

1. Introducción a los Microcontroladores

En este capítulo se da una introducción al tema, exponiendo conceptos generales, es decir, conceptos que no están enfocados a un MCU particular. Se describen los alcances y limitaciones de estos dispositivos y se muestra una organización común a la mayoría de microcontroladores.

1.1 Sistemas Electrónicos

La electrónica ha evolucionado de manera sorprendente en los últimos años, tanto que actualmente no es posible concebir la vida sin los sistemas electrónicos. Los sistemas electrónicos son una parte fundamental en el trabajo de las personas, proporcionan entretenimiento y facilitan las actividades en los hogares.

Un sistema electrónico puede ser representado con el diagrama de la figura 1.1, sin importar la funcionalidad para la cual haya sido diseñado.

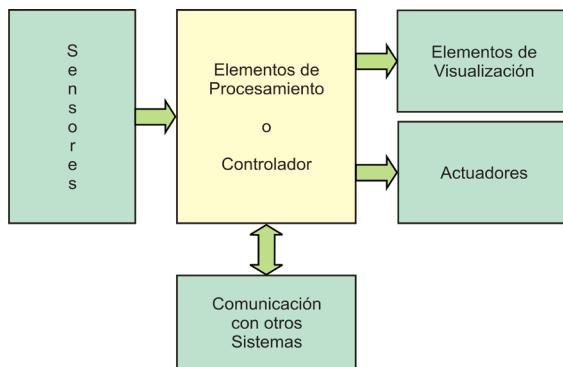


Figura 1.1 Abstracción de un sistema electrónico

El sistema recibe las peticiones de los usuarios o conoce lo que ocurre en su entorno por medio de los sensores. Los sensores son dispositivos electrónicos que se encargan de acondicionar diferentes tipos de información a un formato reconocido por los elementos de procesamiento. Un sensor puede ser tan simple como un botón o tan complejo como un reconocedor de huella digital, pero si los elementos de procesamiento son digitales, en ambos casos la salida va a estar codificada en 1's y 0's. Con los sensores se pueden monitorear diferentes parámetros, como: temperatura, humedad, velocidad, intensidad luminosa, etc.

Los elementos de visualización son dispositivos electrónicos que muestran el estado actual del sistema, notificando al usuario si debe tomar acciones. Los elementos de visualización típicos son: LEDs individuales o matrices de LEDs, displays de 7 segmentos o de cristal líquido.

Los actuadores son dispositivos electrónicos o electromecánicos que también forman parte de las salidas de un sistema, pero con la capacidad de modificar el entorno, es decir, van más allá de la visualización, algunos ejemplos son: motores, electroválvulas, relevadores, etc.

Los elementos de comunicación proporcionan a un sistema la capacidad de comunicarse con otros sistemas, son necesarios cuando una tarea compleja va a ser resuelta por diferentes sistemas. Entonces, un sistema complejo está compuesto por diferentes sistemas simples, cada uno con sus elementos de procesamiento, cada sistema simple o sub-sistema está orientado a resolver una etapa de la tarea compleja.

Los elementos de procesamiento son dispositivos electrónicos que determinan la funcionalidad del sistema, con el desarrollo de uno o varios procesos. Ocasionalmente a estos elementos de procesamiento se les refieren como la Tarjeta de Control de un sistema o simplemente el Controlador. El controlador recibe la información proveniente de los sensores y, considerando el estado actual que guarda el sistema, genera algunos resultados visuales, activa algún actuador o notifica sobre nuevas condiciones a otro sistema.

1.2 Controladores y Microcontroladores

El concepto de controlador ha permanecido invariable a través del tiempo, aunque su implementación física ha variado con los cambios tecnológicos. En principio, los controladores se construyeron con base en circuitos analógicos, las decisiones se tomaban con diferentes configuraciones de transistores o amplificadores operacionales. En los setentas se empleaba lógica discreta con circuitos digitales con baja o mediana escala de integración.

El primer microprocesador (4004 de Intel) fue puesto en operación en 1971, esto dio lugar al empleo de un microprocesador con sus elementos de soporte (memoria, entrada/salida, etc.) como tarjetas de control. A estas tarjetas también se les conoce como Computadoras en una Sola Tarjeta (SBC, *single board computer*). Actualmente se han integrado todos estos elementos en un solo circuito integrado y a éste se le refiere como Unidad Micro Controladora (MCU, *Micro Controller Unit*) o simplemente microcontrolador, esta tendencia se ilustra en la figura 1.2.

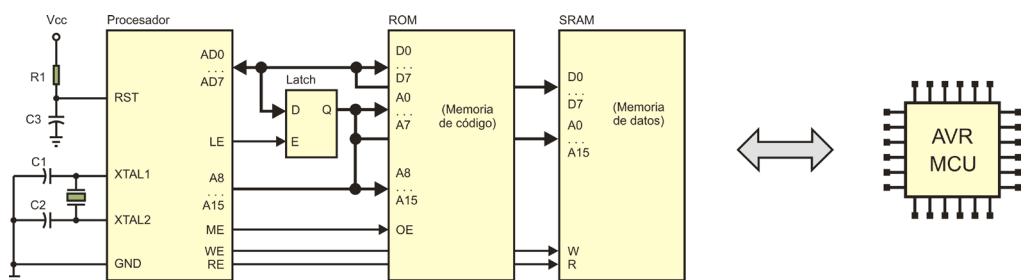


Figura 1.2 Los microcontroladores remplazan a tarjetas con varios CI

Un microcontrolador es un Circuito Integrado con una escala de integración muy grande (VLSI¹, *very large scale integration*) que internamente contiene una Unidad Central de Procesamiento (CPU, *Cental Processing Unit*), memoria para código, memoria para datos, temporizadores, fuentes de interrupción y otros recursos necesarios para el desarrollo de aplicaciones, por lo general con un propósito específico.

Si bien, un MCU incluye prácticamente los elementos necesarios para ser considerado como una computadora en un circuito integrado, frecuentemente no es tratado como tal, ya que su uso típico consiste en el desempeño de funciones de “control” interactuando con el “mundo real” para monitorear condiciones (a través de sensores) y en respuesta a ello, encender o apagar dispositivos (por medio de actuadores).

1.3 Microprocesadores y Microcontroladores

Ocasionalmente estos dispositivos se tratan como iguales, sin embargo existen diferencias fundamentales a considerar.

Un microprocesador básicamente contiene una CPU, mientras que un microcontrolador además de la CPU contiene memoria, temporizadores, interrupciones y otros recursos útiles para el desarrollo de aplicaciones, todos estos elementos en un circuito integrado.

El microcontrolador tiene más recursos que el microprocesador, pero su CPU está limitada en términos de su capacidad de procesamiento. Las limitaciones principales son:

- **Velocidad de procesamiento:** Actualmente los microcontroladores trabajan a frecuencias máximas de 20 MHz, mientras que los microprocesadores están en el orden de GHz.
- **Capacidad de direccionamiento:** Un microcontrolador promedio dispone de 8 Kbyte para instrucciones y 1 Kbyte para datos, los microprocesadores modernos pueden direccionar hasta 1 Terabyte, espacio compartido para instrucciones y datos. Por lo que en su repertorio de instrucciones, los microprocesadores deben incluir modos de direccionamiento que les permitan este alcance.
- **Tamaño de los datos:** Los microcontroladores populares son de 8 bits y dentro de sus instrucciones incluyen algunas que permiten evaluar o modificar bits individuales. Los microprocesadores actuales trabajan con datos de 32 ó 64 bits. Sus instrucciones operan directamente sobre palabras de esta magnitud y generalmente no cuentan con instrucciones dedicadas a bits.

Estas notables diferencias entre microprocesadores y microcontroladores los enfocan a diferentes aplicaciones. Un microprocesador se utiliza como la CPU de una

1 Circuitos integrados con más de 10,000 transistores

computadora, una computadora es un **Sistema de Propósito General**, es decir, un sistema de procesamiento intensivo capaz de realizar cualquier tarea que se le solicite por programación.

Los microcontroladores están enfocados a **Sistemas de Propósito Específico**, sistemas que se crean con una funcionalidad única, la cual no va a cambiar durante su tiempo de vida útil. Por ejemplo: cajas registradoras, hornos de microondas, sistemas de control de tráfico, videojuegos, equipos de sonido, instrumentos musicales, máquinas de escribir, fotocopiadoras, etc.

Las limitaciones de los microcontroladores con respecto a los microprocesadores no son una restricción para este tipo de aplicaciones, si se consideran los siguientes aspectos:

- El tiempo de respuesta en una aplicación de propósito específico no es crítico, las operaciones para monitorear parámetros o actualizar resultados requieren de períodos en el orden de cientos de microsegundos o milisegundos, períodos que pueden conseguirse con un microcontrolador operando a unos cuantos mega hertzios.
- La aplicación es única, eso significa que la memoria de código no debe alojar otros programas que nada tengan que ver con la aplicación, como un cargador o un sistema operativo, lo cual es fundamental en un sistema de propósito general. Por lo tanto, la capacidad de memoria incluida en los microcontroladores llega a ser suficiente. Por otro lado, también hay microcontroladores con diferentes capacidades de memoria de código, que van desde 1 Kbyte hasta 256 Kbyte, el desarrollador de sistemas puede seleccionar el modelo que mejor se ajuste a sus requerimientos.
- Las aplicaciones por lo general utilizan pocas entradas, algunas son directamente de 1 bit y otras pueden ser agrupadas en un puerto de 8 bits, para su procesamiento es suficiente con una CPU que trabaje por bytes. De manera poco frecuente estas aplicaciones requieren datos de 16 bits, por ello, algunos microcontroladores incluyen instrucciones que operan directamente sobre 16 bits, o bien, puede buscarse un microcontrolador con una CPU de 16 bits. Para las salidas, es muy común que se requiera la manipulación directa de 1 bit. El encendido o apagado de un motor, un elevador, una lámpara, etc., no requiere más de 1 bit. Si fuera necesario algún tipo de variación en la intensidad de la salida, puede utilizarse modulación por ancho de pulso (*PWM, pulse width modulation*).

Puede observarse que un MCU efectivamente contiene los elementos suficientes para ser considerado como una computadora en un CI. Aunque sería una computadora con una capacidad de procesamiento limitada. No obstante, los recursos incluidos en un MCU son suficientes para aplicaciones de propósito específico, que no demanden un alto rendimiento y que no requieran manejar un conjunto masivo de datos. Aplicaciones como procesamiento de imágenes o video, están fuera del alcance de un microcontrolador.

1.4 FPGAs y Microcontroladores

Los FPGAs son dispositivos electrónicos programables que también pueden emplearse como elementos de procesamiento en sistemas electrónicos. La sigla FPGA (*Field Programmable Gate Array*) hace referencia a un Arreglo de Compuertas Programable en Campo. En la figura 1.3 se muestra la organización general de un FPGA, en donde puede notarse una disposición matricial de Bloques Lógicos Configurables (CLB, *Configurable Logic Block*) rodeados por Bloques de Entrada/ Salida (IOB, *Input/Output Block*), además de los recursos necesarios para la conexión de CLBs con IOBs o entre CLBs.

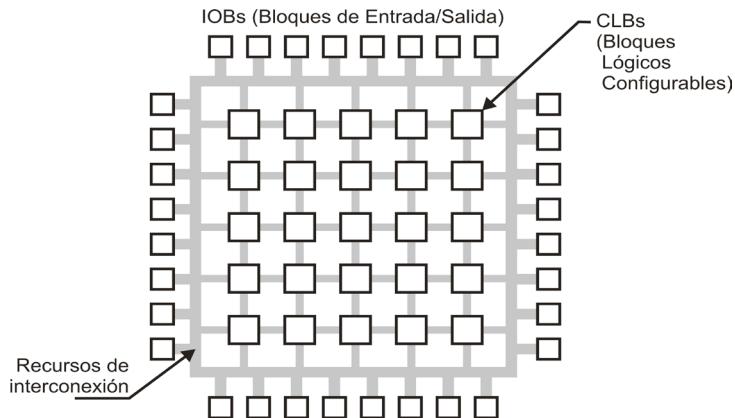


Figura 1.3 Organización típica de un FPGA

En los CLBs se pueden programar funciones lógicas combinacionales o secuenciales, con los recursos de interconexión es posible vincular diferentes bloques para construir funciones más complejas. Dependiendo del fabricante, un CLB puede contener una tabla de búsqueda (LUT, *look-up table*) o un arreglo de compuertas básicas más elementos de estado. Los IOBs proporcionan el mecanismo para que el FPGA se comunique con su entorno.

Los FPGAs se pueden programar por medio de diagramas esquemáticos, utilizando símbolos básicos y conexiones entre estos símbolos. No obstante, por la alta densidad de los dispositivos actuales, es mejor emplear un Lenguaje de Descripción de Hardware (HDL, *Hardware Description Language*). Existen diferentes HDLs, como VHDL, Verilog o ABEL.

Un aspecto común entre FPGAs y MCUs es que ambos son dispositivos configurables, con ambos se pueden construir sistemas flexibles, cuyo comportamiento se puede alterar al reprogramar al dispositivo. Con todo, debe distinguirse el papel del programa en cada caso, en un FPGA el programa determina cómo se van a conectar sus elementos internos, es decir, el programa define al hardware y de esta manera determina el comportamiento del sistema. En cambio, en un MCU el hardware es fijo y el programa establece la operación de ese hardware.

La organización de los FPGAs hace que el proceso de desarrollo de un sistema sea más complejo y tardado, con respecto al uso de microcontroladores. La ventaja de su uso es que la tecnología actual empleada en su fabricación y el hecho de trabajar directamente en hardware hacen que se alcance una velocidad de procesamiento muy alta (100 MHz o más) en relación a la velocidad de un MCU promedio.

Otra ventaja es que en un FPGA puede hacerse procesamiento concurrente real, si un sistema está organizado en forma modular, los módulos van a revisar sus entradas para generar sus salidas en forma concurrente. En un MCU el procesamiento es secuencial, aunque la inclusión de múltiples recursos facilita la realización simultánea de tareas, en el momento en que éstas generan un evento que requiere atención, la atención se realiza mediante líneas de código secuenciales.

En forma práctica, siempre debería intentarse emplear un MCU como el controlador de un sistema electrónico, si se requiere de más velocidad o capacidad de direccionamiento, la alternativa sería un microprocesador con sus elementos de soporte. Si se va a hacer un procesamiento aritmético intensivo, podría optarse por un procesador digital de señales (DSP, *Digital Signal Processor*), el cual es un circuito integrado que contiene un microprocesador más elementos de hardware enfocados a operaciones aritméticas, como sumadores y multiplicadores. Y sólo en aquellos casos donde se requiera de un hardware especializado, a la medida del sistema, que trabaje a altas velocidades y con módulos concurrentes, la mejor opción es el uso de un FPGA.

1.5 Organización de los Microcontroladores

Existe una gama muy amplia de fabricantes de microcontroladores y cada fabricante maneja diferentes familias con una variedad de modelos, a pesar de ello, hay bloques que son comunes a la mayoría de modelos, en la figura 1.4 se muestra la organización típica de un microcontrolador y en los siguientes apartados se describen sus bloques internos.

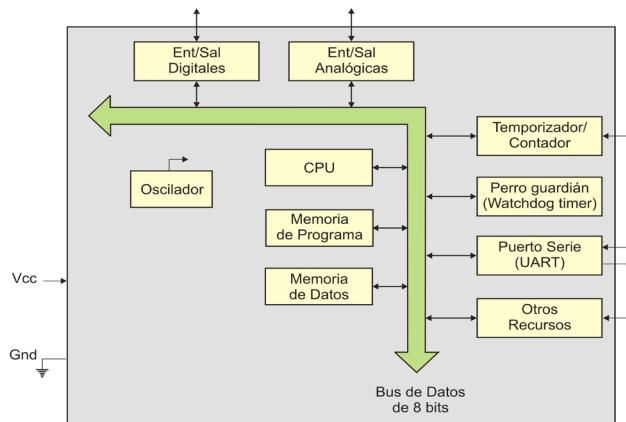


Figura 1.4 Organización típica de un Microcontrolador

1.5.1 La Unidad Central de Procesamiento (CPU)

Este bloque administra todas las actividades en el sistema y ejecuta todas las operaciones sobre los datos. Esto mediante la ejecución de las instrucciones ubicadas en la memoria de código, con las cuales se determina el comportamiento del sistema. Un **programa** se define como una serie de instrucciones, combinada para realizar algún trabajo específico. El grado en el cual los trabajos son realizados eficiente y correctamente depende muchas veces del software y no de qué tan sofisticada es la CPU.

El trabajo de la CPU puede resumirse en tres tareas fundamentales:

- a) Captura de una Instrucción.
- b) Decodificación de la misma.
- c) Ejecución.

Trabajo que realiza a altas velocidades, por lo que el usuario observa el efecto de un programa completo y no de instrucciones individuales.

Cada procesador tiene su propio repertorio de instrucciones. Si un grupo de computadoras o microcontroladores comparten el mismo repertorio de instrucciones, pero los elementos del grupo difieren en recursos, costo y rendimiento, entonces este grupo forma una familia de computadoras.

Los repertorios de instrucciones difieren entre microcontroladores o microprocesadores, no obstante, existen algunos grupos de instrucciones que son comunes a la mayoría de este tipo de dispositivos. Estos grupos incluyen:

- a) Aritméticas: suma, resta, producto, división, etc.
- b) Lógicas: AND, OR, NOT, etc.
- c) Transferencias de datos.
- d) Bifurcaciones o saltos (condicionales o incondicionales).

Una computadora es un sistema originalmente planeado para procesamiento de datos, por lo que podría pensarse que las instrucciones de mayor uso son aritméticas o lógicas, sin embargo, actualmente las computadoras han ampliado tanto su campo de acción que las aplicaciones comunes hacen un uso exhaustivo de transferencias de datos. El ejemplo típico es el procesador de palabras, el cual transfiere datos del espacio disponible para entradas y salidas, a memoria principal y a memoria de video, cuando se respalda un documento, la información es transferida de memoria principal a memoria secundaria.

Una instrucción es una cadena de 1's y 0's que la computadora reconoce e interpreta, en esa cadena existen diferentes grupos de bits que se conocen como campos de la instrucción. Una instrucción incluye un campo para el **código de operación (opcode)**, éste determina la operación a realizar, y típicamente uno o dos campos para los operandos, que corresponden a los datos sobre los cuales se aplica la operación.

Las CPUs se clasifican como CISC o RISC, esto de acuerdo con su organización interna. Con CISC se hace referencia a computadoras con un Repertorio de Instrucciones Complejo (CISC, *Complex Instruction Set Computers*) y RISC es para referir a computadoras con un Repertorio de Instrucciones Reducido (RISC, *Reduced Instruction Set Computers*).

Las primeras computadoras se construyeron con la filosofía CISC, en la cual se buscaba que el programador escribiera programas compactos, por lo tanto, cada instrucción requería de un hardware complejo. Esto afecta el rendimiento de las computadoras dado que se requiere de un ciclo de reloj duradero o de varios ciclos de reloj para la ejecución de una instrucción. La filosofía RISC es opuesta, busca que el hardware sea simple y que resuelva pocas instrucciones, con ello el hardware puede trabajar a frecuencias mayores.

Una arquitectura RISC tiene pocas instrucciones y generalmente éstas son del mismo tamaño; en la CISC hay demasiadas instrucciones con diferentes tamaños y formatos, que pueden ocupar varios bytes, uno para el opcode y los demás para los operandos. La tarea realizada por una instrucción CISC puede requerir de varias instrucciones RISC. En contraste, el hardware de un procesador RISC es tan simple, que se puede implementar en una fracción de la superficie ocupada en un circuito integrado por un procesador CISC.

La organización de los procesadores RISC hace que, aun con tecnologías de semiconductores comparables e igual frecuencia de reloj, su capacidad de procesamiento sea de dos a cuatro veces mayor que la de un CISC, esto porque permite la aplicación de técnicas como la segmentación, mediante la cual es posible solapar diferentes instrucciones en diferentes etapas del procesador, por ejemplo, mientras una instrucción se está ejecutando, otra puede estar en proceso de decodificación y la siguiente en la etapa de captura. El número de instrucciones que simultáneamente están en el procesador depende del número de etapas de segmentación incluidas.

1.5.1.1 Organización de una CPU

A pesar de que existe una diversidad de fabricantes de procesadores, hay elementos que son comunes a todos ellos. En la figura 1.5 se muestran los bloques típicos de una CPU, los cuales se pueden clasificar en dos grupos: el Camino de Datos y la Unidad de Decodificación y Control. El Camino de Datos involucra los elementos en donde puede fluir la información cuando se ejecuta una instrucción y la Unidad de Decodificación y Control determina qué elementos se activan dentro del Camino de los Datos para la correcta ejecución de una instrucción.

El Contador de Programa (PC, *Program Counter*), el Registro de Instrucción (IR, *Instruction Register*) y el Apuntador de Pila (SP, *Stack Pointer*), son registros con una función específica en una CPU.

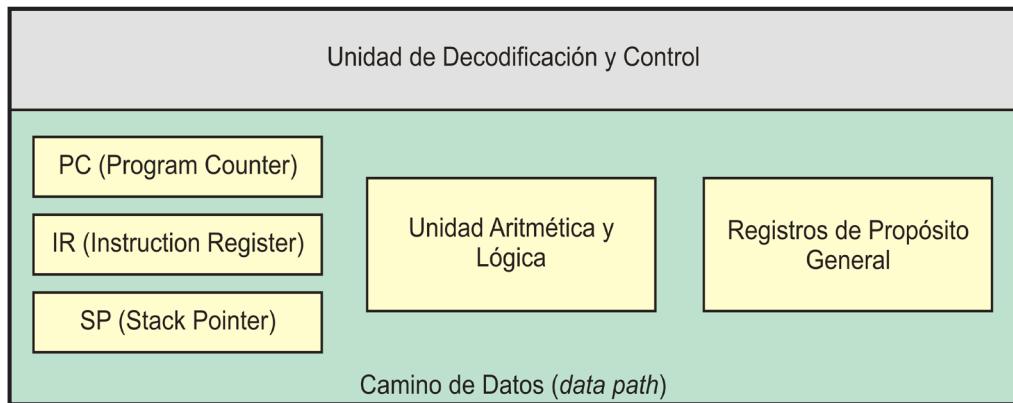


Figura 1.5 Elementos comunes en una CPU

El PC contiene la dirección de la instrucción que se va a ejecutar en un instante de tiempo determinado y mientras esa instrucción se ejecuta, el PC automáticamente actualiza su valor para apuntar a la siguiente instrucción a ejecutar.

El registro IR contiene la cadena de bits que conforman a la instrucción bajo ejecución, de esa cadena, la unidad de control considera el campo del opcode para determinar la activación de las señales en los demás elementos en la CPU.

El SP contiene la dirección del tope de la pila, que es un espacio de almacenamiento utilizado durante la invocación de rutinas. La llamada a una rutina requiere que el valor del PC sea respaldado en la pila, con ello, el SP se ajusta automáticamente al nuevo tope. Cuando la rutina termina, se extrae el valor del tope de la pila y con éste se reemplaza al PC, para que el programa continúe con la instrucción posterior al llamado de la rutina, esto también requiere un ajuste del SP. Además de las llamadas a rutinas, algunos procesadores incluyen instrucciones para hacer respaldos (*push*) y recuperaciones (*pop*) en forma explícita. Instrucciones que también producen cambios automáticos en el SP.

La Unidad Aritmética y Lógica es el bloque que se encarga de realizar las operaciones aritméticas y lógicas con los datos, no obstante, en ocasiones también opera sobre direcciones para calcular el destino de un salto o la ubicación de una localidad a la que se va a tener acceso para una transferencia de memoria a registro o viceversa.

Los registros de propósito general son los elementos más rápidos para el almacenamiento de variables. Dado que el número de registros en una CPU es limitado, si éste no es suficiente para todas las variables requeridas, debe utilizarse la memoria de datos para su almacenamiento.

1.5.1.2 Tareas de la CPU

Con cada instrucción, la CPU realiza tres tareas fundamentales: Captura, Decodificación y Ejecución.

La **Captura de una Instrucción** es una tarea que involucra los siguientes pasos:

- a. El contenido del PC se coloca en el bus de direcciones.
- b. La CPU genera una señal de control, para habilitar la lectura de memoria de código.
- c. Una instrucción se lee de la memoria de código y se coloca en el bus de datos.
- d. La instrucción se toma del bus de datos y se coloca en el IR.
- e. El PC es preparado para la siguiente instrucción.

Una vez que la instrucción está en el IR, el procesador continúa con la **decodificación** de la instrucción. Decodificar una instrucción consiste en descifrar el opcode para generar las señales de control necesarias, dependiendo del tipo de instrucción.

Finalmente, la tercera de las tareas de la CPU es la **Ejecución**. Ejecutar una instrucción puede involucrar: habilitar a la ALU para que genere algún resultado, cargar un dato de memoria a un registro, almacenar el contenido de un registro en memoria o modificar el valor del PC, según las señales generadas por la unidad de decodificación y control.

1.5.2 Sistema de Memoria

Una computadora (y por lo tanto, también un microcontrolador) debe contar con espacios de memoria para almacenar los programas (código) y los datos. En relación a cómo se organizan estos espacios se tienen dos modelos de computadoras, un modelo en donde el código y datos comparten el mismo espacio de memoria y el otro en donde se tienen memorias separadas, una para código y otra para datos, éstos se ilustran en la figura 1.6.

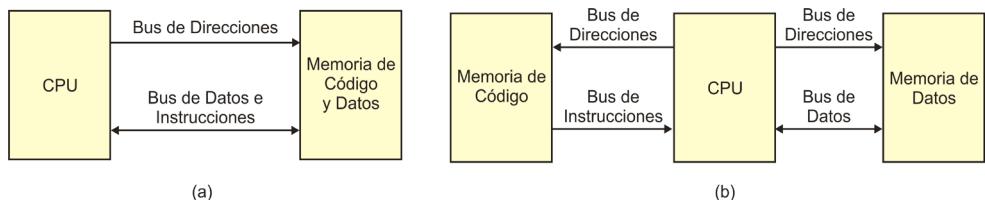


Figura 1.6 Modelos de computadoras respecto a la organización de la memoria (a) Arquitectura von Neumann y (b) Arquitectura Harvard

John von Neumann² propuso el concepto de programa almacenado, el cual establece que las instrucciones se lleven a memoria como si fueran datos, para que posteriormente se ejecuten sin tener que escribirlas nuevamente, por lo tanto, se requiere de un solo espacio de memoria para almacenar instrucciones y datos. Este concepto fue primeramente aplicado en la Computadora Automática Electrónica de Variable Discreta (EDVAC, *Electronic Discrete-Variable Automatic Computer*), desarrollada por Von Neumann, Eckert y Mauchly. Actualmente ha sido adoptado por los diseñadores de computadoras porque proporciona flexibilidad a los sistemas. Si una computadora se basa en este concepto, se dice que tiene una Arquitectura tipo Von Neumann.

Mientras la tendencia natural para los diseñadores de computadoras fue adoptar el concepto de programa almacenado, en la Universidad de Harvard desarrollaron la Mark I, la cual almacenaba instrucciones y datos en cintas perforadas, pero incluía interruptores rotatorios de 10 posiciones para el manejo de registros. Actualmente, si una computadora tiene un espacio para el almacenamiento de código físicamente separado del espacio de almacenamiento de datos, se dice que tiene una Arquitectura Harvard.

La mayoría de microcontroladores utilizan una Arquitectura Harvard. En la memoria de código se alojan las instrucciones que conforman el programa y algunas constantes. Algunos microcontroladores, además de su memoria interna, tienen la capacidad de direccionar memoria externa de código, para soportar programas con una cantidad grande de instrucciones.

Usualmente la memoria de programa es no volátil y suele ser del tipo EPROM, EEPROM, Flash, programable una sola vez (OTP, *one-time programmable*) o ROM enmascarable. Los primeros 3 tipos son adecuados durante las etapas de prototipado, la memoria OTP es conveniente si se va a hacer una producción de pocos volúmenes de un sistema y la ROM enmascarable es lo más acertado para una producción masiva.

Para la memoria de datos, los microcontroladores pueden contener RAM o EEPROM, la RAM se utiliza para almacenar algunas variables y contener una pila. La EEPROM es para almacenar aquellos datos que se quieran conservar aun en ausencia de energía. Todos los microcontroladores tienen memoria interna de datos, en diferentes magnitudes, algunos además cuentan con la capacidad de expansión usando una memoria externa.

1.5.3 Oscilador

La CPU va tomando las instrucciones de la memoria de programa para su posterior ejecución a cierta frecuencia. Esta frecuencia está determinada por el circuito de oscilación, el cual genera la frecuencia de trabajo a partir de elementos externos como un circuito RC, un resonador cerámico o un cristal de cuarzo, aunque algunos

² John von Neumann, (28 de diciembre de 1903 - 8 de febrero de 1957) Matemático húngaro-estadounidense, doctorado por la Universidad de Budapest a los 23 años. Realizó contribuciones importantes en física cuántica, análisis funcional, teoría de conjuntos, informática, economía, análisis numérico, estadística y muchos otros campos.

microcontroladores ya incluyen un oscilador RC calibrado interno. Tan pronto como se suministra la alimentación eléctrica, el oscilador empieza con su operación.

1.5.4 Temporizador/Contador

El Temporizador/Contador (*timer/counter*) es un recurso con una doble función, como temporizador se utiliza para manejar intervalos de tiempo y como contador es la base para programar alguna tarea cada que ocurra una cantidad predeterminada de eventos externos al microcontrolador.

Se compone de un registro de n-bits que se incrementa en cada ciclo de reloj o cuando ocurra un evento externo, según el modo de operación. Cuando ocurre un desbordamiento del registro genera alguna señalización, poniendo en alto una bandera, para indicar a la CPU que ha pasado un intervalo de tiempo o que ha ocurrido un número esperado de eventos. El desbordamiento ocurre cuando el registro alcanza su valor máximo (todos los bits del registro en 1) y reinicia la cuenta (todos los bits en 0's). La organización básica de un temporizador se muestra en la figura 1.7.

El registro del temporizador tiene un comportamiento ascendente y puede ser pre-cargado para reducir el número de eventos a contar. La CPU puede emplear su tiempo de procesamiento en otras tareas, dentro de las cuales debe reservar un espacio para monitorear la bandera, o bien, configurar al recurso para que genere una interrupción.

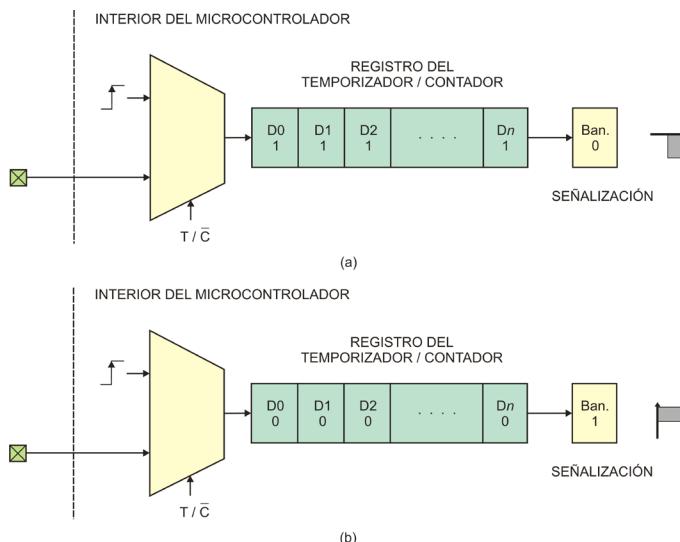


Figura 1.7 Organización básica de un Temporizador/Contador, en (a) el registro ha alcanzado su valor máximo y en (b) al reiniciar la cuenta se genera una señalización

En algunos microcontroladores la entrada del temporizador es precedida por un pre-escalador, el cual básicamente es un divisor de frecuencia configurable, con el que se puede contar un número más grande de eventos y por lo tanto, alcanzar intervalos de tiempo mayores.

