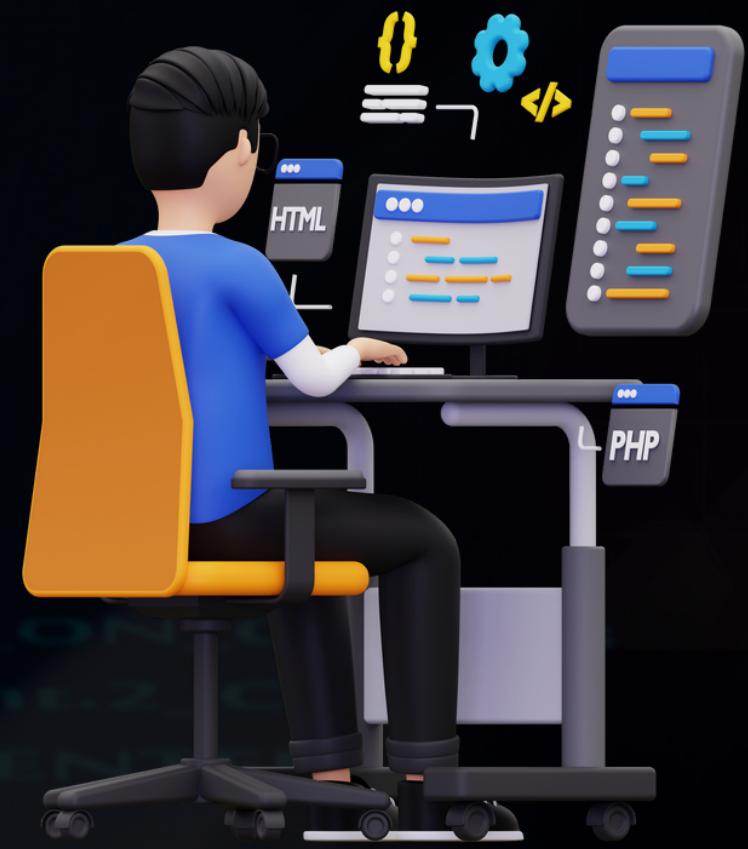


# ASSEMBLER EN ARDUINO



# ACERCA DE MÍ

- 9no semestre de Ingeniería en Ciencias y Sistemas (Cierre)
- Auxiliar de Organización Computacional
- Bachiller Industrial y Perito en Electrónica



javgut2001@gmail.com



4254-3549



PikaGuty



**Javier Alejandro  
Gutierrez de León**



POR QUE  
SISTEMAS  
VE ESTO?

# LENGUAJE DE PROGRAMACIÓN

Su función principal es permitirnos la comunicación usuario-máquina.



