DIRECTORIO DE PROFESORES DEL CURSO MICROPROCESADORES: TEORIA Y APLICACIONES, 1979

M. EN C. MANUEL ESTEVEZ KUBLI
CENTRO DE INSTRUMENTOS DE LA
UNAM
MEXICO 20, D.F.
TEL.550.04.16

M. EN C. ANGEL KURI MORALES
INVESTIGADOR
I.I.M.A.S.
UNAM
MEXICO 20, D.F.
TEL. 550.52.15 EXT.4573

ING. ALFREDO PATRON DE RUEDA
GERENTE DE MERCADOTECNIA
MEXEL S.A.
TLACOQUEMACATL 139-401
COL. DEL VALLE
MEXICO 12, D.F.
TEL. 575.79.24

M. EN C. MARIO RAMOS VALENZUELA
PROFESOR
UNIVERSIDAD AUTONOMA DE SINALOA
MAZATLAN, SIN.
TEL.9167137512

ING. MARIO RODRIGUEZ MANZANERA
INVESTIGADOR
I. I. M. A. S.
UNAM
MEXICO 20, D.F.

'edcs.

# MICROPROCESADORES: TEORIA Y APLICACIONES
(del 26 de marzo al 6 de abril)
1979

| FECHA | HORARIO | TEMA | PROFESORES |
|---|---|---|---|
| 26 de marzo | 17 a 21 h | 1. RESUMEN DEL DESARROLLO DE MICROPROCESADORES | M. en C. Angel Kuri Morales |
| 27 de marzo | 17 a 19 h | 2. INTRODUCCION A LA ARQUITECTURA DE MICROPROCESADORES | M. en C. Angel Kuri Morales |
| 27 de marzo1 | 19 a 21 h | 3. LA MICROCOMPUTADORA INTEL 8080 | Ing. Mario Rodríguez Manzanera |
| 28 de marzo | 17 a 21 h | LA MICROCOMPUTADORA INTEL 8080 | Ing. Mario Rodríguez Manzanera |
| 29 de marzo | 17 a 21 h | 4. LA MICROCOMPUTADORA INTEL 8080 | M. en C. Mario Ramos Valenzuela |
| 30 de marzo | 17 a 21 h | LA MICROCOMPUTADORA INTEL 8080 | M. en C. Mario Ramos Valenzuela |
| 2 de abril | 17 a 21 h | 5. LA MICROCOMPUTADORA MOTOROLA 6800 | M. en C. Manuel Estevez Kubli |
| 3 de abril | 17 a 21 h | 6. LA MICROCOMPUTADORA MOTOROLA 6800 | M. en C. Manuel Estevez Kubli |
| 4.de abril | 17 a 21 h | 7. LA MICROCOMPUTADORA MOZTEK Z80 | Ing. Alfredo Patrón de Rueda |
| 5 de abril | 17 a 21 h | 8. LA MICROCOMPUTADORA MOZTEK Z80 | M. en C. Angel Kuri Morales |
| 6 de abril | 17 a 21 h | 9. DEMOSTRACION DE UN SISTEMA DE DESARROLLO | Ing. Alfredo Patrón de Rueda y<br>M. en C. Angel Kuri Morales |
|  |  | CLAUSURA |  |

'pmc

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# INTRODUCCION

M.EN C. ANGEL KURI MORALES

MARZO, 1979

# INTRODUCCION

El Mercado para las Microcomputadoras ha crecido enormemente en los últimos años y la rapidez de su crecimiento de nuevos productos para este Mercado es muy grande.

El Microprocesador Z-80 fué diseñado por la Compañía ZILOG para participar en este Mercado.

Entre algunas de sus principales características tenemos las siguientes:

1. El Z-80 es completamente compatible en Software con el Microprocesador 8080 A, que actualmente es Circuito standard en el Mercado y es fabricado por diferentes compañías tales como INTEL, NATIONAL SEMICONDUCTOR, NEC, AMD, etc.

2. Las características de Software y Hardware del Z-80 son superiores a la mayoría de los microprocesadores de 8 bits que compiten en el Mercado actual.

3. Existe una versión Z-80A la cual opera a 4MHZ, lo que le da una ventaja significativa en velocidad de operación.

Un Sistema típico de Microcomputadora con el Z-80 consiste básicamente de:

1. CPU (Unidad Central de Proceso)

2. Memoria

3. Circuito de Interface para dispositivos periféricos.

El CPU es el corazón del Sistema, su función es obtener instrucciones de la Memoria y ejecutar las operaciones deseadas. La Memoria es usada para contener las instrucciones, en la mayoría de los casos los datos que van a ser procesados. Por ejemplo: una secuencia de instrucción típica sería leer datos de un periférico específico, guardarlas en memoria, checar la paridad y escribirlos en otro dispositivo periférico.

## ARQUITECTURA DEL Z-80

En la Figura No. 2 se muestra un diagrama a bloques de la arquitectura interna del CPU Z-80.

El CPU Z-80 contiene 208 Bits de memoria RAM estática, los cuales son accesibles al Programador.

La Figura 3 ilustra como esta memoria está configurada a 18 registros de 8 bits y 4 registros de 16 bits. Los registros incluyen dos conjuntos de 6 registros de propósito general, los cuales pueden ser usados individualmente como registros de 8 bits ó en pares como registro de 16 bits. También hay los conjuntos de registros acumuladores y banderas.

## REGISTROS DE PROPOSITO GENERAL

### 1. CONTADOR DE PROGRAMA (PC)

El Contador de Programa contiene la dirección con 16 bits de la instrucción a descargar de la memoria, este contador es incrementado automáticamente después que su contenido ha sido transferido al bus de direcciones.

Cuando ocurre un salto por programa, el nuevo valor es puesto automáticamente, sin tomar en cuenta el incrementador automático.

### 2. VECTOR DEL STACK (SP)

Este vector contiene la dirección más alta con 16 bits de un stack localizado en cualquier dirección de Memoria RAM Externa. Este stack está organizado como un file con última entrada - primera salida (LIFO). Este stack permite la implementación simple de interrupciones de niveles múltiples, inplementación de subrutinas y la simplificación de la manipulación de datos.

### 3. REGISTROS INDEXADOS (IX & IY)

Estos dos registros independientes contienen direcciones base en 16 bits las cuales son usadas en los modos de direccionamiento Indexado.

En este modo, un registro indexado es usado como base para apuntar a una región de memoria de la cual se va a leer ó escribir datos. En este tipo de instrucción se incluye un byte adicional para especificar el desplazamiento desde la base.

4. REGISTROS DE DIRECCION DE PAGINA PARA INTERRUPCION (I)

El Z-80 puede ser operado en un modo en el cual una llamada indirecta para cualquier dirección de memoria puede ser obtenida en respuesta a una interrupción.

El Registro I es usado para este propósito, al guardar los últimos 8 bits de la dirección indirecta, mientras que el dispositivo que genera la interrupción provee los primeros 8 bits de la dirección.

5. REGISTRO PARA REFRESCAR MEMORIAS (R)

Este registro permite refrescar los datos cuando se usa memoria dinámica, siete de los ocho bits de este registro son incrementados automáticamente después de cada descarga de una instrucción. El bit No. 8 permanecerá como fué programado como resultado de una instrucción L, D R, A. Los datos en este registro contador son enviados en los primeros 8 bits del bus de dirección al mismo tiempo que la señal de control para refrescar, mientras que el CPU está decodificando y ejecutando la instrucción descargada. Este modo de operación es transparente al Programador y no reduce la velocidad de operación del CPU.

REGISTROS DE BANDERA Y ACUMULADORES

El CPU cuenta con dos acumuladores de 8 bits independientes y sus registros de bandera asociados.

El acumulador contiene los resultados de las operaciones en 8 bits aritméticas ó lógicas, mientras que el registro bandera indica condiciones específicas para operaciones con 8 ó 16 bits, tales como resultados iguales a cero.

REGISTRO DE PROPOSITO GENERAL

El CPU contiene 2 conjuntos iguales de registro de propósito general, cada conjunto conteniendo seis registros de 8 bits, los cuales pueden ser usados individualmente como registro de 8 bits ó como registro de 16 bits, usándolos por pares. Un conjunto es llamado BC, DE y HL, mientras que el conjunto complementario es llamado BD', DE' y HC'.

## UNIDAD DE LOGICA Y ARITMETICA (ALU)

Las instrucciones en 8 bits lógicas y aritméticas son ejecutadas en esta unidad. Internamente el ALU se comunica con los registros y con el BUS de datos externo, con un BUS de datos interno. El tipo de funciones que hace el ALU incluye:

Suma:

Resta:
and Logica
or "
or Exclusiva
Comparación

Cambios y rotaciones (aritméticas y lógicas).
Incrementos.
Decrementos
Encendido de bits
Apagado de bits
Prueba de bits

AO - A15
(Bus de direcciones)

Salida de triple estado, activas altas, puede direccionar hasta 64K bytes, para el direccionamiento de dispositivos I/O se usan los primeros 8 bits para permitir la selección de 256 puertos de entradas ó salidas, durante el tiempo de refresco, los 7 primeros bits contienen direcciónes válidas de refresco de memoria.

DO B7
(Bus de Datos)

Salidas-Entradas de tres estados, activas altas, se usa para intercambio de datos con la memoria y dispositivos I/O.

$\overline{M\ 1}$
Ciclo de Máquinas

Señal de Salida activa baja, indica que

X...

el CPU está ejecutando un ciclo de descarga de una instrucción para ejecución. También se presenta cuando ocurre IORQ para indicar un ciclo de interrupción aceptado.

$\overline{\text{MREQ}}$ Salida de tres estados, activa baja, indica que el bus de dirección contiene una dirección válida para una operación de lectura ó escritura en memoria.

$\overline{\text{IORQ}}$ Salida de tres estados, activa baja, indica que los primeros 8 bits del bus de dirección contiene una dirección de dispositivo I/O para una operación de lectura ó escritura en el mismo.

$\overline{\text{RO}}$ Salida de tres estados, activa baja, indica que el CPU quiere leer datos de memoria ó de un dispositivo I/O.

$\overline{\text{WR}}$ Salida de tres estados. Activa baja, Indica que el bus de datos del CPU contiene datos válidos para ser almacenados en memoria ó en un dispositivo I/O.

$\overline{\text{RFSH}}$ Salida, activa baja, indica que los primeros 7 bits del bus de dirección contienen una dirección para refrescar memoria dinámica.

$\overline{\text{HALT}}$ Salida, activa baja, indica que el CPU ha ejecutado en instrucción de PARO y está esperando una interrupción enmascarable ó no enmascarable antes que la operación sea reanudada.

$\overline{\text{WAIT}}$

Entrada, activa baja, indica al CPU que el dispositivo I/O ó de Memoria direccionado no está listo para transferir sus datos. El CPU continúa esperando mientras esta señal esté presente.

$\overline{\text{INT}}$
(Interrupción
Requerida)

Entrada, activa baja, esta señal es generada por el dispositivo I/O que requiere.una rutina de servicio. Esta señal será aceptada al finalizar la instrucción en ejecución si el Flip Flop interno (IFF) controlado por programa está activado y si la señal BUSRQ no está activa, cuando el CPU acepta la intervención, se envía una señal de reconocimiento IORQ durante el tiempo M1 al empiezo del siguiente ciclo de instrucción. Hay tres modos diferentes de interrupción.

$\overline{\text{NMI}}$
(Interrupción No
Enmascarable)

Entrada, disparada en la cída negativa, esta señal tiene mayor prioridad que INT y siempre será reconocida al final de la instrucción corriente independientemente del status, del Flip Flop de interrupción, NMI Forza automáticamente al CPU del Z-80 a iniciar en la dirección 0066H.

$\overline{\text{RESET}}$

Entrada, activa baja, Forza al contador de programa a cero e inicializa al CPU.

$\overline{\text{BUSRQ}}$

Entrada, activa baja, esta señal es usada para pedir a los buses de dirección, de datos y de control del CPU se vayan a un estado de alta impedancia para permitir a

X...

otros dispositivos el control de dichos buses.

$\overline{\text{BUSAK}}$ Salida, activa baja, se usa para indicar al sispositivo requiriente que los buses del CPU están en alta impedancia.

## T1
## CONTROL DEL TIEMPO DEL CPU

El CPU ejecuta instrucciones por medio de un conjunto preciso de operaciones básicas que incluyen:

Lectura ó Escritura en Memoria
Lectura ó Escritura en Dispositivos I/O
Reconocimiento de Interrupciones.

Todas las instrucciones son meramente series de estas operaciones básicas, cada una de estas operaciones básicas puede tomar de tres a seis períodos de reloj para completarlos ó pueden ser retardados para sincronizar la velocidad del CPU con la de dispositivos externos. Los períodos de reloj básicos son referidos como ciclos T y las operaciones básicas como ciclos M. La figura 5 ilustra como una instrucción típica consiste de una serie específica de ciclos M y T. Hay que notar que la instrucción consiste de tres ciclos de máquina (M1, M2 y M3) El primer ciclo de máquina de cualquier instrucción consiste en un ciclo de descarga, el cual dura 5, 5 ó 6 ciclos T. El ciclo M1 es usado para descargar el código operativo de la siguiente instrucción para ser ejecutada. Los ciclos subsecuentes mueven datos entre el CPU y la memoria ó los dispositivos I/O y pueden tener de 3 a 5 ciclos T.

Toda la operación en el tiempo del CPU puede ser desglosada en diagramas muy simples de tiempo, tales como los mostrados a continuación. Estos diagramas muestran las siguientes operaciones básicas.

X...

Descarga de Código Operativo de Instrucción (Ciclo M1)

Ciclo de Lectura ó Escritura en Memoria

Ciclo de Lectura ó Escritura en Dispositivo I/O

Ciclo de Reconocimiento de requerimiento de Bus

Ciclo de Reconocimiento de requerimiento de Interrupción

Ciclo de Reconocimiento de requerimiento de Interrupción no enmascarable.

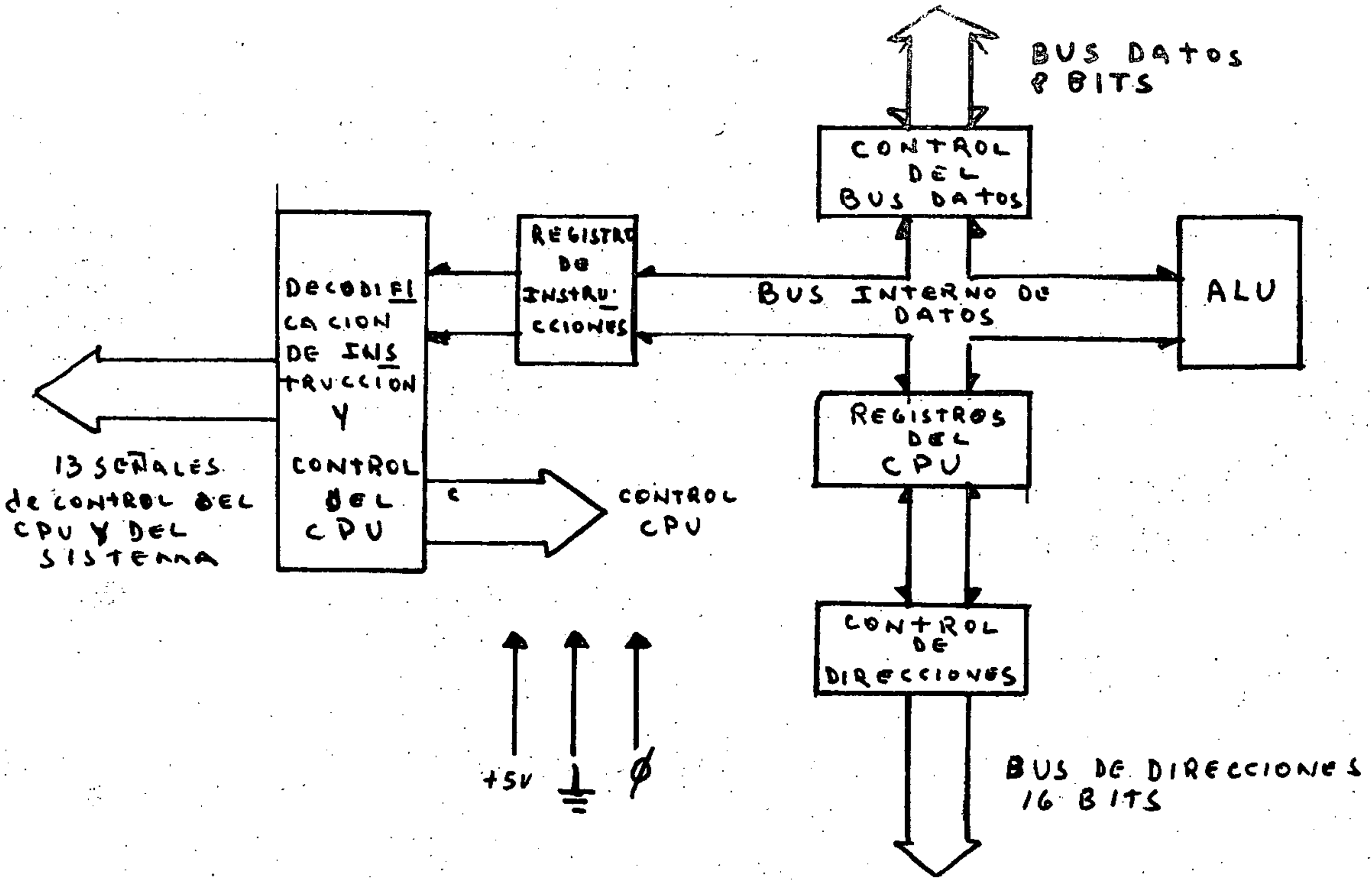
Salida de Instrucción HALT.

AP'ltre

# COMPARACION DE CARACTERISTICAS DE MICROPROCESADORES

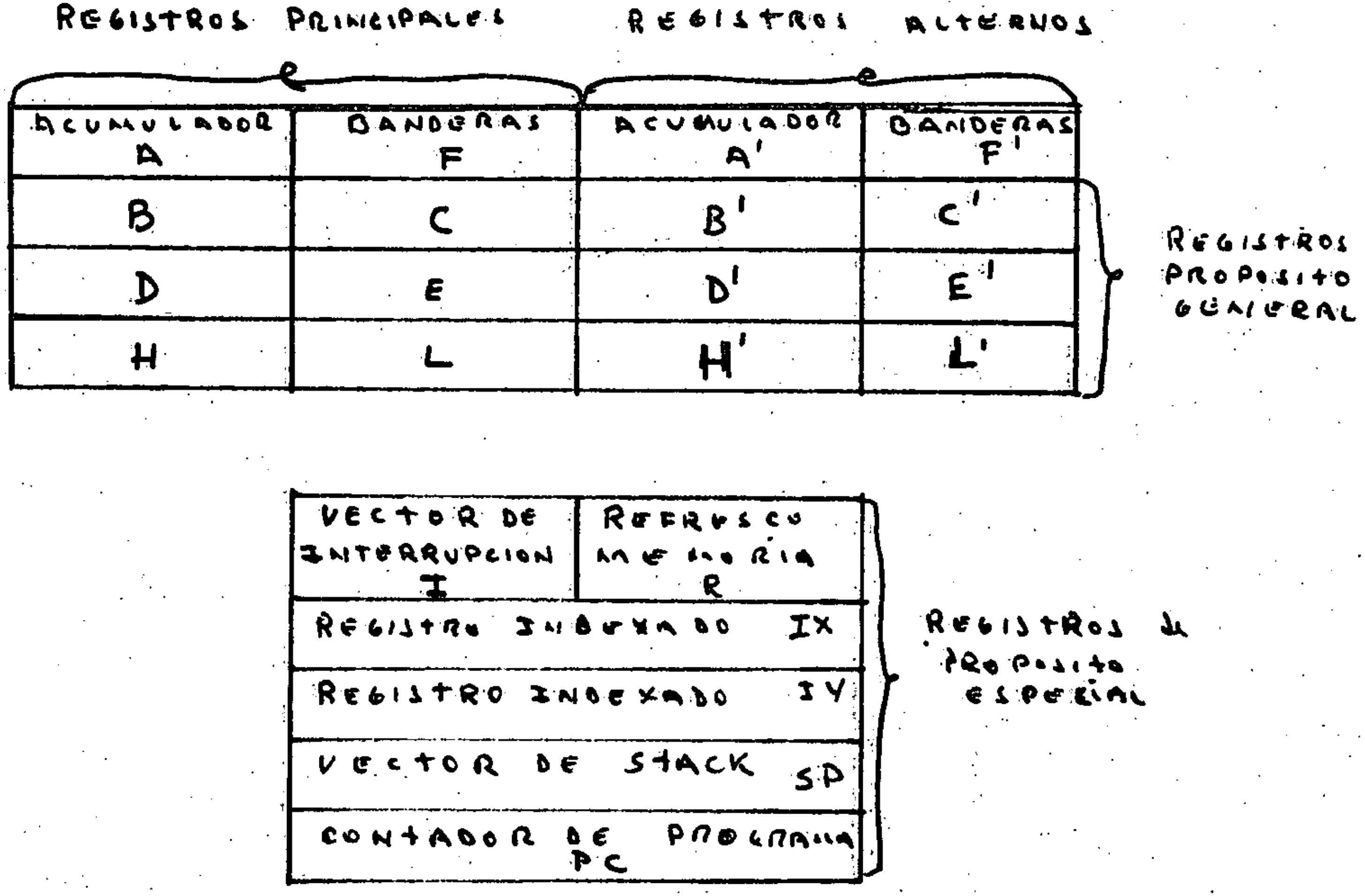
| | 8080 | Z80 | Z80A | Z8000 |
|---|---|---|---|---|
| Fecha de Inicio de Producción | 1974 | 1976 | 1977 | 1978 |
| Consumo de Potencia (Watts) | 1.2 | 1.0 | 1.0 | 1.5 |
| Número de Transistores | 4,800 | 8,200 | 8,200 | 17,500 |
| Número de Compuerta | 1,600 | 2,733 | 2,733 | 5,833 |
| Tamaño del Chip (mm2) | 22.3 | 27.1 | 22.4 | 39.3 |
| Densidad (Compuertas/mm2) | 72 | 101 | 122 | 148 |
| Número de Instrucciones distintas | 34 | 52 | 52 | 81 |
| Combinación de Instrucciones distintas, tipos de datos y modos de direccionamiento | 65 | 128 | 128 | 414 |

Marzo/79
AP'ltre



ARQUITECTURA DEL Z-80

FIG 2



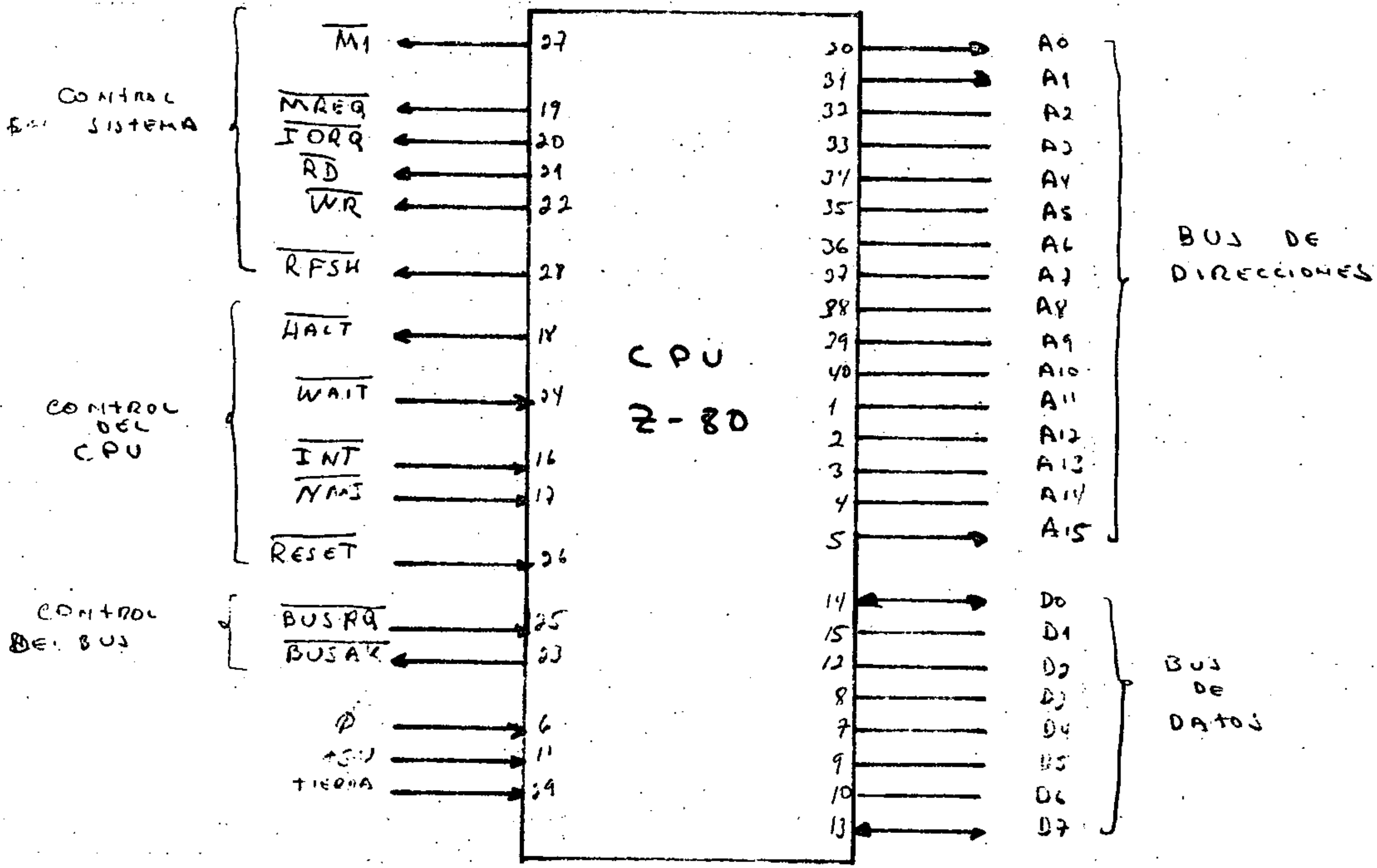
| REGISTROS PRINCIPALES | | REGISTROS ALTERNOS | |
|---|---|---|---|
| ACUMULADOR<br>A | BANDERAS<br>F | ACUMULADOR<br>A' | BANDERAS<br>F' |
| B | C | B' | C' |
| D | E | D' | E' |
| H | L | H' | L' |

REGISTROS PROPOSITO GENERAL

| | |
|---|---|
| VECTOR DE INTERRUPCION<br>I | REFRESCO MEMORIA<br>R |
| REGISTRO INDEXADO IX | |
| REGISTRO INDEXADO IY | |
| VECTOR DE STACK SP | |
| CONTADOR DE PROGRAMA<br>PC | |

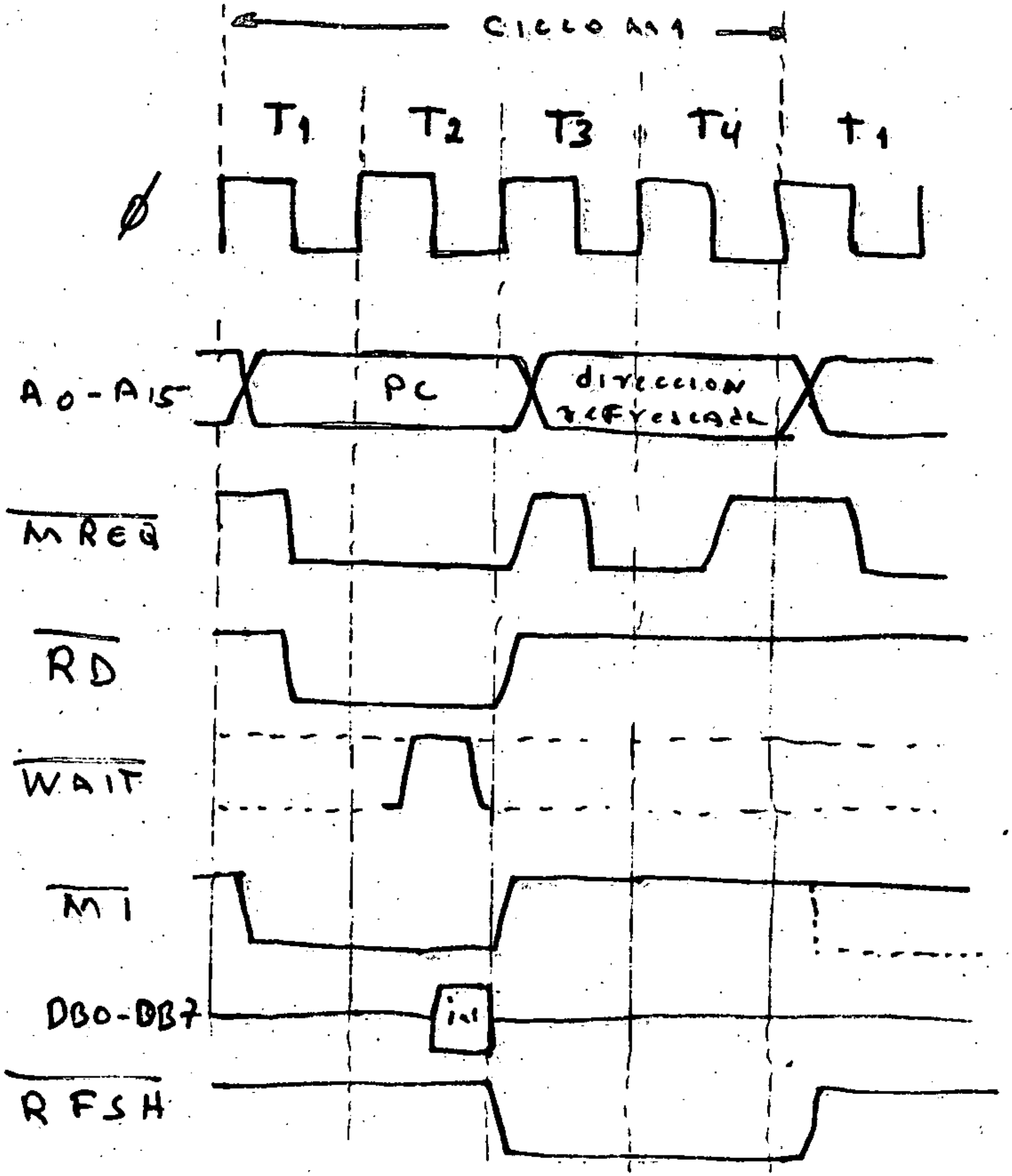
REGISTROS DE PROPOSITO ESPECIAL

FIG — 3

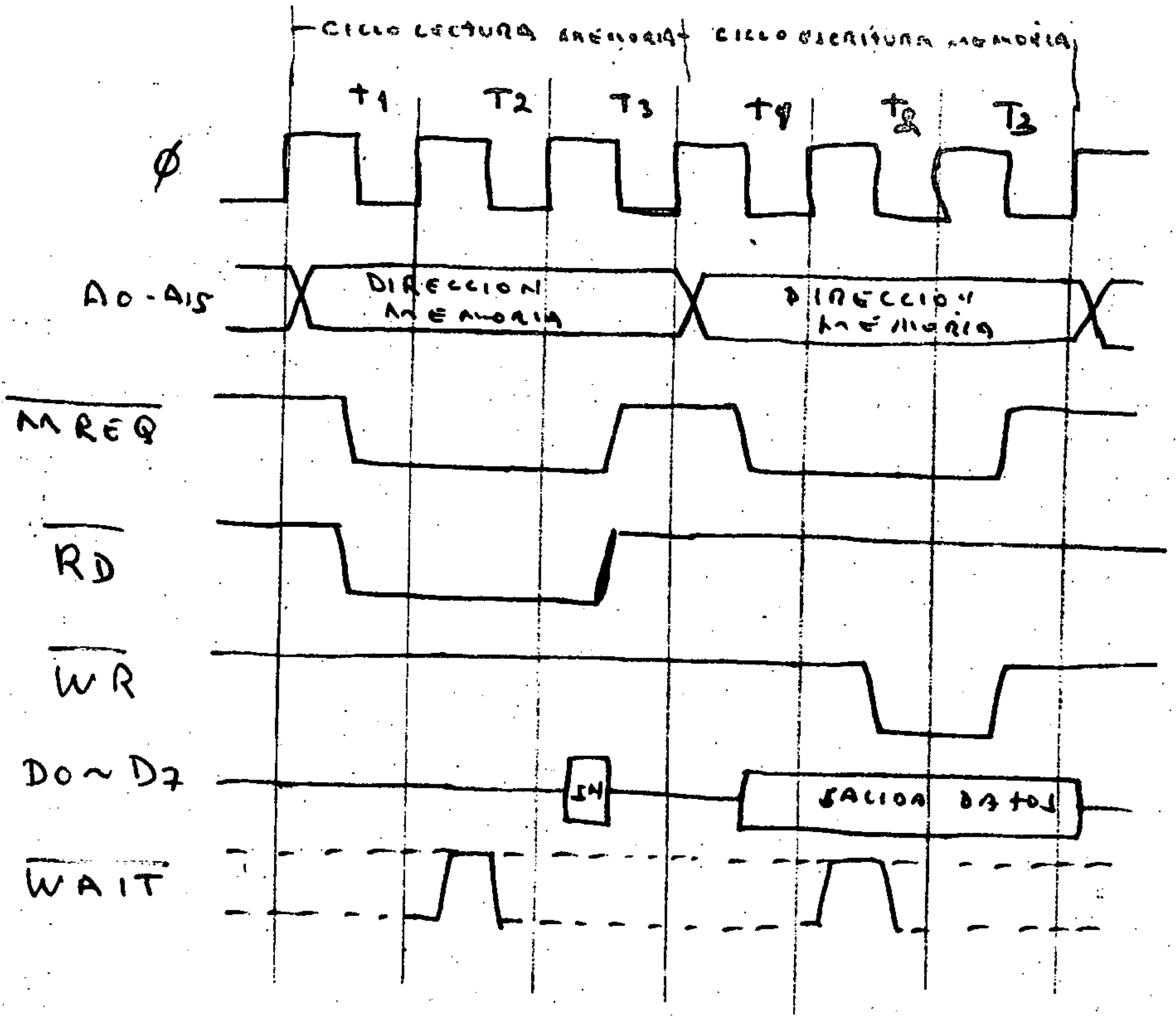


CONFIGURACION DEL Z-80

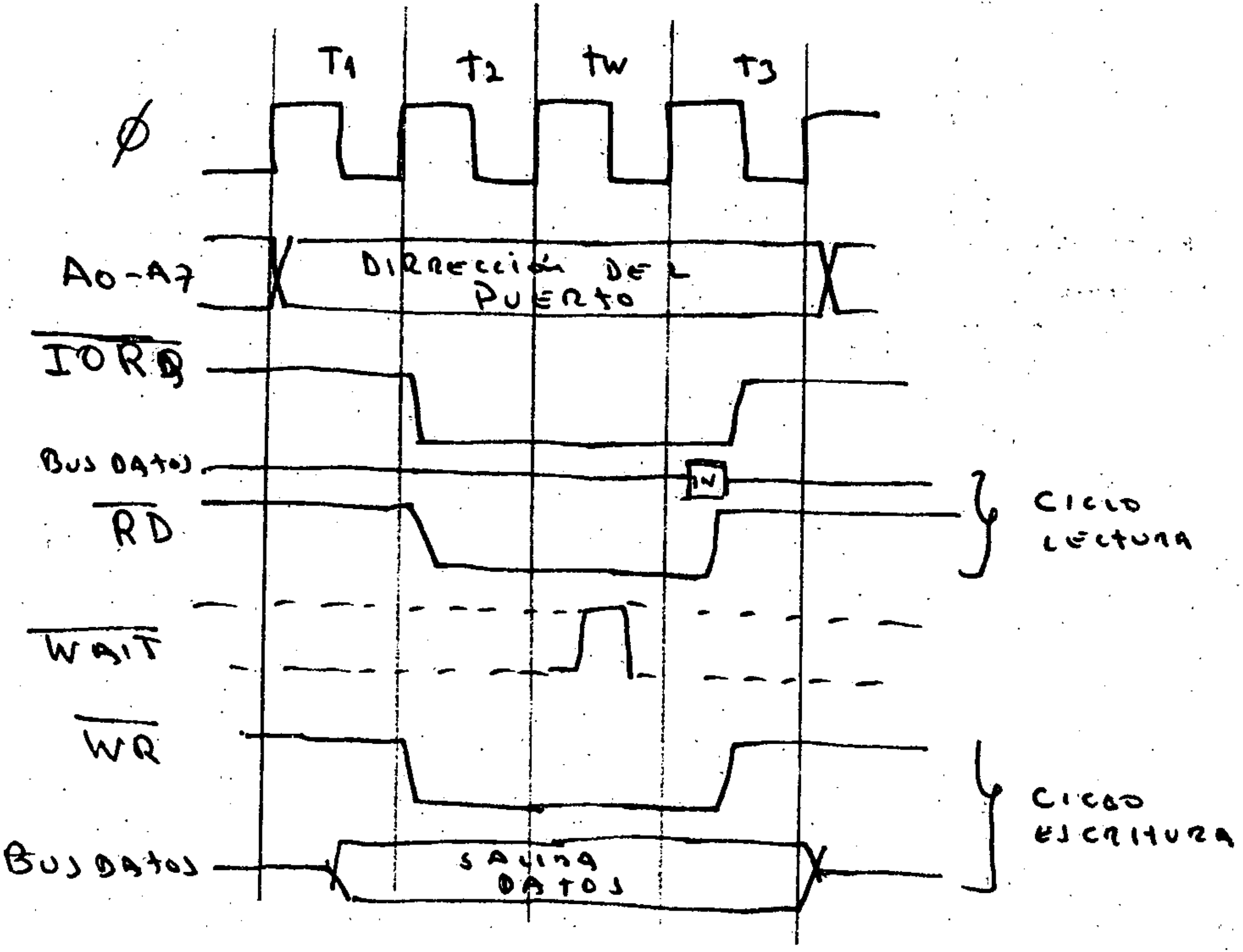
FIG 4



DESCARGA de CODIGO OPERATIVO
DE INSTRUCCION

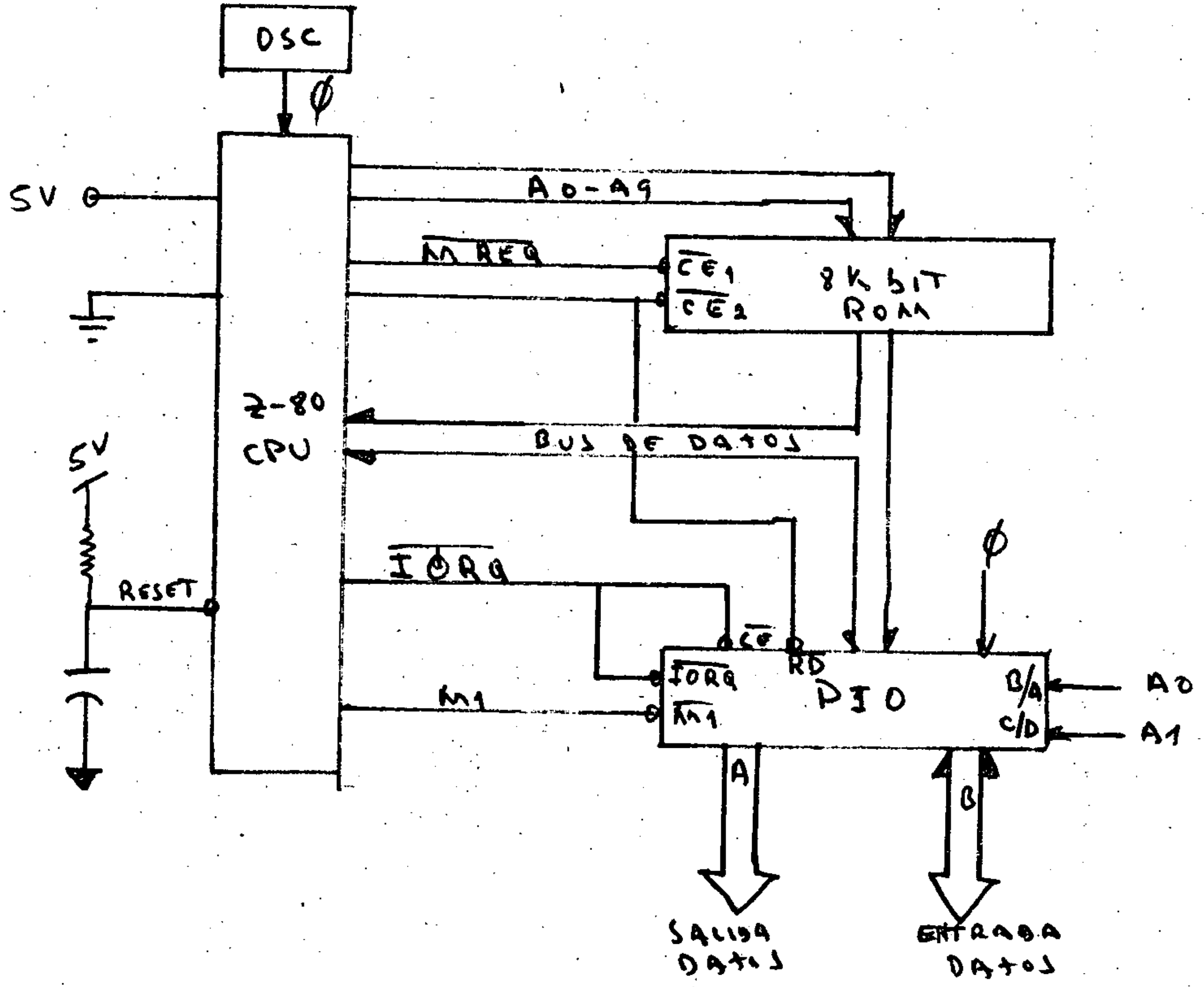


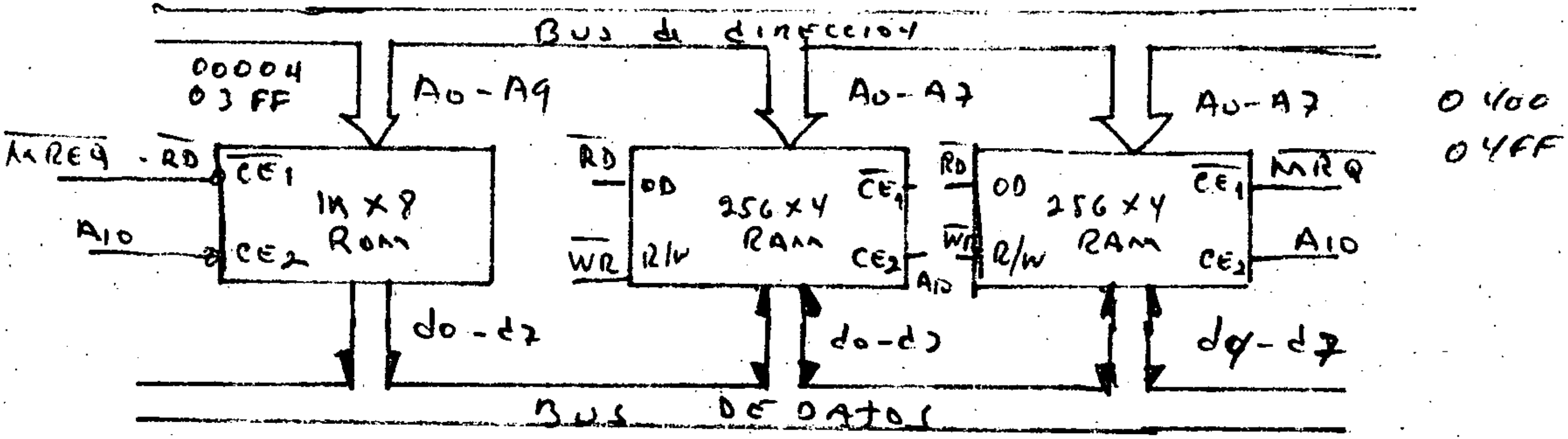
CICLOS de LECTURA O ESCRITURA EN MEMORIA



CICLOS DE ENTRADA
y SALIDA

# SISTEMA MINIMO





IMPLEMENTACIÓN DE MEMORIA

# Z80®-CPU
# Z80A-CPU

Zilog®

# Product Specification

MARCH 1978

The Zilog Z80 product line is a complete set of microcomputer components, development systems and support software. The Z80 microcomputer component set includes all of the circuits necessary to build high-performance microcomputer systems with virtually no other logic and a minimum number of low cost standard memory elements.

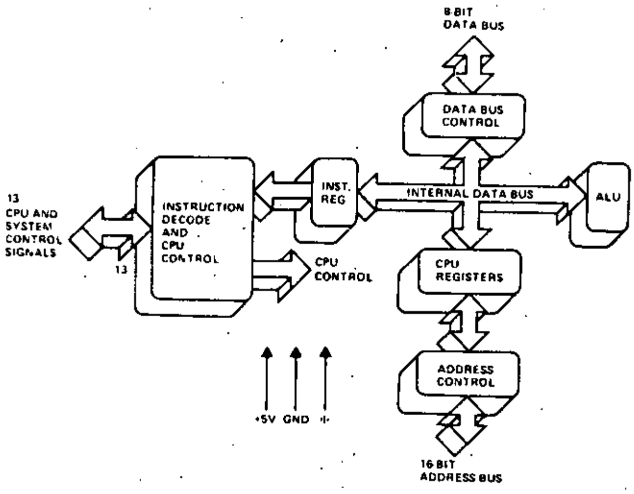
The Z80 and Z80A CPU's are third generation single chip microprocessors with unrivaled computational power. This increased computational power results in higher system through-put and more efficient memory utilization when compared to second generation microprocessors. In addition, the Z80 and Z80A CPU's are very easy to implement into a system because of their single voltage requirement plus all output signals are fully decoded and timed to control standard memory or peripheral circuits. The circuit is implemented using an N-channel, ion implanted, silicon gate MOS process.

Figure 1 is a block diagram of the CPU, Figure 2 details the internal register configuration which contains 208 bits of Read/Write memory that are accessible to the programmer. The registers include two sets of six general purpose registers that may be used individually as 8-bit registers or as 16-bit register pairs. There are also two sets of accumulator and flag registers. The programmer has access to either set of main or alternate registers through a group of exchange instructions. This alternate set allows foreground/background mode of operation or may be reserved for very fast Interrupt response. Each CPU also contains a 16-bit stack pointer which permits simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data handling.

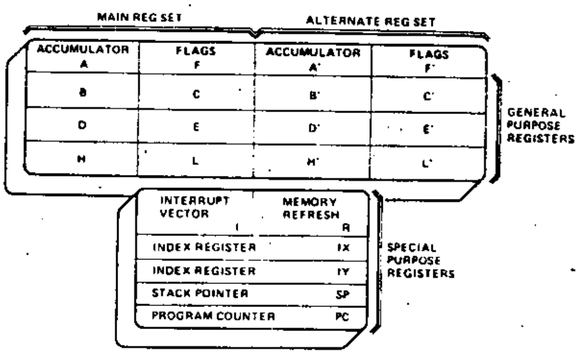
The two 16-bit index registers allow tabular data manipulation and easy implementation of relocatable code. The Refresh register provides for automatic, totally transparent refresh of external dynamic memories. The I register is used in a powerful interrupt response mode to form the upper 8 bits of a pointer to a interrupt service address table, while the interrupting device supplies the lower 8 bits of the pointer. An indirect call is then made to this service address.

## FEATURES

- Single chip, N-channel Silicon Gate CPU.
- 158 instructions–includes all 78 of the 8080A instructions with total software compatibility. New instructions include 4-, 8- and 16-bit operations with more useful addressing modes such as indexed, bit and relative.
- 17 internal registers.
- Three modes of fast interrupt response plus a non-maskable interrupt.
- Directly interfaces standard speed static or dynamic memories with virtually no external logic.
- 1.0 μs instruction execution speed.
- Single 5 VDC supply and single-phase 5 volt Clock.
- Out-performs any other single chip microcomputer in 4-, 8-, or 16-bit applications.
- All pins TTL Compatible
- Built-in dynamic RAM refresh circuitry.

