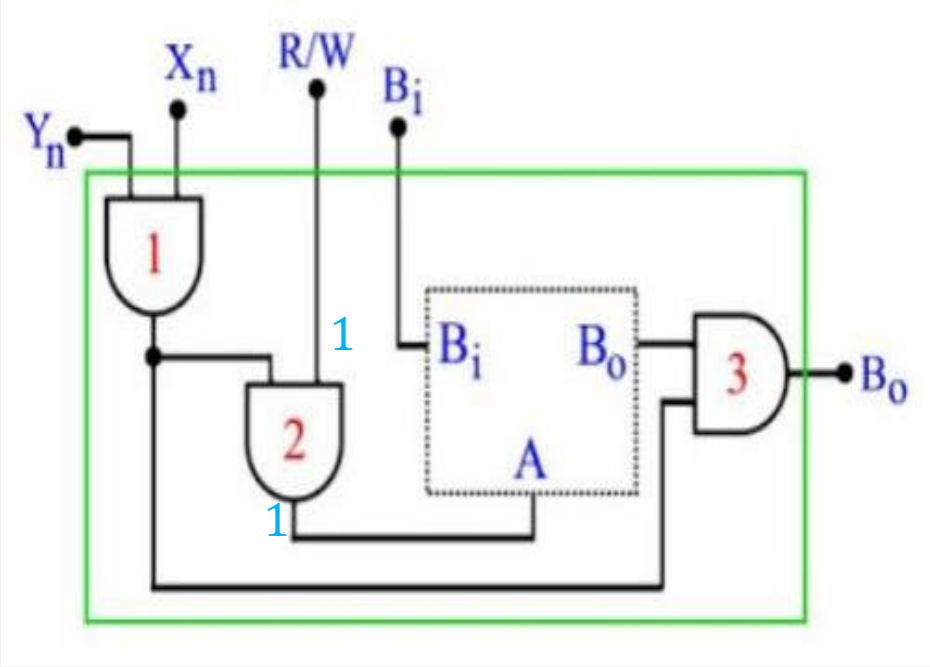
La terminal R/W (Read/Write, Leer/Escribir) está activada con un 1, por lo que, la salida del AND2 es 1, lo cual implica que la terminal de acceso A está activada.

La salida del flip-flop (Bo) que es la salida Bo de la celda depende de la información que tome Bi.

Por lo tanto, se escribe información en la celda.
Para preservar la información se desactiva esta terminal poniendo de nuevo a “0”.

Para encontrar el elemento mostrado se requiere activar (poner a “1” y dejando un “0” en todas las demás) las terminales : X3 y Y4. terminales.

Bi es el bit que va a ser almacenado por el flip-flop seleccionado y Bo es el bit ya almacenado por el mismo flip-flop .



Función W (Write) de lectura

La terminal R/W (Read/Write, Leer/Escribir) está activada con un “1”, por lo que, la salida del AND2 es “1”, lo cual implica que la terminal de acceso A está activada. La salida del flip-flop (Bo) que es la salida Bo de la celda depende de la información que tome Bi. Por lo tanto, se escribe información en la celda.
Para preservar la información se desactiva esta terminal poniendo de nuevo a “0”.

Memorias ROM

•**ROM (Memoria de Solo Lectura):** Almacena el firmware, que es el software fundamental para iniciar la computadora. No es volátil, puesto que no puede modificarse, ni borrarse intencionalmente o por accidente. Los datos en tipo de memoria son grabados durante la fabricación, al exponer un material fotosensible a través de una máscara que contiene el patrón de bits deseado y después eliminado por grabado de la superficie expuesta.

En diversas aplicaciones como aparatos domésticos y electrónicos, juguetes, el automóvil, entre otro, el programa de inicio y otros datos deben conservarse en la memoria aunque se interrumpa el suministro de electricidad.

La ***ROM*** es más económica que la RAM (si se produce en grandes cantidades).

