Universidad Europea

LAUREATE INTERNATIONAL UNIVERSITIES

REGISTROS Y MEMORIA EN ATMEL

Índice

Presentación

A estos niveles ya no es necesario decir que un ordenador y, por consiguiente, una CPU, solo pueden manejar **dos valores** (1/0, H/L, T/F, etc.) y que esta unidad de información mínima se llama **bits**. Históricamente, por razones constructivas, es fácil de manejar grupos de 8 bits, a lo que se le llama **byte**.

Pero, ¿y la palabra? ¿Qué es una palabra en términos de almacenamiento de información? Una palabra es una **unidad de tamaño lógica** (no está completamente definida) de ahí que cada fabricante la especifique. Para Intel* actualmente palabra son 64 bits, es decir, 8 bytes. Hasta no hace mucho, palabra era 32 bits, cuatro bytes (para máquinas 32 bits), es una unidad ficticia, que cada fabricante establece su tamaño de palabra.

La decisión de establecer uno u otro tamaño viene definido para lograr la máxima **efectividad y rendimiento** en un microprocesador. La palabra, es el **tamaño de datos** donde el microprocesador es **óptimo en su funcionamiento**, ya que todo el hardware del ordenador está diseñado para trabajar con ese conjunto de datos, de manera que la velocidad de procesamiento de información sea máxima. Otros tamaños, ya sean mayores o menores, provocan una pérdida de efectividad.

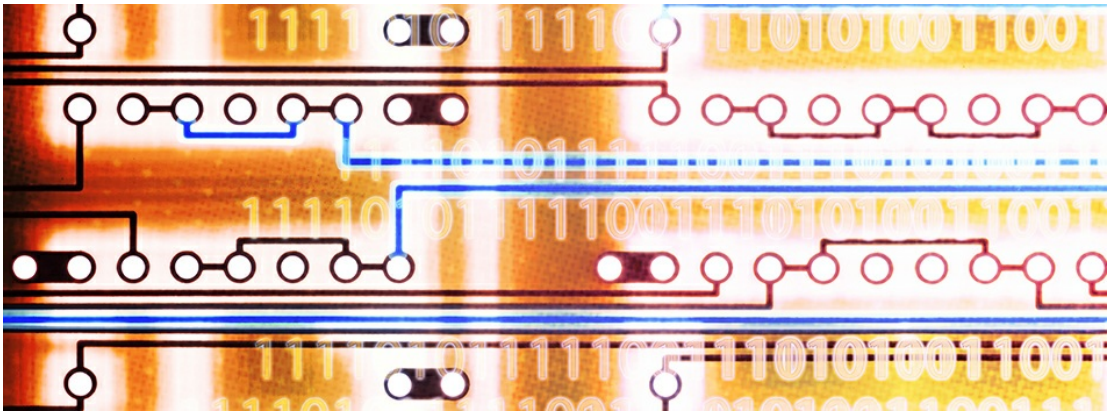
Para los microprocesadores ATmega * (los que incorporan muchas de las placas Arduino *), la palabra se define en 16 bits, es decir, dos bytes.

Siguiendo con obviedades, está claro que todo en un ordenador está representado, almacenado y manipulado utilizando **solo bits** (números enteros, números en punto flotante, caracteres, documentos, programas, archivos, carpetas, imágenes, sonidos, vídeos, etc., los cuales se codifican mediante números).

Esto lleva a la necesidad de definir **estándares** para la codificación utilizando los bits.

Los **objetivos** que se pretenden alcanzar con este recurso son los siguientes:

- Analizar el **almacenamiento de datos** en ensamblador.
- Identificar los estándares para la **codificación** utilizando bits.



* *Esta marca es una marca registrada que se cita para uso exclusivamente docente.*

Ejemplos de codificación estándar

A continuación, se muestra una serie de ejemplos de estándares, con su respectiva codificación:

- **Número entero sin signo** : binario natural: $17_d = 00010001_2$.
- **Número entero con signo** en complemento a 2: $-17_d = 11101111_2$.
- **Número real en punto flotante**.
- **Carácter**: en codificación ASCII: 'a' = $97_d = 1100001_2$.
- **Imagen**: matriz de píxeles (se dispone como una matriz solo en la pantalla).
    - El color de cada píxel se debe especificar (utilizando los números de colores, ex RGB).
    - Una imagen de 1600x900 píxeles (resolución de mi escritorio) en color verdadero usa 46080000bits (5MiB).
- **Documento**: conjunto de caracteres, imágenes, posiciones y formatos, también codificada como números.
- **Sonido**: una matriz de números (se interpreta como sonido de los altavoces).
- **Vídeo**: una matriz con una secuencia de imágenes y sonidos intercalados (AVI).
- ASCII

ASCII

American Standard Code for Information Interchange (ASCII)