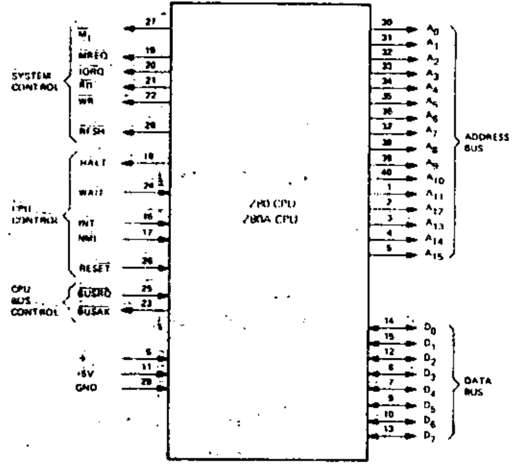


Z80, Z80A CPU BLOCK DIAGRAM



Z80, Z80A CPU REGISTERS

# Z80, Z80A-CPU Pin Description



**Z80, Z80A CPU PIN CONFIGURATION**

$A_0$-$A_{15}$ (Address Bus)

Tri-state output, active high. $A_0$-$A_{15}$ constitute a 16-bit address bus. The address bus provides the address for memory (up to 64K bytes) data exchanges and for I/O device data exchanges.

$D_0$-$D_7$ (Data Bus)

Tri-state input/output, active high. $D_0$ - $D_7$ constitute an 8-bit bidirectional data bus. The data bus is used for data exchanges with memory and I/O devices.

$\overline{M_1}$ (Machine Cycle one)

Output, active low. $\overline{M_1}$ indicates that the current machine cycle is the OP code fetch cycle of an instruction execution.

$\overline{\text{MREQ}}$ (Memory Request)

Tri-state output, active low. The memory request signal indicates that the address bus holds a valid address for a memory read or memory write operation.

$\overline{\text{IORQ}}$ (Input/Output Request)

Tri-state output, active low. The $\overline{\text{IORQ}}$ signal indicates that the lower half of the address bus holds a valid I/O address for a I/O read or write operation. An $\overline{\text{IORQ}}$ signal is also generated when an interrupt is being acknowledged to indicate that an interrupt response vector can be placed on the data bus.

$\overline{\text{RD}}$ (Memory Read)

Tri-state output, active low. $\overline{\text{RD}}$ indicates that the CPU wants to read data from memory or an I/O device. The addressed I/O device or memory should use this signal to gate data onto the CPU data bus.

$\overline{\text{WR}}$ (Memory Write)

Tri-state output, active low. $\overline{\text{WR}}$ indicates that the CPU data bus holds valid data to be stored in the addressed memory or I/O device.

$\overline{\text{RFSH}}$ (Refresh)

Output, active low. $\overline{\text{RFSH}}$ indicates that the lower 7 bits of the address bus contain a refresh address for dynamic memories and the current $\overline{\text{MREQ}}$ signal should be used to do a refresh read to all dynamic memories.

$\overline{\text{HALT}}$ (Halt state)

Output, active low. $\overline{\text{HALT}}$ indicates that the CPU has executed a HALT software instruction and is awaiting either a non-maskable or a maskable interrupt (with the mask enabled) before operation can resume. While halted, the CPU executes NOP's to maintain memory refresh activity.

$\overline{\text{WAIT}}$ (Wait)

Input, active low. $\overline{\text{WAIT}}$ indicates to the Z-80 CPU that the addressed memory or I/O devices are not ready for a data transfer. The CPU continues to enter wait states for as long as this signal is active.

$\overline{\text{INT}}$ (Interrupt Request)

Input, active low. The Interrupt Request signal is generated by I/O devices. A request will be honored at the end of the current instruction if the internal software controlled interrupt enable flip-flop (IFF) is enabled.

$\overline{\text{NMI}}$ (Non Maskable Interrupt)

Input, active low. The non-maskable nterrupt request line has a higher priority than $\overline{\text{INT}}$ and is always recognized at the end of the current instruction, independent of the status of the interrupt enable flip-flop. $\overline{\text{NMI}}$ automatically forces the Z-80 CPU to restart to location $0066_H$.

$\overline{\text{RESET}}$

Input, active low. $\overline{\text{RESET}}$ initializes the CPU as follows: reset interrupt enable flip-flop, clear PC and registers I and R and set interrupt to 8080A mode. During reset time, the address and data bus go to a high impedance state and all control output signals go to the inactive state.

$\overline{\text{BUSRQ}}$ (Bus Request)

Input, active low. The bus request signal has a higher priority than $\overline{\text{NMI}}$ and is always recognized at the end of the current machine cycle and is used to request the CPU address bus, data bus and tri-state output control signals to go to a high impedance state so that other devices can control these busses.

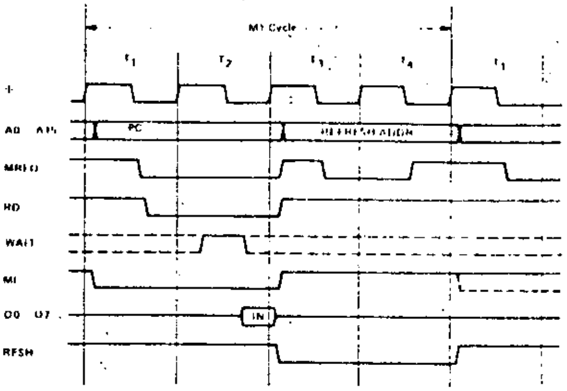
$\overline{\text{BUSAK}}$ (Bus Acknowledge)

Output, active low. Bus acknowledge is used to indicate to the requesting device that the CPU address bus, data bus and tri-state control bus signals have been set to their high impedance state and the external device can now control these signals.

# Timing Waveforms

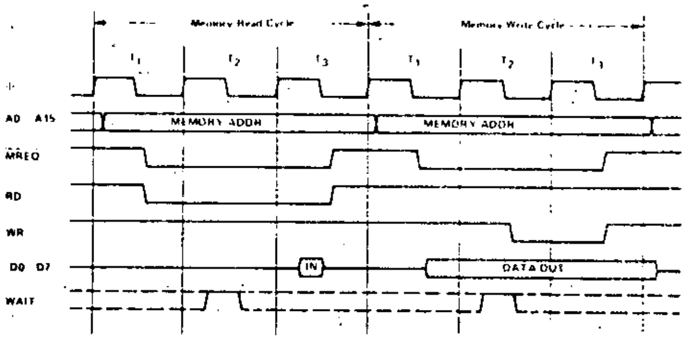
## INSTRUCTION OP CODE FETCH

The program counter content (PC) is placed on the address bus immediately at the start of the cycle. One half clock time later $\overline{MREQ}$ goes active. The falling edge of $\overline{MREQ}$ can be used directly as a chip enable to dynamic memories. $\overline{RD}$ when active indicates that the memory data should be enabled onto the CPU data bus. The CPU samples data with the rising edge of the clock state $T_3$. Clock states $T_3$ and $T_4$ of a fetch cycle are used to refresh dynamic memories while the CPU is internally decoding and executing the instruction. The refresh control signal $\overline{RFSH}$ indicates that a refresh read of all dynamic memories should be accomplished.



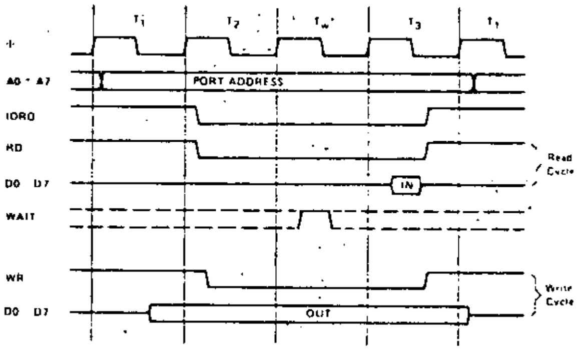
## MEMORY READ OR WRITE CYCLES

Illustrated here is the timing of memory read or write cycles other than an OP code fetch ($M_1$ cycle). The $\overline{MREQ}$ and $\overline{RD}$ signals are used exactly as in the fetch cycle. In the case of a memory write cycle, the $\overline{MREQ}$ also becomes active when the address bus is stable so that it can be used directly as a chip enable for dynamic memories. The $\overline{WR}$ line is active when data on the data bus is stable so that it can be used directly as a R/W pulse to virtually any type of semiconductor memory.



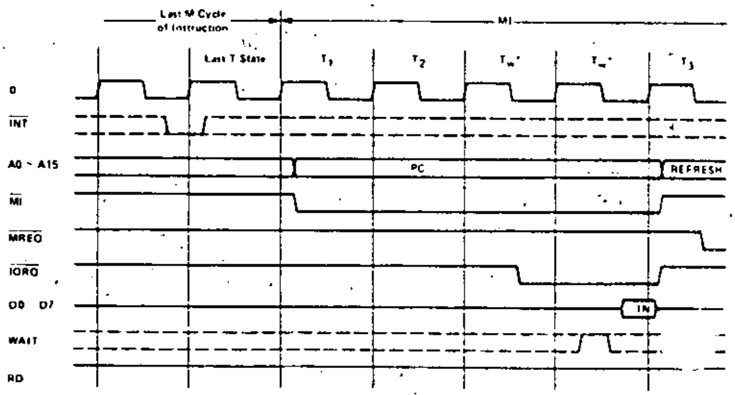
## INPUT OR OUTPUT CYCLES

Illustrated here is the timing for an I/O read or I/O write operation. Notice that during I/O operations a single wait state is automatically inserted (Tw*). The reason for this is that during I/O operations this extra state allows sufficient time for an I/O port to decode its address and activate the $\overline{WAIT}$ line if a wait is required.



## INTERRUPT REQUEST/ACKNOWLEDGE CYCLE

The interrupt signal is sampled by the CPU with the rising edge of the last clock at the end of any instruction. When an interrupt is accepted, a special $M_1$ cycle is generated. During this $M_1$ cycle, the $\overline{IORQ}$ signal becomes active (instead of $\overline{MREQ}$) to indicate that the interrupting device can place an 8-bit vector on the data bus. Two wait states (Tw*) are automatically added to this cycle so that a ripple priority interrupt scheme, such as the one used in the Z80 peripheral controllers, can be easily implemented.



# Z80, Z80A Instruction Set

The following is a summary of the Z80, Z80A instruction set showing the assembly language mnemonic and the symbolic operation performed by the instruction. A more detailed listing appears in the Z80-CPU technical manual, and assembly language programming manual. The instructions are divided into the following categories:

- 8-bit loads
- 16-bit loads
- Exchanges
- Memory Block Moves
- Memory Block Searches
- 8-bit arithmetic and logic
- 16-bit arithmetic
- General purpose Accumulator & Flag Operations
- Miscellaneous Group
- Rotates and Shifts
- Bit Set, Reset and Test
- Input and Output
- Jumps
- Calls
- Restarts
- Returns

In the table the following terminology is used.

b ≡ a bit number in any 8-bit register or memory location
cc ≡ flag condition code
NZ ≡ non zero
Z ≡ zero
NC ≡ non carry
C ≡ carry
PO ≡ Parity odd or no over flow
PE ≡ Parity even or over flow
P ≡ Positive
M ≡ Negative (minus)

d ≡ any 8-bit destination register or memory location
dd ≡ any 16-bit destination register or memory location
e ≡ 8-bit signed 2's complement displacement used in relative jumps and indexed addressing
L ≡ 8 special call locations in page zero. In decimal notation these are 0, 8, 16, 24, 32, 40, 48 and 56
n ≡ any 8-bit binary number
nn ≡ any 16-bit binary number
r ≡ any 8-bit general purpose register (A, B, C, D, E, H, or L)
s ≡ any 8-bit source register or memory location
$s_b$ ≡ a bit in a specific 8-bit register or memory location
ss ≡ any 16-bit source register or memory location
subscript "L" ≡ the low order 8 bits of a 16-bit register
subscript "H" ≡ the high order 8 bits of a 16-bit register
( ) ≡ the contents within the ( ) are to be used as a pointer to a memory location or I/O port number.

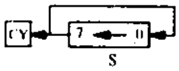
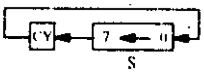
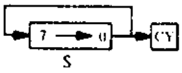
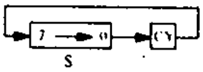
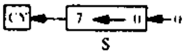
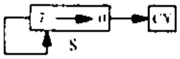
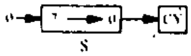
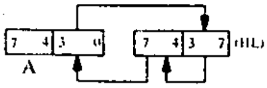
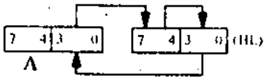
8-bit registers are A, B, C, D, E, H, L, I and R
16-bit register pairs are AF, BC, DE and HL
16-bit registers are SP, PC, IX and IY

Addressing Modes implemented include combinations of the following:

- Immediate
- Immediate extended
- Modified Page Zero
- Relative
- Extended
- Indexed
- Register
- Implied
- Register Indirect
- Bit

| | Mnemonic | Symbolic Operation | Comments |
|---|---|---|---|
| 8-BIT LOADS | LD r, s | r ← s | s ≡ r, n, (HL), (IX+e), (IY+e) |
| | LD d, r | d ← r | d ≡ (HL), r (IX+e), (IY+e) |
| | LD d, n | d ← n | d ≡ (HL), (IX+e), (IY+e) |
| | LD A, s | A ← s | s ≡ (BC), (DE), (nn), I, R |
| | LD d, A | d ← A | d ≡ (BC), (DE), (nn), I, R |
| 16-BIT LOADS | LD dd, nn | dd ← nn | dd ≡ BC, DE, HL, SP, IX, IY |
| | LD dd, (nn) | dd ← (nn) | dd ≡ BC, DE, HL, SP, IX, IY |
| | LD (nn), ss | (nn) ← ss | ss ≡ BC, DE, HL, SP, IX, IY |
| | LD SP, ss | SP ← ss | ss = HL, IX, IY |
| | PUSH ss | (SP-1) ← $ss_H$; (SP-2) ← $ss_L$ | ss = BC, DE, HL, AF, IX, IY |
| | POP dd | $dd_L$ ← (SP); $dd_H$ ← (SP+1) | dd = BC, DE, HL, AF, IX, IY |
| EXCHANGES | EX DE, HL | DE ↔ HL | |
| | EX AF, AF' | AF ↔ AF' | |
| | EXX | $\begin{pmatrix}BC\\DE\\HL\end{pmatrix} \leftrightarrow \begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$ | |
| | EX (SP), ss | (SP) ↔ $ss_L$, (SP+1) ↔ $ss_H$ | ss ≡ HL, IX, IY |

| | Mnemonic | Symbolic Operation | Comments |
|---|---|---|---|
| MEMORY BLOCK MOVES | LDI | (DE) ← (HL), DE ← DE+1 HL ← HL+1, BC ← BC-1 | |
| | LDIR | (DE) ← (HL), DE ← DE+1 HL ← HL+1, BC ← BC-1 Repeat until BC = 0 | |
| | LDD | (DE) ← (HL), DE ← DE-1 HL ← HL-1, BC ← BC-1 | |
| | LDDR | (DE) ← (HL), DE ← DE-1 HL ← HL-1, BC ← BC-1 Repeat until BC = 0 | |
| MEMORY BLOCK SEARCHES | CPI | A-(HL), HL ← HL+1 BC ← BC-1 | |
| | CPIR | A-(HL), HL ← HL+1 BC ← BC-1, Repeat until BC = 0 or A = (HL) | A-(HL) sets the flags only. A is not affected |
| | CPD | A-(HL), HL ← HL-1 BC ← BC-1 | |
| | CPDR | A-(HL), HL ← HL-1 BC ← BC-1, Repeat until BC = 0 or A = (HL) | |
| 8-BIT ALU | ADD s | A ← A + s | |
| | ADC s | A ← A + s + CY | CY is the carry flag |
| | SUB s | A ← A - s | |
| | SBC s | A ← A - s - CY | s ≡ r, n, (HL) (IX+e), (IY+e) |
| | AND s | $A \leftarrow A \wedge s$ | |
| | OR s | $A \leftarrow A \vee s$ | |
| | XOR s | $A \leftarrow A \oplus s$ | |

| | Mnemonic | Symbolic Operation | Comments |
|---|---|---|---|
| 8-BIT ALU | CP s | A - s | s = r, n (HL)<br>(IX+e), (IY+e) |
| | INC d | d ← d + 1 | d = r, (HL)<br>(IX+e), (IY+e) |
| | DEC d | d ← d - 1 | |
| 16-BIT ARITHMETIC | ADD HL, ss | HL ← HL + ss | ss ≡ BC, DE<br>HL, SP |
| | ADC HL, ss | HL ← HL + ss + CY | |
| | SBC HL, ss | HL ← HL - ss - CY | |
| | ADD IX, ss | IX ← IX + ss | ss ≡ BC, DE,<br>IX, SP |
| | ADD IY, ss | IY ← IY + ss | ss ≡ BC, DE,<br>IY, SP |
| | INC dd | dd ← dd + 1 | dd ≡ BC, DE,<br>HL, SP, IX, IY |
| | DEC dd | dd ← dd - 1 | dd ≡ BC, DE,<br>HL, SP, IX, IY |
| GP ACC. & FLAG | DAA | Converts A contents into packed BCD following add or subtract. | Operands must be in packed BCD format |
| | CPL | A ← $\overline{A}$ | |
| | NEG | A ← 00 – A | |
| | CCF | CY ← $\overline{CY}$ | |
| | SCF | CY ← 1 | |
| MISCELLANEOUS | NOP | No operation | |
| | HALT | Halt CPU | |
| | DI | Disable Interrupts | |
| | EI | Enable Interrupts | |
| | IM 0 | Set interrupt mode 0 | 8080A mode |
| | IM 1 | Set interrupt mode 1 | Call to $0038_H$ |
| | IM 2 | Set interrupt mode 2 | Indirect Call |
| ROTATES AND SHIFTS | RLC s |  | |
| | RL s |  | |
| | RRC s |  | |
| | RR s |  | |
| | SLA s |  | s ≡ r, (HL)<br>(IX+e), (IY+e) |
| | SRA s |  | |
| | SRL s |  | |
| | RLD |  | |
| | RRD |  | |

| | Mnemonic | Symbolic Operation | Comments |
|---|---|---|---|
| BIT S, R, & T | BIT b, s | Z ← $\overline{s_b}$ | Z is zero flag |
| | SET b, s | $s_b$ ← 1 | s ≡ r, (HL) |
| | RES b, s | $s_b$ ← 0 | (IX+e), (IY+ |
| INPUT AND OUTPUT | IN A, (n) | A ← (n) | |
| | IN r, (C) | r ← (C) | Set flags |
| | INI | (HL) ← (C), HL ← HL + 1<br>B ← B - 1 | |
| | INIR | (HL) ← (C), HL ← HL + 1<br>B ← B - 1<br>Repeat until B = 0 | |
| | IND | (HL) ← (C), HL ← HL - 1<br>B ← B - 1 | |
| | INDR | (HL) ← (C), HL ← HL - 1<br>B ← B - 1<br>Repeat until B = 0 | |
| | OUT(n), A | (n) ← A | |
| | OUT(C), r | (C) ← r | |
| | OUTI | (C) ← (HL), HL ← HL + 1<br>B ← B - 1 | |
| | OTIR | (C) ← (HL), HL ← HL + 1<br>B ← B - 1<br>Repeat until B = 0 | |
| | OUTD | (C) ← (HL), HL ← HL - 1<br>B ← B - 1 | |
| | OTDR | (C) ← (HL), HL ← HL - 1<br>B ← B - 1<br>Repeat until B = 0 | |
| JUMPS | JP nn | PC ← nn | cc { NZ PO<br>Z PE<br>NC P<br>C M |
| | JP cc, nn | If condition cc is true PC ← nn, else continue | |
| | JR e | PC ← PC + e | |
| | JR kk, e | If condition kk is true PC ← PC + e, else continue | kk { NZ NC<br>Z C |
| | JP (ss) | PC ← ss | ss = HL, IX, IY |
| | DJNZ e | B ← B - 1, if B = 0 continue, else PC ← PC + e | |
| CALLS | CALL nn | (SP-1) ← $PC_H$<br>(SP-2) ← $PC_L$, PC ← nn | cc { NZ PO<br>Z PE<br>NC P<br>C M |
| | CALL cc, nn | If condition cc is false continue, else same as CALL nn | |
| RESTARTS | RST L | (SP-1) ← $PC_H$<br>(SP-2) ← $PC_L$, $PC_H$ ← 0<br>$PC_L$ ← L | |
| RETURNS | RET | $PC_L$ ← (SP),<br>$PC_H$ ← (SP+1) | |
| | RET cc | If condition cc is false continue, else same as RET | cc { NZ PO<br>Z PE<br>NC P<br>C M |
| | RETI | Return from interrupt, same as RET | |
| | RETN | Return from non-maskable interrupt | |

# A.C. Characteristics

# Z80-CPU

$T_A$ = 0°C to 70°C, Vcc = +5V ± 5%, Unless Otherwise Noted.

| Signal | Symbol | Parameter | Min | Max | Unit | Test Condition |
|---|---|---|---|---|---|---|
| Φ | $t_c$ | Clock Period | .4 | [12] | μsec | |
| | $t_{w(\Phi H)}$ | Clock Pulse Width, Clock High | 180 | [E] | nsec | |
| | $t_{w(\Phi L)}$ | Clock Pulse Width, Clock Low | 180 | 2000 | nsec | |
| | $t_{r,f}$ | Clock Rise and Fall Time | | 30 | nsec | |
| $A_{0-15}$ | $t_{D(AD)}$ | Address Output Delay | | 145 | nsec | $C_L$ = 50pF |
| | $t_{F(AD)}$ | Delay to Float | | 110 | nsec | |
| | $t_{acm}$ | Address Stable Prior to $\overline{MREQ}$ (Memory Cycle) | [1] | | nsec | |
| | $t_{aci}$ | Address Stable Prior to $\overline{IORQ}$, $\overline{RD}$ or $\overline{WR}$ (I/O Cycle) | [2] | | nsec | |
| | $t_{ca}$ | Address Stable from $\overline{RD}$, $\overline{WR}$, $\overline{IORQ}$ or $\overline{MREQ}$ | [3] | | nsec | |
| | $t_{caf}$ | Address Stable From $\overline{RD}$ or $\overline{WR}$ During Float | [4] | | nsec | |
| $D_{0-7}$ | $t_{D(D)}$ | Data Output Delay | | 230 | nsec | $C_L$ = 50pF |
| | $t_{F(D)}$ | Delay to Float During Write Cycle | | 90 | nsec | |
| | $t_{S\Phi(D)}$ | Data Setup Time to Rising Edge of Clock During M1 Cycle | 50 | | nsec | |
| | $t_{S\overline{\Phi}(D)}$ | Data Setup Time to Falling Edge of Clock During M2 to M5 | 60 | | nsec | |
| | $t_{dcm}$ | Data Stable Prior to $\overline{WR}$ (Memory Cycle) | [5] | | nsec | |
| | $t_{dci}$ | Data Stable Prior to $\overline{WR}$ (I/O Cycle) | [6] | | nsec | |
| | $t_{cdf}$ | Data Stable From $\overline{WR}$ | [7] | | | |
| | $t_H$ | Any Hold Time for Setup Time | 0 | | nsec | |
| $\overline{MREQ}$ | $t_{DL\overline{\Phi}(MR)}$ | $\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ Low | | 100 | nsec | $C_L$ = 50pF |
| | $t_{DH\Phi(MR)}$ | $\overline{MREQ}$ Delay From Rising Edge of Clock, $\overline{MREQ}$ High | | 100 | nsec | |
| | $t_{DH\overline{\Phi}(MR)}$ | $\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ High | | 100 | nsec | |
| | $t_{w(\overline{MRL})}$ | Pulse Width, $\overline{MREQ}$ Low | [8] | | nsec | |
| | $t_{w(\overline{MRH})}$ | Pulse Width, $\overline{MREQ}$ High | [9] | | nsec | |
| $\overline{IORQ}$ | $t_{DL\Phi(IR)}$ | $\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ Low | | 90 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(IR)}$ | $\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ Low | | 110 | nsec | |
| | $t_{DH\Phi(IR)}$ | $\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ High | | 100 | nsec | |
| | $t_{DH\overline{\Phi}(IR)}$ | $\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ High | | 110 | nsec | |
| $\overline{RD}$ | $t_{DL\Phi(RD)}$ | $\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ Low | | 100 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(RD)}$ | $\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ Low | | 130 | nsec | |
| | $t_{DH\Phi(RD)}$ | $\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ High | | 100 | nsec | |
| | $t_{DH\overline{\Phi}(RD)}$ | $\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ High | | 110 | nsec | |
| $\overline{WR}$ | $t_{DL\Phi(WR)}$ | $\overline{WR}$ Delay From Rising Edge of Clock, $\overline{WR}$ Low | | 80 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(WR)}$ | $\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ Low | | 90 | nsec | |
| | $t_{DH\overline{\Phi}(WR)}$ | $\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ High | | 100 | nsec | |
| | $t_{w(\overline{WRL})}$ | Pulse Width, $\overline{WR}$ Low | [10] | | nsec | |
| $\overline{M1}$ | $t_{DL(M1)}$ | $\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ Low | | 130 | nsec | $C_L$ = 50pF |
| | $t_{DH(M1)}$ | $\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ High | | 130 | nsec | |
| $\overline{RFSH}$ | $t_{DL(RF)}$ | $\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ Low | | 180 | nsec | $C_L$ = 50pF |
| | $t_{DH(RF)}$ | $\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ High | | 150 | nsec | |
| $\overline{WAIT}$ | $t_{s(WT)}$ | $\overline{WAIT}$ Setup Time to Falling Edge of Clock | 70 | | nsec | |
| $\overline{HALT}$ | $t_{D(HT)}$ | $\overline{HALT}$ Delay Time From Falling Edge of Clock | | 300 | nsec | $C_L$ = 50pF |
| $\overline{INT}$ | $t_{s(IT)}$ | $\overline{INT}$ Setup Time to Rising Edge of Clock | 80 | | nsec | |
| $\overline{NMI}$ | $t_{w(\overline{NML})}$ | Pulse Width, $\overline{NMI}$ Low | 80 | | nsec | |
| $\overline{BUSRQ}$ | $t_{s(BQ)}$ | $\overline{BUSRQ}$ Setup Time to Rising Edge of Clock | 80 | | nsec | |
| $\overline{BUSAK}$ | $t_{DL(BA)}$ | $\overline{BUSAK}$ Delay From Rising Edge of Clock, $\overline{BUSAK}$ Low | | 120 | nsec | $C_L$ = 50pF |
| | $t_{DH(BA)}$ | $\overline{BUSAK}$ Delay From Falling Edge of Clock, $\overline{BUSAK}$ High | | 110 | nsec | |
| $\overline{RESET}$ | $t_{s(RS)}$ | $\overline{RESET}$ Setup Time to Rising Edge of Clock | 90 | | nsec | |
| | $t_{F(C)}$ | Delay to Float ($\overline{MREQ}$, $\overline{IORQ}$, $\overline{RD}$ and $\overline{WR}$) | | 100 | nsec | |
| | $t_{mr}$ | $\overline{M1}$ Stable Prior to $\overline{IORQ}$ (Interrupt Ack.) | [11] | | nsec | |

[12] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$

[1] $t_{acm} = t_{w(\Phi H)} + t_f - 75$

[2] $t_{aci} = t_c - 80$

[3] $t_{ca} = t_{w(\Phi L)} + t_r - 40$

[4] $t_{caf} = t_{w(\Phi L)} + t_r - 60$

[5] $t_{dcm} = t_c - 210$

[6] $t_{dci} = t_{w(\Phi L)} + t_r - 210$

[7] $t_{cdf} = t_{w(\Phi L)} + t_r - 80$

[8] $t_{w(\overline{MRL})} = t_c - 40$

[9] $t_{w(\overline{MRH})} = t_{w(\Phi H)} + t_f - 30$

[10] $t_{w(\overline{WRL})} = t_c - 40$

[11] $t_{mr} = 2t_c + t_{w(\Phi H)} + t_f - 80$

NOTES

A. Data should be enabled onto the CPU data bus when $\overline{RD}$ is active. During interrupt acknowledge data should be enabled when $\overline{M1}$ and $\overline{IORQ}$ are both active.

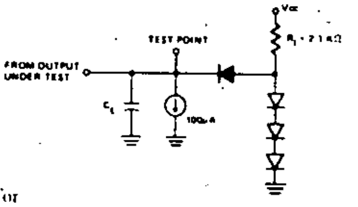
B. All control signals are internally synchronized, so they may be totally asynchronous with respect to the clock.

C. The $\overline{RESET}$ signal must be active for a minimum of 3 clock cycles.

D. Output Delay vs. Loaded Capacitance
TA = 70°C  Vcc = +5V ±5%

Add 10nsec delay for each 50pf increase in load up to a maximum of 200pf for the data bus & 100pf for address & control lines

E. Although static by design, testing guarantees $t_{w(\Phi H)}$ of 200 μsec maximum

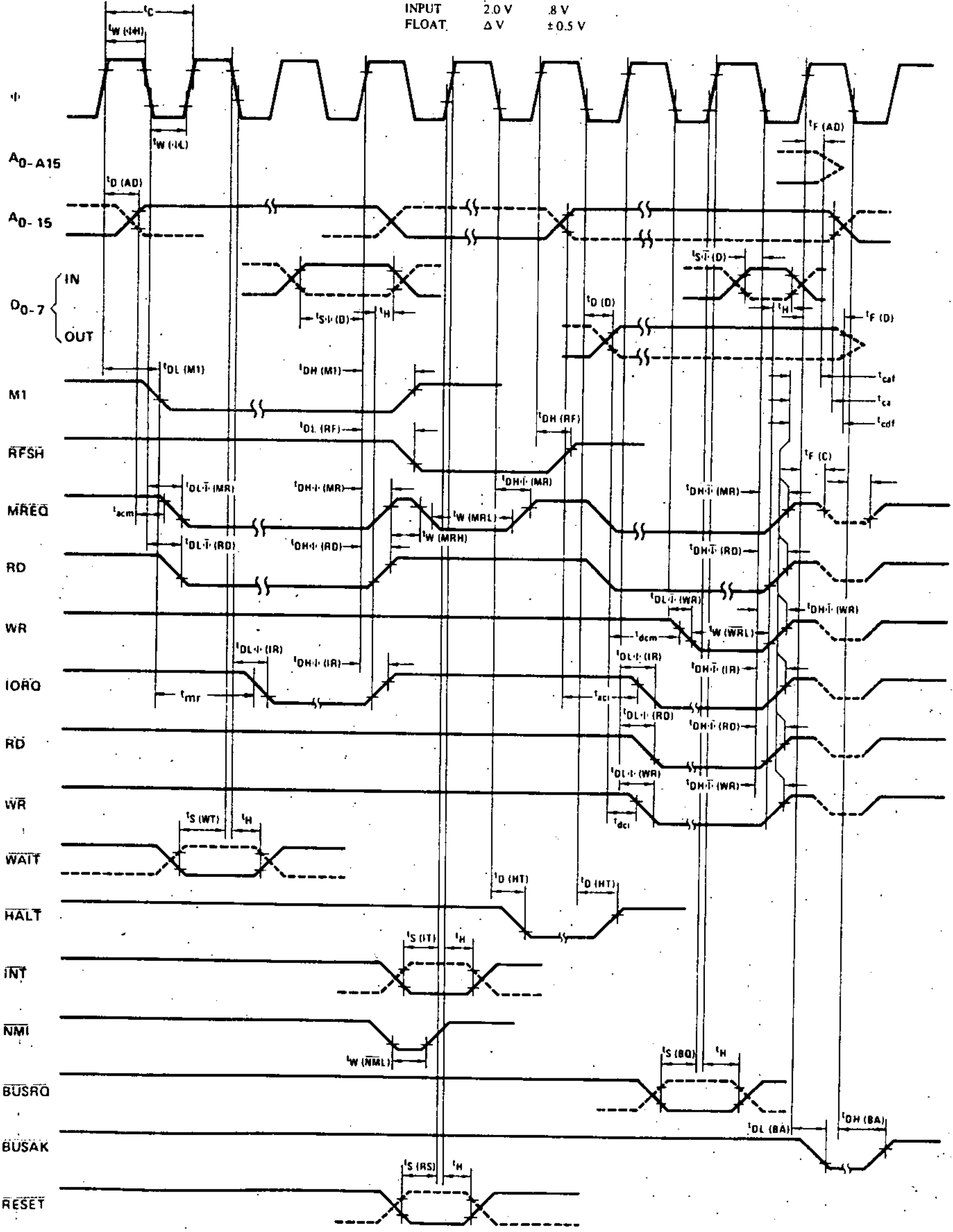


Load circuit for Output

# A.C. Timing Diagram

Timing measurements are made at the following voltages, unless otherwise specified:

| | "1" | "0" |
|---|---|---|
| CLOCK | $V_{cc}$ -.6V | .45V |
| OUTPUT | 2.0 V | .8 V |
| INPUT | 2.0 V | .8 V |
| FLOAT | Δ V | ± 0.5 V |



# Absolute Maximum Ratings

| | |
|---|---|
| Temperature Under Bias | Specified operating range. |
| Storage Temperature | -65°C to +150°C |
| Voltage On Any Pin with Respect to Ground | -0.3V to +7V |
| Power Dissipation | 1.5W |

*Comment

Stresses above those listed under "Absolute Maximum Rating" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other condition above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Note: For Z80-CPU all AC and DC characteristics remain the same for the military grade parts except $I_{cc}$.

$I_{cc}$ = 200 mA

## Z80-CPU D.C. Characteristics

$T_A$ = 0°C to 70°C, $V_{cc}$ = 5V ± 5% unless otherwise specified

| Symbol | Parameter | Min. | Typ. | Max. | Unit | Test Condition |
|---|---|---|---|---|---|---|
| $V_{ILC}$ | Clock Input Low Voltage | -0.3 | | 0.45 | V | |
| $V_{IHC}$ | Clock Input High Voltage | $V_{cc}$ -.6 | | $V_{cc}$+.3 | V | |
| $V_{IL}$ | Input Low Voltage | -0.3 | | 0.8 | V | |
| $V_{IH}$ | Input High Voltage | 2.0 | | $V_{cc}$ | V | |
| $V_{OL}$ | Output Low Voltage | | | 0.4 | V | $I_{OL}$=1.8mA |
| $V_{OH}$ | Output High Voltage | 2.4 | | | V | $I_{OH}$ = -250µA |
| $I_{CC}$ | Power Supply Current | | | 150 | mA | |
| $I_{LI}$ | Input Leakage Current | | | 10 | µA | $V_{IN}$=0 to $V_{cc}$ |
| $I_{LOH}$ | Tri-State Output Leakage Current in Float | | | 10 | µA | $V_{OUT}$=2.4 to $V_{cc}$ |
| $I_{LOL}$ | Tri-State Output Leakage Current in Float | | | -10 | µA | $V_{OUT}$=0.4V |
| $I_{LD}$ | Data Bus Leakage Current in Input Mode | | | ±10 | µA | 0 ≤ $V_{IN}$ ≤ $V_{cc}$ |

## Capacitance

$T_A$ = 25°C, f = 1 MHz,
unmeasured pins returned to ground

| Symbol | Parameter | Max. | Unit |
|---|---|---|---|
| $C_\Phi$ | Clock Capacitance | 35 | pF |
| $C_{IN}$ | Input Capacitance | 5 | pF |
| $C_{OUT}$ | Output Capacitance | 10 | pF |

## Z80-CPU Ordering Information

C – Ceramic
P – Plastic
S – Standard 5V ±5% 0° to 70°C
E – Extended 5V ±5% -40° to 85°C
M – Military 5V ±10% -55° to 125°C

## Z80A-CPU D.C. Characteristics

$T_A$ = 0°C to 70°C, $V_{cc}$ = 5V ± 5% unless otherwise specified

| Symbol | Parameter | Min. | Typ. | Max. | Unit | Test Condition |
|---|---|---|---|---|---|---|
| $V_{ILC}$ | Clock Input Low Voltage | -0.3 | | 0.45 | V | |
| $V_{IHC}$ | Clock Input High Voltage | $V_{cc}$ -.6 | | $V_{cc}$+.3 | V | |
| $V_{IL}$ | Input Low Voltage | -0.3 | | 0.8 | V | |
| $V_{IH}$ | Input High Voltage | 2.0 | | $V_{cc}$ | V | |
| $V_{OL}$ | Output Low Voltage | | | 0.4 | V | $I_{OL}$=1.8mA |
| $V_{OH}$ | Output High Voltage | 2.4 | | | V | $I_{OH}$ = -250µA |
| $I_{CC}$ | Power Supply Current | | 90 | 200 | mA | |
| $I_{LI}$ | Input Leakage Current | | | 10 | µA | $V_{IN}$=0 to $V_{cc}$ |
| $I_{LOH}$ | Tri-State Output Leakage Current in Float | | | 10 | µA | $V_{OUT}$=2.4 to $V_{cc}$ |
| $I_{LOL}$ | Tri-State Output Leakage Current in Float | | | -10 | µA | $V_{OUT}$=0.4V |
| $I_{LD}$ | Data Bus Leakage Current in Input Mode | | | ±10 | µA | 0 ≤ $V_{IN}$ ≤ $V_{cc}$ |

## Capacitance

$T_A$ = 25°C, f = 1 MHz,
unmeasured pins returned to ground

| Symbol | Parameter | Max. | Unit |
|---|---|---|---|
| $C_\Phi$ | Clock Capacitance | 35 | pF |
| $C_{IN}$ | Input Capacitance | 5 | pF |
| $C_{OUT}$ | Output Capacitance | 10 | pF |

## Z80A-CPU Ordering Information

C – Ceramic
P – Plastic
S – Standard 5V ±5% 0° to 70°C

$T_A$ = 0°C to 70°C, Vcc = +5V ± 5%, Unless Otherwise Noted.

| Signal | Symbol | Parameter | Min | Max | Unit | Test Condition |
|---|---|---|---|---|---|---|
| Φ | $t_c$ | Clock Period | .25 | [12] | μsec | |
| | $t_{w(\Phi H)}$ | Clock Pulse Width, Clock High | 110 | [E] | nsec | |
| | $t_{w(\Phi L)}$ | Clock Pulse Width, Clock Low | 110 | 2000 | nsec | |
| | $t_{r,f}$ | Clock Rise and Fall Time | | 30 | nsec | |
| $A_{0-15}$ | $t_{D(AD)}$ | Address Output Delay | | 110 | nsec | $C_L$ = 50pF |
| | $t_{F(AD)}$ | Delay to Float | | 90 | nsec | |
| | $t_{acm}$ | Address Stable Prior to $\overline{\text{MREQ}}$ (Memory Cycle) | [1] | | nsec | |
| | $t_{aci}$ | Address Stable Prior to $\overline{\text{IORQ}}$, $\overline{\text{RD}}$ or $\overline{\text{WR}}$ (I/O Cycle) | [2] | | nsec | |
| | $t_{ca}$ | Address Stable from $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{IORQ}}$ or $\overline{\text{MREQ}}$ | [3] | | nsec. | |
| | $t_{caf}$ | Address Stable From $\overline{\text{RD}}$ or $\overline{\text{WR}}$ During Float | [4] | | nsec | |
| $D_{0-7}$ | $t_{D(D)}$ | Data Output Delay | | 150 | nsec | $C_L$ = 50pF |
| | $t_{F(D)}$ | Delay to Float During Write Cycle | | 90 | nsec | |
| | $t_{S\Phi(D)}$ | Data Setup Time to Rising Edge of Clock During M1 Cycle | 35 | | nsec | |
| | $t_{S\overline{\Phi}(D)}$ | Data Setup Time to Falling Edge of Clock During M2 to M5 | 50 | | nsec | |
| | $t_{dcm}$ | Data Stable Prior to $\overline{\text{WR}}$ (Memory Cycle) | [5] | | nsec | |
| | $t_{dci}$ | Data Stable Prior to $\overline{\text{WR}}$ (I/O Cycle) | [6] | | nsec | |
| | $t_{cdf}$ | Data Stable From $\overline{\text{WR}}$ | [7] | | | |
| | $t_H$ | Any Hold Time for Setup Time | | 0 | nsec | |
| $\overline{\text{MREQ}}$ | $t_{DL\overline{\Phi}(MR)}$ | $\overline{\text{MREQ}}$ Delay From Falling Edge of Clock, $\overline{\text{MREQ}}$ Low | | 85 | nsec | $C_L$ = 50pF |
| | $t_{DH\Phi(MR)}$ | $\overline{\text{MREQ}}$ Delay From Rising Edge of Clock, $\overline{\text{MREQ}}$ High | | 85 | nsec | |
| | $t_{DH\overline{\Phi}(MR)}$ | $\overline{\text{MREQ}}$ Delay From Falling Edge of Clock, $\overline{\text{MREQ}}$ High | | 85 | nsec | |
| | $t_{w(\overline{MRL})}$ | Pulse Width, $\overline{\text{MREQ}}$ Low | [8] | | nsec | |
| | $t_{w(\overline{MRH})}$ | Pulse Width, $\overline{\text{MREQ}}$ High | [9] | | nsec | |
| $\overline{\text{IORQ}}$ | $t_{DL\Phi(IR)}$ | $\overline{\text{IORQ}}$ Delay From Rising Edge of Clock, $\overline{\text{IORQ}}$ Low | | 75 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(IR)}$ | $\overline{\text{IORQ}}$ Delay From Falling Edge of Clock, $\overline{\text{IORQ}}$ Low | | 85 | nsec | |
| | $t_{DH\Phi(IR)}$ | $\overline{\text{IORQ}}$ Delay From Rising Edge of Clock, $\overline{\text{IORQ}}$ High | | 85 | nsec | |
| | $t_{DH\overline{\Phi}(IR)}$ | $\overline{\text{IORQ}}$ Delay From Falling Edge of Clock, $\overline{\text{IORQ}}$ High | | 85 | nsec | |
| $\overline{\text{RD}}$ | $t_{DL\Phi(RD)}$ | $\overline{\text{RD}}$ Delay From Rising Edge of Clock, $\overline{\text{RD}}$ Low | | 85 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(RD)}$ | $\overline{\text{RD}}$ Delay From Falling Edge of Clock, $\overline{\text{RD}}$ Low | | 95 | nsec | |
| | $t_{DH\Phi(RD)}$ | $\overline{\text{RD}}$ Delay From Rising Edge of Clock, $\overline{\text{RD}}$ High | | 85 | nsec | |
| | $t_{DH\overline{\Phi}(RD)}$ | $\overline{\text{RD}}$ Delay From Falling Edge of Clock, $\overline{\text{RD}}$ High | | 85 | nsec | |
| $\overline{\text{WR}}$ | $t_{DL\Phi(WR)}$ | $\overline{\text{WR}}$ Delay From Rising Edge of Clock, $\overline{\text{WR}}$ Low | | 65 | nsec | $C_L$ = 50pF |
| | $t_{DL\overline{\Phi}(WR)}$ | $\overline{\text{WR}}$ Delay From Falling Edge of Clock, $\overline{\text{WR}}$ Low | | 80 | nsec | |
| | $t_{DH\overline{\Phi}(WR)}$ | $\overline{\text{WR}}$ Delay From Falling Edge of Clock, $\overline{\text{WR}}$ High | | 80 | nsec | |
| | $t_{w(\overline{WRL})}$ | Pulse Width, $\overline{\text{WR}}$ Low | [10] | | nsec | |
| $\overline{\text{M1}}$ | $t_{DL(M1)}$ | $\overline{\text{M1}}$ Delay From Rising Edge of Clock, $\overline{\text{M1}}$ Low | | 100 | nsec | $C_L$ = 50pF |
| | $t_{DH(M1)}$ | $\overline{\text{M1}}$ Delay From Rising Edge of Clock, $\overline{\text{M1}}$ High | | 100 | nsec | |
| $\overline{\text{RFSH}}$ | $t_{DL(RF)}$ | $\overline{\text{RFSH}}$ Delay From Rising Edge of Clock, $\overline{\text{RFSH}}$ Low | | 130 | nsec | $C_L$ = 50pF |
| | $t_{DH(RF)}$ | $\overline{\text{RFSH}}$ Delay From Rising Edge of Clock, $\overline{\text{RFSH}}$ High | | 120 | nsec | |
| $\overline{\text{WAIT}}$ | $t_{s(WT)}$ | $\overline{\text{WAIT}}$ Setup Time to Falling Edge of Clock | 70 | | nsec | |
| $\overline{\text{HALT}}$ | $t_{D(HT)}$ | $\overline{\text{HALT}}$ Delay Time From Falling Edge of Clock | | 300 | nsec | $C_L$ = 50pF |
| $\overline{\text{INT}}$ | $t_{s(IT)}$ | $\overline{\text{INT}}$ Setup Time to Rising Edge of Clock | 80 | | nsec | |
| $\overline{\text{NMI}}$ | $t_{w(\overline{NML})}$ | Pulse Width, $\overline{\text{NMI}}$ Low | 80 | | nsec | |
| $\overline{\text{BUSRQ}}$ | $t_{s(BQ)}$ | $\overline{\text{BUSRQ}}$ Setup Time to Rising Edge of Clock | 50 | | nsec | |
| $\overline{\text{BUSAK}}$ | $t_{DL(BA)}$ | $\overline{\text{BUSAK}}$ Delay From Rising Edge of Clock, $\overline{\text{BUSAK}}$ Low | | 100 | nsec | $C_L$ = 50pF |
| | $t_{DH(BA)}$ | $\overline{\text{BUSAK}}$ Delay From Falling Edge of Clock, $\overline{\text{BUSAK}}$ High | | 100 | nsec | |
| $\overline{\text{RESET}}$ | $t_{s(RS)}$ | $\overline{\text{RESET}}$ Setup Time to Rising Edge of Clock | 60 | | nsec | |
| | $t_{F(C)}$ | Delay to Float ($\overline{\text{MREQ}}$, $\overline{\text{IORQ}}$, $\overline{\text{RD}}$ and $\overline{\text{WR}}$) | | 80 | nsec | |
| | $t_{mr}$ | M1 Stable Prior to IORQ (Interrupt Ack.) | [11] | | nsec | |

[12] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$

[1] $t_{acm} = t_{w(\Phi H)} + t_f - 65$

[2] $t_{aci} = t_c - 70$

[3] $t_{ca} = t_{w(\Phi L)} + t_r - 50$

[4] $t_{caf} = t_{w(\Phi L)} + t_r - 45$

[5] $t_{dcm} = t_c - 170$

[6] $t_{dci} = t_{w(\Phi L)} + t_r - 170$

[7] $t_{cdf} = t_{w(\Phi L)} + t_r - 70$

[8] $t_{w(\overline{MRL})} = t_c - 30$

[9] $t_{w(\overline{MRH})} = t_{w(\Phi H)} + t_f - 20$

[10] $t_{w(\overline{WRL})} = t_c - 30$

[11] $t_{mr} = 2t_c + t_{w(\Phi H)} + t_f - 65$

NOTES:

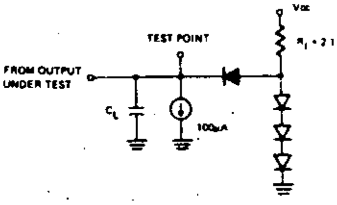
A. Data should be enabled onto the CPU data bus when $\overline{\text{RD}}$ is active. During interrupt acknowledge data should be enabled when $\overline{\text{M1}}$ and $\overline{\text{IORQ}}$ are both active.

B. All control signals are internally synchronized, so they may be totally asynchronous with respect to the clock.

C. The $\overline{\text{RESET}}$ signal must be active for a minimum of 3 clock cycles.

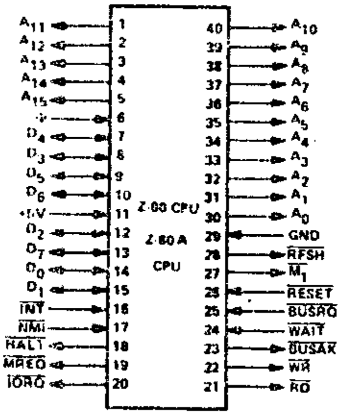
D. Output Delay vs. Loaded Capacitance
TA = 70°C Vcc = +5V ±5%
Add 10nsec delay for each 50pf increase in load up to maximum of 200pf for data bus and 100pf for address & control lines.