## >\_ Código fuente



## >\_ Analizador Léxico



Expresiones regulares  
([a-zA-Z])[a-zA-Z0-9\_]\*

Palabras reservadas  
if print for

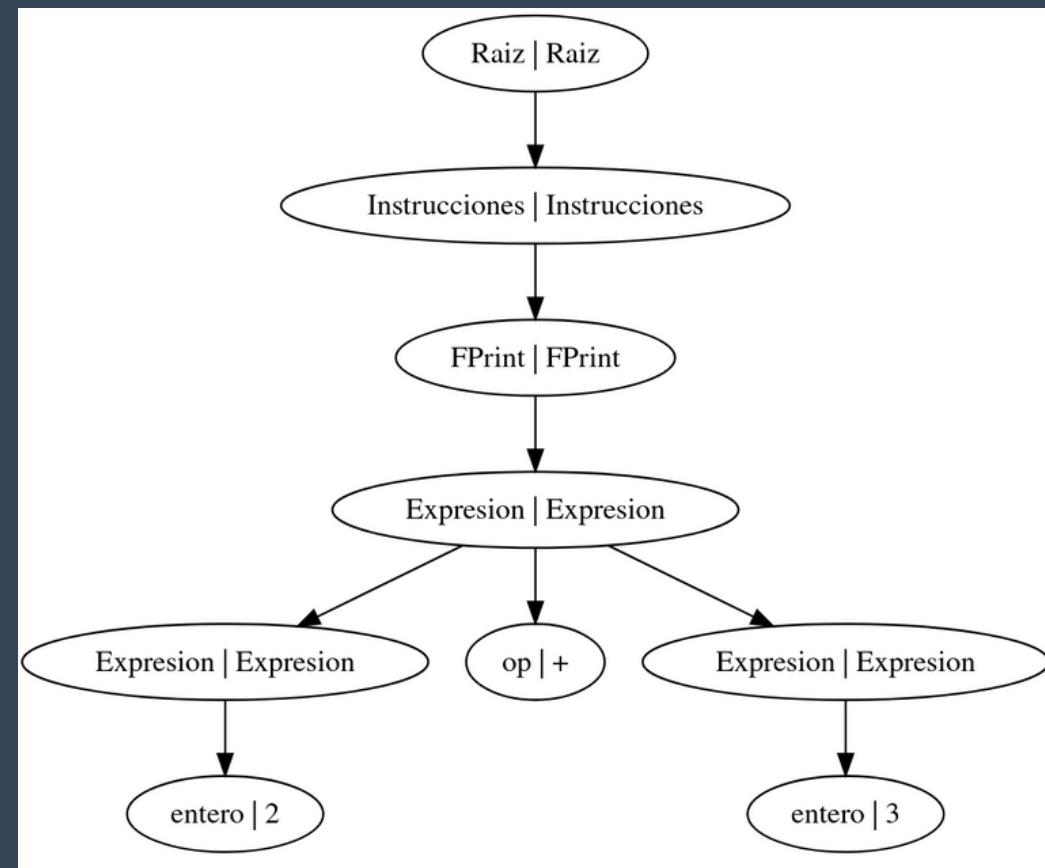
Símbolos permitidos  
+ - \* % /



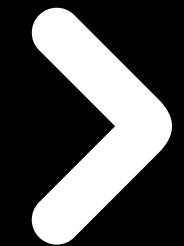
## >\_ Analizador semántico



("Hola mundo")print;  
print("Hola mundo");

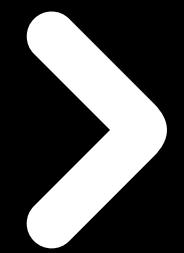
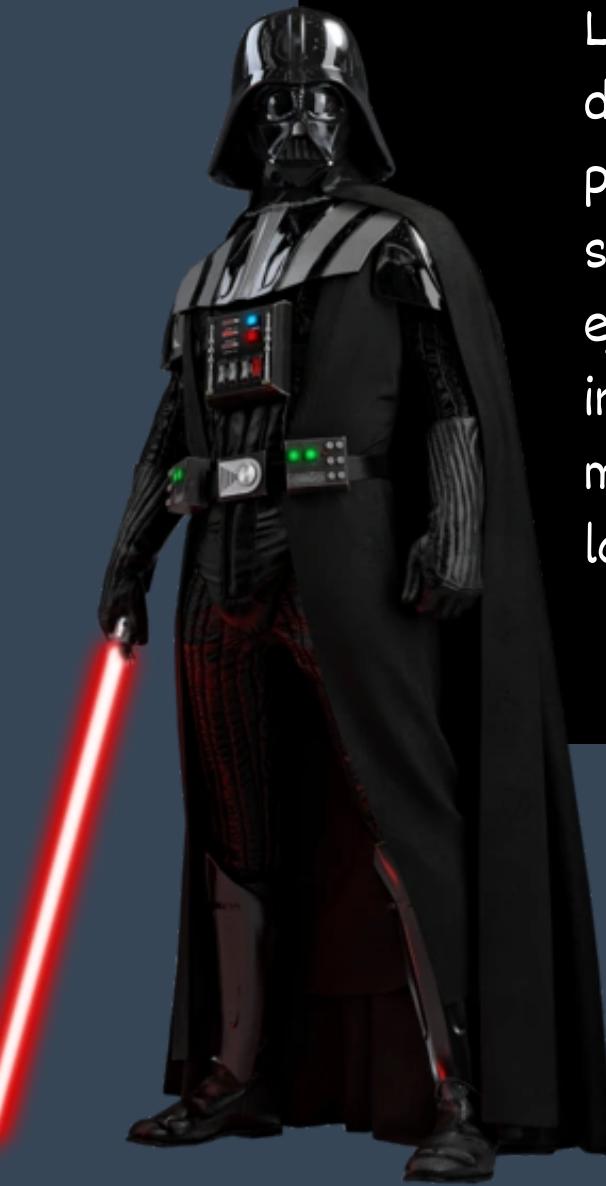


## >\_ Analizador sintáctico



## Lenguajes Compilados

Los lenguajes compilados son convertidos directamente a código máquina que el procesador puede ejecutar. Como resultado, suelen ser más rápidos y más eficientes al ejecutarse en comparación con los lenguajes interpretados. También le dan al desarrollador más control sobre aspectos del hardware, como la gestión de memoria y el uso del CPU.



## Lenguajes Interpretados

Estos lenguajes ejecutan línea por línea el programa y a la vez ejecutan cada comando. Los lenguajes interpretados alguna vez fueron significativamente más lentos que los lenguajes compilados. Pero, con el desarrollo de la compilación justo a tiempo, esa diferencia se está reduciendo.