000 (nul)	016 (dle)	032 sp	048 0	064 @	080 P	096 `	112 p
001 (soh)	017 (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q
002 (stx)	018 (dc2)	034 "	050 2	066 B	082 R	098 b	114 r
003 (etx)	019 (dc3)	035 #	051 3	067 C	083 S	099 c	115 s
004 (eot)	020 (dc4)	036 S	052 4	068 D	084 T	100 d	116 t
005 (enq)	021 (nak)	037 %	053 5	069 E	085 U	101 e	117 u
006 (ack)	022 (syn)	038 &	054 6	070 F	086 V	102 f	118 v
007 (bel)	023 (etb)	039 '	055 7	071 G	087 W	103 g	119 w
008 (bs)	024 (can)	040 (	056 8	072 H	088 X	104 h	120 x
009 (tab)	025 (em)	041)	057 9	073 I	089 Y	105 i	121 y
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z
011 (vt)	027 (esc)	043 +	059 ;	075 K	091 [	107 k	123 {
012 (np)	028 (fs)	044 ,	060 <	076 L	092 \	108 l	124 |
013 (cr)	029 (gs)	045 -	061 =	077 M	093]	109 m	125 }
014 (so)	030 (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~
015 (si)	031 (us)	047 /	063 ?	079 O	095 _	111 o	127 □
Control codes		Printable characters (punctuation, numbers, upper and lower-case letters					

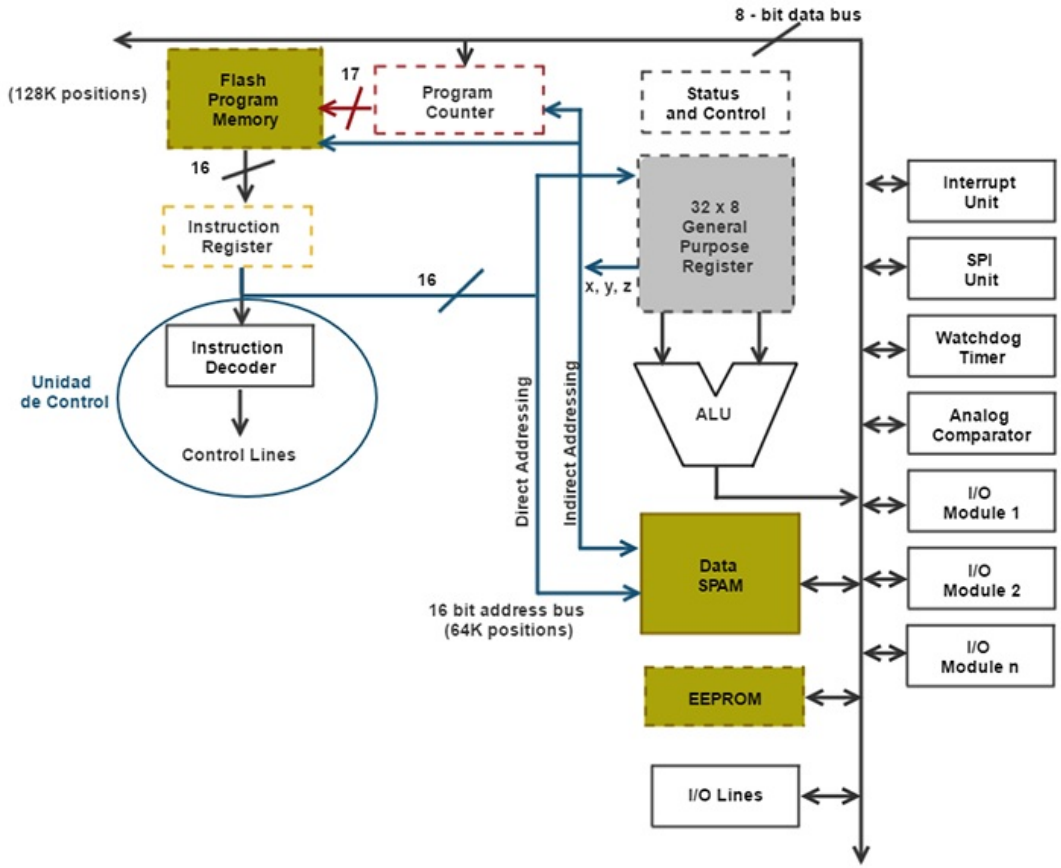
La CPU trabaja con los bits, pero, ¿qué es en realidad un bit? ¿Qué aspecto físico tiene un bit?

Un bit puede tener una serie de **cualidades** dependiendo de cómo se almacena o se manipula:

Cuando se almacena	• Es un **potencial eléctrico** (voltios) bajo o alto (L/H), por ejemplo, en un condensador o de transistores (memorias). • Es una **región magnetizada** (disco duro, cintas, disquetes). • Es una **región brillante/opaco** (CD/DVD/BlueRay).
Cuando es manipulado en la CPU	• Es un **potencial eléctrico bajo o alto** (H/L) almacenado en un registro y/o en un bus de comunicación: ◦ Un **bus transmite un bit de un circuito a otro**, de forma instantánea y sin modificarlo.