E. Although static by design, testing guarantees $t_{w(\Phi H)}$ of 200 μsec maximum

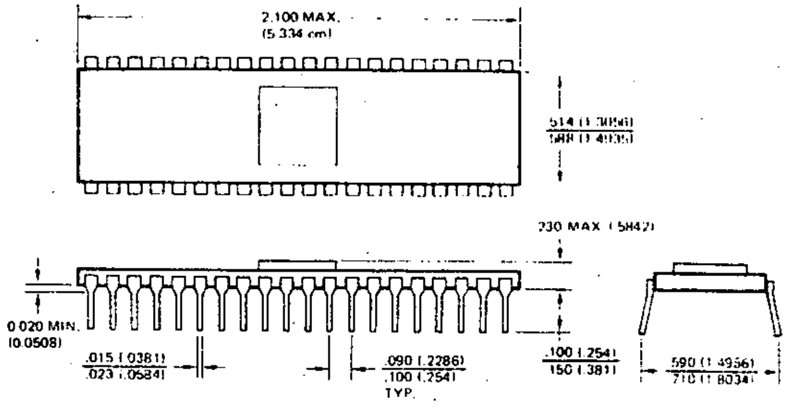


Load circuit for Output

## Package Configuration



## Package Outline



*Dimensions for metric system are in parentheses

## ZILOG U.S. DISTRIBUTORS

**TERN**

ark Electronics
Commercial Drive
tsville, AL 35805
L 205 837 8700
X 810 726 2187

allmark Electronics
2 West McNab Road
rt Lauderdale, FL 33309
L 305 971 9280
WX 510 956 9720

Hallmark Electronics
7233 Lake Ellenor Drive
Orlando, FL 32809
TEL 305 855 4020
TWX 810 850 0183

Hallmark Electronics
3255 Amberton Drive
Baltimore MD 21227
TEL 301 796 9300
TWX 710 862 1942

Hallmark Electronics
1208 Front Street
uilding K
aleigh, NC 27609
919 832 4465
X 510 928 1831

Hallmark Electronics
ike Industrial Park
Huntington Valley, PA 19006
TEL 215 355 7300
TWX 510 667 1750

Summit
916 Main Street
Buffalo, NY 14202
TEL 716 884 3450

Wilshire Electronics
2554 State Street
Hamden, CT 06517
TEL 203 281 1166
TWX 800 922 1734

Wilshire Electronics
1855 New Highway
Farmingdale, LI, NY 11735
TEL 516 293 5775
TWX 212 895 8707

Wilshire Electronics
One Wilshire Road
Burlington, MA 01803
TEL 617 272 8200
TWX 710 332 6359

Wilshire Electronics
1111 Paulison Avenue
Clifton, NJ 07015
TEL 201 340 1900
TWX 710 989 7052

**MIDWESTERN**

Hallmark Electronics
180 Grossen Avenue
Elk Grove Village, IL 60076
TEL 312 437 8800
TWX 910 223 3645

Hallmark Electronics
11870 West 91st Street
Congleton Industrial Park
Shawnee Mission, KS 66214
TEL 913 888 4747
TWX 910 749 6620

Hallmark Electronics
9201 Penn Avenue South
Suite 10
Bloomington, MN 55435
TEL 612 884 9056
TWX 910 576 3187

Hallmark Electronics
13789 Rider Trail
Earth City, MO 63045
TEL 314 291 5350
TWX 910 760 0671

Hallmark Electronics
6969 Worthington Galena Road
Worthington, OH 43085
TEL 614 846 1882

Hallmark Electronics
4846 S. 83rd E. Avenue
Tulsa, OK 74145
TEL 918 835 8458
TWX 910 845 2290

Hallmark Electronics
3100–A Industrial Terrace
Austin, TX 78758
TEL 512 837 2841
TWX 910 874 2031

Hallmark Electronics
9333 Forest Lane
Dallas, TX 75231
TEL 214 234 7300
TWX 910 867 4721

Hallmark Electronics
8000 Westglen
Houston, TX 77063
TEL 713 781 6100
TWX 910 881 2711

Hallmark Electronics
237 South Curtis
West Allis, WI 53214
TEL 414 476 1270
TWX 910 262 3186

RM Electronics
4860 South Division
Kentwood, MI 49508
TEL 616 531 9300
TWX 810 273 8779

RM Electronics
47 Chestnut Lane
Westmont, Illinois 60559
TEL 312 323 9670

**MOUNTAIN**

Century Electronics
121 Elizabeth, NE
Albuquerque, NM 87123
TEL 505 292 2700
TWX 910 989 0625

**WESTERN**

Intermark Electronics
1802 E. Carnegie Avenue
Santa Ana, CA 92705
TEL 714 540 1322
TWX 910 595 1583

Intermark Electronics
4040 Sorrento Valley Blvd.
San Diego, CA 92121
TEL 714 279 5200
714 453 9005
TWX 910 335 1515

Intermark Electronics
1020 Stewart Drive
Sunnyvale, CA 94086
TEL 408 738 1111
TWX 910 339 9312

R.V. Weatherford Co.
6921 San Fernando Road
Glendale, CA 91201
TEL 213 849 3451
TWX 910 498 2223

R.V. Weatherford Co.
1550 Babbitt Avenue
Anaheim, CA 92805
TEL 714 634 9600
TWX 910 593 1334

R.V. Weatherford Co.
1095 East Third Street
Pomona, CA 91765
TEL 714 623 1261
TWX 910 581 3811

R.V. Weatherford Co.
3240 Hillview Avenue
Stanford Industrial Park
Palo Alto, CA 94304
TEL 415 493 5373

R.V. Weatherford Co.
3311 W. Earll Drive
Phoenix, AZ 85017
TEL 602 272 7144
TWX 910 951 0636

Sterling Electronics
5608 6th Avenue South
Seattle, WA 98108
TEL 206 762 9100
TLX 32-9652

Western Microtechnology
977 Benicia Avenue
Sunnyvale, CA 94086
TEL 408 737 1660

**CANADA**

Future Electronics
5647 Ferrier Street
Montreal, Quebec,
CANADA H4P 2K5
TEL 514 735 5775
TWX 610 421 3251

## ZILOG SALES OFFICES

**EASTERN REGION**

Zilog, Inc.
76 Treble Cove Road
No. Billerica, MA 01862
TEL 617 667 2179
TWX 710 347 6660

**MIDATLANTIC REGION**

Zilog, Inc.
P.O. Box 92
Bergenfield, NJ 07625
TEL 201 385 9158
TWX 710 991 9771

**MIDWESTERN REGION**

Zilog, Inc.
1701 Woodfield Place
Suite 417
Schaumburg, IL 60195
TEL 312 885 8080
TWX 910 291 1064

**SOUTHWESTERN REGION**

Zilog, Inc.
17982 Sky Park Circle
Suite C
Irvine, CA 92714
TEL 714 549 2891
TWX 910 595 2803

**EUROPEAN HDQTS**

Zilog (UK) Ltd.
Nicholson House
Maidenhead, Berkshire
England
TEL (0628) 36131/2/3
TWX 848 609

10460 Bubb Road, Cupertino, California 95014 Zilog Telephone: (408) 446-4666 TWX: 910-388-7621

Printed in U.S.A.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# RESUMEN DEL DESARROLLO DE MICROPROCESADORES

M. EN C. ANGEL KURI MORALES

MARZO, 1979

¿Qué es un Microprocesador?

Responder a esta pregunta hubiese sido más fácil hace 2 años que hoy día, ¿Por qué? porque la industria y la tecnología microeléctricas son, quizá, las más cambiantes del mundo, considerando cualquier rama del saber humano. Esto se debe, sin duda, a la amplísima gama de aplicaciones de sus derivados y sub-productos. Es un hecho aceptado, hoy día, que el advenimiento del microprocesador es un acontecimiento de importancia similar al de la utilización de la energía eléctrica, a principios de siglo. Esta afirmación podrá parecer exagerada a algunas personas. Examinemos, sin embargo, el desarrollo histórico de los microprocesadores, como se muestra en la siguiente tabla:

| Año | Procesador | Tecnología | Reloj (MHz) | Transistores | Direccionamiento |
|---|---|---|---|---|---|
| 1972 | 8008 | PMOS | .5 | 2000 | 16 K. |
| 1974 | 8080 | PMOS | 2 | 4500 | 64 K. |
| 1976 | Z80 | NMOS | 4 | 6000 | 64 K. |
| 1978(En.) | 8086 | NMOS (VLSI) | 5-8 | 20000 | 1 M. |
| 1978(Dic) | Z8000 | NMOS (VLSI) | ~4 | 17500 | 8 M. |
| 1979 | 68000 | NMOS (VLSI) | 8 | 68000 | 16 M. |

En esta tabla observamos sólo los más representativos de los microprocesadores: de INTEL, el 8008, 8080 y 8086; de Zilog el Z80 y Z8000; de MOTOROLA, el M68000.

Observamos el dramático incremento en velocidad y grado de integración. De 1972, año de la aparición del 8008, a la fecha, hemos pasado de un ciclo de reloj de 1/2 MHz a 8 MHz, para un aumentot de 16 veces en velocidad. Asimismo de 2000 transistores, hemos pasado a 68000, para un aumento de 34(!!) veces en la densidad básica de la unidad. Con los procesadores hoy, podemos direccionar hasta 16 M bytes, vs. sólo 16 K de un 8008; un incremento de 1000 veces!

Pero no sólo en el procesador central existen estas capacidades. Las memorias han sufrido cambios proporcionales. En 1976, la memoria más densa en el mercado era de 4K bits, hoy día, con CCD's y memoria de burbuja magnética, tenemos capacidad de hasta 256 K bits en un 'chip' Un incremento de 64 veces en la capacidad del almacenamiento.

De la misma manera, los controladores, periféricos, etc., se han modificado. El resultado final es que, hoy en día, con cuatro 'chips', tenemos capacidad de cómputo más o menos equivalente a la de una CDC 6600. Como marco de referencia mencionamos a ustedes que en 1970, el documento de CDC que describe a esta máquina empieza con un capítulo que, traducido literalmente, es "Justificación de las Grandes Computadoras".

La conclusión de esta introducción un poco atropellada es la siguiente:

1) Hoy día es posible adquirir capacidad de cómputo a bajo precio en magnitudes que hasta hace una década eran difíciles de justificar por su complejidad.

2) Los costos de cómputo se han reducido dramáticamente. - La tendencia es a seguir disminuyendo. Con la integración que aumenta, los costos se abaten.

3) El 'software' es cada vez más sofisticado. Los micro— computadores de hoy aceptan los lenguajes de ayer (COBOL, FORTRAN, BASIC, PL/I, PASCAL, etc.).

4) Los costos bajos y la alta integración propician nuevas arquitecturas en máquinas 'grandes'. Las computadoras del mañana serán arreglos de decenas o cientos de microprocesadores.

Pero, apesar de nuestras 'conclusiones', no hemos aún respondido a la pregunta inicial. ¿Qué es un microprocesador?

De la anterior discusión se desprenden varias características de los microprocesadores:

1) Son pequeños (físicamente).
2) Son baratos (comparativamente).
3) Tienen capacidad de cómputo.
4) Poseen memoria.

¿En resumen, pues, podemos decir que un microprocesador es un computador pequeño y barato? Si y no. En la introducción mencionamos que es hoy más difícil responder a la pregunta que hace algunos años. Esto se debe a que hay, hoy, varios tipos de microprocesadores. ¿Por qué? porque a cambio de reducir el tamaño (y disminuir el consumo de energía) perdemos, también, velocidad. A las 4 características de la lista anterior, hay que agregar una más:

5) Son "lentos".

¿Qué entendemos por "lentos"? Un micro (procesador) típico actual, tiene un reloj de 2-5 MHz, es decir su ciclo básico de de instrucción es de ~250 ns. En contraste, las máquinas rápidas pueden trabajar a velocidades de ~5 ns. Es decir, 50 veces más rápido.

Es claro que no en todas los casos (de hecho en muy pocos) necesitamos velocidades de ese orden. Entonces los microsistemas se han subdividido en varios grupos, dependiendo del problema a que están orientados. Básicamente, pues, podemos dividir a los microprocesadores en familias:

a) Microprocesadores rápidos, "bit slice" (MSI)
b) Microprocesadores orientados a bytes (L,SI)
c) Microcomputadoras. (VLSI)
d) Micro-miniprocesadores (VLSI)

Para que nuestra definición quede completa, pues, hay que decir que un microprocesador cae dentro de alguna de las anteriores familias.

---

La definición de un microprocesador es pues, la siguiente:

Un microprocesador es un dispositivo electrónico digital de alta integración, que incorpora todas las características básicas de una computadora convencional; que es de bajo costo, bajo consumo de potencia y que pertenece a alguna de las 4 categorías mencionadas anteriormente.

Para que nuestra definición quede clara, hay que especificar qué entendemos por computador 'convencional'.

Este es un sistema digital que:

1) Tiene medios de entrada.
2) Tiene un almacen, en donde pueden estar instrucciones o datos.
3) Tiene una sección capaz de ejecutar cálculos aritméticos y lógicos.
4) Tiene medios de salida.
5) Tiene una unidad de control, capaz de escoger de entre distintos cursos de acción, dependiendo de los datos.

En este curso nos restringiremos a estudiar microprocesadores del tipo (b) debido a que:

1) Son los más usados.
2) Son los más desarrollados.
3) Son los más útiles en el contexto de nuestro país.
4) Son los más baratos.

Con el objeto de introducir al estudioso al campo de los microprocesadores, sin embargo, mendionaremos brevemente, a un representante de las categorías a, c y d.

Microprocesadores '<u>Bit Slice</u>'.

Este tipo de micros tienen la característica de ser "rebanadas" ("slices") de procesado. Esto significa que cada elemento del procesador está diseñado para un número pequeño de bits (digamos N bits). De esta manera, uniendo M elementos, es posible configurar una computadora de MXN bits.

El ejemplo que presentamos es el procesador serie 3000, de Intel. Este procesador tiene un ancho (o rebanada) de 2 bits por elemento. Para lograr un computador de 16 bits de ancho es necesario, pues, ligar o unir 8 CPU's y sus correspondientes memorias y unidades de control.

La serie 3000 es, además microprogramable. No se debe confundir el têrmino "microprogramable" con el de microprocesador. -

Estos conceptos no están ligados en forma alguna. De hecho la mayor parte de los procesadores "grandes" son microprogramados.

En esencia, el microprograma es un conjunto de instrucciones, llamada micropasos, que son los elementos básicos de una instrucción de la máquina.

Por ejemplo, la instrucción:

ADD A,B (suma A = A+B)

en donde A y B son números de punto flotante, consta de una serie de pasos más básicos. Se puede decir que cada instrucción de máquina:

ADD A,B

de un procesador microprogramable, es una llamada al microcódigo (de hecho, a una rutina de éste).

Un procesador microprogramable, pues, tiene en su interior, una pequeña computadora con memoria ROM, en la cual están los micropasos del acervo de instrucciones de la máquina en cuestión.

A este tipo de programas se les conoce como 'firmware', para distinguirlos del 'software' y del 'hardware'.

La serie 3000, como decíamos, es microprogramable. Esto hace que el diseñador defina su propio conjunto de instrucciones.

Además, este micro es bipolar. Esto significa que los tiempos de conmutación (de 0 a 1 y viceversa) son del orden de 30-50 - macrosegundos (es decir 30-50 x $10^{-9}$ seg.), con ciclos básicos de 150 ns en el procesador central.

Esto hace que este procesador sea rápido y versátil; por - otro lado, es caro (relativamente) e implica que, para cada diseño, el ingeniero debe de elaborar su propio conjunto de instrucciones y su propia arquitectura y, por supuesto, su propio 'software'.

Es posible, sin embargo, emular otro procesador conocido y mejorar su 'thruput' (relación de resultados/seg.) copiando el conjunto de instrucciones de algún procesador comercial.

Sería posible, por ejemplo, diseñar un procesador idéntico a un Z80 con serie 3000. De esta forma tendríamos el 'software' del Z80 y la velocidad de los dispositivos bipolares.

Microprocesadores Orientados a 'Bytes'.

Estos comprenden al 8080, 6800 y Z80 y son el tema fundamental de este curso.

Sus características básicas son las siguientes:

1) CPU en un 'Chip'.
2) Palabras de 8 bits (1 byte).
3) Bajo costo.

4) Amplio acervo de periféricos.

5) 'Software' de alto nivel ya desarrollado (compiladores intérpretes, ensambladores, etc.)

6) Pequeño número de integrados para lograr una configuración básica (que cumpla con la definición de computadora).

Puesto que van a ser el tema de las sesiones siguientes, dejaremos su tratamiento para capítulos posteriores, en donde son tratados con gran detalle.

Microcomputadoras.

De lo que se ha venido discutiendo, parece no ser muy obvio que a una familia se le llame microcomputadoras, cuando todas lo son. En realidad, lo que queremos señalar es que, en realidad, los procesadores antes mencionados conforman a una microcomputadora sólo mediante el uso de varios 'Chips'. En esta familia incluímos a aquellos circuitos que incorporan todo lo necesario para tener una computadora en un circuito integrado. Es decir, en un 'Chip' está concentrado el procesador, la memoria de programas y de datos, los puertos de entrada/salida y la unidad de control.

De aquí que hagamos la distinción entre un microprocesador (3000, 6800, etc.) y un microcomputador. Ejemplo de este último es el procesador 8748 de Intel. Esta micro posee puertos, CPU, 64 bytes de RAM y 1K de EPROM, todo en el mismo circuito integrado.

Micro-miniprocesadores.

Lo que deseamos señalar, al acuñar este término es que este tipo de microprocesadores tienen ya, las características de un miniprocesador:

a) Palabras de 16 o más bits de ancho.

b) Amplio espacio de direccionamiento (del orden de Megapalabras).

c) Instrucciones comunmente asociadas a máquinas "grandes".

Ejemplos de ésto son el 8086 de INTEL, el Z8000 de Moztek y el 68000 de Motorola.

Estas máquinas incluyen en su 'set' de instrucciones aritmética "complicada" (multiplicación y división), direccionamiento indirecto, et.

## Perspectivas.

Al desarrollo de los micros hay que asociar el desarrollo de otro tipo de electrónica. En particular, la electrónica de la transducción hace que sea posible atacar problemas del mundo analógico - en forma digital.

Por ejemplo, TRW ha desarrollado convertidores A/D (analógica o digital) con velocidad de conversión de 35 mseg. para 8 bits. Compañías como Analog Devices, por mencionar sólo una, tienen convertidores D/A (digital a analógico) de tiempos de conversión de 40 mseg para 8 - bits.

Las perspectivas que esto abre son prácticamente ilimitadas En la figura se muestra un sistema de control automático basado en un 8085 y los convertidores arriba mencionados.

| | | | | |
|---|---|---|---|---|
| A | I/0 | 8 | I/0 | D |
| / | | 0 | R | / |
| D | RAM | 8 | 0 | A |
| | | 5 | M | |

Este es un control analógico/analógico logrado con sólo 5 'Chips'. ¿Qué tipo de control? El que el programador desee! - Con sólo cambiar el programa interno, el sistema se convierte en otro cualquiera. Los viejos problemas de inestabilidad en controles de máquinas herramientas se eliminan con un motor de pa-

La solución de largas ecuaciones diferenciales simplemente pierde sentido.

Tiempo de respuesta en modo estable:

200 μseg.

Hay, en nuestra opinión dos campos de acción básicos para los microprocesadores: el campo industrial y el campo de la informática.

En el campo de la industria, es fácil visiumbrar las repercusiones que la siguiente consideración podrá tener:

Cualquier sistema de control puede constar de una computadora para controlar el proceso.

En el campo de la informática, es obvio que la caída de precios en los sistemas de cómputo debe repercutir grandemente. Más aún si se toma en cuenta que no sólo se abaten los precios, sino también aumenta la capacidad de cómputo.

Especular al respecto en esta área es fácil. Algunas predicciones: las computadoras se convierten en artículos de hogar; los sistemas de reconocimiento y síntesis de voz permiten rápidos avances en robótica; las memorias de dico desaparecen para ceder su lugar a CCD's y memorias de burbuja; las arquitecturas de computadoras se orientan a multi-microprocesamientos (pioneros en esta ârea son Cm* y Tandem, de propósito especial, así como AHR para LISP); en resumen, la necesidad de conocimiento en el área de la microelectrónica y, en particular, de los microprocesadores se presenta como algo inmediato. La tecnología de los microprocesadores está firme y bien establecida.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# INTRODUCCION A LA ARQUITECTURA DE LOS MICROPROCESADORES

ING. MARIO RODRIGUEZ MANZANERA

MARZO, 1979

La microcomputadora 8080 es una unidad central de proceso (CPU) de 8 bits para uso en sistemas de computación de propósito general. Se encuentra encapsulado en un sólo circuito integrado (CHIP) LSI de gran integración construido con tecnología MOS (METAL OXIDE SEMICONDUCTOR) en silicón compuerta canal n, lo cual le permite tener un ciclo de instrucción de 2 µseg. Al interconectarlo con puertos de entrada/salida y memorias RAM y ROM de cualquier tipo y velocidad, constituye un micro sistema de cómputo completo.

Las características de mayor interés en su arquitectura se indican a continuación:

El CPU (8080) contiene 6 registros de trabajo de 8 bits cada uno, un acumulador (8 bits), 4 registros temporales (8 bits), 4 banderas accesibles al programador y una unidad lógica aritmética en paralelo de 8 bits.

El conjunto de instrucciones del 8080 incluye operaciones de aritmética decimal y cuenta con la facilidad de realizar operaciones sobre palabras de 16 bits, además cuenta con una arquitectura de Stack lo que le permite accesar cualquier porción de su memoria externa como área reservada. Aquí puede almacenar el contenido de registros y banderas bajo el control del programador utilizando las instrucciones de Stack, además de actualizar en cada acceso el apuntador de 16 bits (SP) reservado para este fin.

El sistema de interrupciones del 8080 hace uso del SP en forma automática guardando al contador de programas al recibirse - una interrupción. Por las características de funcionamiento de - stack se permiten múltiples niveles de interrupción.

La 8080 contiene además un PC contador de programa de 16 bits el cual puede accesar cualquier localidad de memoria en donde el usuario haya colocado su programa.

La comunicación al exterior del 8080 se lleva a cabo por - medio de 16 líneas de dirección, y un bus bidireccional de datos de 8 bits. Las señales de control que no requieren de codificación salen y entran directamente del/al procesador. Los buses del sistema son TTL compatibles.

En la figura 1 se presenta un diagrama a bloques de la arquitectura del CPU 8080.

En la figura 2 la configuración de pins del 8080.

Como puede observarse el circuito tiene 40 pins, los que a continuación se describen:

$A_0$-$A_{15}$ Bus de direcciones, salida 3 estados, puede direccionar 64K palabras de 8 bits, 256 puertos de entrada y 256 puertos de salida. $A_0$ es el bit menos significativo.

$D_0$-$D_7$ Bus de datos, bidireccional, 3 estados, permite la comunicación con el mundo exterior del 8080.

SYNC Señal de sincronía, salida, indica el inicio de un ciclo de máquina.

DBIN Bus de datos disponibles para entrada, salida.

READY Señal de listo, entrada, se emplea para sincronizar al CPU con dispositivos lentos. Si después de mandar una dirección la 8080 no recibe la señal de ready, cae en un estado de espera (wait) hasta que ésta se restablece.

WAIT Señal para que el mundo externo conozca que el CPU está en estado de espera.

$\overline{WR}$ Señal que indica a la memoria o puerto direccionado que el contenido del bus es para una escritura.

HOLD  Señal de entrada para pedir acceso directo a memoria.

HLDA  Señal de salida en respuesta de aceptación a HOLD.

INTE  Señal de salida indica que el CPU puede ser interrumpido, indica el estado de 1 flip flop que es manejado por el usuario con las instrucciones EI y DI.

INT  Señal de entrada por donde el CPU reconoce que alguien desea interrumpir su actividad.

RESET  Señal de entrada por medio de la que el usuario limpia el contenido del PC e inicializa su actividad en la dirección 0.

$\phi_1$ , $\phi_2$  Señales de reloj. (2 fases).

Polarizaciones  Vcc = +5 Volts  Vss = TIERRA (Referencia A ...)
Vdd = +12 Volts  Vbb = -5 Volts.

En la figura 3 se presenta el ciclo básico de instrucción. Como puede observarse este ciclo se encuentra dividido en 6 estados:

T1, T2, Tw, T3, T4 y T5.

En la figura 4 se muestra el diagrama de transición de estados del CPU.

Como puede observarse de T1 a T3 se realiza el ciclo de - Fetch, y de T4 a T5 la ejecución de la instrucción.

Cada ciclo de máquina M1,.M2,.M5, puede requerir de 3 a 5 estados T1 a T5 para su realización.

Cada estado tiene una duración de 0.5 μseg, los ciclos de máquina M2,...M5, normalmente tienen 3 períodos de reloj cada uno.

Existen además 3 estados independientes entre sí, el estado de (WAIT, HOLD y HALT) espera captura y alto.

A máxima frecuencia 2MHZ las instrucciones de máquina dependiendo del número de ciclos M utilizados en su ejecución serán de 2 a 9 μseg.

Como se había hecho notar anteriormente, algunas de las señales de control se encuentran codificadas por la 8080.

En la figura 5 se incluye la codificación de la palabra de estatus y su significado.

El conocimiento de la palabra de estatus nos permitirá en el futuro realizar interfases con dispositivos externos para los que algo de sincronización es requerida. Por ejemplo: un circuito paso a paso (single step) para monitorear el funcionamiento del sistema.

## Descripción funcional.

En la misma forma que los sistemas medianos y grandes cada registro y banderas indicadoras corresponden a una codificación interna, la cual será considerada por la unidad de control en la ejecución de las instrucciones a continuación de cada ciclo de fetch.

Esta codificación corresponde a la mostrada.

| Codificación. | Registro. |
|---|---|
| 000 | B |
| 001 | C |
| 010 | D |
| 011 | E |
| 100 | H |
| 101 | L |
| 110 | Memoria |
| 111 | Acumulador |

| Codificación | Bandera |
|---|---|
| 01 | Carry (acarreo) |
| 00 | Zero (cero) |
| 11 | Sign (signo) |
| 10 | Parity (paridad) |

Su efecto será tratado en el capítulo dedicado a la programación del sistema y su relación con el conjunto de señales de control durante la exposición.

Sistema mínimo:

Ya que el CPU no es independiente de un elemento que le alimente con las frecuencias seguidas $\phi_1$, $\phi_2$ y de otro que almacene la palabra de estatus, será pues necesario que el CPU se encuentre conectado a un circuito del tipo 8224 y a un 8228 que explicaremos posteriormente, por lo pronto en la figura 6, se incluye la arquitectura del "Sistema" 8080, dividida en lo que llamaremos interfase estandar y sistema estandar.

Generador de reloj 8224:

El 8224 es un circuito integrado que genera las fases de reloj para el CPU, a partir de la frecuencia de oscilación de un - cristal seleccionado por el diseñador, lo cual le permite cubrir sus regeneramientos de velocidad de operación, además incluye la circuitería necesaria para la sincronización del sistema en lo referente a encendido, en la figura 7, se incluye el diagrama a bloques del generador de reloj, y a continuación la información relativa a este circuito.

Controlador del sistema.

El 8228 es un circuito integrado que maneja las relaciones del CPU con el sistema externo en lo referente a transacciones del Bus de datos y almacenamiento y decodificación de la palabra de estatus.

En la figura 8, se muestra el diagrama a bloques del 8228 y en la exposición se hace referencia a la figura 5.

## Sistema 8080.

Más de 12 diferentes circuitos se han desarrollado para formar y expander el sistema 8080, tanto memorias ROM (read only memory) y RAM (random access memory), como puertos con todo tipo de facilidades y provistos con la circuitería necesaria para efectuar la interfase independientemente del CPU. Entre éstos enlistamos los que pensamos pueden resultar de mayor aplicación:

| | |
|---|---|
| 8212 | Puerto de 8 bits para entrada/salida. |
| 8255 | Interfase de periféricos (3 puertos) programable. |
| 8251 | Interfase universal de comunicación |
| 8214 | Unidad de control de prioridades para interrupción. |

de los cuales se incluye información a continuación y serán tratados minuciosamente en el curso.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

# MICROPROCESADORES: TEORIA Y APLICACIONES

## LA MICROCOMPUTADORA 6 800

M. EN C. MANUEL ESTEVEZ

MARZO, 1979

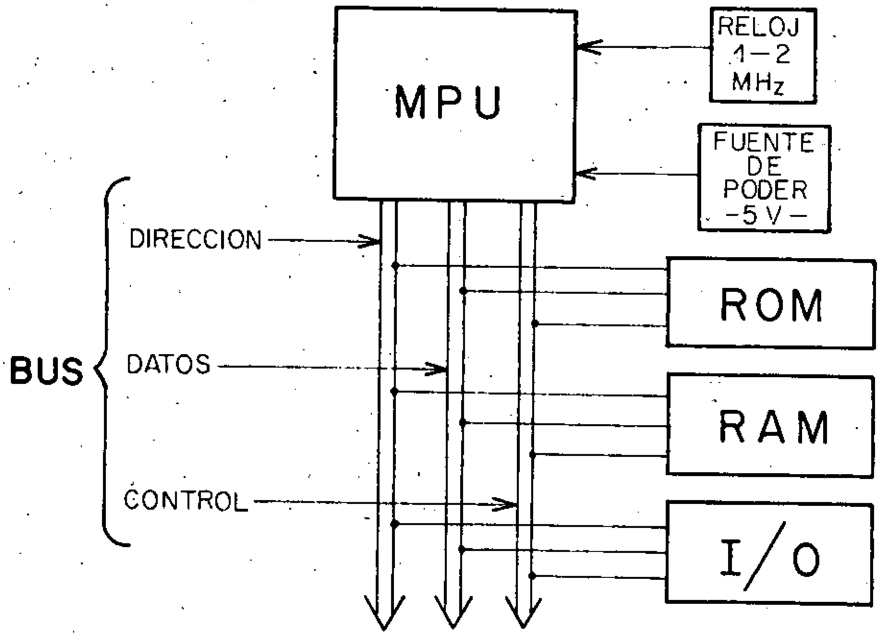
# INTRODUCCION

El MPU* 6800 es el alma de una serie de bloques de diseño de tecnología NMOS, conectados a través del sistema de líneas (BUS) y que son interconectados en una determinada configuración para formar una microcomputadora.

## COMPONENTES BASICOS DE UNA MICROCOMPUTADORA

Un sistema mínimo puede ser ensamblado con 4 componentes LSI** orientados en el sistema de "BUS":

| | |
|---|---|
| MPU | Microprocesador |
| RAM | (Random Access Memory) |
| ROM | (Read only memory) |
| I/O | (Input/output) |



* MPU: Microprocessing Unit

** LSI: Large Scale Integration

MPU: * Componente central de la microcomputadora

* Sistema ejecutivo
* Desarrolla la mayoría de las funciones lógicas

*Bus del Sistema:

* El microprocesador se comunica con otros componentes vía el BUS de datos.
* El BUS de dirección selecciona las localidades de memoria
* El BUS de control proporciona el secuenciamiento (timing) y otras señales requeridas para la operación adecuada de los otros componentes de la familia.

ROM: * Usado para el almacenamiento de instrucciones

* Bajo costo por bit
* No volátil, permanente
* Tablas

RAM: * Usado para almacenar datos

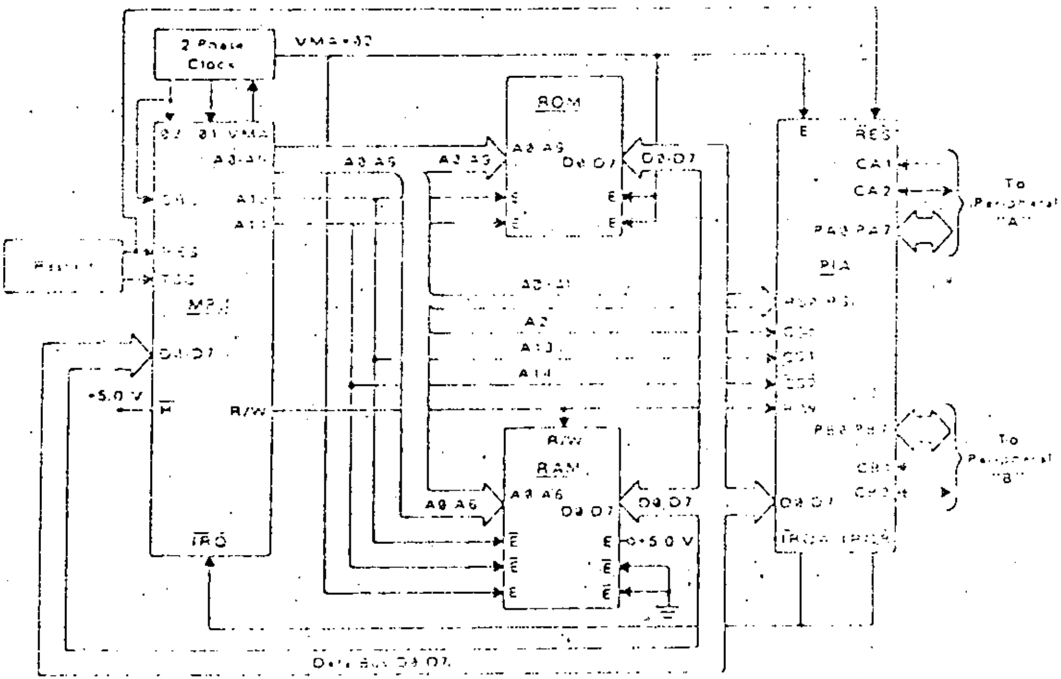
* Resultados de cálculos, datos temporales
* Puede ser cargado con instrucciones

I/O: * Todos los sistemas deben tener interfases para salida/entrada.

* Un adaptador periférico de entrada/salida

Estas partes pueden ser interconectadas sin necesidad de interfases entre ellas, haciendo un sistema funcional mínimo, el cual -- puede ser facilmente adaptado a un buen número de aplicaciones, simplemente cambiando el programa de aplicación contenido en el ROM.

Este sistema mínimo puede ser expandido, previniendo que la carga no exceda la capacidad del MPU. El MPU tiene capacidad para manejar

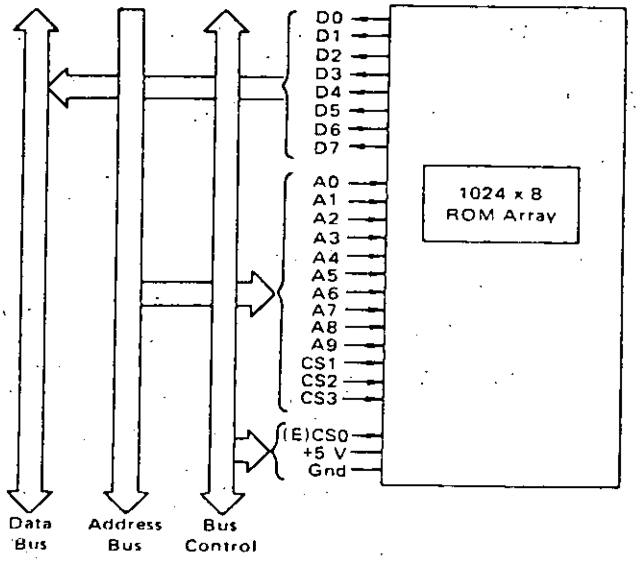

2 Phase Clock
VMA•φ2
φ2 φ1 VMA
A0-A15
A0-A5
A0-A9
A8-A9
D0-D7
ROM
A10
A11
Reset
RES
TSC
MPU
D0-D7
+5.0 V
R/W
IRQ
A0-A1
A2
A13
A14
R/W
RAM
A0-A6
D0-D7
+5.0 V
Data Bus D0-D7
E
RES
CA1
CA2
PA0-PA7
PIA
RS0-RS1
CS0
CS1
CS2
R/W
PB0-PB7
CB1
CB2
D0-D7
IRQA IRQB
To Peripheral "A"
To Peripheral "B"

un TTL * estandard y 130 picofarads a 1 megahertz.

MPU: Unidad de procesamiento

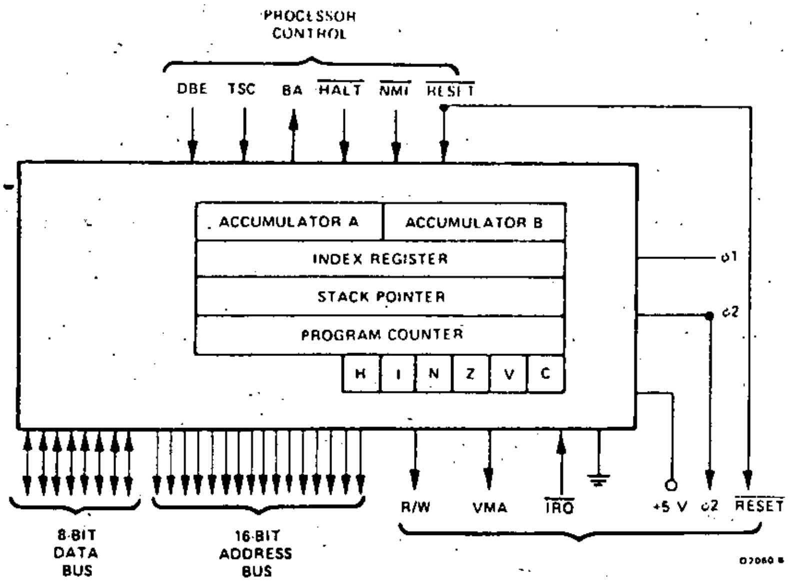
- * Procesador de 8 bits en paralelo
- * Tiempo mínimo de ejecución de una instrucción 2 μS a 1MHz
- * Una sola fuente de poder de + 5 V
- * BUS de datos bidireccional (8 bits)
- * BUS de dirección de 16 bits, pudiendo adireccionar 65000 bytes**
- * 72 instrucciones
- * 7 modos de direccionamiento
- * Varios tipos de interrupción
- * Dos acumuladores, registro de índice, apuntador de stack, apuntador de programa, registro de condición (6 bits)
- * Control para halt y DMA***
- * Señales de control, como: R/W, DBE, VWA, etc.

ROM: READ ONLY MEMORY



* TTL: Transistor-Transistor Logic
** Byte: Elemento binario de 8 bits, llamado también palabra
*** DMA: Direct memory access

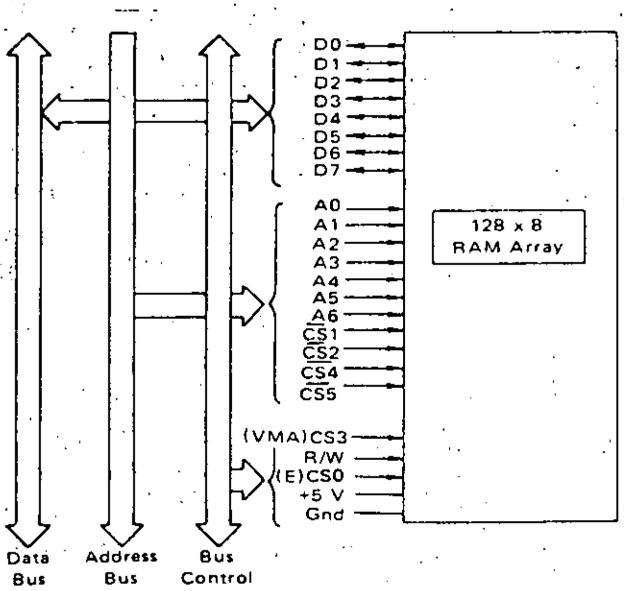
## M6800 MICROPROCESSOR



La mayoría de estas memorias son compatibles con el 6800, comunmente se usan memorias programadas por mascarilla, es decir se programan desde fábrica, son efectivas en el costo cuando se va hacer -- una producción en masa de algún sistema. Por otro lado, cuando el volumen es pequeño o se trata de un prototipo, es preferible usar las eléctricamente programables*, que pueden ser borradas con luz ultravioleta.

RAM: RANDOM ACCESS MEMORY

Se muestra especfficamente la memoria estática de 128 x 8 de Motorola (6810) proporciona 128 bytes, es muy fácil de aplicar y para expansión está provista de varias entradas para la selección del componente (CS*), para sistemas que requieran mayor cantidad - de memoria es preferible usar las de 1024 x 1 hasta 8K bytes y para mayores las de 4096 x 1 ó 16K bits que ya existen en el mercado.



* EPROM: MM 5204, 2708
(512x8)(1024x8)

* CS: Chip Select

I/O: IMPUT/OUTPUT

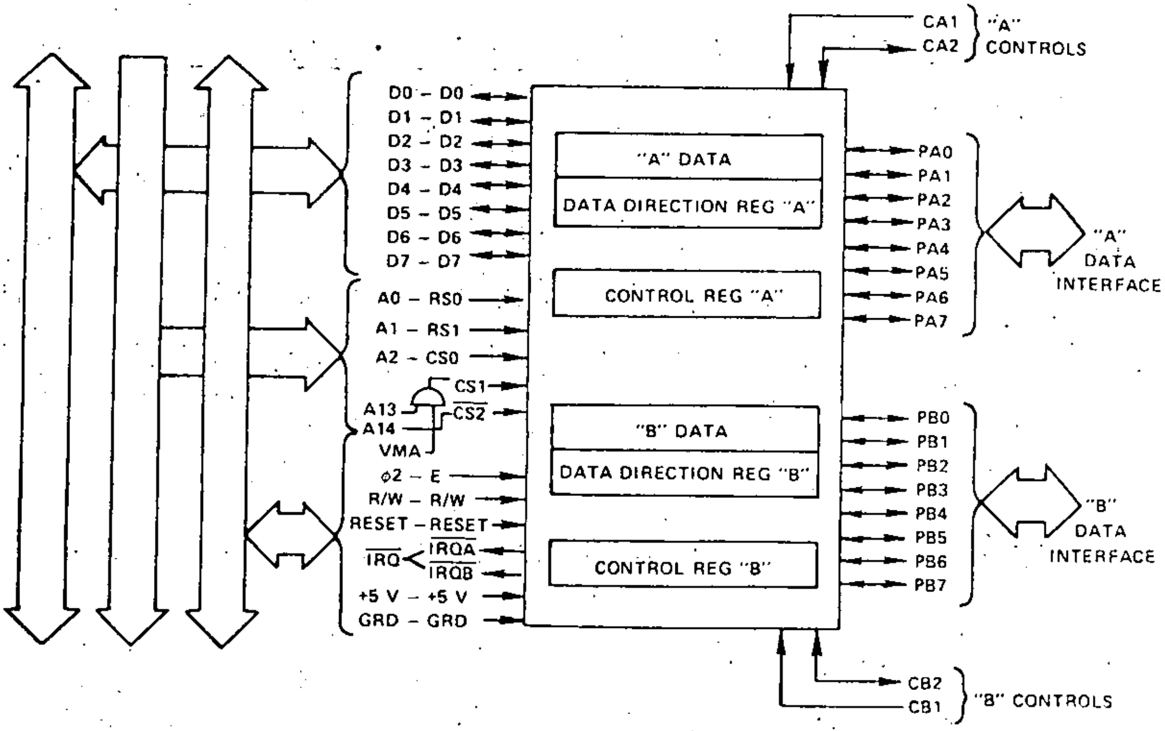
*PIA - Peripheral interface adapter.

Este periférico es el componente más poderoso de la familia, es una interface de entrada/salida programable.

Proporciona 16 líneas bidireccionales de I/O en dos puertos de 8 cada uno, cada BIT es individualmente programado para que se comporte como entrada o salida.

Puede programarse para detectar interrupciones y proporciona enlace automático (handshaking). Se interconecta tan simplemente como una memoria, ocupando dos localidades de memoria.

**MC6820 PERIPHERAL INTERFACE ADAPTER**



# PROGRAMACION DEL MICROPROCESADOR 6800

## CODIGO DE MAQUINA

Cada una de las 72 instrucciones son ensambladas de 1 a 3 bytes de código de máquina. El número de bytes depende en una instrucción en particular y en el modo de direccionamiento (que se discutirá posteriormente).

El código del primer (o único) byte correspondiente a una instrucción ejecutable es suficiente para identificar la instrucción y el modo de direccionamiento.

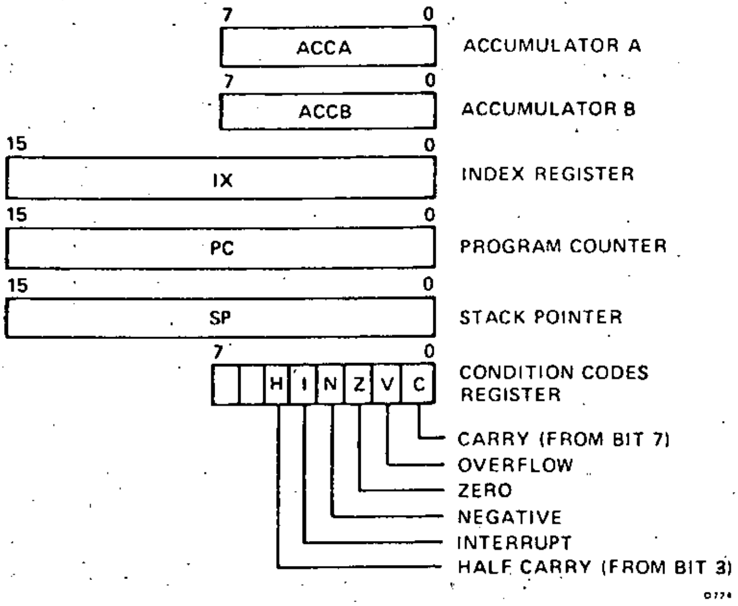
Cuando una instrucción se traduce en dos o tres bytes, el segundo byte o el segundo y el tercero contienen, un operando, una dirección o información que se obtiene de una dirección.

El Contador de Programa es un registro de 16 bits el cual es usado para dirigir el flujo del programa de una instrucción a otra. Este puede direccionar directamente instrucciones en cualquier parte de los 65 K de memoria.

Después que una instrucción ha sido ejecutada, el contador de programa es incrementado automáticamente a la siguiente localidad en memoria de la cual será tomada la siguiente instrucción, a menos que la instrucción actual, dirija al pro

NOTA: Se entenderá modo de direccionamiento como: addressing mode.

## PROGRAMMING MODEL OF THE MICROPROCESSING UNIT



cesador a una localidad diferente de memoria como un salto o instrucción de retorno.

Los dos Acumuladores del 6800 son los caballos de fuerza desde el punto de vista de programación, estos acumuladores, A y B, son de 8 bits cada uno.

La capacidad de estos acumuladores es prácticamente la misma. Todas las operaciones aritméticas y lógicas "Acumulan" su resultado en uno de estos registros, además, casi toda la transferencia de datos de una localidad de memoria a otra, usa uno de estos acumuladores como almacén intermedio. También un sin número de operaciones como rotar, comparar, usa estos acumuladores. Además que las banderas del registro del código condicional, son afectadas en casi todas las operaciones con los acumuladores.

El Registro de Indice, es un registro de 16 bits usado principalmente como un apuntador de memoria. Puede ser usado para apuntar a una localidad de memoria en la que varias instrucciones operan de memoria a acumulador o puede indicar una tabla o área de almacén para transferir datos de memoria a acumulador o viceversa o también puede servir como un contador hacia arriba o hacia abajo y para instrucciones que usan el modo indizado permite el desplazamiento de la dirección sumado al registro de índice para determinar la localidad de memoria que se usará.

El Apuntador de Stack, es un registro de 16 bits que de be de prepararse para apuntar un área en memoria que será usada como Stack.

El "Stack" ès un área de almacenamiento en la que el 6800 guarda las direcciones de retorno de subrutina y los datos -- pertinentes que deben almacenarse cuando ocurre una interrupción. Por lo tanto debe de asignársele un área en memoria tipo RAM.

## LA ESTRUCTURA DE ENTRADA/SALIDA.

El 6800 permite la transferencia de datos con interfases periféricas al asignar direcciones de memoria al periférico. Al asignar localidades de memoria como el canal a través del cual se transfieren datos a y de periféricos, nos hace posi-- ble el uso de cualquier instrucción que refiere a memoria para la transferencia de datos de entrada/salida. Esto le da - gran flexibilidad al programador para probar el status y con- trolar apropiadamente los periféricos.

## STACK Y APUNTADOR DE STACK.

El Stack consiste en cualquier número de localidades en memoria RAM. El Stack proporciona almacenamiento temporal y recuperación de bytes sucesivos e información, que puede incluir lo siguiente:

* El status actual del MPU

* La Dirección de regreso

* Datos

El Stack puede ser usado para los siguientes propósitos:

* Control de interrupciones

* Enlace de subrutinas

* Almacenamiento temporal de datos (bajo control de programa).

* Código reentrante

El microprocesador incluye un apuntador de stack de 16 bits, este contiene la dirección que permite al MPU encontrar la localidad actual del stack.

Cuando un byte de información se almacena en el stack, - es almacenado en la dirección que contiene el apuntador de -- stack. El apuntador del stack es decrementado, por uno, inmediatamente, después de almacenar en el stack cada byte de información. Recíprocamente, el apuntador de stack es incrementado por uno inmediatamente antes de recuperar cada byte de - información del stack y este byte es obtenido de la dirección contenida en el apuntador del stack. El programador debe asegurarse que el apuntador del stack es inicializado con la dirección requerida antes de ejecutar alguna instrucción que manipule el stack.