En el caso de los microcontroladores AVR, además de los eventos de desbordamientos se pueden manejar eventos de coincidencias por comparación y de captura.

1.5.5 Perro Guardián (WDT, *watchdog timer*)

El WDT (*watchdog timer*) también es un temporizador y por lo tanto, también se compone de un registro de n-bits, sólo que cuando el WDT desborda ocasiona un reinicio del sistema (*reset*). El objetivo del WDT es evitar que el microcontrolador se cicle en estados no contemplados, lo cual llega a ser bastante útil en sistemas autónomos. Un microcontrolador puede ciclarse en estados no deseados ante situaciones inesperadas, como variaciones en la fuente de alimentación, desconexión repentina de un periférico, etc.

Algunos microcontroladores que poseen WDT requieren de su activación en el momento en que se programa al dispositivo, otros permiten activarlo o desactivarlo dentro del programa de aplicación, siempre que se siga alguna secuencia de seguridad para evitar activations no deseadas. Si se utiliza al WDT, en posiciones estratégicas del programa principal deben incluirse instrucciones que lo reinicien para evitar su desbordamiento.

1.5.6 Puerto Serie

La mayoría de microcontroladores cuentan con un receptor/transmisor universal asíncrono (UART, *Universal Asynchronous Receiver Transceiver*), para una comunicación serial con dispositivos o sistemas externos, bajo protocolos y razones de transmisiones estándares. La comunicación serial puede ser de dos tipos:

- Síncrona: Además de la línea de datos se utiliza una línea de reloj.
- Asíncrona: Sólo hay líneas para los datos, el transmisor y el receptor se deben configurar con la misma velocidad de transferencia (bits/segundo, *Baud Rate*), además de definir el mismo formato para cada trama.

La comunicación serial es bastante útil porque sólo requiere de un alambre o línea de conexión y tiene un alcance mucho mayor que una transmisión paralela (de varios bits). El hardware para la comunicación serial básicamente consiste en una conversión de paralelo a serie, durante una transmisión, o de serie a paralelo, cuando se hace una recepción. Puede realizarse entre un microcontrolador con una computadora, entre

microcontroladores o un microcontrolador con otros sistemas que incluyan un puerto de comunicación serial. En la figura 1.8 se muestra la diferencia entre una comunicación síncrona y asíncrona.

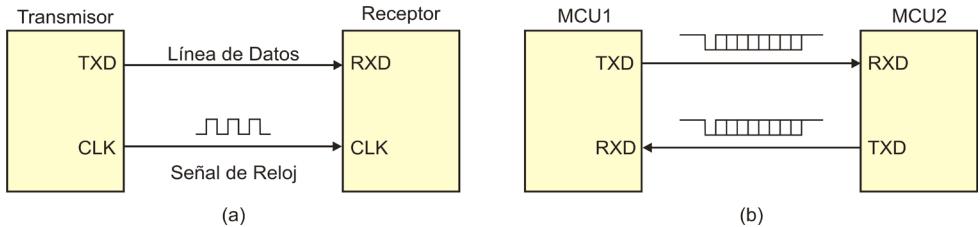


Figura 1.8 Comunicación Serial (a) Síncrona y (b) Asíncrona

1.5.7 Entradas/Salidas Digitales

Los microcontroladores incluyen puertos de Entradas/Salidas digitales para intercambiar datos con el mundo exterior. A diferencia de un puerto serie, en donde se transfiere un bit a la vez, en los puertos digitales es posible realizar un intercambio de bytes.

Todos los microcontroladores tienen puertos digitales, aunque el número de puertos o el número de bits por puerto puede variar entre dispositivos. Como entradas se utilizan para el monitoreo de dispositivos digitales como botones, interruptores, teclados, sensores con salida a relevador, etc., y como salidas para el manejo de LEDs, displays de 7 segmentos, activación de motores, LCDs, etc.

1.5.8 Entradas/Salidas Analógicas

Para entradas analógicas algunos microcontroladores incorporan Convertidores Analógico a Digital (ADC, *Analogic-Digital Converter*) o comparadores analógicos. Éstos son muy útiles porque sin elementos externos, permiten obtener información analógica del exterior, para monitorear parámetros como temperatura, velocidad, humedad, etc.

Para salidas analógicas podría pensarse en un Convertidor Digital a Analógico (DAC, *Digital-Analogic Converter*) pero no es común que se incluya en un microcontrolador. Para solventar esta carencia, algunos microcontroladores incluyen salidas con Modulación por Ancho de Pulso (PWM, *Pulse Width Modulation*), por medio de ellas, con pocos elementos externos es posible generar una señal analógica en una salida digital.

1.6 Clasificación de los Microcontroladores

Existen diferentes formas de clasificar a los microcontroladores y no son excluyentes unas de otras. A continuación se describen las formas típicas de clasificaciones.

Por la arquitectura de la CPU, los microcontroladores pueden clasificarse como **RISC** o **CISC**. Prácticamente todos los nuevos microcontroladores son **RISC**.

En relación al tamaño de los datos, se tienen microcontroladores de **4, 8, 16** y hasta **32** bits. Por el tamaño de los datos debe entenderse el tamaño de los registros de trabajo y por lo tanto, corresponde con el número de bits de los operandos en la ALU, éste generalmente difiere del tamaño de las instrucciones que no es un parámetro para la clasificación.

Tomando como base la organización y el acceso a la memoria de código y datos, se tienen 2 modelos: Arquitectura **Von Neumann** y Arquitectura **Harvard**.

Considerando la memoria y sus capacidades de expansión, cuando un microcontrolador está acondicionado para tener acceso a memoria externa, se dice que tiene una arquitectura **abierta**, en caso contrario, su arquitectura es **cerrada**. Con una arquitectura **abierta**, además de manejar memoria externa, es posible manipular periféricos externos, mapeándolos en memoria de datos y reservándoles un espacio de direcciones para su manejo.

La última clasificación tiene que ver con la forma en que los datos son almacenados y manipulados internamente dentro de la CPU. Los microcontroladores manipulan datos por medio de un programa de usuario, en este esquema de clasificación se distingue a las arquitecturas de acuerdo a como la CPU ejecuta las instrucciones y tiene acceso a los datos que involucra cada instrucción. Bajo este esquema, se tienen los siguientes cuatro modelos: **Pila**, **Acumulador**, **Registro-Memoria** y **Registro-Registro**.

En una arquitectura tipo **Pila**, una pila es la base para el procesamiento, los datos a operar deben ingresarse en la pila, las operaciones se realizan sobre los últimos datos de la pila y dejan el resultado en el tope de la pila. Por ejemplo, para realizar la operación de alto nivel:

$$A = B - C$$

Suponiendo que A, B y C son variables almacenadas en memoria, se tendrían las siguientes instrucciones:

```
PUSH  B      ; Ingresa la variable B en la pila
PUSH  C      ; Ingresa la variable C en la pila
SUB   ; Resta los datos del tope de la pila
      ; el resultado queda en la pila
POP   A      ; Extrae el tope de la pila y lo almacena en A
```

Una arquitectura tipo **Acumulador** basa su operación en un registro con el mismo nombre. El Acumulador es el registro de trabajo, las instrucciones que únicamente

requieren un operando se aplican sobre el acumulador y en la ALU, necesariamente un operando debe ser el acumulador.

Si la misma operación de resta se va a realizar bajo una arquitectura tipo acumulador, las instrucciones resultantes son las siguientes (Acc representa al acumulador):

```
MOV    Acc, B ; Transfiere la variable B al acumulador  
SUB    Acc, C ; Resta C del acumulador y ahí mismo queda el resultado  
MOV    A, Acc ; Transfiere el acumulador a la variable A
```

Una arquitectura del tipo **Registro-Memoria** implica que el procesador está acondicionado para que uno de los operandos de la ALU esté en memoria, mientras el otro debe estar en uno de los registros de trabajo. La operación bajo consideración se realizaría con las siguientes instrucciones:

```
LOAD  R1, B ; Carga la variable B en el registro R1  
SUB   R1, C ; Resta C que está en memoria, del registro R1  
MOV   A, R1 ; Almacena R1 en la variable A, que está en memoria
```

Finalmente, en una arquitectura del tipo **Registro-Registro**, los dos operandos que llegan a la ALU deben estar en los registros de propósito general. Las arquitecturas de este estilo también son conocidas como Arquitecturas tipo **Carga-Almacenamiento**, esto porque cuando se van a operar variables que están en memoria, primeramente deben ser cargadas en registros, el resultado queda en un registro y, por lo tanto, se requiere de un almacenamiento para llevarlo a una variable de memoria. Para el mismo ejemplo se tendrían las instrucciones siguientes:

```
LOAD  R1, B      ; Carga la variable B en un registro  
LOAD  R2, C      ; Carga la variable C en otro registro  
SUB   R1, R2     ; La ALU opera sobre registros  
MOV   A, R1      ; Almacena el resultado en la variable A
```

Los microcontroladores bajo estudio son el ATMega8 y el ATMega16, estos microcontroladores son RISC, de 8 bits, con una Arquitectura tipo Harvard que es cerrada, y su operación es del tipo Registro-Registro.

1.7 Criterios para la Selección de los Elementos de Procesamiento

Existe una gama muy amplia de fabricantes de microprocesadores o microcontroladores, cada fabricante ha desarrollado diferentes familias y en cada familia se tiene un número variable de dispositivos, es por esto que resulta complejo determinar cuál sería el dispositivo adecuado para alguna aplicación. A continuación se listan algunos criterios que pueden tomarse en consideración.

La primera consideración son las **prestaciones** del dispositivo, las cuales se deben vincular con los requerimientos de procesamiento que debe realizar el sistema.

Considerando la capacidad de procesamiento, los dispositivos se pueden agrupar en 3 clases diferentes:

- **Gama baja:** Procesadores de 4, 8 y 16 bits. Dedicados fundamentalmente a tareas de control (electrodomésticos, cabinas telefónicas, tarjetas inteligentes, algunos periféricos de computadoras, etc.). Generalmente se emplean microcontroladores.
- **Gama media:** Dispositivos de 16 y 32 bits. Para tareas de control con cierto grado de procesamiento (control en automóvil, teléfonos móviles, PDA, etc.). En este caso puede utilizarse un microcontrolador o microprocesador, además de periféricos y memoria externa.
- **Gama alta:** 32, 64 y 128 bits. Fundamentalmente para procesamiento (computadoras, videoconsolas, etc.). Casi en su totalidad son microprocesadores más circuitería periférica y memoria.

Referente a la **tecnología** de fabricación, debe considerarse:

- El **consumo de energía**, algunos dispositivos cuentan con modos de ahorro de energía que les permiten un consumo de algunos micro-Watts, mientras que otros llegan a consumir algunas décimas de Watts.
- Otro aspecto es el **voltaje de alimentación**, algunos dispositivos puede operar con 5 V, 3.3 V, 2.5 V o 1.5 V, éste es fundamental si el sistema va a ser alimentado con baterías.
- La **frecuencia** de operación también es un factor bajo consideración, dado que los dispositivos pueden operar desde KHz a GHz, si un microcontrolador puede trabajar en un rango amplio de frecuencias, es conveniente operarlo en la frecuencia más baja que le permita un desempeño correcto en la aplicación, esto porque a menor velocidad de procesamiento, el consumo de energía es menor.

El siguiente criterio bajo consideración es el **costo**, este aspecto es esencial una vez que se ha comprobado que el dispositivo cumple con las prestaciones requeridas, es decir, después de un análisis del rendimiento del hardware y software, considerando el uso medio o el peor de los casos. El costo de un microcontrolador o microprocesador puede variar de 2 a 1000 dólares.

Un aspecto muy importante son las **herramientas de desarrollo**, debe considerarse su precio, complejidad y prestaciones. Actualmente muchos fabricantes de microcontroladores dejan disponible de manera gratuita alguna suite de desarrollo, buscando ponerse a la vanguardia entre los desarrolladores de sistemas.

Un factor importante es la **experiencia** del desarrollador, muchas veces se prefiere acondicionar un microcontrolador conocido para incluir un recurso externo, antes de aprender a manejar un nuevo dispositivo que ya tiene al recurso empotrado.

Una vez que se domina un microcontrolador no es complejo manejar uno diferente, por lo que este hecho puede deberse a la carencia de dispositivos o herramientas, o bien a

la ausencia de soporte técnico. Si no se tienen problemas de disponibilidad y soporte, la emigración a dispositivos con un mayor número de recursos es lo más adecuado, dado que con una sola compuerta externa que se ahorre en un sistema, puede representar grandes beneficios económicos si se considera una producción masiva.

La experiencia en el manejo de un dispositivo va a reflejarse en el **tiempo de desarrollo** de un producto, la rápida evolución de la tecnología requiere de tiempos de desarrollo cada vez más cortos para mantener competitividad. Por ejemplo, si se desea desarrollar un decodificador para un receptor satelital con base en un microprocesador y se invierte un tiempo aproximado de dos años, para cuando el producto sea puesto en el mercado, tal vez existan versiones de procesadores que trabajen al doble de velocidad y que consuman la mitad de la potencia, eso implicaría que el producto ya no sería competitivo. Por lo tanto, los retrasos de la puesta en el mercado de los nuevos productos pueden producir grandes pérdidas.

El último criterio bajo consideración es la **compatibilidad** entre los dispositivos de una misma familia, éste es importante cuando se proyecta el desarrollo de diferentes versiones de un producto. En una familia, todos los dispositivos manejan el mismo repertorio de instrucciones, pero se distinguen por los recursos de hardware incluidos en cada miembro, por lo que el desarrollador debe seleccionar el más adecuado ante estas diferentes versiones del producto, las cuales pueden ir desde dispositivos con características muy limitadas hasta las versiones altamente sofisticadas.

La **compatibilidad** implica que se requiera de pocos ajustes en hardware y software, para obtener una versión mejorada de un producto, empleando un microcontrolador con mayores prestaciones. Este factor es importante si se toma en cuenta que la vida media de los productos es cada vez más corta, actualmente se llega a considerar como obsoleto a un sistema después que ha trabajado un par de años. Esto resalta la conveniencia de utilizar microcontroladores que pertenecen a familias con una gama amplia de dispositivos.

1.8 Ejercicios

1. Explique la importancia de los sistemas electrónicos.
2. ¿Qué es un microcontrolador?
3. Exprese las diferencias principales entre un microcontrolador y un microprocesador.
4. Explique a qué tipo de aplicaciones se enfocan los microcontroladores y dé ejemplos de ellas.
5. Justifique en qué situaciones sería conveniente o necesario el uso de un FPGA en lugar de un MCU.
6. Muestre un diagrama con la organización típica de un microcontrolador.
7. Describa el papel de una CPU en un microcontrolador (o computadora) y explique las tareas que realiza con cada instrucción.

8. Indique el objetivo de los registros de propósito específico comúnmente encontrados en una CPU:
 - a. *Program Counter* (PC)
 - b. *Instruction Register* (IR)
 - c. *Stack Pointer* (SP)
 9. Liste los grupos de instrucciones típicos que maneja una CPU.
 10. En qué difiere una arquitectura Harvard de una arquitectura basada en el modelo de Neumann.
 11. Explique las diferencias entre una arquitectura RISC y una arquitectura CISC.
 12. Indique los tipos de memoria utilizados por los microcontroladores para el almacenamiento de instrucciones y para el almacenamiento de datos.
 13. Explique la función de los siguientes recursos en un microcontrolador:
 - a. Oscilador Interno
 - b. Temporizador (*timer*)
 - c. Perro guardián (*watchdog timer*)
 - d. Puerto Serie
 - e. Entradas y salidas digitales
 - f. Entradas y salidas analógicas
 14. Muestre cómo se realizaría la suma: $A = B + C + D$, en una arquitectura:
 - a. Tipo Pila
 - b. Tipo Acumulador
 - c. Tipo Memoria-Registro
 - d. Tipo Registro-Registro (Carga-Almacenamiento)
- Suponiendo que A, B, C y D son variables ubicadas en memoria de datos.
15. En orden de consideración, explique tres criterios que tomaría en cuenta al seleccionar un microcontrolador para una aplicación.

2. Organización de los Micrcontroladores AVR de ATMEL

Los microcontroladores AVR incluyen un procesador **RISC** de **8 bits**, su arquitectura es del tipo **Harvard** y sus operaciones se realizan bajo un esquema **Registro-Registro**.

Este capítulo hace referencia al hardware de los microcontroladores AVR, específicamente del ATMega8 y ATMega16, se describe su organización interna y sus características de funcionamiento.

2.1 Características Generales

Los microcontroladores AVR se basan en un núcleo cuya arquitectura fue diseñada por Alf-Egil Bogen y Vegard Wollan, estudiantes del Instituto Noruego de Tecnología, arquitectura que posteriormente fue refinada y desarrollada por la firma Atmel. El término AVR no tiene un significado implícito, a veces se considera como un acrónimo en el que se involucra a los diseñadores del núcleo, es decir AVR puede corresponder con Alf-Vegard-RISC.

El núcleo es compartido por más de 50 miembros de la familia, proporcionando una amplia escalabilidad entre elementos con diferentes recursos. En la figura 2.1 se ilustra este hecho, los miembros con menos recursos caen en la gama Tiny, los miembros con más recursos pertenecen a la categoría Mega, además de que se cuenta con miembros orientados para aplicaciones específicas.

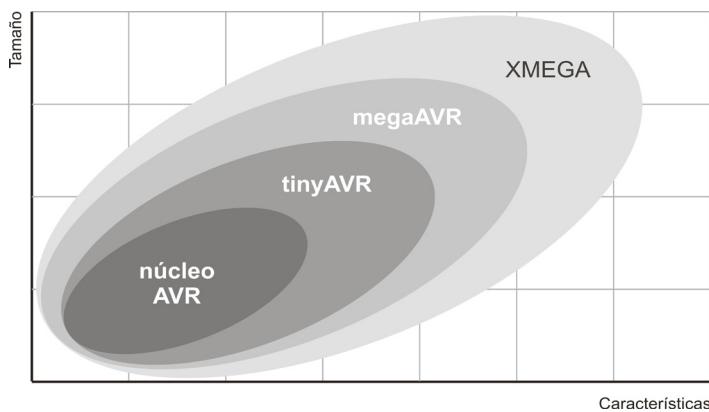


Figura 2.1 Escalabilidad entre dispositivos que comparten el núcleo

En concreto, este libro se enfoca a los dispositivos **ATMega8** y **ATMega16**, para ambos, sus principales características técnicas son:

- **Memoria de código:** 8 Kbyte (ATMega8) o 16 Kbyte (ATMega16) de memoria flash.
- **Memoria de datos:** 1 Kbyte de SRAM y 512 bytes de EEPROM.
- **Terminales para entrada/salida:** 23 (ATMega8) o 32 (ATMega16).
- **Frecuencia máxima de trabajo:** 16 MHz.
- **Voltaje de alimentación:** de 2.7 a 5.5 Volts.
- **Temporizadores:** 2 de 8 bits y 1 de 16 bits.
- **Canales PWM:** 3 (ATMega8) o 4 (ATMega16).
- **Fuentes de interrupción:** 19 (ATMega8) o 21 (ATMega16).
- **Interrupciones externas:** 2 (ATMega8) o 3 (ATMega16).
- **Canales de conversión Analógico/Digital:** 8 de 10 bits.
- Reloj de tiempo real.
- Interfaz SPI Maestro/Esclavo.
- Transmisor/Receptor Universal Síncrono/Asíncrono (USART).
- Interfaz serial de dos hilos.
- Programación “*In System*”.
- Oscilador interno configurable.
- *Watchdog timer*.

Comercialmente el ATMega8 se encuentra disponible en encapsulados PDIP de 28 terminales o bien, encapsulados TQFP o MLF de 32 terminales. Para el ATMega16 se tiene una versión en PDIP de 40 terminales y otras con encapsulados TQFP, QFN o MLF de 44 terminales. Las versiones en PDIP son las más convenientes durante el desarrollo de prototipos por su compatibilidad con las tablillas de pruebas (*protoboard*). En la figura 2.2 se muestra el aspecto externo para ambos dispositivos, considerando un encapsulado PDIP.

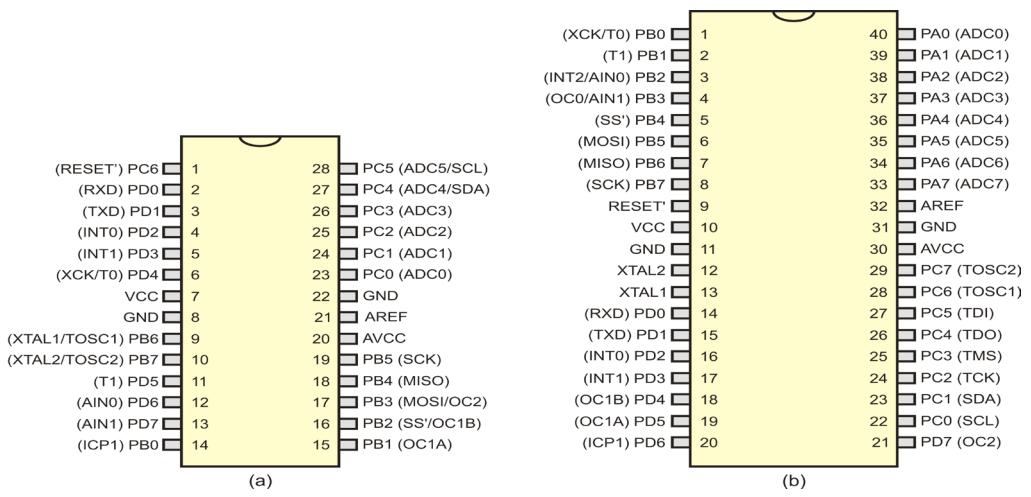


Figura 2.2 Aspecto externo de (a) un ATMega8 y (b) un ATMega16

El ATMega8 incluye 3 puertos, 2 de 8 bits y 1 de 7 bits; mientras que el ATMega16 contiene 4 puertos, todos de 8 bits. También se observa que todas las terminales incluyen una doble o triple función, esto significa que además de utilizarse como entrada o salida de propósito general, las terminales pueden emplearse con un propósito específico, relacionado con alguno de los recursos del microcontrolador.

2.2 El Núcleo AVR

La organización interna de los microcontroladores bajo estudio se fundamenta en el núcleo AVR, el núcleo es la unidad central de procesamiento (CPU), es decir, es el hardware encargado de la captura, decodificación y ejecución de instrucciones, su organización se muestra en la figura 2.3. En torno al núcleo se encuentra un bus de 8 bits al cual están conectados los diferentes recursos del microcontrolador, estos recursos pueden diferir entre dispositivos.

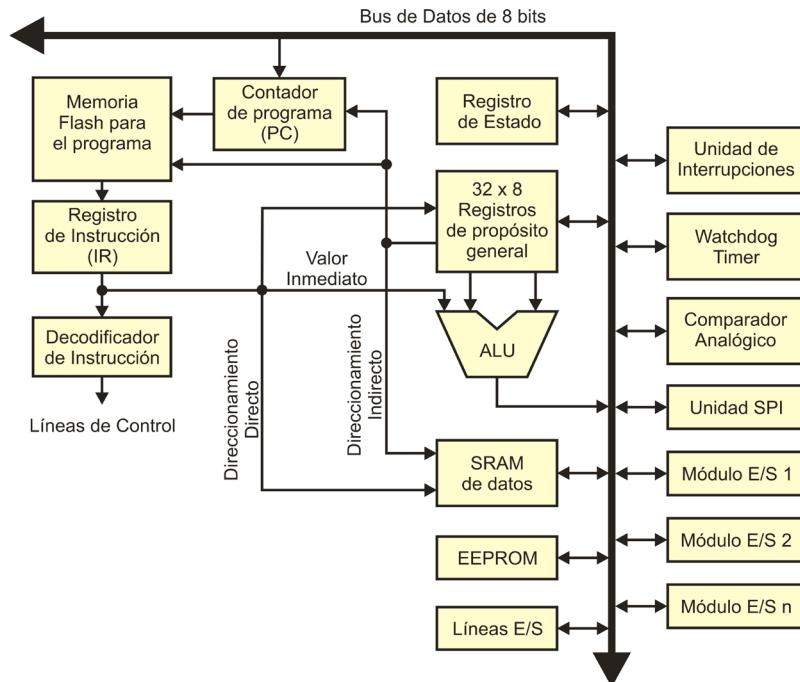


Figura 2.3 Diagrama a bloques del núcleo AVR

La principal función de la CPU es asegurar la correcta ejecución de programas. La CPU debe tener acceso a los datos, realizar cálculos, controlar periféricos y manejar interrupciones.

Para maximizar el rendimiento y paralelismo, el AVR usa una arquitectura Harvard con memorias y buses separados para el programa y los datos. Esto se observa en la figura 2.3, el programa se ubica en la memoria flash y los datos están en 3 espacios diferentes: En el archivo de registros (32 registros de 8 bits), en la SRAM y en la EEPROM.

De la memoria flash se obtiene cada instrucción del programa y se coloca en el registro de instrucción (IR) para su decodificación y ejecución. La memoria flash es direccionada por el contador de programa (PC) o bien, por uno de los registros de propósito general. El PC es en sí el registro que indica la ubicación de la instrucción a ejecutar, sin embargo, es posible que un registro de propósito general proporcione esta dirección a modo de que funcione como apuntador y se haga un acceso utilizando direccionamiento indirecto.

La ALU soporta operaciones aritméticas y lógicas entre los 32 registros de propósito general o entre un registro y una constante, para cualquier operación, al menos uno de los operandos es uno de estos registros. Los 32 registros son la base para el procesamiento de datos porque la arquitectura es del tipo registro-registro, esto implica que si un dato de SRAM o de EEPROM va a ser modificado, primero debe ser llevado a cualquiera de los 32 registros de 8 bits, dado que todos tienen la misma jerarquía.

El Registro de Estado principalmente contiene banderas que se actualizan después de una operación aritmética, para reflejar información relacionada con el resultado de la operación. Las banderas posteriormente pueden ser utilizadas por diversas instrucciones para tomar decisiones.

2.2.1 Ejecución de Instrucciones

El flujo del programa por naturaleza es secuencial, con incrementos automáticos del PC. Este flujo secuencial puede ser modificado con instrucciones de saltos condicionales o incondicionales y llamadas a rutinas, éstas son instrucciones que modifican directamente al PC, permitiendo abarcar completamente el espacio de direcciones.

La CPU va a capturar las instrucciones para después ejecutarlas, su organización hace posible que este proceso se segmente en dos etapas, solapando la captura con la ejecución de instrucciones. Es decir, mientras una instrucción está siendo ejecutada, la siguiente es capturada en IR. Con ello, aunque el tiempo de ejecución por instrucción es de dos ciclos de reloj, la productividad va a ser de una instrucción por ciclo de reloj, esto se muestra en la figura 2.4. Con lo cual el rendimiento de la CPU va a ser muy aproximado a 1 MIPS³ por cada MHz de la frecuencia del oscilador.

³ MIPS, métrica para medir el rendimiento de procesadores, significa Millones de Instrucciones por Segundo.

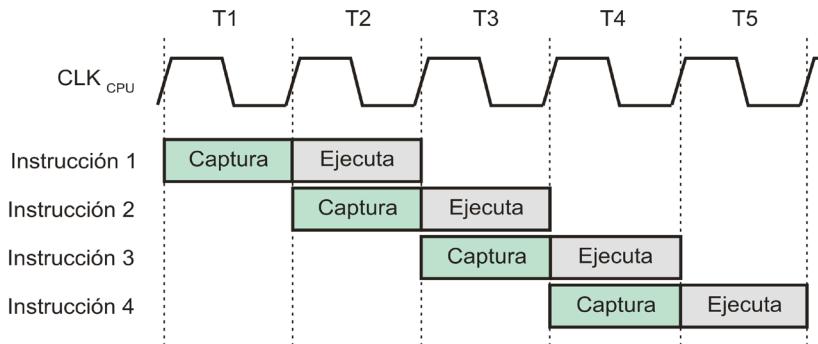


Figura 2.4 Segmentación a dos etapas realizado por el núcleo AVR

En los saltos y llamadas a rutinas no se puede anticipar la captura de la siguiente instrucción porque se ignora cuál es, por lo tanto, se pierde un ciclo de reloj. Algo similar ocurre con los accesos a memoria (cargas o almacenamientos), instrucciones que gastan un ciclo de reloj para la manipulación de direcciones, antes de hacer el acceso.

Para las instrucciones aritméticas y lógicas es suficiente con un ciclo de reloj para su ejecución (posterior a la captura), al comienzo del ciclo se capturan los operandos de los registros de propósito general, la ALU trabaja sincronizada con el flanco de bajada y prepara el resultado para que sea escrito en el siguiente flanco de subida, esto se muestra en la figura 2.5.

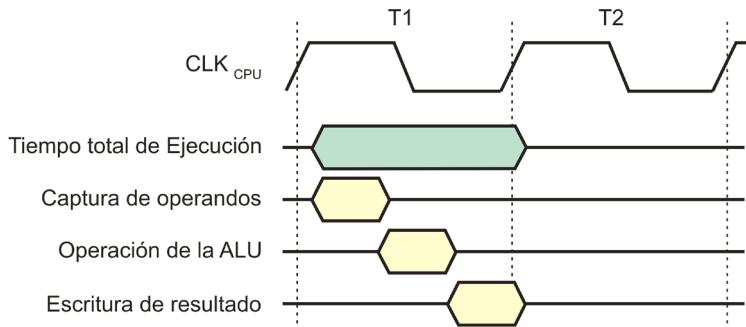


Figura 2.5 Temporización de la fase de ejecución

2.2.2 Archivo de Registros

El archivo de registros contiene 32 registros de 8 bits de propósito general, el núcleo AVR está acondicionado para tener un acceso rápido a ellos. La organización del archivo de registros se muestra en la figura 2.6, los registros se denominan R0, R1, R2, etc.; las instrucciones que operan sobre registros se ejecutan en 1 ciclo de reloj.

		7	0	Dirección
		R0		0x00
		R1		0x01
		R2		0x02
		...		
		R14		0x0E
		R15		0x0F
		R16		0x10
		R17		0x11
		...		
X	{	R26 (XL)		0x1A
		R27 (XH)		0x1B
Y	{	R28 (YL)		0x1C
		R29 (YH)		0x1D
Z	{	R30 (ZL)		0x1E
		R31 (ZH)		0x1F

Figura 2.6 Archivo de Registros

Los últimos 6 registros se organizan por pares, formando 3 registros de 16 bits, de esta manera se pueden utilizar como apuntadores para direccionamiento indirecto en el espacio de datos, para ello, estos registros se denominan X, Y y Z. El registro Z también puede usarse como apuntador a la memoria de programa.

Al contar con registros que funcionan como apuntadores, acondicionados para realizar cálculos eficientes de direcciones, e incluir en su repertorio instrucciones de comparaciones con auto-incrementos o auto-decrementos, el núcleo AVR está optimizado para que ejecute código C compilado, ya que entre las características de este lenguaje se encuentra el uso extensivo de apuntadores y ciclos repetitivos con incrementos o decrementos.

2.3 Memoria de Programa

La memoria de programa es un espacio continuo de memoria Flash cuyo tamaño varía entre los miembros de la familia AVR, para el ATMega8 es de 8 KB y para el ATMega16 es de 16 KB. La memoria está organizada en palabras de 16 bits y la mayoría de las instrucciones utilizan una palabra, por lo tanto, el rango de direcciones es de 0x000 a 0xFFFF en un ATMega8 y hasta 0x1FFF en un ATMega16. Esto se muestra en la figura 2.7, también se observa que la memoria puede ser particionada en una sección de aplicación y una sección de arranque. En la sección de arranque es posible manejar un cargador para auto programación, con esto, un sistema de manera autónoma puede revisar si existe una versión más actual de su aplicación, en la sección 7.2 se describe cómo tener acceso a la sección de arranque. Si la sección de arranque no es requerida, todo el espacio es dedicado a la aplicación.