Atmega328 arquitectura de CPU

En el siguiente gráfico se muestra el diagrama en bloques de la **arquitectura de un microcontrolador Atmega328**:

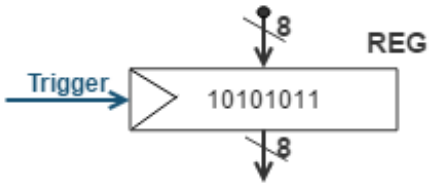


Fuente: cortesía de Atmel

Registros

Un registro es un componente electrónico básico, cuya función es **almacenar temporalmente un valor**.

- Funciona como una pequeña (pero extremadamente rápida) memoria para uno o más bits (típicamente una palabra).

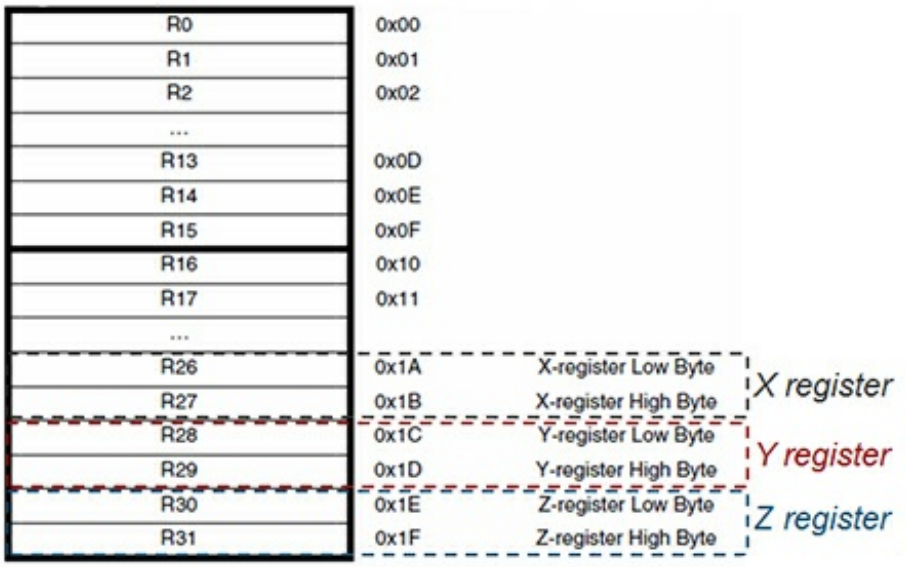


- Solo se actualiza el valor almacenado cuando se activa.
    - La señal de disparo para modificar su valor almacenado proviene de la unidad de control.
- En caso de que no sea disparada, mantiene el valor almacenado indefinidamente, no necesitando procesos de refresco de la información.
- El registro más utilizado en microprocesadores, es el registro paralelo-paralelo:
    - Los datos se almacenan o leen todos a la vez (paralelo).

En detalle

Banco de registros de propósito general del Atmega2560

Banco de registros de propósito general del Atmega2560



Atmega328 tiene **32 registros** de propósito general de **ocho bits cada** uno.

Los registros ocupan las primeras posiciones de la memoria de datos SRAM.

Aunque son registros de carácter general, y todos **son accesibles desde programa**, algunos de ellos presentan características especiales, que modifican ligeramente sus posibles usos:

- Del registro R0 a R15 **no se puede almacenar un valor constante** directamente, solo acepta datos de otros registros. Por lo que pueden ser solo utilizables como fuente y destino de operaciones aritméticas.
- Del registro R16 a R31 **permiten escritura de constantes**, así como la utilización en operaciones. Estos son los más fáciles de usar.
- Los registros del R26 a R31 también se pueden utilizan para **formar registros dobles de 16 bits** (para datos de 16 bits y direcciones).

Registros de 16 bits

Los registros del R26 a R31 pueden ser **utilizados en pares** para formar tres **registros de 16 bits**:

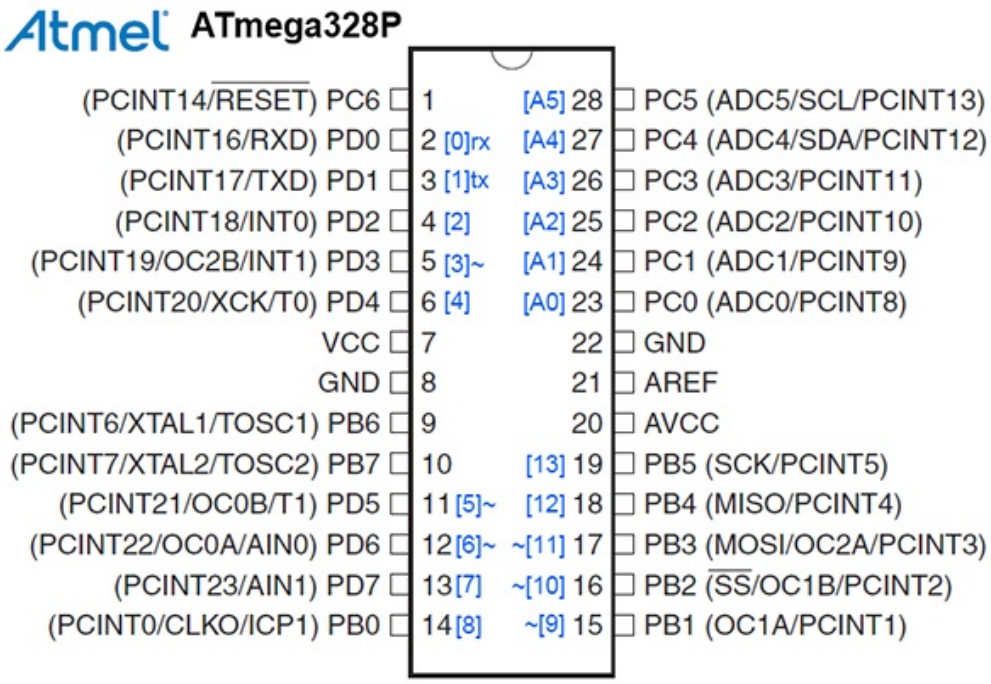
- R27:R26 forman el registro "X".
- R29:R28 forman el registro "Y".
- R31:R30 forman el registro "Z".