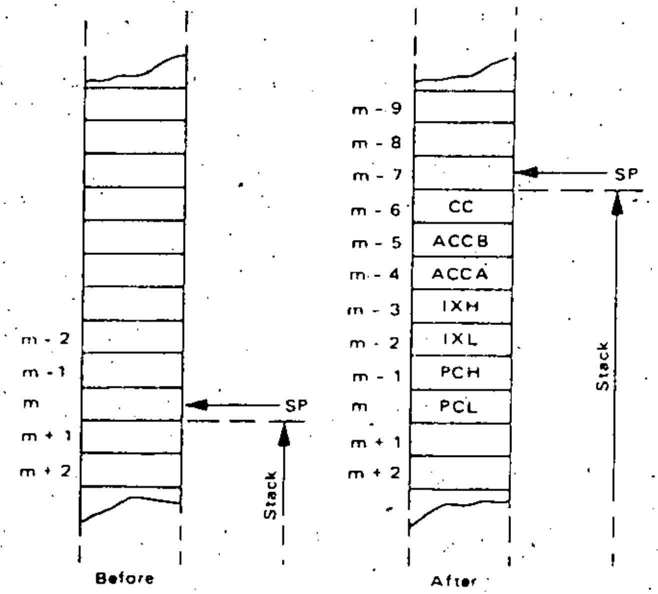
## SALVAGUARDANDO LOS REGISTROS DEL MPU (STATUS)

El status del microprocesador es guardado en el stack - durante las siguientes operaciones:

* En respuesta a una condición externa indicada por una transición negativa del control de interrupción no mascarillable. (NMI)

- * Durante la ejecución de un código de lenguaje de máquina que corresponda a las instrucciones SWI (Software Interrupt) o WAI (Wait for Interrupt).
- * Al dar servicio a una interrupción de un periférico, en respuesta a una transición negativa de la señal de interrupción $\overline{\text{IRQ}}$ (Interrupt Request), siempre y cuando el BIT de interrupción I (en el registro condicional) está apagado.

El status es almacenado en el Stack, de acuerdo a la siguiente figura:



SP = Stack Pointer.
CC = Condition Codes (Also called the Processor Status Byte)
ACCB = Accumulator B
ACCA = Accumulator A
IXH = Index Register, Higher Order 8 Bits
IXL = Index Register, Lower Order 8 Bits
PCH = Program Counter, Higher Order 8 Bits
PCL = Program Counter, Lower Order 8 Bits

Saving the Status of the Microprocessor in the Stack

Antes de almacenar el status, el apuntador de Stack, - contiene la dirección de la localidad de memoria "m" (Fig. ). El status es almacenado en bytes de memoria, empezan do en la localidad "m" y acabando en la "m $-6$"; el Stack se decrementa en uno cada vez que entra un byte de información.

El valor almacenado para el contador de Programa (PCH y PCL), sigue las siguientes reglas:

1.- En respuesta a una interrupción no mascarillable o a la interrupción de un periférico, el valor guardado para el contador de programa es la dirección de la instrucción que sería la siguiente en ser ejecutada, si la interrupción no hubiera sucedido.

2.- Durante la ejecución de las instrucciones SWI o WAI, el valor guardado para el contador del programa es la dirección de SWI o WAI más uno.

El valor almacenado para los otros registros:

CC: Código Condicional

ACCB: Acumulador B

ACCA: Acumulador A

IXH: Registro de Indice (HIGH)

IHL: Registro de Indice (LOW)

y de acuerdo con las siguientes reglas:

1) En respuesta a una interrupción no mascarillable o a la interrupción de un periférico, los valores - - guardados son aquello que resultaron de la última - instrucción que se ejecutó antes que se diera servicio a la interrupción.

2) Durante la ejecución de las instrucciones SWI o WAI, los valores guardados son aquellos que resultaron de

la última instrucción que se ejecutó antes de la instrucción SWI o WAI.

3) Los códigos de condición H, I, N, Z, V, y C.

H - Half carry (carry out of bit 3)

I - Interrupt mask

N - Negative (sign bit)

Z - Zero

V - Overflow

C - Carry - borrow

De la posición 5 a 0 del registro del código de condición del procesador

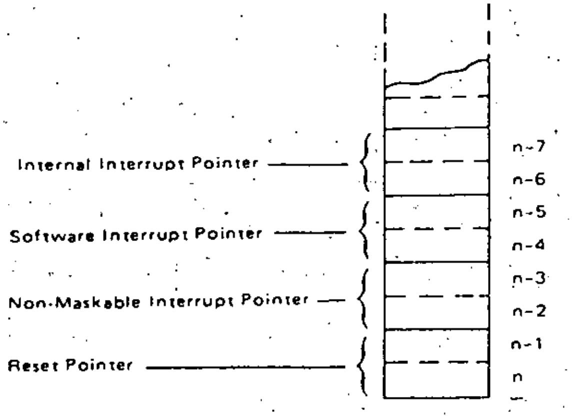
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | H | I | N | Z | V | C |

son almacenados respectivamente en la posición 5 a 0 de la localidad de memoria del Stack. (localidades 6 y 7 se ponen en uno).

## APUNTADOR DE INTERRUPCION

Un bloque de memoria se reserva para apuntadores, que proporcionan (en ROM) la dirección de programa que deberán ejecutados en el evento de encendido (Reset), de una transición baja de la interrupción no mascarillada, una interrupción por programa (SWI) o a la respuesta a una interrupción de un periférico.

Los respectivos apuntadores ocupan 2 bytes de memoria y se localizan de n - 7 a n como se muestra.


Internal Interrupt Pointer
Software Interrupt Pointer
Non-Maskable Interrupt Pointer
Reset Pointer
n-7
n-6
n-5
n-4
n-3
n-2
n-1
n

n = Memory Location Addressed When All Lines of The Address Bus are in the High (1) State.

Reset and Interrupt Pointers

### Reset (Encendido)

Esta entrada de control es usada para empezar la ejecución del programa, ya sea por un encendido inicial o de una condición de baja de potencia que sigue de una falla en la alimentación.

Cuando se detecta una transición positiva, el contador de programa es cargado con la dirección almacenada en el - apuntador de restauración (Reset Pointer) el MPU procede a la ejecución del programa de restauración, que comienza con la instrucción que se encuentra en la localidad direccionada por el contador de programa.

NMI (Interrupción no mascarillable).

La secuencia es iniciada cuando el MPU recibe una transición negativa, en la línea de control NMI. En respuesta, el MPU termina la ejecución de la instrucción actual o de - una mas, debido al procedimiento de realizar por adelantado (look-ahead). El Status del MPU es aguardado en el Stack como se ha descrito anteriormente, el contador de programa se carga con la dirección almacenada en el apuntador NMI, localizado en m-3 y m-2.

El MPU comienza a ejecutar el programa que empieza en - la instrucción direccionada por el contador de programa.

SWI (Interrupción por programa).

Durante la ejecución de la instrucción SWI, el Status - del MPU es guardado en el Stack, como fue descrito anteriormente. El valor guardado para el apuntador del programa es la dirección de la instrucción SWI - mas uno.

Después de ser guardado el Status, el bit de interrupción I (en el registro del código condicional) es puesto en 1, (I=1), el MPU no responderá a la petición de una interrupción de un periférico mientras que el bit de interrupción -- esté opuesto en 1. El contador de programa es cargado con -

la dirección almacenada en el apuntador de interrupción por programa puesto en las localidades m-5, m-4, el MPU procede a la ejecución del programa determinado por la primera instrucción direccionada por el contado de programa.

IRQ (Petición a interrupción).

La petición de interrupción por un periférico es señalada por una transición baja de la línea de control IRQ.

El MPU no responderá a este llamado mientras el bit de interrupción I, esté en 1 (I=I). La ejecución del programa continúa normalmente hasta que el bit de interrupción sea puesto en 0, (i=0), habilitando al MPU a responder a esta petición de interrupción.

La ejecución de la instrucción en progreso siempre se completara antes que el MPU responda a esta interrupción.

La respuesta del MPU a la petición de interrupción sigue el siguiente procedimiento:

1.- "Salvar el Status".

Siempre que la última instrucción no sea WAI el Status es guardado en el Stack, el valor guardado para el contador de programa es la dirección de la instrucción que sería la siguiente en ejecutarse si la interrupción no hubiera ocurrido. Si la última instrucción fue WAI, la dirección de la siguiente instrucción fue ya guardada por esta instrucción, preparándose para interrupción.

2.- "Bit de interrupción".

El bit de interrupción es puesto en 1 (I=1).

Esto previene al MPU de responder a otras peticiones de interrupción hasta que este bit no sea puesto en cero.

3.- "Apuntador interno de interrupción y programa".

El apuntador de programa es cargado en la dirección - almacenada en el apuntador interno que está en las localidades n-7 y n-6. El MPU entonces procede a la ejecución del - programa de interrupción interna, que comienza con la instrucción direccionada por el contador de programa.

En un sistema en el que hay mas de una fuente posible de petición de interrupción, el programa interno de interrupción debe de incluir alguna rutina para identificar el origen de esta petición.

En el sistema motorola se puede interrogar los registros de las PIAS como manera de identificar quién hizo la petición para poderle dar servicio.

WAI - Instrucción de espera de una interrupción.

Durante la ejecución de la instrucción WAI, el Status del MPU es guardado en el Stack, como se describió anteriormente. La instrucción WAI no cambia el bit the interrupcion.

Si el bit the interrupción es puesto en 1 (I=1) el - MPU no puede responder a ninguna petición de interrupción - y se para, solo con un reset o una interrupción no masca - -

rillable puede resuminr la ejecución.

Si el bit de interrupción está apagado (I=0) el MPU dará servicio a cualquier petición de interrupción y a continuación pondrá el bit de interrupción en 1 (I=1). El contador de programa será cargado con la dirección almacenada en el apuntador interno y la ejecución del programa empezará.

"Manipulación del bit de interrupción".

Este bit (I) es afectado por instrucciones de lenguaje fuente como SWI y RTI y al dar servicio a una petición de interrupción de un periférico como se ha visto.

Este bit I de interrupción puede ser afectado también por las siguientes instrucciones:

CLI - Clear interrupt mask bit
(Pone en cero I=0 el bit de interrupción).

SEI - Set interrupt mask bit.
(Pone en uno I=1 el bit de interrupción).

TAP - Transfer Acc. A to processor condition codes register.
(Transfiere el contenido del acumulador A al registro del código condicional del procesador).

TPA - Transfer the processor condition codes register to Acc. A.

(Transfiere el registro de código condicional al acumulador A).

RTI - Regreso de una interrupción.

La ejecución de esta instrucción consiste de la restauración del Status sacado del Stack, en el procesador, la - operación es el reverso de la realizada en la Fig. son sacados 7 bytes de información del Stack y almacenados en los registros respectivos del MPU. La dirección del - - apuntador del Stack es incrementada antes de sacar cada byte del Stack.

Después de la ejecución de la instrucción RTI, el estado de cada código condicional (H, I, N, Z, V, C) será el -- que haya sido sacado del Stack.

Debe notarse que el bit de interrupción puede ponerse en cero o uno al ejecutarse la instrucción RTI.

## ENLACE DE SUBRUTINAS

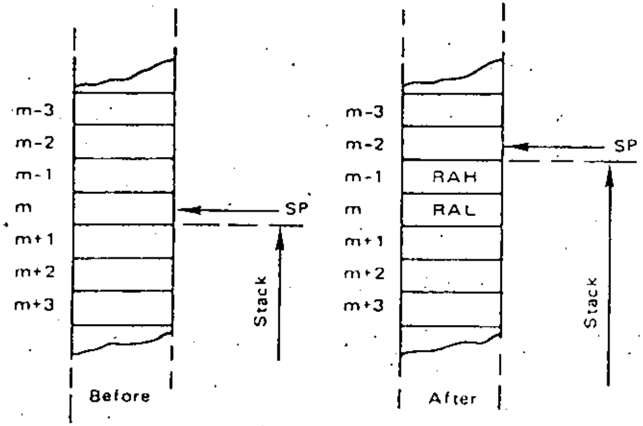
El Stack proporciona un método ordenado de llamar a -- subrutinas y regresar de ellas. Además nos permite llamar subrutinas dentro de otra subrutina, (Subroutine Nest).

## LLAMADOS DE SUBRUTINAS, (BSR, JSR)

En la ejecución de un código de máquina correspondiente a una instrucción BSR, (Branch to subroutine), o JSR - (Jump to subroutine), es guardada una dirección de retorno en el Stack.

La dirección de retorno es guardada en el Stack de acuerdo a la Fig.

Para cualquiera de las instrucciones (BSR o JSR), la dirección de retorno guardada en el Stack es aquella dirección que sigue a los bytes de código que corresponden a las instrucciones de BSR o JSR.



SP = Stack Pointer

RAH = Return Address, Higher Order 8-Bits

RAL = Return Address, Lower Order 8-Bits

Saving a Return Address in the Stack

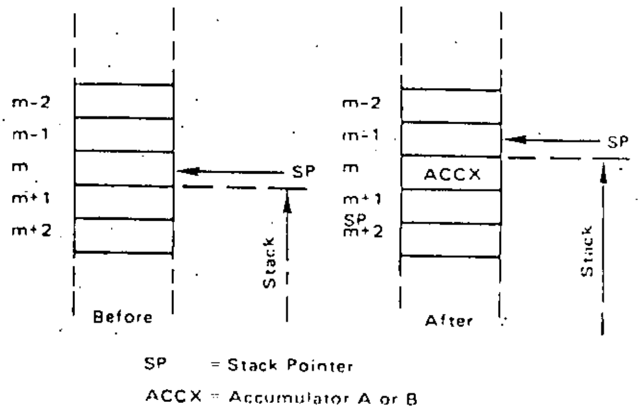
RTS Retorno de una subrutina.

Durante la ejecución de la instrucción RTS, la dirección de retorno es obtenida del Stak y cargada en el contador de programa. La dirección almacenada en el apuntador de Stack es incrementada antes de que cada byte sea sacado del Stack.

Almacenamiento de datos en el Stack.

La instrucción PSH es usada para almacenar un solo byte de información en el Stack, la instrucción direcciona al registro A o al B. De acuerdo a la Fig.

De manera contraria la instrucción PUL trae datos del Stack.



Data Storage in the Stack

La dirección en el apuntador de Stack puede también ser manipulada sin tener que almacenar o sacar información del Stack; esto se lleva a cabo con las siguientes instrucciones de lenguaje de máquina:

DES - Decrementar el apuntador de Stack.

INS - Incrementar el apuntador de Stack.

LDS - Cargar el apuntador de Stack.

TXS - Transfiere el registro de índice al apuntador de Stack.

STS - Almacena el apuntador de Stack.

TSX - Transfiere el apuntador de Stack al registro de índice.

## MODOS DE DIRECCIONAMIENTO.

"Direccionamiento inherente"

En muchos casos, el operador nemotécnico por sí mismo especifica uno o más registros que contienen operandos o en el que los resultados son guardados. Por ejemplo el operador "ABA" (sumar acumuladores AIB) requiere dos operandos - que están localizados en el acumulador A y acumulador B del microprocesador. El operador también determina que el resultado de la ejecución sea guardado en el acumulador A.

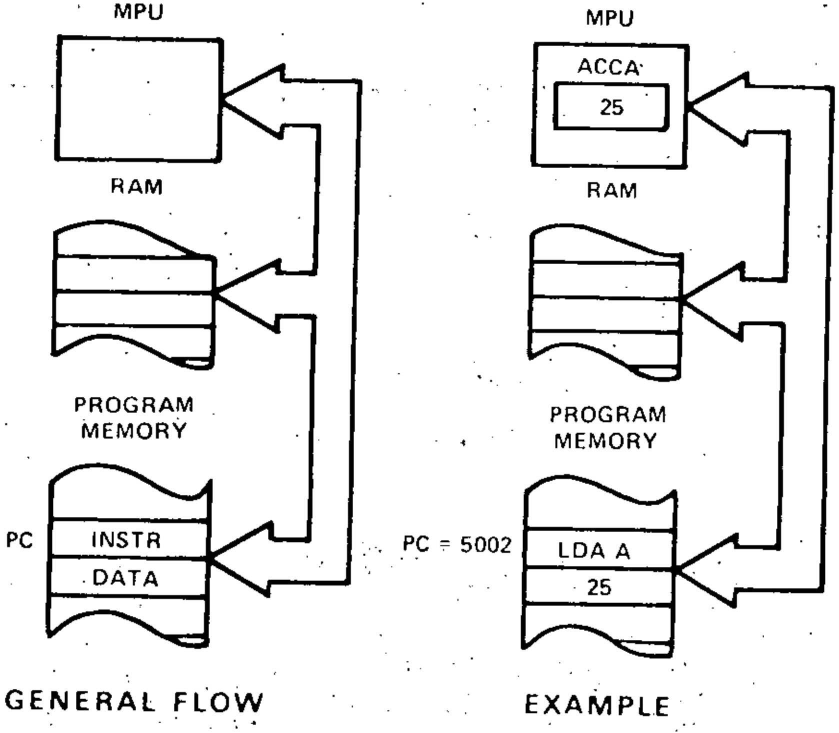
El ensamble de este tipo de instrucciones resulta de - un solo byte de lenguaje de máquina.

"Direccionamiento inmediato"

Este mod de direccionamiento es seleccionado en el -- lenguaje ensamblador por el caracter # cuando el modo inmediato de direccionar es usado el operado contiene el valor numérico actual en un rango de 0 a 255.

En este modo, toma el operando, de la localidad de memoria siguiente a la instrucción, su valor.

# IMMEDIATE ADDRESSING



D727-1

## IMMEDIATE ADDRESSING

- Use for program constants
- 1-$\mu$s/byte execution time

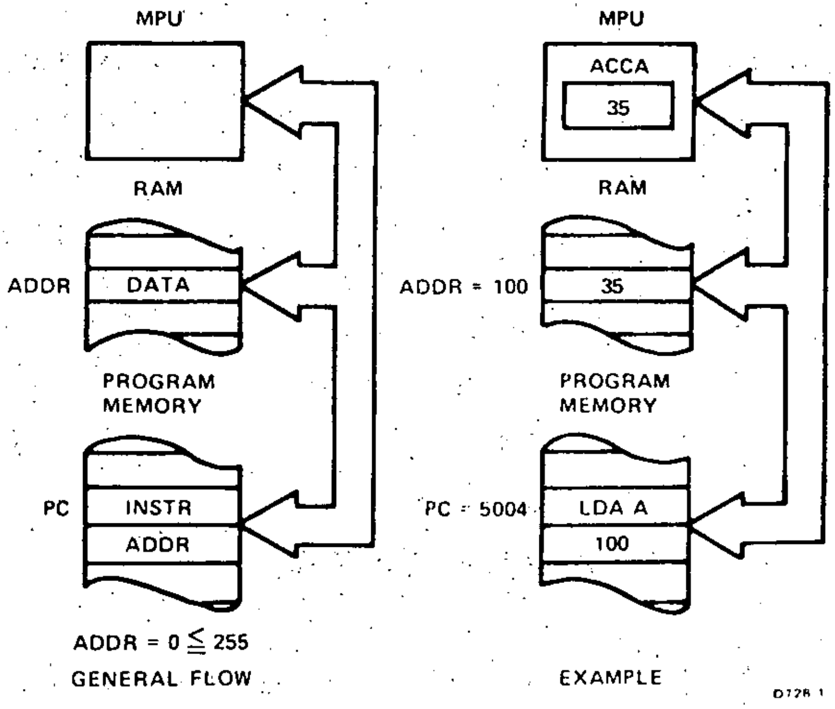
"Direccionamiento Directo y Extendido".

En el direccionamiento directo la instrucción es traducida en dos bytes de código de máquina. El segundo byte contiene la dirección en forma de 8 bits.

En el direccionamiento extendido la instrucción es traducida en 3 bytes de lenguaje de máquina. El segundo byte contiene los 8 bits más altos de la dirección. El tercer - byte contiene los 8 bits más bajos de la dirección.

Para aquellas instrucciones que pueden usar el modo directo así como el extendido, el directo será usado para tener acceso a los primeros 256 bytes de memoria y el extendido para tener acceso a cualquier localidad.

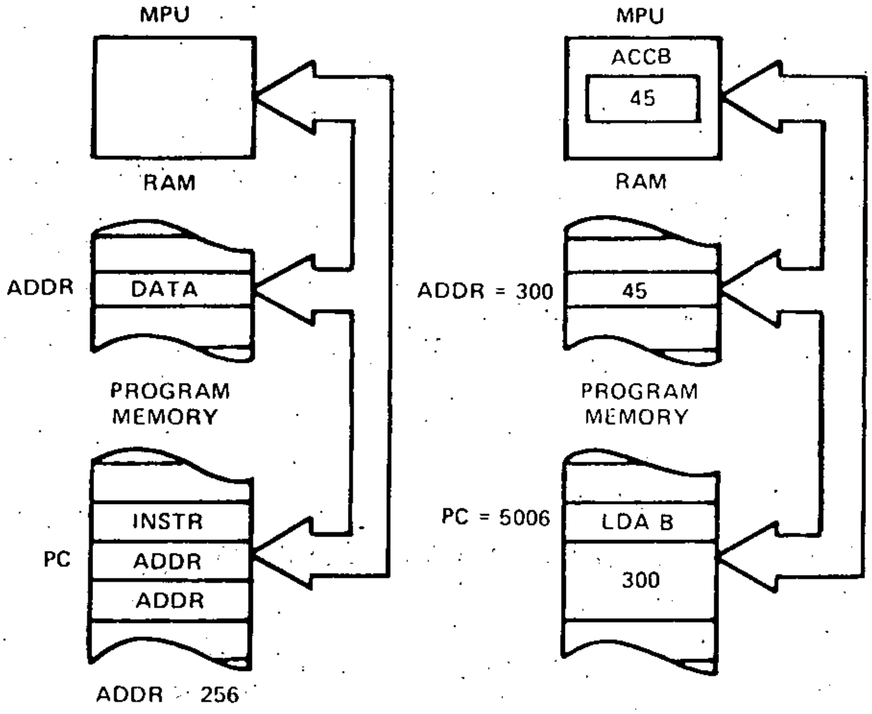
**DIRECT ADDRESSING**



**DIRECT ADDRESSING**

- Use for access to first 256 bytes
- 3-μs execution time (4 μs for store)

## EXTENDED ADDRESSING



GENERAL FLOW

EXAMPLE

## EXTENDED ADDRESSING

- Use for access to any location
- 4-μs execution time (5 μs for store)

### "Direccionamiento Indizado".

Con este modo de direccionamiento, la dirección numérica es variable; dependiendo en el contenido del registro de índice. La dirección actual es obtenida cada vez que es requerida durante la ejecución de un programa en vez de ser predeterminada como es en los otros modos de direccionamiento. El segundo byte de la instrucción contiene un valor numérico que, cuando es sumado al contenido del registro de índice durante la ejecución de un programa, proporciona la dirección numérica.

Como lo muestra la siguiente fórmula:

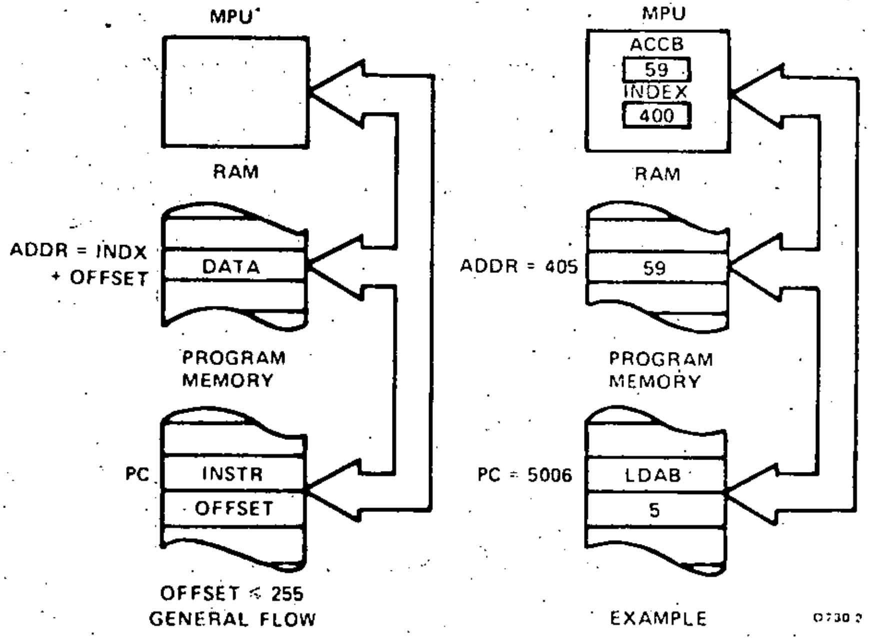
D= Valor numérico +X

DONDE:

X= Contenido del registro de índice

D= Dirección final

## INDEXED ADDRESSING



## INDEXED ADDRESSING

- For execution time address selection e.g., looping, subroutining, or table access
- Second byte of instruction has 8-bit (0-255) offset — reduces address maintenance in program
- 5-μs execution time (6 μs for store)

"Direccionamiento Relativo".

Para que el modo de direccionar relativo sea válido, hay una regla que limita la distancia en el programa en - lenguaje de máquina de una instrucción de bifurcación - - (branch) a la dirección de destino de esta bifurcación. La regla que se aplica al modo relativo de direccionamiento, es en la que el destino de la dirección de la derivación - debe de estar comprendida en el rango especificado por:

(PC+2)-128 (PC+2)+127.

PC= Dirección del primer byte de la instrucción.

D= Dirección del destino de esta instrucción.

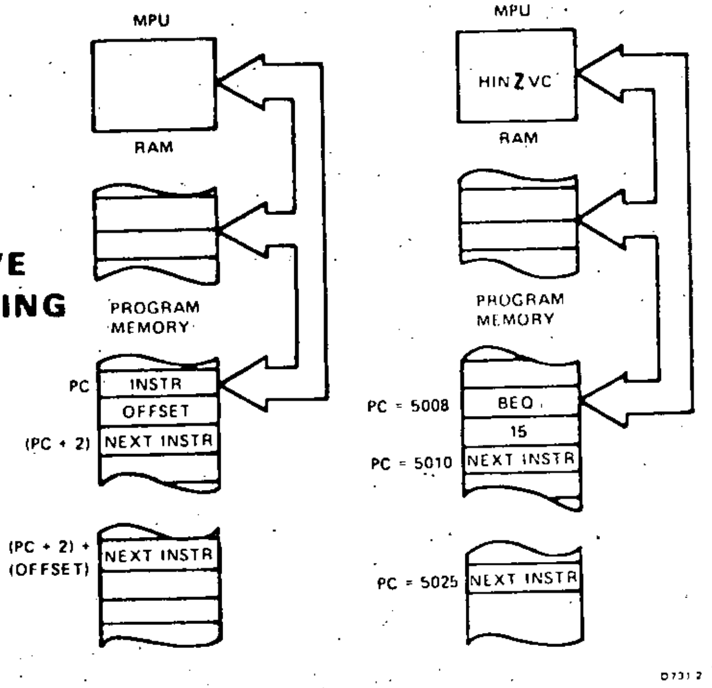
Cuando se desea transferir el control mas allá del -- rango que permite la instrucción de bifurcación, se puede usar la instrucción "JMP" (salto incondicional) o "JSR" - (salto a una subrutina) estas instrucciones no usan el modo relativo de direccionar.

Estas instrucciones se traducen en dos bytes de lenguaje de máquina, el segundo byte contiene la dirección relativa.

Como es deseable poder brincar en ambas direcciones -- (hacia arriba o hacia abajo de la instrucción). La dirección de 8 bits es interpretada como un valor de 7 bits con signo, el octavo bit es tratado como el bit del signo "0"= mas y "1"= menos el resto de los 7 bits representan el valor numérico.

Ejemplo:

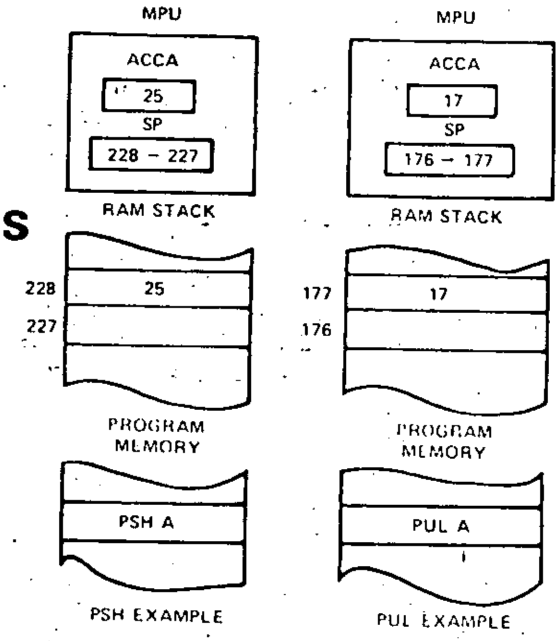
## RELATIVE ADDRESSING



## RELATIVE ADDRESSING

- For control transfers
- Second byte of instruction has 8-bit (–127—+127) offset
- Makes programs relocatable
- 4-μs execution time

## STACK OPERATIONS



## STACK OPERATIONS

- External stack provides LIFO storage in RAM
- System uses stack to store program return address for subroutine calls
- System uses stack to store all registers on interrupt

## DATA HANDLING INSTRUCTIONS
### (Data Movement)

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| LOAD ACMLTR | LDAA | $M \rightarrow A$ |
| | LDAB | $M \rightarrow B$ |
| PUSH DATA | PSHA | $A \rightarrow M_{SP}, SP - 1 \rightarrow SP$ |
| | PSHB | $B \rightarrow M_{SP}, SP - 1 \rightarrow SP$ |
| PULL DATA | PULA | $SP + 1 \rightarrow SP, M_{SP} \rightarrow A$ |
| | PULB | $SP + 1 \rightarrow SP, M_{SP} \rightarrow B$ |
| STORE ACMLTR | STAA | $A \rightarrow M$ |
| | STAB | $B \rightarrow M$ |
| TRANSFER ACMLTRS | TAB | $A \rightarrow B$ |
| | TBA | $B \rightarrow A$ |

## DATA MOVE

- Use all addressing modes

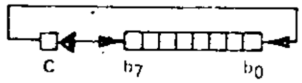
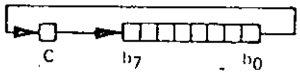
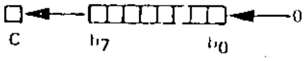
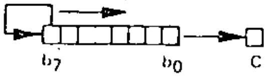
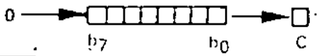
## DATA HANDLING INSTRUCTIONS (ALTER DATA)

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| CLEAR | CLR<br>CLRA<br>CLRB | 00 → M<br>00 → A<br>00 → B |
| DECREMENT | DEC<br>DECA<br>DECB | M − 1 → M<br>A − 1 → A<br>B − 1 → B |
| INCREMENT | INC<br>INCA<br>INCB | M + 1 → M<br>A + 1 → A<br>B + 1 → B |
| COMPLEMENT, 2'S (NEGATE) | NEG<br>NEGA<br>NEGB | 00 − M → M<br>00 − A → A<br>00 − B → B |
| COMPLEMENT, 1'S | COM<br>COMA<br>COMB | $\overline{M}$ → M<br>$\overline{A}$ → A<br>$\overline{B}$ → B |

## ALTER DATA

- In memory or accumulator
- Extended or indexed addressing available

## DATA HANDLING INSTRUCTIONS
### (SHIFT AND ROTATE)

| FUNCTION | MNEMONIC | | OPERATION |
|---|---|---|---|
| ROTATE LEFT | ROL | M |  |
| | ROLA | A | |
| | ROLB | B | |
| ROTATE RIGHT | ROR | M |  |
| | RORA | A | |
| | RORB | B | |
| SHIFT LEFT, ARITHMETIC | ASL | M |  |
| | ASLA | A | |
| | ASLB | B | |
| SHIFT RIGHT, ARITHMETIC | ASR | M |  |
| | ASRA | A | |
| | ASRB | B | |
| SHIFT RIGHT, LOGIC | LSR | M |  |
| | LSRA | A | |
| | LSRB | B | |

## DATA TEST INSTRUCTIONS

| FUNCTION | MNEMONIC | TEST |
|---|---|---|
| BIT TEST | BITA | A • M |
| | BIT B | B • M |
| COMPARE | CMPA | A – M |
| | CMPB | B – M |
| | CBA | A – B |
| TEST, ZERO OR MINUS | TST | M – 00 |
| | TSTA | A – 00 |
| | TSTB | B – 00 |

## DATA TESTS

- Set condition codes without modifying data
- Non-destructive test

### ARITHMETIC INSTRUCTIONS

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| ADD | ADDA<br>ADDB | A + M → A<br>B + M → B |
| ADD ACCUMULATORS | ABA | A + B → A |
| ADD WITH CARRY | ADCA<br>ADCB | A + M + C → A<br>B + M + C → B |
| COMPLEMENT, 2'S (NEGATE) | NEG<br>NEGA<br>NEGB | 00 – M → M<br>00 – A → A<br>00 – B → B |
| DECIMAL ADJUST, A | DAA | CONVERTS BINARY ADD. OF BCD CHARACTERS INTO BCD FORMAT |
| SUBTRACT | SUBA<br>SUBB | A – M → A<br>B – M → B |
| SUBTRACT ACCUMULATORS | SBA | A – B → A |
| SUBTRACT WITH CARRY | SBCA<br>SBCB | A – M – C → A<br>B – M – C → B |

## ARITHMETIC AND LOGICAL

- Dual operand instructions always have one operand in accumulator
- Immediate, direct, indexed, or extended addressing
- DAA for decimal arithmetic
- Complement or negate accumulator or memory

### LOGIC INSTRUCTIONS

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| AND | ANDA<br>ANDB | A • M → A<br>B • M → B |
| COMPLEMENT, 1'S | COM<br>COMA<br>COMB | $\overline{M}$ → M<br>$\overline{A}$ → A<br>$\overline{B}$ → B |
| EXCLUSIVE OR | EORA<br>EORB | A ⊕ M → A<br>B ⊕ M → B |
| OR, INCLUSIVE | ORA<br>ORB | A + M → A<br>B + M → B |

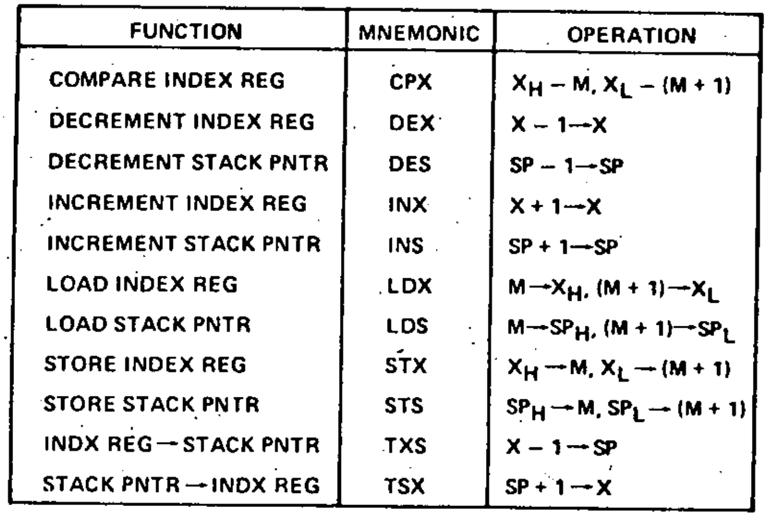
## CONDITION CODE REGISTER INSTRUCTIONS

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| CLEAR CARRY | CLC | 0 → C |
| CLEAR INTERRUPT MASK | CLI | 0 → I |
| CLEAR OVERFLOW | CLV | 0 → V |
| SET CARRY | SEC | 1 → C |
| SET INTERRUPT MASK | SEI | 1 → I |
| SET OVERFLOW | SEV | 1 → V |
| ACMLTR A → CCR | TAP | A → CCR |
| CCR → ACMLTR A | TPA | CCR → A |

## CONDITION CODES

- I, N, and V may be set and cleared with single-byte instructions
- Full set of CC.s may be transferred in and out of accumulator

## INDEX REGISTER AND STACK POINTER INSTRUCTIONS



| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| COMPARE INDEX REG | CPX | $X_H - M, X_L - (M + 1)$ |
| DECREMENT INDEX REG | DEX | $X - 1 \rightarrow X$ |
| DECREMENT STACK PNTR | DES | $SP - 1 \rightarrow SP$ |
| INCREMENT INDEX REG | INX | $X + 1 \rightarrow X$ |
| INCREMENT STACK PNTR | INS | $SP + 1 \rightarrow SP$ |
| LOAD INDEX REG | LDX | $M \rightarrow X_H, (M + 1) \rightarrow X_L$ |
| LOAD STACK PNTR | LDS | $M \rightarrow SP_H, (M + 1) \rightarrow SP_L$ |
| STORE INDEX REG | STX | $X_H \rightarrow M, X_L \rightarrow (M + 1)$ |
| STORE STACK PNTR | STS | $SP_H \rightarrow M, SP_L \rightarrow (M + 1)$ |
| INDX REG → STACK PNTR | TXS | $X - 1 \rightarrow SP$ |
| STACK PNTR → INDX REG | TSX | $SP + 1 \rightarrow X$ |

## JUMP AND BRANCH INSTRUCTIONS

| FUNCTION | MNEMONIC | BRANCH TEST |
|---|---|---|
| BRANCH ALWAYS | BRA | NONE |
| BRANCH IF CARRY CLEAR | BCC | C = 0 |
| BRANCH IF CARRY SET | BCS | C = 1 |
| BRANCH IF = ZERO | BEQ | Z = 1 |
| BRANCH IF ≥ ZERO | BGE | N ⊕ V = 0 |
| BRANCH IF > ZERO | BGT | Z + (N ⊕ V) = 0 |
| BRANCH IF HIGHER | BHI | C + Z = 1 |
| BRANCH IF ≤ ZERO | BLE | Z + (N ⊕ V) = 1 |
| BRANCH IF LOWER OR SAME | BLS | C + Z = 1 |
| BRANCH IF < ZERO | BLT | N ⊕ V = 1 |
| BRANCH IF MINUS | BMI | N = 1 |
| BRANCH IF NOT EQUAL ZERO | BNE | Z = 0 |

## CONTROL TRANSFERS

- Branch instructions use relative addressing
- Branches test conditions and complements, 2's complement and binary relations
- Branch to subroutine and jump to subroutine save return address (old PC value) in stack
- Return from subroutine restores PC from stack

## JUMP AND BRANCH INSTRUCTIONS

| FUNCTION | MNEMONIC | BRANCH TEST |
|---|---|---|
| BRANCH IF OVERFLOW CLEAR | BVC | V = 0 |
| BRANCH IF OVERFLOW SET | BVS | V = 1 |
| BRANCH IF PLUS | BPL | N = 0 |
| BRANCH TO SUBROUTINE | BSR | |
| JUMP | JMP | |
| JUMP TO SUBROUTINE | JSR | |
| NO OPERATION | NOP | ADVANCES PROG. CNTR. ONLY |
| RETURN FROM SUBROUTINE | RTS | |

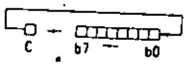
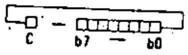
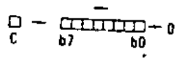
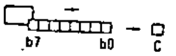
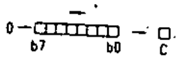
## INTERRUPT HANDLING INSTRUCTIONS

| FUNCTION | MNEMONIC | OPERATION |
|---|---|---|
| SOFTWARE INTERRUPT (TRAP) | SWI | REGS → $M_{SP}$<br>SP-7 → SP<br>$M_{FFFA}$ → PCH<br>$M_{FFFB}$ → PCL<br>1 → I |
| RETURN FROM INTERRUPT | RTI | $M_{SP}$ → REGS<br>SP+7 → SP |
| WAIT FOR INTERRUPT | WAI | REGS → $M_{SP}$<br>SP-7 → SP |

## INTERRUPT HANDLING

- SWI simulates interrupt for debugging and error handling
- RTI restores all registers
- WAI saves registers and halts

## TABLE 3 – ACCUMULATOR AND MEMORY INSTRUCTIONS

| OPERATIONS | MNEMONIC | ADDRESSING MODES | | | | | | | | | | | | | | | BOOLEAN/ARITHMETIC OPERATION | COND. CODE REG. | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | IMMED | | | DIRECT | | | INDEX | | | EXTND | | | IMPLIED | | | (All register labels refer to contents) | 5 | 4 | 3 | 2 | 1 | 0 |
| | | OP | ~ | = | OP | ~ | = | OP | ~ | = | OP | ~ | = | OP | ~ | = | | H | I | N | Z | V | C |
| Add | ADDA | 8B | 2 | 2 | 9B | 3 | 2 | AB | 5 | 2 | BB | 4 | 3 | | | | A + M → A | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| | ADDB | CB | 2 | 2 | DB | 3 | 2 | EB | 5 | 2 | FB | 4 | 3 | | | | B + M → B | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| Add Acmltrs | ABA | | | | | | | | | | | | | 1B | 2 | 1 | A + B → A | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| Add with Carry | ADCA | 89 | 2 | 2 | 99 | 3 | 2 | A9 | 5 | 2 | B9 | 4 | 3 | | | | A + M + C → A | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| | ADCB | C9 | 2 | 2 | D9 | 3 | 2 | E9 | 5 | 2 | F9 | 4 | 3 | | | | B + M + C → B | ↕ | ● | ↕ | ↕ | ↕ | ↕ |
| And | ANDA | 84 | 2 | 2 | 94 | 3 | 2 | A4 | 5 | 2 | B4 | 4 | 3 | | | | A · M → A | ● | ● | ↕ | ↕ | R | ● |
| | ANDB | C4 | 2 | 2 | D4 | 3 | 2 | E4 | 5 | 2 | F4 | 4 | 3 | | | | B · M → B | ● | ● | ↕ | ↕ | R | ● |
| Bit Test | BITA | 85 | 2 | 2 | 95 | 3 | 2 | A5 | 5 | 2 | B5 | 4 | 3 | | | | A · M | ● | ● | ↕ | ↕ | R | ● |
| | BITB | C5 | 2 | 2 | D5 | 3 | 2 | E5 | 5 | 2 | F5 | 4 | 3 | | | | B · M | ● | ● | ↕ | ↕ | R | ● |
| Clear | CLR | | | | | | | 6F | 7 | 2 | 7F | 6 | 3 | | | | 00 → M | ● | ● | R | S | R | R |
| | CLRA | | | | | | | | | | | | | 4F | 2 | 1 | 00 → A | ● | ● | R | S | R | R |
| | CLRB | | | | | | | | | | | | | 5F | 2 | 1 | 00 → B | ● | ● | R | S | R | R |
| Compare | CMPA | 81 | 2 | 2 | 91 | 3 | 2 | A1 | 5 | 2 | B1 | 4 | 3 | | | | A − M | ● | ● | ↕ | ↕ | ↕ | ↕ |
| | CMPB | C1 | 2 | 2 | D1 | 3 | 2 | E1 | 5 | 2 | F1 | 4 | 3 | | | | B − M | ● | ● | ↕ | ↕ | ↕ | ↕ |
| Compare Acmltrs | CBA | | | | | | | | | | | | | 11 | 2 | 1 | A − B | ● | ● | ↕ | ↕ | ↕ | ↕ |
| Complement, 1's | COM | | | | | | | 63 | 7 | 2 | 73 | 6 | 3 | | | | $\overline{M}$ → M | ● | ● | ↕ | ↕ | R | S |
| | COMA | | | | | | | | | | | | | 43 | 2 | 1 | $\overline{A}$ → A | ● | ● | ↕ | ↕ | R | S |
| | COMB | | | | | | | | | | | | | 53 | 2 | 1 | $\overline{B}$ → B | ● | ● | ↕ | ↕ | R | S |
| Complement, 2's (Negate) | NEG | | | | | | | 60 | 7 | 2 | 70 | 6 | 3 | | | | 00 − M → M | ● | ● | ↕ | ↕ | ① | ② |
| | NEGA | | | | | | | | | | | | | 40 | 2 | 1 | 00 − A → A | ● | ● | ↕ | ↕ | ① | ② |
| | NEGB | | | | | | | | | | | | | 50 | 2 | 1 | 00 − B → B | ● | ● | ↕ | ↕ | ① | ② |
| Decimal Adjust, A | DAA | | | | | | | | | | | | | 19 | 2 | 1 | Converts Binary Add. of BCD Characters into BCD Format. | ● | ● | ↕ | ↕ | ↕ | ③ |
| Decrement | DEC | | | | | | | 6A | 7 | 2 | 7A | 6 | 3 | | | | M − 1 → M | ● | ● | ↕ | ↕ | 4 | ● |
| | DECA | | | | | | | | | | | | | 4A | 2 | 1 | A − 1 → A | ● | ● | ↕ | ↕ | 4 | ● |
| | DECB | | | | | | | | | | | | | 5A | 2 | 1 | B − 1 → B | ● | ● | ↕ | ↕ | 4 | ● |
| Exclusive OR | EORA | 88 | 2 | 2 | 98 | 3 | 2 | A8 | 5 | 2 | B8 | 4 | 3 | | | | A⊙M → A | ● | ● | ↕ | ↕ | R | ● |
| | EORB | C8 | 2 | 2 | D8 | 3 | 2 | E8 | 5 | 2 | F8 | 4 | 3 | | | | B⊙M → B | ● | ● | ↕ | ↕ | R | ● |
| Increment | INC | | | | | | | 6C | 7 | 2 | 7C | 6 | 3 | | | | M + 1 → M | ● | ● | ↕ | ↕ | ⑤ | ● |
| | INCA | | | | | | | | | | | | | 4C | 2 | 1 | A + 1 → A | ● | ● | ↕ | ↕ | ⑤ | ● |
| | INCB | | | | | | | | | | | | | 5C | 2 | 1 | B + 1 → B | ● | ● | ↕ | ↕ | ⑤ | ● |
| Load Acmltr | LDAA | 86 | 2 | 2 | 96 | 3 | 2 | A6 | 5 | 2 | B6 | 4 | 3 | | | | M → A | ● | ● | ↕ | ↕ | R | ● |
| | LDAB | C6 | 2 | 2 | D6 | 3 | 2 | E6 | 5 | 2 | F6 | 4 | 3 | | | | M → B | ● | ● | ↕ | ↕ | R | ● |
| Or, Inclusive | ORAA | 8A | 2 | 2 | 9A | 3 | 2 | AA | 5 | 2 | BA | 4 | 3 | | | | A + M → A | ● | ● | ↕ | ↕ | R | ● |
| | ORAB | CA | 2 | 2 | DA | 3 | 2 | EA | 5 | 2 | FA | 4 | 3 | | | | B + M → B | ● | ● | ↕ | ↕ | R | ● |
| Push Data | PSHA | | | | | | | | | | | | | 36 | 4 | 1 | A → $M_{SP}$, SP − 1 → SP | ● | ● | ● | ● | ● | ● |
| | PSHB | | | | | | | | | | | | | 37 | 4 | 1 | B → $M_{SP}$, SP − 1 → SP | ● | ● | ● | ● | ● | ● |
| Pull Data | PULA | | | | | | | | | | | | | 32 | 4 | 1 | SP + 1 → SP, $M_{SP}$ → A | ● | ● | ● | ● | ● | ● |
| | PULB | | | | | | | | | | | | | 33 | 4 | 1 | SP + 1 → SP, $M_{SP}$ → B | ● | ● | ● | ● | ● | ● |
| Rotate Left | ROL | | | | | | | 69 | 7 | 2 | 79 | 6 | 3 | | | | M  | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ROLA | | | | | | | | | | | | | 49 | 2 | 1 | A | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ROLB | | | | | | | | | | | | | 59 | 2 | 1 | B | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| Rotate Right | ROR | | | | | | | 66 | 7 | 2 | 76 | 6 | 3 | | | | M  | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | RORA | | | | | | | | | | | | | 46 | 2 | 1 | A | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | RORB | | | | | | | | | | | | | 56 | 2 | 1 | B | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| Shift Left, Arithmetic | ASL | | | | | | | 68 | 7 | 2 | 78 | 6 | 3 | | | | M  | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ASLA | | | | | | | | | | | | | 48 | 2 | 1 | A | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ASLB | | | | | | | | | | | | | 58 | 2 | 1 | B | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| Shift Right, Arithmetic | ASR | | | | | | | 67 | 7 | 2 | 77 | 6 | 3 | | | | M  | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ASRA | | | | | | | | | | | | | 47 | 2 | 1 | A | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| | ASRB | | | | | | | | | | | | | 57 | 2 | 1 | B | ● | ● | ↕ | ↕ | ⑥ | ↕ |
| Shift Right, Logic | LSR | | | | | | | 64 | 7 | 2 | 74 | 6 | 3 | | | | M  | ● | ● | R | ↕ | ⑥ | ↕ |
| | LSRA | | | | | | | | | | | | | 44 | 2 | 1 | A | ● | ● | R | ↕ | ⑥ | ↕ |
| | LSRB | | | | | | | | | | | | | 54 | 2 | 1 | B | ● | ● | R | ↕ | ⑥ | ↕ |
| Store Acmltr. | STAA | | | | 97 | 4 | 2 | A7 | 6 | 2 | B7 | 5 | 3 | | | | A → M | ● | ● | ↕ | ↕ | R | ● |
| | STAB | | | | D7 | 4 | 2 | E7 | 6 | 2 | F7 | 5 | 3 | | | | B → M | ● | ● | ↕ | ↕ | R | ● |
| Subtract | SUBA | 80 | 2 | 2 | 90 | 3 | 2 | A0 | 5 | 2 | B0 | 4 | 3 | | | | A − M → A | ● | ● | ↕ | ↕ | ↕ | ↕ |
| | SUBB | C0 | 2 | 2 | D0 | 3 | 2 | E0 | 5 | 2 | F0 | 4 | 3 | | | | B − M → B | ● | ● | ↕ | ↕ | ↕ | ↕ |
| Subtract Acmltrs. | SBA | | | | | | | | | | | | | 10 | 2 | 1 | A − B → A | ● | ● | ↕ | ↕ | ↕ | ↕ |
| Subtr. with Carry | SBCA | 82 | 2 | 2 | 92 | 3 | 2 | A2 | 5 | 2 | B2 | 4 | 3 | | | | A − M − C → A | ● | ● | ↕ | ↕ | ↕ | ↕ |
| | SBCB | C2 | 2 | 2 | D2 | 3 | 2 | E2 | 5 | 2 | F2 | 4 | 3 | | | | B − M − C → B | ● | ● | ↕ | ↕ | ↕ | ↕ |
| Transfer Acmltrs | TAB | | | | | | | | | | | | | 16 | 2 | 1 | A → B | ● | ● | ↕ | ↕ | R | ● |
| | TBA | | | | | | | | | | | | | 17 | 2 | 1 | B → A | ● | ● | ↕ | ↕ | R | ● |
| Test, Zero or Minus | TST | | | | | | | 6D | 7 | 2 | 7D | 6 | 3 | | | | M − 00 | ● | ● | ↕ | ↕ | R | R |
| | TSTA | | | | | | | | | | | | | 4D | 2 | 1 | A − 00 | ● | ● | ↕ | ↕ | R | R |
| | TSTB | | | | | | | | | | | | | 5D | 2 | 1 | B − 00 | ● | ● | ↕ | ↕ | R | R |
| | | | | | | | | | | | | | | | | | | H | I | N | Z | V | C |

