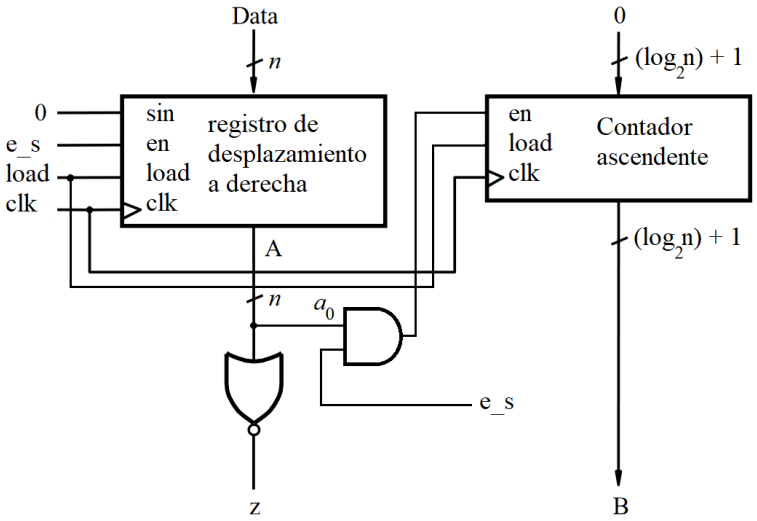
# Introducción a Arquitectura de procesadores

Juan Alarcón. jalarcon@frba.utn.edu.ar

# Camino de datos

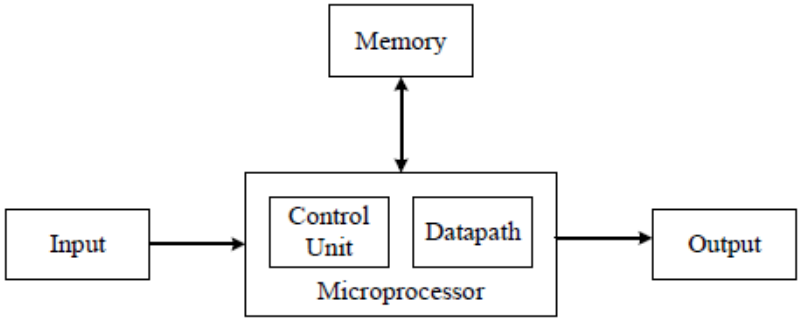



- La técnica de camino de datos (datapath) se basa en la utilización de registros, contadores, multiplexores y demás como bloques para construir el "procesamiento" de los datos a tratar.
- Los caminos de datos se "ayudan" de máquinas de estados finitos para controlar el que momento se habilitan, deshabilitan, etc, los bloques del mismo. También generan señales de estados para informar a la máquina de estados.
- En el diagrama de camino de datos de la derecha se ve un ejemplo que "cuenta la cantidad de unos de una palabra dada"