Estos pares de registros, X, Y, y Z se pueden utilizar para **almacenar datos de 16 bits**, lo que permite a una CPU de 8 bits como los ATmega operar rápidamente con datos de 16 bits.

También tienen la función de aumentar y facilitar el **manejo de punteros**. Ya que están conectados a circuitos específicos que permiten auto-incremento y auto-decremento en las instrucciones de movimiento de datos (lo que permite manejarse rápidamente y fácilmente con arrays y matrices).

Registros de entrada/salida del ATmega328p

El Atmega328 tiene tres puertos de E/S de ocho bits cada uno, para comunicarse con "el mundo". Estos puertos tienen su **correspondiente terminal** en la placa Arduino*, tal y como se puede ver en la siguiente imagen:



Fuente: cortesía de Atmel

Todos los puertos tienen pines I/O bidireccionales que pueden actuar como **entradas** o **salidas digitales** (cada bit/pin se puede configurar de forma independiente como entrada o como salida).

Estos puertos, además de servir como comunicación con el exterior y de ser utilizables desde programa, **tienen funciones alternativas**:

Puerto B (bits PB7, ..., PB1, PB0)	Cuenta con una **mejor capacidad de carga eléctrica** (se puede manejar más corriente) que los otros puertos. También proporciona salidas PWM (Pulse-Width Modulation), comunicación SPI e I2C.
Puerto D (bits PD7, ..., PD1, PD0)	También actúa como **byte de direcciones** para el control de una memoria externa, la comunicación a la USART1 (Universal Synchronous and Asynchronous serial Receiver and Transmitter) y alguna PWM (Pulse-Width Modulation), puede leer interrupciones externas.

Puerto C (bits PC7, ..., PC1, PC0)	También entradas **analógicas o como conversos ADC** (Analog to Digital Conversor).

Algunas placas Arduino integran microprocesadores más grandes, con más puertos de entrada salida, proporcionando mayores posibilidades de comunicación. Por ejemplo, el ATmega*, que integra el Arduino mega, tiene 11 puertos de comunicación, nombrados del PORTA al PORTL.

* *Esta marca es una marca registrada que se cita para uso exclusivamente docente.*

Instrucciones de movimiento de datos. LDI

Son muchas las instrucciones de movimiento de datos que existen, pero quizás la de mayor importancia es la que permite **cargar un registro con un valor constante**. Esta operación es: LDI (LoaD Immediate).

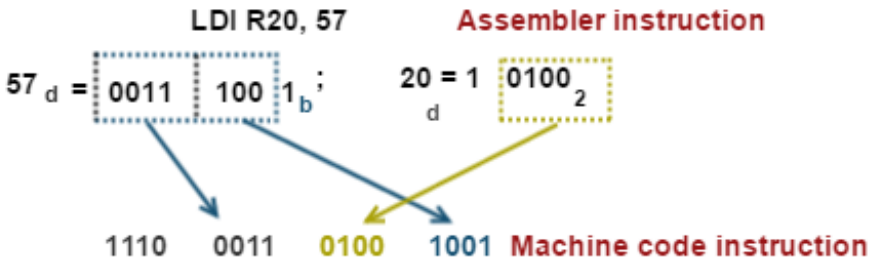
Formato	**LDI Rd, k** Donde el valor "k" puede ser: • Un **numero constante**, menos que 256 (8 bits). • La constante **se incluye en la misma instrucción** , por lo que se llama "un valor inmediato". • **Rd debe ser uno de R16 .... R31** (esta operación no funciona con R0 ... R15). ○ d" es de 0000 (R16) a 1111 (R31).
Codificación	La instrucción ocupa **16bits en memoria del microprocesador** , codificados de la siguiente manera: • Código de operación: 4 bits → 1110_2 (=16_{10}). • Inmediato: 8 bits: k7...k0. • Registro: 4 bits: d3...d0.