LEGEND:

| | |
|---|---|
| OP | Operation Code (Hexadecimal); |
| ~ | Number of MPU Cycles; |
| = | Number of Program Bytes; |
| + | Arithmetic Plus; |
| − | Arithmetic Minus; |
| · | Boolean AND; |
| $M_{SP}$ | Contents of memory location pointed to be Stack Pointer; |
| + | Boolean Inclusive OR; |
| ⊙ | Boolean Exclusive OR; |
| $\overline{M}$ | Complement of M; |
| → | Transfer Into; |
| 0 | Bit = Zero; |
| 00 | Byte = Zero; |

Note – Accumulator addressing mode instructions are included in the column for IMPLIED addressing

CONDITION CODE SYMBOLS:

| | |
|---|---|
| H | Half-carry from bit 3; |
| I | Interrupt mask |
| N | Negative (sign bit) |
| Z | Zero (byte) |
| V | Overflow, 2's complement |
| C | Carry from bit 7 |
| R | Reset Always |
| S | Set Always |
| ↕ | Test and set if true, cleared otherwise |
| ● | Not Affected |

### TABLE 4 – INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

| | | IMMED | | | DIRECT | | | INDEX | | | EXTND | | | IMPLIED | | | | COND. CODE REG. | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 |
| POINTER OPERATIONS | MNEMONIC | OP | ~ | # | OP | ~ | # | OP | ~ | # | OP | ~ | # | OP | ~ | # | BOOLEAN/ARITHMETIC OPERATION | H | I | N | Z | V | C |
| Compare Index Reg | CPX | 8C | 3 | 3 | 9C | 4 | 2 | AC | 6 | 2 | BC | 5 | 3 | | | | $X_H - M, X_L - (M+1)$ | ● | ● | ⑦ | ↕ | ⑧ | ● |
| Decrement Index Reg | DEX | | | | | | | | | | | | | 09 | 4 | 1 | X – 1 → X | ● | ● | ● | ↕ | ● | ● |
| Decrement Stack Pntr | DES | | | | | | | | | | | | | 34 | 4 | 1 | SP – 1 → SP | ● | ● | ● | ● | ● | ● |
| Increment Index Reg | INX | | | | | | | | | | | | | 08 | 4 | 1 | X + 1 → X | ● | ● | ● | ↕ | ● | ● |
| Increment Stack Pntr | INS | | | | | | | | | | | | | 31 | 4 | 1 | SP + 1 → SP | ● | ● | ● | ● | ● | ● |
| Load Index Reg | LDX | CE | 3 | 3 | DE | 4 | 2 | EE | 6 | 2 | FE | 5 | 3 | | | | $M \rightarrow X_H, (M+1) \rightarrow X_L$ | ● | ● | ⑨ | ↕ | R | ● |
| Load Stack Pntr | LDS | 8E | 3 | 3 | 9E | 4 | 2 | AE | 6 | 2 | BE | 5 | 3 | | | | $M \rightarrow SP_H, (M+1) \rightarrow SP_L$ | ● | ● | ⑨ | ↕ | R | ● |
| Store Index Reg | STX | | | | DF | 5 | 2 | EF | 7 | 2 | FF | 6 | 3 | | | | $X_H \rightarrow M, X_L \rightarrow (M+1)$ | ● | ● | ⑨ | ↕ | R | ● |
| Store Stack Pntr | STS | | | | 9F | 5 | 2 | AF | 7 | 2 | BF | 6 | 3 | | | | $SP_H \rightarrow M, SP_L \rightarrow (M+1)$ | ● | ● | ⑨ | ↕ | R | ● |
| Indx Reg → Stack Pntr | TXS | | | | | | | | | | | | | 35 | 4 | 1 | X – 1 → SP | ● | ● | ● | ● | ● | ● |
| Stack Pntr → Indx Reg | TSX | | | | | | | | | | | | | 30 | 4 | 1 | SP + 1 → X | ● | ● | ● | ● | ● | ● |

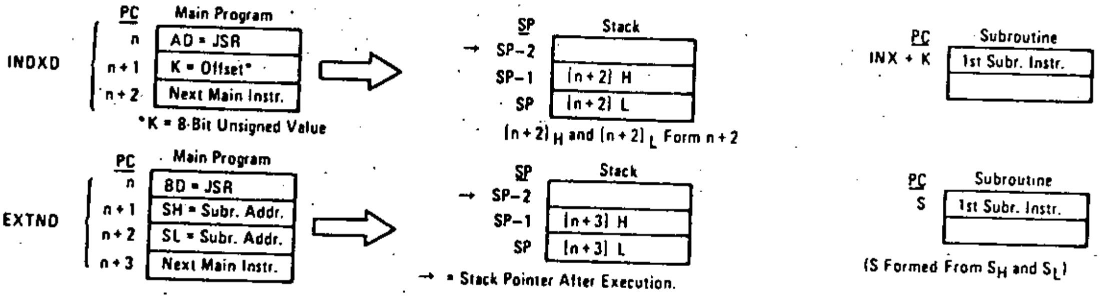
### TABLE 5 – JUMP AND BRANCH INSTRUCTIONS

| | | RELATIVE | | | INDEX | | | EXTND | | | IMPLIED | | | | COND. CODE REG. | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 |
| OPERATIONS | MNEMONIC | OP | ~ | # | OP | ~ | # | OP | ~ | # | OP | ~ | # | BRANCH TEST | H | I | N | Z | V | C |
| Branch Always | BRA | 20 | 4 | 2 | | | | | | | | | | None | ● | ● | ● | ● | ● | ● |
| Branch If Carry Clear | BCC | 24 | 4 | 2 | | | | | | | | | | C = 0 | ● | ● | ● | ● | ● | ● |
| Branch If Carry Set | BCS | 25 | 4 | 2 | | | | | | | | | | C = 1 | ● | ● | ● | ● | ● | ● |
| Branch If = Zero | BEQ | 27 | 4 | 2 | | | | | | | | | | Z = 1 | ● | ● | ● | ● | ● | ● |
| Branch If ≥ Zero | BGE | 2C | 4 | 2 | | | | | | | | | | N ⊕ V = 0 | ● | ● | ● | ● | ● | ● |
| Branch If > Zero | BGT | 2E | 4 | 2 | | | | | | | | | | Z + (N ⊕ V) = 0 | ● | ● | ● | ● | ● | ● |
| Branch If Higher | BHI | 22 | 4 | 2 | | | | | | | | | | C + Z = 0 | ● | ● | ● | ● | ● | ● |
| Branch If ≤ Zero | BLE | 2F | 4 | 2 | | | | | | | | | | Z + (N ⊕ V) = 1 | ● | ● | ● | ● | ● | ● |
| Branch If Lower Or Same | BLS | 23 | 4 | 2 | | | | | | | | | | C + Z = 1 | ● | ● | ● | ● | ● | ● |
| Branch If < Zero | BLT | 2D | 4 | 2 | | | | | | | | | | N ⊕ V = 1 | ● | ● | ● | ● | ● | ● |
| Branch If Minus | BMI | 2B | 4 | 2 | | | | | | | | | | N = 1 | ● | ● | ● | ● | ● | ● |
| Branch If Not Equal Zero | BNE | 26 | 4 | 2 | | | | | | | | | | Z = 0 | ● | ● | ● | ● | ● | ● |
| Branch If Overflow Clear | BVC | 28 | 4 | 2 | | | | | | | | | | V = 0 | ● | ● | ● | ● | ● | ● |
| Branch If Overflow Set | BVS | 29 | 4 | 2 | | | | | | | | | | V = 1 | ● | ● | ● | ● | ● | ● |
| Branch If Plus | BPL | 2A | 4 | 2 | | | | | | | | | | N = 0 | ● | ● | ● | ● | ● | ● |
| Branch To Subroutine | BSR | 8D | 8 | 2 | | | | | | | | | | See Special Operations | ● | ● | ● | ● | ● | ● |
| Jump | JMP | | | | 6E | 4 | 2 | 7E | 3 | 3 | | | | | ● | ● | ● | ● | ● | ● |
| Jump To Subroutine | JSR | | | | AD | 8 | 2 | BD | 9 | 3 | | | | | ● | ● | ● | ● | ● | ● |
| No Operation | NOP | | | | | | | | | | 01 | 2 | 1 | Advances Prog. Cntr. Only | ● | ● | ● | ● | ● | ● |
| Return From Interrupt | RTI | | | | | | | | | | 3B | 10 | 1 | | ⑩ | | | | | |
| Return From Subroutine | RTS | | | | | | | | | | 39 | 5 | 1 | See Special Operations | ● | ● | ● | ● | ● | ● |
| Software Interrupt | SWI | | | | | | | | | | 3F | 12 | 1 | | ● | ● | ● | ● | ● | ● |
| Wait for Interrupt* | WAI | | | | | | | | | | 3E | 9 | 1 | | ● | ⑪ | ● | ● | ● | ● |

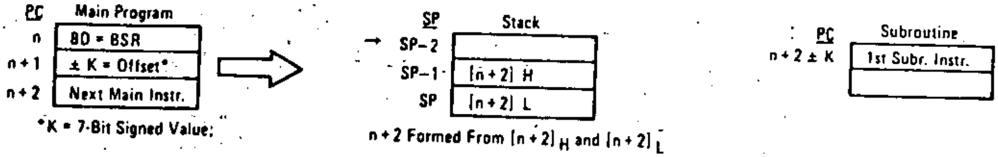
*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

## SPECIAL OPERATIONS

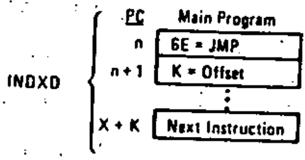
### JSR, JUMP TO SUBROUTINE:

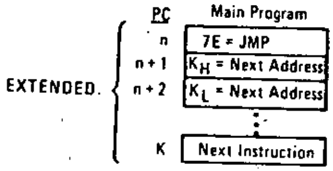


### BSR, BRANCH TO SUBROUTINE:

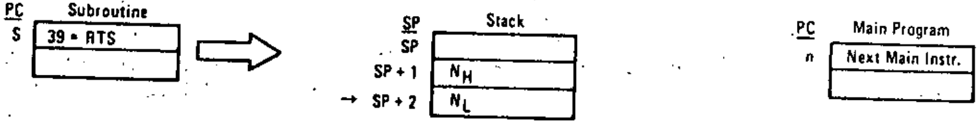


### JMP, JUMP:

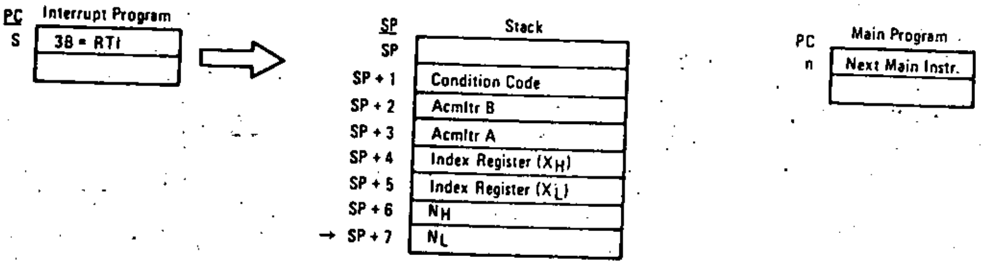




### RTS, RETURN FROM SUBROUTINE:



### RTI, RETURN FROM INTERRUPT:



### TABLE 6 – CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

| | | IMPLIED | | | | COND. CODE REG. | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 5 | 4 | 3 | 2 | 1 | 0 |
| OPERATIONS | MNEMONIC | OP | ~ | # | BOOLEAN OPERATION | H | I | N | Z | V | C |
| Clear Carry | CLC | 0C | 2 | 1 | 0 → C | ● | ● | ● | ● | ● | R |
| Clear Interrupt Mask | CLI | 0E | 2 | 1 | 0 → I | ● | R | ● | ● | ● | ● |
| Clear Overflow | CLV | 0A | 2 | 1 | 0 → V | ● | ● | ● | ● | R | ● |
| Set Carry | SEC | 0D | 2 | 1 | 1 → C | ● | ● | ● | ● | ● | S |
| Set Interrupt Mask | SEI | 0F | 2 | 1 | 1 → I | ● | S | ● | ● | ● | ● |
| Set Overflow | SEV | 0B | 2 | 1 | 1 → V | ● | ● | ● | ● | S | ● |
| Acmltr A → CCR | TAP | 06 | 2 | 1 | A → CCR | 12 | | | | | |
| CCR → Acmltr A | TPA | 07 | 2 | 1 | CCR → A | ● | ● | ● | ● | ● | ● |

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

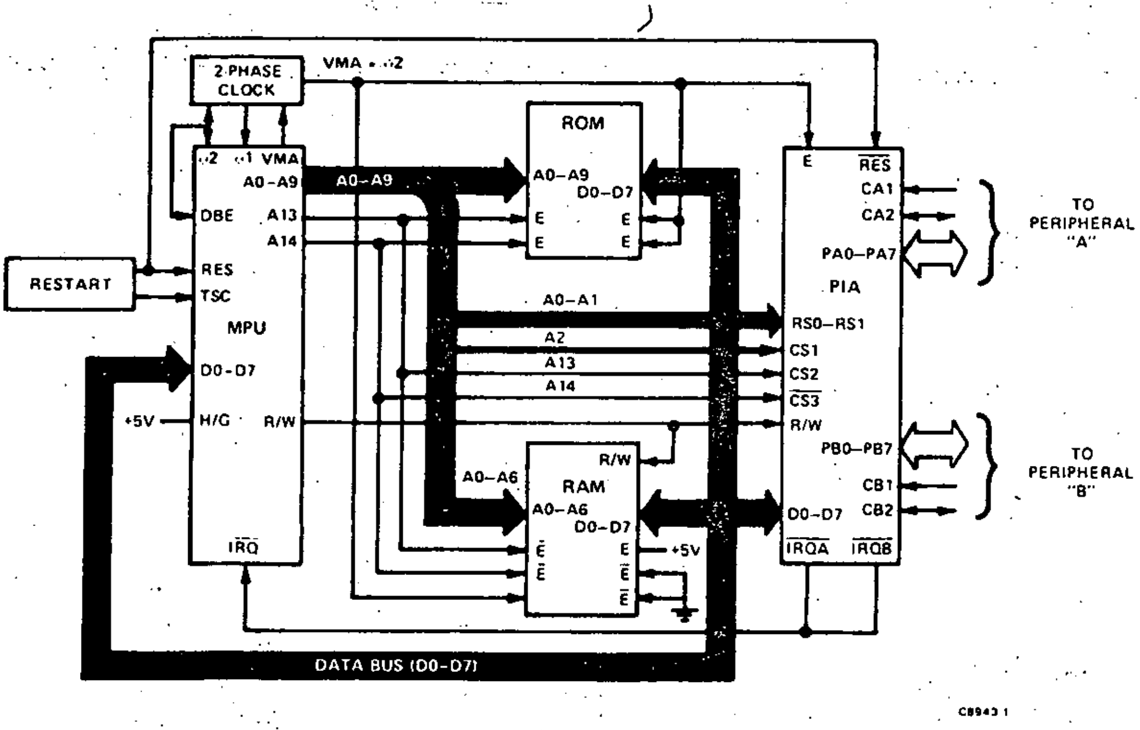
1 (Bit V) Test: Result = 10000000?
2 (Bit C) Test: Result = 00000000?
3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
4 (Bit V) Test: Operand = 10000000 prior to execution?
5 (Bit V) Test: Operand = 01111111 prior to execution?
6 (Bit V) Test: Set equal to result of N⊕C after shift has occurred.
7 (Bit N) Test: Sign bit of most significant (MS) byte = 1?
8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes?
9 (Bit N) Test: Result less than zero? (Bit 15 = 1)
10 (All) Load Condition Code Register from Stack. (See Special Operations)
11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
12 (All) Set according to the contents of Accumulator A.

## SISTEMA MINIMO:

La Fig. muestra el CPU 6800 manejando un pequeño "ROM" de 1K X 8, una memoria tipo RAM de 128X8 y una PIA (Peripheral Interface Adapter) que contiene 2 puertos bidireccionales), siendo todos estos chips, (N MOS LSI), - miembros de la familia 6800, pudiendo todos ser operados - con una sola fuente de alimentación de +5V. El procesador requiere un reloj relativamente simple, de dos fases $\emptyset_1$, - $\emptyset$2. (NON-OVERLAPING), que puede ser adquirido en forma híbrida para fácil aplicación con salida $\emptyset_1$ y $\emptyset_2$ para NMOS y TTL, con una frecuencia mínima de 100KHz y una máxima de - 1 MHz que son las frecuencias mínimas y máximas de operación del CPU hoy en día, o se puede hacer con compuertas - rápidas.

El 6800 desde luego usa las líneas de dirección y de datos separadas, norma que se está siguiendo en todas las arquitecturas de la 2a. generación, por consiguiente, como en el 8080, no necesita registros externos (Latches), para retener la dirección, pero el 6800 va más allá del 8080 en simplicidad, puesto que no necesita tampoco registros externos para retener la información del STATUS.

La manera en que el 6800 trata todo lo externo como - memoria sigue la arquitectura de la PDP-11 nótese que no - existe ningún comando de entrada/salida en las instrucciones del 6800 como se puede encontrar en el 8080.


2 PHASE CLOCK
VMA • φ2
φ2
φ1
VMA
A0–A9
DBE
A13
A14
RES
TSC
MPU
D0–D7
+5V
H/G
R/W
IRQ
RESTART
A0–A9
ROM
A0–A9
D0–D7
E
E
E
E
A0–A1
A2
A13
A14
A0–A6
R/W
RAM
A0–A6
D0–D7
Ē
Ē
E
+5V
Ē
Ē
E
RES
CA1
CA2
PA0–PA7
PIA
RS0–RS1
CS1
CS2
CS3
R/W
PB0–PB7
CB1
CB2
D0–D7
IRQA
IRQB
TO PERIPHERAL "A"
TO PERIPHERAL "B"
DATA BUS (D0–D7)
C8943 1

En la Figura anterior se puede observar cómo este sistema de memoria y E/S puede ser direccionado. Se indica cómo las 16 líneas del BUS de direcciones son repartidas con los circuitos externos.

En este sistema mínimo los Bits de dirección $A_{10}$, $A_{11}$, $A_{12}$ y $A_{15}$ no son ni siquiera usado, por lo tanto, estas líneas están a la disposición del diseñador para que las use como mejor le convenga. Los Bits $A_{13}$ y $A_{14}$ seleccionan los tres circuitos externos.

## "DECODIFICACION DE SEÑALES"

| Circuito<br>Externo | Dirección | Øz | R/W | VMA | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ROM | 6000 a<br>63FF | 1 | 1 | 1 |  | 1 | 1 |  |  |  | X | X | X | X | X | X | X | X | X | X |
| PIA | 2004 a<br>2007 | 1 | X | 1 |  | 0 | 1 |  |  |  |  |  |  |  |  |  |  | 1 | X | X |
| RAM | 0000 a<br>007F | 1 | X | 1 | 0 | 0 | 0 |  |  |  |  |  |  | X | X | X | X | X | X | X |

ROM: para direccionar 1024 bytes de esta memoria se necesitan - 10 líneas de dirección que nos dan $2^{10}$ =1024 y la seleccionamos a través de los Chip Select con las líneas $A_{13}$ y $A_{14}$ cuando son 1, junto con la $\emptyset_2$ y las líneas de control R/W y VMA.

El ROM debe de ponerse en la parte más alta de la dirección porque el sistema de inicialización (RESTART) está internamente conectado para "mirar" las localidades en el - espacio de memoria que contengan sólo unos, ésto significa que el procesador en este comando de inicialización pone en el bus de dirección unos y de esta forma, direcciona la localidad más alta, ahí el 6800 espera encontrar la dirección para el punto de partida del programa de aplicación, (pone en esta dirección en el apuntador de programa).

En la arquitectura del 6800, la memoria RAM debe ponerse en donde pueda ser alcanzada por los 8 bits más bajos - del BUS de 16 bits de dirección.

De esta manera las localidades de la memoria RAM pue-- den ser alcanzadas eficientemente por el modo corto y rápido del direccionamiento directo que puede direccionar las - primeras 256 localidades de memoria. ($2^8$ =256) por lo tanto es imprescindible poner RAM en la parte más baja de memoria.

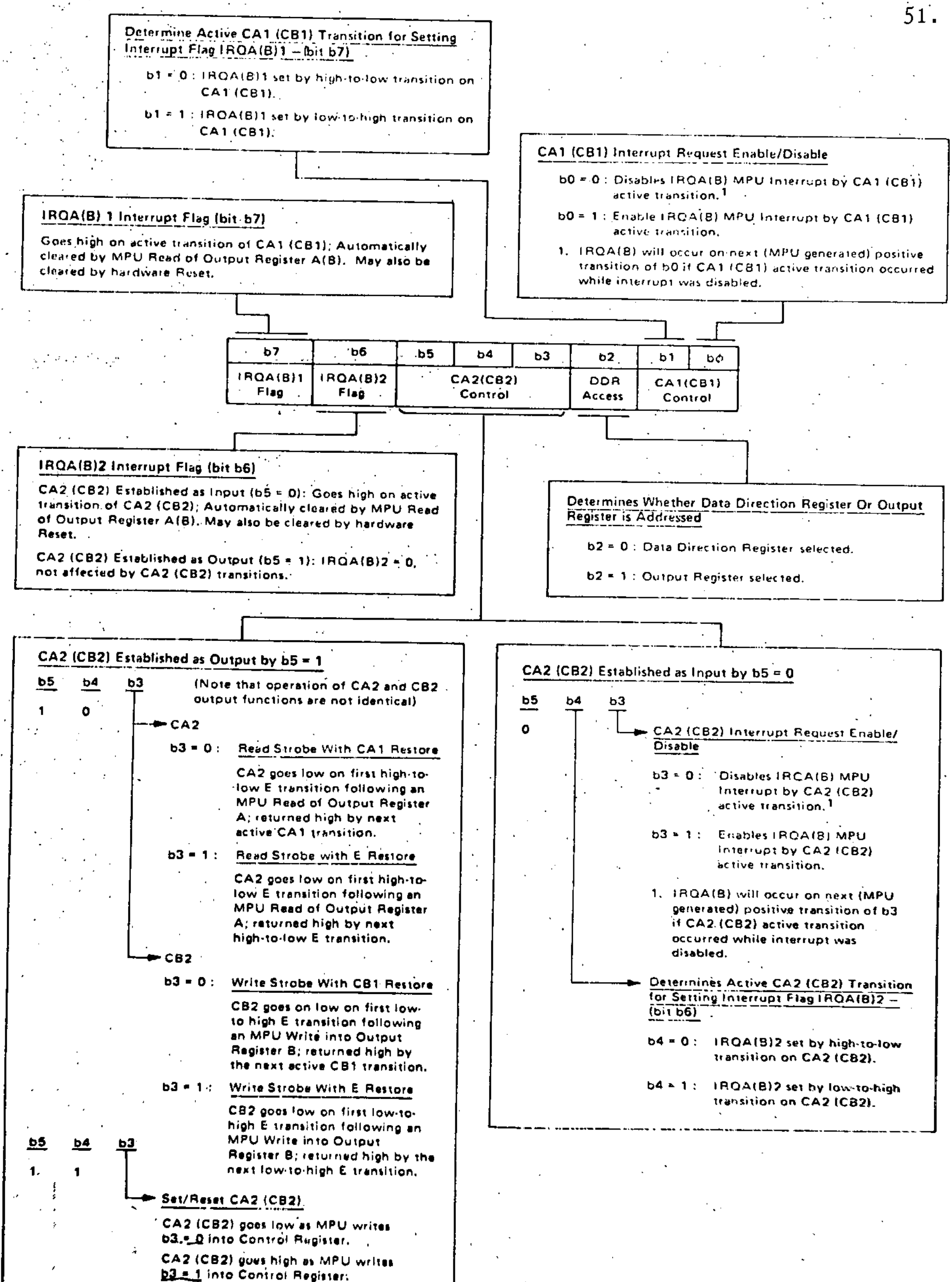
Por consiguiente a la RAM se le asigna el par $A_{13}$ y - $A_{14}$ como 0. En la figura, es implementada al conectar estas líneas de dirección a los $\overline{CS}$, motorola ha dado a estos subsistemas suficiente lógica en las múltiples entradas "CS" que le permiten decodificar las direcciones de una manera fácil y económica.

Para direccionar los 128 bytes se necesitan 7 líneas de dirección y la seleccionamos a través de la $A_{13}$ y $A_{14}$ - como se dijo anteriormente y desde luego con las líneas de control R/W, $\emptyset_2$ y VMA.

El "PIA" o adaptador periférico bidireccional lo podemos poner en cualquier lugar del mapa entre la RAM y el ROM.

Nótese que los bits de dirección más bajos conectados a estos Chips son únicamente los necesarios para seleccionar los registros internos de cada Chip. Hay 7 líneas para los 128 registros de la RAM, 10 para las 1024 localidades del - ROM y dos para los cuatro del PIA. Aunque el PIA tiene 6 registros internos, motorola le da la vuelta al darle a dos de los registros, registros de contro, un bit que localmente programa aquél par de registros restantes.

Probablemente se puede ver por qué el PIA es un Chip - LSI y por qué es aproximadamente un 25% tan complejo como - el CPU.



**Determine Active CA1 (CB1) Transition for Setting Interrupt Flag IRQA(B)1 – (bit b7)**

b1 = 0 : IRQA(B)1 set by high-to-low transition on CA1 (CB1).

b1 = 1 : IRQA(B)1 set by low-to-high transition on CA1 (CB1).

**CA1 (CB1) Interrupt Request Enable/Disable**

b0 = 0 : Disables IRQA(B) MPU Interrupt by CA1 (CB1) active transition.[1]

b0 = 1 : Enable IRQA(B) MPU Interrupt by CA1 (CB1) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b0 if CA1 (CB1) active transition occurred while interrupt was disabled.

**IRQA(B) 1 Interrupt Flag (bit b7)**

Goes high on active transition of CA1 (CB1); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

| b7 | b6 | b5 | b4 | b3 | b2 | b1 | bφ |
|---|---|---|---|---|---|---|---|
| IRQA(B)1 Flag | IRQA(B)2 Flag | CA2(CB2) Control | | | DDR Access | CA1(CB1) Control | |

**IRQA(B)2 Interrupt Flag (bit b6)**

CA2 (CB2) Established as Input (b5 = 0): Goes high on active transition of CA2 (CB2); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

CA2 (CB2) Established as Output (b5 = 1): IRQA(B)2 = 0, not affected by CA2 (CB2) transitions.

**Determines Whether Data Direction Register Or Output Register is Addressed**

b2 = 0 : Data Direction Register selected.

b2 = 1 : Output Register selected.

**CA2 (CB2) Established as Output by b5 = 1**

b5 b4 b3

1 0

(Note that operation of CA2 and CB2 output functions are not identical)

→ CA2

b3 = 0 : Read Strobe With CA1 Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next active CA1 transition.

b3 = 1 : Read Strobe with E Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next high-to-low E transition.

→ CB2

b3 = 0 : Write Strobe With CB1 Restore

CB2 goes on low on first low-to high E transition following an MPU Write into Output Register B; returned high by the next active CB1 transition.

b3 = 1 : Write Strobe With E Restore

CB2 goes low on first low-to-high E transition following an MPU Write into Output Register B; returned high by the next low-to-high E transition.

b5 b4 b3

1 1

→ Set/Reset CA2 (CB2)

CA2 (CB2) goes low as MPU writes b3 = 0 into Control Register.

CA2 (CB2) goes high as MPU writes b3 = 1 into Control Register.

**CA2 (CB2) Established as Input by b5 = 0**

b5 b4 b3

0

→ CA2 (CB2) Interrupt Request Enable/Disable

b3 = 0 : Disables IRCA(B) MPU Interrupt by CA2 (CB2) active transition.[1]

b3 = 1 : Enables IRQA(B) MPU Interrupt by CA2 (CB2) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b3 if CA2 (CB2) active transition occurred while interrupt was disabled.

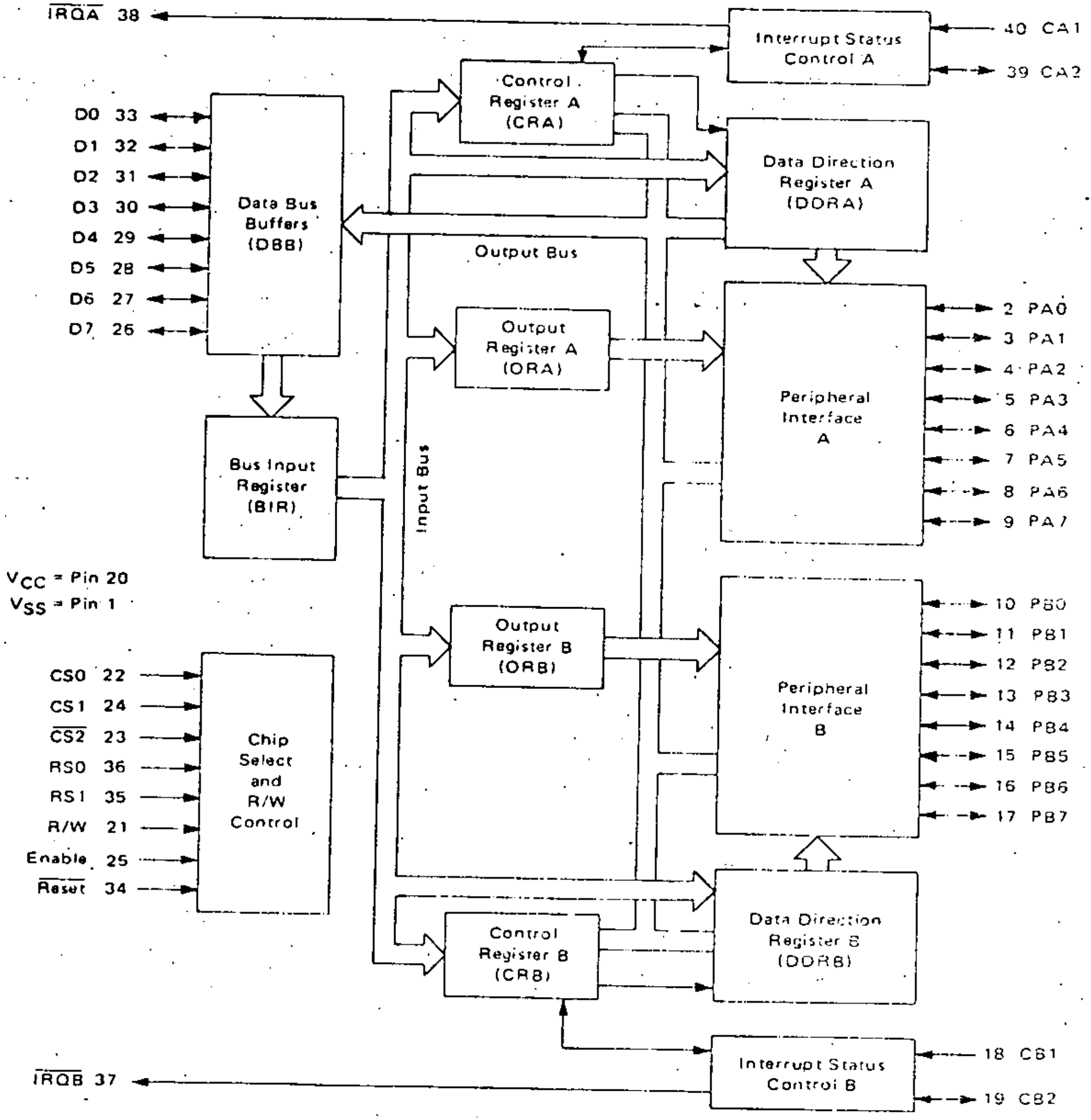
→ Determines Active CA2 (CB2) Transition for Setting Interrupt Flag IRQA(B)2 – (bit b6)

b4 = 0 : IRQA(B)2 set by high-to-low transition on CA2 (CB2).

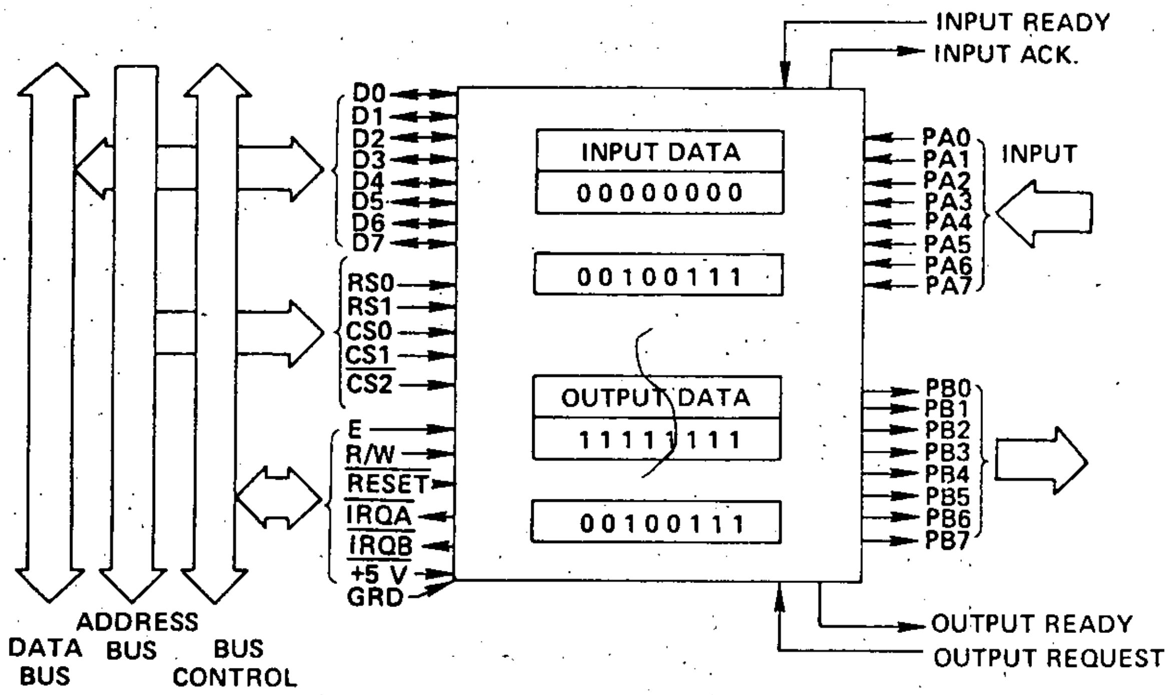
b4 = 1 : IRQA(B)2 set by low-to-high transition on CA2 (CB2).

FIGURE 2-4. PIA Control Register Format

EXPANDED BLOCK DIAGRAM



## PIA APPLICATION EXAMPLE

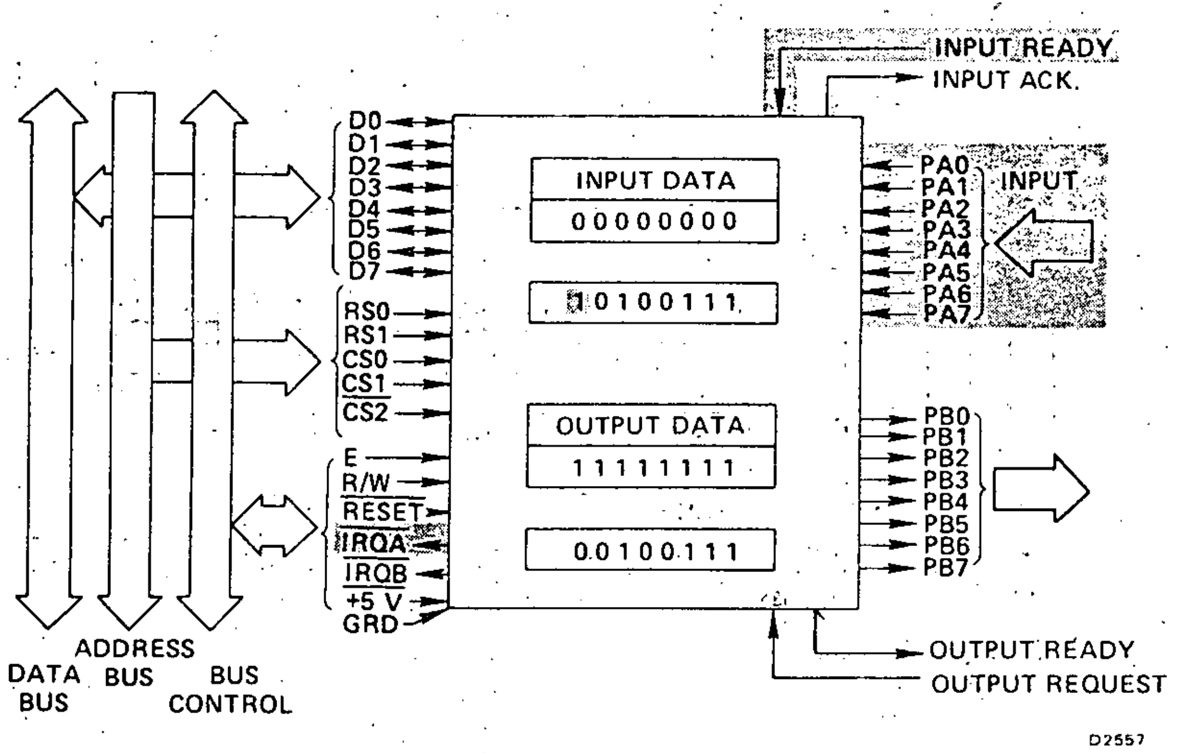


D2559

### I/O EXAMPLE

- PIA used as full duplex, parallel interface
- Full handshake controls for input and output
- "INPUT READY" and "OUTPUT REQUEST" generate interrupts
- "INPUT ACK" and "OUTPUT READY" are automatically generated
- These characteristics are established by storing patterns shown in data direction and control registers

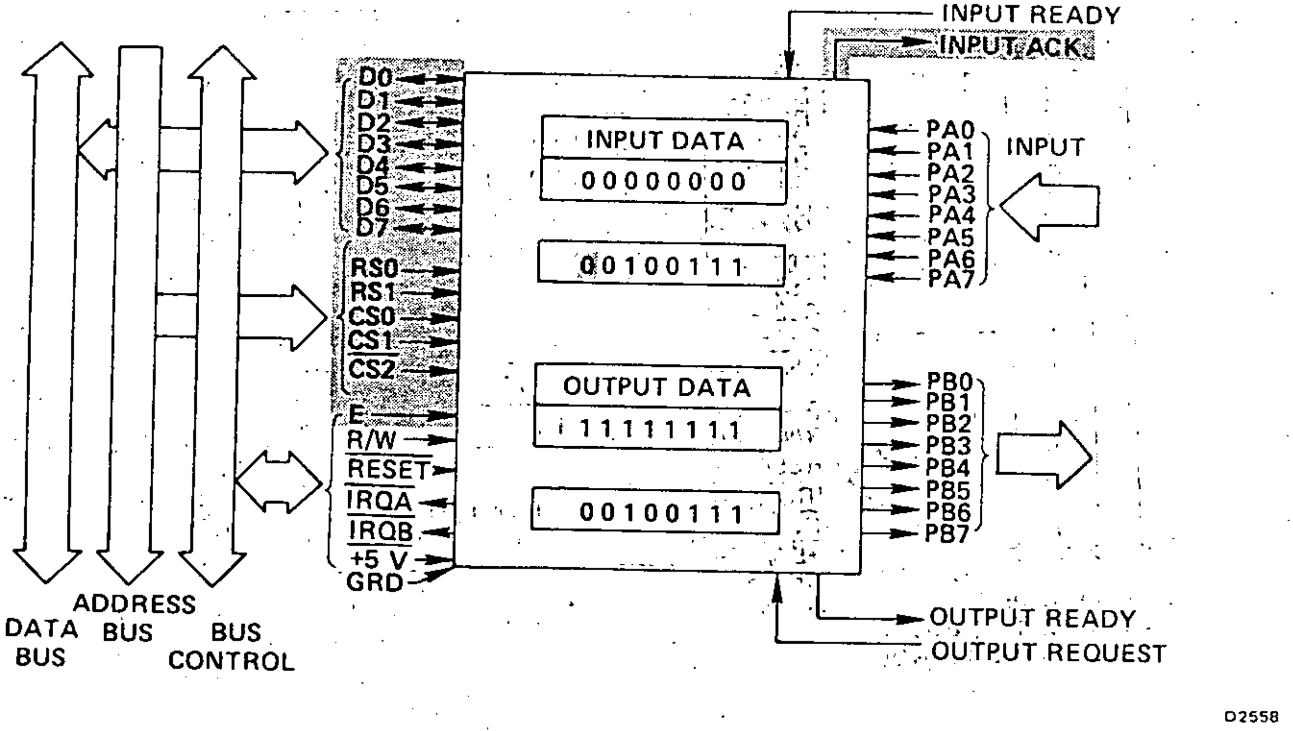
## PIA APPLICATION EXAMPLE



**EXTERNAL DEVICE PRESENTS INPUT DATA AND "INPUT READY" SIGNAL**

- "INPUT READY" transition sets status bit and pulls down $\overline{\text{IRQ}}$
- Interrupt response routine will identify this interrupt by checking status bits

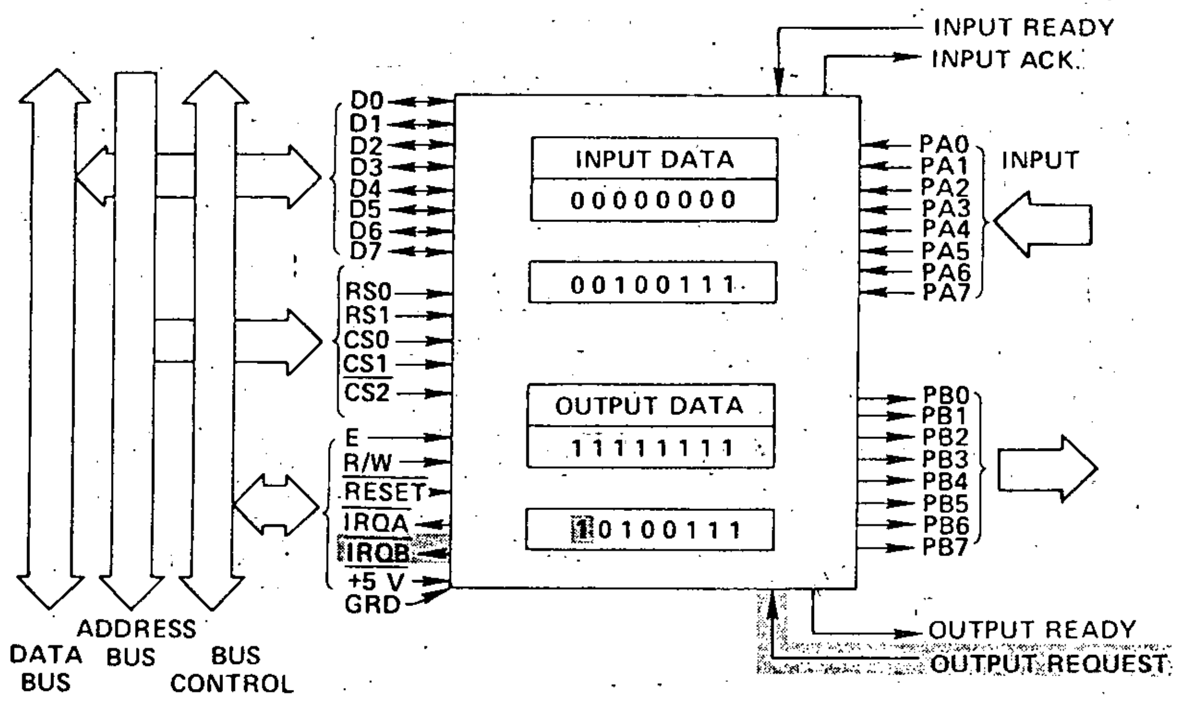
## PIA APPLICATION EXAMPLE



## INTERRUPT RESPONSE ROUTINE READS INPUT DATA

- Interrupt is cleared by the read operation
- "INPUT ACK" signal generated automatically after data is read

## PIA APPLICATION EXAMPLE

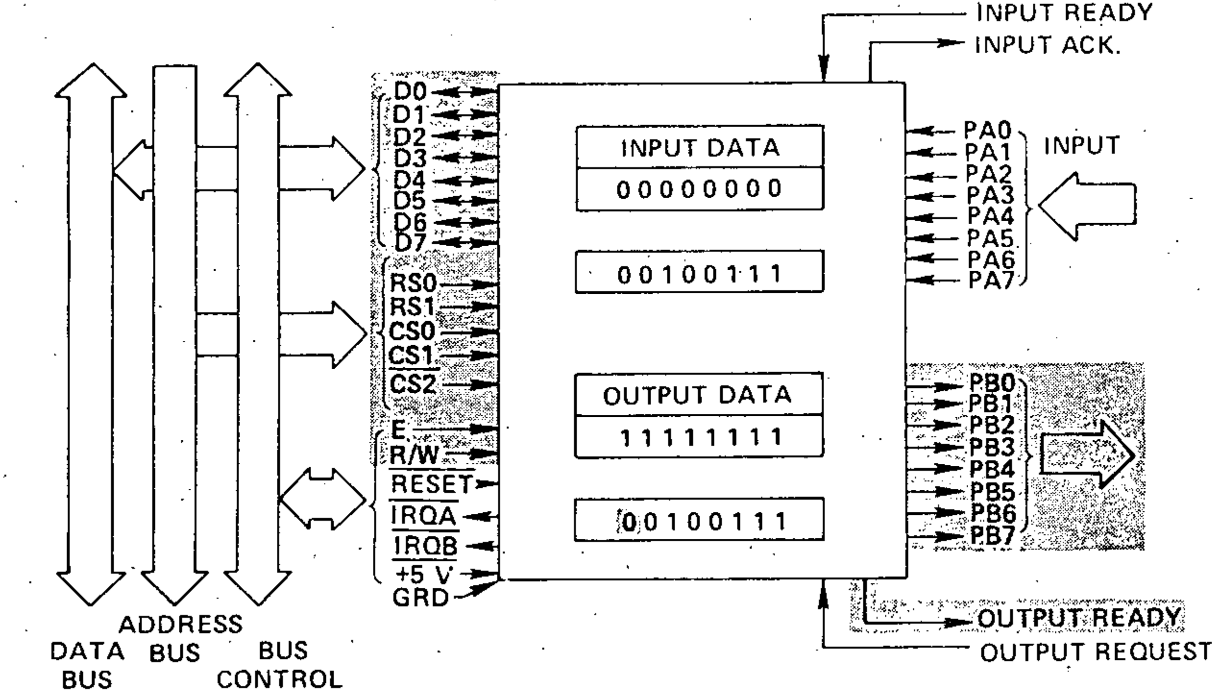


D2555

### EXTERNAL DEVICE REQUESTS OUTPUT DATA

- "OUTPUT REQUEST" transition sets status bit and pulls down $\overline{\text{IRQ}}$
- Interrupt response routine will identify this interrupt by checking status bits

## PIA APPLICATION EXAMPLE



D2556

### INTERRUPT RESPONSE ROUTINE

- Clears interrupt with dummy read
- Writes output data
- "OUTPUT READY" signal generated automatically as a result of the write operation

## CICLO DE OPERACION.

Veamos ahora un ejemplo real de cómo el sistema ejecuta una instrucción que en este caso direcciona una localidad de memoria en la que se encuentra un PIA que a su vez traerá cierta información del mundo exterior.