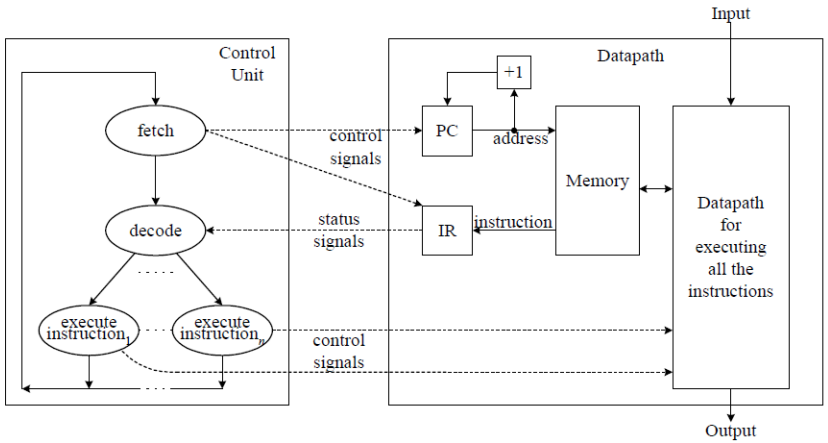
# Microprocesador

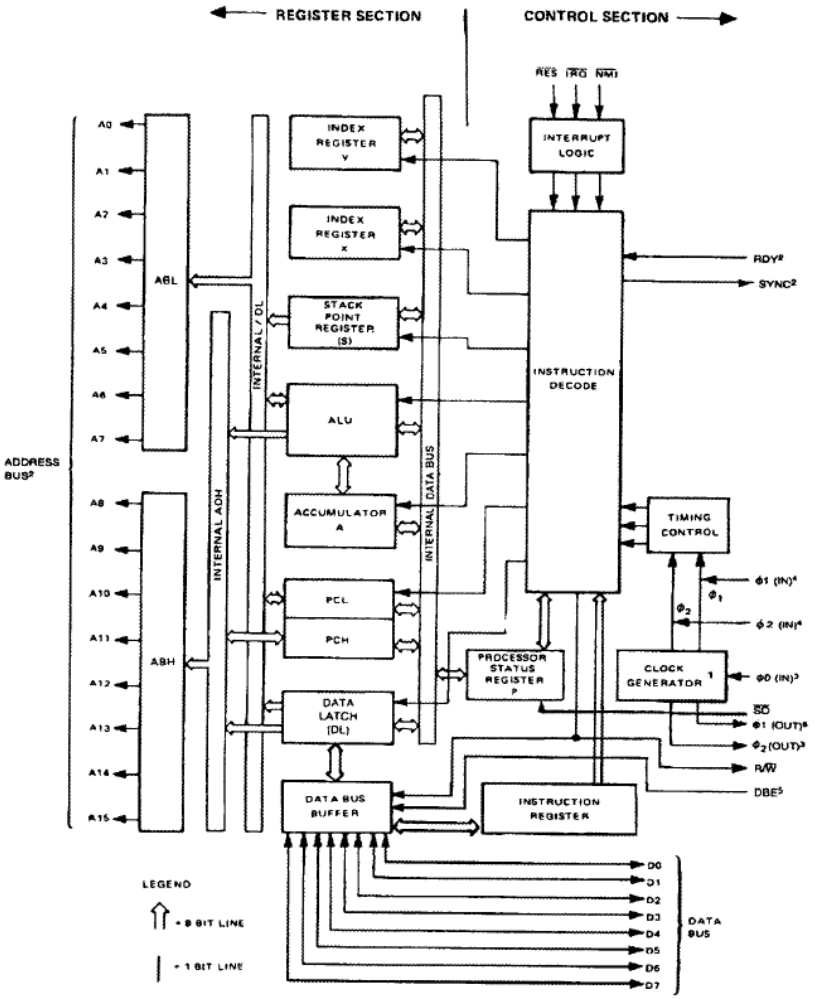
- La técnica para resolver sistemas en diseño digital sincrónico es a través de las máquinas de estados finitos.
- Cuando aumenta la complejidad del problema a las FSM se les agrega el Datapath (capacidad de tratamiento de datos).
- Aún así, en una FSM+D la "personalidad del sistema" está definida por la configuración del hardware.
- Un ***microprocesador*** no es más que una FSM+D con un conjunto de operaciones previamente definidas (***instrucciones***).

- La secuencia de instrucciones entre los ***registros*** del microprocesador están definidos ***externamente al hardware en una memoria de programa.***
- La tarea a realizar por un microprocesador de propósito general entonces queda definida de manera ***independiente al hardware (software)***