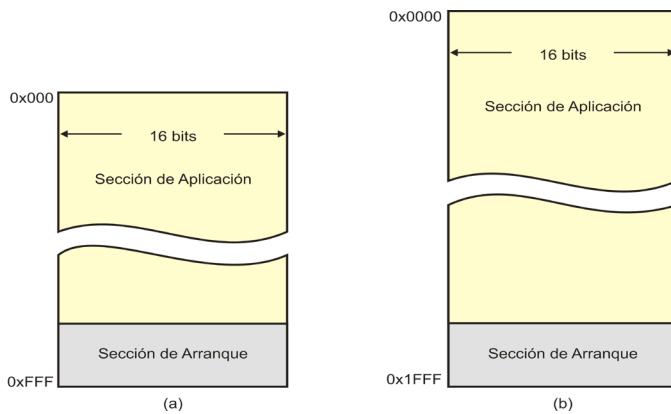


Figura 2.7 Memoria de Programa en (a) un ATMega8 y (b) un ATMega16

La memoria puede ser programada sin necesidad de retirar al MCU de un sistema (programación “*In System*”) y soporta hasta 10,000 ciclos de escritura/borrado. En la memoria de programa se encuentran los vectores de interrupciones, es decir, direcciones que toma el PC para que el flujo del programa bifurque a las rutinas que atienden a los eventos que fueron configurados.

Se tiene 19 fuentes de interrupción en un ATMega8 y 21 en un ATMega16, incluyendo la interrupción por reinicio (*reset*). En las tablas 2.1 y 2.2 se describen los vectores de las interrupciones para el ATMega8 y el ATMega16, respectivamente.

Tabla 2.1 Vectores de las Interrupciones en un ATMega8

Vector	Dirección	Fuente	Definición de la Interrupción
1	0x000	RESET	Reinicio por terminal externa, encendido, voltaje bajo o por <i>watchdog timer</i>
2	0x001	INT0	Petición de interrupción externa 0
3	0x002	INT1	Petición de interrupción externa 1
4	0x003	TIMER2_COMP	Coincidencia por comparación en el temporizador 2
5	0x004	TIMER2_OVF	Desbordamiento del temporizador 2
6	0x005	TIMER1_CAPT	Captura del temporizador 1 ante un evento externo
7	0x006	TIMER1_COMPA	Coincidencia en la comparación A del temporizador 1
8	0x007	TIMER1_COMPB	Coincidencia en la comparación B del temporizador 1
9	0x008	TIMER1_OVF	Desbordamiento del temporizador 1
10	0x009	TIMER0_OVF	Desbordamiento del temporizador 0
11	0x00A	SPI_STC	Transferencia serial completa por SPI
12	0x00B	USART_RXC	Recepción serial completa en la USART
13	0x00C	USART_UDRE	Registro de datos de la USART vacío
14	0x00D	USART_RXC	Transmisión serial completa con la USART

Vector	Dirección	Fuente	Definición de la Interrupción
15	0x00E	ADC	Conversión completa en el ADC
16	0x00F	EE_RDY	Concluye una escritura en la EEPROM
17	0x010	ANA_COMP	Comparador analógico
18	0x011	TWI	Interfaz serial de dos hilos
19	0x012	SPM_RDY	Almacenamiento en memoria de programa listo

Tabla 2.2 Vectores de las Interrupciones en un ATMega16

Vector	Dirección	Fuente	Definición de la Interrupción
1	0x000	RESET	Reinicio por terminal externa, encendido, voltaje bajo o por <i>watchdog timer</i>
2	0x002	INT0	Petición de interrupción externa 0
3	0x004	INT1	Petición de interrupción externa 1
4	0x006	TIMER2_COMP	Coincidencia por comparación en el temporizador 2
5	0x008	TIMER2_OVF	Desbordamiento del temporizador 2
6	0x00A	TIMER1_CAPT	Captura del temporizador 1 ante un evento externo
7	0x00C	TIMER1_COMPA	Coincidencia en la comparación A del temporizador 1
8	0x00E	TIMER1_COMPB	Coincidencia en la comparación B del temporizador 1
9	0x010	TIMER1_OVF	Desbordamiento del temporizador 1
10	0x012	TIMERO_OVF	Desbordamiento del temporizador 0
11	0x014	SPI_STC	Transferencia serial completa por SPI
12	0x016	USART_RXC	Recepción serial completa en la USART
13	0x018	USART_UDRE	Registro de datos de la USART vacío
14	0x01A	USATR_TXC	Transmisión serial completa con la USART
15	0x01C	ADC	Conversión completa en el ADC
16	0x01E	EE_RDY	Concluye una escritura en la EEPROM
17	0x020	ANA_COMP	Comparador analógico
18	0x022	TWI	Interfaz serial de dos hilos
19	0x024	INT2	Petición de interrupción externa 2
20	0x026	TIMERO_COMP	Coincidencia por comparación en el temporizador 0
21	0x028	SPM_RDY	Almacenamiento en memoria de programa listo

2.4 Memoria de Datos

Para el almacenamiento de datos, los microcontroladores incluyen dos espacios con tecnologías diferentes, como se muestra en la figura 2.8, un espacio de SRAM de 1120 bytes, para el almacenamiento de variables o datos volátiles, y un espacio de EEPROM de 512 bytes, para aquellos datos que se quieren preservar aun en ausencia de energía, como contraseñas, parámetros de configuración, etc. Esto tanto para el ATMega8 como para el ATMega16.

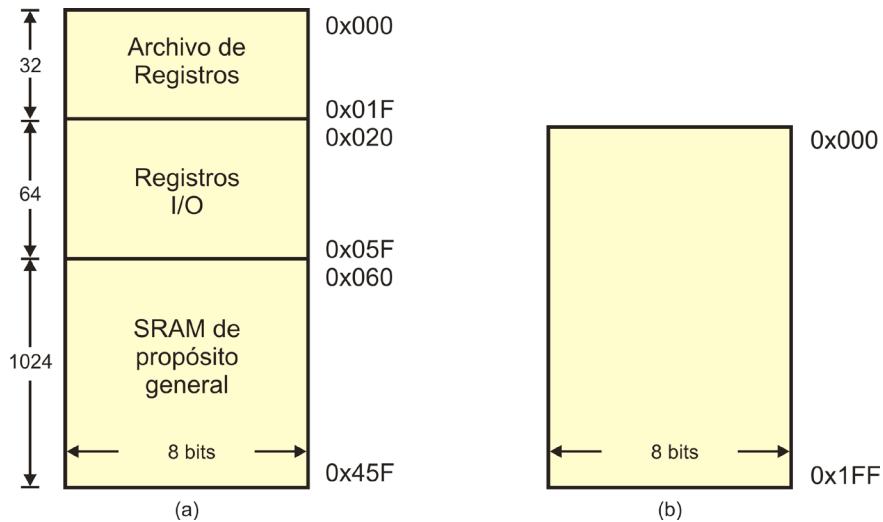


Figura 2.8 Memoria de Datos (a) SRAM y (b) EEPROM

2.4.1 Espacio de SRAM

En la SRAM se tienen tres espacios diferentes en un mapa con direccionamiento lineal, inicia en la dirección 0x000 y concluye en la 0x45F. Las primeras 32 localidades son del Archivo de Registros, luego siguen 64 localidades denominadas como Registros I/O, necesarios para el manejo de recursos, y finalmente se tienen 1024 localidades de SRAM de propósito general.

El núcleo AVR está optimizado para trabajar con los registros de propósito general (sección 2.2.2), las instrucciones los refieren como R0 a R31, o bien como apuntadores (X, Y o Z). No obstante, estos registros también pueden ser referidos como cualquier localidad de SRAM de propósito general, utilizando instrucciones de carga (**LD**) o almacenamiento (**ST**). Esto se muestra en la figura 2.9, en donde se observa que los registros tienen una dirección en el espacio de los datos.

2.4.1.1 Registros I/O

Los Registros I/O son necesarios para el manejo de los recursos internos de un microcontrolador, se tiene un espacio de direcciones para ubicar hasta 64 registros, las direcciones están en el rango de 0x00 a 0x3F. Aunque el número de registros realmente implementados puede variar entre dispositivos, dependiendo de los recursos internos incluidos.

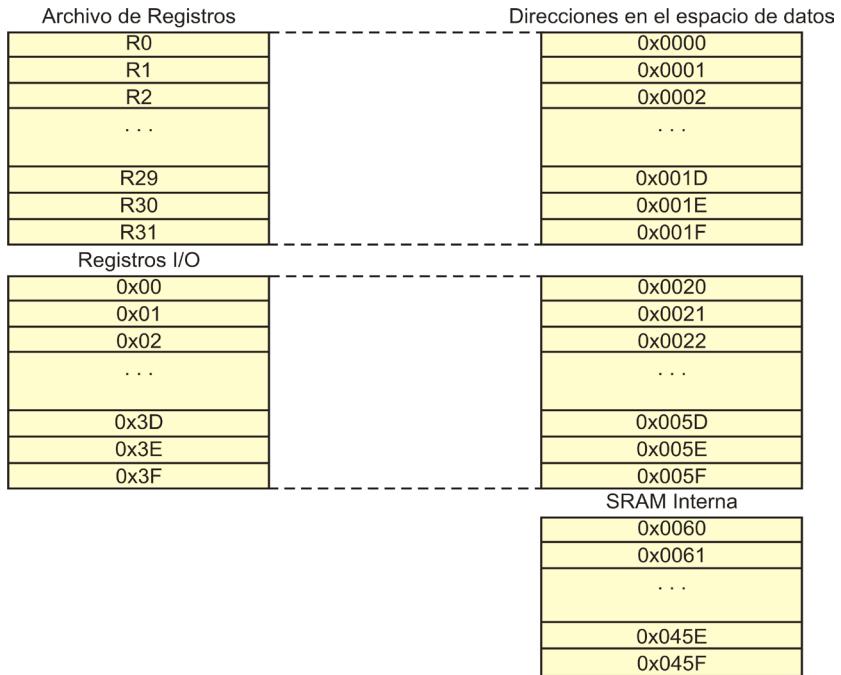


Figura 2.9 Memoria SRAM de Datos

Los Registros I/O se utilizan para definir la configuración, realizar el control o monitorear el estado de los recursos internos. En la figura 2.3 no se encuentra un espacio que explícitamente especifique la ubicación de los Registros I/O, dado que éstos son parte de los módulos con los recursos internos. Por ejemplo, para el manejo de cada uno de los puertos se requiere de 3 registros, uno para configurar al puerto como entrada o salida (configuración), otro para escribir en el puerto (control) y otro para leer del puerto (estado).

La arquitectura de los AVR incluye a las instrucciones **IN** y **OUT** con las que se tiene un acceso rápido a los Registros I/O, con **IN** se transfiere la información de un Registro I/O a un Registro de Propósito General y con **OUT** se realiza la operación complementaria, en ambos casos la ejecución se realiza en 1 ciclo de reloj.

De acuerdo con la figura 2.9, los Registros I/O también pueden ser referidos como cualquier localidad de SRAM de propósito general, utilizando instrucciones de carga (**LDI**) o almacenamiento (**ST**), con direcciones en el rango de 0x20 a 0x5F. Tratar a los Registros de Propósito General o a los Registros I/O como SRAM de propósito general no es conveniente, porque las instrucciones de acceso a memoria se ejecutan en 2 ciclos de reloj.

En la tabla 2.3 se muestra una parte de los Registros I/O, el mapa completo se encuentra en el apéndice A. El objetivo de cada registro se describe conforme se van revisando los recursos a los que pertenece, los recursos internos son descritos en los capítulos 4, 5, 6 y 7.

Tabla 2.3 Parte del mapa de Registros I/O

Dirección (Espacio I/O)	Dirección (SRAM)	Nombre	Función
0x3F	0x5F	SREG	Registro de Estado y Control
0x3E	0x5E	SPH	Apuntador de Pila (byte alto)
0x3D	0x5D	SPL	Apuntador de Pila (byte bajo)
0x3C	0x5C	OCRO	Registro para comparación del Temporizador / Contador 0 (no disponible en ATMega8)
0x3B	0x5B	GICR	Registro General para el Control de las Interrupciones
0x3A	0x5A	GIFR	Registro General de banderas de Interrupciones
0x39	0x59	TIMSK	Registro para enmascarar las interrupciones por los Temporizadores/Contadores
0x38	0x58	TIFR	Registro de bandera de interrupciones por los Temporizadores/Contadores
...

Los Registros I/O que están en el rango de 0x00 a 0x1F pueden ser manipulados por sus bits individuales. Con las instrucciones **SBI** (*Set Bit in I/O Register*, ajusta un bit en un Registro I/O) y **CBI** (*Clear Bit in I/O Register*, limpia un bit en un Registro I/O) es posible cambiar el estado de un bit individual sin modificar al resto, y a través de las instrucciones **SBIS** (*Skip if Bit in I/O Register is Set*, brinca si el bit del Registro I/O está en alto) y **SBIC** (*Skip if Bit in I/O Register Cleared*, brinca si el bit del Registro I/O está en bajo) es posible evaluar el estado de un bit para determinar la realización de un brinco.

Registro de Estado

El Registro de Estado (**SREG**, *State Register*) es parte de los Registros I/O, por lo que su acceso puede hacerse con instrucciones **IN** y **OUT**. Este registro es importante debido a que refleja el estado de la CPU y no de algún recurso específico, por eso existen instrucciones especiales para modificar o evaluar a cada uno de sus bits individualmente. Se ubica en la dirección 0x3F (o 0x5F de SRAM), después de un reinicio, todos sus bits tienen el valor de 0. Los bits del Registro de Estado son:

7	6	5	4	3	2	1	0	SREG
I	T	H	S	V	N	Z	C	

- **Bit 7 – I: Habilitador Global de Interrupciones**

Con un 1 lógico las interrupciones son habilitadas, sin embargo, cada interrupción también tiene su habilitador individual. Debe habilitarse por software. Cuando ocurre una interrupción, este bit es limpiado por hardware, para evitar que durante su atención ocurran otras interrupciones.

- **Bit 6 – T: Bit de Almacenamiento para copias**

Es un espacio para el almacenamiento temporal de 1 bit, puede ser útil si se va a copiar un bit de un registro de propósito general a otro.

- **Bit 5 – H: Bandera de Acarreo en el nibble bajo (*Half Carry*)**

Se pone en alto si después de una operación aritmética, existe un bit de acarreo del nibble bajo al nibble alto.

- **Bit 4 – S: Bit de Signo**

Siempre mantiene una XOR entre la bandera de negativo (**N**) y la bandera de sobreflujo (**V**), ambas del registro de Estado.

- **Bit 3 – V: Bandera de Sobreflujo para operaciones en complemento-2**

Indica que ocurrió un sobreflujo aritmético, es decir, que el resultado de una operación aritmética no alcanzó en la representación de números en complemento a 2 (positivos y negativos). Por ejemplo, al sumar el número 97 (0b01100001) con 42 (0b0101010), el resultado es 139 (0b10001011), con este resultado la bandera V se pone en alto, porque en una representación en complemento a 2 el número 0b10001011 representa al -117. El sobreflujo aritmético se puede presentar en sumas y restas, existen 4 situaciones en las que se presenta sobreflujo, las cuales se resumen en la tabla 2.4.

Tabla 2.4 Situaciones de sobreflujo

Operación	Operando A	Operando B	Resultado (Indicación de sobreflujo)
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

- **Bit 2 – N: Bandera de Negativo**

Indica un resultado negativo en una operación aritmética o lógica, corresponde con el MSB del resultado.

- **Bit 1 – Z: Bandera de Cero**

Indica que el resultado de una operación aritmética o lógica fue cero.

- **Bit 0 – C: Bandera de Acarreo**

Indica que el resultado de una operación aritmética o lógica no alcanzó en 8 bits.

Ocasionalmente los bits de acarreo y sobreflujo suelen confundirse, aunque señalan dos situaciones diferentes. En una operación aritmética, si los operandos son de 8 bits se espera que el resultado ocupe sólo 8 bits, si el resultado no alcanza en 8 bits se genera un bit de acarreo. El sobreflujo tiene que ver con representaciones en complemento a 2 y un sobreflujo no implica que el resultado tenga que requerir de un 9º bit.

El Apuntador de Pila (*Stack Pointer, SP*)

La Pila es un espacio para el almacenamiento temporal de variables, el cual está implementado dentro de la SRAM de propósito general. Una Pila es una estructura en la cual los datos son almacenados o recuperados en uno de sus extremos, denominado tope, de manera que el último dato que ingresa es el primero que es extraído. El Apuntador de Pila (**SP**) es un registro de 16 bits que forma parte de los Registros I/O y que contiene la dirección del tope de la pila, son necesarios 16 bits para poder direccionar todo el espacio de SRAM. El **SP** se compone de 2 registros de 8 bits, **SPH** para la parte alta (0x3E) y **SPL** para la parte baja (0x3D).

	7	6	5	4	3	2	1	0	
0x3E	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL

La Pila tiene un crecimiento de las direcciones altas de SRAM hacia las direcciones bajas. Su acceso puede realizarse en forma explícita, mediante las instrucciones **PUSH** y **POP**. Con **PUSH** se inserta un dato y se disminuye al **SP**, y con **POP** se incrementa al **SP** y luego se extrae un dato. O bien de manera implícita, durante las llamadas y retornos de rutinas. En la llamada a una rutina, en la Pila se almacena la dirección de la instrucción que sigue a la llamada y con un retorno, el tope de la pila remplaza al PC, para continuar con el flujo anterior a la llamada.

Después de un reinicio, el **SP** tiene el valor de 0x0000, por lo tanto, los programas que incluyan rutinas o realicen accesos explícitos a la Pila, deben inicializar al **SP** con 0x045F (última localidad de SRAM, porque la pila crece hacia las direcciones bajas), para que los almacenamientos se realicen dentro de un espacio válido.

2.4.1.2 SRAM de Propósito General

El espacio de propósito general queda disponible para: variables simples que no alcanzan en los 32 registros, para variables compuestas, como arreglos o estructuras, o bien, para la pila de datos temporales. Pero al ser una arquitectura del tipo Registro a Registro, cualquier variable de SRAM que requiera una modificación debe ser llevada a un registro, para ello se realiza una carga (**LDR**, *load*) y para respaldar un registro en SRAM se realiza un almacenamiento (**STR**, *store*), esto se representa en la figura 2.10.

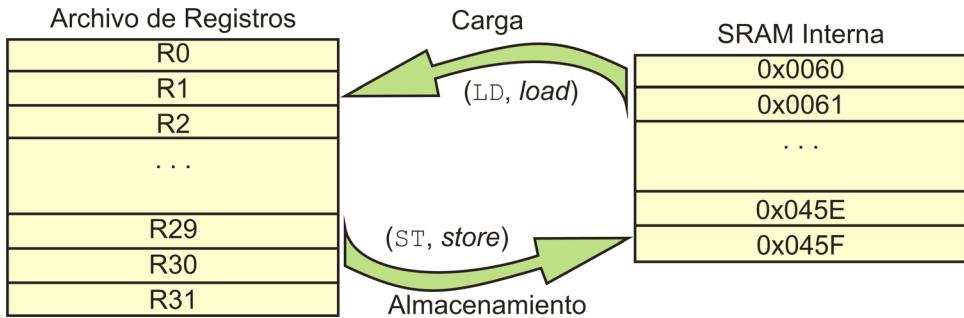


Figura 2.10 Accesos a la memoria SRAM de propósito general

Existe una variedad de instrucciones para el acceso a memoria, ya sean cargas o almacenamientos, utilizando modos de direccionamiento directo o indirecto (por apuntador), así como instrucciones que automáticamente modifican al apuntador, incrementándolo o disminuyéndolo. Todos los accesos a memoria requieren de 3 ciclos de reloj, uno para captura, otro para calcular la dirección de acceso y en el último se habilita la escritura o lectura de la SRAM, esto se muestra en la figura 2.11. Por la segmentación, en el tercer ciclo se captura la siguiente instrucción, aparentando que los accesos a memoria sólo requieren de 2 ciclos de reloj.

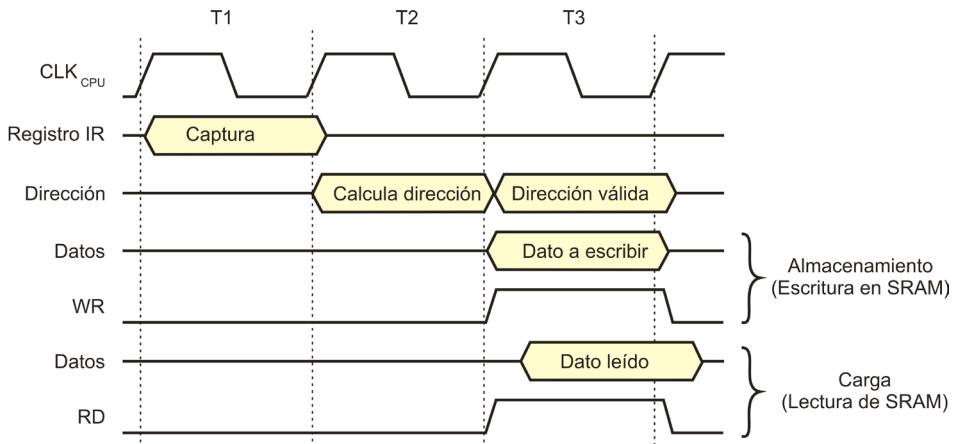


Figura 2.11 Temporización de los accesos a la SRAM de propósito general

2.4.2 Espacio de EEPROM

La EEPROM es un espacio no volátil para el almacenamiento de datos, cuyo tamaño varía en los diferentes integrantes de la familia AVR. Para los microcontroladores ATmega8 y ATmega16 este espacio es de 512 bytes. La memoria EEPROM es un espacio que el núcleo trata como un recurso interno, de manera que su acceso es por medio de los Registros I/O.

Una memoria de cualquier tipo requiere de 3 buses para su manejo: un bus de datos, un bus de direcciones y un bus de control. En el caso de la EEPROM, puesto que es interna al microcontrolador, los buses son manejados por medio de 3 Registros I/O. Para el manejo de las direcciones se tiene al registro **EEAR** (*EEPROM Address Register*), **EEAR** ocupa dos espacios en los Registros I/O para direccionar 512 bytes (9 bits de dirección), estos registros son:

	7	6	5	4	3	2	1	0	
0x1F	-	-	-	-	-	-	-	EEAR8	EEARH
0x1E	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL

Para el manejo de los datos se dispone del registro **EEDR** (*EEPROM Data Register*), el cual se ubica en la dirección 0x1D, dentro del mapa de Registros I/O. Si un dato va a ser escrito en la EEPROM, debe ser colocado en **EEDR**, antes de iniciar con un ciclo de escritura. Para lecturas de la EEPROM, después de un ciclo de lectura, el dato queda disponible en **EEDR**.

Las señales de control son manejadas con el registro **EECR** (*EEPROM Control Register*), en este registro se hacen las habilitaciones requeridas para iniciar los ciclos de lectura o escritura, únicamente los 4 bits menos significativos están implementados, éstos son:

	7	6	5	4	3	2	1	0	
0x1C	-	-	-	-	EERIE	EEMWE	EEWE	EERE	ECCR

- **Bit 3 – EERIE: Habilitador de Interrupción por fin de Escritura en la EEPROM**

Las escrituras en la EEPROM requieren varios ciclos de reloj, su final puede detectarse sondeando el estado del bit **EEWE** o por interrupción. Si el bit **EERIE** está en alto, se genera una interrupción cuando culmina una escritura en la EEPROM, siempre que el habilitador global de interrupciones (bit **I** de **SREG**) también esté puesto en alto.

- **Bit 2 – EEMWE: Habilitador Maestro para Escrituras en la EEPROM**

La EEPROM está orientada para datos importantes en un sistema, como bits de configuración o contraseñas, por lo tanto, es importante evitar que su contenido se pierda por escrituras erróneas. El bit **EEMWE** es parte de un esquema de seguridad y protección del contenido de la EEPROM, al poner en alto a este bit se cuenta con 4 ciclos de reloj dentro de los cuales se puede iniciar con un ciclo de escritura. Pasados estos 4 ciclos el bit **EEMWE** es puesto automáticamente en un nivel bajo y ya no es posible escribir en la EEPROM.

- **Bit 1 – EEWE: Habilitador de Escritura en la EEPROM**

Al poner en alto a este bit se inicia con un ciclo de escritura, siempre que el bit **EEMWE** haya sido puesto en alto en los 4 ciclos de reloj anteriores. No es posible iniciar con un ciclo de escritura si hay una escritura en proceso. Cuando concluye la escritura, el bit **EEWE** es automáticamente puesto en bajo, este evento puede ser detectado por sondeo o bien, si el bit **EERIE** está en alto, se genera una interrupción.

- **Bit 0 – EERE: Habilitador de Lectura en la EEPROM**

Al poner en alto a este bit se inicia con un ciclo de lectura, para lo cual sólo se requiere que no haya una escritura en proceso. La lectura es inmediata, el dato leído puede manipularse con la siguiente instrucción.

Ejemplo 2.1: Muestre cómo se codificaría una rutina en lenguaje ensamblador para realizar una escritura en EEPROM.

Para esta rutina se asume que el dato a escribir está en **R16** y que la dirección a utilizar se describe en **R18:R17**, el código es:

```
EEPROM_write:
    SBIC  EECR, EEWE          ; Asegura que no hay escritura en proceso
    RJMP  EEPROM_write

    OUT   EEARH, R18          ; Establece la dirección
    OUT   EEARL, R17
    OUT   EEDR, R16          ; Coloca el dato a escribir

    SBI   EECR, EEMWE         ; Pone en alto al habilitador maestro
    SBI   EECR, EEWE           ; Inicia la escritura

    RET
```

Con la instrucción **SBIC EECR, EEWE** (brinca si el bit **EEWE** del registro **EECR** está en bajo) se está sondeando al bit **EEWE**. El brinco no se realiza si hay una escritura en proceso y se continúa con la siguiente instrucción, la cual reinicia la rutina, el ciclo se va a mantener mientras no concluya la escritura previa.

Ejemplo 2.2: Realice una rutina en lenguaje ensamblador para hacer una lectura en EEPROM.

La rutina lee de la dirección formada por **R18:R17** y el dato leído es colocado en **R16**.

```
EEPROM_read:
    SBIC  EECR, EEWE; Asegura que no hay escritura en proceso RJMP
    EEPROM_read
```

OUT	EEARH , R18	<i>; Establece la dirección</i>
OUT	EEARL , R17	
SBI	EECR , EERE	<i>; Inicia la lectura</i>
IN	R16 , EEDR	<i>; Coloca el dato leído</i>
RET		

Ejemplo 2.3: Desarrolle 2 funciones en Lenguaje C, una para realizar una escritura en EEPROM y la otra para hacer una lectura.

En lenguaje C, las instrucciones deben realizar las mismas operaciones sobre los registros. La función para la escritura recibe como parámetros el dato a escribir y la dirección en donde va a ser escrito:

```
void EEPROM_write (unsigned char dato, unsigned int direccion)
{
    while ( EECR & 1 << EEWE ) // Asegura que no hay escritura en proceso
    ;
    EEAR = direccion;          // Establece la dirección
    EEDR = dato;               // Coloca el dato a escribir

    EECR |= ( 1 << EEMWE );   // Pone en alto al habilitador maestro
    EECR |= ( 1 << EEWE );    // Inicia la escritura
}
```

La función para la lectura recibe la dirección a leer y regresa el dato leído:

```
unsigned char EEPROM_read(unsigned int direccion)
{
    while ( EECR & 1 << EEWE ) // Asegura que no hay escritura en proceso
    ;
    EEAR = direccion;          // Establece la dirección
    EECR |= ( 1 << EERE );    // Inicia la lectura

    return EEDR;                // Regresa el dato leído
}
```

2.5 Puertos de Entrada/Salida

Los puertos de entrada/salida proporcionan el mecanismo por medio del cual un microcontrolador se comunica con su entorno. En la figura 2.2 se mostró el aspecto externo de los 2 dispositivos bajo estudio, puede notarse que el ATMega8 tiene tres puertos: Puerto B, C y D, donde el puerto C es de 7 bits, mientras que los puertos B y D son de 8 bits. Con respecto al ATMega16 se observan 4 puertos: Puerto A, B, C y D, todos de 8 bits.

Los puertos son de propósito general, es decir, el usuario puede utilizarlos para monitorear entradas o generar salidas como mejor le convenga, no obstante, todas las terminales tienen una función alterna, esto es necesario porque muchos de los recursos internos van a requerir información del exterior. Por ejemplo, el puerto serie necesita una terminal externa para recepción y otra para transmisión, ésta es la función alterna para PD0 y PD1, respectivamente. La función alterna de las terminales se revisa conforme se van describiendo los recursos internos del microcontrolador, los recursos internos se describen en los capítulos 4, 5, 6 y 7.

Para el manejo de cada puerto se requiere de 3 registros del espacio de Registros I/O, éstos son:

- **PORTx:** Registro para la escritura o lectura de un *latch* que está conectado a la terminal del puerto por medio de un buffer de 3 estados. El buffer está activo cuando el puerto es configurado como salida, bajo estas circunstancias, todo lo que se escribe en este registro se ve reflejado en las terminales de los puertos.
- **DDRx:** Registro para la escritura o lectura de un *latch* que está conectado a la terminal de activación del buffer de 3 estados, con este registro se define la dirección del puerto. Si se escribe un **1 lógico**, se activa al buffer haciendo que el puerto funcione como **salida**, al escribir un **0 lógico**, el buffer está inactivo y el puerto funciona como **entrada**. Cada terminal tiene sus propios recursos, de manera que la dirección de una terminal puede diferir de las demás, aun en el mismo puerto.
- **PINx:** Registro sólo de lectura, para hacer lecturas directas en las terminales de los puertos.

La **x** hace referencia a cada uno de los puertos, puede ser A, B, C o D. Para el ATMega8, como tiene 3 puertos requiere de 9 Registros I/O, éstos son:

	7	6	5	4	3	2	1	0	
0x18	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
0x17	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	DDR_B
0x16	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PIN_B
0x15	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORT_C
0x14	-	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0	DDR_C
0x13	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINCO	PIN_C
0x12	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORT_D
0x11	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0	DDR_D
0x10	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIN_D

El bit 7 en los registros del puerto C no está implementado en un ATMega8.

El ATMega16 tiene 4 puertos, los registros de los puertos B, C y D tienen la misma dirección que en un ATMega8, difiriendo en que el bit 7 del puerto C si está implementado.

El ATmega16 también tiene al puerto A, el cual requiere de los siguientes 3 registros:

	7	6	5	4	3	2	1	0	
0x1B	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
0x1A	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	DDRA
0x19	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA

Si no se considera la función alterna en las terminales de los puertos, éstos incluyen el mismo hardware, el cual es mostrado en la figura 2.12, en donde se presenta la organización de la terminal **n** en el puerto **x**.