Para facilitar el desarrollo de productos basados en ROM se inventó la ***PROM (Programmable ROM)*** que se puede grabar una vez en campo. Muchas Prom están basadas en una matriz de fusibles que se pueden quemar seleccionando el renglón y la columna, aplicando un voltaje alto a una terminal específica del chip.

El contenido es inalterable una vez programado (con equipo especializado).

Las memorias RPROM (Reprogramable ROM) es posible reprogramarlas borrando previamente su contenido. Según la forma de realizar el borrasdo, se subclasifican en:

EPROM (Erase PROM) se puede programar en campo y borrar en campo. Posee una ventana de cuarzo que si se expone a luz ultravioleta por 15 min aproximadamente todos los bits se ponen en 1.
Una clase de EPROM es la memoria ***Flash***, esta memoria puede borrarse y reescribirse por bloque. Y sin extraerla del circuito dónde esté trabajando.

EEPROM o E2PROM (Electrically EPROM) borrables eléctricamente, puede borrarse por aplicación de pulsaciones, sin tener que exponerla a luz UV. La desventaja de ésta es que, tiene poca capacidad (1/64 que las EPROM).

A diferencia de la RAM, en la implementación de la memoria **ROM no hay escritura** (**we** ni **data_in**).

Los **datos** están **predefinidos** (**se cargan en la declaración**). Solo se necesita la **señal de dirección** y una **salida** (**data_out**).

La lectura se realiza por flanco de subida de reloj (clk)

El atributo rom_style= “distributed” fuerza al uso de LUTs.

Ejemplo de implementación de ROM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ROM_16x8 is
    Port (
        clk      : in  STD_LOGIC;
        addr     : in  STD_LOGIC_VECTOR(3 downto 0); -- 4 bits que seleciona 1 de las  16 dir
        data_out : out STD_LOGIC_VECTOR(7 downto 0)  -- dato de 8 bits
    );
end ROM_16x8;

architecture Behavioral of ROM_16x8 is
```

```
-- Declaración de la ROM
type rom_array is array (0 to 15) of STD_LOGIC_VECTOR(7 downto 0);

-- La señal memoria_rom contiene los datos predefinidos
signal memoria_rom : rom_array := (
   0  => x"01",
   1  => x"02",
   2  => x"03",
   3  => x"04",
   4  => x"05",
   5  => x"06",
   6  => x"07",
   7  => x"08",
   8  => x"09",
   9  => x"0A",
   10 => x"0B",
   11 => x"0C",
   12 => x"0D",
   13 => x"0E",
   14 => x"0F",
   15 => x"10"
);
```

```
    -- Forzar implementación distribuida (en LUTs)
    attribute rom_style : string;
    attribute rom_style of memoria_rom : signal is "distributed";

    signal dato_leido : STD_LOGIC_VECTOR(7 downto 0);

begin

    process(clk)
    begin
        if rising_edge(clk) then
            dato_leido <= memoria_rom(to_integer(unsigned(addr)));
        end if;
    end process;

    data_out <= dato_leido;