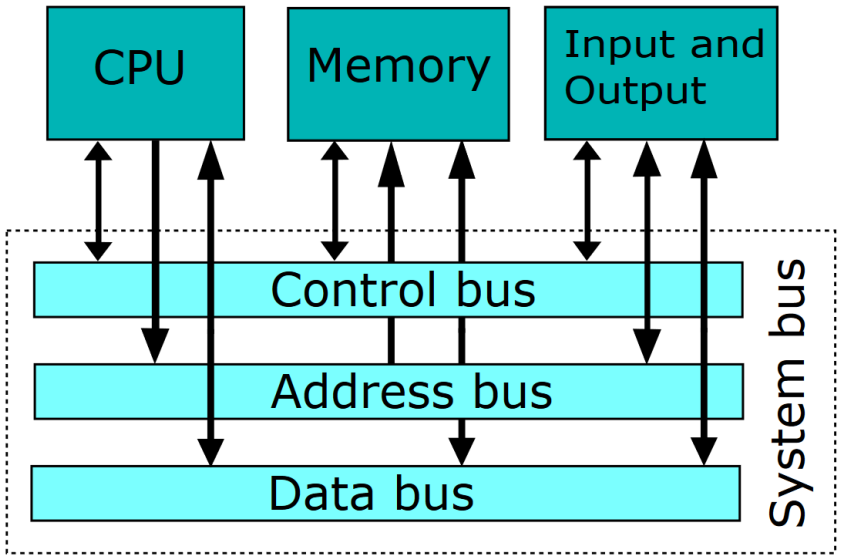


# Partes de un microprocesador



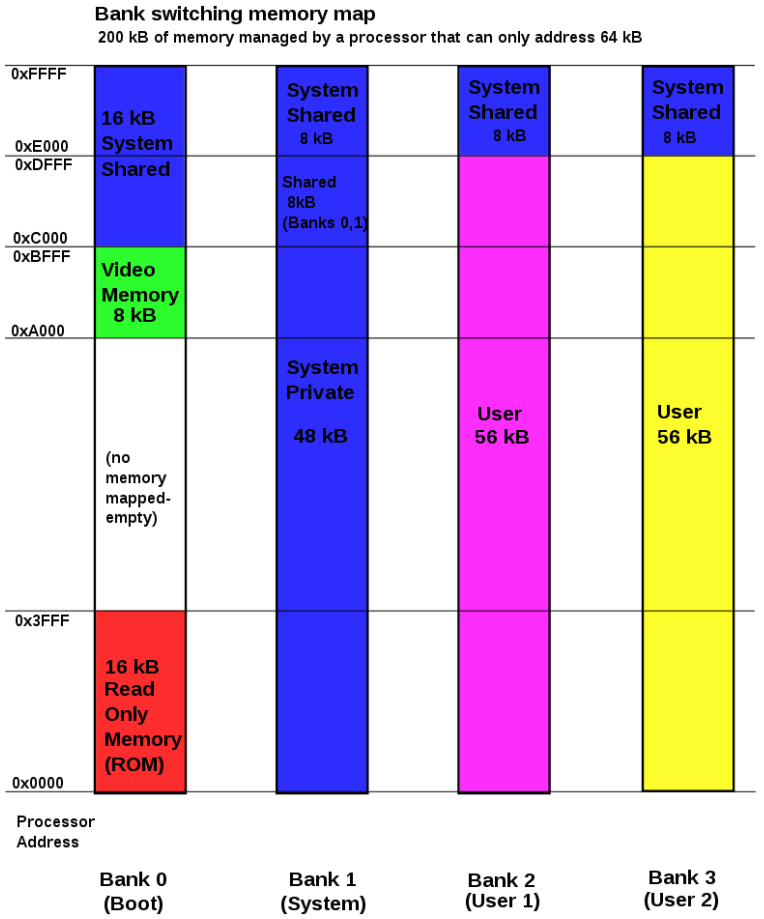
# Partes de un microprocesador




- Toda la interfaz de un microprocesador es a través de los ***buses del sistema***.
- Para que el sistema con microprocesador funciones hay que conectar memorias y periféricos (I/O, timers, comunicaciones, etc.)

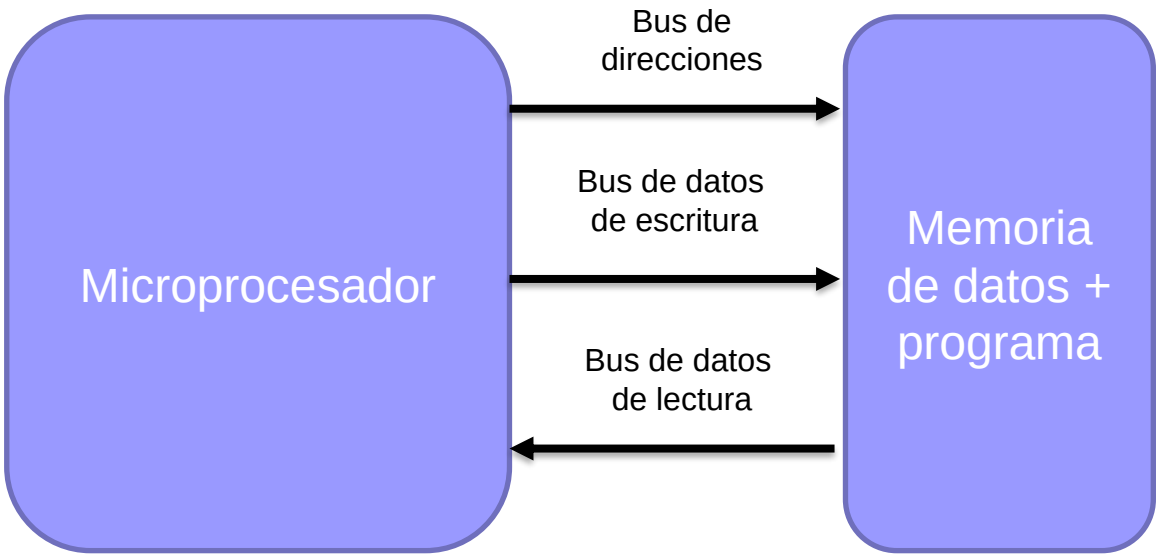
# Mapa de memoria.