La instrucción "ROL", que usaremos nos dice: rotar -- los datos a la izquierda en el modo de direccionamiento extendido aqui podremos ver como en la ejecución de una sola instrucción, están realizadas operaciones con registros externos tal como si fueran localidades de memoria.

Cada ciclo es compuesto primero de la fase $\emptyset_1$ del reloj que al ponerse en uno, ocasiona que el 6800 mande una dirección sobre las 16 líneas del BUS.

Para completar el ciclo la fase $\emptyset_2$ del reloj al ponerse en uno activa las 8 líneas del BUS de datos para mover la instrucción o dato, direccionado durante la $\emptyset_1$. Como - puede verse de la figura; la $\emptyset_2$ maneja la entrada DBE (Habilitación del BUS de datos).

También, durante la $\emptyset_2$ el 6800 pone dos señales que - usa como "Mastras" del BUS de datos para comandar a los -- "Esclavos" externos. La señal VMA (Dirección Válida) dice si un dato es movido o no durante la $\emptyset_2$, la señal R/W - - (leer/escribir) nos dice en qué sentido se van a mover los datos en el BUS, si es hacia el CPU (Read) o fuera del CPU (Write) ésto es esencialmente toda la interface necesaria para este sistema mínimo.

Durante el primer ciclo, el CPU direcciona el primer byte de la instrucción ROL en ROM, cuadro 1a, y pone el código operacional de la instrucción ROL en el registro de instrucción IR, este registro maneja un ROM el cual manda el código operacional al control lógico del CPU, que a su vez dirige al CPU para los siguientes ciclos.

La primera cosa que el código operacional le dice al CPU es que debe traer dos bytes más del ROM para la dirección necesitada en esta instrucción, (modo extendido), el CPU lo hace así en los ciclos 2, cuadro 2a. y 2b, y 3, cuadro 3a. y 3b, almacenando estos bytes en el registro temporal 1 y 2.

Durante el ciclo 4 (cuadro 4a. y 4b) el CPU manda la dirección, parte de la instrucción ROL, y trae los datos del registro de entrada/salida del puerto A de la PIA al registro temporal 1.

En el 6800 los datos no son puestos en el acumulador como sería en la mayoría de las computadoras sino que los pone directamente en el ALU, (Unidad Lógica-Aritmética) que puede ser usado como registro-acumulador.

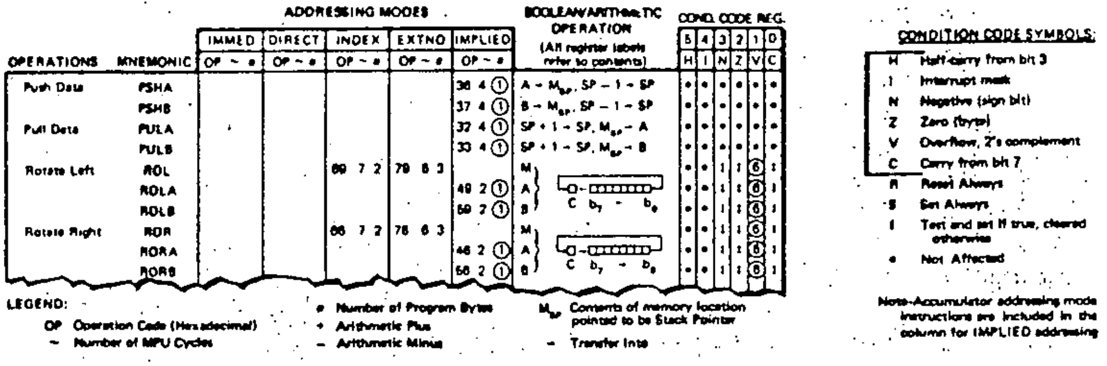
El corrimiento es hecho en el ciclo 5 (cuando 5a. y 5b). La instrucción ROL rota los datos a través del FLIP-FLOP del acarreador. Nótese que durante la $\emptyset_2$ de este ciclo (cuadro 5b) el BUS de datos no está activado con la señal VMA del CPU, no tiene sentido habilitar el BUS de datos en ciclos donde no va a ser utilizado.

Al final de este ciclo cualquier bit del registro de código condicional que haya sido afectado por esta operación habrá sido modificado. Si el dato fue 10000000 antes del corrimiento, después de éste será 0000 0000 y el F-F del carry estará puesto en 1.

Finalmente, durante el ciclo 6 el registro del PIA es direccionado de nuevo y los datos se mueven del ALU de regreso al registro de donde partieron. Esta instrucción con sólo tres bytes y una ejecución de sólo 6MS nos permite jugar con palabras en cualquier lado de la memoria.

En una computadora típica sin este tipo de instrucciones se tiene como mínimo que escribir otras dos instrucciones para tener este mismo resultado.

Primero se tienen que mover los datos al acumulador del CPU antes de realizar la operación en el ALU y después debe moverse los datos de regreso al registro, esto significa escribir 3 veces más de código y posiblemente tres veces más de tiempo de ejecución.



| | | ADDRESSING MODES | | | | | BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents) | COND. CODE REG. | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | IMMED | DIRECT | INDEX | EXTND | IMPLIED | | 5 | 4 | 3 | 2 | 1 | 0 |
| OPERATIONS | MNEMONIC | OP ~ # | OP ~ # | OP ~ # | OP ~ # | OP ~ # | | H | I | N | Z | V | C |
| Push Data | PSHA | | | | | 36 4 ① | A → $M_{SP}$, SP − 1 → SP | • | • | • | • | • | • |
| | PSHB | | | | | 37 4 ① | B → $M_{SP}$, SP − 1 → SP | • | • | • | • | • | • |
| Pull Data | PULA | | | | | 32 4 ① | SP + 1 → SP, $M_{SP}$ → A | • | • | • | • | • | • |
| | PULB | | | | | 33 4 ① | SP + 1 → SP, $M_{SP}$ → B | • | • | • | • | • | • |
| Rotate Left | ROL | | | 69 7 2 | 79 6 3 | | M | • | • | ↕ | ↕ | ⑥ | ↕ |
| | ROLA | | | | | 49 2 ① | A C ← $b_7$ − $b_0$ | • | • | ↕ | ↕ | ⑥ | ↕ |
| | ROLB | | | | | 59 2 ① | B | • | • | ↕ | ↕ | ⑥ | ↕ |
| Rotate Right | ROR | | | 66 7 2 | 76 6 3 | | M | • | • | ↕ | ↕ | ⑥ | ↕ |
| | RORA | | | | | 46 2 ① | A C $b_7$ → $b_0$ | • | • | ↕ | ↕ | ⑥ | ↕ |
| | RORB | | | | | 56 2 ① | B | • | • | ↕ | ↕ | ⑥ | ↕ |

LEGEND:

OP Operation Code (Hexadecimal)
~ Number of MPU Cycles
# Number of Program Bytes
+ Arithmetic Plus
− Arithmetic Minus
$M_{SP}$ Contents of memory location pointed to be Stack Pointer
→ Transfer Into

CONDITION CODE SYMBOLS:

H Half-carry from bit 3
I Interrupt mask
N Negative (sign bit)
Z Zero (byte)
V Overflow, 2's complement
C Carry from bit 7
R Reset Always
S Set Always
↕ Test and set if true, cleared otherwise
• Not Affected

Note-Accumulator addressing mode instructions are included in the column for IMPLIED addressing

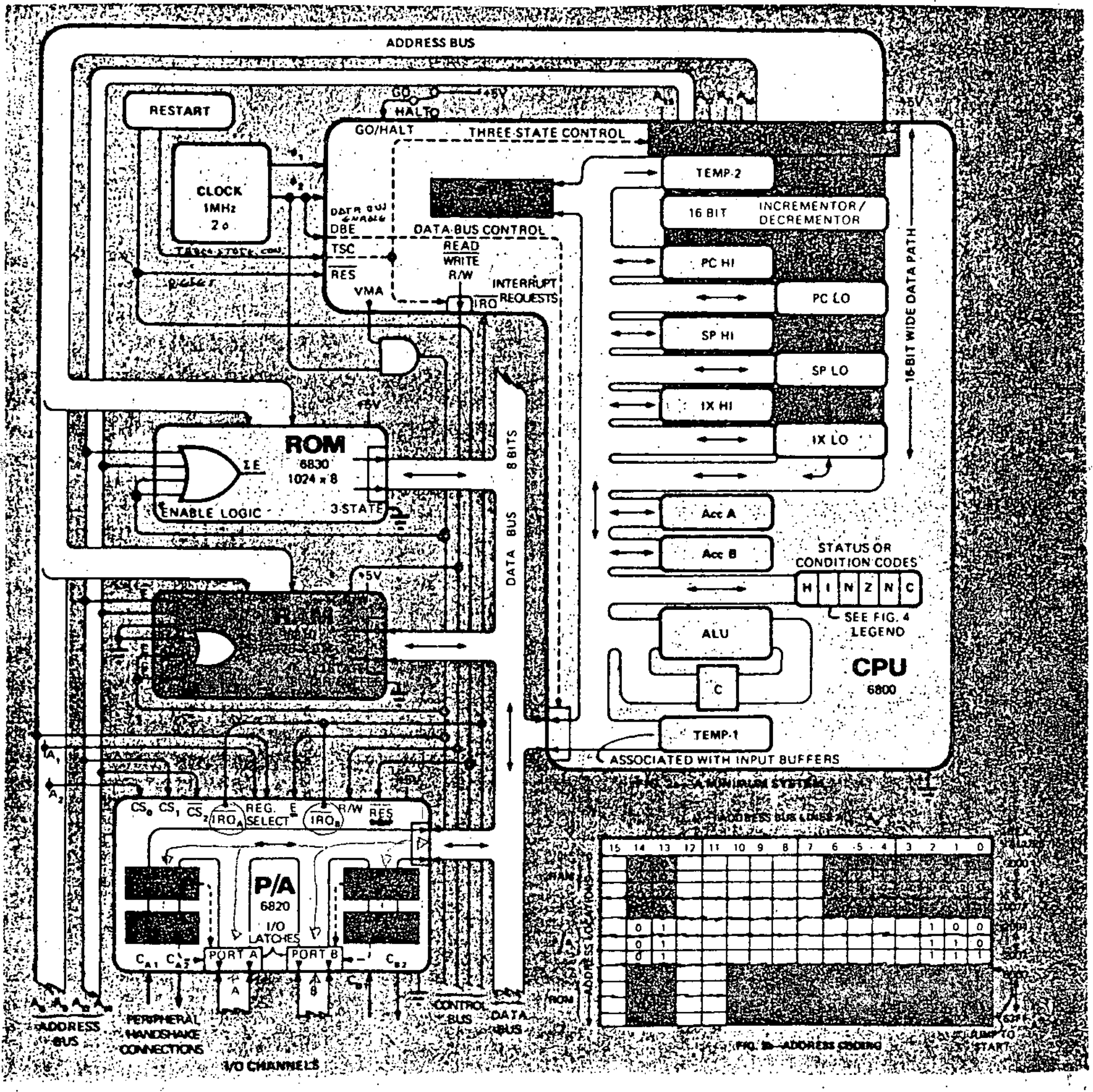
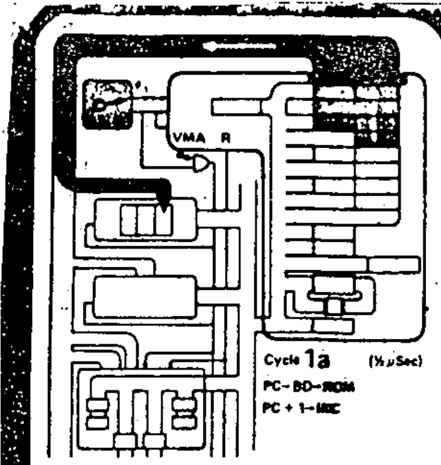
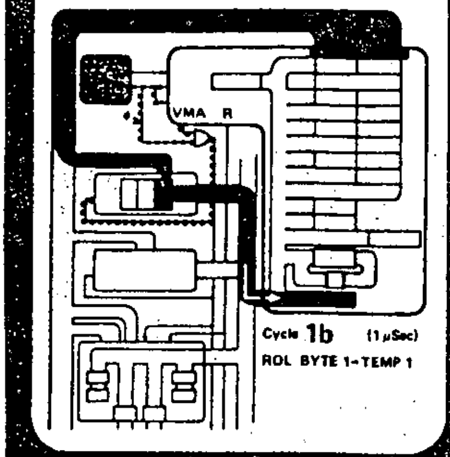


FIG. 3a—A MINIMUM SYSTEM

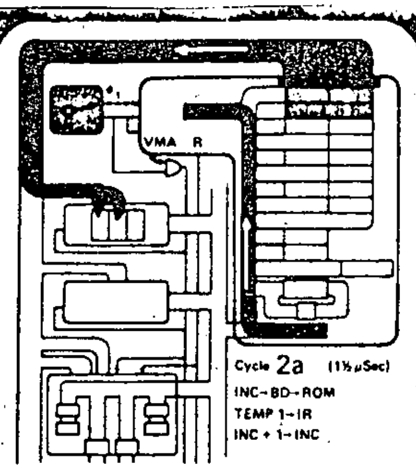
FIG. 3b—ADDRESS CODING



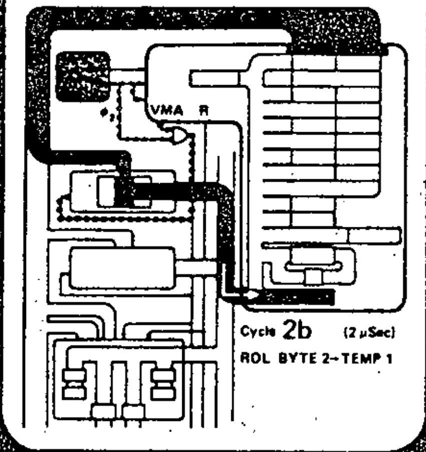
Cycle 1a (½ μSec)
PC→BD→ROM
PC + 1→INC



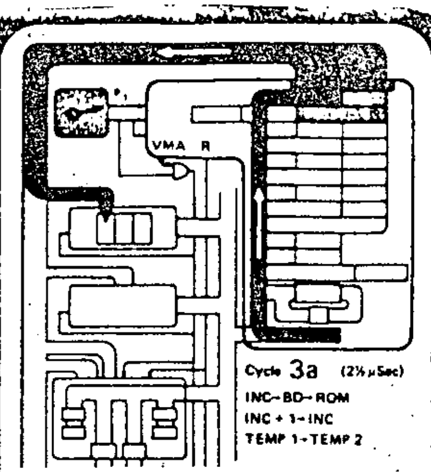
Cycle 1b (1 μSec)
ROL BYTE 1→TEMP 1



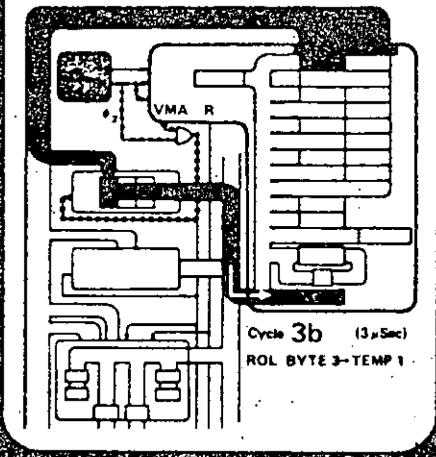
Cycle 2a (1½ μSec)
INC→BD→ROM
TEMP 1→IR
INC + 1→INC



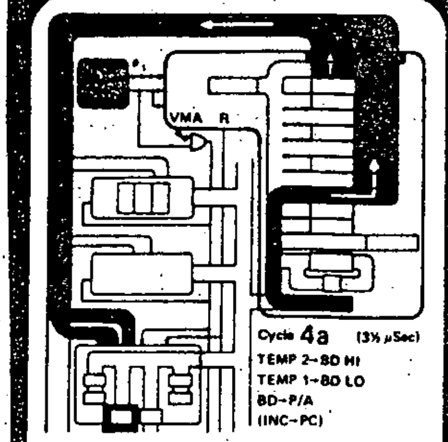
Cycle 2b (2 μSec)
ROL BYTE 2→TEMP 1



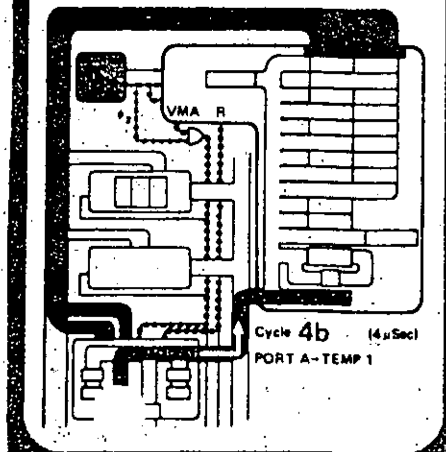
Cycle 3a (2½ μSec)
INC→BD→ROM
INC + 1→INC
TEMP 1→TEMP 2



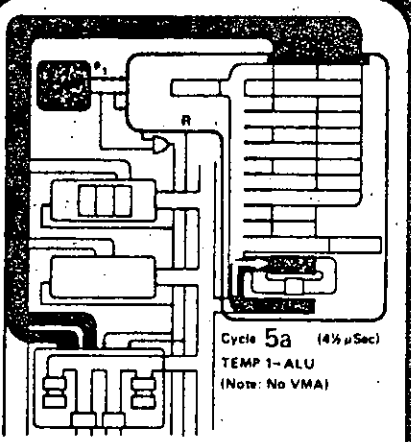
Cycle 3b (3 μSec)
ROL BYTE 3→TEMP 1



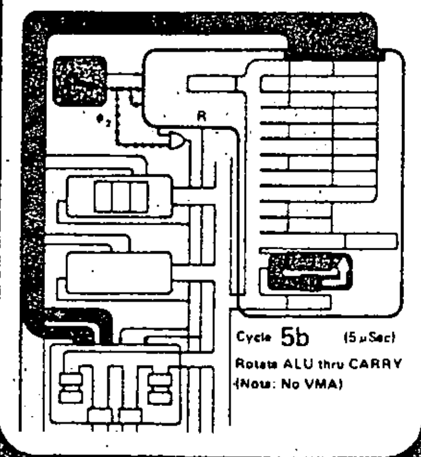
Cycle 4a (3½ μSec)
TEMP 2→BD HI
TEMP 1→BD LO
BD→P/A
(INC→PC)



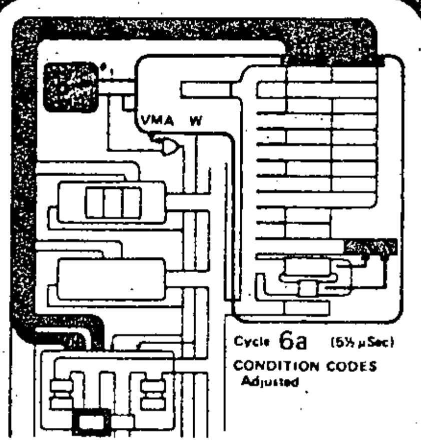
Cycle 4b (4 μSec)
PORT A→TEMP 1



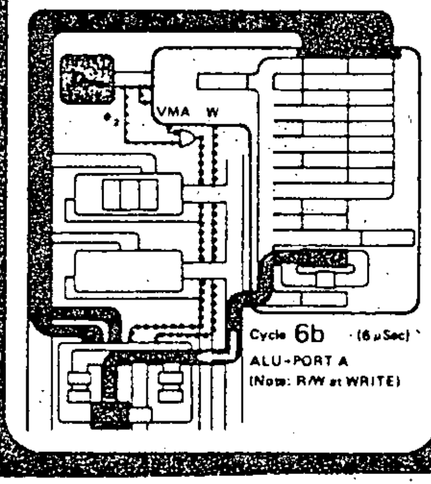
Cycle 5a (4½ μSec)
TEMP 1→ALU
(Note: No VMA)



Cycle 5b (5 μSec)
Rotate ALU thru CARRY
(Note: No VMA)

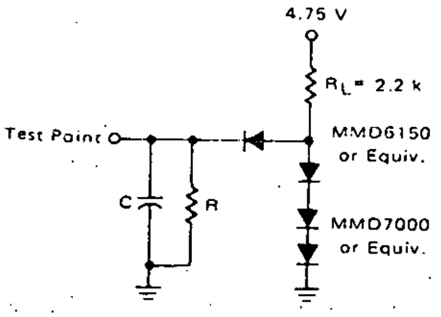


Cycle 6a (5½ μSec)
CONDITION CODES
Adjusted



Cycle 6b (6 μSec)
ALU→PORT A
(Note: R/W at WRITE)

FIGURE 6 -- BUS TIMING TEST LOAD



C = 130 pF for D0-D7
= 90 pF for A0-A15, R/W, and VMA
= 30 pF for BA
R = 11.7 kΩ for D0-D7
= 16.5 kΩ for A0-A15, R/W, and VMA
= 24 kΩ for BA

## TYPICAL POWER SUPPLY CURRENT

FIGURE 7 – VARIATIONS WITH FREQUENCY

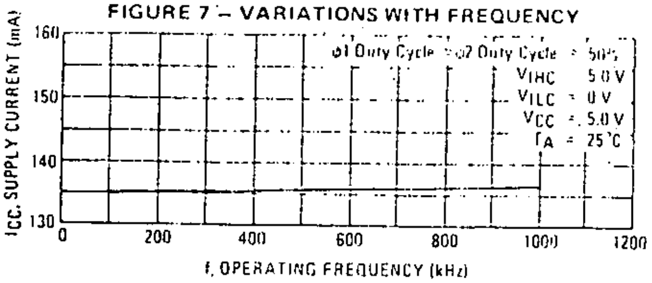
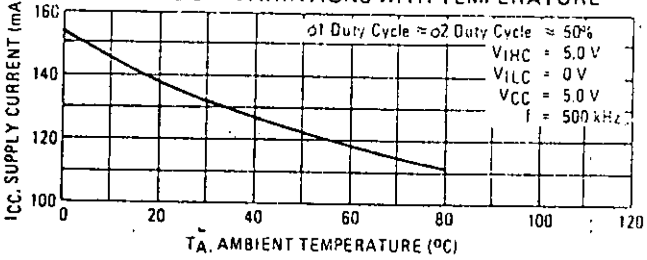
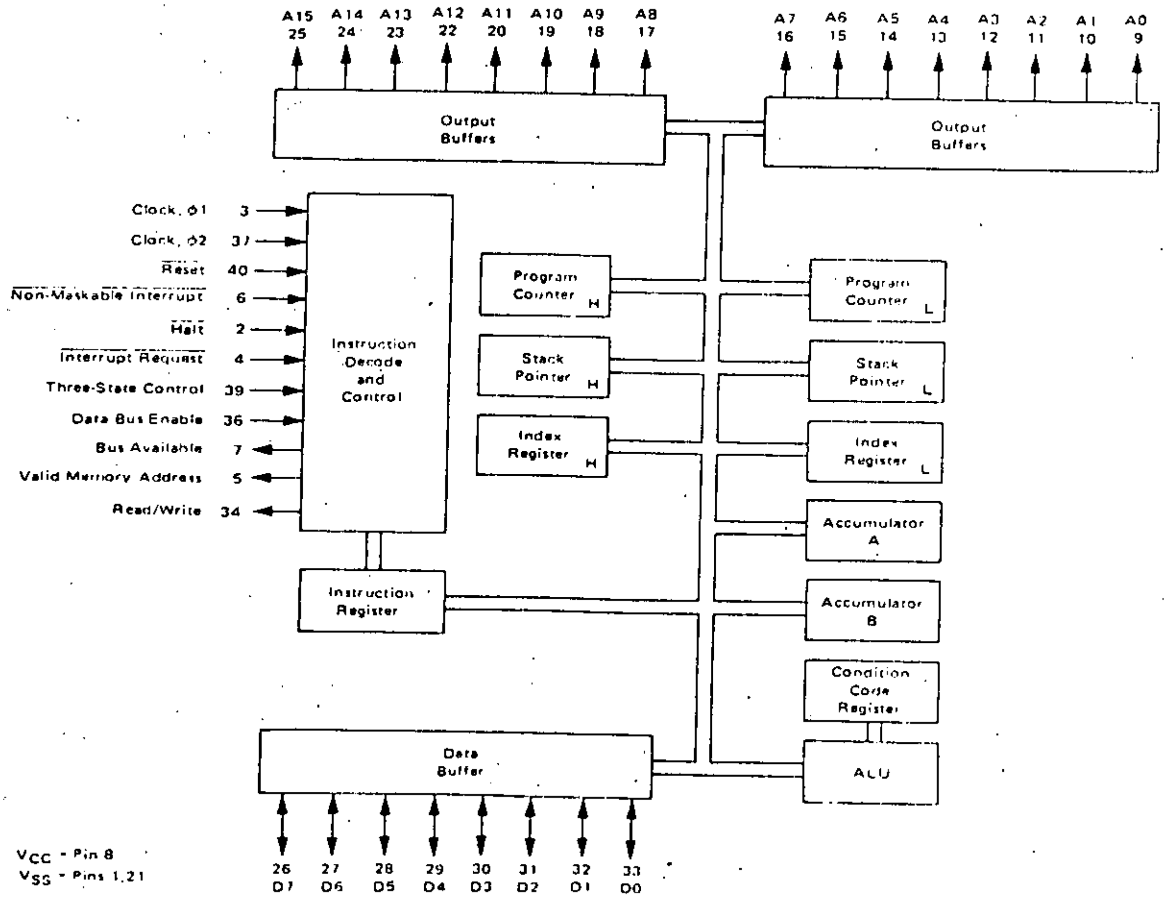


FIGURE 8 – VARIATIONS WITH TEMPERATURE

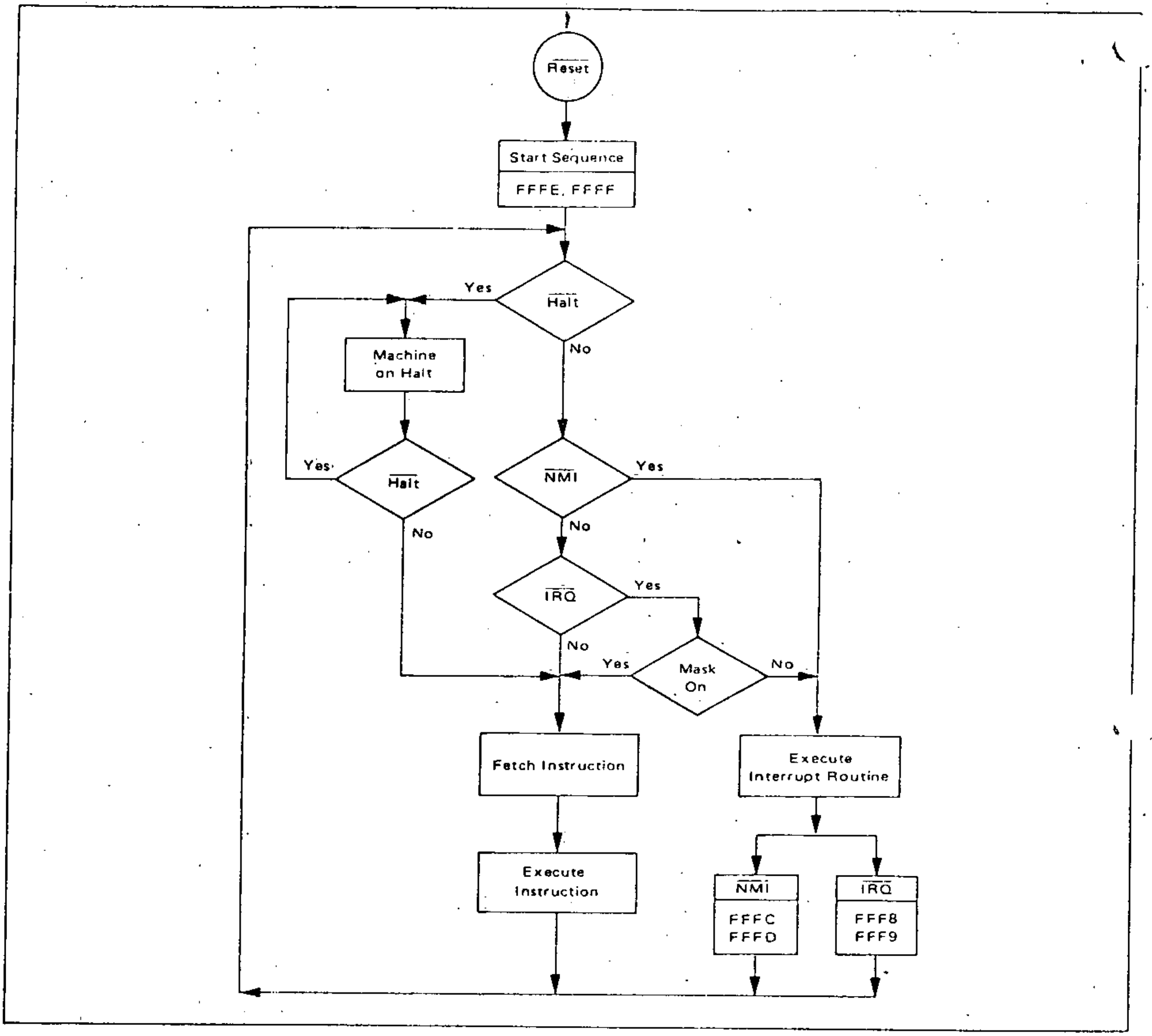


## EXPANDED BLOCK DIAGRAM



$V_{CC}$ = Pin 8
$V_{SS}$ = Pins 1, 21

FIGURE 10 – MPU FLOW CHART



## MPU REGISTERS

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 11).

**Program Counter** – The program counter is a two byte (16-bits) register that points to the current program address.

**Stack Pointer** – The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may have any location (address) that is convenient. In those applications that require storage of information in the stack when power is lost, the stack must be non-volatile.

**Index Register** – The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

**Accumulators** – The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).

MOTOROLA Semiconductors

# MC6802

## Advance Information

## MICROPROCESSOR WITH CLOCK AND RAM

The MC6802 is a monolithic 8-bit microprocessor that contains all the registers and accumulators of the present MC6800 plus an internal clock oscillator and driver on the same chip. In addition, the MC6802 has 128 bytes of RAM on board located at hex addresses 0000 to 007F. The first 32 bytes of RAM, at hex addresses 0000 to 001F, may be retained in a low power mode by utilizing $V_{CC}$ standby, thus facilitating memory retention during a power-down situation.

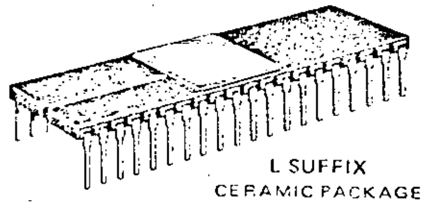
The MC6802 is completely software compatible with the MC6800 as well as the entire M6800 family of parts. Hence, the MC6802 is expandable to 65K words.

- On-Chip Clock Circuit
- 128 x 8 Bit On-Chip RAM
- 32 Bytes of RAM Are Retainable
- Software-Compatible with the MC6800
- Expandable to 65K words
- Standard TTL-Compatible Inputs and Outputs
- 8 Bit Word Size
- 16 Bit Memory Addressing
- Interrupt Capability

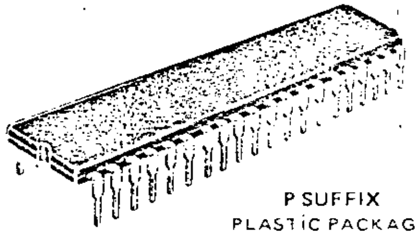
**MOS**

(N-CHANNEL, SILICON-GATE, DEPLETION LOAD)

MICROPROCESSOR
WITH CLOCK AND RAM



L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711

FIGURE 1 – TYPICAL MICROCOMPUTER

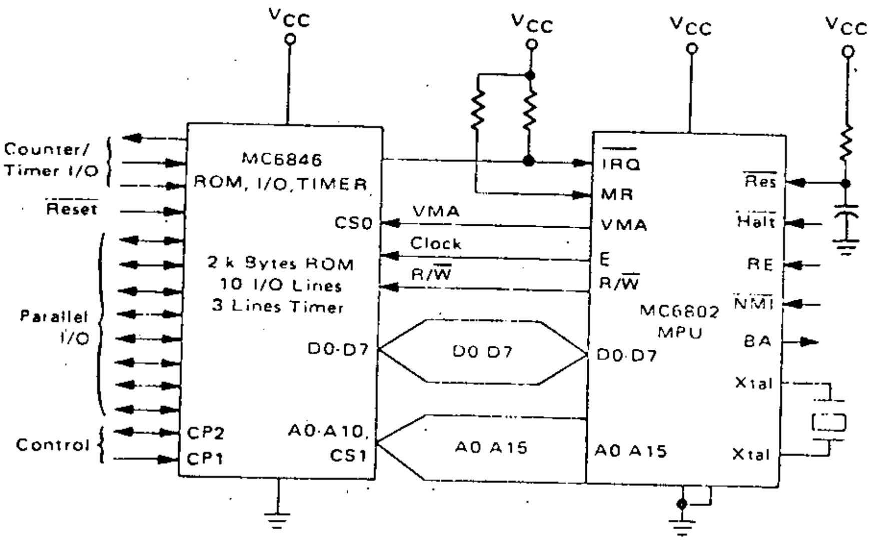
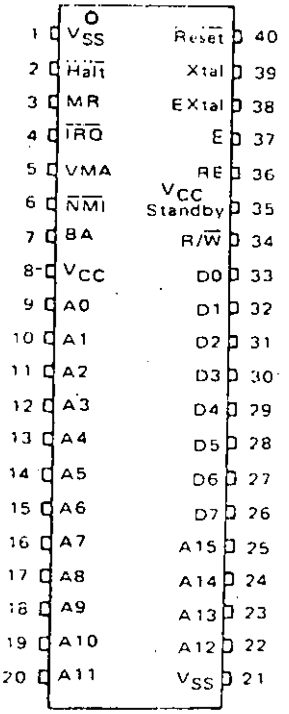


Figure 1 is a block diagram of a typical cost effective microcomputer. The MPU is the center of the microcomputer system and is shown in a minimum system interfacing with a ROM combination chip. It is not intended that this system be limited to this function but that it be expandable with other parts in the M6800 Microcomputer family.

PIN ASSIGNMENT



This is advance information and specifications are subject to change without notice.

FIGURE 5 – TYPICAL DATA BUS OUTPUT DELAY versus CAPACITIVE LOADING

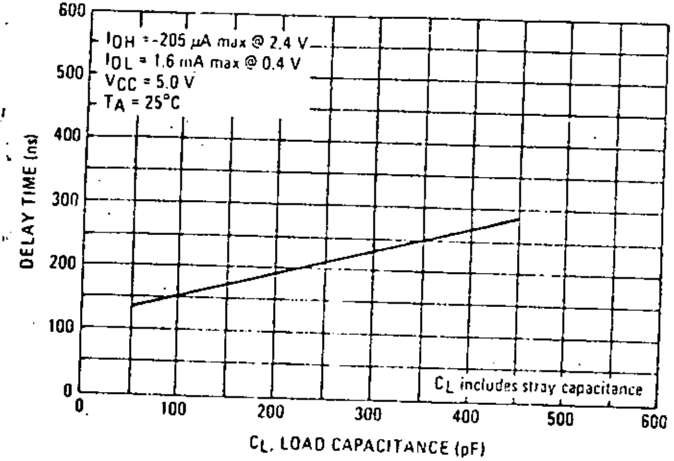


FIGURE 6 – TYPICAL READ/WRITE, VMA, AND ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

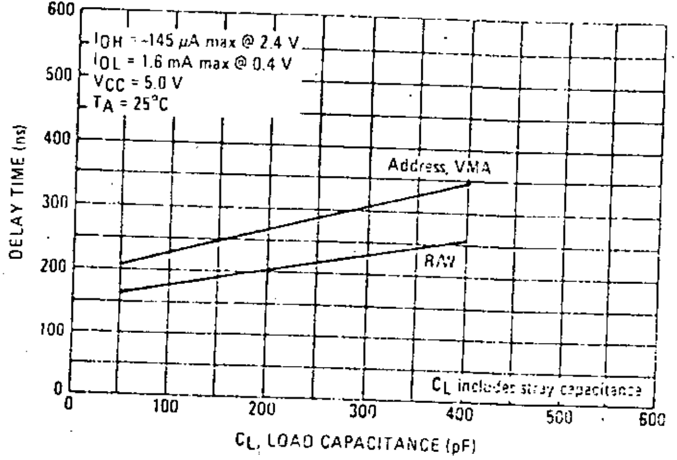
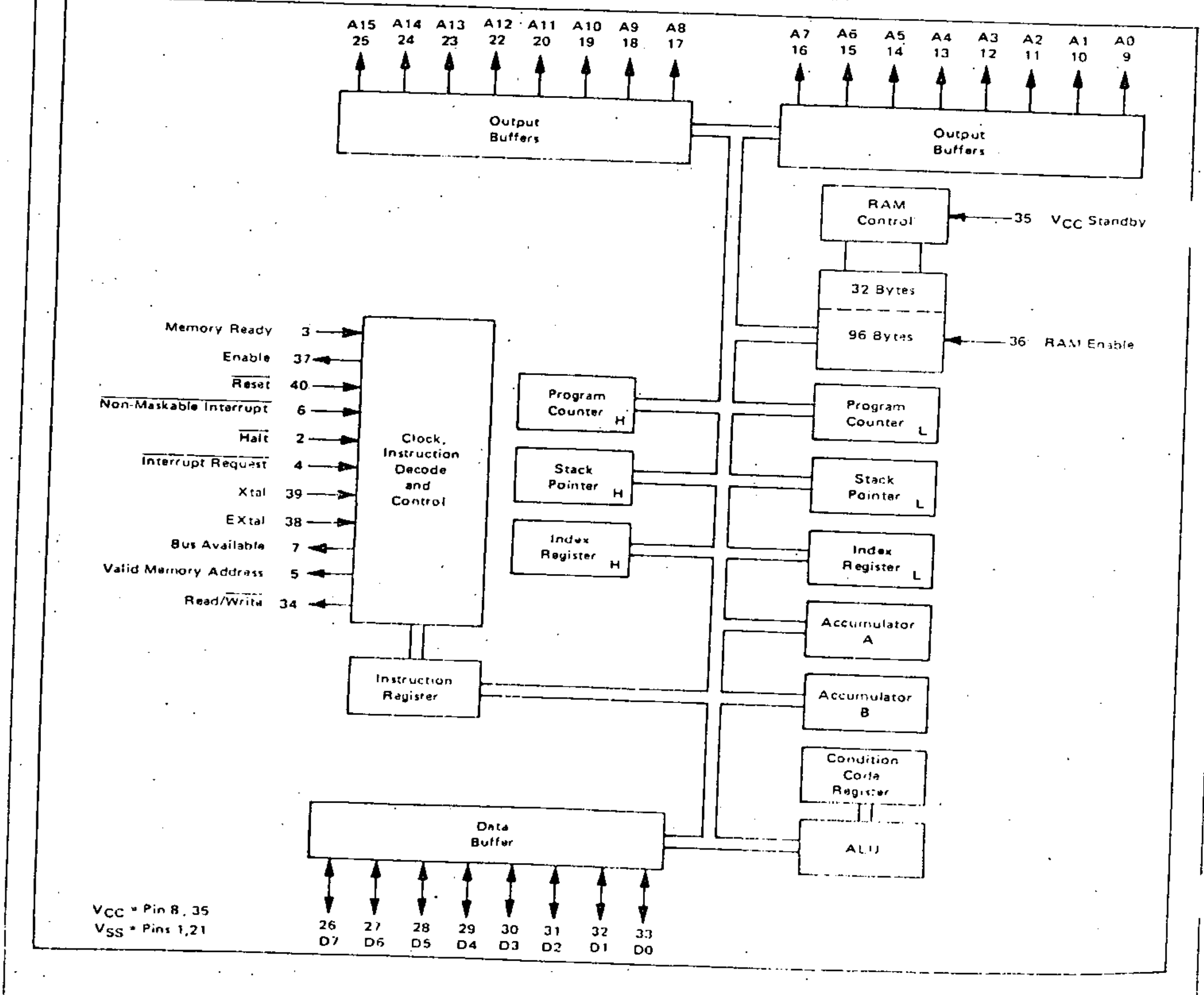


FIGURE 7 – MC6802 EXPANDED BLOCK DIAGRAM



MOTOROLA Semiconductors

# MC6850

(0 to 70°C; L or P Suffix)

# MC6850C

(-40 to 85°C; L Suffix only)

## MOS

(N-CHANNEL, SILICON-GATE)

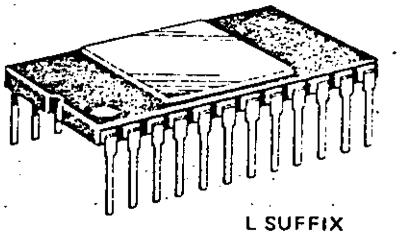
ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER

## ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER (ACIA)

The MC6850 Asynchronous Communications Interface Adapter provides the data formatting and control to interface serial asynchronous data communications information to bus organized systems such as the MC6800 Microprocessing Unit.

The bus interface of the MC6850 includes select, enable, read/write, interrupt and bus interface logic to allow data transfer over an 8-bit bi-directional data bus. The parallel data of the bus system is serially transmitted and received by the asynchronous data interface, with proper formatting and error checking. The functional configuration of the ACIA is programmed via the data bus during system initialization. A programmable Control Register provides variable word lengths, clock division ratios, transmit control, receive control, and interrupt control. For peripheral or modem operation three control lines are provided. These lines allow the ACIA to interface directly with the MC6860L 0-600 bps digital modem.

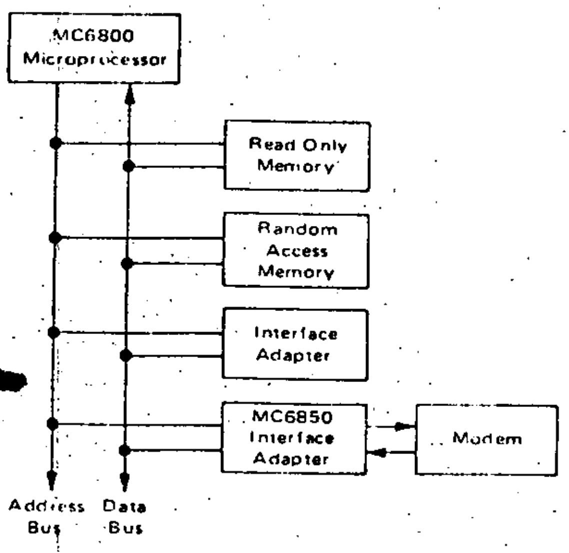
- Eight and Nine-Bit Transmission
- Optional Even and Odd Parity
- Parity, Overrun and Framing Error Checking
- Programmable Control Register
- Optional ÷1, ÷16, and ÷64 Clock Modes
- Up to 500 kbps Transmission
- False Start Bit Deletion
- Peripheral/Modem Control Functions
- Double Buffered
- One or Two Stop Bit Operation



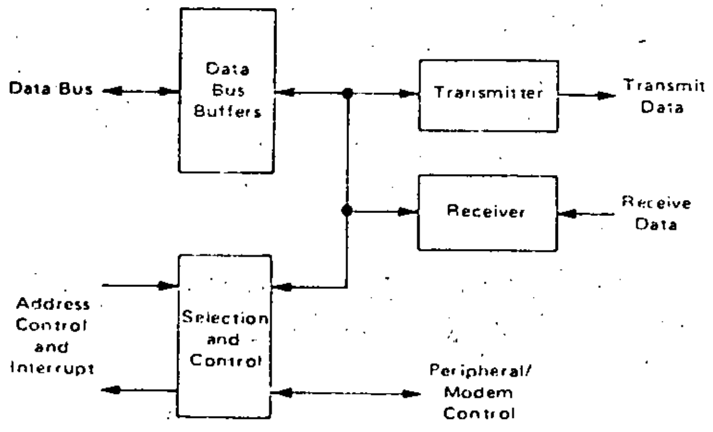
L SUFFIX
CERAMIC PACKAGE
CASE 716

NOT SHOWN: P SUFFIX
PLASTIC PACKAGE
CASE 709

### M6800 MICROCOMPUTER FAMILY BLOCK DIAGRAM



### MC6850 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER BLOCK DIAGRAM



MOTOROLA Semiconductors

[illegible]

# MC6860

(0 to 70°C; L or P Suffix)

# MC6860C

(-40 to 85°C; L Suffix only)

## 0-600 bps DIGITAL MODEM

The MC6860 is a MOS subsystem designed to be integrated into a wide range of equipment utilizing serial data communications.

The modem provides the necessary modulation, demodulation and supervisory control functions to implement a serial data communications link, over a voice grade channel, utilizing frequency shift keying (FSK) at bit rates up to 600 bps. The MC6860 can be implemented into a wide range of data handling systems, including stand alone modems, data storage devices, remote data communication terminals and I/O interfaces for minicomputers.

N-channel silicon gate technology permits the MC6860 to operate using a single voltage supply and be fully TTL compatible.

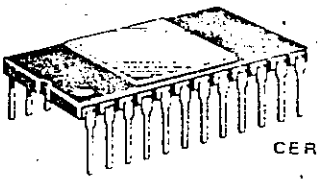
The modem is compatible with the M6800 microcomputer family, interfacing directly with the Asynchronous Communications Interface Adapter to provide low-speed data communications capability.

- Originate and Answer Mode
- Crystal or External Reference Control
- Modem Self Test
- Terminal Interfaces TTL-Compatible
- Full-Duplex or Half-Duplex Operation
- Automatic Answer and Disconnect
- Compatible Functions for 100 Series Data Sets
- Compatible Functions for 1001A/B Data Couplers

**MOS**

(N-CHANNEL, SILICON-GATE)

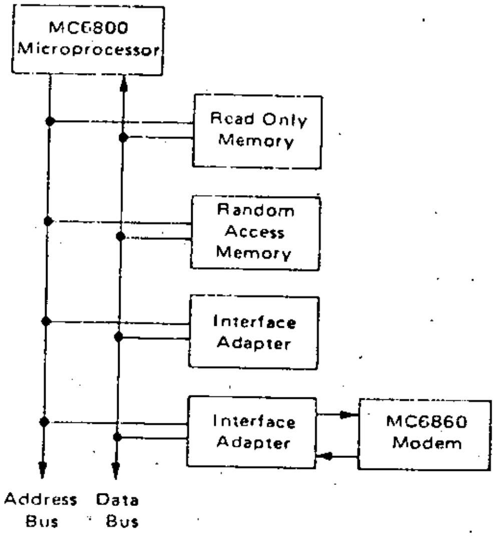
0-600 bps
DIGITAL MODEM



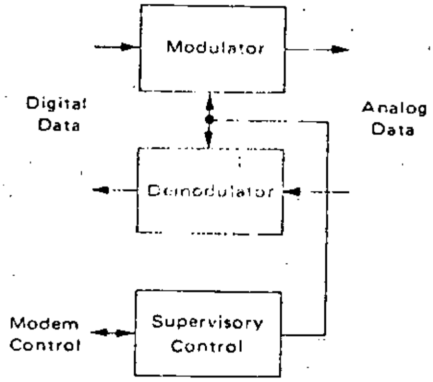
L SUFFIX
CERAMIC PACKAGE
CASE 716

NOT SHOWN: P SUFFIX
PLASTIC PACKAGE
CASE 709

M6800 MICROCOMPUTER FAMILY
BLOCK DIAGRAM



MC6860 DIGITAL MODEM
BLOCK DIAGRAM



MOTOROLA
Semiconductors

# MCM68708L

## Product Preview

### 1024 X 8-BIT ALTERABLE READ ONLY MEMORY

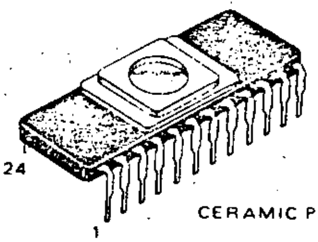
The MCM68708 is an 8192-bit Alterable Read Only Memory designed for system debug usage and similar applications requiring non-volatile memory that must be reprogrammed periodically. The transparent lid on the package allows the memory content to be erased with ultraviolet light. The memory can then be electrically reprogrammed.