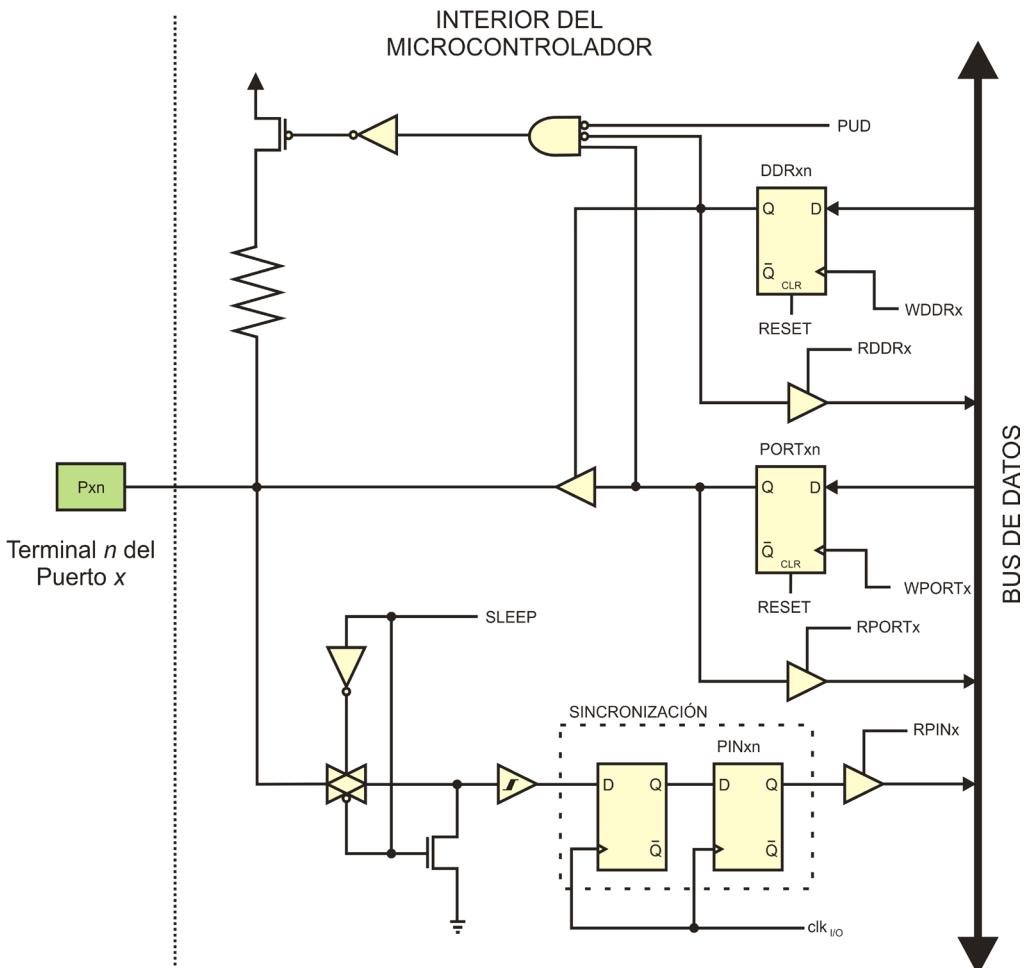


Figura 2.12 Organización del hardware de la terminal n del puerto x

En la figura 2.12 se observa que para cada terminal se tienen 3 rutas:

- Una ruta de salida que es manejada por el flip-flop tipo D denominado **PORTxn**, el cual puede ser leído o escrito. Este flip-flop se conecta a la terminal de salida por medio de un buffer de 3 estados. Una lectura en **PORTxn** no devuelve el valor de la terminal, sino lo último que fue escrito en el flip-flop.
- Una ruta de entrada que es manejada por el flip-flop tipo D denominado **PINxn**, en el cual continuamente se va a escribir el contenido de la terminal del puerto. Este flip-flop va precedido por otro similar, ambos manejados por la señal del reloj de entrada y salida. Esta conexión en cascada garantiza una señal estable al realizar las lecturas del puerto, con el inconveniente de que un cambio en la terminal se ve reflejado hasta el ciclo de reloj siguiente. También se observa un circuito para desconectar al puerto ante algún modo de reposo (SLEEP) del microcontrolador, así como un buffer con histéresis, para eliminar ruido. Si la terminal fue configurada como salida (buffer de 3 estados habilitado), con una lectura en **PINxn** se obtiene el dato escrito en **PORTxn**.
- Una ruta para el buffer de tres estados que es manejada por el flip-flop tipo D denominado **DDRxn**, el cual puede ser leído o escrito. Este flip-flop se conecta con el buffer de tres estados para determinar si el puerto va a estar configurado como entrada (se escribe un 0 lógico para desactivar al buffer) o como salida (se escribe un 1 lógico para activar al buffer).

También se observa una resistencia de fijación hacia un nivel alto de voltaje (*pull-up*), su habilitación depende de valor del flip-flop **PORTxn**, del flip-flop **DDRxn** y del bit **PUD** (*PUD: Pull-Up Disable*). El bit **PUD** se encuentra en el Registro I/O denominado **SFIOR** (*SFIOR: Special Function I/O Register*; Registro I/O de función especial). Con estos 3 bits se generan las combinaciones que se resumen en la tabla 2.5.

Tabla 2.5 Estado del Puerto ante las diferentes combinaciones entre DDRxn, PORTxn y PUD

DDRxn	PORTxn	PUD (en SFIOR)	E/S	Pull-Up	Comentario
0	0	X	Entrada	No	Entrada sin resistor de <i>Pull-Up</i>
0	1	0	Entrada	Si	Entrada con resistor de <i>Pull-Up</i>
0	1	1	Entrada	No	Entrada sin resistor de <i>Pull-Up</i>
1	0	X	Salida	No	Salida en bajo
1	1	X	Salida	No	Salida en alto

De las 3 combinaciones en que el puerto es entrada, sólo en la segunda queda habilitado el resistor de *Pull-Up*, esta combinación es adecuada para el manejo de botones o interruptores, porque garantiza un 1 lógico cuando el dispositivo está abierto. El otro extremo del botón o interruptor se conecta a tierra, de manera que cuando se cierra el circuito introduce un 0 lógico, el resistor de *Pull-Up* evita un corto entre el voltaje de alimentación y tierra.

Ejemplo 2.4: Muestre el código requerido para configurar la parte alta del puerto B como entradas y la parte baja como salidas, y habilite los resistores de *Pull-Up* de las 2 entradas más significativas.

La versión en lenguaje ensamblador:

```
LDI    R16, 0B00001111      ; Configuración de entradas y salidas  
OUT    DDRB, R16  
  
LDI    R17, 0B11000000      ; Resistencia de Pull-Up en los 2 MSB  
OUT    PORTB, R17
```

En lenguaje C el código se simplifica:

```
DDRB = 0B00001111; // Configuración de entradas y salidas  
PORTB = 0B11000000; // Resistencia de Pull-Up en los 2 MSB
```

Ejemplo 2.5: Muestre la secuencia de código que configure al puerto A como entrada y al puerto B como salida, para luego transferir la información del puerto A al puerto B.

En la versión en ensamblador, se utiliza un registro interno para poder hacer la transferencia:

```
LDI    R16, 0x00          ; Configura el Puerto A como entrada  
OUT    DDRA, R16  
LDI    R16, 0xFF          ; Configura al Puerto B como salida  
OUT    DDRB, R16  
  
IN     R16, PINA          ; Las lecturas se hacen en el registro PIN  
OUT    PORTB, R16         ; Las escrituras se hacen en el registro PORT
```

En lenguaje C no se requiere de una variable de manera explícita, para hacer la transferencia:

```
DDRA = 0x00;              // Configura el Puerto A como entrada  
DDRB = 0xFF;              // Configura al Puerto B como salida  
  
PORTB = PINA;             // Se lee en PIN y se escribe en PORT
```

Cabe aclarar que después de un reinicio los registros de los puertos tienen el valor de 0x00, por lo que por omisión, un puerto es configurado como entrada.

2.6 Sistema de Interrupciones

Una **interrupción** es la ocurrencia de un evento producido por algún recurso del microcontrolador, que ocasiona la suspensión temporal del programa principal. La CPU atiende al evento con una función conocida como rutina de servicio a la interrupción (*ISR, Interrupt Service Routine*). Una vez que la CPU concluye con las instrucciones de la ISR, continúa con la ejecución del programa principal, regresando al punto en donde fue suspendida su ejecución.

El núcleo AVR cuenta con la unidad de interrupciones, un módulo que va a determinar si se tienen las condiciones para que ocurra una interrupción. Son tres las condiciones necesarias para que un recurso produzca una interrupción: El habilitador global de interrupciones (bit **I** de **SREG**) debe estar activado, el habilitador individual de la interrupción del recurso también debe estar activado y en el recurso debe ocurrir el evento esperado.

Al configurar los recursos adecuadamente, por hardware se van a monitorear diferentes eventos, reduciendo la tarea que la CPU realiza por software, porque no necesita un sondeo continuo para determinar el estado de los recursos. Puesto que son muchos los eventos que pueden producir interrupciones, con el trabajo de la unidad de interrupciones se tiene la ilusión de que se están haciendo diferentes tareas en forma simultánea.

La ISR debe colocarse en una dirección preestablecida por Hardware, la cual corresponde con un vector de interrupciones.

Existen 2 esquemas para organizar a los vectores de interrupciones, en el primer esquema se maneja una dirección única (sólo un vector), de manera que cualquier evento ocasiona que la ejecución del programa bifurque a la misma dirección. Posterior a ello, por software se evalúa un conjunto de banderas para determinar la causa de la interrupción. En el segundo esquema se tienen diferentes direcciones (diversos vectores de interrupciones), una por cada evento, de manera que la dirección destino de la bifurcación depende de la causa de la interrupción.

El segundo esquema es el empleado por los microcontroladores AVR, requiere una estructura de Hardware más compleja que en el primer esquema, pero se simplifica el software, dado que no requiere de una revisión de banderas.

Un aspecto importante es que los eventos pueden ocurrir en cualquier momento, es decir, en forma asíncrona. Esto hace la diferencia entre una rutina normal y una ISR, dado que en el primer caso se puede calcular con certeza cuándo se va a realizar una llamada, podría decirse que el flujo de un programa sin interrupciones es conocido con antelación. Por el contrario, en un programa con interrupciones el flujo puede cambiar en cualquier momento, es como si el programa trabajara en dos niveles, el nivel del programa base y el nivel de atención a las interrupciones, por lo tanto, como parte de las ISRs debe incluirse el código necesario para proteger a las variables o registros cuyo contenido sea vital, al nivel del programa base. En la figura 2.13 se ilustra la idea.

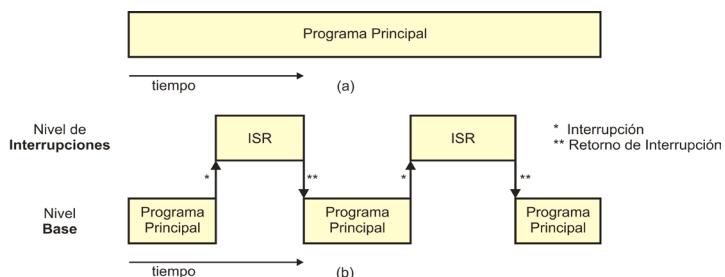


Figura 2.13 Representación de la ejecución de (a) un programa sin interrupciones y (b) un programa con interrupciones

En los microcontroladores AVR se tienen diferentes fuentes de Interrupción:

- Una interrupción por inicialización o *Reset*.
- Dos interrupciones externas (tres en el ATMega16).
- Siete interrupciones por los temporizadores (ocho en el ATMega16), pueden ser por comparación, captura o desbordamiento.
- Una interrupción al completar una transferencia serial (puerto SPI).
- Tres debidas el puerto serie (USART): por transmisión, por recepción y por buffer vacío.
- Una al finalizar una conversión analógica a digital.
- Una al finalizar una escritura en EEPROM.
- Una por el comparador analógico.
- Una por la interfaz serial a dos hilos (TWI).
- Una para la escritura en memoria de programa.

Se observa que son 2 interrupciones más en un ATMega16 que en un ATMega8, en otros miembros de la familia AVR el número de interrupciones puede variar, ya que depende de los recursos empotrados en el dispositivo.

Cuando el microcontrolador se enciende o reinicia, las interrupciones no están habilitadas, su habilitación requiere la puesta en alto del bit **I** de **SREG** y de los habilitadores individuales de los periféricos incorporados en el microcontrolador. En los capítulos 4, 5, 6 y 7 se describen los recursos internos, incluyendo una descripción de los bits para activar las interrupciones y las condiciones necesarias para que se generen.

Al generarse una interrupción, el PC es almacenado en la pila de datos y toma el valor de una entrada en el vector de interrupciones (según sea la interrupción). Además de desactivar al bit **I** para no aceptar más interrupciones. En la tabla 2.1 se mostraron las direcciones de los vectores de interrupciones en un ATMega8 y en la tabla 2.2 las de un ATMega16.

Una rutina de atención a interrupciones es finalizada con la instrucción **RETI**, con la cual el PC recupera el valor del tope de la pila y pone en alto nuevamente al bit **I**, para que la CPU pueda recibir más interrupciones.

El manejo de interrupciones necesariamente hace uso de la pila, por este motivo, cuando se programa en lenguaje ensamblador se debe inicializar al apuntador de pila en el programa principal. En lenguaje C esto no es necesario, dado que es parte del código que se agrega al hacer la traducción de alto nivel a código máquina.

Ejemplo 2.6: Muestre como se organizaría un programa en lenguaje ensamblador para utilizar interrupciones en un ATMega8.

```
; Un programa generalmente inicia en la dirección 0

.ORG 0x000
RJMP Principal ; Se evita el vector de interrupciones

.ORG 0x001
RJMP Externa_0 ; Bifurca a su ISR externa 0

.ORG 0x002
RJMP Externa_1 ; Bifurca a su ISR externa 1

; Si fuera necesario, aquí estarían otras bifurcaciones

.ORG 0x013
Principal:          ; Aquí se debe ubicar el código principal
...                 ; Debe activar las interrupciones
...
; Posterior al código principal, deben situarse las ISRs

Externa_0:          ; Respuesta a la interrupción externa 0
...
RETI                ; Debe terminar con RETI

Externa_1:          ; Respuesta a la interrupción externa 1
...
RETI                ; Debe terminar con RETI
```

La directiva **.ORG** indica en donde se van a ubicar las instrucciones subsecuentes en la memoria de programa. Al comparar las tablas 2.1 y 2.2, se observa que en un ATMega8 sólo se dispone de una dirección para cada bifurcación, mientras que en un ATMega16 se dispone de 2 direcciones, esto porque el espacio completo de direcciones de un ATMega8 puede ser alcanzado con una instrucción **RJMP**, que sólo ocupa una palabra de 16 bits. Pero con esa instrucción no se puede cubrir el espacio completo de un ATMega16, para ello puede usarse la instrucción **JMP**, que tiene un alcance mayor por lo que ocupa 2 palabras de 16 bits. La diferencia en el alcance entre **RJMP** y **JMP** se debe a que manejan diferentes modos de direccionamiento, los cuales son descritos en la sección 3.2.

Por lo tanto, si el código del ejemplo 2.6 va a utilizarse en un ATMega16, únicamente deben cambiarse las direcciones de los vectores de interrupciones y en las bifurcaciones puede emplearse **RJMP** o **JMP**, dependiendo de la ubicación de la ISR.

En lenguaje C todas las funciones de atención a interrupciones se llaman ISR, difieren en que reciben argumentos diferentes, el argumento corresponde con una etiqueta proporcionada por el fabricante, seguida de la palabra **vect**. Las etiquetas se encuentran en la columna titulada como **fuente** en las tablas 2.1 y 2.2.

Ejemplo 2.7: Muestre cómo se organizaría un programa en lenguaje C para utilizar interrupciones en un AVR.

```
#include <avr/io.h>           // Definiciones de entradas y salidas
#include <avr/interrupt.h>      // Manejo de interrupciones

// Las ISRs se ubican antes del programa principal
ISR (INT0_vect)             // Servicio a la interrupción externa 0
{
    . . .
}

ISR (INT1_vect)              // Servicio a la interrupción externa 1
{
    . . .
}

int main(void)                // Programa Principal
{
    . . .
    // Debe activar las interrupciones
}
```

El código descrito en el ejemplo 2.7 es independiente del dispositivo a utilizar, funciona para cualquier miembro de la familia AVR.

2.6.1 Manejo de Interrupciones

Si en un programa se utilizan las interrupciones, se requiere:

- Configurar el recurso o recursos para monitorear el evento o eventos.
- Habilitar a la interrupción o interrupciones (individual y global, en cada caso).
- Continuar con la ejecución normal de la aplicación.

En aplicaciones que utilizan interrupciones, es frecuente ciclar al programa principal en un lazo infinito sin realizar alguna actividad, con ello, el MCU permanece ocioso, dejando la funcionalidad del sistema a las ISRs.

Cuando ocurre una interrupción, el microcontrolador automáticamente realiza lo siguiente:

- Concluye con la instrucción bajo ejecución.
- Desactiva al habilitador global de interrupciones, para que no pueda recibir una nueva interrupción mientras atiende a la actual.
- Respalda en la pila al PC (previamente incrementado).
- Asigna al PC una dirección de los vectores de interrupciones, para dar paso a la ISR.
- Atiende al evento con la ISR.

Cuando una ISR termina (con la instrucción **RETI**, si se programó en ensamblador), en el MCU ocurre lo siguiente:

- Se limpia la bandera del evento que generó la interrupción.
- El habilitador global se activa.
- El PC toma el valor del tope de la pila, para que la ejecución continúe en el programa principal.

2.7 Inicialización del Sistema (*reset*)

La inicialización o *reset* de un microcontrolador es fundamental para su operación adecuada, porque garantiza que sus registros internos van a tener un valor inicial conocido. En un ATmega8 se tienen cuatro causas o fuentes de *reset*, en el ATmega16 se agrega una más, dado que tiene recursos adicionales para el manejo de la interfaz JTAG⁴. En la figura 2.14 se muestra la organización del hardware de *reset*, se observa cómo las diferentes causas pueden producir la señal denominada Reset Interno, que es en sí la que afecta a la CPU del microcontrolador. Las causas de inicialización son:

- **Reset de Encendido (Power-on Reset):** El MCU es inicializado cuando el voltaje de la fuente está por abajo del voltaje de umbral de encendido (V_{POT}), el cual tiene un valor típico de 2.3 V.
- **Reset Externo:** El MCU es inicializado cuando un nivel bajo está presente en la terminal RESET por un tiempo mayor a 1.5 uS, que es la longitud mínima requerida (t_{RST}).

⁴ JTAG es un acrónimo para **Joint Test Action Group** y es el nombre común de la norma IEEE 1149.1, originalmente para la evaluación de circuitos impresos. Actualmente **JTAG** hace referencia a una interfaz serial utilizada para la prueba de circuitos integrados y como medio para depurar sistemas empotrados.

- **Reset por Watchdog:** El MCU es inicializado cuando se ha habilitado al *Watchdog Timer* y éste se ha desbordado.
- **Reset por reducción de voltaje (Brown out).** Se inicializa al MCU cuando el detector de reducción de voltaje está habilitado y el voltaje de la fuente de alimentación está por debajo del umbral establecido (V_{BOD}). El valor de V_{BOD} es configurable a 2.7 V ó 4.0 V, y el tiempo mínimo necesario (t_{BOD}) para considerar una reducción de voltaje es de 2 uS.
- **Reset por JTAG:** El MCU es inicializado tan pronto como existe un 1 lógico en el Registro de Reset del Sistema JTAG.

Referente a la figura 2.14, los bits BODEN, BODLEVEL, CKSEL y SUT son fusibles que forman parte de los *Bits de Configuración y Seguridad*, su valor se define en el momento en que se programa al dispositivo y no pueden ser modificados en tiempo de ejecución.

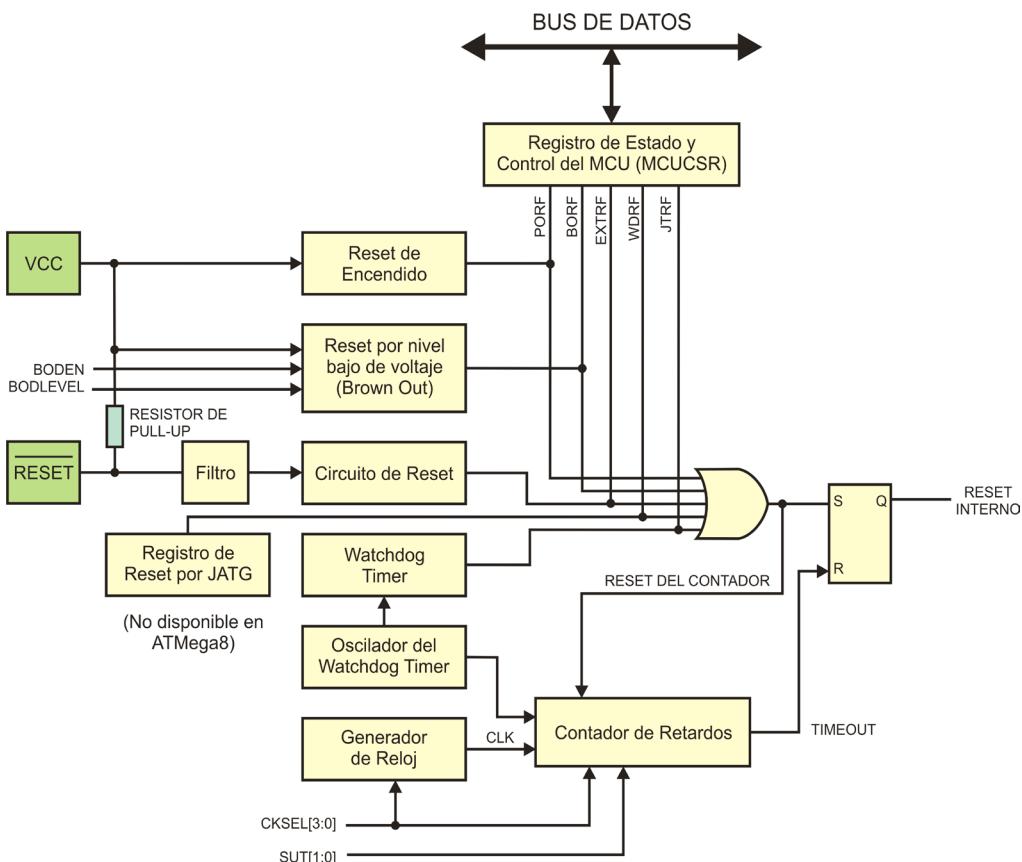


Figura 2.14 Organización del hardware de inicialización o *reset*

Puesto que hay diferentes causas de reinicio, los AVR incluyen al Registro de Estado y Control del MCU (**MCUCSR**, *MCU Control and Status Register*) en el cual queda indicada la causa de *reset* por medio de una bandera. Los bits del registro **MCUCSR** son:

7	6	5	4	3	2	1	0	MCUCSR
0x34	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF

- **Bits 7, 6 y 5:** No tienen relación con el *reset* del sistema, en el ATMega8 no están implementados.
- **Bit 4 – JTRF: Bandera de reinicio por JTAG**
No está implementada en el ATMega8.
- **Bit 3 – WDRF: Bandera de reinicio por desbordamiento del *Watchdog timer***
- **Bit 2 – BORF: Bandera de reinicio por reducción de voltaje (*Brown out*)**
- **Bit 1 – EXTRF: Bandera de reinicio desde la terminal de *reset***
- **Bit 0 – PORF: Bandera de reinicio por encendido**

En la figura 2.15 se muestra la generación del Reset Interno inmediatamente después de que se ha alcanzado el voltaje de encendido (V_{POT}) en la terminal de V_{CC} . La terminal de RESET internamente está conectada a V_{CC} , a través de un resistor de *pull-up*, en la figura 2.15 se observa que su comportamiento es el mismo que el de la terminal V_{CC} .

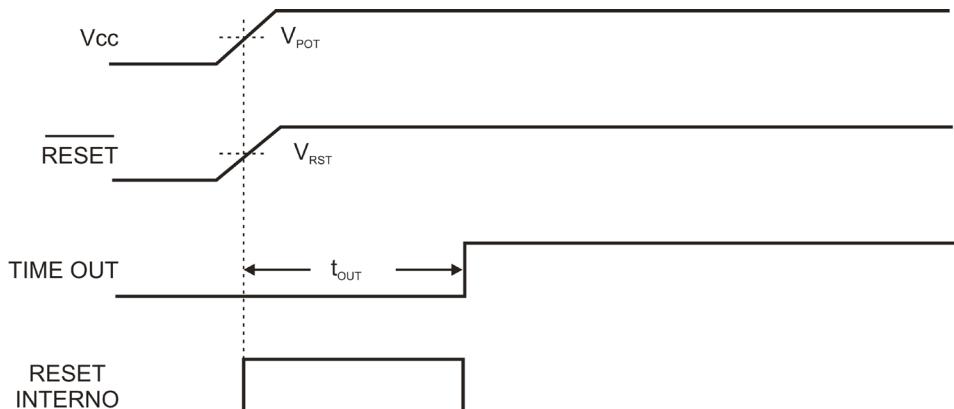


Figura 2.15 Reset de encendido

La duración de la señal Reset Interno es determinada por el módulo Contador de Retardos, que también puede verse en la figura 2.14.

El Reset Interno es generado mientras transcurre el tiempo de establecimiento (t_{OUT}), cuya duración depende de la frecuencia de trabajo y del valor de los fusibles CKSEL y SUT. Los MCUs ATMega8 y ATMega16 son comercializados con un tiempo de establecimiento ajustado a 65 mS.

En la figura 2.16 se muestra la generación del Reset Interno como una respuesta a la terminal RESET, en la cual debe conectarse un botón a tierra para su activación. En (a) el RESET externo está activo al alimentar al dispositivo, por lo que la señal de Reset Interno inicia en alto, el tiempo de establecimiento inicia desde el momento en que la terminal RESET alcanza el umbral de encendido (V_{RST} , valor máximo de 0.9 V) y al finalizar el tiempo de establecimiento la señal de Reset Interno se va a un nivel bajo. En la figura 2.16 (b) se muestra como en cualquier momento puede activarse la terminal externa de RESET, generando nuevamente la señal de Reset Interno.

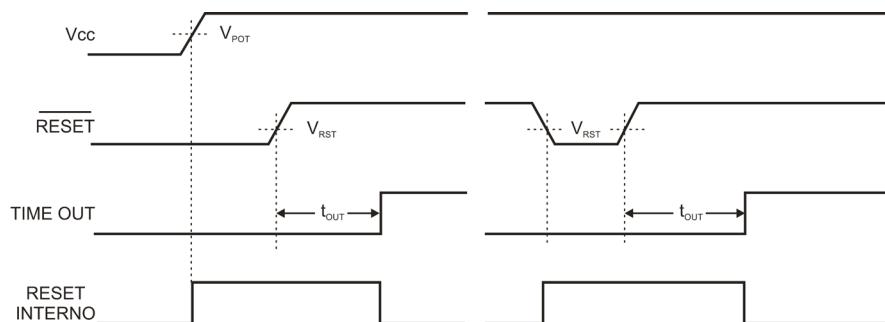


Figura 2.16 Reset externo (a) en el encendido y (b) en cualquier otro instante

En la figura 2.17 se muestra una condición de inicialización debida a una reducción de voltaje (*Brown out*) en la terminal de alimentación (V_{CC}). El Reset Interno se genera cuando V_{CC} cae por debajo del límite inferior de un voltaje de umbral (V_{BOT-}), el tiempo de establecimiento inicia cuando V_{CC} alcanza al límite superior del voltaje de umbral (V_{BOT+}). La señal de Reset Interno concluye al terminar el tiempo de establecimiento. La inicialización por reducción de voltaje se habilita con el fusible BODEN, que es parte de los *Bits de Configuración y Seguridad*, y con BODLEVEL se determina si el límite inferior del umbral es de 2.7 V o de 4.0 V; el voltaje de histéresis (V_{HYST}) es para evitar oscilaciones ante variaciones continuas y su valor es de 1.3 mV.

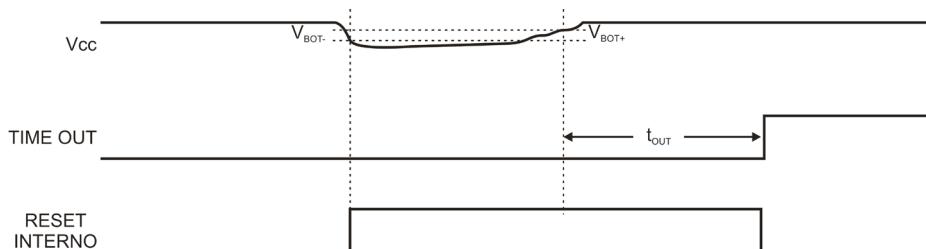


Figura 2.17 Inicialización por reducción de voltaje (*Brown out*)

Cuando el *Watchdog Timer* se desborda, genera un pulso en alto por un ciclo de reloj que es suficiente para provocar una inicialización del MCU, esto se muestra en la figura 2.18, donde sólo se incluyen las señales involucradas. El tiempo de establecimiento inicia inmediatamente después del desbordamiento.

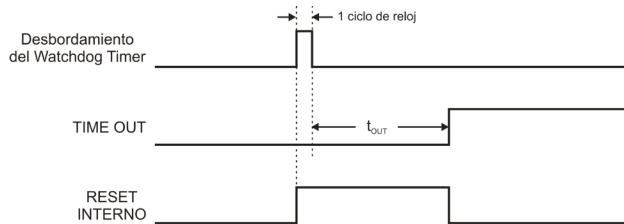


Figura 2.18 Inicialización por desbordamiento del *Watchdog Timer*

2.8 Reloj del Sistema

Para la sincronización de los microcontroladores ATMega8 y ATMega16 se tienen diferentes fuentes de reloj, la selección se realiza por medio de un multiplexor. La fuente de reloj seleccionada es distribuida por medio de un módulo de control del reloj hacia los diferentes módulos del sistema, esto se observa en la figura 2.19.

La selección de la fuente de reloj se realiza por medio de los fusibles CKSEL que son parte de los *Bits de Configuración y Seguridad*, otros fusibles involucrados son CKOPT y SUT. El fusible CKOPT proporciona más alternativas al utilizar un cristal o resonador externo o habilita la conexión de capacitores internos si se selecciona un reloj externo o un cristal de baja frecuencia; mientras que con los fusibles SUT es posible modificar el tiempo de establecimiento después de un reinicio.

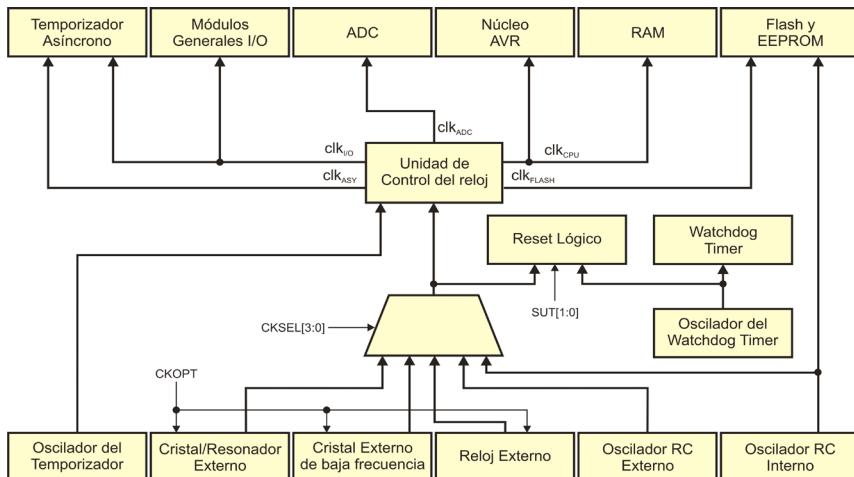


Figura 2.19 Generación y distribución de la señal de reloj

En la tabla 2.6 se indica cuál es la fuente de reloj en función del valor de los fusibles CKSEL, en algunos casos se tiene más de una combinación, porque se tienen diferentes alternativas.

Tabla 2.6 Selección de Reloj a partir de los fusibles CKSEL

Opción para el Reloj del Sistema	CKSEL[3:0]
Resonador Cerámico o Cristal Externo	1010 – 1111
Cristal de Baja Frecuencia Externo	1001
Oscilador RC Externo	0101 – 1000
Oscilador RC Calibrado Interno	0001 – 0100
Reloj Externo	0000

Además de las 5 posibles fuentes de reloj que ingresan al multiplexor, en la figura 2.19 se muestra el módulo del oscilador del temporizador. El módulo está optimizado para trabajar con un oscilador externo de 32.768 KHz, para generar una señal de reloj (clk_{ASY}) disponible para que con el temporizador 2 pueda manejarse un contador de tiempo real, con ello, es posible que el temporizador 2 trabaje a una frecuencia diferente a la del resto del sistema.