- Los microprocesadores de propósito general sólo se interconectan con otros dispositivos a través de sus ***mapas de memoria***.
- Para conectarse con otros dispositivos el microcontrolador genera uno o varios ***buses de datos***.
- Un ***bus de datos*** es un esquema donde un dispositivo (el microprocesador en este caso) direcciona a través de ***M líneas hasta $2^M$ posiciones de N bits de memoria (dispositivos) distintos a los que podrá leer o escribir de a uno por vez.***
- ***Las $2^M$ posiciones de ese bus se lo conoce como mapa de memoria***



# Arquitectura Harvard

# Arquitectura Von Neumann



# Códigos de operación e instrucciones

- Se entiende por instrucción al conjunto de datos mínimo que el microprocesador lee, interpreta y ejecuta.

- Las instrucciones tienen al menos dos campos:
  - El código de operación. Es la parte que indica que operación se va a realizar (suma).
  - Datos de la operación. Parámetro(s) de la operación e incluso puede no existir.

- Otra forma de clasificar las instrucciones es a través de su operación:
  - Transferencia de datos.
  - Aritméticas
  - Lógicas
  - Transferencia de control.

# Codigos de operación. Ejemplo

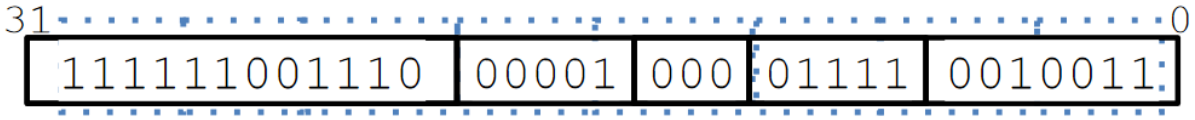
**Encoding A1** ARMv4*, ARMv5T*, ARMv6*, ARMv7

ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}

| 31 30 29 28 | 27 26 | 25 | 24 23 22 21 | 20 | 19 18 17 16 | 15 14 13 12 | 11 10 9 8 7 | 6 5 | 4 | 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 0 | 0 | 0 1 0 0 | S | Rn | Rd | imm5 | type | 0 | Rm |

- RISCV Instruction: `addi x15,x1,-50`

Field representation (binary):



# Procesadores, set de instrucciones.

- ***CISC.*** Un procesador con este tipo de diseño tiene instrucciones muy específicas, de ancho variable y que suelen ejecutarse en varios ciclos de reloj. Esto hace que el procesador se convierta en un hardware complejo (ejemplo: x86).
- ***RISC.*** Se diseñan pocas instrucciones, de ancho fijo y que se ejecuten en un único ciclo de reloj. Se busca simplificar el procesador. Se pone la complejidad en el software (ejemplos: AVR, ARM, MIPS)

# Actividades

- Para las actividades se usó el software Xilinx ISE 14.7 (versión WebPack)
- https://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html
- Para instalar el ISE 14.7 en Windows 10. https://www.micro-nova.com/xilinx-ise-win10

# Actividades. Procesador UP1

```
entity up1 is
    Port (
            -- Señales de reloj y Reset
            clk : in    STD_LOGIC;
            rst : in    STD_LOGIC;
            --Buses de memoria de programa
            pdb : in    STD_LOGIC_VECTOR (11 downto 0);
            pab : out   STD_LOGIC_VECTOR (7  downto 0);
            --Buses de memoria de datos.
            ddib: in    STD_LOGIC_VECTOR (7  downto 0);
            ddob: out   STD_LOGIC_VECTOR (7  downto 0);
            dab : out   STD_LOGIC_VECTOR (7  downto 0);
            wr  : out   STD_LOGIC);
end up1;
```

- Junto con esta presentación se provee el código VHDL de un sistema que utiliza este procesador (up1.vhd). Responder:
  - ¿Qué arquitectura tiene el procesador?
  - ¿Qué cantidad de bits tiene cada instrucción?¿El código de operación?
  - ¿Cuántos registros tiene?
  - ¿Cuánto tarda cada instrucción en ejecutarse (en ciclos de reloj)?