Interpretarlo  
/ ejecutarlo

>  
Generador de  
código intermedio



```
File Edit Confirm Menu Utilities Compilers Test Help
EDIT      P53.ASM.LABS(ASSIST6) - 01.08          Columns 00001 00072
Command ==>                                         Scroll ==> CSR
***** **** Top of Data ****
000001 //P53AST  JOB (SYS),ASSIST ASM #6',CLASS=A,MSGCLASS=H,
000002 //           MSGLEVEL=(1,1),NOTIFY=P53,REGION=3M
000003 /*JOBPARM ROOM=MB
000004 //ASSEMBLE EXEC ASSIST
000005 //SYSPRINT DD SYSOUT=*
000006 //SYSUDUMP DD SYSOUT=*
000007 //SYSIN  DD *
000008 $JOB  ASSIST TEST6,XREF=2
000009 TEST6  CSECT
000010     USING TEST6,15
000011     XREAD CARD,80
000012 LOOP    BC B'0100',EXIT
000013     XDECI 2,CARD
000014     BC B'0001',GETNXT
000015     XDECI 3,B(1)
000016     BC B'0001',GETNXT
000017     SR 2,3
000018     ST 2,WORD
000019     XDUMP WORD,4
                                         READ CARD INTO BUFFER
                                         AT END OF FILE GO TO EXIT
                                         GET FIRST NUMBER FROM CARD
                                         SKIP CARD IF BAD VALUE
                                         GET SECOND NUMBER FROM CARD
                                         SKIP IF BAD VALUE
                                         GET DIFFERENCE INTO R2
                                         STORE DIFFERENCE INTO WORD
                                         SNAP IT
```

>\_  
Optimización



>\_  
Generar  
Código

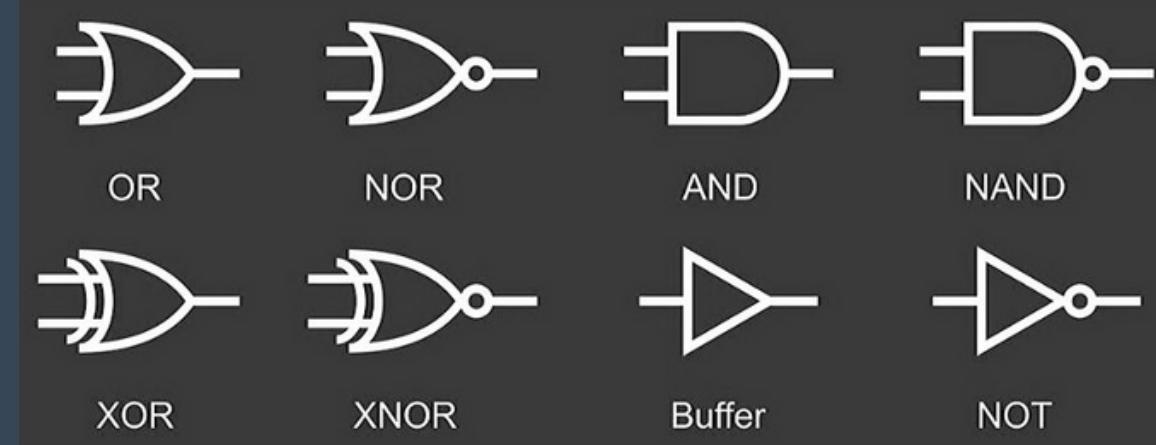
```
section    .text
global     _start
_start:
        mov    edx,len
        mov    ecx,msg
        mov    ebx,1
        mov    eax,4
        int    0x80

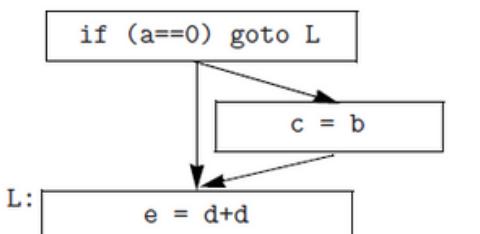
        mov    eax,1
        int    0x80

section    .data
msg     db 'Hello, world!',0xa
len     equ $ - msg
```

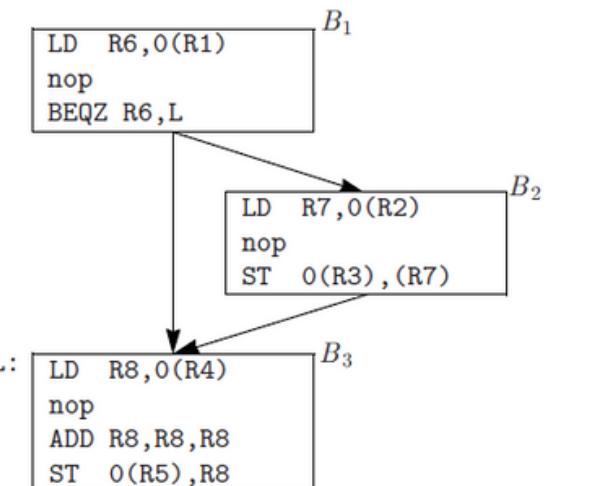
;must be declared for linker (ld)  
;tell linker entry point  
;message length  
;message to write  
;file descriptor (stdout)  
;system call number (sys\_write)  
;call kernel  
  