La unidad de control del reloj se encarga de generar diferentes señales de reloj y distribuirlas a los diferentes módulos, las señales de reloj son:

- clk_{CPU} : Ruteado al núcleo AVR, incluyendo al archivo de registros, registro de Estado, Memoria de datos, apuntador de pila, etc.
- $\text{clk}_{\text{FLASH}}$: Señal de reloj suministrada a las memorias FLASH y EEPROM.
- clk_{ADC} : Reloj dedicado al ADC, el ADC trabaja a una frecuencia menor que la CPU con el objetivo de reducir el ruido generado por interferencia digital y mejorar las conversiones.
- $\text{clk}_{\text{I/O}}$: Es la señal de reloj utilizada por los principales módulos de recursos: Temporizadores, interfaz SPI y USART. Además de ser requerido por el módulo de interrupciones externas.
- clk_{ASY} : Es una señal de reloj asíncrona, con respecto al resto del sistema, es empleada para sincronizar al temporizador 2, el módulo que genera esta señal está optimizado para ser manejado con un cristal externo de 32.768 KHz. Frecuencia que permite al temporizador funcionar como un contador de tiempo real, aun cuando el dispositivo está en reposo.

En los siguientes apartados se describen las fuentes de reloj y cómo seleccionarlas.

2.8.1 Resonador Cerámico o Cristal Externo

En la tabla 2.6 se presentaron 6 combinaciones de los fusibles CKSEL para esta opción de reloj, de 1010 a 1111. Sin embargo, sólo son 3 opciones porque el bit menos significativo está dedicado a la definición de los tiempos de establecimiento. Las opciones se amplían al utilizar al fusible CKOPT, en la tabla 2.7 se muestran las diferentes opciones con sus rangos de frecuencias. Para la primera opción (CKSEL = ‘101’, CKOPT = ‘1’), por el rango de frecuencias no se deberían usar cristales, sólo resonadores cerámicos.

Tabla 2.7 Opciones para el empleo de un resonador cerámico o cristal externo

CKOPT	CKSEL [3:1]	Rango de Frecuencias (MHz)	Valores recomendados para C1 y C2 (pF)
1	101	0.4 – 0.9	-
1	110	0.9 – 3.0	12 – 22
1	111	3.0 – 8.0	12 – 22
0	101, 110, 111	1.0 ≤	12 – 22

La figura 2.20 muestra como se debe hacer la conexión del cristal o resonador cerámico junto con sus capacitores.

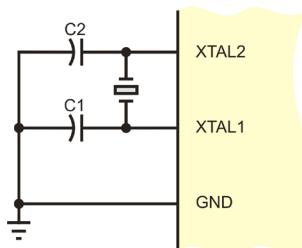


Figura 2.20 Resonador Cerámico o Cristal Externo

Con esta opción de reloj, se tienen los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio, mostrados en la tabla 2.8, en donde ck significa: ciclos de reloj.

2.8.2 Cristal de Baja Frecuencia Externo

Con esta opción para el reloj, el hardware está acondicionado para ser manejado con un cristal de 32.768 KHz, esta frecuencia proporciona la base para un contador de tiempo real. Todo el sistema trabajaría a esta frecuencia, por lo que sólo es conveniente si la aplicación requiere ejecutar instrucciones en periodos que son fracciones de segundos.

Tabla 2.8 Tiempos de establecimiento y retardos al emplear un resonador cerámico o cristal externo

CKSEL0	SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un reset ($V_{cc} = 5$ V)
0	00	258 ck	4.1 mS
0	01	258 ck	65 ms
0	10	1K ck	-
0	11	1K ck	4.1 mS
1	00	1K ck	65 ms
1	01	16K ck	-
1	10	16K ck	4.1 mS
1	11	16K ck	65 ms

Esta opción se selecciona con CKSEL = “1001”. Si se programa al fusible CKOPT se habilitan dos capacitores internos de 36 pF, en caso contrario, deben conectarse dos capacitores externos con la configuración mostrada en la figura 2.20.

Con esta opción para el reloj, los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio son mostrados en la tabla 2.9.

Tabla 2.9 Tiempos de establecimiento y retardos con un cristal de baja frecuencia externo

SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un reset ($V_{cc} = 5$ V)
00	1 K ck	4.1 mS
01	1 K ck	65 ms
10	32 K ck	65 ms
11	Reservado	

2.8.3 Oscilador RC Externo

Esta es una alternativa de bajo costo si se requiere de una frecuencia diferente a las que maneja el oscilador interno y si esta frecuencia no es un factor determinante en el funcionamiento de un sistema, dado que el valor de R y C puede variar por sus tolerancias intrínsecas y en función de la temperatura. El circuito RC se conecta como se muestra en la figura 2.21, la frecuencia se determina como $f = 1/(3RC)$ y el valor de C debe ser por lo menos de 22 pF.

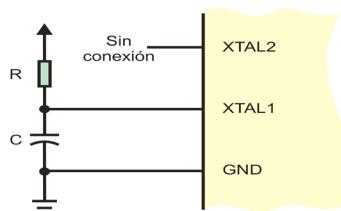


Figura 2.21 Oscilador RC Externo

Con este tipo de oscilador, se tienen 4 combinaciones de CKSEL, debe seleccionarse la que mejor se adapte a la frecuencia de trabajo. En la tabla 2.10 se muestra el rango de frecuencias para cada una de las combinaciones.

Tabla 2.10 Opciones para el empleo de un oscilador RC externo

CKSEL [3:0]	Rango de Frecuencias (MHz)
0101	0.1 – 0.9
0110	0.9 – 3.0
0111	3.0 – 8.0
1000	8.0 – 12.0

Para esta fuente de reloj, los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio se muestran en la tabla 2.11.

Tabla 2.11 Tiempos de establecimiento y retardos con un oscilador RC externo

SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un reset ($V_{cc} = 5$ V)
00	18 ck	-
01	18 ck	4.1 mS
10	18 ck	65 mS
11	6 ck	4.1 mS

2.8.4 Oscilador RC Calibrado Interno

Los microcontroladores ATMega8 y ATMega16 incluyen un oscilador interno que puede ser configurado para trabajar en frecuencias de 1, 2, 4 y 8 MHz. En la tabla 2.12 se muestran las combinaciones de los fusibles CKSEL para utilizar al oscilador interno. Se requiere que el dispositivo esté alimentado con 5 V y a una temperatura de 25 °C.

Tabla 2.12 Opciones para el empleo de un Oscilador RC interno

CKSEL [3:0]	Frecuencia Nominal (MHz)
0001	1.0
0010	2.0
0011	4.0
0100	8.0

Ésta es la mejor alternativa para la mayoría de aplicaciones, puesto que el oscilador ya está incluido en el MCU, con ello se minimiza el número de componentes externos y por lo tanto, el circuito impreso de la aplicación se reduce.

Con esta fuente de reloj, después de algún modo de bajo consumo se tiene un tiempo de establecimiento de 6 ciclos de reloj y después de un reinicio se tienen los retardos mostrados en la tabla 2.13.

Los dispositivos se comercializan con CKSEL = “0001” y SUT = “10”, por lo que inicialmente trabajan a una frecuencia de 1 MHz y con un retardo de 65 mS.

Tabla 2.13 Retardos con un oscilador RC calibrado interno y con un reloj externo

SUT [1:0]	Retardo después de un <i>reset</i> ($V_{cc} = 5$ V)
00	-
01	4.1 mS
10	65 mS
11	Reservado

Sin embargo, el voltaje de alimentación o la temperatura puede variar y esto va a modificar la frecuencia del oscilador interno, la cual también es afectada por las variaciones en los procesos de fabricación de cada dispositivo. Para compensar estas variaciones, en los AVR se ha incorporado un registro para la calibración del oscilador, el registro se denomina **OSCCAL** y sus bits son:



Después de un *reset* en el registro **OSCCAL** se carga automáticamente el valor de calibración para 1 MHz, el cual está en el byte alto de la palabra ubicada en la dirección 0 de la firma del dispositivo. La firma del dispositivo es un conjunto de palabras que puede leerse desde un programador serial o paralelo, los bytes bajos contienen información que identifica al fabricante, el tamaño de memoria y al tipo de dispositivo. En los bytes altos se encuentran los valores de calibración para 1, 2, 4 y 8 MHz, en las direcciones 0, 1, 2 y 3, respectivamente.

Si un dispositivo opera con una frecuencia diferente a 1 MHz, debe cambiarse el valor de calibración. Para ello, se sugiere leer los valores de calibración de la firma del dispositivo e incluirlos como constantes en Flash o en EEPROM. Con estos valores se garantiza un error en la frecuencia menor al 3 %. En las notas de aplicación de Atmel se describen algunas técnicas para calibrar al oscilador en tiempo de ejecución, con las cuales es posible conseguir un error menor al 1 %, a cualquier voltaje y temperatura.

2.8.5 Reloj Externo

El MCU puede ser manejado por un generador de señales, con la combinación de “0000” para CKSEL. La salida del generador debe conectarse en la terminal XTAL1, dejando a la terminal XTAL2 sin conexión, como se muestra en la figura 2.22. Si se programa al fusible CKOPT, se habilita un capacitor de 36 pF entre XTAL1 y GND, con el propósito de eliminar ruido.

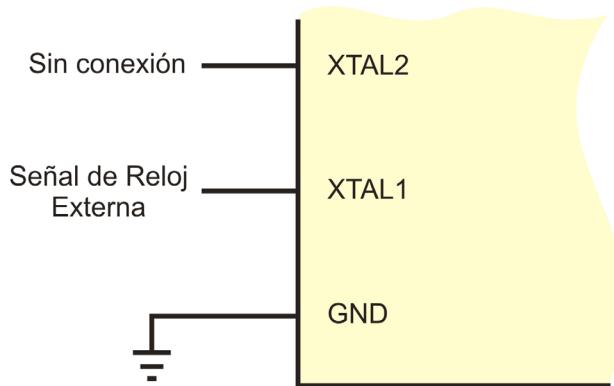


Figura 2.22 Reloj Externo

Con un reloj externo, el tiempo de establecimiento después de algún modo de bajo consumo es de 6 ciclos de reloj y los retardos después de un reinicio corresponden con los de la tabla 2.13.

2.9 Modos de Bajo Consumo de Energía

Los modos de bajo consumo permiten el ahorro de energía al “apagar” módulos de un MCU, cuyo uso no es requerido en un sistema. También se les conoce como modos de reposo y son adecuados si el sistema va a operar con baterías.

En la arquitectura AVR se simplifica el manejo de los modos de bajo consumo por la forma en que distribuye su señal de reloj, de manera que si un módulo no se va a requerir para alguna aplicación, simplemente se anula su señal de reloj, con lo cual el módulo no trabaja y por lo tanto, no consume energía.

Un ATMega8 tiene 5 modos de reposo y un ATMega16 tiene 6. En la tabla 2.14 se listan los modos de bajo consumo con los dominios de reloj que se mantienen activos, las fuentes presentes para el oscilador y los eventos que “despiertan” al MCU, haciendo que abandone el modo en que se encuentre. El último modo de la tabla 2.14 no está disponible en los ATMega8.

Tabla 2.14 Dominios de reloj activos en los diferentes modos de reposo y eventos que lo “despiertan”

Modo de Bajo Consumo de Energía	Reloj Activo					Reloj Principal	Oscilador del temporizador	Eventos que despiertan al MCU				
	clk _{CPU}	clk _{FLASH}	clk _{IO}	clk _{ADC}	clk _{ASY}			INT0, INT1, INT2	Interfaz de dos hilos (TWI)	Temporizador 2	EEPROM, Memoria de programa lista	ADC
Modo ocioso (<i>idle</i>)			X	X	X	X	X	X	X	X	X	X
Reducción de ruido en el ADC			X	X	X	X	X	X	X	X	X	X
Baja potencia								X	X			
Ahorro de potencia				X			X	X	X	X		
Modo de espera (<i>standby</i>)					X			X	X			
Modo de espera extendido				X	X	X	X	X	X			

Los diferentes modos de bajo consumo se describen a continuación:

- **Modo ocioso:** En este modo todos los recursos del MCU trabajan, pero la CPU no ejecuta instrucciones porque no tienen señal de reloj. El suministro del reloj principal y del oscilador del temporizador están activos. El hecho de mantener los recursos activos hace que, prácticamente cualquier evento de los diferentes recursos provoque una salida del modo de reposo.
- **Modo de reducción de ruido en el ADC:** En este modo únicamente trabaja el ADC y el oscilador asíncrono para el temporizador 2. Ambos suministros de reloj están activos, por ello, este modo es adecuado para aplicaciones que requieren el monitoreo de parámetros analógicos en períodos preestablecidos de tiempo. La salida del modo es posible por eventos en los recursos principales
- **Modo de baja potencia:** En este modo no hay reloj en los módulos de recursos y tampoco están activas las fuentes de oscilación, por lo tanto, es el modo con el menor consumo de energía. El MCU puede ser reactivado por eventos en la interfaz de dos hilos o por las interrupciones externas. Una posible aplicación para este modo es un control remoto, porque casi siempre está inactivo, sólo cuando se presiona una tecla del control remoto el MCU se despierta, emite el código seleccionado y regresa al modo de reposo.
- **Modo de ahorro de potencia:** En este modo sólo se tiene al reloj asíncrono, con su correspondiente oscilador. Por lo que prácticamente sólo se mantiene activo el temporizador 2, sincronizado con una fuente de reloj externa. Esto hace al modo ideal para aplicaciones que involucren un reloj de tiempo real. La salida del modo es posible con eventos en la interfaz de dos hilos, interrupciones externas o del temporizador 2.

- **Modo de espera:** Este modo es muy similar al modo de baja potencia, la única diferencia es que en este modo se mantiene el suministro del reloj principal. Con ello, el dispositivo despierta en 6 ciclos de reloj.
- **Modo de espera extendido:** Este modo es muy similar al modo de ahorro de potencia, la única diferencia es que en este modo también está activo el suministro del reloj principal. Este modo no está disponible en el ATMega8.

Para seleccionar uno de los modos debe configurarse al registro de Control del MCU (**MCUCR**, *MCU Control Register*), los bits de este registro son:

0x35	7	6	5	4	3	2	1	0	MCUCR
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	

- **Bit 7 – SE: Habilitador del modo de reposo (SE, Sleep Enable)**

Este bit se pone en alto después de ingresar a cualquiera de los modos de reposo, lo cual se consigue con la ejecución de la instrucción **SLEEP**, pero antes, debe seleccionarse el modo de reposo deseado.

- **Bits 6 al 4 – SM[2:0]: Bits de Selección del modo de reposo**

En la tabla 2.15 se muestra la combinación de estos bits para seleccionar los diferentes modos de bajo consumo de energía.

- **Bits 3 al 0 – No están relacionados con los modos de reposo**

Los modos de espera sólo pueden ser empleados con resonadores cerámicos o cristales externos. La combinación “111” de la tabla 2.15 está sin uso en un ATMega8, porque no cuenta con el modo de espera extendido. En un ATMega16, la posición de los bits **SE** y **SM2** está intercambiada, **SM2** es el bit 7 y **SE** está en la posición 6.

Tabla 2.15 Selección del modo de reposo

SM2	SM1	SM0	Modo de reposo
0	0	0	Modo ocioso (<i>idle</i>)
0	0	1	Reducción de ruido en el ADC
0	1	0	Baja potencia
0	1	1	Ahorro de potencia
1	0	0	Reservado
1	0	1	Reservado
1	1	0	Modo de espera (<i>standby</i>)
1	1	1	Modo de espera extendido

2.10 Ejercicios

1. Complemente la tabla 2.16 con las características de los microcontroladores bajo estudio.

Tabla 2.16 Características de microcontroladores

	ATMega8	ATMega16
Cantidad de Memoria de Código:		
Cantidad de RAM (de propósito general):		
Cantidad de EEPROM:		
Registros de Propósito General:		
Registros I/O:		

2. Explique las ventajas de que el núcleo, en los AVR, trabaje en una segmentación de 2 etapas.
3. ¿Por qué se dice que los AVR están optimizados para trabajar con código C compilado?
4. Complemente la tabla 2.17 con el estado de las banderas, después de la ejecución de algunas instrucciones.

Tabla 2.17 Estado de las banderas

LDI R16, 0x5A	LDI R16, 0x4E	LDI R16, 0xA5
LDI R17, 0x93	LDI R17, 0xB2	LDI R17, 0x9B
ADD R16, R17	ADD R16, R17	ADD R16, R17
Z =	H =	V =

5. Explique para qué se emplean los Registros I/O en los microcontroladores AVR.
6. Indique qué Registros I/O se emplean para el acceso a la EEPROM y el objetivo de cada uno de ellos.
7. Muestre la organización de una terminal en un puerto y explique la funcionalidad de cada uno de los registros que se requieren para su manejo.
8. Defina qué es una interrupción y explique por qué es conveniente el uso de interrupciones.
9. Los microcontroladores AVR pueden ser operados por una gama muy amplia de osciladores ¿Por qué es conveniente incorporar esta flexibilidad y no únicamente manejarlos con un solo tipo de oscilador?
10. Describa tres fuentes de inicialización (o *reset*).
11. Explique el objetivo de los modos de bajo consumo de energía e indique cómo ingresar y salir de cualquiera de estos modos de operación.

3. Programación de los Microcontroladores

En este capítulo se describen los aspectos relacionados con la programación de los microcontroladores, se revisa el repertorio de instrucciones en ensamblador, los modos de direccionamiento, aspectos para programar en lenguaje C y la forma en que se puede vincular al lenguaje ensamblador con el lenguaje C.

3.1 Repertorio de Instrucciones

Aunque la arquitectura es tipo RISC, el repertorio de instrucciones es de un tamaño considerable, para el ATMega8 se tienen 128 instrucciones y para el ATMega16 son 131. Por su funcionalidad, las instrucciones están organizadas en 5 grupos:

- Instrucciones aritméticas y lógicas (28)
- Instrucciones para el control de flujo (Bifurcaciones) (34 en el ATMega8 y 36 en el ATMega16)
- Instrucciones de transferencia de datos (35)
- Instrucciones para el manejo de bits (28)
- Instrucciones especiales (3 en el ATMega8 y 4 en el ATMega16)

La mayoría de las instrucciones son de 16 bits. En las siguientes secciones se presentan algunas características propias de cada uno de los diferentes grupos de instrucciones, el apéndice B contiene una referencia rápida del repertorio completo.

3.1.1 Instrucciones Aritméticas y Lógicas

Este grupo incluye instrucciones para las operaciones básicas de suma y resta, con diferentes variantes, las cuales se muestran en la tabla 3.1.

Tabla 3.1 Instrucciones para las operaciones básicas de suma y resta

Instrucción	Descripción	Operación	Banderas
ADD Rd, Rs	Suma sin acarreo	$Rd = Rd + Rs$	Z,C,N,V,H,S
ADC Rd, Rs	Suma con acarreo	$Rd = Rd + Rs + C$	Z,C,N,V,H,S
ADIW Rd, k	Suma constante a palabra	$[Rd + 1:Rd] = [Rd + 1:Rd] + k$	Z,C,N,V,S
SUB Rd, Rs	Resta sin acarreo	$Rd = Rd - Rs$	Z,C,N,V,H,S
SUBI Rd, k	Resta constante	$Rd = Rd - k$	Z,C,N,V,H,S
SBC Rd, Rs	Resta con acarreo	$Rd = Rd - Rs - C$	Z,C,N,V,H,S
SBCI Rd, k	Resta constante con acarreo	$Rd = Rd - k - C$	Z,C,N,V,H,S
SBIW Rd, k	Resta constante a palabra	$[Rd + 1:Rd] = [Rd + 1:Rd] - k$	Z,C,N,V,S

Todas las instrucciones de suma y resta modifican las banderas de cero (Z), acarreo (C), negado(N), sobreflujo(V) y signo(S). Exceptuando a las instrucciones que operan

sobre palabras, el resto también modifica la bandera de acarreo del nibble menos significativo (**H**). Las instrucciones que operan sobre datos de 8 bits se ejecutan en 1 ciclo de reloj, las otras en 2 ciclos de reloj.

Las sumas y restas de 8 bits son entre registros, con o sin acarreo. No hay una suma de 8 bits con una constante, pero si se requiere, puede hacerse una resta con el negado de la constante que se desea sumar, si hay una resta con una constante de 8 bits, por ejemplo, para sumar 10 a R17 puede emplearse la instrucción **SUBI R17, -10**. Por el formato de las instrucciones, el cual se revisa en la siguiente sección, con los modos de direccionamiento, las instrucciones que involucran constantes sólo son aplicables a los registros R16 a R31.

Para la suma y resta de 16 bits, el segundo operando es una constante entre 0 y 63. La palabra de 16 bits se forma por 2 registros consecutivos entre R24 y R31, en la instrucción puede especificarse al registro menos significativo del par (**ADIW R26, k**), a ambos registros (**ADIW R27:R26, k**) o si es aplicable, el nombre de un registro de 16 bits (**ADIW X, k**).

El grupo también cuenta con instrucciones aritméticas más complejas, como multiplicaciones entre enteros y fracciones, éstas se listan en la tabla 3.2, todas se ejecutan en 2 ciclos de reloj.

Tabla 3.2 Instrucciones para las operaciones de multiplicación

Instrucción	Descripción	Operación	Banderas
MUL Rd, Rs	Multiplicación sin signo	R1:R0 = Rd * Rs	Z,C
MULS Rd, Rs	Multiplicación con signo	R1:R0 = Rd * Rs	Z,C
MULSU Rd, Rs	Multiplicación de un número con signo y otro sin signo	R1:R0 = Rd * Rs	Z,C
FMUL Rd, Rs	Multiplicación fraccional sin signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULS Rd, Rs	Multiplicación fraccional con signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULSU Rd, Rs	Multiplicación fraccional de un número con signo y otro sin signo	R1:R0 = (Rd * Rs) << 1	Z,C

Puesto que los operandos son de 8 bits, el resultado a lo más es de 16 bits. Todas las instrucciones dejan el resultado en R1 y R0 (en R1 la parte alta). Para las instrucciones con enteros, los operandos pueden ser interpretados como números sin signo o con signo. En la tabla 3.3 se muestran ejemplos que ilustran la diferencia entre instrucciones.

Tabla 3.3 Ejemplos de multiplicaciones con enteros

R16	R17	Instrucción	R1:R0	Valores en Decimal		
				R16	R17	R1:R0
0000 0010	1111 1111	MUL R16, R17	0000 0001 1111 1110	2	255	510
0000 0010	1111 1111	MULS R16, R17	1111 1111 1111 1110	2	-1	-2
0000 0010	1111 1111	MULSU R16, R17	0000 0001 1111 1110	2	255	510
1111 1111	1111 1111	MUL R16, R17	1111 1110 0000 0001	255	255	65, 025
1111 1111	1111 1111	MULS R16, R17	0000 0000 0000 0001	-1	-1	1
1111 1111	1111 1111	MULSU R16, R17	1111 1111 0000 0001	-1	255	-255

En la tabla 3.3 se observa como las instrucciones interpretan a los operandos de diferentes maneras. El resultado debe ser interpretado de acuerdo con el tipo de instrucción. Para la instrucción **MULSU** el primer operando corresponde con un número con signo, el segundo como un número sin signo y para que el resultado sea correcto, debe interpretarse como un número con signo.

La multiplicación fraccionaria utiliza una notación de punto fijo, bajo un esquema de 1.7 para los operandos y 1.15 para el resultado. En los operandos se tiene 1 dígito para la parte entera y 7 para la parte fraccionaria. Lo natural sería que $1.7 \times 1.7 = 2.14$, por eso es que en la tabla 3.2 aparece un desplazamiento a la izquierda, al desplazar el punto decimal el resultado queda en 1.15.

Para comprender la notación se tiene el siguiente ejemplo:

$$1110\ 0000_2 = 1.11_2 = 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

No obstante, si el número anterior es interpretado como un número con signo, el número es negativo porque su bit más significativo es 1, para obtener la magnitud se obtiene el complemento a 2, que corresponde con: $0001\ 1111 + 1 = 0010\ 0000$. La magnitud es:

$$0010\ 0000_2 = 0.01_2 = 1 \times 2^{-2} = 0.25$$

Por lo tanto, el número $1110\ 0000_2$ también puede corresponder con - 0.25, dependiendo de la instrucción que se utilice.

En la tabla 3.4 se tienen algunos ejemplos del uso de las instrucciones de multiplicación fraccionaria, considerando números sin signo y con signo, con sus interpretaciones en decimal.

Tabla 3.4 Ejemplos de multiplicaciones con fracciones en punto fijo

R16	R17	Instrucción	R1:R0	Valores en Decimal		
				R16	R17	R1:R0
0100 0000	1100 0000	FMUL R16, R17	0110 0000 0000 0000	0.5	1.5	0.75
0100 0000	1100 0000	FMULS R16, R17	1110 0000 0000 0000	0.5	-0.5	-0.25
0100 0000	1100 0000	FMULSU R16, R17	0110 0000 0000 0000	0.5	1.5	0.75
1100 0000	1100 0000	FMUL R16, R17	0010 0000 0000 0000 ¹	1.5	1.5	2.25 ¹
1100 0000	1100 0000	FMULS R16, R17	0010 0000 0000 0000	-0.5	-0.5	0.25
1100 0000	1100 0000	FMULSU R16, R17	1010 0000 0000 0000	-0.5	1.5	-0.75

En el 4º renglón de la tabla (nota¹) el resultado en binario parece incorrecto, esto se debe a que el 2 no puede representarse con la notación 1.15, por lo que queda un 0 en la parte entera y se genera un bit de acarreo ($2 = 10_2$). Para la instrucción **FMULSU** ocurre algo similar que en la versión de enteros (**MULSU**) en cuanto a la interpretación de los números, el primer operando corresponde con un número con signo, el segundo como un número sin signo y el resultado es un número con signo.

En este grupo también se cuenta con las instrucciones lógicas binarias que se muestran en la tabla 3.5 (requieren 2 operandos). Estas instrucciones se aplican bit a bit, se ejecutan en 1 solo ciclo de reloj y modifican a las banderas **Z**, **N**, **V** y **S**.

La **EOR** (OR exclusiva) sólo puede aplicarse entre registros, las operaciones **AND** y **OR** además pueden aplicarse entre un registro con una constante.

Tabla 3.5 Instrucciones lógicas binarias

Instrucción	Descripción		Operación
AND	Rd, Rs	Operación lógica AND	$Rd = Rd \text{ AND } Rs$
ANDI	Rd, k	Operación lógica AND con una constante	$Rd = Rd \text{ AND } k$
OR	Rd, Rs	Operación lógica OR	$Rd = Rd \text{ OR } Rs$
ORI	Rd, k	Operación lógica OR con una constante	$Rd = Rd \text{ OR } k$
EOR	Rd, Rs	Operación lógica OR Exclusiva	$Rd = Rd \text{ XOR } Rs$

Las operaciones lógicas por lo general se utilizan para enmascarar la información, una máscara consiste en modificar algunos bits de un byte, poniéndolos en alto o en bajo, sin modificar al resto. Las máscaras usualmente se realizan con operaciones **AND** y **OR**, pero por su extenso uso, el grupo incluye dos instrucciones dedicadas, las cuales se muestran en la tabla 3.6, en donde una constante especifica los bits a modificar en un registro. Estas instrucciones se ejecutan en 1 ciclo de reloj y modifican a las banderas **Z**, **C**, **N**, **V** y **S**; aunque por su enfoque, el valor de las banderas generalmente es ignorado.

Tabla 3.6 Instrucciones para enmascarar información

Instrucción	Descripción		Operación
SBR	Rd, k	Pone en alto los bits indicados en la constante	$Rd = Rd \text{ OR } k$
CBR	Rd, k	Pone en bajo los bits indicados en la constante	$Rd = Rd \text{ AND } (0xFF - k)$

El grupo se complementa con 7 instrucciones unarias (1 operando) con funciones diversas, las cuales se muestran en la tabla 3.7. Todas se ejecutan en 1 ciclo de reloj y las banderas que modifican difieren entre instrucciones.

Tabla 3.7 Instrucciones aritméticas o lógicas unarias

Instrucción	Descripción		Operación	Banderas
COM	Rd	Complemento a 1	$Rd = 0xFF - Rd$	Z,C,N,V,S
NEG	Rd	Negado o complemento a 2	$Rd = 0x00 - Rd$	Z,C,N,V,H,S
INC	Rd	Incrementa un registro	$Rd = Rd + 1$	Z,N,V,S
DEC	Rd	Disminuye un registro	$Rd = Rd - 1$	Z,N,V,S
TST	Rd	Evalúa un registro	$Rd = Rd \text{ AND } Rd$	Z,C,N,V,S
CLR	Rd	Limpia un registro (pone en bajo)	$Rd = 0x00$	Z,C,N,V,S
SER	Rd	Ajusta un registro (pone en alto)	$Rd = 0xFF$	Ninguna

La instrucción **TST** no modifica al registro destino, puede usarse para evaluar si el registro es cero, evaluando la bandera **Z** después de su ejecución.

3.1.2 Instrucciones para el Control de Flujo

Durante la ejecución de un programa, el control de flujo se cambia al modificar el contenido del contador del programa (registro PC), para ello, este grupo cuenta con diversas instrucciones: saltos incondicionales, condicionales, llamadas y retornos de rutinas. En la tabla 3.8 se muestran las instrucciones de saltos incondicionales, las cuales no modifican banderas.

Tabla 3.8 Saltos incondicionales

Instrucción	Descripción	Operación
RJMP k	Salto relativo	$PC = PC + k + 1$
IJMP	Salto indirecto	$PC = Z$
JMP k	Salto absoluto	$PC = k$

Los saltos relativo e indirecto se ejecutan en 2 ciclos de reloj, el salto absoluto se ejecuta en 3 ciclos. El salto absoluto no está implementado en el ATMega8, no se requiere debido a que un salto relativo es suficiente para direccionar todo su espacio.

La instrucción **RJMP** es relativa con respecto a $PC + 1$, a partir de ahí se bifurca hacia adelante o atrás, utilizando una constante positiva o negativa. Los saltos relativos permiten crear código “reubicable”, es decir, código que puede moverse a cualquier dirección, sin necesidad de hacer ajustes en la dirección destino, por lo tanto, las rutinas que sólo incluyen bifurcaciones relativas pueden ser compartidas como código máquina. Esto no se puede hacer con saltos absolutos, dado que hacen referencia a una dirección específica. Los saltos indirectos están enfocados a tablas de saltos, las cuales pueden resultar de codificar una estructura del tipo *switch-case*, en lenguaje C.

Relacionado con el manejo de rutinas, se tienen las instrucciones que se muestran en la tabla 3.9. Para las llamadas se tienen 3 instrucciones bajo el mismo esquema que los saltos incondicionales, incluyendo una llamada absoluta (**CALL**) que no está en el ATMega8.

Tabla 3.9 Instrucciones para el manejo de rutinas

Instrucción	Descripción	Operación
RCALL k	Llamada relativa a una rutina	$PC = PC + k + 1$, PILA $\leftarrow PC + 1$
ICALL	Llamada indirecta a una rutina	$PC = Z$, PILA $\leftarrow PC + 1$
CALL k	Llamada absoluta a una rutina	$PC = k$, PILA $\leftarrow PC + 1$
RET	Retorno de una rutina	$PC \leftarrow PILA$
RETI	Retorno de rutina de interrupción	$PC \leftarrow PILA$, I = 1

La llamada a una rutina también es un salto incondicional, con el agregado de que en la pila se almacena el valor de $PC + 1$, el cual corresponde con la dirección de la instrucción que sigue a la llamada, esta dirección se recupera de la pila y se escribe en el PC con la instrucción **RET** o **RETI**.