- Organized as 1024 Bytes of 8-Bits
- Static Operation
- Standard Power Supplies of +12 V, +5 V, and -5 V.
- Access Time = 500 ns
- Low Power Dissipation
- Chip Select Input for Memory Expansion.
- TTL Compatible
- Three-State Outputs
- Compatible with the 2708

## MOS

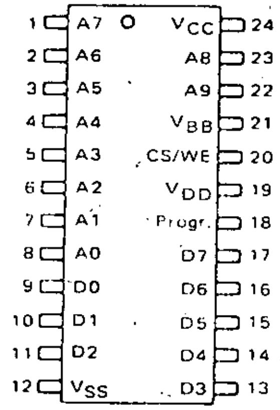
(N-CHANNEL, SILICON-GATE)

1024 X 8-BIT ALTERABLE READ ONLY MEMORY

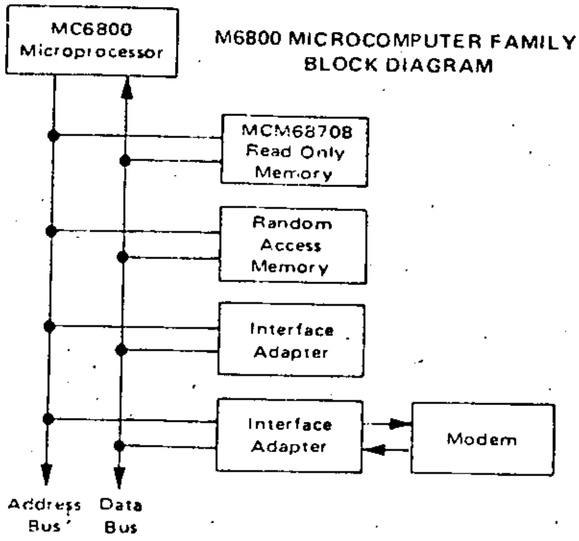


CERAMIC PACKAGE

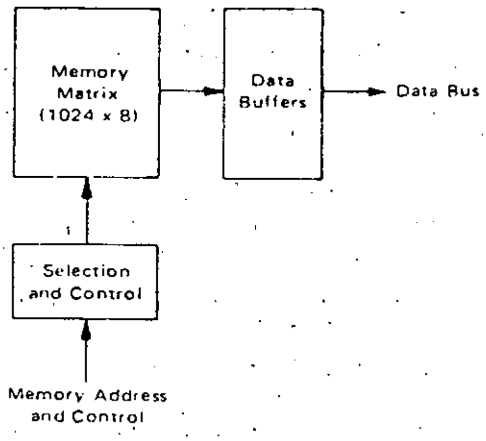
PIN ASSIGNMENT



M6800 MICROCOMPUTER FAMILY BLOCK DIAGRAM



MCM68708 READ ONLY MEMORY BLOCK DIAGRAM



This is advance information and specifications are subject to change without notice.

MOTOROLA

SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

# Product Preview

# MC6809(E)
(1.0 MHz)

# MC68A09(E)
(1.5 MHz)

# MC68B09(E)
(2.0 MHz)

## MOS

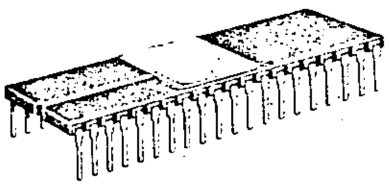
(N-CHANNEL, SILICON-GATE, DEPLETION LOAD)

HIGH-PERFORMANCE MICROPROCESSOR

## HIGH-PERFORMANCE MICROPROCESSOR

- MC6800 COMPATIBLE
  - Hardware – Interfaces With All M6800 Peripherals
  - Software – Upward Compatible Instruction Set and Addressing Modes
- HARDWARE FEATURES
  - On-Chip Oscillator (MC6809) 4 X fo Clock
  - Optional ÷1 External Clock Inputs (MC6809E)
  - MRDY Input Extends Data Access Times for Use With Slow Memory
  - $\overline{\text{BREQ}}$/TSC Allows Quick Access to Bus for DMA and Memory Refresh
  - Last Instruction Cycle Output for Identification of Opcode Fetch (MC6809E)
  - Fast Interrupt Request Input Stacks Only Program Counter and Condition Code,
  - Interrupt Acknowledge Output Allows Vectoring by Device
  - Busy Output Eases Multiprocessor Design (MC6809E)
- ARCHITECTURAL FEATURES
  - Two 8-Bit Accumulators Can Be Concatenated to Form One 16-Bit Accumulator
  - Two 16-Bit Index Registers
  - Two 16-Bit Indexable Stack Pointers
  - Direct Page Register Allows Direct Addressing Throughout Memory Space
- INSTRUCTION SET
  - Extended Range Branches
  - 16-Bit Arithmetic
  - Push/Pull Any Register or Set of Registers To/From Either Stack
  - 8 X 8 Unsigned Multiply
  - Transfer/Exchange Any Two Registers of Equal Size
  - Enhanced Pointer Register Manipulation
- ADDRESSING MODES
  - All MC6800 Modes, Plus PC Relative, Extended Indirect, Indexed Indirect, and PC Relative Indirect
  - Direct Addressing Available for All Memory Access Instructions
  - Index Mode Options Include Accumulator or Up to 16-Bit Constant Offset, and Auto-Increment/Decrement (by 1 or 2) With Any of the Four Pointer Registers

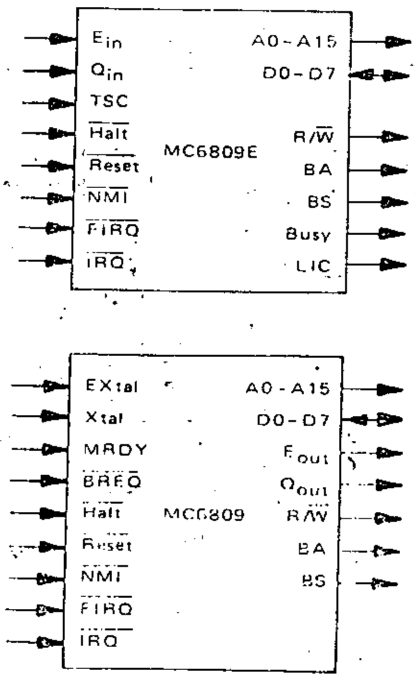
This is advance information and specifications are subject to change without notice.



L SUFFIX
CERAMIC PACKAGE
CASE 715

NOT SHOWN:
P SUFFIX
PLASTIC PACKAGE
CASE 711

FIGURE 1 – BLOCK DIAGRAMS



## introducing

# the Motorola MC6809

An 8-Bit Microprocessor (MPU) designed with particular attention to providing an enhanced software and higher performance central processing unit.

2.5 to 5 Times the Performance of the MC6800

MC6800 Software Compatible (Source Code)

Architectural Improvements

- Additional 16-Bit Registers
- Expanded Addressing Mode Common to All Index (3) Registers
- 16-Bit Operations
- 8 X 8 Multiplier

Software Improvements Support

- Tailored to Higher-Level-Language
- Position Independence
- Structured, Highly Subroutined Code
- Multi-task and Multi-processor Organization
- Stack-Oriented Compiler Instructions
- Reentrancy and Recursion

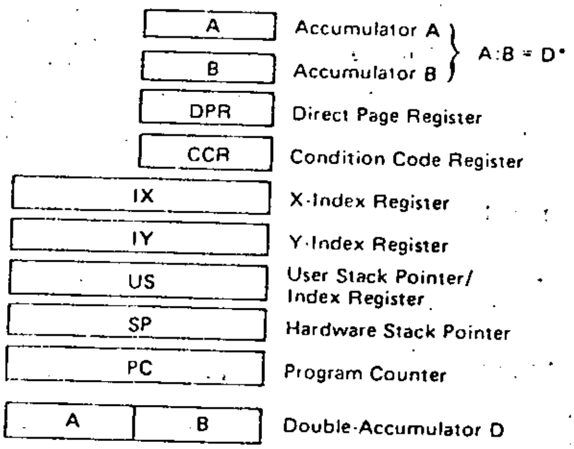
Hardware Improvements

- On-Chip Clock (MC6809) or Off-Chip Clock (MC6809E)
- Additional Control (Fast IRQ, Ready, Busy, Last Instruction Cycle Output)
- 2-MHz Base Operation
- Interrupt Acknowledge

Compatible with All 6800 Family Peripherals

# the architectural improvements

## Programming Model



*The concatenation of A:B is the Double-Accumulator.

## 10 Addressing Modes

- Inherent
- Immediate
- Direct
- Extended
- Extended Indirect
- Register
- Indexed
- Indexed Indirect
- Relative
- Long Relative

## Powerful Indexing Capabilities

- There are 5 Indexable Registers: X, Y, S, U, and PC
- With 4 Options for Both Indexed and Indexed Indirect:

  Constant-Offset
  Accumulator Offset Using A, B, or D
  Auto-Increment or Auto-Decrement
  Indirection

## 6 Interrupts

- NMI Non-maskable
- IRQ Normal-Maskable
- FIRQ Fast-Maskable
- SWI Software
- SWI2 Software
- SWI3 Software

## 16-Bit Operations

- Add to D Accumulator
- Subtract from D Accumulator
- Load D Accumulator
- Store to D Accumulator
- Compare D Accumulator
- Load X, Y, S, U Registers
- Store X, Y, S, U Registers
- Compare X, Y, S, U Registers
- Load Effective Address
- Sign Extend
- Transfer Registers
- Exchange Registers
- Push/Pull X, Y, S, U
- Add B Accumulator to X Register

## Additional Operations

- 8 X 8 Unsigned Multiply = 16-Bit
- Push/Pull Multiple Registers
- Long Branches
- Synchronize with Interrupt Line (to synchronize instructions with an external event)
- 3 Software Interrupts

AN 8-BIT MICROPROCESSOR WITH 16-BIT OPERATIONS

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# EL MICROPROCESADOR Z-80. SOFTWARE

M. EN C. ANGEL KURI MORALES

MARZO, 1979

Como resultado de su arquitectura, el Z80 incorpora una serie de instrucciones que lo hacen el más poderoso de los sistemas de su tipo. Sus instrucciones contienen todas las del 8080 y agregan:

- Acceso por registros de índice.
- Acceso a bits particulares de memoria.
- Operaciones aritméticas (suma y resta) de 16 bits.
- Movimientos de bloques.
- Comparaciones de bloques.
- Saltos relativos.

Todo el 'software' del Z80 es compatible con el 8080 y, - aunque el 8080 ocupa 244 códigos de operación (OP CODES), [utilizando las combinaciones de 8 bits ($2^8 = 256$) excepto 12] el Z80 - incorpora, a esos 244 OP CODES, 452 adicionales, para un total de 696 OP CODES.

Este notable incremento y, a la vez, compatibilidad de códigos se logra mediante la utilización de los códigos no usados por el 8080 como 'piedras de toque', de tal manera que cada código inválido (del 8080) se usa como indicador de que el <u>siguiente</u> byte contendrá la información de la operación a efectuarse.

En las hojas siguientes se incluye un resumen de las instrucciones del Z80.

## Arquitectura y software.

Como se recordará, la arquitectura del CPU Z80 es la que se muestra en la figura 1.

Los registros IX e Iy son registros de índice. Los registros AF, BC, DE y HL, así como sus alternos AF', BC', DE' y HL' son de propósito general. Aquí, los resgistros F y F' representan a uña cadena de Flip-Flops que son utilizados por el sistema como banderas. Estas bandaras toman sus valores dependiendo de la operación efectuada por el CPU. En general estos registros no son directamente accesibles (es decir, no contienen datos de variables). La estructura del registro F es la siguiente:

| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|
| S | Z | X | H | X | P/V | N | C |

X significa que el bit no se usa.

Las funciones de las banderas son:

S — bandera de signo. Esta bandera se enciende (1) si el signo del resultado en las operaciones aritméticas en el acumulador (para 8 bits) o en el par HL (para 16 bits) es 1. En este caso el bit 7 (para A) y el bit 15 (para HL) funciona como bit de signo.

Z -- bandera cero. En los mismos casos que el anterior, el bit se enciende si el resultado es cero y se apaga en caso contrario.

H -- Carry medio. Este bit índica que hubo carry del cuarto (b3) al quinto bit (b4) en operaciones del acumulador. Se usa para aritmética decimal.

P/V -- bandera de paridad/sobreflujo. Este bit funciona indicando paridad par (1) o impar (0) cuando se ejecutan operaciones lógicas (AND, OR, XOR). Asimismo, - en operaciones aritméticas se enciende si hubo - 'overflow" para aritmética complemento a dos y se apaga en caso contrario.

N -- bandera de resta. Este bit se usa para operaciones decimales internas. Es cero si la operación fue suma y uno si fue resta.

C -- bandera de acarreo. La bandera de 'carry" se enciende si hay 'overflow' del bit más significativo del - acumulador o del par HL, o si hubo un 'borrow' (a<b) en una resta (a-b).

La ejecución de un programa se ve afectada de diversas formas por las banderas, como se ve de los programas a continuación:

```
; Encuentra una cadena de caracteres en memoria.
; a la entrada:
```

```
; BC contiene el tamaño del bloque en donde se va a bus-
; car.
; HL contiene la dirección de la primera localidad de me
; moria en donde se va a buscar la cadena.
; DE  contiene la localidad en donde se encuentra el pri
; mer carácter de la cadena que se va a buscar.


EJEMPLO 1:   JR        START
CADENA:      DEFB     'ESTA ES UNA CADENA'
START:       LD        BC, BLOCKSIZE
             LD        DE, CADENA
             LD        HL, BLOCKADD
             PUSH      DE
             BUSCA     18
             HLT
BUSCA:       MACRO     LCAD
BUSCA 1:     POP       DE ; TOMA DIRECCION DE CADENA
             PUSH      DE ; Y GUARDALA EN EL 'STACK'
             LD        A, (DE) ; TOMA EL PRIMER BYTE
             CPIR              ; BUSCA EL CARACTER.
             JP        PO, NOTFOUND; AQUI LA BANDERA P/V ES
                       ; CERO SI BC = 0.
             EXX       ; INTERCAMBIA REGISTROS.
             LD B,LCAD-1 ; CARACTERES QUE FALTAN
BUSCA 2:     EXX
             INC DE    ; APUNTA AL SIGUIENTE BYTE DE LA CA-
                       ; DENA.
```

```
        LD A,(DE)     ; TRAELO AL ACUMULADOR
        CPI           ; COMPARA Y ACTUALIZA
        JP PO, NOTFOUND ; IGUAL QUE ARRIBA
        JR NZ, BUSCA1 ; SI NO SE COMPLETA LA CADENA, RECOMIENZA.
        EXX           ; EXAMINA B!;
        DJNZ BUSCA2   ; CONTINUA CON LA CADENA
FOUND:  SCF           ; SI ENCONTRAMOS LA CADENA, CARRY = 1.
        JR BUSCA FIN ; SALTA AL FINAL
NOTFOUND; OR  A       ; SI NO ENCONTRAMOS LA CADENA, CARRY = 0
BUSCAFIN:  POP  DE    ; RECUPERA EL DATO DEL
        ENDM          ; STACK
```

Este programa ilustra varias características del lenguaje ensamblador AZM80.

1) Etiquetas. Las direcciones donde se almacenará el código no necesitan ser explicitadas. AZM80 sustituirá el semibólico por su valor al momento de ligar.

2) Vectores de datos. La pseudo-instrucción DEFB, deja espacio suficiente para almacenar la cadena que se define a continuación.

3) Macros. Las macroinstrucciones o simplemente macros, son conjuntos de instrucciones que el ensamblador reconoce y que pueden tener parámetros. Cuando el ensamblador encuentra el nombre del macro, lo sustituye por las instrucciones que se incluyen

en su declarativa. Asimismo, sustituye los parámetros del macro (de haberlo) por su valor a la llamada. En este caso la llamada "BUSCA 18" inducirá la expansión del macro en donde la instrucción:

```
LD B, LCAD-1
```

será sustituida por

```
LD B, 17.
```

4) Evaluación de constantes. Como se vió en el ejemplo anterior, el ensamblador calculó la expresión (LCAD-1) y la sustituyó por su valor. Hya que tener presente, sin embargo, que los valores de las expresiones son evaluados solamente durante el ensamblado.

Macros vs. subrutinas. Frecuentemente, el programador debe decidir entre usar macros o utilizar subrutinas. Para evaluar qué es más conveniente, es necesario analizar sus diferencias.

¿Qué es un macro? Como se desprende del ejemplo anterior, un macro es, sumplemente, un conjunto de instrucciones que se repetirán varias veces a lo largo de un programa, y a las cuales el programador define por medio de una construcción del tipo:

```
MACRO      NOMBRE    (PARA, PARZ,...)
           ______
           ______       CUERPO.
           ______
           ENDM
```

Cada vez que NOMBRE seguido por sus parámetros aparece, el ensamblador lo sustituirá por el cuerpo del macro. De esta manera, si un macro (digamos BUSCA) consta de 19 instrucciones, seis llamadas a BUSCA generarán seis expansiones y, por consecuencia; 114 instrucciones ejecutables.

¿Qué es una subrutina? Una subrutina es un grupo de instrucciones que no aceptan parámetros en el sentido de un macro - (es decir; el ensamblador no los asigna); no está en localidades contiguas al lugar donde se hizo la llamada y, finalmente, implica el uso del 'stack'.

Escribamos el ejemplo anterior, convirtiendo BUSCA en subrutina. En este caso el valor de LCAD se incluirá en el registro C'.

```
EJEMPLO 2:     JR    START
         ;
CADENA   :     DEFB 'ESTA EN OTRA CADENA'
         ;
START:   LD  C,18   ;  C=LCAD-1
         EXX
         LD  BC,  BLOCKSIZE
         LD  DE,  CADENA
         LD  HL,  BLOCKADD
         PUSH  DE
         CALL  BUSCA
```

```
          HLT
          ;
BUSCA:    POP IX ; dirección de RETURN
BUSCA1:   POP DE
          PUSH DE
          LD A,(DE)
          CPIR
          JP PO,NOT
          EXX
          LD  B,C ; B' = C'
BUSCA2:   EXX
          INC DE
          LD A,(DE)
          CPI
          JP PO,NOT
          JR NZ,BUSCA1
          EXX
          DJNZ  BUSCA2
VES:      SCF
          JR FIN
NOT:      OR  A
FIN:      POP  DE
          JP  (IX)  ;  RETURN
```

En el ejemplo, uno de los parámetros es pasando vía el 'stack'. Esto corresponde a PUSH DE antes de 'CALL'. Ahora bien, al ejecutarse el 'CALL', la localidad inmediata posterior al CALL se almacena en el 'stack' (en este caso, la dirección del HLT). Por ello, la primera instrucción de la subrutina BUSCA es un 'POP', cuya función es almacenar esta dirección en el registro IX.

Por otro lado, la evaluación de LCAD es ahora por medio de una copia de C' a B' (LD B,C).

Finalmente, la última instrucción debe ser un salto a la localidad de RETURN. En este caso, el salto se hace por medio de un :

JP (IX) , que introduce al PC el valor de IX. Normalmente, esto se haría con un RET.

Un ejemplo de ese tipo de subrutina es el siguiente:

```
INTERRUPT:    PUSH   HL ; GUARDA  REGISTROS
              PUSH   DE ; *
              PUSH   BC ; *
              PUSH   AF ; *
              EXX
              EX AF,AF'
              PUSH   HL ; y  SUS ALTERNOS
              PUSH   DE ; *
              PUSH   BC ; *
              PUSH   AF ; *
              EI        ; HABILITA INTERRUPCIONES
              IN    CAD ; LEE DEL PUERTO CAD
              LD   B,A
              IN   CAD
              LD   C,A
              LD   HL,TABLA
              LD   DE,FACTOR
              EXX
              LD   BC, NUMERO
              LD   DE, TABLA
              LD   HL,0
              LDI
              LDI
              POP  AF   ; RECUPERA REGISTROS
              POP  BC   ; ALTERNOS
POP    DE
POP    HL
EX  AF;AF'
EXX
POP  AF ;  y REGISTROS BASE
POP  BC
POP  DE
POP  HL
RET         ;  REGRESA
```

La operación de la subrutina es, en sí, irrelevante. Aquí se ilustran, básicamente, las condiciones que debe reunir una subrutina de INTERRUPCION:

1) En este tipo de interrupciones el 'CALL' (en este caso CALL INTERRUPT) se genera por 'hardware' en alguno de los tres modos del Z80, es decir, en ningún momento se presenta explícitamente (en el programa) dicho CALL. Es responsabilidad del diseñador proveer del 'hardware' respetivo.

2) Lo primero que debe hacer la rutina es guardar el 'status' del CPU en el momento de la interrupción. Eso se hace, como se ilustra con una serie de PUSH que guardan una 'fotografía' del CPU en el stack.

3) Una vez que el 'status' se ha preservado, pueden habilitarse otras interrupciones. Es decir, una interrupción puede

interrumpir a otra.

4) Al finalizar la rutina, se re-instaura el 'status'. Esto se logra con una serie de POP's en el orden inverso a los PUSH, de tal manera que, al terminar el servicio de interrupción, el CPU esté igual que al principio.

5) Se genera un REGRESO (return) que devuelve al PC su valor antes de la interrupción.

Modos de interrupción.

El Z80 tiene cuatro modos de interrupción:

a) No enmascarable (NMI)

b) Enmascarable

Modo 0.

Modo 1.

Modo 2.

Modo 0. Este modo es idéntico al (técnico) del 8080. Normalmente se utiliza la instrucción RST N y la rutina de interrupción está en la localidad N*8.

Modo 1. En este modo, el CPU responde a la interrupción efectuada en CALL a la dirección 0038H (56).

Modo 2. Este modo es el más poderoso del Z80. La dirección de llamada se forma concatenando los 8 bits del registro I (más significativos) y 7 bits del dispositivo que interrumpe, de la siguiente manera:

Dirección de apuntador:

| | | |
|---|---|---|
| REG. I | 7 BITS | 0 |

En el apuntador debe estar la dirección de la rutina de interrupción.

Tenemos el siguiente ejemplo:

(Reg. I) = 04H

7 bits del periférico: 0010 001

Concatenando: 0422H

NMI. El modo no inmascarable, como lo indica su nombre, no puede ser deshabilitado ("enmascarado") en ningún momento. La dirección de llamada es la 0066H (102).

Para entrar a modo 0,1 o 2 es necesario ejecutar alguna de las siguientes instrucciones:

```
IM  0
IM  1
IM  2
```

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# MICROCOMPUTADORA 8080 SOFTWARE

MARZO, 1979.

## 1.- ORGANIZACION DE LA COMPUTADORA

A continuación se describen las partes principales de una microcomputadora 8080. Para que un programador escriba programas eficientes es necesario que conosca cuales son éstas partes, su función y la relación que guardan entre sí para un mejor aprovechamiento de los recursos de la computadora.

Para un programador, la computadora consiste de las siguientes partes.

(1) Siete registros de trabajo en los cuales se efectúan operaciones con datos y que facilitan un medio para direccionar la memoria.

(2) Memoria, la cual almacena datos e instrucciones de un programa y que, para accesar estos, se direcciona localidad por localidad.

(3) El contador de programa, cuyo contenido indica la siguiente instrucción a ejecutarse.

(4) El " Stack Pointer ", es un registro que facilita que una porción de memoria se utilice como un "Stack". Esta función facilita el manejo de subrutinas e interrupciones que se describen posteriormente.

(5) Entrada/Salida, la cual és la interface entre un programa y el mundo exterior.

### REGISTROS DE TRABAJO

El CPU 8080 cuenta con 7 registros de 8 bits que se pueden referenciar por medio de los enteros 0, 1, 2, 3, 4, 5, 7 ; por convención, también se pueden referenciar por medio de las letras B, C, D E, H, L y A (para el acumulador) respectivamente.

Algunas de las instrucciones del 8080 hacen referencia a pares de registros, en lugar de uno solo, por medio de las letras B, D, H y PSW.

| Par de registros | registros referenciados |
|---|---|
| B | B y C (0 y 1) |
| D | D y E (2 y 3) |
| H | H y L (4 y 5) |
| PSW | se detalla à continuación |

El par de registros PSW (program status word) se refiere al registro A y à un byte especial que refleja el estado actual de las banderas de la máquina. Este byte se describe en detalle posteriormente.

## MEMORIA

El microprocesador 8080 puede utilizarse con memoria de solo lectura (ROM), memoria de solo lectura programable (EPROM) y memoria de lectura y escritura (RAM). Un programa puede causar que se lean datos de cualquier tipo de memoria, pero únicamente puede escribir en memoria RAM.

El programador debe ver la memoria como una secuencia de - bytes, cada uno de los cuales puede almacenar 8 bits. Una memoria puede consistir de hasta 65,536 bytes , los cuales pueden ser direccionados mediante los números secuenciales desde 0 a 65,535 - que es el número más grande que se puede representar con 16 bits.

## CONTADOR DE PROGRAMA (program counter)

El contador de programa es un registro de 16 bits que puede accesar el programador y cuyo contenido indica la dirección de la siguiente instrucción que será ejecutada.

## "STACK POINTER"

Un 'stack' es un área de memoria seleccionada por el programador en la cual se pueden almacenar y recuperar datos y direcciones. Las operaciones con el 'stack', efectuadas por varias instrucciones del 8080, facilitan el manejo de subrutinas e interrupciones. Todas las operaciones efectuadas sobre el 'stack' se basan en un registro especial de 16 bits llamado 'stack pointer'.

## ENTRADA / SALIDA

Para el 8080, el mundo exterior consiste de hasta 256 dispositivos deentrada y 256 de salida. Cada dispositivo se comunica - con el 8080 enviando o recibiendo bytes de datos por medio del acumulador.

## REPRESENTACION DE UN PROGRAMA EN MEMORIA

Un programa de computadora consiste de una secuencia de instrucciones. Cada instrucción efectúa una operación elemental tal - como el movimiento de un byte de datos, operaciones lógicas y aritméticas o un cambio en la secuencia de ejecución de las instrucciones.

Un programa se almacena en memoria como una secuencia de -- bits que representan las instrucciones del programa. La dirección de memoria de la siguiente instrucción a ejecutarse es manejada en el contador de programa. La ejecución de un programa se efectúa secuencialmente a menos que se ejecute una instrucción de transferencia de control, con la cual, el contenido del contador de programa será modificado con la dirección especificada. La ejecución continúa de una manera secuencial a partir de esta nueva dirección de - memoria.

Examinando el contenido de un byte de memoria, no es posible decir que el byte contiene el código de una instrucción o un dato. Por ejemplo, el código hexadecimal 1FH ha sido seleccionado para - representar la instrucción RAR (rotación del acumulador a través - del carry); entonces, el valor 1FH almacenado en un byte de memoria podría representar la instrucción RAR o bien el dato 1FH. Es responsabilidad del programador el asignarle lógica a un programa para - que en éste se distingan las instrucciones de los datos. Una ins-- trucción puede ocupar 1, 2 o 3 bytes de memoria.

## DIRECCIONAMIENTO DE MEMORIA.

Hasta aquí, se ha hecho palpable que el direccionamiento de memoria es una parte importante de cualquier programa de computadora; existen diferentes maneras de hacer esto, como se describe a continuación.

Direccionamiento directo.

Con el direccionamiento directo, una instrucción suministra la dirección exacta de memoria.

La instrucción:

"Carga el acumulador con el contenido de la localidad de memoria cuya dirección es 1E2B "

es un ejemplo de una instrucción que utiliza el direccionamiento - directo, siendo 1E2B la dirección directa.

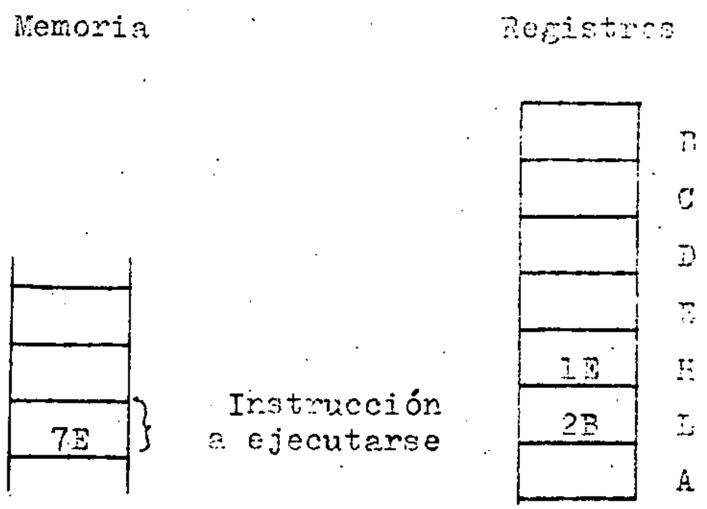
Esta instrucción aparecería en memoria de la siguiente manera:

| dirección de memoria | Memoria | |
|---|---|---|
| n | 3A | INSTRUCCION A EJECUTARSE |
| n+1 | 2B | |
| n+2 | 1E | |

La instrucción ocupa 3 bytes de memoria, el segundo y tercer byte contienen la dirección directa.

Direccionamiento utilizando un par de registros.

Una dirección de memoria puede especificarse mediante el contenido de un par de registros. La mayoría de las instrucciones que utilizan este tipo de direccionamiento, emplean los registros H y L. El registro H contiene los 8 bits más significativos de la dirección y el registro L los 8 bits menos significativos. A continuación se muestra una instrucción de un byte que carga el acumulador con el contenido de la localidad de memoria cuya dirección es 1E2B.



Además, hay dos instrucciones que pueden usar los registros B y C o el D y E para direccionar memoria (STAX y LDAX).

Direccionamiento mediante el "stack pointer"

Unicamente existen dos operaciones que pueden efectuarse en un 'stack'; a la operación de insertar datos en un 'stack' se le llama "PUSH" mientras que a la de retirar datos se le llama "POP".

Inserción en el 'stack' (PUSH)

En una inserción en el stack, se transfieren 16 bits de datos de un par de registros o los 16 bits del contador de programa a el área de memoria donde se localiza el stack. Las direcciones del área de memoria que serán accesadas durante la inserción están determinadas mediante el contenido del 'stack pointer' de la siguiente manera:

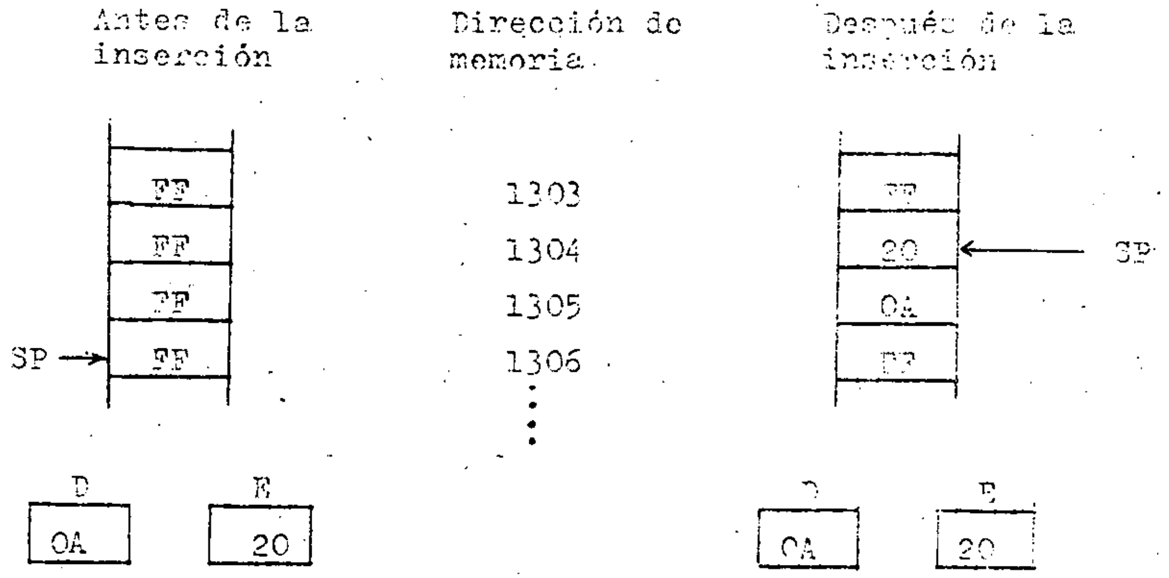
Sea SP el contenido del 'stack pointer'.

(1) Los 8 bits más significativos del dato se almacenan en la dirección de memoria SP - 1.

(2) Los 8 bits menos significativos del dato se almacenan en la dirección de memoria SP - 2.

(3) El 'stack pointer' se decrementa automáticamente en dos (SP ←-- SP - 2).

Por ejemplo, suponga que el 'stack pointer' contiene la dirección 1306H, el registro D contiene CAH y el registro E, 30H. A continuación se muestra esquemáticamente la operación de inserción del par de registro D en el 'stack':



Retiro del 'stack' (POP)

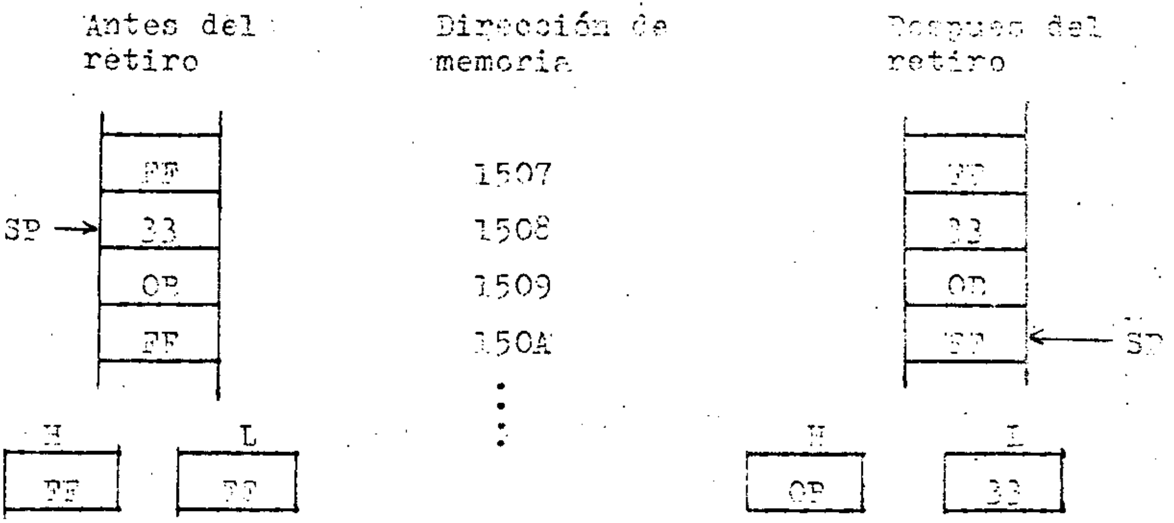
Mediante esta operación, se transfieren 16 bits del stack a un par de registros o al contador de programa. Las direcciones de memoria que serán accesadas durante esta operación están determinadas por el contenido del 'stack pointer' de la siguiente manera:

(1) El segundo registro del par o los 8 bits menos significativos del contador de programa, se cargan con el contenido de la dirección de memoria SP.

(2) El primer registro del par o los 8 bits más significativos del contador de programa, se cargan con el contenido de la dirección de memoria SP + 1.

(3) El contenido del 'stack pointer' se incrementa en dos (SP ←-- SP + 2).

Por ejemplo, suponga que el 'stack pointer' contiene ladirección 1508H y las localidades de memoria 1508H y 1509H contienen 33H y 0BH respectivamente. A continuación se muestra un retiro del stack en el par de registros H y L.



El programador debe inicializar el 'stack pointer' ante de efectuar cualquier operación con el 'stack'.

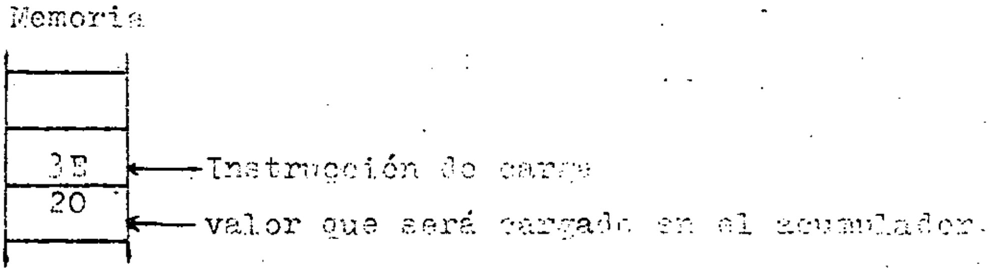
Direccionamiento inmediato.

Una instrucción que utilice este tipo de direccionamiento es aquella que contiene datos.

Una instrucción que emplea el direccionamiento inmediato es:

"Carga el acumulador con el valor 20H "

El código de la instrucción aparece en memoria como:



Como se puede observar, en este modo de direccionamiento, los datos están en el byte que sigue al código de la instrucción y no se hace referencia a memoria de una manera explícita.

BITS DE CONDICION

El 8080 cuenta con 5 bits de condición (o status) que reflejan el resultado de las operaciones con los datos. Todos estos bits, excepto uno, pueden ser analizados por instrucciones de un programa. A continuación diremos que un bit está "encendido" si su valor es 1 y "apagado" si es cero.

- Bit de carry

El bit de carry es encendido o apagado por algunas instrucciones y su status se puede analizar directamente dentro de un programa.

Las operaciones que afectan el bit de carry son la suma, resta, rotaciones y operaciones lógicas.

Por ejemplo, el carry puede ser encendido en la suma de dos números si se produce un acarreo al sumar los bits más significativos;

```
Bit No.        7 6 5 4 3 2 1 0
_______________________________
               1 0 1 0 1 1 1 0
      +        0 1 1 1 0 1 0 0
               _______________
          ___  0 0 1 0 0 0 1 0
         ↓
      acarreo     Bit de carry = 1
```

- Bit de carry auxiliar

Este bit indica que hubo un acarreo del bit 3 al bit 4. El estado de este bit no puede analizarse directamente con las instrucciones de un programa pero es utilizado para llevar a cabo la función de la instrucción DAA.

Este bit es afectado por todas las instrucciones de suma, resta, incrementos, decrementos y comparaciones.

- Bit de signo

Este bit refleja el estado del bit 7 obtenido en algún resultado. Utilizando la representación de complemento a dos, si el bit 7 es 1, el número está en el rango $-128_{10}$ a $-1$. Si el bit 7 es cero, el número está en el rango 0 a $+127_{10}$.

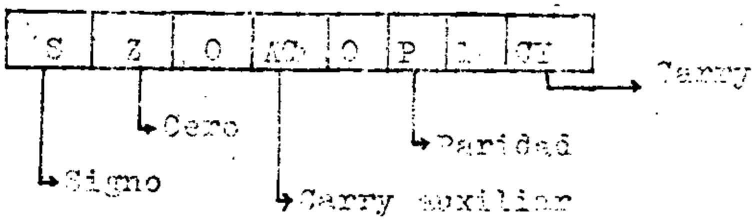
- Bit de cero

Si durante la ejecución de alguna instrucción se genera un resultado igual a cero, el bit se enciende. Si el resultado no es cero, el bit se apaga.

- Bit de paridad

Este bit se enciende cuando en un byte, resultado de una operación, el número de bits encendidos es par. De lo contrario, este bit tomará el valor de cero para indicar una paridad impar.

La palabra que contiene todos estos bits tiene la siguiente distribución:



## 3.- CONJUNTO DE INSTRUCCIONES DEL 8080

El conjunto de instrucciones del 8080 incluye 5 tipos diferentes de instrucciones:

- Grupo de transferencia de datos
- Grupo de operaciones aritméticas
- Grupo de operaciones lógicas
- Grupo de instrucciones para bifurcaciones
- Operaciones con el stack, entrada/salida y operaciones con los bits de control interno.

A continuación se describe la nomenclatura que será empleada en la descripción del conjunto de instrucciones.

| | |
|---|---|
| A | El acumulador (registro A) |
| An | El bit 'n' del contenido del acumulador, 'n' puede tomar un valor entre 0 y 7. El bit 0 es el menos significativo |
| addr | Cualquier dirección de memoria (16 bits) |
| AC | El bit de carry auxiliar |
| CY | El bit de carry |
| dato | 8 bits de datos (1 byte) |
| dato16 | 16 bits de datos (2 bytes) |
| DDD | Registro de destino en el campo de operandos |
| M | Un byte de memoria |
| P | El bit de paridad |
| PC | El contador de programa |
| PCH | Los 8 bits más significativos del contador de programa |
| PCL | Los 8 bits menos significativos del contador de programa |
| r | Un registro (B, C, D, E, H, L o A) |
| rp | Un par de registros. Los símbolos utilizados son:<br>B para los registros B y C<br>D para los registros D y E |

H para los registros H y L

SP para los 16 bits del 'stack pointer'

PSW para los bits de condición y el registro A

rp1 El primer registro del par rp

rp2 El segundo registro del par rp

S El bit de signo

SSS 3 bits que indican el registro fuente de un operando

000 para el registro B

001 " " " C

010 " " " D

011 " " " E

100 " " " H

101 " " " L

110 para referencia a memoria (M)

111 para el registro A (el acumulador)

SP Los 16 bits del 'stack pointer'

Z El bit de cero

-- Grupo de transferencia de datos

Este grupo de instrucciones transfiere datos memoria-registro y registro-registro. Los bits de condición (Z, S, P, CY, AC) no son afectados por las instrucciones de este grupo.

MOV r1, r2 (r1) ←-- (r2)

Mueve el contenido del registro r2 al registro r1.

Código: 0 1 D D D S S S

MOV r, M (r) ←-- ((H)(L))

Mueve el contenido de la localidad de memoria, cuya dirección está en los registros H y L, al registro r

Código: 0 1 D D D 1 1 0

MOV M, r ((H)(L)) ←-- (r)

Mueve el contenido del registro r a la localidad de memoria cuya dirección está en los registros H y L.

Código: 0 1 1 1 0 S S S

MVI r, dato (r) ←-- (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Mueve el contenido del segundo byte de la instrucción al registro r.

Código: (byte 1) 0 0 D D D 1 1 0
(byte 2) dato

MVI M, dato ((H)(L)) ←-- (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Mueve el contenido del segundo byte de la instrucción a la localidad de memoria cuya dirección está en los registros H y L

Código: (byte 1) 0 0 1 1 0 1 1 0
(byte 2) dato

LXI rp, dato16 (rp1) ←-- (byte 3)
(rp2) ←-- (byte 2)

Esta instrucción ocupa 3 bytes de memoria. Mueve el contenido del tercer byte de la instrucción al primer registro del par. Mueve el contenido del segundo byte de la instrucción al segundo registro del par.

Código: (byte 1) 0 0 r p 0 0 0 1
(byte 2) byte menos significativo de dato16
(byte 3) byte más significativo de dato16

donde r p es:

| | |
|---|---|
| 00 | para B y C |
| 01 | para D y E |
| 10 | para H y L |
| 11 | para los bits de condición y el registro A |

LDA addr (A) ←-- ((byte 3)(byte 2))

Carga el acumulador con el contenido de la localidad de me-

-moria cuya dirección está especificada por el contenido del byte 2 y byte 3 de la instrucción.

Código: (byte 1) 0 0 1 1 1 0 1 0
(byte 2) byte menos significativo de addr
(byte 3) byte más significativo de addr

STA addr ((byte 3)(byte 2)) ←-- (A)

Esta instrucción ocupa 3 bytes de memoria. Almacena el contenido del acumulador en la localidad de memoria cuya dirección está especificada por el contenido del byte 2 y byte 3 de la instrucción.

Código: (byte 1) 0 0 1 1 0 0 1 0
(byte 2) byte menos significativo de addr
(byte 3) byte más significativo de addr

LHLD addr (L) ←-- ((byte 3)(byte 2))
(H) ←-- ((byte 3)(byte 2) + 1)

Esta instrucción ocupa 3 bytes de memoria. Carga el registro L con el contenido de la localidad de memoria cuya dirección está especificada por el contenido del byte 2 y byte 3 de la instrucción. También, carga el registro H - con el contenido de la localidad de memoria de la siguiente dirección.

Código: (byte 1) 0 0 1 0 1 0 1 0
(byte 2) byte menos significativo de addr
(byte 3) byte más significativo de addr

SHLD addr ((byte 3)(byte 2)) ←-- (L)
((byte 3)(byte 2) + 1) ←-- (H)

Esta instrucción ocupa 3 bytes de memoria. Esta instrucción efectúa la operación inversa de la instrucción LHLD.

Código: (byte 1) 0 0 1 0 0 0 1 0
(byte 2) byte menos significativo de addr
(byte 3) byte más significativo de addr

LDAX rp                    (A) ←-- ((rp))

Esta instrucción ocupa 1 byte de memoria. Carga el acumulador con el contenido de la localidad de memoria cuya dirección está en el par de registros especificado. Unicamente se puede especificar como rp los pares de registro BC y DE.

Código: 0 0 r p 1 0 1 0

STAX rp                    ((rp)) ←-- (A)

Esta instrucción ejecuta la operación inversa de LDAX. - Tiene las mismas restricciones en la especificación de rp.

Código: 0 0 r p 0 0 1 0

XCHG                    (H) ←→ (D)
                        (L) ←→ (E)

Intercambia los contenidos de los registros H y L por el contenido de los registros D y E respectivamente.

Código: 1 1 1 0 1 0 1 1

-- Grupo de operaciones aritméticas.

Este grupo de instrucciones ejecuta operaciones aritméticas sobre datos en memoria o en registros. A menos que se indique lo contrario, todés las instrucciones de esté grupo afectan los bits de condición.

Todas la operaciones de resta se efectúan en aritmética de complemento a 2 y el estado del bit de carry indica si hubo un 'borrow' durante la operación.

ADD r                    (A) ←-- (A) + (r)

Suma el contenido del registro especificado al acumulador. El resultado queda en el acumulador.

Código: 1 0 0 0 0 S S S

Bits de condición afectados: Z,S,P,CY,AC

ADD M (A) ←--- ((H)(L)) + (A)

Suma al acumulador el contenido de la localidad de memoria cuya dirección está en los registros H y L. El resultado queda en el acumulador.

Código: 1 0 0 0 0 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

ADI dato (A) ←--- (A) + (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Suma al acumulador el contenido del segundo byte de la instrucción. El resultado queda en el acumulador.

Código: (byte 1) 1 1 0 0 0 1 1 0

(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

ADC r (A) ←--- (A) + (r) + (CY)

Suma al acumulador el contenido del registro r y el contenido del bit de carry. El resultado queda en el acumulador.

Código: 1 0 0 0 1 S S S

Bits de condición afectados: Z,S,P,CY,AC

ADC M (A) ←--- (A) + ((H)(L)) + (CY)

Suma al acumulador el contenido de la localidad de memoria, cuya dirección está en los registros H y L, y el contenido del bit de carry. El resultado queda en el acumulador.

Código: 1 0 0 0 1 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

ACI dato (A) ←--- (A) + (byte 2) + (CY)

Esta instrucción ocupa 2 bytes de memoria. Suma al acumulador el contenido del segundo byte de la instrucción y el contenido del bit de carry. El resultado queda en el acumulador.

Código: (byte 1) 1 1 0 0 1 1 1 0

(byte 2) dato

Bits de condición afectados: Z;S,P,CY,AC

SUB r (A) ←-- (A) - (r)

Resta al acumulador el contenido del registro r. El resultado queda en el acumulador.

Código: 1 0 0 1 0 S S S

Bits de condición afectados: Z,S,P,CY,AC

SUB M (A) ←-- (A) - ((H)(L))

Resta al acumulador el contenido de la localidad de memoria cuya dirección está en los registros H y L. El resultado queda en el acumulador.

Código: 1 0 0 1 0 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

SUI dato (A) ←-- (A) - (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Resta al acumulador el contenido del segundo byte de la instrucción. El resultado queda en el acumulador.

Código: (byte 1) 1 1 0 1 0 1 1 0

(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

SBB r (A) ←-- (A) - (r) - (CY)

Resta al acumulador el contenido del registro r y el contenido del bit de carry. El resultado queda en el acumulador.

Código: 1 0 0 1 1 S S S

Bits de condición afectados: Z,S,P,CY,AC

SBB M (A) ←-- (A) - ((H)(L)) - (CY)

Resta al acumulador el contenido de la localidad de memoria, cuya dirección está en los registros H y L, y el contenido del bit de carry. El resultado queda en el acumulador.

Código: 1 0 0 1 1 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

SBI dato (A) ←-- (A) - (byte 2) - (CY)

Esta instrucción ocupa 2 bytes de memoria. Resta al acumulador el contenido del segundo byte de la instrucción y el contenido del bit de carry. El resultado queda en el acumulador.