;system call number (sys\_exit)  
;call kernel  
  
;our dear string  
;length of our dear string

>\_  
Lenguaje  
máquina

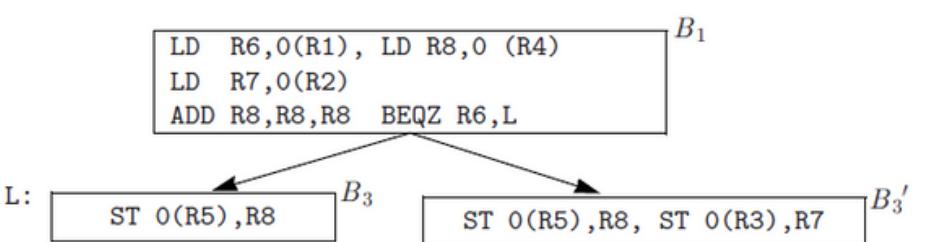




(a) Programa fuente



(b) Código máquina programado en forma local



(c) Código máquina programado en forma global

Figura 10.12: Grafos de flujo antes y después de la programación global en el ejemplo 10.9



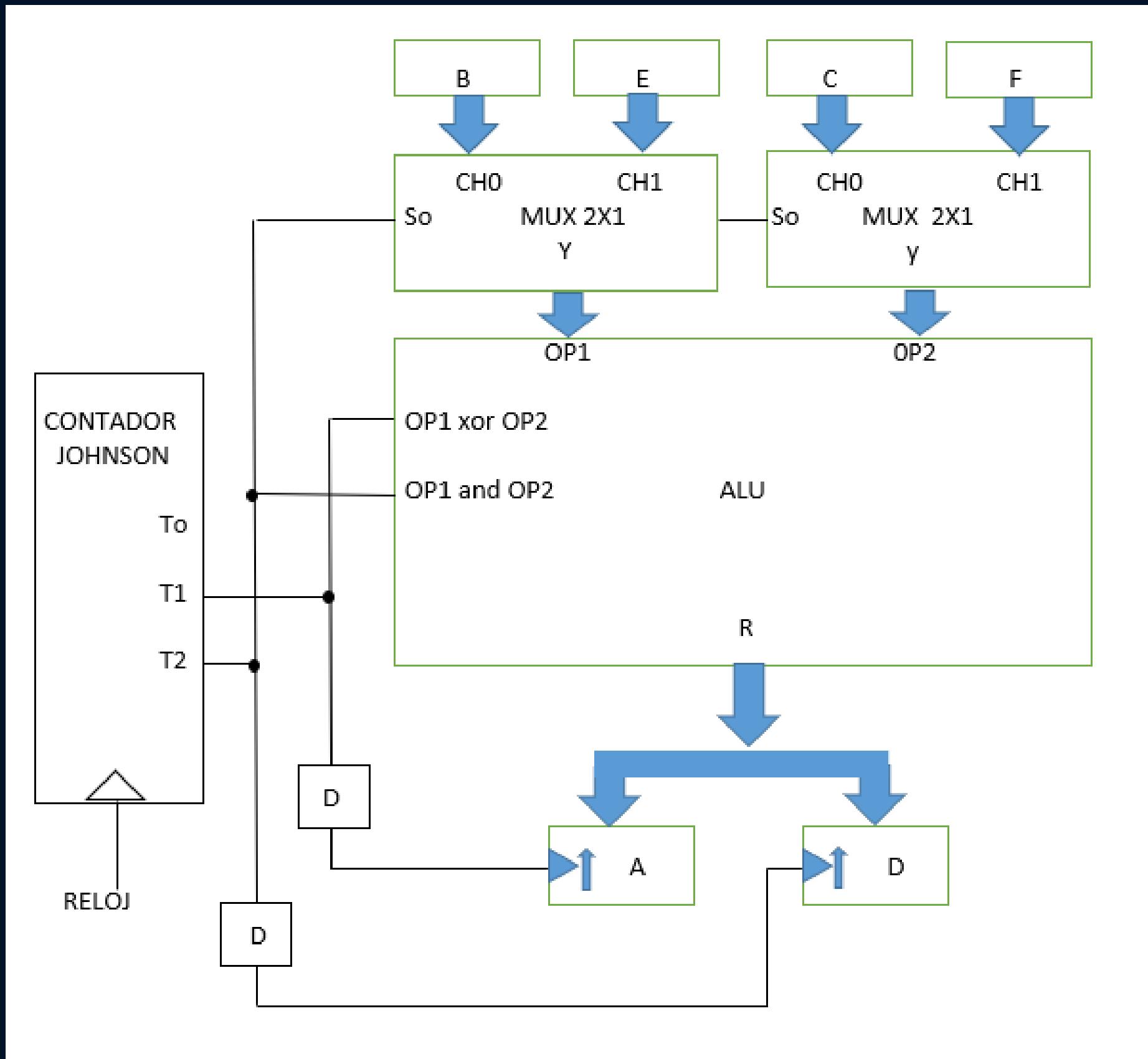
# ASSEMBLER

---

Es el lenguaje de programación de nivel más bajo que los humanos pueden usar para comunicarse con el hardware de una computadora. La programación de lenguaje de ensamblaje se trata de escribir instrucciones en un lenguaje que la computadora puede entender. Requiere una comprensión profunda de la arquitectura del procesador y la capacidad de pensar de una manera lógica y paso a paso.