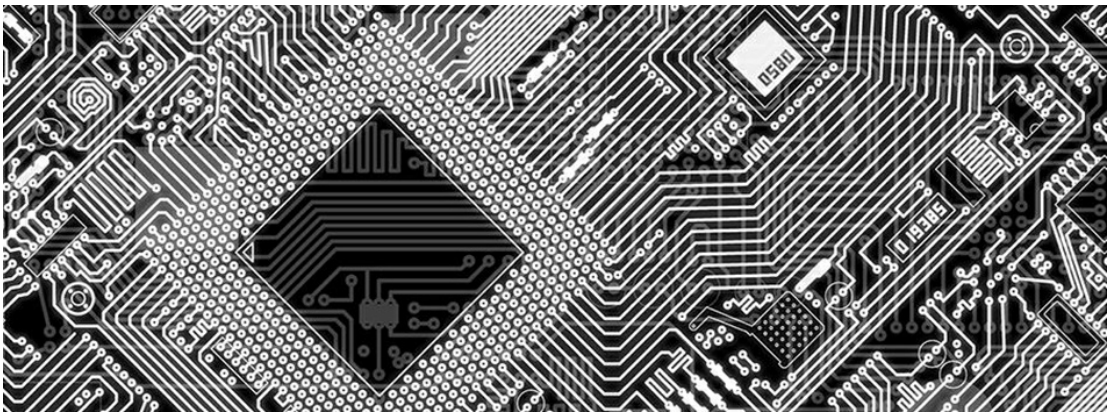
Ejemplo

Ejemplo de codificación: LDI R20, 57

Ejemplo de codificación: LDI R20, 57

- Cargar un 57d en R20 y calcular el código de máquina para esta instrucción.

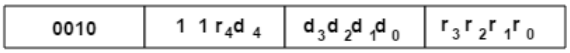




Otras instrucciones de movimiento de datos

A continuación, veremos más detenidamente y ejemplificaremos el funcionamiento de las instrucciones MOV y las instrucciones de entrada y salida.

Instrucción MOV (y MOVW)

Instrucción MOV (MOVe): copia el contenido de un registro a otro registro	**Formato**: MOV Rd, Rr. • **Copia el contenido de Rr en Rd (Rd ← Rr).** El primer operando es destino, el segundo origen. • Rd y Rr puede ser cualquier registro (R0 ... R31). • La **instrucción se codifica en 16 bits** , con la siguiente estructura: 
Instrucción MOVW (MOVe Word): variante del MOV, copia 16 bits a la vez.	**Formato**: MOVW Rd, Rr (copia [Rd+1:Rd] ← [Rr+1:Rr]). • **Copias 2 registros adyacentes a otros 2 registros adyacentes** (en un solo ciclo de reloj). • Ejemplo: MOVW R10, R15: ◦ Copia el contenido del registro R15 en R10 y R16 en R11 respectivamente.

Ejemplo de la instrucción MOV: MOV R9, R20

Ejemplo de la instrucción MOV: MOV R9, R20

- Copiar el contenido de R20 en la R9.

Instrucción: OUT / IN

Los pines que comunican con el "mundo" físico, son accesibles a través de un registro especial, el **registro "PORTp"**. Por consiguiente, para poder escribir o leer un valor del mundo, es necesario utilizar este registro. Para poder de escribir o leer valores en estos registros hay que utilizar las **instrucciones OUT e IN**.

Enviar un valor a un puerto: OUT (OUT port)	**Formato**: OUT PORTp, Rr Copia el contenido de Rr al puerto P (P ← RR) • Rr puede ser cualquier registro (R0 ... R31). • El puerto, puede ser un numero entre 0 ... 63, cada número especifica un registro asociando a un puerto. Atmel Studio*, proporciona alias a todos los puertos de manera automática.
Leer un valor de un puerto: IN (IN port)	**Formato**: IN Rd, PORTp • Copia el contenido del canal P hacia Rd (Rd ← PORTp).

Ejemplo de mapeado con número de puerto y alias (Atmel Studio)

Ejemplo

Ejemplo de mapeado con número de puerto y alias (Atmel Studio)