end Behavioral;
```

Memorias ROM multipuerto

La ***ROM multipuerto*** tiene varios puertos de acceso, lo que permite que múltiples procesadores o unidades de control puedan acceder al mismo tiempo. En una ***ROM multipuerto***, cada puerto tiene acceso a una parte de la memoria para realizar lecturas sin interferir con otras operaciones.

Características

Acceso concurrente: Permite que múltiples unidades de procesamiento puedan leer datos de la memoria al mismo tiempo sin que haya conflictos.

Almacenamiento no volátil: En una ROM multipuerto se pueden almacenar instrucciones o datos esenciales para el sistema, y los puertos múltiples permiten que varias unidades del sistema o de control accedan a estos datos simultáneamente

Memorias ROM multipuerto

Tipos ROM multipuerto

Dual-port ROM: Permite dos operaciones simultáneas de lectura o escritura en diferentes partes de la memoria.

ROM de acceso compartido: Puede ser utilizada por varios dispositivos o procesadores para acceder de manera simultánea a información crucial almacenada en la memoria ROM sin bloquear el acceso entre los diferentes puertos.

Análisis con ROM y RAM multipuerto

Ventajas

1.Mejor rendimiento y velocidad:

1. Permiten **acceso simultáneo a múltiples datos**, lo que **reduce el tiempo de espera** para operaciones concurrentes.

2.Paralelismo:

1. En sistemas que requieren **operaciones simultáneas** (como en arquitecturas multiprocesador o dispositivos de red), permiten que cada componente realice sus tareas sin tener que esperar a que otro termine, mejorando la capacidad de procesamiento global.

3.Flexibilidad en aplicaciones:

- Son ideales para **sistemas en tiempo real.**
- Mejoran la **eficiencia en sistemas embebidos** o de **comunicaciones**, donde múltiples unidades pueden necesitar acceder a los mismos datos sin interferencias.

Análisis con ROM y RAM multipuerto

Desventajas

1. **Costo y complejidad**:
   Las **memorias multipuerto** son más costosas de fabricar que las tradicionales, debido a la **complejidad** adicional que implica agregar puertos de acceso y la lógica necesaria para gestionar el acceso concurrente.