# ARQUITECTURA DE UN PROCESADOR



# SINTAXIS DE LA PROGRAMACIÓN DEL LENGUAJE DE ENSAMBLAJE

---

Cada lenguaje de ensamblaje tiene su propia sintaxis. Sin embargo, la estructura básica es similar en todos los idiomas de ensamblaje. El código de ensamblaje se compone de instrucciones, directivas y comentarios.

- Las instrucciones son las operaciones reales que realizará la computadora, como mover datos entre registros o realizar operaciones aritméticas.
- Las directivas proporcionan información adicional al ensamblador, como dónde almacenar datos o cómo alinear las instrucciones en la memoria.
- Los comentarios se utilizan para anotar el código con texto legible por humanos.

# TIPOS DE ASSEMBLER

---

- MASM (Microsoft Macro Assembler): usado en arquitectura x86, el que utiliza sintaxis Intel para MS-DOS y Microsoft Windows.
- GAS (GNU Assembler): usado en el proyecto GNU, y es el back-end por defecto de GCC.
- NASM (Netwide Assembler): usado en arquitectura x86 para escribir programas de 16, 32 (IA-32) y 64 bits (x86-64).
- FASM (Flat Assembler): usado en arquitectura x86, y soporta el lenguaje assembly en estilo Intel en IA-32 y x86-64.
- AVR: Es utilizado en la programación de microcontroladores.

# MNEMÓNICOS

---

Es una palabra que sustituye a un código de operación (lenguaje de máquina), con lo cual resulta más fácil la programación.



# PRINCIPALES MNEMÓNICOS

---

- **MOV (Move):** La instrucción MOV copia datos de una ubicación a otra. Se utiliza para transferir datos entre registros, entre registros y memoria, o entre direcciones de memoria.
- **ADD (Addition):** Esta instrucción realiza la suma aritmética. Suma dos operandos y almacena el resultado en el primer operando.
- **SUB (Subtraction):** Realiza la resta aritmética. Resta el segundo operando del primer operando y almacena el resultado en el primer operando.
- **CMP (Compare):** Compara dos operandos sin realizar una operación de modificación. Compara los operandos y actualiza los bits de estado (flags) en función del resultado de la comparación.
- **JMP (Jump):** Transfiere el control de ejecución del programa a una ubicación específica. Se utiliza para implementar saltos condicionales e incondicionales.

# PRINCIPALES MNEMÓNICOS

---

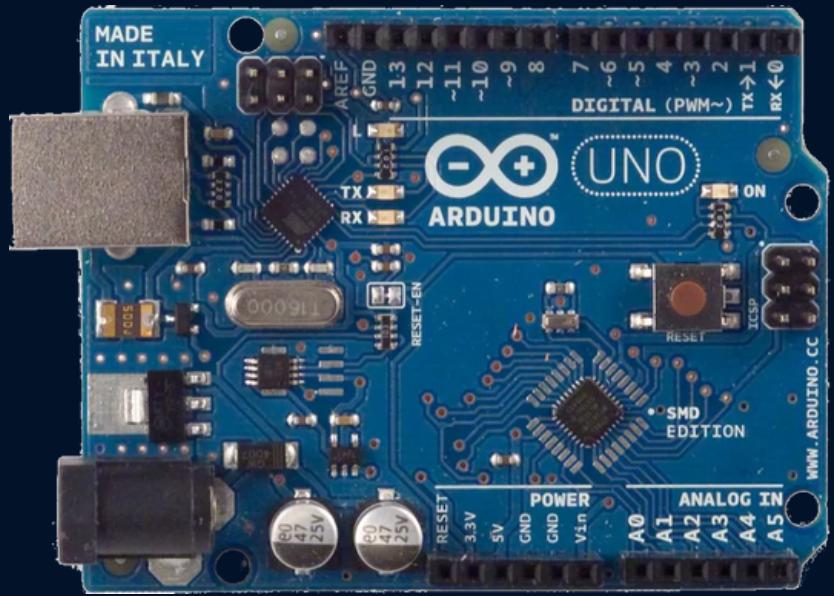
- **JE (Jump if Equal):** Realiza un salto condicional si los operandos son iguales. Se utiliza en conjunto con la instrucción CMP para realizar un salto si el resultado de la comparación indica igualdad.
- **JNE (Jump if Not Equal):** Realiza un salto condicional si los operandos no son iguales. Similar a JE, pero realiza un salto si los operandos no son iguales.
- **LOOP:** Implementa un bucle para repetir una secuencia de instrucciones. Se utiliza en conjunto con la instrucción CX para repetir un conjunto de instrucciones un número determinado de veces.
- **CALL:** Llama a una subrutina o procedimiento. Transfiere el control del programa a una subrutina, almacenando la dirección de retorno en la pila.
- **RET (Return):** Retorna de una subrutina a la instrucción de llamada. Restaura la dirección de retorno de la pila y devuelve el control del programa a la instrucción después de la llamada.