En la siguiente tabla, podemos ver los números de puerto y sus correspondientes alias.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x1A (0x3A)	TIFR5	-	-	ICF5	-	OCF5C	OCF5B	OCF5A	TOV5	166
0x19 (0x39)	TIFR4	-	-	ICF4	-	OCF4C	OCF4B	OCF4A	TOV4	167
0x18 (0x38)	TIFR3	-	-	ICF3	-	OCF3C	OCF3B	OCF3A	TOV3	167
0x17 (0x37)	TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2	193
0x16 (0x36)	TIFR1	-	-	ICF1	-	OCF1C	OCF1B	OCF1A	TOV1	167
0x15 (0x35)	TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0	134
0x14 (0x34)	PORTG	-	-	PORTG5	PORTG4	PORTG3	PORTG2	PORTG1	PORTG0	102
0x13 (0x33)	DDRG	-	-	DDG5	DDG4	DDG3	DDG2	DDG1	DDG0	102
0x12 (0x32)	PING	-	-	PING5	PING4	PING3	PING2	PING1	PING0	102
0x11 (0x31)	PORTF	PORTF7	PORTF6	PORTF5	PORTF4	PORTF3	PORTF2	PORTF1	PORTF0	101
0x10 (0x30)	DDRF	DDF7	DDF6	DDF5	DDF4	DDF3	DDF2	DDF1	DDF0	102
0x0F (0x2F)	PINF	PINF7	PINF6	PINF5	PINF4	PINF3	PINF2	PINF1	PINF0	102
0x0E (0x2E)	PORTE	PORTE7	PORTE6	PORTE5	PORTE4	PORTE3	PORTE2	PORTE1	PORTE0	101
0x0D (0x2D)	DDRE	DDE7	DDE6	DDE5	DDE4	DDE3	DDE2	DDE1	DDE0	101
0x0C (0x2C)	PINE	PINE7	PINE6	PINE5	PINE4	PINE3	PINE2	PINE1	PINE0	102
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	101
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	101
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	101
0x08 (0x28)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	101
0x07 (0x27)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	101
0x06 (0x26)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	101
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	100
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	100
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	100
0x02 (0x22)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	100
0x01 (0x21)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	100
0x00 (0x20)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	100

Notes: When using the I/O specific commands IN and OUT, the I/O addresses $00 - $3F must be used. When addressing I/O registers as data space using LD and ST instructions, $20 must be added to these addresses. The ATmega640/1280/1281/2560/2561 is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from $60 - $1FF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

Fuente: cortesía de Atmel

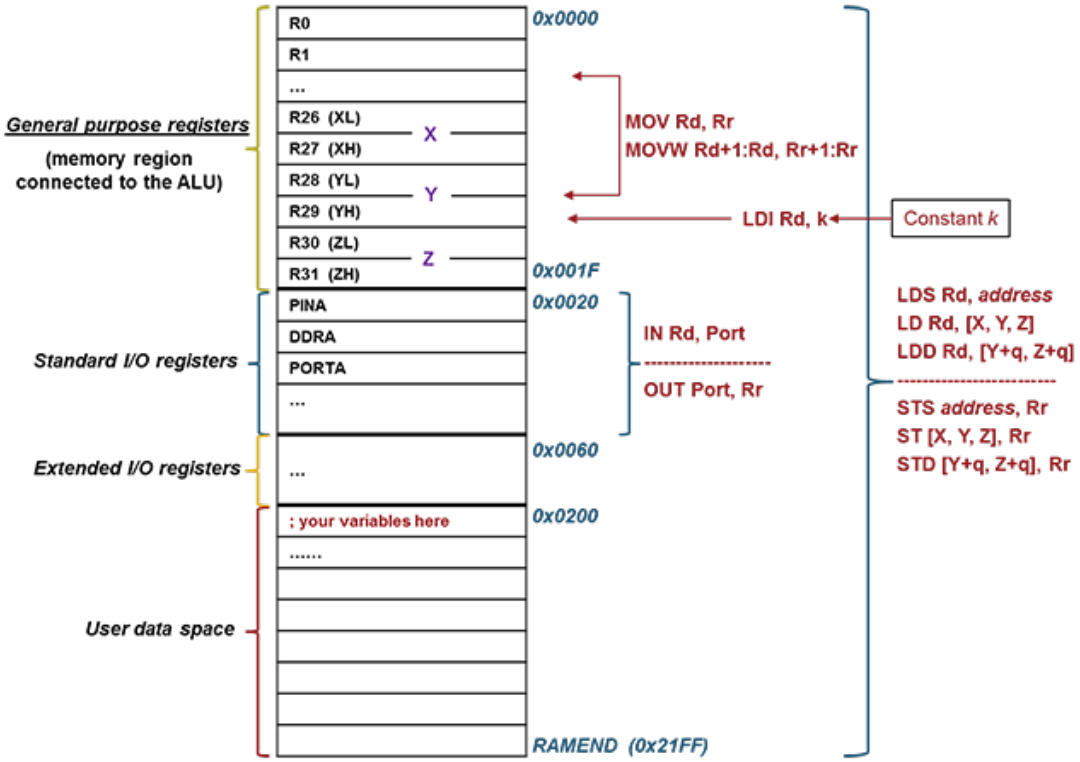
* *Esta marca es una marca registrada que se cita para uso exclusivamente docente.*

La memoria del ATmega328p

La memoria del Atmel* ATmega328p está organizada en **cuatro secciones**, con un **mapa de memoria común**. Esto significa, que, aunque se utilicen componentes y equipos distintos para los registros o para el banco de variables de usuario, el programador tiene la sensación de la existencia de un solo banco de almacenamiento continuo. Ya que las direcciones pasan de un componente a otro de manera transparente para el usuario.