# Procesador UP1. Respuestas

- ¿Qué arquitectura tiene el procesador?
  - Es un procesador con arquitectura Harvard, ya que tiene dos buses de direcciones (pab y dab)
- ¿Qué cantidad de bits tiene cada instrucción?¿El código de operación?
  - Las instrucciones son de 12 bits. Y los códigos de operación de 4 bits, (los bits más significativos de pdb, ver up1.vhd líneas 33 a 50)
- ¿Cuántos registros tiene?
  - Tiene dos registros A (resultados de la ALU) y PC (contador de programa). Ver up1.vhd
- ¿Cuánto tarda cada instrucción en ejecutarse (en ciclos de reloj)?
  - Todas las instrucciones tardan 1 ciclo de reloj en ejecutarse, ya que se decodifica por tabla y no por máquina de estados. Debido a la naturaleza en extremo simple de este procesador.

# Sistema con el uP-1

- Para poder implementar una aplicación que pone 8 pines en 0xAA y en función de que si un pin de entrada está en uno invierte el estado de los pines de salida cada 3ms es necesario que al microprocesador uP-1 se le incorporen periféricos.
- Al ser un procesador Harvard se colocan todos los periféricos en la memoria de datos. Esto es lo que se conoce como periféricos mapeados como memoria.
- Sistema con uP-1, periféricos:
  - Puerto de salida (0x30)
  - Puerto de entrada (0x20)
  - Temporizador de 8 bits (0x10)
  - Un byte de memoria RAM (0x00)

# Memoria y periféricos

- Un microprocesador de propósito general sólo tiene como interfaz los buses de memoria de los que dispone.
- Muy a menudo se necesita que el sistema utilice pines de entrada – salida, temporizadores, etc. Por lo que se diseña hardware que pueda, por un lado, tenga interfaz compatible con el bus y por el otro con la tarea a realizar.
- Para conectar todos estos periféricos y que compartan el bus de un microprocesador, es necesario generar señales de ***decodificación.***
- La decodificación no es otra cosa que compartir los buses de datos entre la memoria y los diferentes dispositivos para que el procesador pueda leer o escribir ***cada uno de ellos en diferentes operaciones.***
- Básicamente las señales de decodificación generan a partir de las líneas de direcciones de datos señales de habilitación para escribir o leer estos periféricos.

# Puertas de I/O

## Puerto de Salida

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

entity PuertaSalida is
    Port ( clk      : in  STD_LOGIC;
           rst      : in  STD_LOGIC;
           en       : in  STD_LOGIC;
           datos    : in  STD_LOGIC_VECTOR (7 downto 0);
           pines    : out STD_LOGIC_VECTOR (7 downto 0));
end PuertaSalida;

architecture Behavioral of PuertaSalida is
begin
    process(clk,rst)
    begin
        if(rst = '1') then
            pines <= (others => '0');
        elsif(rising_edge(clk)) then
            if(en = '1') then
                pines <= datos;
            end if;
        end if;
    end process;
end Behavioral;
```

## Puerto de Entrada

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

entity PuertoEntrada is
    Port (  clk             : in    STD_LOGIC;
            pines           : in    STD_LOGIC_VECTOR (7 downto 0);
            salida_datos    : out   STD_LOGIC_VECTOR (7 downto 0));
end PuertoEntrada;

architecture Behavioral of PuertoEntrada is
    signal q1               : std_logic_vector(7 downto 0);
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
                q1              <= pines;
                salida_datos    <= q1;
        end if;
    end process;
end Behavioral;
```

# Decodificación en sistema UP1

```
constant    ADDR_OP :   std_logic_vector(7 downto 0) := x"30";
constant    ADDR_IP :   std_logic_vector(7 downto 0) := x"20";
constant ADDR_TIM   :   std_logic_vector(7 downto 0) := x"10";
constant ADDR_MEM   :   std_logic_vector(7 downto 0) := x"00";
```

```
cs_po   <= wr when dab = ADDR_OP   else '0';
cs_tim  <= wr when dab = ADDR_TIM  else '0';
cs_mem  <= wr when dab = ADDR_MEM  else '0';

with dab select
    ddib    <=  dbi_pi              when ADDR_IP,
                dbi_tim             when ADDR_TIM,
                dbi_mem             when ADDR_MEM,
                (others => '0')     when others;
```

- Explique por qué no hay cs_pi ni participa del multiplexor la dirección de la puerta de salida.
- ¿La decodificación responde a alguna lógica o es arbitraria?
- ¿Tiene algún impacto en la lógica del programa esta decodificación?