2. **Consumo de energía**:
   Si bien las memorias multipuerto mejoran el rendimiento, pueden consumir más **energía** que las memorias de un solo puerto, ya que requieren una mayor cantidad de **circuitos de acceso y control**.
3. **Latencia**:
   Aunque el acceso concurrente puede mejorar el rendimiento, también puede generar **congestionamiento** si no se gestiona adecuadamente, lo que puede resultar en **latencia adicional** si muchos componentes intentan acceder a la memoria al mismo tiempo.

Implementación de Memoria

Especificación de necesidades de la memoria

1. *Tamaño de la memoria*
   - No de instrucciones o palabras de datos que se van a almacenar (depende del número de bits por instrucción y dela cantidad total de instrucciones)
2. *Tipo de acceso a la memoria*
   - Acceso secuencial (ROM) o aleatorio (RAM)
3. *Ancho de la palabra*
   - Bits por palabra (depende de la arquitectura del sistema, ya sea 8 bits, 16 bits o 32 bits.
4. *Velocidad*
   - Tiempo que va a tardar el acceso a una palabra de la memoria
5. *Tecnología de almacenamiento*
   - FPGA o CPLD

Implementación de Memoria

Definir la dirección y el acceso

(Se define como se realiza el acceso a la memoria definiendo las señales de entrada y salida)

1. **Señales de dirección**

   Se requiere un bus de direcciones para especificar la ubicación de memoria que se va a leer. Para una memoria de N palabras se requiere log_2 (N). No. de bits=log_2 (N)

2. **Señales de datos**

   La memoria requiere un bus de datos de longitud X (X es el tamaño de la palabra.

3. **Control de lectura/escritura**

   Para una ROM se requiere sólo una señal de lectura (Para la RAM se requiere la señal de lectura y escritura).

4. **Control de lectura/escritura**

   Se requiere una señal de reloj para sincronizar las operaciones de la memoria.

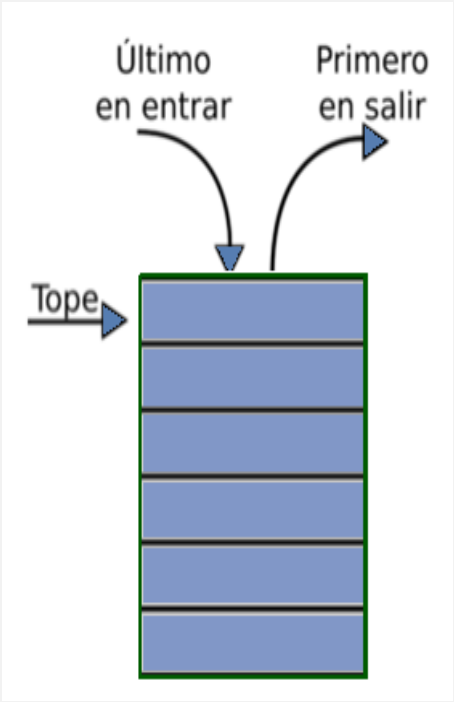
Especificaciones de la estructura de una Pila

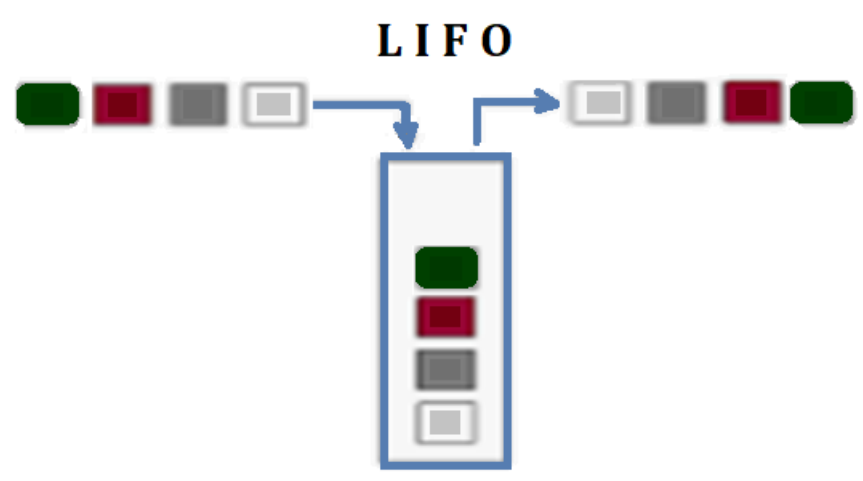
Las operaciones que definen una pila y manipulan su contenido son:

Crear Pila	Inicia la pila como vacía
Insertar (push)	Pone un dato en la pila
Quitar (pop)	Retira un dato de la pila