Así, la organización lógica (a nivel de direcciones, no físicamente), se divide en cuatro secciones:

- Las primeras 32 direcciones son los **registros de propósito general**, R0...R31.
- Las siguientes 64 direcciones, están los **registros de entrada/salida** (PORTp, DDRp, PINp, etc).
- Las siguientes 416 direcciones son **registros de entrada/salida extra**.
- Y desde la dirección 0x200 en adelante, es la **memoria disponible para variables de usuario**.



* *Esta marca es una marca registrada que se cita para uso exclusivamente docente.*

Las variables del programa y vectores (array)

Tal como hemos visto hasta ahora, las variables **pueden ser almacenadas en los registros de la CPU**, pero solo disponemos de 32 registros, por lo que estos se nos pueden quedar cortos. ATmega, dispone de memoria **SRAM para el almacenaje de variables** por parte del usuario. La cantidad de variables a almacenar, depende del tipo de microprocesador y del tipo de variables. Así, el ATmega328p tiene 32K de memoria de 8 bits.

Más rápido	En las **variables almacenadas en registros**, el acceso es el más rápido y dado que los registros se conectan directamente a la ALU, los datos puedan ser manipulados tan pronto como se cargan con un nuevo valor.
Más lento	Por el contrario, si utilizamos memoria para los datos, dispondremos de **mayor capacidad de almacenamiento**, pero deberán copiarse en un registro antes de operar sobre ellos (nuevo acceso a la memoria), y el resultado deberá ser escrito de nuevo en la memoria (nuevo acceso a la memoria). Lo que originará que sea una ejecución muy lenta.

Tipos de variables

En los **lenguajes de programación de alto nivel**, se definen las variables por el tipo de dato que va a contener, ajustando el compilador el tamaño y localización de las mismas para su correcta utilización. Así los tipos de variables como **bool**, **char**, **int**, **float**, **string**, **objetos**, etc. son de uso muy común.

Sin embargo, los tipos de variables **no existen en código máquina**, y por tanto, tampoco en ensamblador. Todas las variables son iguales, la diferencia es el tamaño de esa variable en memoria, ya que de eso depende el valor máximo que se puede almacenar en ella. Es una responsabilidad del programador operar con los bytes correctos para cada variable (el ensamblador no se hará cargo de esto).

Afortunadamente el compilador (o traductor) de ensamblador acepta utilizar etiquetas "humanas" para las variables:

- Así la directiva ".DEF" permite redefinición de un registro:
    - Ejemplo: .DEF STUDENT_SCORE = R26
        - Esto nos permite utilizar indistintamente "STUDENT_SCORE" o R26. De todos modos, el traductor ensamblador los sustituirá por la dirección de memoria real para R26 (0x1A).

Para las variables almacenadas en memoria, usaremos la directiva ".DSEG" (Data SEGment) que utilizaremos así:

```
.DSEG   ;Indican que vamos a utilizar memoria de datos
    var1: .BYTE 2       ; llamar "var1" a la primera dirección de memoria y reserva 2 bytes
    var2: .BYTE 1       ; llamar "var2" a la siguiente dirección disponible y reserva 1 byte más
    array1: .BYTE 32    ; llamar "array1" a la siguiente dirección disponible y reserva 32 bytes más
```

Para acceder a estas variables vasta con utilizar el **nombre que se le ha dado**, dejando al traductor de ensamblador reemplazar esas "etiquetas humanas" por las correspondientes direcciones de memoria: 0x200 para var1, var2 para 0x202, 0x203 para array1, etc.

Cabe destacar que, para acceder a las diferentes posiciones de una array, se pueden utilizar los formatos:

- array1[0], array1[1], array1[31].
- array1, array1+1, array1+31.

Instrucciones para la lectura/escritura de memoria. LDS

Leyendo una posición de memoria SRAM: LDS (LoaD Sram)

Formato: LDS Rd, k

Copia el contenido de la dirección de memoria k en Rd: Rd ← (k) = contenido de k:

- Rd puede ser cualquier registro (R0 ... R31).
- k debe estar entre: 0 ≤ k ≤ 32768 (32K).

Esta es una instrucción de 32 bits. El código de operación utiliza 2 posiciones de memoria de programa:

1001	$000d_4$	$d_3d_2d_1d_0$	0000
kkkk	kkkk	kkkk	kkkk

Y consume dos ciclos de reloj para ejecutarse porque **necesita dos accesos de memoria de programa**.

Tal como vimos anteriormente, en ensamblador podemos utilizar etiquetas de las variables en la SRAM, etiquetas creadas con la directiva ".DSEG":

- Ejemplo: LDS R1, var1

Será el programa ensamblador el encargado de calcular la dirección de memoria que estamos utilizando.

NOTA: Fíjese que se está utilizando **direccionamiento directo**, ya que la dirección de la memoria final se declaró directamente en la instrucción.

Escribiendo una posición de memoria: STS (STore Sram)

Formato: STS k, Rr

Copia el contenido de Rr en la dirección de memoria k: (k) (Rr:

- Rr puede ser cualquier registro (R0 ... R31).
- k debe estar entre: 0 ≤ k ≤ 32768 (32K).