# PRINCIPALES MNEMÓNICOS

---

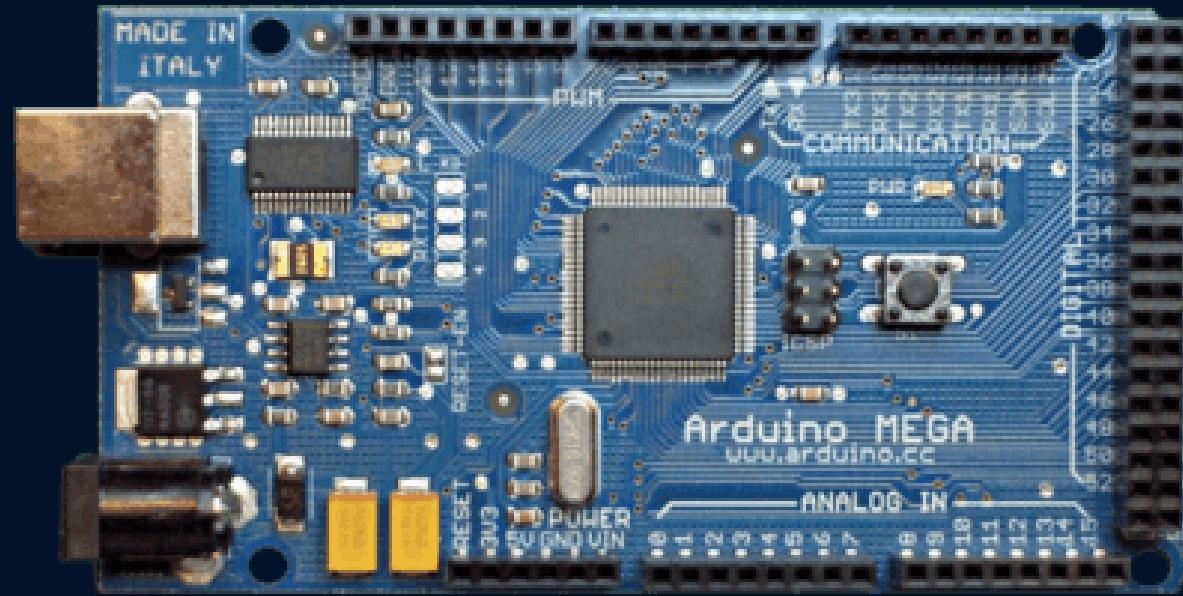
- **JE (Jump if Equal):** Realiza un salto condicional si los operandos son iguales. Se utiliza en conjunto con la instrucción CMP para realizar un salto si el resultado de la comparación indica igualdad.
- **JNE (Jump if Not Equal):** Realiza un salto condicional si los operandos no son iguales. Similar a JE, pero realiza un salto si los operandos no son iguales.
- **LOOP:** Implementa un bucle para repetir una secuencia de instrucciones. Se utiliza en conjunto con la instrucción CX para repetir un conjunto de instrucciones un número determinado de veces.
- **CALL:** Llama a una subrutina o procedimiento. Transfiere el control del programa a una subrutina, almacenando la dirección de retorno en la pila.
- **RET (Return):** Retorna de una subrutina a la instrucción de llamada. Restaura la dirección de retorno de la pila y devuelve el control del programa a la instrucción después de la llamada.