La instrucción **RETI** es el retorno de una ISR (en la sección 2.6 se describió el funcionamiento de las interrupciones), por lo tanto, con esta instrucción también se pone en alto al bit **I** del registro de estado (**SREG**), para habilitar nuevamente al sistema de interrupciones.

Las llamadas relativa e indirecta a rutinas se ejecutan en 3 ciclos de reloj, la llamada absoluta y los retornos se ejecutan en 4. Esta cantidad de ciclos de ejecución se debe a que requieren de un acceso a SRAM por los respaldos y recuperaciones en la pila de datos y un ajuste en el apuntador de pila (registro **SP**).

Dentro del grupo se encuentran 3 instrucciones para la comparación de datos, éstas se muestran en la tabla 3.10. Aunque no modifican el flujo de ejecución, modifican las banderas **Z**, **C**, **N**, **V**, **H** y **S**, en el registro estado, y el valor de estas banderas es la base para las instrucciones de brincos condicionales que se muestran en la tabla 3.11.

Tabla 3.10 Instrucciones para comparar datos

Instrucción		Descripción	Operación
CP	Rd, Rs	Compara dos registros	Rd – Rs
CPC	Rd, Rs	Compara registros con acarreo	Rd – Rs – C
CPI	Rd, k	Compara un registro con una constante	Rd – k

Las instrucciones de la tabla 3.10 realizan la comparación con base en una resta, pero sin almacenar el resultado. Todas se ejecutan en 1 ciclo de reloj. En los programas es frecuente encontrar una instrucción de comparación seguida de un salto condicional (tabla 3.11).

Tabla 3.11 Instrucciones de brincos condicionales

Instrucción		Descripción	Operación
BRBS	s, k	Brinca si el bit s del registro estado está en alto	si(SREG(s) == 1) PC = PC + k + 1
BRBC	s, k	Brinca si el bit s del registro estado está en bajo	si (SREG(s) == 0) PC = PC + k + 1
BRIE	k	Brinca si las interrupciones están habilitadas	si (I == 1) PC = PC + k + 1
BRID	k	Brinca si las interrupciones están inhabilitadas	si (I == 0) PC = PC + k + 1
BRTS	k	Brinca si el bit T está en alto	si (T == 1) PC = PC + k + 1
BRTC	k	Brinca si el bit T está en bajo	si (T == 0) PC = PC + k + 1
BRHS	k	Brinca si hubo acarreo del nibble bajo al alto	si (H == 1) PC = PC + k + 1
BRHC	k	Brinca si no hubo acarreo del nibble bajo al alto	si (H == 0) PC = PC + k + 1
BRGE	k	Brinca si es mayor o igual que (con signo)	si (S == 0) PC = PC + k + 1
BRLT	k	Brinca si es menor que (con signo)	si (S == 1) PC = PC + k + 1
BRVS	k	Brinca si hubo sobreflugo aritmético	si (V == 1) PC = PC + k + 1
BRVC	k	Brinca si no hubo sobreflugo aritmético	si (V == 0) PC = PC + k + 1
BRMI	k	Brinca si es negativo	si (N == 1) PC = PC + k + 1
BRPL	k	Brinca si no es negativo	si (N == 0) PC = PC + k + 1
BREQ	k	Brinca si los datos son iguales	si (Z == 1) PC = PC + k + 1
BRNE	k	Brinca si los datos no son iguales	si (Z == 0) PC = PC + k + 1
BRSH	k	Brinca si es mayor o igual	si (C == 0) PC = PC + k + 1
BRLO	k	Brinca si es menor	si (C == 1) PC = PC + k + 1
BRCS	k	Brinca si hubo acarreo	si (C == 1) PC = PC + k + 1
BRCC	k	Brinca si no hubo acarreo	si (C == 0) PC = PC + k + 1

Los brincos condicionales pueden tardar 1 ó 2 ciclos de reloj en su ejecución, 1 ciclo si el brinco no se realizó, porque la siguiente instrucción ya está en la etapa de captura y 2 ciclos cuando el brinco se realiza, porque se debe anular a la instrucción que está en la etapa de captura y continuar con la siguiente después del brinco. Estos brincos son relativos al PC, es decir, la constante de la instrucción que se suma al PC + 1 puede ser positiva o negativa, para bifurcar hacia adelante o atrás de la instrucción del brinco.

En la tabla 3.11 hay 20 instrucciones y todas evalúan los bits del registro de Estado (**SREG**), sin embargo, sólo en las 2 primeras se especifica el número del bit a evaluar (bit s, entre 0 y 7), ya que en las siguientes 18 el número del bit queda implícito en la instrucción, podría decirse que son versiones particularizadas de las primeras 2 instrucciones. El nombre y descripción de cada instrucción está relacionado con la aplicación que se le puede dar a cada bandera, por eso se tienen 2 versiones para la bandera de acarreo. Estas 18 instrucciones se ordenaron de acuerdo con la disposición de los bits en el registro Estado (del bit más significativo al menos significativo).

Otras instrucciones para bifurcaciones condicionales se muestran en la tabla 3.12, la diferencia es que estas instrucciones son para saltos pequeños, o “saltitos”, que sólo excluyen a la instrucción siguiente cuando la condición se cumple. Por lo general, estas instrucciones están seguidas por un salto incondicional, para complementar el control de flujo en un programa.

Tabla 3.12 Instrucciones para saltitos condicionales

Instrucción		Descripción	Operación
CPSE	Rd, Rs	Un saltillo si los registros son iguales	si(Rd == Rs) PC = PC + 2 ó 3
SBRS	Rs, b	Un saltillo si el bit b del registro Rs está en alto	si(Rs(b) == 1) PC = PC + 2 ó 3
SBRC	Rs, b	Un saltillo si el bit b del registro Rs está en bajo	si(Rs(b) == 0) PC = PC + 2 ó 3
SBIS	P, b	Un saltillo si el bit b del registro P está en alto, P es un Registro I/O	si(I/O(P, b) == 1) PC = PC + 2 ó 3
SBIC	P, b	Un saltillo si el bit b del registro P está en bajo, P es un Registro I/O	si(I/O(P, b) == 0) PC = PC + 2 ó 3

Si la condición es verdadera, las instrucciones pueden incrementar al PC con 2 ó 3, esto porque se desconoce si la instrucción que se omite ocupa 1 ó 2 palabras de 16 bits. Por la misma razón, para su ejecución pueden requerirse 2 ó 3 ciclos de reloj, dado que se debe evaluar si ya es una nueva instrucción o si aún es parte de la que intenta omitirse. Si la condición no se cumple, se continúa con la siguiente instrucción (PC + 1) y por lo tanto, sólo se requiere de 1 ciclo de reloj.

Las últimas 2 instrucciones son frecuentemente utilizadas para monitorear los puertos o para evaluar el estado de otros recursos, dado que todos los recursos son manejados a través de los Registros I/O (sección 2.4.1.1).

3.1.3 Instrucciones de Transferencia de Datos

El grupo incluye instrucciones para diferentes tipos de transferencias. Puesto que la arquitectura es del tipo Registro-Registro, las transferencias están centradas en los 32 registros de propósito general. En la figura 3.1 se ilustra gráficamente este esquema, en donde se puede observar que, por ejemplo, si se quiere llevar una constante a memoria, primero se debe ubicar la constante en uno de los registros, para luego transferir de registro a memoria.

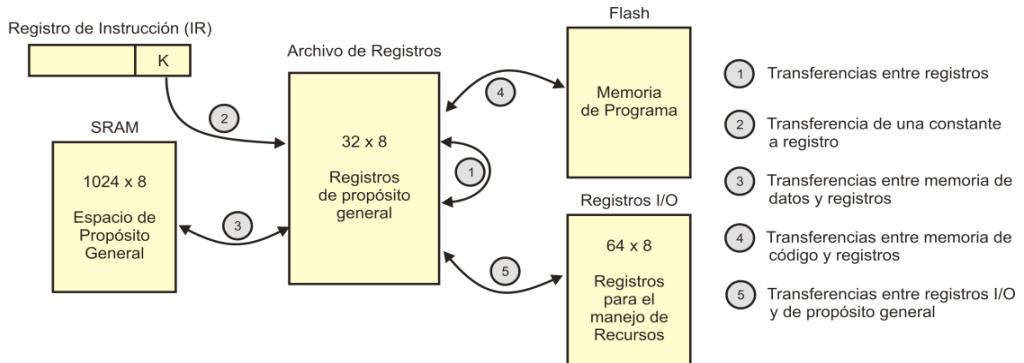


Figura 3.1 Transferencias de datos permitidas en la arquitectura AVR

Las transferencias de datos no modifican las banderas del registro Estado. Para transferencias entre registros sólo se tienen las 2 instrucciones mostradas en la tabla 3.13, una para mover un byte y la otra para mover una palabra de 16 bits. En las transferencias de palabras, para cada operando, se debe especificar un registro con número par, haciendo referencia a éste, como byte menos significativo, y al siguiente. Ambas instrucciones se ejecutan en 1 ciclo de reloj.

Tabla 3.13 Transferencias entre registros

Instrucción		Descripción	Operación
MOV	Rd, Rs	Copia un registro	$Rd = Rs$
MOVW	Rd, Rs	Copia un par de registros	$Rd + 1: Rd = Rs + 1: Rs, Rd y Rs$ registros pares

Sólo se requiere de una instrucción para transferir una constante a un registro, la constante es parte de la instrucción por lo que se toma del registro IR. La instrucción se muestra en la tabla 3.14 y su ejecución requiere sólo de 1 ciclo de reloj.

Tabla 3.14 Transferencia de una constante a un registro

Instrucción		Descripción	Operación
LDI	Rd, k	Copia la constante en el registro	$Rd = k$

Para transferencias entre la memoria de datos y los registros se tiene una gama amplia de instrucciones, a la transferencia de memoria a registro se le conoce como una carga y a la transferencia de registro a memoria se le conoce como un almacenamiento.

En la figura 2.10 se ilustraron estas transferencias. En la tabla 3.15 se muestran las instrucciones para cargas y almacenamientos, en ambos casos se tiene sólo 1 instrucción que utiliza direccionamiento directo, es decir, en la instrucción se especifica la dirección de la localidad a la que se tiene acceso, y 11 instrucciones con direccionamiento indirecto, empleando a uno de los registros de 16 bits (X, Y o Z) como apuntador, en la instrucción se especifica al apuntador.

La variedad de instrucciones se debe a que además de la transferencia, en la mayoría de los casos se modifica al apuntador. Todas las transferencias con SRAM, cargas o almacenamientos, directas o indirectas, requieren de 2 ciclos de reloj para su ejecución.

Tabla 3.15 Transferencias entre memoria de datos y registros (Cargas y Almacenamientos)

Instrucción	Descripción	Operación
LDS Rd, k	Carga directa de memoria	$Rd = \text{Mem}[k]$
LD Rd, X	Carga indirecta de memoria	$Rd = \text{Mem}[X]$
LD Rd, X+	Carga indirecta con post-incremento	$Rd = \text{Mem}[X], X = X + 1$
LD Rd, -X	Carga indirecta con pre-decremento	$X = X - 1, Rd = \text{Mem}[X]$
LD Rd, Y	Carga indirecta de memoria	$Rd = \text{Mem}[Y]$
LD Rd, Y+	Carga indirecta con post-incremento	$Rd = \text{Mem}[Y], Y = Y + 1$
LD Rd, -Y	Carga indirecta con pre-decremento	$Y = Y - 1, Rd = \text{Mem}[Y]$
LDD Rd, Y + q	Carga indirecta con desplazamiento	$Rd = \text{Mem}[Y + q]$
LD Rd, Z	Carga indirecta de memoria	$Rd = \text{Mem}[Z]$
LD Rd, Z+	Carga indirecta con post-incremento	$Rd = \text{Mem}[Z], Z = Z + 1$
LD Rd, -Z	Carga indirecta con pre-decremento	$Z = Z - 1, Rd = \text{Mem}[Z]$
LDD Rd, Z + q	Carga indirecta con desplazamiento	$Rd = \text{Mem}[Z + q]$
STS k, Rs	Almacenamiento directo en memoria	$\text{Mem}[k] = Rs$
ST X, Rs	Almacenamiento indirecto en memoria	$\text{Mem}[X] = Rs$
ST X+, Rs	Almacenamiento indirecto con post-incremento	$\text{Mem}[X] = Rs, X = X + 1$
ST -X, Rs	Almacenamiento indirecto con pre-decremento	$X = X - 1, \text{Mem}[X] = Rs$
ST Y, Rs	Almacenamiento indirecto en memoria	$\text{Mem}[Y] = Rs$
ST Y+, Rs	Almacenamiento indirecto con post-incremento	$\text{Mem}[Y] = Rs, Y = Y + 1$
ST -Y, Rs	Almacenamiento indirecto con pre-decremento	$Y = Y - 1, \text{Mem}[Y] = Rs$
STD Y + q, Rs	Almacenamiento indirecto con desplazamiento	$\text{Mem}[Y + q] = Rs$
ST Z, Rs	Almacenamiento indirecto en memoria	$\text{Mem}[Z] = Rs$
ST Z+, Rs	Almacenamiento indirecto con post-incremento	$\text{Mem}[Z] = Rs, Z = Z + 1$
ST -Z, Rs	Almacenamiento indirecto con pre-decremento	$Z = Z - 1, \text{Mem}[Z] = Rs$
STD Z + q, Rs	Almacenamiento indirecto con desplazamiento	$\text{Mem}[Z + q] = Rs$

En el grupo hay otras dos instrucciones que también hacen transferencias entre registros y SRAM, éstas no se incluyeron en la tabla 3.15 porque están enfocadas a trabajar en el espacio destinado para la Pila, para insertar o extraer un dato en el tope de la pila, el cual está referido por el registro **SP**, que es el apuntador de pila. En la tabla 3.16 se muestran las instrucciones de acceso a la Pila, las cuales también se ejecutan en 2 ciclos de reloj.

Tabla 3.16 Transferencias entre memoria de datos y registros (acceso a la pila)

Instrucción	Descripción		Operación
PUSH Rs	Inserta a Rs en la pila		Mem[SP] = Rs, SP = SP – 1
POP Rd	Extrae de la pila y coloca en Rd		SP = SP + 1, Rd = Mem[SP]

Las transferencias con la memoria de código incluyen tanto cargas como almacenamientos. Las cargas son muy frecuentes, dado que en una aplicación conviene utilizar a la memoria de código para todos aquellos datos constantes, liberando con ello espacio en la SRAM. Se tienen 3 instrucciones de carga, se ejecutan en 3 ciclos de reloj y todas usan al registro Z como apuntador (direcciónamiento indirecto).

Los almacenamientos son poco usados en aplicaciones ordinarias, porque significa cambiar datos que en principio fueron considerados constantes, como los parámetros de un sistema de control, el contenido de un mensaje para un LCD, etc. Sólo se tiene 1 instrucción para almacenar datos en la memoria de código (**SPM**), no obstante, debe considerarse que una memoria Flash se modifica por páginas y cuenta con mecanismos de seguridad para proteger su contenido. El uso de la instrucción **SPM** se describe en la sección 7.2.3.

En la tabla 3.17 se muestran estas instrucciones, cuando se emplean, debe tenerse en cuenta que la memoria de programa está organizada en palabras de 16 bits.

Tabla 3.17 Transferencias entre memoria de código y registros

Instrucción	Descripción		Operación
LPM	Carga indirecta de memoria de programa en R0		R0 = Flash[Z]
LPM Rd, Z	Carga indirecta de memoria de programa en Rd		Rd = Flash[Z]
LPM Rd, Z+	Carga indirecta con post-incremento		Rd = Flash[Z], Z = Z + 1
SPM	Almacenamiento indirecto en memoria de programa		Flash[Z] = R1:R0

Por último, para las transferencias entre Registros I/O y registros de propósito general se tienen las instrucciones mostradas en la tabla 3.18, en donde la variable P hace referencia a cualquiera de los 64 Registros I/O, sin importar a qué recurso pertenecen. Estas instrucciones se ejecutan en 1 ciclo de reloj y no modifican las banderas del registro de Estado.

Tabla 3.18 Transferencias entre Registros I/O y registros de propósito general

Instrucción	Descripción		Operación
IN Rd, P	Lee de un Registro I/O, deja el resultado en Rd		Rd = P
OUT P, Rs	Escribe en un Registro I/O, el valor de Rs		P = Rs

3.1.4 Instrucciones para el Manejo de Bits

En este grupo se encuentran las instrucciones para desplazamientos y rotaciones que se muestran en la tabla 3.19. Todas se ejecutan en 1 ciclo de reloj y modifican las banderas **Z**, **C**, **N**, **V** y **S** del registro **SREG**. Con excepción del desplazamiento aritmético a la derecha, las demás instrucciones funcionan con apoyo de la bandera de acarreo.

Tabla 3.19 Instrucciones para desplazamientos y rotaciones

Instrucción	Descripción		Operación
LSL Rd	Desplazamiento lógico a la izquierda		$C = Rd(7), Rd(n+1) = Rd(n), Rd(0) = 0$
LSR Rd	Desplazamiento lógico a la derecha		$C = Rd(0), Rd(n) = Rd(n + 1), Rd(7) = 0$
ROL Rd	Rotación a la izquierda		$C = Rd(7), Rd(n + 1) = Rd(n), Rd(0) = C$
ROR Rd	Rotación a la derecha		$C = Rd(0), Rd(n) = Rd(n + 1), Rd(7) = C$
ASR Rd	Desplazamiento aritmético a la derecha		$Rd(n) = Rd(n + 1), n = 0 \dots 6$

En la figura 3.2 se ilustran de manera gráfica los diferentes tipos de desplazamientos y rotaciones.

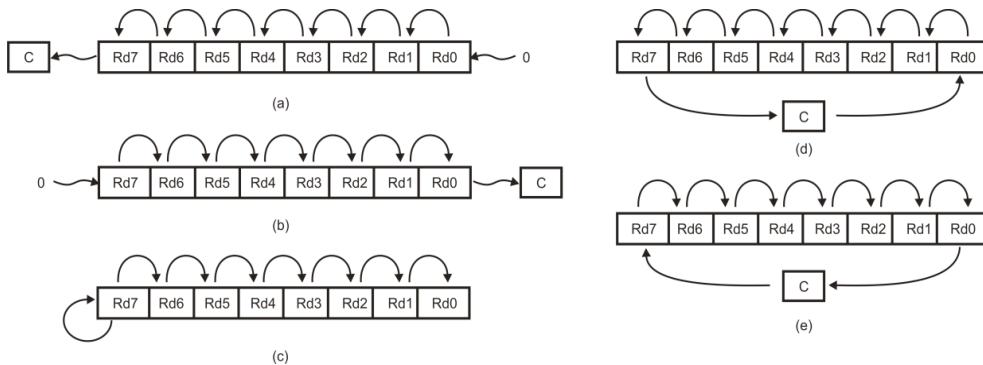


Figura 3.2 (a) Desplazamiento lógico a la izquierda, (b) a la derecha, (c) desplazamiento aritmético a la derecha, (d) rotación a la izquierda y (e) rotación a la derecha

Los desplazamientos lógicos insertan un 0 en el espacio generado, el desplazamiento a la izquierda equivale a una multiplicación por 2 y el desplazamiento a la derecha a una división entre 2. La diferencia entre un desplazamiento lógico y uno aritmético es que el último conserva el valor del bit en el espacio generado, con ello, el desplazamiento aritmético a la derecha equivale a una división entre 2 considerando números con signo, conservando el signo en el resultado. El desplazamiento aritmético a la izquierda no está implementado porque carece de significado.

Otra instrucción que es parte de este grupo es para intercambiar el nibble alto con el nibble bajo en un registro, ésta se muestra en la tabla 3.20. Se ejecuta en 1 ciclo de reloj y no modifica banderas.

Tabla 3.20 Instrucción para el intercambio de nibbles

Instrucción	Descripción		Operación
SWAP Rd	Intercambia nibbles en Rd		Rd(7..4) = Rd(3..0), Rd(3..0) = Rd(7..4)

También se incluyen instrucciones para modificar un bit en un Registro I/O, para ponerlo en alto o en bajo, siempre que esté permitido, dado que sólo los Registros I/O que están en el rango de 0x00 a 0x1F pueden ser manipulados por sus bits individuales. Estas instrucciones se muestran en la tabla 3.21, se ejecutan en 2 ciclos de reloj y tampoco modifican banderas.

Tabla 3.21 Instrucciones para modificar bits en los Registros I/O

Instrucción	Descripción		Operación
SBI P, b	Pone en alto al bit b del Registro P		P(b) = 1
CBI P, b	Pone en bajo al bit b del Registro P		P(b) = 0

El grupo incluye 2 instrucciones para transferencias de bits, una para cargar (*load*) del bit T a un bit de un registro de propósito general y otra para almacenar (*store*) un bit de un registro de propósito general en el bit T. El bit T está en el registro **SREG** y es empleado para respaldar el valor de un bit de un registro de propósito general. Estas instrucciones se muestran en la tabla 3.22, se ejecutan en 1 ciclo de reloj y es evidente que la instrucción de almacenamiento modifica al bit T.

Tabla 3.22 Instrucciones para transferencias de bits

Instrucción	Descripción		Operación
BTS Rs, b	Almacena al bit b de Rs en el bit T de SREG		T = Rs(b)
BTL Rd, b	Carga en el bit b de Rd desde el bit T de SREG		Rd(b) = T

El grupo se complementa con 18 instrucciones dedicadas a manipular los 8 bits del registro Estado. Éstas se muestran en la tabla 3.23, en las 2 primeras se especifica el número del bit a modificar, una es para poner al bit en alto y la otra para ponerlo en bajo. Las 16 restantes son una versión particular de las 2 primeras, son 8 pares y cada par está dedicado a uno de los bits del registro Estado, poniéndolo en alto o en bajo.

Todas se ejecutan en 1 ciclo de reloj y cada instrucción va a modificar la bandera del registro Estado que le corresponda. Las instrucciones en la tabla 3.23 están ordenadas de acuerdo con la ubicación de los bits en **SREG**, del bit más significativo al menos significativo.

Tabla 3.23 Instrucciones para manipular los bits del registro Estado

Instrucción	Descripción		Operación
BSET s	Pone en alto al bit s del registro Estado		SREG(s) = 1
BCLR s	Pone en bajo al bit s del registro Estado		SREG(s) = 0
SEI	Pone en alto al habilitador de interrupciones		I = 1

Instrucción	Descripción	Operación
CLI	Pone en bajo al habilitador de interrupciones	I = 0
SET	Pone en alto al bit de transferencias	T = 1
CLT	Pone en bajo al bit de transferencias	T = 0
SEH	Pone en alto a la bandera de acarreo del nibble bajo	H = 1
CLH	Pone en bajo a la bandera de acarreo del nibble bajo	H = 0
SES	Pone en alto a la bandera de signo	S = 1
CLS	Pone en bajo a la bandera de signo	S = 0
SEV	Pone en alto a la bandera de sobreflujo	V = 1
CLV	Pone en bajo a la bandera de sobreflujo	V = 0
SEN	Pone en alto a la bandera de negativo	N = 1
CLN	Pone en bajo a la bandera de negativo	N = 0
SEZ	Pone en alto a la bandera de cero	Z = 1
CLZ	Pone en bajo a la bandera de cero	Z = 0
SEC	Pone en alto a la bandera de acarreo	C = 1
CLC	Pone en bajo a la bandera de acarreo	C = 0

3.1.5 Instrucciones Especiales

Este grupo está integrado por 3 instrucciones para el ATMega8 y 4 para el ATMega16, son instrucciones que, por sus características, no pueden integrarse en los otros grupos. Estas instrucciones no modifican algún operando, su objetivo es auxiliar para la adecuada ejecución de un programa.

En la tabla 3.24 se muestran y describen estas instrucciones, la última de la lista no está en el ATMega8 porque no cuenta con el recurso JTAG.

Tabla 3.24 Instrucciones especiales

Instrucción	Descripción
NOP	No operación, empleada para forzar una espera de 1 ciclo de reloj
SLEEP	Introduce al MCU en el modo de bajo consumo previamente configurado
WDR	Reinicia al Watchdog Timer, para evitar reinicios por su desbordamiento
BREAK	Utilizada para depuración, desde el puerto JTAG

3.2 Modos de Direccionamiento

Los modos de direccionamiento son un aspecto fundamental cuando se diseña la arquitectura de un procesador, porque definen muchas de las características que se ven reflejadas cuando es puesto en marcha, características que involucran: la ubicación de los datos sobre los que operan las instrucciones, el tamaño de las constantes, los registros que se pueden considerar como operandos en una instrucción y el alcance de los saltos. En los microcontroladores AVR se observan 7 modos de direccionamiento:

1. Directo por registro
2. Directo a Registros I/O
3. Directo a memoria de datos
4. Indirecto a memoria de datos
5. Indirecto a memoria de código
6. Inmediato
7. Direccionamientos en bifurcaciones

En las siguientes secciones se describen e ilustran los diferentes modos de direccionamiento. Debe considerarse que para algunos modos se tienen diferentes variantes y que la mayoría de instrucciones sólo requieren de una palabra de 16 bits.

3.2.1 Direccionamiento Directo por Registro

En la instrucción se especifica el registro o registros que funcionan como operandos, ya que puede ser sólo uno o dos registros, este modo se ilustra en la figura 3.3, en donde se observa que se dispone de 5 bits para definir cada registro, por ello, bajo este modo de direccionamiento se puede utilizar a cualquiera de los 32 registros. Ejemplos de instrucciones que emplean sólo un registro son:

COM	R1
INC	R2
SER	R3

Ejemplos de instrucciones en donde se emplean dos registros son:

ADD	R0, R1
SUB	R2, R3
AND	R4, R5
MOV	R6, R7

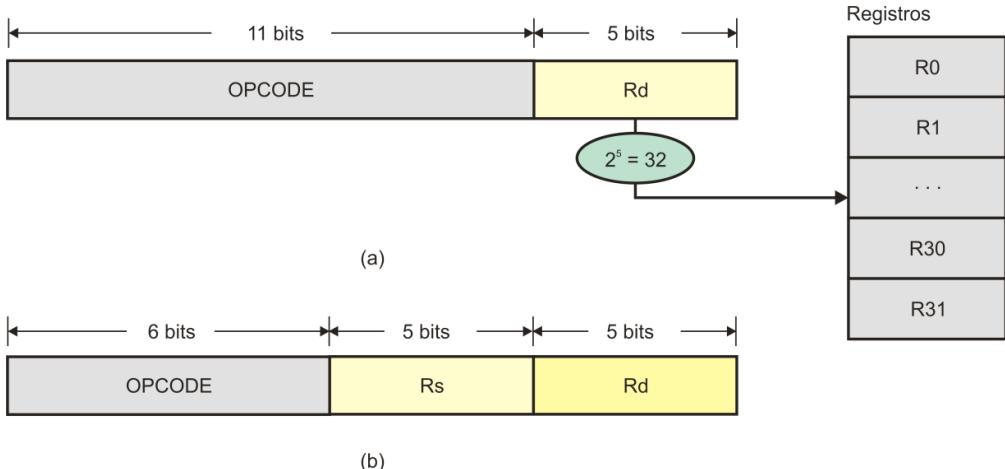


Figura 3.3 Direccionamiento directo empleando (a) un registro y (b) dos registros

3.2.2 Direccionamiento Directo a Registros I/O

Con este modo de direccionamiento se tiene acceso a los Registros I/O, que son la base para el manejo de los recursos en un MCU. En la figura 3.4 se ilustra el modo, se tienen 5 bits para el registro de propósito general (R), por lo que se puede utilizar cualquiera de ellos, y 6 bits para especificar al Registro I/O (P), también puede referirse a cualquiera de los 64 Registros I/O. Ejemplos de instrucciones con este modo de direccionamiento son:

```
OUT    PORTB, R13
IN     R15, PINA
```

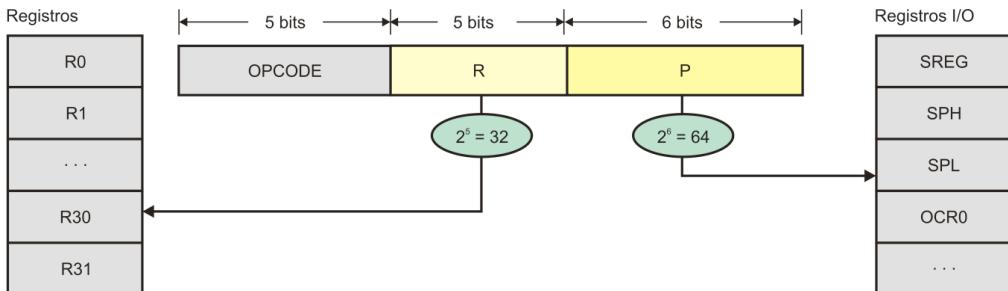


Figura 3.4 Direccionamiento directo a Registros I/O

3.2.3 Direcciónamiento Directo a Memoria de Datos

La instrucción incluye la dirección de la localidad en memoria de datos a la que tiene acceso, las instrucciones con este modo requieren de 2 palabras de 16 bits, en la segunda palabra se indica la dirección del dato en memoria. Con 16 bits es posible direccionar hasta 64 K de datos, sin embargo, el límite real está determinado por el espacio disponible en cada dispositivo, para el ATMega8 y ATMega16 el límite superior es de 0x45F. En la figura 3.5 se ilustra este modo de direcciónamiento. Ejemplos de instrucciones que lo utilizan son:

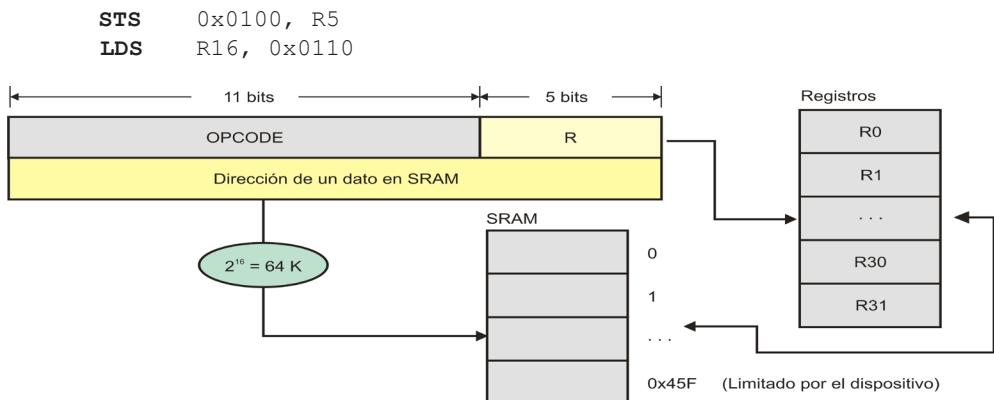


Figura 3.5 Direcciónamiento directo a memoria de datos

3.2.4 Direcciónamiento Indirecto a Memoria de Datos

Este modo de direcciónamiento utiliza a los registros de 16 bits: X, Y o Z como apuntadores, el apuntador a usar queda implícito en el opcode, por lo que en la instrucción sólo se especifica al registro de propósito general con el que opera. El modo de direcciónamiento se ilustra en la figura 3.6. Ejemplos de instrucciones que utilizan este modo son:

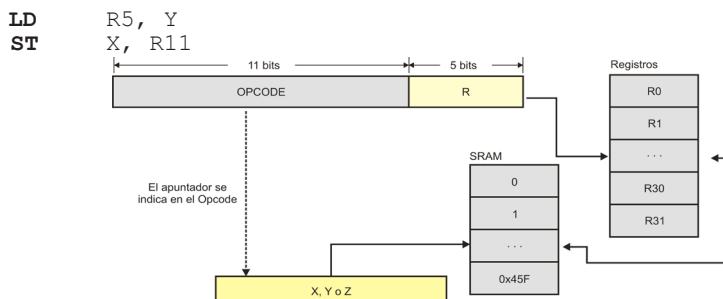


Figura 3.6 Direcciónamiento indirecto a memoria de datos, X, Y o Z como apuntadores.