# Decodificación en sistema UP1(2)

```
with pab select
    pdb <=      "110000000000" when "00000000",     --NOP
                "110000000000" when "00000001",     --NOP
                "000010101010" when "00000010",     --MOV A,0xAA
                "100100110000" when "00000011",     --MOV [0x30],A
                "100100000000" when "00000100",     --MOV [0x00],A
                "000000000011" when "00000101",     --MOV A,0x03
                "100100010000" when "00000110",     --MOV [0x10],A
                "100000010000" when "00000111",     --MOV A,[0x10]
                "101100000111" when "00001000",     --JZ $-1        ;Salto inst. Anterior
                "100000100000" when "00001001",     --MOV A,[0x20]
                "000100000001" when "00001010",     --AND A,1       ;A&=1
                "101100000101" when "00001011",     --JZ 0x05
                "100000000000" when "00001100",     --MOV A,[0x00]
                "001111111111" when "00001101",     --XOR A, 0xFF   ;A^=0xFF
                "100100110000" when "00001110",     --MOV [0x30],A
                "100100000000" when "00001111",     --MOV [0x00],A
                "101000000101" when "00010000",     --JUMP 0x5
                "111100000000" when others;         --HALT
```

# Actividades. Sistema con procesador UP1

- Dado el archivo sysup1.vhd y su archivo de simulación sysup1_tb.vhd. Responder:
  - ¿Qué determina la dirección de memoria de cada dispositivo (puertos, temporizador, memoria)?
  - ¿De qué depende la demora de 3ms para invertir las salidas?
  - ¿Por qué es necesario el byte de memoria RAM en la posición 0x00?

# Respuestas. Sistema con procesador UP1

- ¿Qué determina la dirección de memoria de cada dispositivo (puertos, temporizador, memoria)?
  - De las señales de habilitación cs_po, cs_tim y cs_mem que habilitan la escritura en los dispositivos y del multiplexor que elige que dispositivo puede leer el procesador (ver líneas 76 a 84 de sysup1.vhd y las constantes en las líneas 14 a 17 del mismo archivo).

- ¿De qué depende la demora de 3ms para invertir las salidas?
  - Del temporizador instanciado como ti en sysup1.vhd (decrementa el valor de su contador interno una vez por ms y cuando llega a cero se detiene y setea un FF. Esto lo hace le código de programa (líneas 93 a 95 sysup1.vhd):
    - A=0x03 ;Carga A con 3
    - [0x10]=A ;Carga el timer con A
    - A = [0x10] ; Lee el timer
    - JZ $-1 ; Si es cero vuelve a la línea anterior ¿Por qué?

- ¿Por qué es necesario el byte de memoria RAM en la posición 0x00?
  - Para invertir el led se necesita conocer en que estado se encontraba, como el puerto se salida (PuertaSalida.vhd) es de “sólo escritura” se genera un byte de memoria lectura escritura que se pueda leer para poder invertir el puerto

# Microprocesador nano_16

```
entity nano16 is
    Port ( clk    : in     STD_LOGIC;
           rst    : in     STD_LOGIC;
           din    : in     STD_LOGIC_VECTOR (15 downto 0);
           dout   : out    STD_LOGIC_VECTOR (15 downto 0);
           addr   : out    STD_LOGIC_VECTOR (15 downto 0);
           wr     : out    STD_LOGIC);
end nano16;
```

- Dados los archivos nano16.vhd, nano16_fsm.vhd y nano16_tb.vhd determine:
  - ¿Qué tipo de arquitectura tiene este procesador?
  - ¿Cuál es su capacidad de direccionamiento?
  - ¿Cuánto tardan en ejecutarse las instrucciones (en ciclos de reloj)? ¿Todas tardan lo mismo?