PilaVacía	Comprueba si la pila está vacía
PilaLlena	Comprueba si la pila está llena
LimpiarPila	Quita todos los elementos de la pila
Cima	Obtiene el elemento en la cima de la pila
Tamaño Pila	Número de elementos en la pila

Implementación de pila en software

Una **pila** en software es una estructura de datos, como arreglos (arrays) o listas enlazadas, que sigue el principio **LIFO (Last In, First Out)**, lo que significa que el último elemento que se agrega a la pila es el primero en ser retirado.



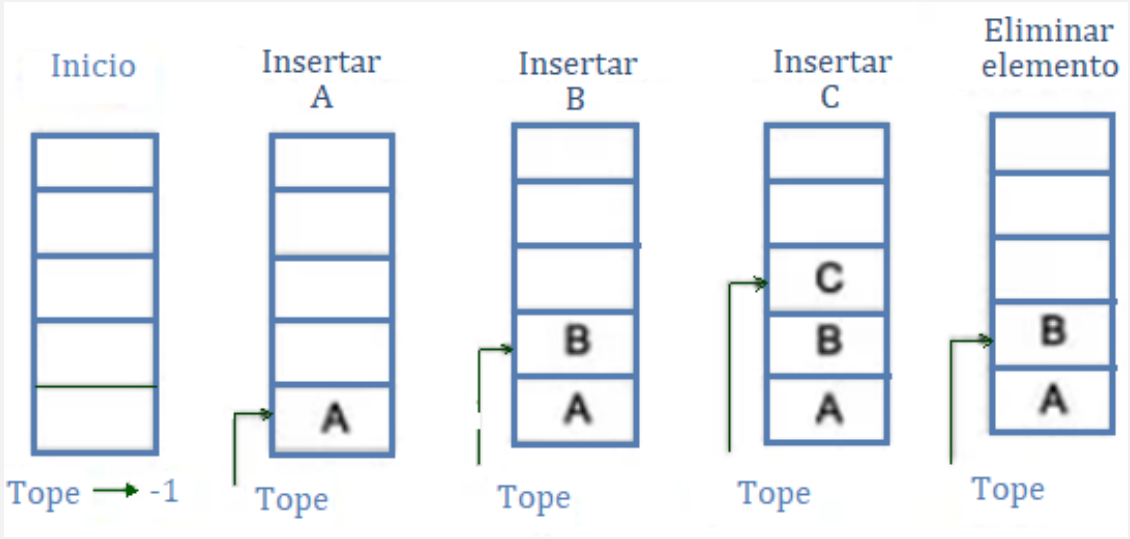


Último en entrar-Primero en salir

Implementación de pila en software

Implementación de Pila usando Arreglos (Arrays)

La implementación más sencilla de una *pila en software* es usando un **arreglo estático** (si conocemos el tamaño máximo de la pila de antemano). En este caso, un puntero o índice, llamado **tope**, se usa para rastrear la posición del último elemento insertado en la pila.



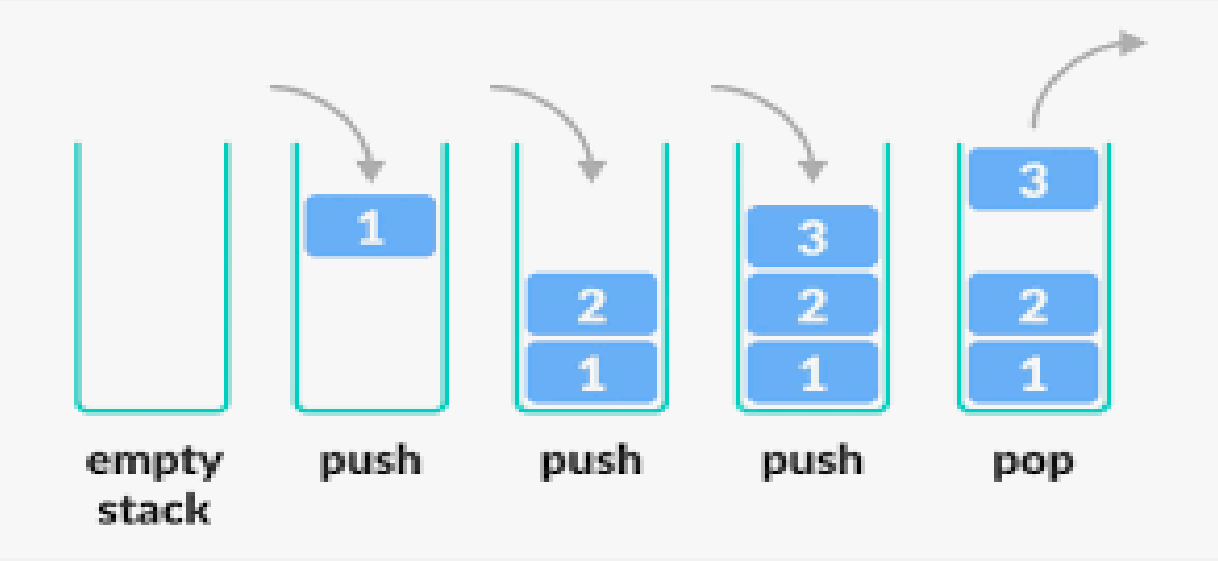
Implementación de pila en software

Operaciones

Push: Agrega un nuevo elemento en la parte superior de la pila (posición top + 1, e incrementa el índice top)

Pop: Devuelve el valor en la posición top y elimina el actual dato superior de la pila. (después decrementa el índice top)

Peek: Devuelve el valor en la posición top sin eliminarlo (solo ve el último elemento de la pila)



Implementación de pila en software

Ejemplos de uso de pila en software

Llamadas a funciones

Cuando un programa realiza una llamada a una función, se guarda el **dirección de retorno** (dónde debe continuar la ejecución después de que la función termine) en la pila. Las variables locales de las funciones también se almacenan en la pila.

Recursión

Cuando una función se llama a sí misma, cada llamada recursiva agrega un nuevo **marco de pila** con su propio conjunto de variables locales y dirección de retorno.

Cuando una llamada recursiva finaliza, su marco de pila es retirado (pop) y el control vuelve a la llamada anterior.

Deshacer/rehacer en aplicaciones