Existen dos variantes de este modo de direccionamiento en donde además se modifica al apuntador. La primera variante es con un post-incremento, donde primero obtiene el dato de memoria y después se incrementa al apuntador, y la segunda con un pre-decrecimiento, en donde primero se disminuye al apuntador y luego se realiza el acceso a memoria. Esto también es aplicable a los 3 registros, X, Y o Z, estas ideas se muestran en la figura 3.7 y 3.8, respectivamente. Ejemplos de instrucciones con post-incrementos y pre-decrementos son:

LD	R5, Y+	; Carga con post-incremento
ST	X+, R6	; Almacenamiento con post-incremento
LD	R7, -X	; Carga con pre-decrecimiento
ST	-Z, R11	; Almacenamiento con pre-decrecimiento

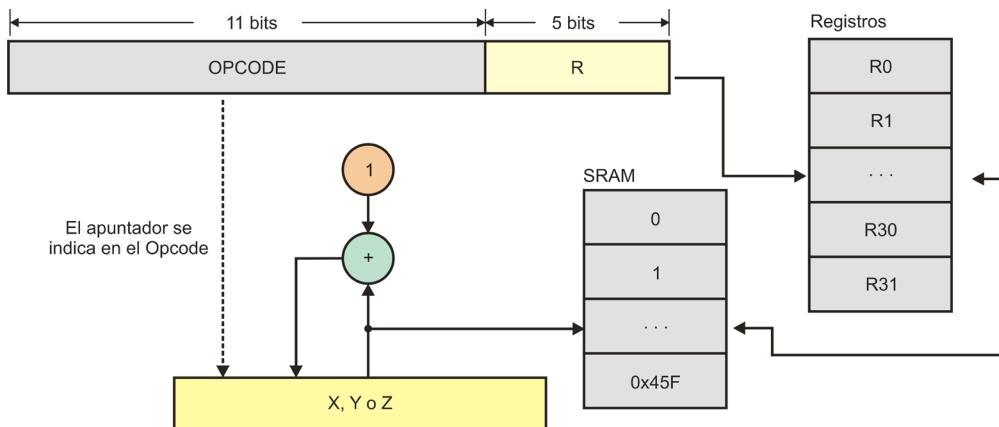


Figura 3.7 Direccionamiento indirecto con post-incremento

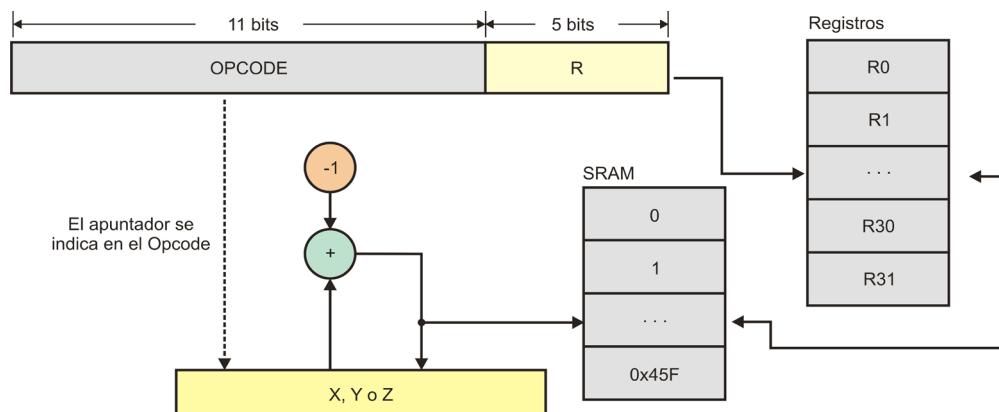


Figura 3.8 Direccionamiento indirecto con pre-decrecimiento

Otra opción es el direccionamiento indirecto con desplazamiento, en este modo se tiene acceso a una dirección formada por la suma entre el apuntador y una constante, pero sin modificar al apuntador. Para la constante se dispone de 6 bits, por lo que debe estar en el rango de 0 a 63. Sólo es posible con los apuntadores Y o Z, esta idea se muestra en la figura 3.9. Ejemplos de instrucciones que usan este modo de direccionamiento son:

```
LDD      R5, Y+0x020
STD      Z+0x10, R11
```

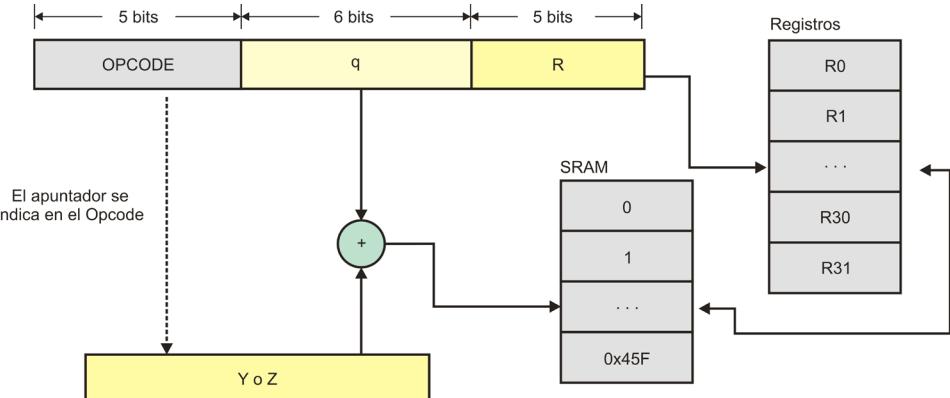


Figura 3.9 Direccionamiento indirecto con desplazamiento

3.2.5 Direccionamiento Indirecto a Memoria de Código

Este modo de direccionamiento utiliza al registro Z como apuntador, el cual queda implícito en el opcode. En la figura 3.10 se ilustra este modo de direccionamiento, ejemplos de instrucciones para acceso indirecto a memoria de código son:

```
LPM          ; R0 y Z están implícitos en la instrucción
LPM      R3, Z
SPM          ; R1:R0 y Z están implícitos
```

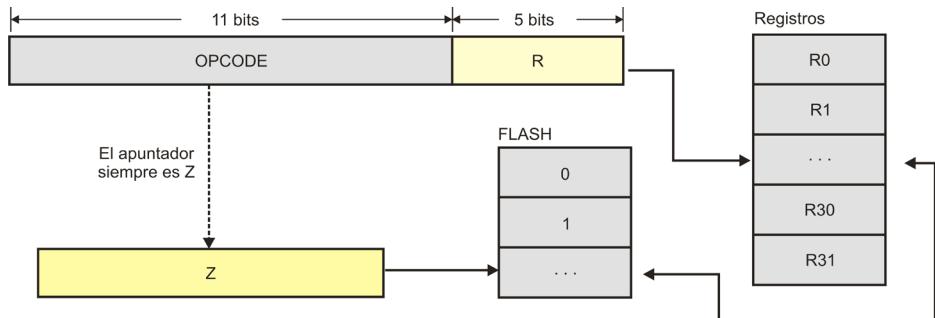


Figura 3.10 Direccionamiento indirecto a memoria de código

También existe una carga indirecta de memoria de código con post-incremento del apuntador Z, su comportamiento sería similar al mostrado en la figura 3.7.

3.2.6 Direccionamiento Inmediato

El direccionamiento inmediato es utilizado por aquellas instrucciones que incluyen una constante como uno de sus operandos. La constante es parte de la instrucción, debido a ello, uno de los operandos de la ALU se conoce de manera “inmediata”, a esto se debe el nombre del modo de direccionamiento.

En la figura 3.11 se muestra cómo se organizan las instrucciones que utilizan este modo de direccionamiento. Se observa que al quedar disponibles únicamente 4 bits para definir el número de registro, sólo se puede referenciar a 16 de ellos, es por eso que las instrucciones con constantes sólo pueden incluir registros en el rango de R16 a R31.

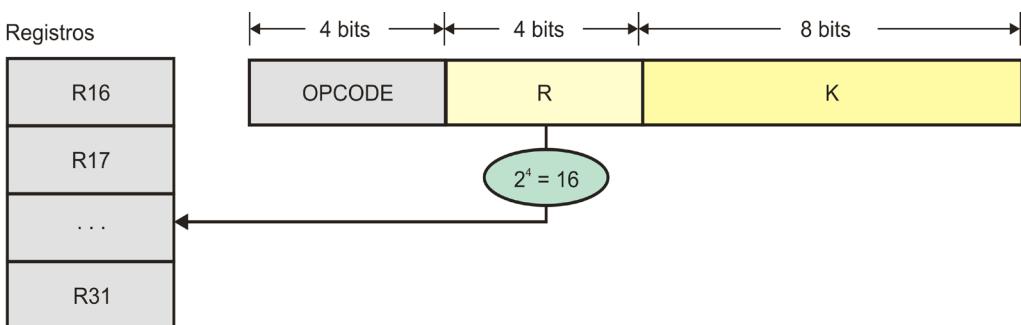


Figura 3.11 Direccionamiento inmediato

Ejemplos de instrucciones que utilizan direccionamiento inmediato son:

ANDI	R17, 0xF3
SUBI	R19, 0x12
ORI	R31, 0x03
LDI	R16, 0x25

Para la constante se dispone de 8 bits, puede estar en el rango de 0 a 255.

3.2.7 Direccionamientos en Bifurcaciones

Las bifurcaciones o saltos (condicionales o incondicionales) permiten cambiar el flujo secuencial durante la ejecución de un programa, esto se consigue modificando el valor del PC (contador del programa). Para estas instrucciones se tienen tres modos de direccionamiento: relativo, indirecto y absoluto.

3.2.7.1 Bifurcaciones con Direccionamiento Relativo

Las bifurcaciones con direccionamiento relativo incluyen una constante que se suma al PC, esta constante puede ser positiva o negativa, para bifurcar hacia adelante o atrás, con respecto a la ubicación de la instrucción de la bifurcación, en realidad son relativas a $PC + 1$, dado que el incremento del PC es automático. En la figura 3.12 se ilustra el funcionamiento de las bifurcaciones relativas.

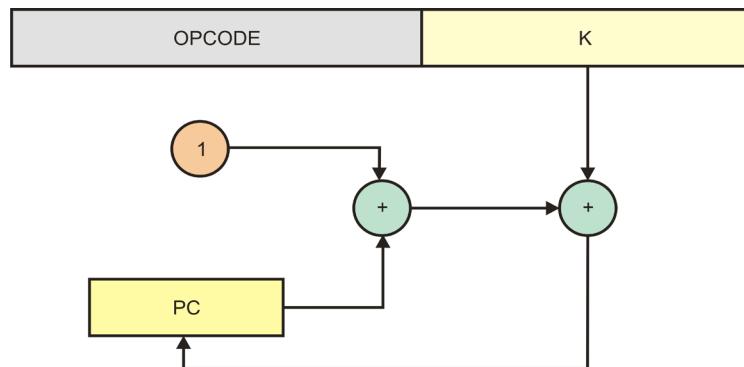


Figura 3.12 Bifurcaciones con direccionamiento relativo

El direccionamiento relativo se aplica en bifurcaciones incondicionales, en esos casos la constante K es de 12 bits. Ejemplos de estas instrucciones son:

RJMP -20
RCALL 32

Las instrucciones para bifurcaciones condicionales sólo dejan disponibles 7 bits para la constante. Ejemplos de estas instrucciones son:

BREQ 15
BRNE -10
BRGE 10

Cuando se escribe un programa se utilizan etiquetas, de acuerdo con su ubicación, el ensamblador calcula el valor de las constantes.

Existen bifurcaciones condicionales que sólo brincan la siguiente instrucción, estos "saltitos" también son relativos al PC, aunque el formato de la instrucción es más simple, dado que sólo se omite la ejecución de una instrucción.

3.2.7.2 Bifurcaciones con Direccionamiento Indirecto

El direccionamiento indirecto involucra el uso del registro Z como apuntador, un apuntador es una variable cuyo contenido es una dirección. En una bifurcación con este modo de direccionamiento, el contenido del apuntador Z se escribe en el PC, el cual también es un apuntador. Básicamente se realiza un reemplazo de direcciones, esto se ilustra en la figura 3.13.

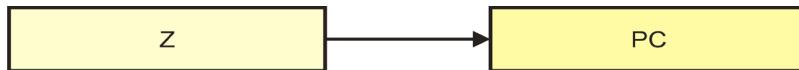


Figura 3.13 Bifurcaciones con direccionamiento indirecto

Ejemplos de instrucciones que utilizan este modo de direccionamiento son:

IJMP
ICALL

El apuntador Z queda implícito en el opcode de la instrucción.

3.2.7.3 Bifurcaciones con Direccionamiento Absoluto

El direccionamiento es absoluto cuando se conoce la dirección destino de la bifurcación, la dirección forma parte de la misma instrucción. El formato que emplean las instrucciones con este modo de direccionamiento se muestra en la figura 3.14, en donde se observa que se dispone de 22 bits para el destino de la bifurcación, esta capacidad de direccionamiento queda sobrada para un ATMega16, no obstante, el formato de las instrucciones se dispone de esta manera para ser utilizado por otros miembros de la familia AVR que cuenten con un espacio mayor en su memoria de código.

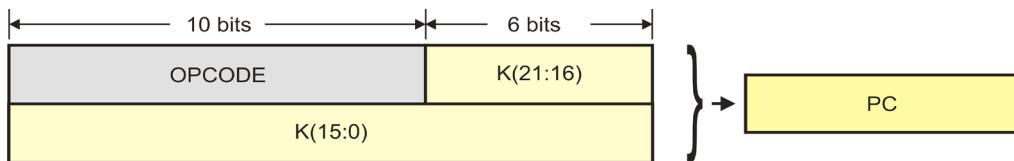


Figura 3.14 Bifurcaciones con direccionamiento absoluto

Un ATMega8 no incluye bifurcaciones que utilicen este modo de direccionamiento, dado que su espacio total puede cubrirse con bifurcaciones relativas o indirectas. Ejemplos de instrucciones que utilizan este modo de direccionamiento son:

JMP 0x001FFF
CALL 0x003000

3.3 Programación en Lenguaje Ensamblador

Un programa en lenguaje Ensamblador contiene:

- **Instrucciones:** Elementos del lenguaje que se traducen a código máquina, cada instrucción tiene su opcode y sus operandos. El procesador ejecuta las instrucciones para determinar el comportamiento de un sistema.
- **Directivas:** Elementos del lenguaje que ayudan en la organización de un programa, indicando diferentes aspectos como la ubicación del código, definiciones, etc. Las directivas no generan código máquina, son elementos propios de la herramienta empleada para ensamblar un programa.

En la sección 3.1 se mostraron las instrucciones que puede ejecutar un MCU AVR y en la sección 3.2, con los modos de direccionamiento, se revisaron los diferentes formatos de las instrucciones. En esta sección, para contar con los elementos suficientes para programar en Ensamblador, se revisan algunas de las directivas incluidas en el ensamblador desarrollado por Atmel (AVRASM) y distribuido con la herramienta AVR Studio⁵. No se revisan todas las directivas, sólo aquellas que son usadas con mayor frecuencia, mostrando ejemplos de su uso con los microcontroladores bajo estudio. Las directivas generalmente se preceden con un punto.

En la sección 3.5 se exponen 3 problemas simples con sus correspondientes soluciones. Las soluciones codificadas en ensamblador muestran la estructura general que deben seguir los programas escritos en este lenguaje e ilustran la forma en que se pueden traducir eficientemente las estructuras de control de flujo de alto nivel a bajo nivel, así como el uso de directivas.

3.3.1 Directiva INCLUDE

Esta directiva se utiliza para leer el código fuente de otro archivo, se coloca al inicio de un programa y es común utilizarla para incluir todas las definiciones relacionadas con un dispositivo particular. Por ejemplo, si se utiliza un ATMega8 se debe agregar:

```
.include <m8def.inc> ; Incluye definiciones de un ATMega8
```

Al emplear un ATMega16 debe agregarse:

```
.include <m16def.inc> ; Incluye definiciones de un ATMega16
```

3.3.2 Directivas CSEG, DSEG y ESEG

Estas directivas se utilizan para definir diferentes tipos de segmentos en un programa. Todo lo que se escribe, posterior a la directiva, es encausado al segmento correspondiente.

- **CSEG:** Marca el inicio de un segmento de código, es decir, un segmento en la memoria FLASH. Es el segmento por omisión, esto significa que la directiva puede omitirse en un programa si no se hace uso de otros segmentos. Un programa en ensamblador puede tener múltiples segmentos de código, los cuales son concatenados en el momento en que el programa es ensamblado. Cuando se ensambla un programa se genera un archivo con todos los segmentos de código definidos (*archivo.hex*).

⁵ AVR Studio es una suite para el desarrollo de aplicaciones con los microcontroladores AVR, proporciona las facilidades para la simulación y ensamblado de programas, es de distribución libre y puede descargarse del sitio de ATTEL (<http://www.atmel.com>).

- **DSEG:** Marca el inicio de un segmento para los datos, con ello, en un programa se pueden reservar localidades de SRAM para un propósito especial. De manera similar, puede haber múltiples segmentos de datos.
- **ESEG:** Indica que la siguiente información es guardada en EEPROM. También al ensamblar se crea un archivo a descargarse en la memoria EEPROM del MCU (*archivo.eep*), el cual sirve para definir el contenido inicial de variables en EEPROM. Para la lectura y modificación de estas variables puede utilizarse el código mostrado en la sección 2.4.2.

Un ejemplo del uso de estas directivas es el siguiente:

```
.DSEG ; Inicia un segmento de datos
var1: .BYTE 1 ; Variable de 1 byte en SRAM

.CSEG ; Inicia un segmento de código

ldi R16, 0x25 ; Instrucciones para alguna tarea
mov R0,R16
. .

const: .DB 0x02 ; Constante con 0x02 en memoria de programa

.ESEG ; Inicia un segmento de EEPROM

eevar1:.DB 0x3f ; Constante con 0x3f en memoria EEPROM
```

3.3.3 Directiva DB y DW

Con estas directivas se definen constantes o variables a utilizarse en memoria de programa o en EEPROM. **DB** es para datos que ocupan 1 byte (*Define Byte*) y **DW** para datos de 16 bits (*Define Word*). Si se requiere más de un dato, éstos deben organizarse en una lista de datos separados por comas. Ejemplos:

```
.CSEG

; Constantes en memoria FLASH

const1: .DB 0x33
consts1: .DB 0, 255, 0b01010101, -128, 0xaa
consts2: .DW 0, 0xffff, 0b1001110001010101, -32768, 65535

.ESEG

; Constantes o variables en EEPROM

eevar1: .DB 0x37
```

```

eelist1:      .DB      1,2,3,4
eelist2:      .DW      0,0xffff,10

```

La etiqueta con la definición puede utilizarse como referencia (dirección) para tener acceso a las constantes en un programa. Si es una lista, con la etiqueta se tiene acceso al inicio de la lista.

3.3.4 Directiva EQU

Se utiliza para definir constantes simbólicas, estas definiciones posteriormente pueden usarse en las instrucciones. Por ejemplo:

```

.EQU    io_offset = 0x23

.EQU    portA      = io_offset + 2

.CSEG
CLR    R2          ; Limpia al registro 2
OUT    portA, R2   ; Escribe a portA

```

El ensamblador hace referencia a los Registros I/O por su dirección y a sus bits por su posición, sin embargo, con la directiva **INCLUDE** se incluye una biblioteca con las definiciones de un dispositivo, por lo que al escribir un programa es posible utilizar sus nombres. Estas bibliotecas hacen un uso extensivo de la directiva **EQU**, asociando etiquetas a las direcciones.

3.3.5 Directiva ORG

Esta directiva se utiliza para ubicar la información subsecuente en cualquiera de los segmentos de memoria, indica la dirección a partir de la cual se colocan las variables en SRAM, constantes en EEPROM o en FLASH, o código en FLASH. Después de la directiva debe indicarse una dirección válida en el rango del segmento considerado.

Por ejemplo:

```

.DSEG
.var1: .ORG 0x120
        .Byte 1           ; Variable ubicada en la dirección 0x120 del
                           ; segmento de datos
.CSEG
.inicio: .ORG 0x000
        .RJMP inicio
        .ORG 0x010
        .MOV R1, R2       ; Código ubicado en la dirección 0x10
                           ; . . .

```

Se emplea principalmente para organizar a las instrucciones de un programa cuando se manejan interrupciones, porque hace posible la ubicación del código, de acuerdo con los vectores de interrupciones, como se mostró en la sección 2.6.

3.3.6 Directivas HIGH y LOW

Se utilizan para separar constantes de 16 bits, con **High** se hace referencia a la parte alta y con **Low** a la parte baja. Por ejemplo, para cargar la constante 578 en los registros R27 y R26:

```
LDI    R27, HIGH(578)
LDI    R26, LOW(578)
```

Se comentó al inicio que las directivas se preceden con un punto, esto no ocurre en **High** y **Low**, pero son consideradas como directivas porque no generan código máquina.

Un uso importante de estas directivas es la referencia a constantes que correspondan con direcciones de un programa, para después hacer accesos mediante direccionamiento indirecto. Por ejemplo, para posicionar al apuntador Z (R31:R30) al comienzo de una tabla de datos:

```
LDI    R30, LOW(tabla)
LDI    R31, HIGH(tabla)
...
tabla: .DB    0x01,0x02,0x03,0x04
```

3.3.7 Directiva BYTE

Reserva uno o más bytes en SRAM o EEPROM para el manejo de variables, la cantidad de bytes debe especificarse. Este espacio no es inicializado, sólo queda reservado para que posteriormente sea referido con una etiqueta, por ejemplo:

```
.DSEG

var1: .BYTE   1           ; variable de 1 byte
var2: .BYTE   10          ; variable de 10 bytes

.CSEG

STS    var1, R17         ; acceso a var1 por direccionamiento directo
LDI    R30, LOW(var2)    ; Z apunta a Var2
LDI    R31, HIGH(var2)
LD    R1, Z               ; acceso a var2 por el apuntador Z
```

3.4 Programación en Lenguaje C

Si bien, el entorno del AVR Studio únicamente incluye al programa ensamblador (AVRASM), proporciona las facilidades para enlazarse con compiladores de lenguaje C desarrollados por fuentes diferentes a Atmel. Instalando al compilador adecuado, desde el mismo entorno es posible la edición de programas, la invocación del compilador con exhibición de resultados, su simulación y depuración en lenguaje C.

Uno de estos compiladores es el avr-gcc, incluido en una suite conocida como WinAVR⁶, la cual es parte del proyecto GNU. Después de instalar a la suite, el compilador es llamado automáticamente desde el entorno del AVR Studio cada vez que se requiere, su uso queda transparente al programador. Además del compilador, la suite incluye un conjunto de bibliotecas con funciones enfocadas a los recursos de los AVR.

El compilador está orientado al estándar ANSIC, se pueden emplear a todos los elementos del lenguaje, como tipos de datos y estructuras de control de flujo. En esta sección se revisan algunas características del lenguaje, principalmente las consideraciones para trabajar con los microcontroladores AVR.

Los problemas de la sección 3.5 también se han resuelto en Lenguaje C, de manera que también muestran la estructura general que deben seguir los programas escritos en este lenguaje.

3.4.1 Tipos de Datos

Pueden usarse los tipos básicos con sus diferentes modificadores, en la tabla 3.25 se muestran los diferentes tipos de datos, con la cantidad de bits que requieren y el rango de combinaciones que cubren.

Tabla 3.25 Tipos de datos aceptados por el compilador

Tipo	Tamaño (bits)	Rango
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127

6 WinAVR™ es una suite de herramientas de desarrollo para la serie de microcontroladores AVR de Atmel. Incluye al compilador AVR-GCC de GNU para C y C++. La suite contiene todas las herramientas necesarias para desarrollar aplicaciones con los AVR: Compilador, programador, depurador y más. Es un proyecto de código abierto, más información puede obtenerse de <http://winavr.sourceforge.net/>.

Tipo	Tamaño (bits)	Rango
int	16	-32,768 a 32,767
short int	16	-32,768 a 32,767
unsigned int	16	0 a 65,535
signed int	16	-32,768 a 32,767
long int	32	-2,147,483,648 a 2,147,483,647
unsigned long int	32	0 a 4,294,967,295
signed long int	32	-2,147,483,648 a 2,147,483,647
float	32	+/- 1.175 x 10 ⁻³⁸ a +/- 3.402 x 10 ⁺³⁸
double	32	+/- 1.175 x 10 ⁻³⁸ a +/- 3.402 x 10 ⁺³⁸

En la tabla 3.25 se muestran diferentes tipos de datos con el mismo comportamiento, por ejemplo *float* y *double* prácticamente hacen referencia al mismo tipo de datos, la inclusión de ambos debe ser para mantener compatibilidad con el estándar ANSI C. Las variables de los tipos *char*, *unsigned char* o *signed char* pueden asignarse directamente con los Registros I/O.

Al trabajar con un microcontrolador, debe considerarse el limitado espacio de memoria para no agotarlo con un mal manejo de los tipos de datos, por ejemplo, si en un ciclo repetitivo se realizan pocas iteraciones, es mejor usar un dato del tipo *unsigned char* en lugar de un *int*. Ya que no sólo es un byte extra, sino que con un *int* se ejecutan operaciones de 16 bits, requiriendo más instrucciones de bajo nivel.

En el entorno de desarrollo WinAVR se encuentran una serie de definiciones de tipos de datos que surgen a partir de los estándares mostrados en la tabla 3.25, ejemplos de estas definiciones de tipos son:

```
typedef signed char     int8_t
typedef unsigned char   uint8_t
typedef signed int      int16_t
typedef unsigned int    uint16_t
typedef signed long int int32_t
typedef unsigned long int uint32_t
```

Al programar se pueden utilizar los tipos de datos del ANSI C o las definiciones del WinAVR, esto no afecta el comportamiento de un programa. En el código desarrollado en este documento se utiliza al estándar y, sólo si es necesario, se emplea algún modificador propio del entorno de desarrollo.

3.4.2 Operadores Lógicos y para el Manejo de Bits

Aunque el compilador acepta todos los operadores del ANSI C (aritméticos, relacionales, de incremento y decremento), únicamente se describen los operadores lógicos y para el manejo de bits, porque son muy utilizados al trabajar con microcontroladores.

En la tabla 3.26 se muestran los operadores lógicos, sus operandos son expresiones lógicas (proporcionan falso o verdadero) y se emplean para crear expresiones lógicas de mayor complejidad, cualquier valor diferente de 0 es interpretado como verdadero.

Tabla 3.26 Operadores lógicos

Operador	Símbolo
AND	&&
OR	

En la tabla 3.27 se muestran los operadores para el manejo de bits, éstos trabajan sobre datos de los tipos *char*, *int* o *long* (no trabajan sobre punto flotante) y afectan el resultado al nivel de bits. Son muy utilizados para enmascarar información, es decir, modificar únicamente algunos bits de un dato sin alterar a los demás, o bien para evaluar el estado de un bit.

Tabla 3.27 Operadores para el manejo de bits

Operador	Símbolo
Complemento a 1	~
Desplazamiento a la izquierda	<<
Desplazamiento a la derecha	>>
AND	&
OR	
OR exclusivo	^

3.4.3 Tipos de Memoria

El microcontrolador tiene 3 espacios diferentes de memoria: SRAM (incluye a los registros de propósito general), FLASH y EEPROM, las aplicaciones pueden requerir que las variables y constantes se almacenen en diferentes tipos de memoria.

3.4.3.1 Datos en SRAM

Las variables son datos que van a ser leídos o escritos continuamente, por lo tanto, deben estar en SRAM, éste es el espacio de almacenamiento por default, por ejemplo, las siguientes declaraciones:

```
unsigned char x, y;
unsigned int a, b, c;
```

Colocan a las variables en SRAM. Si es posible, el compilador utiliza los registros de propósito general (R0 a R31) para algunas variables, pero como los registros están limitados en número, al agotarse se utiliza la SRAM de propósito general.

Los apuntadores son fundamentales al programar en lenguaje C y por lo tanto, no se han excluido de los microcontroladores AVR. Un apuntador también se maneja en SRAM porque es una variable, su contenido es una dirección que debe hacer referencia a algún objeto de SRAM. Por ejemplo:

```

char  cadena[]  = "hola mundo";
char  *pcad;
pcad = cadena;

```

3.4.3.2 Datos en FLASH

Las constantes son datos que no cambian durante la ejecución normal de un programa, los datos de este tipo pueden ser almacenados en la memoria FLASH. Al manejar las constantes en FLASH se liberan espacios significativos de SRAM, esto es fundamental para aplicaciones que incluyan cadenas de texto o tablas de búsqueda.

Para que el compilador dirija las constantes hacia la FLASH, éstas deben identificarse con la palabra reservada **const** e incluir al atributo **PROGMEM**, definido en la biblioteca **pgmspace.h** que es parte del WinAVR. Ejemplos de declaraciones de constantes en FLASH son:

```

const char cadena[] PROGMEM = "Cadena con un mensaje constante";
const unsigned char tabla[] PROGMEM = { 0x24, 0x36, 0x48, 0x5A, 0x6C };

```

Las declaraciones anteriores tienen efecto cuando se realizan en un ámbito global, es decir, fuera de la función principal (**main**). Con ello, los datos quedan en FLASH y para su acceso deben utilizarse las funciones adecuadas, de lo contrario, al compilar se agrega una secuencia de código que realiza copias de FLASH a SRAM.

Las funciones para la lectura de constantes en FLASH están incluidas en la misma biblioteca (**pgmspace.h**), algunas de ellas son:

```

pgm_read_byte(address);           // Lee 8 bits
pgm_read_word(address);          // Lee 16 bits
pgm_read_dword(address);         // Lee 32 bits

```

Las funciones reciben como argumento la dirección del dato en FLASH, un ejemplo para la lectura del dato *i* de la tabla de constantes es:

```
x = pgm_read_byte(&tabla[i]);
```

Si se quiere emplear un apuntador a la memoria FLASH, debe declararse como **PGM_P**. La biblioteca **pgmspace.h** incluye funciones que trabajan con bloques completos de memoria FLASH, una de las más empleadas es **strcpy_P**, esta función sirve para leer una cadena de memoria FLASH y depositarla en SRAM, su uso se muestra en el siguiente código:

```

char  buf[32];                // Buffer destino, en SRAM
PGM_P p;                     // Apuntador a memoria FLASH

p = cadena;                   // p apunta a la cadena en FLASH
strcpy_P(buf, p);             // copia de FLASH a SRAM

```

En el ejemplo 3.2, descrito en la sección 3.5, se presenta una aplicación que hace uso de constantes en memoria FLASH.

3.4.3.3 Datos en EEPROM

Otra consideración debe tenerse para aquellas variables en las que se requiera conservar su contenido, aun en ausencia de energía, estas variables no pueden manejarse en SRAM porque es memoria volátil, deben almacenarse en la EEPROM.