# ARDUINO UNO



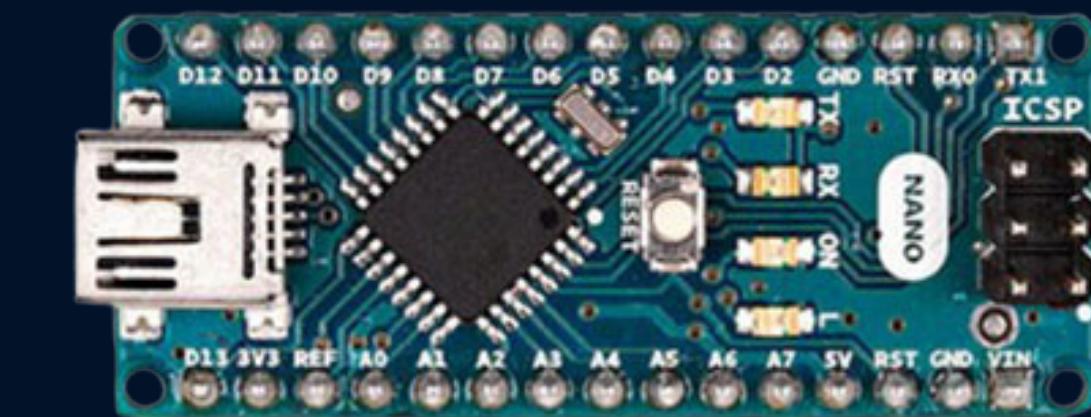
Microcontrolador: ATMega328P

# ARDUINO MEGA



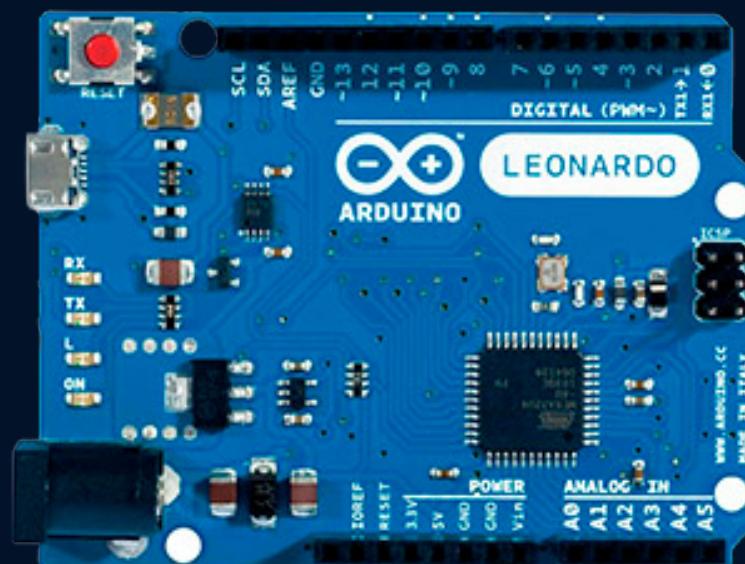
Microcontrolador: ATMega2560

# ARDUINO NANO



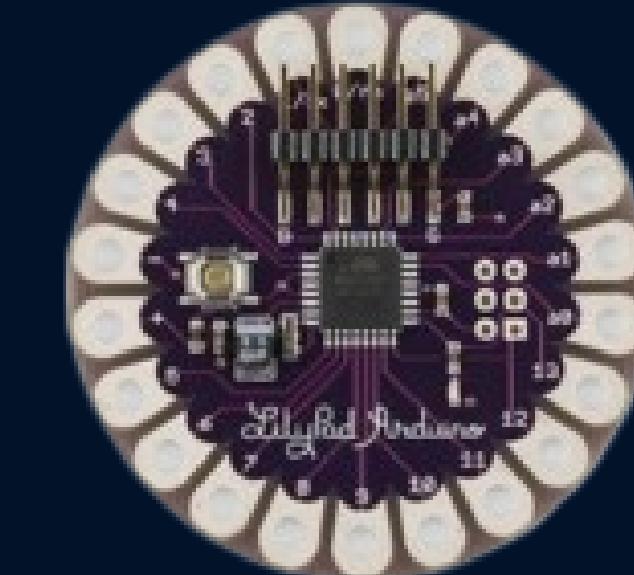
Microcontrolador: ATMega328P

# ARDUINO LEONARDO



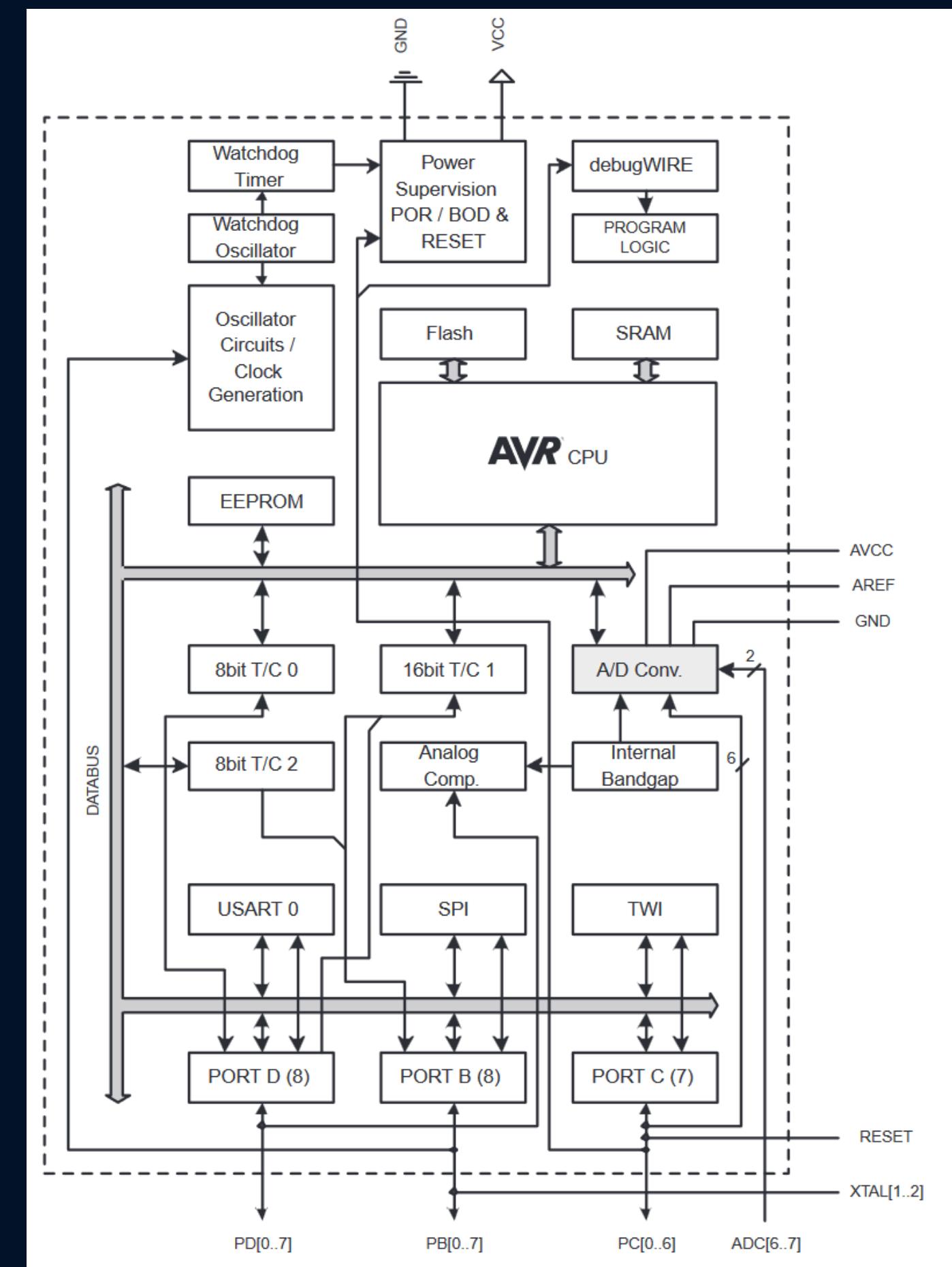
Microcontrolador: ATMega32u4

# ARDUINO LILYPAD



Microcontrolador: ATMega328V

# ATMEGA328P



# ATMEGA328P

---

PDF

# Data Sheet

# REGISTROS

10.11.3 PRR – Power Reduction Register								
Bit (0x64)	7	6	5	4	3	2	1	0
Read/Write	PRTWI	PRTIM2	PRTIMO	-	PRTIM1	PRSPI	PRUSART0	PRADC
Initial Value	0	0	0	0	0	0	0	0
	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W

## BIT 7 - PRTWI: POWER REDUCTION TWI

Escribir un uno lógico en este bit apaga el TWI al detener el reloj del módulo. Al despertar el TWI nuevamente, el TWI debe reiniciarse para garantizar un funcionamiento adecuado.

## BIT 6 - PRTIM2: POWER REDUCTION TIMER/COUNTER2

Escribir un uno lógico en este bit apaga el módulo Temporizador/Contador2 en modo síncrono (AS2 es 0). Cuando el Temporizador/Contador2 está habilitado, la operación continuará como antes del apagado.