# Microprocesador nano_16. Respuestas

- ¿Qué tipo de arquitectura tiene este procesador?
  - Es Von Neumann tiene un solo bus de direcciones.
- ¿Cuál es la capacidad de direccionamiento de memoria?
  - Es de 16 bits por lo que direcciona 65536 palabras de 16 bits
- Cuánto tardan en ejecutarse las instrucciones (en ciclos de reloj)? ¿Todas tardan lo mismo?
  - Todas las instrucciones tardan lo mismo (ver nano_fsm.vhd) y todas tardan 5 ciclos de reloj en ejecutarse.

# nano_16. Arquitectura

- Este sistema tiene los siguiente registros:
  - ***PC***. Contador de programa de 16 bits. Mantiene la dirección de la instrucción en ejecución.
  - ***R0 – R15***. Registros de propósito general de 16bits. R0 siempre se lee cero no importa lo que se escriba en el.
  - ¿Hay algún registro no visible desde el set de instrucciones? ¿Cuál? ¿Para que cree que se usa?

# Nano_16. Set de instrucciones

| | |
|---|---|
| ***STR Rd,[Rs]*** | `//*Rs = Rd` |
| ***LRE Rd,[cte 8 bits]*** | `// Rd = *[PC + cte 8 bits en Complemento a dos]` |
| ***LDC Rd,[cte 8 bits]*** | `// Rd = Rd[15..8]Cte` |
| ***LDR Rd, [Rs]*** | `// Rd =*Rs` |
| ***JREL [cte 8 bits],Rx*** | `// PC += cte (con signo) sí Rx = 0` |
| ***JMP Rd, Rr, Rx*** | `// PC = Rd, Rr = PC+1 sí Rx = 0` |
| ***AND Rd, Rs1, Rs2*** | `// Rd = Rs1 & Rs2` |
| ***OR Rd, Rs1, Rs2*** | `// Rd = Rs1 \| Rs2` |
| ***XOR Rd, Rs1, Rs2*** | `// Rd = Rs1 ^ Rs2` |
| ***ADD Rd, Rs1, Rs2*** | `// Rd = Rs1 + Rs2` |
| ***NOT Rd, Rs*** | `// Rd = ~Rs` |
| ***INC Rd, Rs*** | `// Rd = Rs + 1` |
| ***DEC Rd, Rs*** | `// Rd = Rs - 1` |
| ***NEQ Rd, Rs1, Rs2*** | `// Rd = (Rs1!=Rs2)?0xFFFF:0` |
| ***GE Rd, Rs1, Rs2*** | `// Rd = (Rs1>=Rs2)?0xFFFF:0 //Sin signo` |

# Nano_16. Set de instrucciones

- Dado el set de instrucciones de la página anterior responda:
  - ¿Es posible implementar un salto incondicional?¿Cómo? Ayuda. Recuerde el comportamiento de R0.
  - ¿Cómo  implementa un llamado a función?¿Y el retorno de la misma?
  - ¿Qué estrategia tiene que tomar si tiene un llamado a función anidado?
  - En este set de instrucciones no existe la instrucción MOV Rd, Rs ¿Es necesaria?¿Es posible emularla?¿Cómo?

# Nano_16. Set de instrucciones. Respuestas

- ¿Es posible implementar un salto incondicional?¿Cómo? Ayuda. Recuerde el comportamiento de R0.
  - Jmp Rd, R0, R0
- ¿Cómo  implementa un llamado a función?¿Y el retorno de la misma?
  - Jmp Rd, R1, R0 guarda la dirección de retorno en R1
  - Jmp R1,R0,R0 vuelve a donde apunta R1
- ¿Qué estrategia tiene que tomar si tiene un llamado a función anidado?
  - Guardar el contenido de la dirección de retorno en otro registro o en una posición de memoria con STR
- En este set de instrucciones no existe la instrucción MOV Rd, Rs ¿Es necesaria?¿Es posible emularla?¿Cómo?. Sí, absolutamente necesaria.
  - ADD Rd, Rs, R0 (R0 siempre vale cero)

# Sistema con nano16

- Dados los archivos:
  - Temporizador.vhd
  - PuertaSalida.vhd
  - Todos los archivos del nano16.
  - sys_nano16.vhd

- Del archivo sys_nano16.vhd Determinar:
  - En que posición del mapa de memoria debe colocar la memoria de programa ¿Por qué?
  - ¿Dónde se ubica los periféricos? ¿La memoria RAM?
  - Explique el funcionamiento del programa.
  - Explique como se cargan las constantes en las dos primeras líneas de programa.