Para ubicar una variable en la EEPROM, debe estar precedida con el atributo **EEMEM**, el cual está definido en la biblioteca **eeprom.h**, que también es parte del entorno de WinAVR. Este atributo hace que las declaraciones siguientes sean direccionadas a la EEPROM, éstas deben ser globales y realizarse antes de la declaración de constantes para la FLASH o variables para la SRAM. En el siguiente ejemplo se declaran variables para la EEPROM:

```
#include      <avr/eeprom.h>           //Biblioteca para la EEPROM

EEMEM int contador = 0x1234;           // Un entero requiere 2 bytes

EEMEM unsigned char clave[4] = { 1, 2, 3, 4}; // Arreglo de 4 bytes
```

La dirección para cada dato en EEPROM se define en el orden de su declaración, para el ejemplo anterior, la variable contador ocupa las direcciones 0 y 1, mientras que el arreglo utiliza de la localidad 2 a la 5. El contenido en la EEPROM con estas declaraciones es:

```
34 12 01 02 03 04 FF FF FF . . .
```

El acceso a los datos en la EEPROM se realiza por medio de los registros **EEAR**, **EEDR** y **EECR**, para ello pueden emplearse las funciones mostradas en la sección 2.4.2, o bien, desarrollar otras funciones en donde se involucre a la interrupción por fin de escritura en EEPROM. En la biblioteca **eeprom.h** se cuenta con diversas funciones para el acceso a la EEPROM, 2 de éstas son:

- **eeprom_read_byte**: Lee un byte de la EEPROM, en la dirección que recibe como argumento.
- **eeprom_write_byte**: Escribe un byte en la EEPROM, como argumentos recibe la dirección y el dato.

También se incluyen funciones para datos de 16 y 32 bits, así como para el manejo de bloques de memoria.

3.5 Programas de Ejemplo

En esta sección se muestran 3 ejercicios resueltos en Lenguaje C y en ensamblador, para familiarizarse con la programación de los microcontroladores AVR. Los ejercicios se probaron en un ATMega8, pero pueden funcionar en un ATMega16. En lenguaje C no se necesitarían cambios en el código y para ensamblador, sólo habría que remplazar la biblioteca de definiciones.

Cuando se inicia con un proyecto nuevo en el AVR Studio, se debe especificar el dispositivo destino. El apéndice C contiene un tutorial sobre el AVR Studio, mostrando los pasos a seguir durante la creación y simulación de un proyecto.

3.5.1 Parpadeo de un LED

Al iniciarse en algún lenguaje de programación, normalmente se codifica al típico programa “Hola Mundo”, a pesar de que el programa no tiene algún propósito, sirve para mostrar la estructura de los programas futuros, éste es el objetivo del presente ejemplo.

Ejemplo 3.1 Realice un programa que haga parpadear un LED conectado en la terminal PB0 a una frecuencia aproximada de 1 Hz (periodo de 1 S), considerando un ciclo útil del 50 % ($\frac{1}{2}$ S encendido y $\frac{1}{2}$ S apagado). En la figura 3.15 se muestra el hardware requerido y en la 3.16 un diagrama de flujo con el comportamiento esperado.

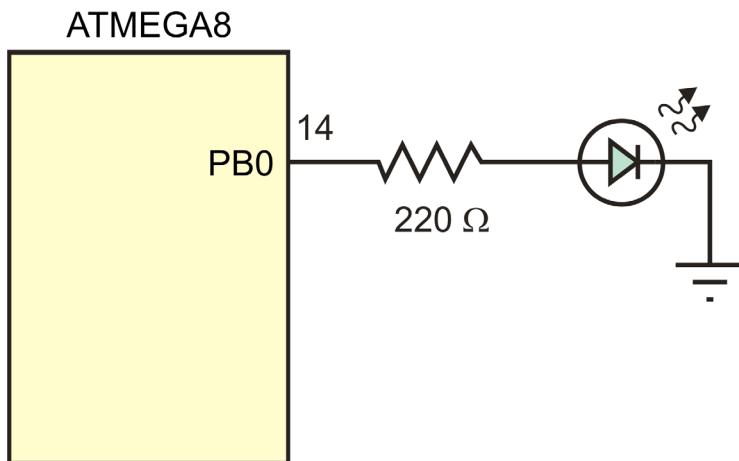


Fig. 3.15 Hardware para el problema del parpadeo de un LED

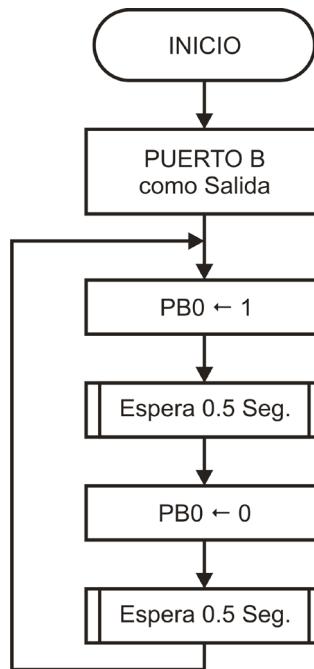


Fig. 3.16 Comportamiento deseado en el problema del parpadeo de un LED

En la figura 3.16 se observa que el programa nunca termina, permanece en un lazo infinito, esto es normal en sistemas basados en MCUs.

Para el programa en Ensamblador, se debe realizar una rutina para los retrasos con base en ciclos repetitivos, para ello, se asume un oscilador interno de 1 MHz, que corresponde con la configuración inicial de un ATMega8.

```

.include <m8def.inc>      ; Biblioteca con definiciones

LDI    R16, 0xFF
OUT   DDRB, R16          ; Puerto B como salida

LDI    R16, 0x04          ; Ubica al apuntador de pila al final de SRAM
OUT   SPH, R16           ; porque hay un llamado a una rutina
LDI    R16, 0x5F
OUT   SPL, R16

Lazo:
SBI   PORTB, 0           ; Lazo infinito
; PB0 en alto
RCALL Espera_500ms
CBI   PORTB, 0           ; PB0 en bajo
RCALL Espera_500ms
RJMP  Lazo

;
; Una rutina de espera se debe revisar del lazo interno al externo
;   500 mS = 500 000 uS = 2 x ( 250 x (250 x 4 uS) )
;
```

```

Espera_500mS:
    LDI    R18, 2
et3:   LDI    R17, 250
et2:   LDI    R16, 250
et1:   NOP    ; Itera 250 veces, emplea 4 uS por iteración
        DEC    R16    ; 250 x 4 uS = 1000 uS = 1 mS
        BRNE  et1    ; La instrucción evalúa la bandera de cero
                    ; brinca si no hay bandera de cero
        DEC    R17
        BRNE  et2    ; 1 mS x 250 = 250 mS

        DEC    R18
        BRNE  et3    ; 250 mS x 2 = 500 mS
    RET

```

Para la versión en Lenguaje C, puesto que se desconoce cuantas instrucciones de bajo nivel corresponden a las instrucciones de alto nivel, lo mejor para los retardos es emplear la función `_delay_ms(double ms)` de la biblioteca `delay.h`, que es parte del entorno de desarrollo WinAVR. Esta función también se basa en iteraciones, el número de iteraciones depende del argumento recibido y de la frecuencia de la CPU (`F_CPU`).

La función `_delay_ms(double ms)` recibe como argumento un número en punto flotante de doble precisión indicando la cantidad de milisegundos que se requieren de espera, el retraso máximo es de 262.14 mS/`F_CPU`, con la frecuencia en MHz. Si se intentan intervalos de espera mayores, no se producen errores de sintaxis, pero la función no proporciona el retardo esperado.

```

#define F_CPU 1000000UL          // Frecuencia de trabajo de 1 MHz

#include     <util/delay.h>      // Funciones para retrasos
#include     <avr/io.h>         // Definiciones de Registros I/O

int main() {                      // La función es int, aunque no haya
                                    // retorno, si
                                    // se define como void, se obtiene una
                                    // precaución
    DDRB = 0xFF;                  // Puerto B como salida

    while(1) {                   // Lazo infinito
        PORTB = PORTB | 0x01;     // PB0 en alto (máscara con OR)
        _delay_ms(250);
        _delay_ms(250);
        PORTB = PORTB & 0xFE;     // PB0 en bajo (máscara con AND)
        _delay_ms(250);
        _delay_ms(250);

    }
}

```

Se observa que sin importar el lenguaje de programación, se debe conseguir el comportamiento mostrado en la figura 3.16. En lenguaje C, la inicialización del apuntador de pila no se realiza en el código fuente, estas instrucciones las agrega el compilador en el momento en que genera el código de bajo nivel.

La biblioteca *delay.h* también cuenta con la función *_delay_us(double us)* para retardos en el orden de microsegundos, con un máximo de 768 us/F_CPU.

3.5.2 Decodificador de Binario a 7 Segmentos

En este ejemplo se resaltan 2 aspectos importantes al trabajar con MCUs, el primero es que los códigos de 7 segmentos son constantes, por lo tanto conviene depositarlos en memoria FLASH, y el segundo es que se debe tener el mismo tiempo de respuesta para todas las combinaciones, es decir, obtener el código de la F debe requerir el mismo tiempo que obtener el código del 0.

En otras palabras, en este ejemplo se ilustra el manejo de una búsqueda, en una tabla de constantes.

Ejemplo 3.2 Desarrolle un programa que lea los 4 bits menos significativos del Puerto D [PD3:PD0] y genere su código en 7 segmentos en el Puerto B. En la figura 3.17 se muestra el hardware requerido con el valor de las salidas para cada una de las entradas, la solución debe comportarse como el diagrama de flujo mostrado en la figura 3.18

Núm.	g	f	e	d	c	b	a	HEX
0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	1	1	0	0x06
2	1	0	1	1	0	1	1	0x5B
3	1	0	0	1	1	1	1	0x4F
4	1	1	0	0	1	1	0	0x66
5	1	1	0	1	1	0	1	0x6D
6	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	1	1	1	0x07
8	1	1	1	1	1	1	1	0x7F
9	1	1	0	0	1	1	1	0x67
A	1	1	1	0	1	1	1	0x77
B	1	1	1	1	1	0	0	0x7C
C	0	1	1	1	0	0	1	0x39
D	1	0	1	1	1	1	0	0x5E
E	1	1	1	1	0	0	1	0x79
F	1	1	1	0	0	0	1	0x71

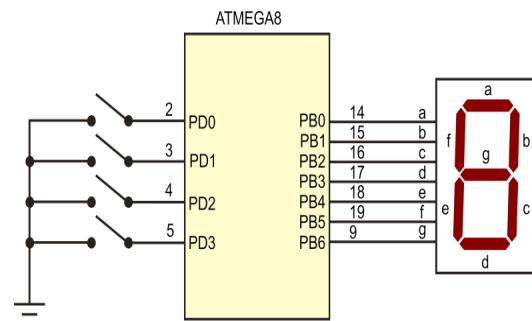


Figura 3.17 Decodificador de binario a 7 Segmentos

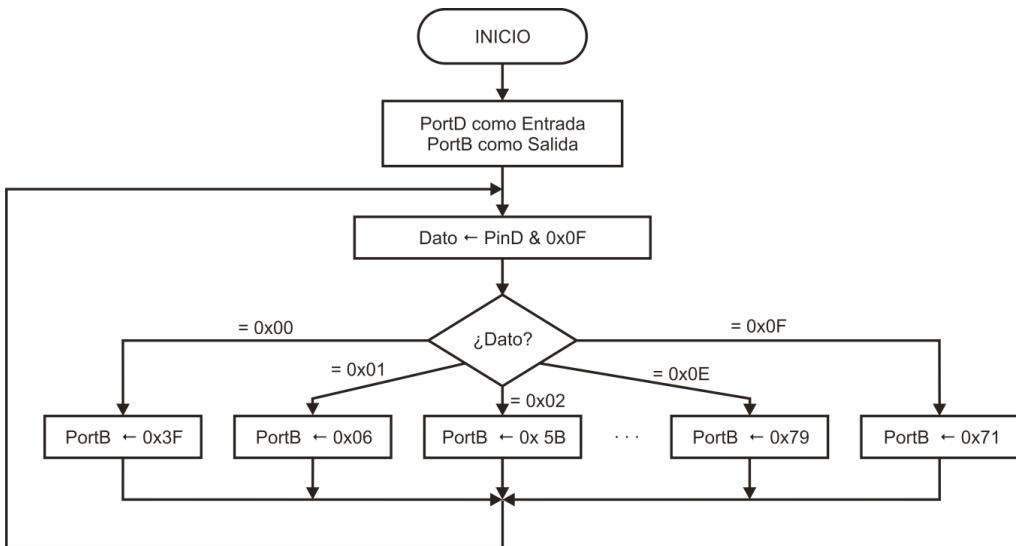


Figura 3.18 Comportamiento esperado en el decodificador de binario a 7 Segmentos

Una idea inmediata para solucionar este problema involucra la realización de 15 comparaciones, esta propuesta es ineficiente porque los tiempos de respuesta serían diferentes para cada caso.

Para la solución en lenguaje C, el diagrama de flujo sugiere el uso de una estructura *switch-case*, no obstante, con la variable *dato* se define la asignación de una constante de un conjunto de 16, por ello, una solución más simple involucra el uso de un arreglo de 16 constantes y el empleo de la variable *dato* como índice. La solución en lenguaje C es la siguiente:

```

#include      <avr/io.h>           // Definiciones de Registros I/O
#include      <avr/pgmspace.h>       // Acceso a memoria FLASH

// Tabla de constantes en memoria FLASH
const char  tabla[] PROGMEM = { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
                                0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71 };

int main() {

    unsigned char  dato;
    DDRD = 0x00;                           // Puerto D como entrada
    PORTD = 0xFF;                          // Habilita resistores de Pull-Up
    DDRB = 0xFF;                           // Puerto B como salida

    while(1) {                            // Lazo infinito
        // Lee las entradas y anula los 4
        // bits más significativos
        dato = PIND & 0x0F;                // Acceso a la FLASH
        PORTB = pgm_read_byte(&tabla[dato]);
    }
}

```

En lenguaje ensamblador también se debe observar una tabla de constantes en memoria de código y un registro funcionando como apuntador, para hacer las lecturas empleando direccionamiento indirecto. El código es el siguiente:

```
.include <mc8def.inc>

LDI    R16, 0x00          ; Puerto D como entrada
OUT   DDRD, R16
LDI    R16, 0xFF          ; Resistores de Pull-Up
OUT   PORTD, R16

OUT   DDRB, R16          ; Puerto B como salida
LOOP:
IN    R16, PIND          ; R16 se utiliza para la variable Dato
ANDI  R16, 0x0F

LDI    R31, HIGH (tabla << 1) ; Apuntador Z al inicio de la tabla
LDI    R30, LOW (tabla << 1)
ADD   R30, R16            ; Suma el índice a Z
BRCC s1
INC   R31                ; Si hay acarreo modifica la parte alta de Z
s1:
LPM   R17, Z              ; Carga de FLASH
OUT   PORTB, R17          ; Coloca la salida
RJMP  LOOP

; Tabla de constantes en memoria FLASH
tabla: .DB 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07
       .DB 0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71
```

Para la versión en ensamblador, con el posicionamiento del apuntador Z al comienzo de la tabla se realiza un desplazamiento a la izquierda de la constante, esto equivale a una multiplicación por 2 y se requiere porque la memoria está organizada en palabras de 16 bits, la etiqueta `tabla` hace referencia a una dirección de palabra y la instrucción `LPM` requiere una dirección de byte, cada palabra incluye 2 bytes. Por ello, si se utiliza un número impar de bytes como constantes, al ensamblar se genera una advertencia por la desalineación del código.

En este ejemplo también se mostró que existe una relación directa entre hardware y software, en aplicaciones con MCUs normalmente ocurre que es posible un ahorro de hardware si se escriben más líneas de código. O en caso contrario, si la memoria de código se ha agotado, algunas tareas de software podrían realizarse con hardware externo.

Respecto al ejemplo, al utilizar la AND para conservar sólo la parte baja del puerto D, hace innecesario tener que aterrizar vía hardware a la parte alta del mismo puerto, estas terminales pueden permanecer abiertas sin alterar el funcionamiento del sistema. También, los resistores de *Pull-Up* ya se encuentran dentro del AVR, sólo fue necesaria su habilitación para evitar el uso de resistores externos.

3.5.3 Diseño de una ALU de 4 Bits

En este ejemplo se muestra el uso de otra característica importante en la programación de MCUs, el uso de una tabla de saltos.

Ejemplo 3.3 Construya una ALU de 4 bits utilizando un ATMega8, en donde los operandos se lean del puerto B (nibble bajo para el operando A y nibble alto para el operando B), el resultado se genere en el puerto D y con los 3 bits menos significativos del puerto C se defina la operación. En la figura 3.19 se muestran las entradas, salidas y las operaciones.

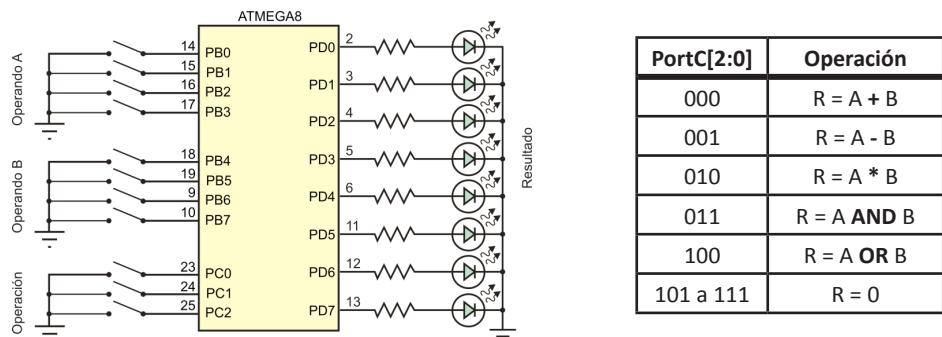


Figura 3.19 ALU de 4 bits con 5 operaciones

El comportamiento esperado para la ALU se muestra en el diagrama de flujo de la figura 3.20. Los 2 operandos se obtienen del mismo puerto, por lo que deben separarse utilizando operaciones lógicas.

Para el programa en lenguaje C, lo natural es el uso de una estructura *switch-case*, porque se tiene una operación diferente en cada caso, a diferencia del ejemplo anterior, en el cual únicamente se obtiene una constante.

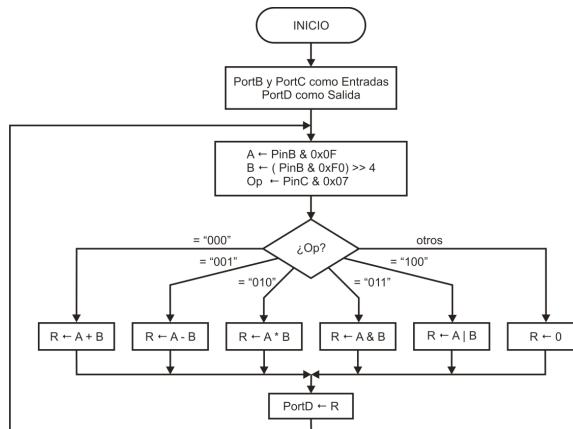


Figura 3.20 Comportamiento para la ALU

El programa en lenguaje C es:

```
#include <avr/io.h>

int main() {
    unsigned char A, B, R, Op; // Variables locales

    DDRB = 0x00; // Configura los puertos de entrada
    DDRC = 0x00;
    PORTB = 0xFF; // Resistores de Pull-Up
    PORTC = 0xFF;
    DDRD = 0xFF; // Puerto D como salida

    while(1) {

        A = PINB & 0x0F;
        B = (PINB & 0xF0) >> 4;
        Op = PINC & 0x07;

        switch( Op ) {
            case 0: R = A + B; // Suma
            break;
            case 1: R = A - B; // Resta
            break;
            case 2: R = A * B; // Producto
            break;
            case 3: R = A & B; // AND lógica
            break;
            case 4: R = A | B; // OR lógica
            break;
            default: R = 0;
        }

        PORTD = R; // Genera la salida
    }
}
```

No se considera la posibilidad de acarreos porque los datos son de 4 bits, para las 3 operaciones aritméticas el resultado alcanza perfectamente en 8 bits.

En una estructura *switch-case* se debe consumir el mismo tiempo para bifurcar a cada uno de los casos, ésta es la principal diferencia con un conjunto de estructuras *if-else* anidadas. Este comportamiento también debe observarse en ensamblador, para ello se utiliza una tabla de saltos, empleando bifurcaciones indirectas. El código en ensamblador correspondiente es:

```
.include <m8def.inc>

LDI    R16, 0x00 ; Puertos B y C como entradas
OUT   DDRB, R16
OUT   DDRC, R16
```

```

LDI    R16, 0xFF          ; Resistores de Pull-Up
OUT    PORTB, R16
OUT    PORTC, R16
OUT    DDRD, R16          ; Puerto D como salida

loop:
IN     R20, PINB          ; Lazo infinito
ANDI   R20, 0x0F          ; Se emplea R20 para el operando A
IN     R21, PINB          ; Se emplea R21 para el operando B
ANDI   R21, 0xF0
SWAP   R21
IN     R22, PINC          ; y R22 para la operación
ANDI   R22, 0x07

CPI    R22, 5             ; Observa si es un caso válido
BRGE  no_valido

valido:
LDI    R30, LOW(tabla); Z apunta al inicio de la tabla de saltos
LDI    R31, HIGH(tabla)
ADD   R30, R22            ; Suma el caso detectado
BRNE  no_carry
INC   R31                ; Se considera el acarreo
no_carry:
IJMP
; 

no_valido:
LDI    R22, 5             ; Todos los casos inválidos se etiquetan con 5
RJMP  valido              ; con 5 el caso ya es válido (default)

tabla:
RJMP  case_0              ; Tabla de saltos
RJMP  case_1
RJMP  case_2
RJMP  case_3
RJMP  case_4
RJMP  default

case_0:
MOV   R23, R20            ; Suma, el resultado queda en R23
ADD   R23, R21            ; para todos los casos
RJMP  salir

case_1:
MOV   R23, R20            ; Resta
SUB   R23, R21
RJMP  salir

case_2:
MUL   R21, R20            ; Producto
MOV   R23, R0              ; ubica el resultado
RJMP  salir

case_3:
MOV   R23, R20            ; AND lógica

```

```

AND      R23, R21
RJMP    salir
case_4:
    MOV      R23, R20          ; OR lógica
    OR       R23, R21
    RJMP    salir
default:
    CLR      R23              ; Operación no válida
salir:
    OUT     PORTD, R23        ; fin del switch-case
    RJMP    loop;             ; Genera la salida

```

En el código en ensamblador se observa como antes de tener acceso a la tabla de saltos se debe garantizar un caso válido, a los casos inválidos se les asigna el valor de 5, que corresponde con el caso por default.

3.6 Relación entre Lenguaje C y Ensamblador

En los ejemplos anteriores se ha mostrado un aspecto importante en el desarrollo de sistemas con microcontroladores. Antes de determinar qué lenguaje se va a emplear, es necesario un análisis de la solución, buscando que ésta sea óptima.

En ocasiones se dice que es “mejor” programar en ensamblador porque tiene una relación directa con el código máquina que se genera. Lo cual es acertado, al programar en alto nivel siempre se produce código adicional, debido a los mecanismos que los compiladores utilizan para el respaldo de variables durante llamadas a funciones y a las políticas en el uso de registros, que llevan a un uso exhaustivo de SRAM para variables.

Sin embargo, el aspecto más importante es la organización de la solución de un problema, un programa en ensamblador resultante de una mala o nula organización, puede ser más ineficiente que un programa en C, resultante de una propuesta de solución bien organizada.

La ventaja de emplear un lenguaje de alto nivel es que cuenta con estructuras de control de flujo que facilitan la codificación de soluciones estructuradas. Ante un problema con una complejidad de mediana a alta, la programación en alto nivel produce un código más compacto y menos confuso, reduciendo con ello la posibilidad de cometer errores.

Por otro lado, la velocidad de ejecución de las instrucciones y la cantidad de memoria con que actualmente cuentan los microcontroladores, hacen que el código adicional y el tiempo que se invierte en su ejecución no sea un factor determinante para no emplear un lenguaje de alto nivel en la mayoría de aplicaciones.

El lenguaje ensamblador sería necesario en las siguientes situaciones:

1. La aplicación requiere un control estricto en la temporización de algunas operaciones y los intervalos de tiempo requeridos no se puede conseguir con los temporizadores internos.
2. El tamaño de la memoria de código realmente es reducido, por ejemplo, algunos AVR de la gama Tiny incluyen 1 Kbyte en su memoria de programa.
3. La aplicación requiere una manipulación extensiva de bits, por ejemplo, para reducir el espacio utilizado para almacenar un conjunto de datos, en lenguaje ensamblador directamente puede hacerse un empaquetamiento de datos para comprimir la información.

Además, es posible ejecutar código ensamblador dentro de un programa en C utilizando la proposición `asm()`, la cual puede recibir hasta 4 argumentos con la siguiente sintaxis:

```
asm(código: operandos de salida: operandos de entrada [:restricciones]);
```

- **Código:** Cadena de texto entre comillas con la instrucción o instrucciones en ensamblador. Si es necesario pueden incluirse especificaciones de conversión con el carácter %.
- **Operandos de salida/entrada:** Lista de operandos que corresponden con las especificaciones de conversión indicadas en el código, pueden ser registros de propósito general o Registros I/O.
- **Restricciones:** Este argumento puede omitirse, se utiliza para indicar los registros que se están incluyendo en el código sin haber sido especificados como operandos de entrada o salida.

Los registros de propósito general son empleados por el compilador en el momento en que realiza la traducción a bajo nivel, por lo tanto, si se van a utilizar como operandos de salida o entrada, deben conocerse y respetarse las políticas que aplica el compilador para el manejo de registros.

Si se busca simplicidad en la escritura de código, sólo es recomendable incluir sentencias con ensamblador al realizar operaciones específicas con los Registros I/O. Algunos ejemplos, bajo este criterio de simplicidad, son:

```
asm("NOP");           // No operación, tarda 1 ciclo de reloj
asm("SBI 0x18, 0"); // Pone en alto al bit 0 de PORTB
asm("CBI 0x18, 0"); // Pone en bajo al bit 0 de PORTB
asm("SEI \n"         // Habilita las interrupciones y
    "CLC");          // limpia la bandera de acarreo
```

La instrucción ejecutada en el primer ejemplo únicamente sirve para perder 1 ciclo de reloj. Por lo que no afecta a los recursos del microcontrolador.

En el segundo y tercer ejemplo se hace referencia a un Registro I/O por su dirección y no por su nombre (PORTB), no es posible utilizar el nombre porque, para el código en ensamblador, no se ha incluido la biblioteca con las definiciones.

En el último ejemplo se han incluido dos instrucciones en la misma proposición, se inserta al carácter de nueva línea porque la sintaxis del lenguaje ensamblador sólo permite incluir una instrucción por línea.

3.7 Ejercicios

Se presenta una serie de problemas que pueden resolverse utilizando lenguaje ensamblador o C. Para la implementación, es posible el uso de un ATMega8 o de un ATMega16.

1. Emule el circuito combinacional mostrado en la figura 3.21, utilice un AVR programado en lenguaje Ensamblador. Sugerencia: Mueva cada bit a la posición menos significativa de un registro y aplique las operaciones lógicas sobre los registros. El resultado debe quedar en el bit menos significativo.

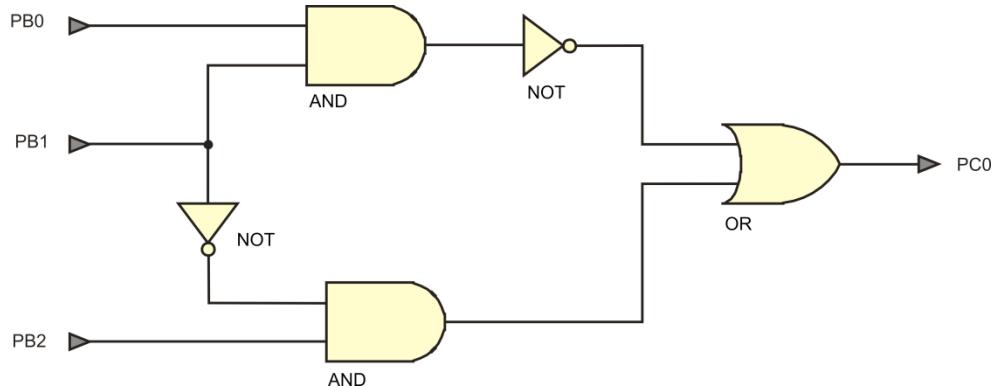


Figura 3.21 Circuito combinacional para el problema 1

Ante cambios en las entradas ¿Cuál es el tiempo de respuesta si el MCU está operando a 1 MHz?

2. Construya un comparador de 2 datos (A y B) de 4 bits leídos en el puerto B del microcontrolador, A en PB[3:0] y B en PB[7:4]. El comparador debe generar 3 salidas en el puerto C para indicar si: A > B (PC0), A = B (PC1) o A < B (PC2).

3. Realice un contador de segundos con salida en un display de 7 segmentos conectado en el puerto B de un AVR. Inicialmente muestre al 0, un segundo después, incremente para mostrar al 1 y así sucesivamente (0, 1, 2, etc.). Al llegar a la F, inicie nuevamente con el 0.
4. Modifique el problema 3 agregando un botón conectado en la terminal PD0 como se muestra en la figura 3.22 (a), habilite al resistor de Pull-Up para tener un 1 lógico mientras el botón se mantiene abierto. Organice el programa para que la salida se incremente en 1 cada vez que se presione el botón.

Al sondear un botón por software, debe considerarse que cuando el usuario lo presiona tarda un tiempo entre 150 y 300 mS, y como las operaciones del MCU están en el orden de microsegundos, la salida se incrementa en forma desmedida si no se inserta un retardo. Para ello, es conveniente utilizar un esquema como el mostrado en la figura 3.22 (b), con el cual también se va a eliminar ruido al sondear 2 veces al botón.

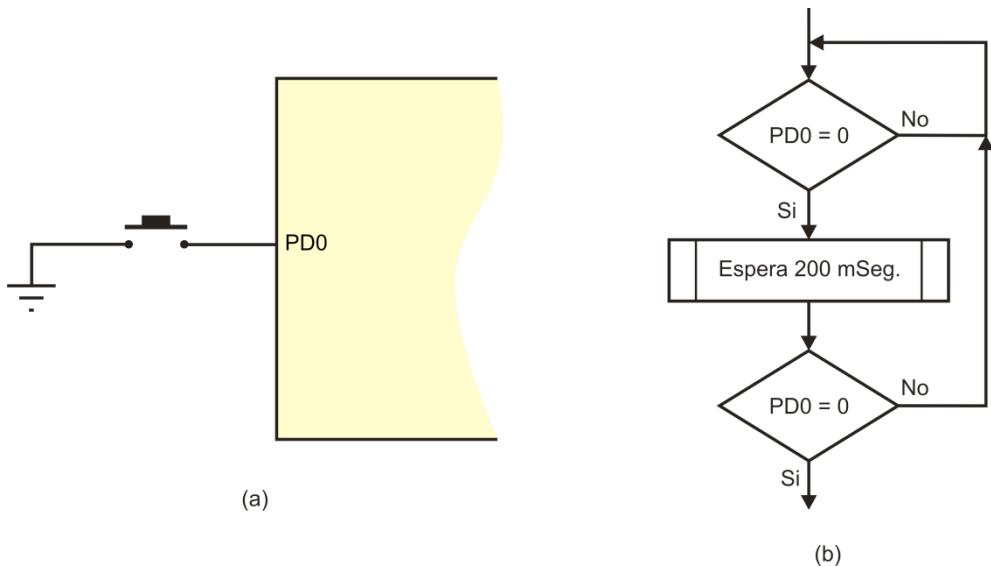


Figura 3.22 (a) Conexión de un botón y (b) espera a que el botón sea presionado

5. Implemente un sistema que maneje 2 semáforos con los 3 colores básicos (Rojo, Amarillo y Verde), siguiendo la secuencia de tiempos mostrada en la tabla 3.28.

Rojo1	Amarillo1	Verde1	Rojo2	Amarillo2	Verde2	T i e m p o (Seg)
1	0	0	0	0	1	15
1	0	0	0	0	parpadeo	5
1	0	0	0	1	0	5
0	0	1	1	0	0	15
0	0	parpadeo	1	0	0	5
0	1	0	1	0	0	5

Tabla 3.28 Secuencia para los semáforos

Para el parpadeo en el color verde, considere medio segundo encendido y medio segundo apagado.