Al igual que en el caso anterior:

- Esta es una **instrucción de 32 bits**. El código de operación utiliza 2 posiciones de memoria de programa:

1001	$001r_4$	$r_3r_2r_1r_0$	0000
kkkk	kkkk	kkkk	kkkk

- Consume dos ciclos de reloj.
- Podemos utilizar etiquetas de las variables en la SRAM.

Lectura/escritura en memoria, direccionamiento indirecto

La lectura indirecta de una posición de memoria LD (LoaD) tiene el siguiente formato:

Formato: LD Rd, X.

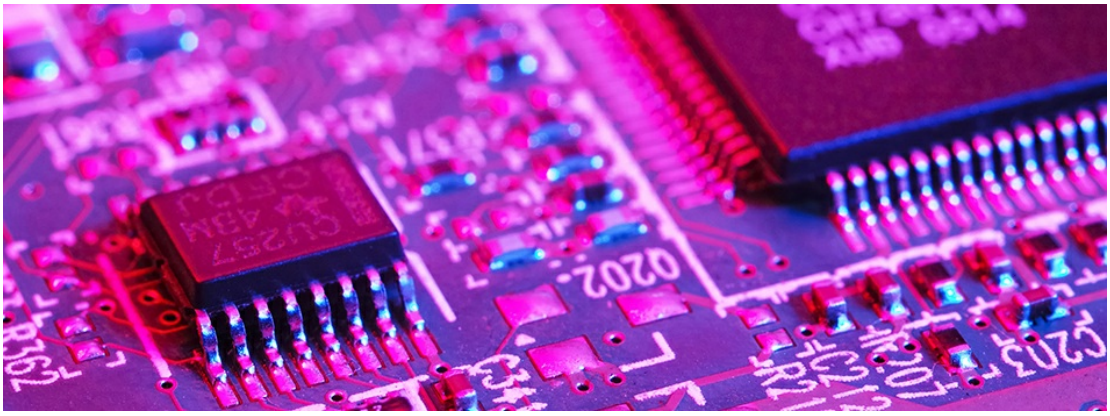
Copia el contenido de la memoria de datos indicado por el par de registros X en Rd: Rd <- (X):

- Rd puede ser cualquier registro (R0 ... R31).
- X puede ser el registro X, Y o Z.

Nota: Este modo de direccionamiento se llama indirecta porque la dirección de la memoria final **no está en la instrucción** , sino que se codifica a través de un registro.

Variantes de la instrucción LD	Los registros dobles X, Y, Z **se pueden utilizar en dos mitades**, la parte alta y parte baja. Así, la secuencia:
	`1. LDI XH, HIGH(my_var)`
	`2. LDI XL, LOW(my_var) ;inicializa el registro X, por medio de la parte alta y baja de la dirección de la variable`
	`3. LD R15, X ;leemos my_var utilizando el puntero X`
	Nos permite señalar registro X a cualquier variable de interés. Por ejemplo, para copiar en my_var R15.
	Los registros X, Y y Z, también permiten pre y post incremento automático. Se utilizan para la manipulación de matrices en un bucle:
	Ejemplo:
	- LD Rd, X+ ; Rd ← (X), X ← X+1
	- LD Rd, -X ; X ← X-1, Rd ← (X)

Escritura indirecta de una posición de memoria: ST (STore)	**Formato**: ST X, Rr. Copia del contenido de Rd en la memoria de datos apuntado por el par de registros X, Y o Z: (X) ← Rr: • Rr puede ser cualquier registro (R0 ... R31). • X puede ser un registro doble X, Y o Z. Al igual que el caso anterior:



Resumen

Esta tabla resume las operaciones de movimiento de datos para el procesador Atmel, incluyendo el ATmega328p:

Registers	Data memory access		
Reg ←Const	Addressing ←mode:	Reg Mem	Mem ←Reg
LDI Rd, K(Rd=R16…R31)(K=0…255)	Direct addressing	LDS Rd, mem(mem = 0…65535)	STS mem, Rr(mem = 0…65535)
Reg ←Reg	Indirect addressing	LD Rd, X	ST X, Rr
MOV Rd, Rr		LD Rd, X+	ST X+, Rr
Ports (P=0…63)		LD Rd, -X	ST –X, Rr
IN Rd, P		Note: LD, ST also work with Y, Z regs	
OUT P, Rr	Indirect with displacement addressing (q=0…63)	LDD Rd, Y+q	Indirect with displacement addressing (q=0…63)

Donde Rd, Rr puede ser cualquiera de R0 ... R31 (excepto para la instrucción LDI).

¡Enhorabuena! Has finalizado con éxito.

Referencias bibliográficas

- Atmel. Disponible en: <http://www.atmel.com/products/microcontrollers/avr/default.aspx> [Consultado el 6 de junio de 2016].

- Atmel. Especificaciones de ATmega48A/PA/88A/PA/168A/PA/328/P. Disponible en: <http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf> [Consultado el 6 de junio de 2016].