En aplicaciones como procesadores de texto, las pilas se utilizan para gestionar acciones de deshacer y rehacer. Cada acción se coloca en una pila, y cuando el usuario solicita deshacer, la última acción se elimina de la pila y se invierte.

Implementación de pila en software

Implementación de Pila usando Listas Enlazadas

Una lista enlazada permite que la pila crezca dinámicamente sin requerir definir un tamaño fijo de antemano. Aquí, cada **nodo** de la lista contiene un valor y un puntero al siguiente nodo.

Operaciones

Push: Crea un nuevo nodo, lo enlaza con el nodo anterior, y actualiza el puntero top.

Pop: Elimina el nodo superior (el nodo al que apunta top), y mueve el puntero top al siguiente nodo.

Peek: Devuelve el valor del nodo apuntado por top.

Implementación de pila en hardware

La implementación de una **pila** en hardware, refiere a un circuito de memoria que sigue el principio **LIFO (Last In, First Out).** Las pilas en Hardware se utilizan en microprocesadores, microcontroladores y sistemas embebidos.

Pila en microprocesadores (pila de ejecución)

Los procesadores utilizan una **pila de ejecución** para almacenar las direcciones de retorno y los valores de los registros cuando se realizan llamadas a subrutinas o interrupciones. El **puntero de pila (stack pointer)** indica la dirección de la última celda de memoria disponible en la pila.

Implementación de pila en hardware

Operaciones de hardware en la pila:

Push: Cuando una operación de "push" se ejecutada por el procesador, el valor que debe almacenar se escribe en la memoria en la dirección indicada por el **stack pointer (SP)**, posteriormente el **stack pointer** se ajusta para apuntar a la siguiente ubicación libre de la pila.

Pop: Cuando se realiza una operación de "pop", el **stack pointer** se ajusta para apuntar al último valor almacenado, y este valor es recuperado de la memoria.

Implementación de pila en hardware

La pila de ejecución es una estructura de datos gestionada en la memoria del sistema, implementada en la memoria (SRAM) y se utiliza para el control de llamadas a funciones, los retornos y gestión de datos locales y control de un programa.

El Stack Pointer mantiene la dirección de la última posición usada en la pila.

EJEMPLO:

```
funcion_a:
   // instrucciones
   CALL funcion_b
   // instrucciones

funcion_b:
   //  instrucciones
   CALL funcion_c
   // instrucciones

funcion_c:
   //Instrucciones de funcion_c
   RET
```

Implementación de pila en software

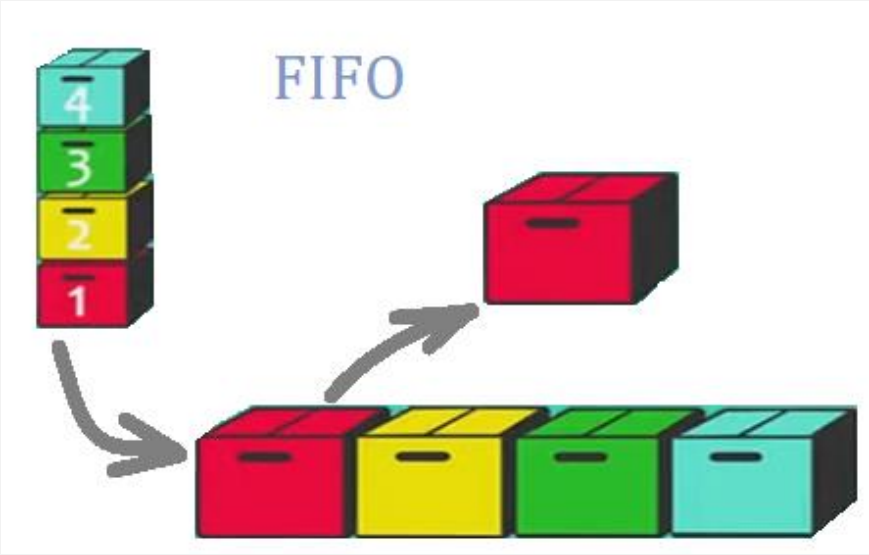
Una **pila FIFO** (First In, First Out) es un concepto que en realidad corresponde más a una ***cola*** que a una pila.
Es decir, si bien "*pila*" se refiere a una estructura de datos LIFO (Last In, First Out).

FIFO es una estructura de datos donde el primer elemento en entrar es el primero en salir, lo cual describe el comportamiento de una ***cola***.

Por ejemplo: que se tiene una fila de personas esperando para usar un cajero. La persona que llega primero es la primera en atendida.

1. Las personas se colocan en la fila (uno detrás del otra).
2. La primera persona en llegar (al frente de la fila) es la primera en realizar su trámite y salir de la fila.

Implementación de pila en software



First In- First Out

- **Pila (LIFO)**: El último elemento en entrar es el primero en salir.
- **Cola (FIFO)**: El primer elemento en entrar es el primero en salir.

Registros

Pequeñas unidades de almacenamiento dentro del procesador (CPU) que permiten guardar datos temporalmente para su procesamiento inmediato y eficiente. Su acceso es mucho más rápidos que a la memoria RAM y pueden almacenarse datos, direcciones de memoria o resultados intermedios durante la ejecución de un programa

Existen varios tipos de registros, entre ellos:

Registros de datos: Guardan valores numéricos o datos que se están procesando.

Registros de direcciones: Almacenan direcciones de memoria, es decir, dónde se encuentran los datos o instrucciones en la memoria principal.

Registros

Registros de control: Contienen información de control o del estado del procesador para supervisar y coordinar el flujo de ejecución de instrucciones.

Registros de propósito general: Son usados por e procesador para diversas operaciones aritméticas o lógicas según las necesidades del programa.

La cantidad y el tamaño de los registros varían según la arquitectura del procesador, éstos permiten un acceso rápido a la información necesaria para ejecutar instrucciones.

Memorias Caché

En las arquitecturas de computadoras modernas, los procesadores suelen contar con varios niveles de caché: **Caché L1**, **Caché L2** y en ocasiones hasta **Caché L3**.

Caché L1 es la más cerca al núcleo y por ende la más rápida pero es pequeña en capacidad.
Caché L2 es más lenta y más grande que L1, almacena datos a los que el procesador accede con frecuencia, evita que el procesador tenga que esperar a obtener datos de la memoria principal, que es mucho más lenta.

La cache L2 actúa como un intermediario entre la RAM (principal) y la cache L1 haciendo óptimo el rendimiento general del sistema al reducir los tiempos de espera.

Chips Memorias RAM

Para un determinado tamaño de memoria, existen diferentes formas de organizar el chip.

Memoria (palabras/bits)	Filas de flip-flops	Columnas de flip-flops	Líneas de Entrada de datos	Líneas de Salida de datos	Línea de direcciones
4x3	4	3	3	3	A0, A1
4x8	4	8	8	8	A0, A1
8x3	**8**	**3**	**8**	**8**	**A0,A1,A2**

La tecnología de circuitos permite fabricar circuitos integrados cuya estructura interna es un patrón bidimensional repetitivo.
Al mejorar la tecnología, el número de bits que se pueden colocar en un chip aumenta continuamente duplicándose por lo regular cada 18 meses (Ley de Moore)