## BIT 5 - PRTIMO: POWER REDUCTION TIMER/COUNTER0

Escribir un uno lógico en este bit apaga el módulo Timer/Counter0. Cuando el Temporizador/Contador0 está habilitado, la operación continuará como antes del cierre.

## BIT 4 - RESERVED

Es un bit reservado y siempre será 0.

## BIT 3 - PRTIM1: POWER REDUCTION TIMER/COUNTER1

Escribir un uno lógico en este bit apaga el módulo Temporizador/Contador1. Cuando el Temporizador/Contador1 está habilitado, la operación continuará como antes del cierre.

## BIT 2 - PRSPI: POWER REDUCTION SERIAL PERIPHERAL INTERFACE

Si utiliza el sistema de depuración en chip debugWIRE, este bit no debe escribirse en uno. Escribiendo uno lógico en este bit apaga la interfaz periférica serie deteniendo el reloj del módulo. Al despertar el SPI nuevamente, el SPI debe reinicializarse para garantizar un funcionamiento adecuado.

## BIT 1 - PRUSART0: POWER REDUCTION USART0

Escribir un uno lógico en este bit apaga el USART al detener el reloj del módulo. Al despertar el USART nuevamente, el USART debe reinicializarse para garantizar un funcionamiento adecuado.

## BIT 0 - PRADC: POWER REDUCTION ADC

Escribir uno lógico en este bit apaga el ADC. El ADC debe desactivarse antes de apagarse. El análogo El comparador no puede utilizar la entrada MUX del ADC cuando el ADC está apagado.

# TMR 0 VS DELAY



modo tieso