Código: (byte 1) 1 1 0 1 1 1 1 0
(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

INR r (r) ←-- (r) + 1

Incrementa en uno el contenido del registro r. Esta instrucción no afecta el bit de carry.

Código: 0 0 D D D 1 0 0

Bits de condición afectados: Z,S,P,AC

INR M ((H)(L)) ←-- ((H)(L)) + 1

Incrementa en uno el contenido de la localidad de memoria cuya dirección está en los registros H y L. Esta instrucción no afecta el bit de carry.

Código: 0 0 1 1 0 1 0 0

Bits de condición afectados: Z,S,P,AC

DCR r (r) ←-- (r) - 1

Decrementa en uno el contenido del registro r. Esta instrucción no afecta el bit de carry.

Código: 0 0 D D D 1 0 1

Bits de condición afectados: Z,S,P,AC

DCR M ((H)(L)) ←-- ((H)(L)) - 1

Decrementa en uno el contenido de la localidad de memoria cuya dirección está en los registros H y L. Esta instrucción no afecta el bit de carry.

Código: 0 0 1 1 0 1 0 1

Bits de condición afectados: Z,S,P,AC

INX rp (rp1)(rp2) ←-- (rp1)(rp2) + 1

Incrementa en uno el contenido del par de registros rp.

Código: 0 0 r p 0 0 1 1

Bits de condición afectados: ninguno

DCX rp (rp1)(rp2) ←-- (rp1)(rp2) - 1

Decrementa en uno el contenido del par de registros rp.

Código: 0 0 r p 1 0 1 1

Bits de condición afectados: ninguno

DAD rp (H)(L) ←-- (H)(L) + (rp1)(rp2)

Suma los 16 bits del par de registros HL con los 16 bits del par de registros rp. El resultado queda en el par de registros HL. Esta instrucción afecta únicamente el bit de carry.

Código: 0 0 r p 1 0 0 1

Bits de condición afectados: CY

DAA (Ajuste decimal del acumulador)

Ajusta el número contenido en el acumulador para formar 2 dígitos BCD mediante el siguiente proceso:

1.- Si el valor de los 4 bits menos significativos del acumulador es mayor que 9 o el bit de carry auxiliar es 1, se le suma 6 al acumulador.

2.- Si el valor de los 4 bits más significativos del acumulador es mayor que 9 o el bit de carry es 1, se le suma 6 a los 4 bits más significativos del acumulador. Todos los bits de condición son afectados.

Código: 0 0 1 0 0 1 1 1

-- Grupo de operaciones lógicas

Este grupo de instrucciones efectúan operaciones lógicas - sobre datos en memoria o en registros y con los bits de condición.

ANA r $(A) \leftarrow (A)\wedge(r)$

Se efectúa el AND lógico entre el contenido del acumulador y el contenido del registro r. El resultado queda en el acumulador y CY = 0.

Código: 1 0 1 0 0 S S S

Bits de condición afectados: Z,S,P,CY,AC

ANA M $(A) \leftarrow (A)\wedge((H)(L))$

Se efectúa el AND lógico entre el contenido del acumulador y el contenido de la localidad de memoria cuya dirección está en los registros H y L. El resultado queda en el acumulador y CY =0.

Código: 1 0 1 0 0 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

ANI dato $(A) \leftarrow (A)\wedge(\text{byte }2)$

Esta instrucción ocupa 2 bytes de memoria. Se efectúa el AND lógico entre el contenido del acumulador y el contenido del segundo byte de la instrucción. El resultado queda en el acumulador, CY =0 y AC =0.

Código: (byte 1) 1 1 1 0 0 1 1 0
(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

XRA r $(A) \leftarrow (A) \veebar (r)$

Se efectúa el OR exclusivo entre el contenido del acumulador y el contenido del registro r. El resultado queda en el acumulador, CY =0 y AC =0.

Código: 1 0 1 0 1 S S S

Bits de condición afectados: Z,S,P,CY,AC

XRA M  (A) ←-- (A) ⊻ ((H)(L))

Se efectúa el OR exclusivo entre el contenido del acumulador y el contenido de la localidad de memoria cuya dirección está en los registros H y L. El resultado queda en el acumulador, CY =0 y AC =0.

Código: 1 0 1 0 1 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

XRI dato  (A) ←-- (A) ⊻ (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Se efectúa el OR exclusivo entre el contenido del acumulador y el contenido del segundo byte de la instrucción. El resultado queda en el acumulador, CY =0 y AC =0.

Código: (byte 1) 1 1 1 0 1 1 1 0
(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

ORA r  (A) ←-- (A) V (r)

Se efectúa el OR inclusivo entre el contenido del acumulador y el contenido del registro r. El resultado queda en el acumulador, CY =0 y AC =0.

Código: 1 0 1 1 0 S S S

Bits de condición afectados: Z,S,P,CY,AC

ORA M  (A) ←-- (A) V ((H)(L))

Se efectúa el OR inclusivo entre el contenido del acumulador y el contenido de la localidad de memoria cuya dirección está en los registros H y L. El resultado queda en el acumulador, CY =0 y AC =0.

Código: 1 0 1 1 0 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

ORI dato                (A) ←-- (A) V (byte 2)

Esta instrucción ocupa 2 bytes de memoria. Se efectúa el OR inclusivo entre el contenido del acumulador y el contenido del segundo byte de la instrucción. El resultado queda en el acumulador, CY =0 y AC =0.

Código: (byte 1) 1 1 1 1 0 1 1 0
(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

CMP r                (A) - (r)

Se resta el contenido del registro r al contenido del acumulador. El contenido del acumulador no se altera pero los bits de condición se encienden o apagan de acuerdo al resultado de la resta. Z =1 si (A) = (r). CY =1 si (A) (r).

Código: 1 0 1 1 1 S S S

Bits de condición afectados: Z,S,P,CY,AC

CMP M                (A) - ((H)(L))

Se resta el contenido de la localidad de memoria, cuya dirección está en los registros H y L, al contenido del acumulador. El contenido del acumulador no se altera pero los bits de condición se encienden o apagan de acuerdo al resultado de la resta. Z =1 si (A) = ((H)(L)) y CY =1 si (A) ((H)(L)).

Código: 1 0 1 1 1 1 1 0

Bits de condición afectados: Z,S,P,CY,AC

CPI dato                (A) - (byte 2)

Se resta el contenido del segundo byte de la instrucción del contenido del acumulador. Z =1 si (A) = (byte 2) y CY =1 si (A) (byte 2).

Código: (byte 1) 1 1 1 1 1 1 1 0
(byte 2) dato

Bits de condición afectados: Z,S,P,CY,AC

RLC $(A_{n+1}) \longleftarrow (A_n)$ ; $(A_0) \longleftarrow (A_7)$

$(CY) \longleftarrow (A_7)$

Se efectúa una rotación a la izquierda, de una posición, con el contenido del acumulador. El bit menos significativo del acumulador y el bit de carry tomarán el valor que tenía el bit más significativo del acumulador antes del corrimiento.

Código: 0 0 0 0 0 1 1 1

Bits de condición afectados: CY

RRC $(A_n) \longleftarrow (A_{n+1})$ ; $(A_7) \longleftarrow (A_0)$

$(CY) \longleftarrow (A_0)$

Se efectúa una rotación a la derecha, de una posición, con el contenido del acumulador. El bit más significativo del acumulador y el bit de carry tomarán el valor que tenía el bit menos significativo del acumulador antes del corrimiento.

Código: 0 0 0 0 1 1 1 1

Bits de condición afectados: CY

RAL $(A_{n+1}) \longleftarrow (A_n)$ ; $(CY) \longleftarrow (A_7)$

$(A_0) \longleftarrow (CY)$

Se efectúa una rotación a la izquierda, de una posición, con el contenido del acumulador e incluyendo el bit de carry. El valor del carry pasa al bit $A_0$ y el bit $A_7$ pasa al bit de carry.

Código: 0 0 0 1 0 1 1 1

Bits de condición afectados: CY

RAR $(A_n) \longleftarrow (A_{n+1})$ ; $(CY) \longleftarrow (A_0)$

$(A_7) \longleftarrow (CY)$

Se efectúa una rotación a la derecha, de una posición, con el contenido del acumulador e incluyendo el bit de carry. El valor del carry pasa al bit $A_7$ y el bit $A_0$ pasa al bit de carry. Unicamente se afecta el bit de carry.

Código: 0 0 0 1 1 1 1 1

CMA $(A) \leftarrow (\overline{A})$

Complementa el contenido del acumulador, es decir, los 1 se transforman a 0 y los 0 se transforman a 1.

Código: 0 0 1 0 1 1 1 1

Bits de condición afectados: ninguno

CMC $(CY) \leftarrow (\overline{CY})$

Complementa el contenido del bit de carry.

Código: 0 0 1 1 1 1 1 1

Bits de condición afectados: CY

STC $(CY) \leftarrow 1$

Asigna el valor 1 al bit de carry (enciende CY).

Código: 0 0 1 1 0 1 1 1

Bits de condición afectados: CY

-- Grupo de instrucciones para bifurcación.

Este grupo de instrucciones altera el flujo secuencial (de ejecución) de un programa. Existen dos tipos de transferencia de control, incondicional y condicional. Las instrucciones de transferencia incondicional efectúan la operación especificada sobre el registro PC (el contador de programa). Las instrucciones de transferencia de control condicional examinan el contenido de los bits de condición para determinar si se efectúa la bifurcación especificada. Las condiciones que pueden ser utilizadas son:

| CONDICION | CCC |
|---|---|
| NZ - no cero (Z=0) | 000 |
| Z - cero (Z=1) | 001 |
| NC - no carry (CY=0) | 010 |
| C - carry (CY=1) | 011 |
| PO - paridad impar(P=0) | 100 |
| PE - paridad par (P=1) | 101 |
| P - positivo (S=0) | 110 |
| M - negativo (S=1) | 111 |

JMP addr (PC) ←-- (byte 3)(byte 2)

El control del programa se transfiere incondicionalmente a la dirección especificada por el segundo y tercer byte de la instrucción.

Código: (byte 1) 1 1 0 0 0 0 1 1
(byte 2) byte menos significativo de la dirección
(byte 3) byte más significativo de la dirección

Bits de condición afectados: ninguno

Jcondición addr Si (CCC), (PC) ←-- (byte 3)(byte 2)

Si se cumple la condición especificada, el control del programa se transfiere a la instrucción cuya dirección se especifica en el segundo y tercer byte de la instrucción. De lo contrario, el control del programa continúa secuencialmente.

Código: (byte 1) 1 1 C C C 0 1 0
(byte 2) byte menos significativo do la dirección
(byte 3) byte más significativo de la dirección

Bits de condición afectados: ninguno

CALL addr
((SP) - 1) ←-- (PCH)
((SP) - 2) ←-- (PCL)
(SP) ←-- (SP) - 2
(PC) ←-- (byte 3)(byte 2)

Esta instrucción se utiliza para ejecutar una transferencia de control incondicional a una subrutina. La dirección de la subrutina se encuentra en el segundo y tercer byte de la instrucción. Antes del salto efectivo a la subrutina, se guarda en el stack el contenido del contador de programa para salvar así la dirección de retorno.

Código: (byte 1) 1 1 0 0 1 1 0 1
(byte 2) byte menos significativo de la dirección
(byte 3) byte más significativo de la dirección

Bits de condición afectados: ninguno

Ccondición addr    Si (CCC), ((SP)-1) ←-- (PCH)
((SP)-2) ←-- (PCL)
(SP) ←-- (SP) - 2
(PC) ←-- (byte 3)(byte 2)

Si se cumple la condición, se efectúa la misma operación de la instrucción CALL. De lo contrario, el control del programa continúa secuencialmente.

Código: (byte 1) 1 1 C C C 1 0 0
(byte 2) byte menos significativo de la dirección
(byte 3) byte más significativo de la dirección

Bits de condición afectados: ninguno

RET    (PCL) ←-- ((SP))
(PCH) ←-- ((SP) + 1)
(SP) ←-- (SP) + 2

Se retiran 2 bytes del tope del stack y se cargan en el contador de programa. El contenido del registro SP se incrementa en dos. Esta instrucción se emplea para realizar un retorno desde una subrutina hacia la parte del programa de donde fue llamada.

Código: 1 1 0 0 1 0 0 1

Bits de condición afectados: ninguno

Rcondición    Si (CCC), (PCL) ←-- ((SP))
(PCH) ←-- ((SP) + 1)
(SP) ←-- (SP) + 2

Si se cumple la condición, se efectúa la misma operación de la instrucción RET. De lo contrario, el control del programa continúa secuencialmente.

Código: 1 1 C C C 0 0 0

Bits de condición afectados: ninguno

RST n

((SP) - 1) ←--- (PCH)
((SP) - 2) ←--- (PCL)
(SP) ←--- (SP) - 2
(PC) ←--- 8 ∗(NNN)

Al ejecutarse esta instrucción, se inserta en el stack el contenido del contador de programa. El registro SP se decrementa en dos. El control se transfiere a la instrucción cuya dirección es 8 veces el valor de NNN (RST 0 a RST 7).

Código: 1 1 N N N 1 1 1

Bits de condición afectados: ninguno

Después de ejecutarse esta instrucción, el contenido del contador de programa es: 0000000000NNN000

PCHL

(PCH) ←--- (H)
(PCL) ←--- (L)

El contenido del registro H se transfiere al byte más significativo del contador de programa. El contenido del registro L se transfiere al byte menos significativo del contador de programa.

Código: 1 1 1 0 1 0 0 1

Bits de condición afectados: ninguno

-- Operaciones con el stack, entrada/salida y operaciones con los bits de control interno.

PUSH rp

((SP) - 1) ←--- (rp1)
((SP) - 2) ←--- (rp2)
(SP) ←--- (SP) - 2.

Inserta en el stack el contenido del par de registros rp. El registro SP se decrementa en dos. Si rp=PSW, (PUSH PSW) se inserta en el stack el contenido del acumulador y los bits de condición.

Código: 1 1 r p 0 1 0 1

Bits de condición afectados: ninguno

EI

Esta instrucción habilita el mecanismo de interrupciones del CPU para que éste acepte interrupciones.

Código: 1 1 1 1 1 0 1 1

Bits de condición afectados: ninguno

DI

Esta instrucción deshabilita el mecanismo de interrupciones del CPU para que éste no acepte interrupciones.

Código: 1 1 1 1 0 0 1 1

Bits de condición afectados: ninguno

HLT

Provoca que el procesador entre al estado de WAIT.

Código: 0 1 1 1 0 1 1 0

Bits de condición afectados: ninguno

NOP

No se efectúa ninguna operación. Tanto los registros como los bits de condición permanecen sin alterarse.

Código: 0 0 0 0 0 0 0 0

# 1.- INTRODUCCION.

## 1.1. ¿ Que es PL/M ?

PL/M es un lenguaje de programación de alto nivel diseñado especialmente para simplificar la tarea de programación de sistemas, para la familia de microcomputadoras que se basan en los microprocesadores 8008 y 8080.

PL/M puede ser utilizado por los diseñadores e implementadores de los sistemas de microcomputadoras como una buena herramienta de software, ya que le dá al programador el suficiente control del procesador como para satisfacer sus necesidades en la programación de sistemas.

PL/M ha sido diseñado para facilitar el uso de las técnicas modernas en programación estructurada. El empleo de estas técnicas permite escribir programas en forma más ráoida, muy legibles y muy fáciles de modificar o depurar.

## 2. COMPONENTES BASICOS DE UN PROGRAMA PL/M.

Los programas PL/M se escriben en formato libre, es decir, las lineas de entrada son independientes de las columnas y se pueden insertar espacios libremente entre los elementos del programa.

### 2.1. Caracteres de PL/M.

Los caracteres válidos en PL/M es un subconjunto de los caracteres ASCII yEBCDIC. Los caracteres válidos en PL/M son los alfanuméricos

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

y los caracteres especiales

$ = . / ( ) + - ' , : ; * < >

cualquier otro caracter no es válido en PL/M. Los caracteres que no pertenecen al conjunto anterior son reemplazados por el caracter "blanco" o espacio.

### 2.2. Identificadores y palabras reservadas.

Los identificadores se utilizan en PL/M para darles - nombre a las variables, "procedures", macros y etiquetas. Un identificador puede formarse hasta con 31 caracteres, de los cuales, el primero debe ser alfabético y el resto, alfabéticos o numéricos. El caracter '$' puede intercalarse en un - identificador para darle mas legibilidad, pero son ignorados por el compilador PL/M. Un identificador que contiene caracteres '$' es equivalente al mismo identificador sin estos - caracteres. Por ejemplo:

```
X
XCERO
PUERTO$DE$SALIDA
CONTADOR$EVENTO1
CONTADOREVENTO1
```

son identificadores válidos y los dos últimos serán considerados por el compilador PL/M como uno solo.

Hay otros identificadores válidos cuyo significado es fijo ya que son parte del lenguaje PL/M y que no pueden ser utilizados como identificadores definidos por el programador. Una lista de las palabras reservadas en PL/M se puede observar en el apéndice A.

Se puede insertar cualquier número de espacios entre identificadores, palabras reservadas y caracteres especiales. Por ejemplo:

```
X  *  X +  ( 2 * X ) -   1
```

es equivalente a

```
X*X+(2*X)-1
```

## 2.3. Comentarios

En un programa PL/M se pueden insertar comentarios con la finalidad de hacerlos más legibles y como documentación de los mismos. Un comentario se inicia con el par de caracteres /* y es terminado por el par de caracteres */. Los comentarios pueden colocarse en cualquier parte de un programa y el texto de los mismos es ignorado por el compilador.

## 3. TIPOS DE DATOS.

Los datos en PL/M pueden ser vàriables o constantes. Las variables son identificadores cuyo valor puede ser cambiado durante la ejecución del programa mientras que las constantes tienen valores fijos. La expresión

PILA + (X + 4)/ TOL

involucra las variables PILA, X y TOL, y la constante 4.

### 3.1. Constantes numéricas.

Una constante es un valor que no cambia durante la ejecución de un programa. Una constante puede ser un número o una cadena de caracteres. Las constantes numéricas pueden expresarse en las bases binario, octal, decimal y hexadecimal.

La base de los números se representa por una de las siguientes letras a continuación del número.

| | |
|---|---|
| B - | binario |
| O,Q - | octal |
| D - | decimal |
| H - | hexadecimal |

El primer caracter de un número hexadecimal debe ser un dígito numérico. Cualquier número que no esté seguido por una de las letras B,O,Q,D o H es concíderado como decimal. Las siguientes constantes son válidas en PL/M:

5 47Q 1Ø11B 3A3H 57D 57 ØBCH

### 3.2. Cadenas de caracteres constantes .

Estas constantes se escriben entre dos apóstrofes. Por ejemplo:

'A' 'BD' 'C7' 'TR'

El compilador de PL/M transforma los caracteres a sus respectivos códigos ASCII, cada caracter ocupa un byte de memoria ya que su código es de 7 bits con el bit más significativo puesto a cero. Las cadenas de longitud uno se traducen a valores de un solo byte mientras que los de longitud 2, se traducen a válores de doble byte. Por ejemplo:

```
'A'  es equivalente a   41H
'AB' es equivalente a   4142H
```

### 3.3. Variables y declaraciones de tipo

Todas las variables utilizadas en un programa PL/M - deben ser declaradas antes que se utilicen en alguna expresión. La declaración define a la variable y sus características.

Una variable puede ser de uno de dos tipos: tipo -- BYTE o tipo ADDRESS. Los datos tipo BYTE ocupan un byte de memoria (8 bits) y los datos tipo ADDRESS dos bytes (16 bits) en el código objeto generado por el compilador.

La declaración de una variable o una lista de variables entre paréntesis empieza con la palabra reservada DECLARE, - la(s) variable(s) y una de las dos palabras reservadas BYTE o ADDRESS. Por ejemplo:

```
DECLARE  X   BYTE ;
DECLARE (Q,R,S) BYTE;
DECLARE (J,T) ADDRESS;
```

son declaraciones válidas.

#### 3.3.1. Arreglos.

Los arreglos que se pueden utilizar en un programa PL/M son, únicamente, de una dimensión que comunmente se conocen - como vectores o variables con subíndice.

La declaración de los arreglos se hace de la misma manera que para las variables, agregando únicamente la dimensión del arreglo. Por ejemplo:

```
            DECLARE  X (100) BYTE ;
```

causa que el identificador X sea asociado con 100 datos tipo BYTE y que pueden ser referenciados de la siguiente manera;

```
  X(0),  X(1), X(2),  ...  , X(98),  X(99)
```

observe que los subíndices comienzan desde cero.

Una lista de arreglos de la misma dimensión puede ser declarada en una sola proposición. Por ejemplo:

```
      DECLARE (A,B,C) (50) ADDRESS;
      DECLARE (X,Y)(20) BYTE, Z(55) ADDRESS ;
```

Los subíndices utilizados para referenciar un elemento de un arreglo,no únicamente pueden ser constantes numéricas, sino que pueden ser cualquier expresión válida de PL/M. Por ejemplo:

```
        X(I) ,   Y(J + 3),   Z(T - 1)
```

### 3.3.2. Atributo INITIAL

Mediante el atributo INITIAL,se puede asignar un valor inicial a las variables en la proposición de declaración. Su forma general es:

```
              INITIAL (lista de constantes)
```

donde 'lista de constantes' es un conjunto de constantes separadas por comas. En una proposición de declaración, este atributo debe escribirse después del atributo de tipo (BYTE o ADDRESS).

Las siguientes, son declaraciones válidas:

```
      DECLARE  T  BYTE  INITIAL (20);
      DECLARE VEC(10) BYTE INITIAL (1,2,3,4,5,6,7,8,9,10);
      DECLARE Z(100) BYTE INITIAL ('ABC','TTY',0FH);
      DECLARE X(100) ADDRESS INITIAL (11101B, 15Q, 0ABH);
      DECLARE (Q, R, S) BYTE  INITIAL (3, 2, 10);
```

El número de bytes requeridos para manejar la lista de constantes, no necesariamente debe coincidir con la longitud declarada de las variables. Es incorrecto especificar atributos INITIAL que se traslapen con otros.

### 3.3.3. Declaración DATA

Suponga que usted desea declarar un arreglo, darle valores iniciales y que estos valores no cambien con la ejecución del programa. Si su sistema tiene diferentes tipos de memoria para almacenar datos y programas, la respuesta es, almacenar el arreglo - junto con el código del programa en memoria de solo lectura (ROM) y no junto a las variables que deberán estar em memoria de lectura-escritura.

PL/M brinda este tipo de control sobre la asignación de memoria con la declaración DATA. Su forma es:

```
DECLARE  identificador  DATA (lista de constantes);
```

un ejemplo de esta construcción es:

```
DECLARE  TITULO  DATA ('8080 PL/M ');
```

El efecto de una declaración DATA es similar al de la declaración de un arreglo con un atributo INITIAL, solo que en esta construcción no se especifica el tipo de datos ya que siempre se tomará como tipo BYTE. Además, no se requiere especificar la longitud del arreglo ya que ésta queda especificada implícitamente con la longitud de la lista de constantes. El 'identificador' de la declaración DATA puede utilizarse en el programa como un arreglo tipo - BYTE, con una excepción: no debe aparecer en el miembro izquierdo de una proposición de asignación.

## 4.- EXPRESIONES Y PROPOSICIONES DE ASIGNACION.

Una expresión PL/M consiste de variables y/o constantes combinadas mediante el uso de operadores aritméticos, lógicos o relacionales de acuerdo con la notación algebraica. Por ejemplo:

A+B

A + B - C

A - B + C/D

### 4.1. Operadores aritméticos.

En PL/M se cuenta con siete operadores aritméticos. Estos son:

| | | |
|---|---|---|
| + | ------- | Suma |
| - | ------- | Resta |
| PLUS | ------- | Suma con carry |
| MINUS | ------- | Resta con carry (borrow) |
| * | ------- | Multiplicación |
| / | ------- | División |
| MOD | ------- | Módulo |

Todos los operadores aritméticos anteriores, ejecutan aritmética binaria sin signo sobre cualquier valor tipo BYTE o ADDRESS.

Los operadores + y - ejecutan la adición y substracción respectivamente. Si ambos operandos son de tipo BYTE, la operación se efectúa en aritmética de 8 bits y el resultado es de tipo BYTE. Si un operando es de tipo ADDRESS y el otro de tipo BYTE, este último será extendido a 16 bits con el byte más significativo puesto a ceros y la operación se efectuará en aritmética de 16 bits dando como resultado un valor de tipo ADDRESS.

Los operadores * y / efectúan multiplicación y división binaria sin signo sobre operandos de tipo BYTE o ADDRESS. El resultado será siempre de tipo ADDRESS. En la división, el resultado siempre se redondea al entero próximo inferior y el resultado de la división por cero es indefinido.

El operador MOD efectúa una operación similar a /, excepto que el resultado de la operación no es el cociente de la división sino el residuo.

### 4.2. Operadores lógicos.

En PL/M se tienen cuatro operadores lógicos (booleanos).

NOT AND OR XOR

Estos operadores efectúan operaciones lógicas sobre 8 o 16 bits en paralelo. NOT es un operador unario ya que únicamente actúa sobre un operando.

El operador NOT produce un resultado en el cual cada bit es el complemento de el bit correspondiente del operando. El resto de los operadores son binarios ya que actúan sobre dos operandos. Si ambos operandos son de tipo BYTE, la operación se efectúa sobre 8 bits y el resultado será de tipo BYTE. Si un operando es de tipo ADDRESS y el otro de tipo BYTE, este último será extendido a 16 bits con el byte más significativo puesto a ceros y la operación se efectúa sobre 16 bits dando como resultado un valor de tipo ADDRESS. Ejemplos de estos operadores son:

| | | | | |
|---|---|---|---|---|
| | NOT | 11100101B | ---------- | 00011010B |
| 10101010B | AND | 11001100B | ---------- | 10001000B |
| 10101010B | OR | 11001100B | ---------- | 11101110B |
| 10101010B | XOR | 11001100B | ---------- | 01100110B |

### 4.3. Operadores relacionales.

Los operadores relacionales se utilizan en PL/M para comparar valores. Ellos son:

| | |
|---|---|
| < | menor que |
| > | mayor que |
| <= | menor o igual que |
| >= | mayor o igual que |
| <> | diferente de |
| = | igual a |

Todos los operadores relacionales son binarios. Los operandos pueden ser de tipo BYTE o ADDRESS. La comparación se efectúa suponiendo siempre que los operandos son enteros binarios sin signo. Si la relación especificada entre los operandos se cumple, el resultado es el valor 0FFH; de lo contrario, el resultado es cero. En todos los casos el resultado es de tipo BYTE, con todos los bits a "1" para una condición de Cierto y a " 0 " para una condición de Falso. Por ejemplo:

| | |
|---|---|
| ( 5 < 6 ) | Resultado= 11111111B |
| ( 7 <= 3 ) | Resultado= 00000000B |
| (6 > 5 ) OR (9 > 12) | Resultado= 0FFH |
| (4 > (3 + 5)) AND (2 > 1) | Resultado= 00H |

### 4.4. Evaluación de expresiones.

Los operadores de PL/M tienen implicada una precedencia la cual es utilizada para determinar la manera en que serán agrupados los operadores y operandos para evaluar una expresión. Por ejemplo, A + B * C causa que A sea sumada al producto de B y C. A continuación se listan los operadores de PL/M empezando con los de precedencia más alta a la más baja. Los operadores que aparecen en la misma linea, tienen la misma precedencia.

```
        *     /    MOD
   +    -   PLUS   MINUS
   <   <=   <>   =   >=   >
            NOT
            AND
        OR    XOR
```

Se pueden emplear paréntesis para modificar el orden de evaluación de una expresión que el especificado por la precedencia de los operadores. La expresión ( A + B ) * C causará que la suma de A y B sea multiplicada por C. Por ejemplo:

A + B + C + D      es equivalente a  ((A + B) + C) + D

A + B * C          es equivalente a  A + (B * C)

A + B - C * D      es equivalente a  (A + B) - (C * D)

### 4.5. Proposiciones de asignación.

Las proposiciones de asignación de PL/M reespecifican los valores de las variables. Su forma es:

variable = expresión ;

por ejemplo

A = B + C/D ;

RESIDUO = Z2 MOD 4 ;

La precisión declarada (BYTE o ADDRESS) de la variable asignada afecta la operación de almacenamiento del valor resultante en la expresión. Si la variable que recibe el valor es de tipo BYTE y el resultado de la expresión es de tipo ADDRESS, se omite el byte más significativo del resultado en la operación de almacenamiento. Similarmente, si el resultado de la expresión es de tipo BYTE y la variable que recibe el resultado es de tipo ADDRESS, el byte más significativo de la variable es llenado con ceros.

A menudo es conveniente asignar la misma expresión a más de una variable. Esto se logra en PL/M listando todas las variables a la izquierda del signo de igual separadas por comas. Por ejemplo, la expresión

A,B,C = X/Y

hace que A, B y C tomen el valor de X/Y.

## 5.- GRUPOS DO

Se puede agrupar varias proposiciones PL/M utilizando las palabras reservadas DO y END para formar un grupo DO. El grupo DO más simple es de la forma

```
DO;
   proposición-1 ;
      . . .
   proposición-n  ;
END;
```

Un grupo de proposiciones (grupo DO) se considera como una sola proposición PL/M y puede aparecer en cualquier parte de un programa donde pueda aparecer una proposición simple.

El control del flujo de un programa se logra mediante el uso de otras formas de grupos DO; éstos se explican a continuación.

### 5.1. Grupo DO - WHILE

El grupo DO - WHILE tiene la forma general

```
DO  WHILE  expresión ;
    proposición-1 ;
    proposición-2 ;
       . . .
    proposición-n
END;
```

El efecto de esta proposición es: primero se evalúa la expresión que sigue a la palabra reservada WHILE. Si el resultado es una cantidad en la cual el bit de más a la derecha es 1, entonces se ejecutan todas las proposiciones hasta el END. Cuando el END es alcanzado, la expresión se vuelve a evaluar. El grupo se ejecuta una y otra vez hasta que la expresión dé un resultado en el cual el bit de más a la derecha (menos significativo) es cero. En este momento, el grupo de proposiciones es saltado y el control del -

programa pasa fuera del grupo. Por ejemplo:

```
          I = 1;
          DO  WHILE  I <= 10;
              I = I + 1;
          END;
```

la proposición I = I + 1 será ejecutada 10 veces. El valor de I, cuando el control del programa sale del grupo, será de 11.

## 5.2. Grupo iterativo DO

Un grupo iterativo DO logra que se ejecute un grupo de proposiciones un número fijo de veces. Su forma general más simple es:

```
     DO  var = expr-1  TO  expr-2 ;
         proposición-1;
          . . .
         proposición-n;
     END;
```

donde 'var' es el nombre de una variable y 'expr-1' y 'expr-2' - son expresiones válidas en PL/M. El efecto de este grupo es el siguiente: primero se almacena el valor de 'expr-1' en la variable 'var'. Segundo, el valor de la variable 'var' es comparado y si es menor o igual que 'expr-2', se ejecutan las proposiciones agrupadas. Cuando el END es alcanzado, la variable se incrementa en 1 y se repite la comparación. El grupo se ejecuta en forma repetida - hasta que el valor de la variable sea mayor que 'expr-2'. Cuando ocurre esta última condición, el control del programa salta el - grupo y pasa fuera del rango del grupo DO. Un ejemplo de este grupo es:

```
    DO  I = 1  TO  10;
        A = A + I;
    END;
```

Este grupo iterativo DO tiene el mismo efecto que el siguiente DO - WHILE.

```
I = 1;
DO  WHILE I <= 10 ;
   A = A + I ;
   I = I + 1 ;
END;
```

Una forma más general del grupo iterativo DO permite que se trabaje con un incremento diferente de 1. Esta forma más general - es:

```
DO  var = expr-1   TO  expr-2   BY  expr-3 ;
    proposición-1;
       . . .
    proposición-n;
END;
```

En este caso, la variable 'var' se incrementa con el valor de -- 'expr-3' en lugar de 1 cada vez que se alcance el END. A continuación se muestra un ejemplo empleando esta forma.

```
/* CALCULO DEL PRODUCTO DE LOS PRIMEROS N
   NUMEROS PARES                                      */
     PRODUCTO = 1;
     DO I = 2  TO (2*N)  BY 2;
        PRODUCTO = PRODUCTO * I;
     END;
```

## 5.3. Grupo DO - CASE.

Una última forma de los grupos DO es el grupo DO - CASE. Su forma general es:

```
DO  CASE  expresión;
    proposición-1;
    proposición-2;
       . . .
    proposición-n;
END;
```

El efecto de este grupo es el siguiente: se evalúa la expresión que sigue a la palabra reservada CASE. El resultado de la expresión es un valor K el cual debe estar en el rango de 0 a n-1. El valor K es utilizado para seleccionar una de las n proposiciones que forman el grupo y la cual será ejecutada. El primer caso (proposición-1) corresponde a K = 0, el segundo caso (proposición-2) corresponde a K = 1, y así sucesivamente. Después de que una proposición del grupo ha sido seleccionada y ejecutada, el control pasa a la proposición que sigue al END de el grupo DO-CASE. Si el valor de K es mayor que el número de proposiciones, el resultado es indefinido. Un ejemplo del grupo DO-CASE es:

```
DO  CASE  L;
   ;            /*  L = 0   */
 L = L + I;     /*  L = 1   */
 DO;            /*  L = 2   */
   X = X + 10;
   Z = X - 3;
 END;
 DO WHILE X <= 6; /*  L = 3    */
    X = X + 2;
 END;
END;  /*   FIN DEL GRUPO DO-CASE   */
```

Este ejemplo ilustra el uso de los grupos DO para agrupar varias proposiciones como una sola.

## 6.- PROPOSICION IF

La proposición IF tiene la siguiente forma general:

```
IF  condición  THEN  proposición-1 ;
ELSE  proposición-2 ;
```

esta proposición tiene el siguiente efecto: se evalúa la 'condición' que sigue a la palabra reservada IF. Si el resultado de la condición es VERDADERO, se ejecuta la proposición-1; si el resultado es FALSO se ejecuta la proposición-2. Dependiendo del resultado de la condición, solamente se ejecuta una de las dos proposiciones (proposición-1 o proposición-2) y el control del programa pasará a la siguiente proposición después de la construcción IF, a menos de que exista una trasferencia de control incondicional - (GOTO o RETURN) dentro de la construcción del IF. Por ejemplo:

```
IF  X>Y  THEN  Z = X ;
ELSE  Z = Y ;
```

la variable Z tomará el valor de X o Y dependiendo de cual tenga el valor mayor. En esta proposición siempre se le asignará un valor a la variable Z pero solamente será ejecutada una de las dos asignaciones.

Volviendo a la forma general de la proposición IF, si no se requiere de la proposición-2, se puede omitir la cláusula ELSE y la construcción tomará la siguiente forma:

```
IF  condición  THEN  proposición-1 ;
```

en esta construcción, únicamente si el resultado de la condición es verdadero, se ejecuta la proposición-1.

Por ejemplo, el siguiente conjunto de proposiciones PL/M - asignará a la variable MAYOR el número 5 o el valor de Y dependiendo de cuál sea mayor. El valor de X cambiará durante la ejecución de la proposición IF solamente si Y es mayor que 5. El valor final de X siempre será asignado a la variable MAYOR.

```
X = 5;
IF  Y> X  THEN  X = Y ;
MAYOR = X ;
```

Ya que un grupo DO es equivalente sintácticamente a una sola proposición, cualquiera de las dos proposiciones de la construcción del IF puede ser un grupo DO. Por ejemplo:

```
IF T <A + 3  THEN
  DO;
     . . .
  END;
ELSE
  DO;
     . . .
  END;
```

La única restricción en el uso de la construcción IF es que proposición-1 no debe ser otra proposición IF si se utiliza la cláusula ELSE. En otras palabras, la construcción

```
IF condición-1  THEN
    IF  condición-2  THEN proposición-3;
 ELSE proposición-2;
```

es ilegal ya que no se sabe a que construcción IF pertenece la cláusula ELSE. Esta construcción puede ser reemplazada por cualquiera de las dos siguientes construcciones, dependiendo de lo que se desee hacer.

```
(1)    IF  condición-1  THEN
         DO;
           IF condición-2  THEN  proposición-3;
         END;
       ELSE  proposición-2;
```

(2)

```
IF  condición-1  THEN
  DO;
    IF  condición-2  THEN  proposición-3;
    ELSE  proposición-2;
  END;
```

## 7.- PROGRAMA DE EJEMPLO

A continuación se muestra un programa que ordena en forma ascendente los elementos de un vector X . Se emplea el método de la burbuja.

```
/* EL ORDEN DEL VECTOR  X  ES ARBITRARIO  */
    DECLARE X(10) ADDRESS INITIAL
    (500, 35, 750, 1, 2, 1979, 4, 4, 6);
          /* METODO DE LA BURBUJA  */
    DECLARE (I, SWITCH) BYTE, TEMP  ADDRESS;
    SWITCH = 1;
    DO WHILE SWITCH;
       SWITCH = 0;      /*  SE INICIA EL RASTREO DE X  */
       DO I = 0  TO  8;
         IF X(I)>X(I+1)  THEN
          DO;          /* SE ENCONTRO UN PAR FUERA DE ORDEN */
           SWITCH = 1;
           TEMP = X(I);
           X(I) = X(I+1);
           X(I+1) = TEMP;
          END;
       END;    /* SE COMPLETO EL RASTREO */
    END /*  WHILE */;
    /*  SE RASTREO EL VECTOR SIN EFECTUAR NINGUN INTERCAMBIO */
 EOF
```

Este programa rastrea el vector X comparando cada par de elementos adiacentes. Si se encuentra un par fuera de orden, se intercambian. Este proceso se efectúa en forma repetida hasta que se complete un rastreo sin haber hecho un intercambio. En este momento el programa termina y el vector estará ordenado.

## 8.- " PROCEDURES "

Un 'procedure' es una sección de código PL/M (en otros lenguajes se les llama subprogramas) que puede ser declarado y "llamado" desde otra parte del programa.

El uso de 'procedures' es la base de la programación modular, facilita usar y construir librerías de programas, facilita la programación, documentación y reduce la cantidad de código objeto generado para un programa.

### 8.1. Declaración de 'procedures'

La declaración de un 'procedure' tiene la siguiente forma general:

```
  nombre : PROCEDURE (lista de argumentos)  tipo ;
             proposición-1;
             proposición-2;
                 . . .
             proposición-n;
      END nombre;
```

El 'nombre' es un identificador PL/M por medio del cual se puede hacer referencia a este 'procedure' en otra parte del programa. La lista de argumentos tiene la forma

(arg-1, arg-2, . . . , arg-n)

donde arg-1 hasta arg-n son identificadores PL/M a los cuales se les denomina parámetros del 'procedure'. Los parámetros deben ser declarados dentro del cuerpo del 'procedure '. Los parámetros se pueden omitir si el 'procedure' no los requiere.

Si el 'procedure' regresa un valor al punto de donde es - llamado, se deberá definir de tipo BYTE o ADDRESS. Si el ' ---- 'procedure' no regresa ningún valor, no es necesario especificar el tipo en la declaración del mismo.

La ejecución de un 'procedure' se termina con la ejecución de la proposición RETURN que puede tomar una de las siguientes formas:

```
RETURN;
RETURN  expresión;
```

La primera forma es utilizada cuando el 'procedure' no regresa ningún valor (y no se declaró con tipo). La segunda forma se utiliza cuando el 'procedure' es de tipo BYTE o ADDRESS, en cuyo caso, el valor de la expresión de la proposición RETURN, será regresado al punto de llamada.

Dentro del cuerpo de un 'procedure' puede aparecer cualquier proposición válida de PL/M incluyendo anidamientos en llamadas y declaraciones de otros 'procedures'.

A continuación se dan algunos ejemplos de declaraciones de 'procedures'.

```
PROMEDIO: PROCEDURE (X,Y) ADDRESS;
 /*  CALCULA EL PROMEDIO DE LOS DOS PARAMETROS  */
   DECLARE (X,Y) ADDRESS;
   RETURN  (X + Y)/2 ;
END PROMEDIO;

PRINT: PROCEDURE (ARG);
   DECLARE  ARG  BYTE;
     OUTPUT(OFFH) = ARG;
   RETURN;
END PRINT;

SALTOS: PROCEDURE(N);
   DECLARE (CR, LF) BYTE INITIAL (ODH, OAH);
   DECLARE (I , N) BYTE;
   DO I = 1  TO  N;
   CALL PRINT(CR); CALL PRINT(LF);
   END;
END SALTOS;
```

No es válido que un 'procedure' sea recursivo. Esto es, un 'procedure' no puede llamarse a sí mismo y tampoco puede llamar a otro de una manera circular.

## 8.2. Llamadas de 'procedures'

Los 'procedures' pueden ser llamados o invocados después de que hayan sido declarados en el programa. Dependiendo si un 'procedure' regresa o no un valor, existen dos formas de llamar a un 'procedure'. La manera de llamar a un 'procedure' que no regresa ningún valor es:

```
CALL  nombre-del-procedure (lista de argumentos);
```

La manera de llamar a un 'procedure' que regresa un valor tiene la forma:

```
nombre-del-procedure (lista de argumentos);
```

el cual debe ser un operando o término de una expresión, tal y como se podría usar el nombre de una variable. Considerando los ejemplos de declaraciones anteriores, los siguientes ejemplos son llamadas a 'procedures'.

```
---   X = PROMEDIO(X,Y);
---   IF PROMEDIO(X,7) = 10  THEN  Y = 1;
---   CALL PRINT(J);
---   DO I=0  TO 10;
         CALL PRINT(MENSAJE(I));
      END;
---   CALL SALTOS(3 + PROMEDIO(X/3 , Y));
```

Cuando el tipo de los parámetros de una llamada a un 'procedure' difiera del tipo de los parámetros de la definición del mismo, automáticamente se harán las conversiones necesarias.

## 9.- APUNTADÓRES Y REFERENCIAS INDIRECTAS.

A menudo es inconveniente o imposible hacer una referencia directa a un dato PL/M. Esto sucede, por ejemplo, cuando se escribe un 'procedure' general que vaya a trabajar con algún arreglo - que se tenga que pasar como parámetro. Ya que a un 'procedure' no se le puede pasar un arreglo como parámetro, PL/M brinda la facilidad de manejar la dirección de los datos en lugar de los datos mismos, considerando que la dirección "apunta" a los datos.

### 9.1. Variables con base

Una variable con base es aquella que está apuntada por otra variable llamada su 'base'. A las variables con base, no se les asigna memoria a tiempo de compilación; su valor se calcula a tiempo de ejecución mediante un acceso indirecto por medio de su base. Una variable con base se declara; declarando primero su base, la cual debe ser de tipo ADDRESS, y a continuación la variable misma. Por ejemplo:

```
DECLARE  POINTER  ADDRESS;
DECLARE  ITEM  BASED  POINTER  BYTE;
```

a partir de este punto del programa, siempre que se escriba ITEM, realmente se está diciendo," el valor del byte apuntado por el valor actual de POINTER ". Esto indica que la secuencia

```
POINTER = 300H;
ITEM = 9AH;
```

cargará el valor 9AH en la localidad de memoria cuya dirección es 300H.

A continuación se dan algunos ejemplos de declaraciones de variables con base.

```
DECLARE (A,ZA,YA,QA)  ADDRESS;
DECLARE  X BASED A BYTE;
DECLARE (Z BASED ZA, Y BASED YA) ADDRESS;
DECLARE (Q BASED QA) (100) BYTE;
```

En el último ejemplo se define un arreglo Q cuya base es QA.

## 9.2. El operador punto

El operador punto da la facilidad de manejar las direcciones de variables y constantes en lugar de los valores mismos. La dirección de una variable, se designa colocando el caracter punto antes del nombre de la variable. Por ejemplo, las expresiones

.AA   y   .Z(7)

dan la dirección de AA y Z(7) respectivamente. Si A es un arreglo de tipo BYTE, el valor de .A(0) +5 es lo mismo que .A(5); si A es un arreglo de tipo ADDRESS, el valor de .A(0)+10 es lo mismo que .A(5).

En general, el operador punto toma las formas

```
.variable
.constante
.(constante)
.(lista de constantes)
```

por ejemplo:

```
.Z(2)        .45
.'MENSAJE'
.(15,'VARIABLES',ODH,OAH, 'CONSTANTES', 27H)
```

## 9.3. Ejemplo

El 'procedure' que se dá  a continuación, calcula el número de caracteres que contiene una cadena de los mismos, finalizada por el caracter '$'. Por ejemplo:

ERROR EN I/O$   y   TECLEE DE NUEVO$

son dos cadenas de caracteres válidas para el 'procedure'.

```
/* PROCEDURE QUE DETERMINA LA LONGITUD DE UNA CADENA
   DE CARACTERES  */
    LONG:PROCEDURE(A) BYTE;
        DECLARE A ADDRESS;
      /* A ES LA DIRECCION DE MEMORIA A PARTIR DE LA CUAL
          SE ENCUENTRA LA CADENA DE CARACTERES  */
        DECLARE  B BASED  A BYTE;
        DECLARE  I  BYTE;
        I = 0;
        DO WHILE B(I)<> '$';
           I = I + 1;
        END;
        RETURN  I;
    END LONG;
```

Se considera que el caracter '$' no forma parte de la cadena, sino que actúa como delimitador.Una posible llamada a este 'procedure' es:

```
IF LONG(.A) = 0 THEN  RETURN;
```

o

```
K = LONG(.TITULO) + X MOD 4;
```

## 10.- PROPOSICION GO TO Y ETIQUETAS

### 10.1. Nombres de etiquetas

Tanto a las proposiciones como grupos de ellas, se les puede asignar una etiqueta para su identificación y referencia. Una proposición etiquetada tiene la forma

ETIQUETA-1:ETIQUETA-2:...:ETIQUETA-n: proposición ;

donde ETIQUETA-i son identificadores PL/M. A continuación se dan ejemplos de proposiciones con etiquetas.

```
LOOP: X = X + 1;
E1:L1:C1: Z = A MOD 5;
```

Una etiqueta también puede ser un número, por ejemplo:

```
50: T = A - X/5;
```

esta etiqueta especifica que el código objeto de esta proposición estará colocado en la dirección de memoria 50. Por razones obvias en una proposición solo puede aparecer una etiqueta numérica. -- Cuando en una misma proposición se deseen colocar etiquetas simbólicas y numérica, la etiqueta numérica deberá ser la primera. Por ejemplo:

```
100H:ALFA:ORIGEN: V = (Z - I) MOD   Y;
```

La forma simbólica de las etiquetas no tiene ningún efecto sobre el origen del código en memoria.

Opcionalmente, las etiquetas pueden declararse al igual que las variables de acuerdo a la siguiente forma general.

DECLARE identificador LABEL ;

por ejemplo:

```
DECLARE L1 LABEL;
DECLARE (ALFA, BETA) LABEL;
```

### 10.2. Proposición GO TO

La proposición GO TO interrumpe incondicionalmente el orden de ejecución secuencial de un programa, transfiriendo el control a la proposición que tenga la etiqueta colocada después del GO TO. La proposición GO TO tiene tres formas distintas;

```
GO TO nombre-etiqueta ;
GO TO número ;
GO TO nombre-variable;
```

En la primera forma, el 'nombre-etiqueta' es un identificador que aparece como etiqueta de una proposición. En la segunda forma, el número es una dirección absoluta de memoria y el control del programa se transfiere directamente a esa dirección. En la tercera - forma, el 'nombre-variable' es una variable cuyo contenido es una dirección de memoria; el control pasa directamente a esa direc--- ción absoluta de memoria.

La palabra reservada GO TO también se puede escribir como GOTO.

Como una nota final sobre etiquetas, se recomienda a los - programadores evitar, siempre que sea posible, el uso del GO TO y utilizar las construcciones IF-THEN-ELSE y grupos DO. En gene-- ral, el efecto de ésto es obtener un código objeto mejor y programas más legibles.

## 11.- PROCESAMIENTO DE MACROS A TIEMPO DE COMPILACION

La declaración LITERALLY define una macro que puede ser expandida a tiempo de compilación. Se puede declarar un identificador que represente una cadena de caracteres, el cual será substituido por ésta en cada ocurrencia del identificador en el texto subsecuente. La forma de la declaración es:

```
DECLARE  identificador LITERALLY  'string';
```

donde el identificador es cualquier identificador válido en PL/M y 'string' es una secuencia de caracteres del conjunto PL/M, cuya longitud no debe exceder a 255, encerrada entre apóstrofes. Las siguientes proposiciones ilustran el uso de las macros.

```
      DECLARE LIT LITERALLY 'LITERALLY' ,
                 DCL LIT  'DECLARE' ;
      DCL TRUE LIT 'OFFH',  FALSE LIT 'O';
      DCL POR$SIEMPRE LIT 'WHILE TRUE';
      DCL (X,Y,Z) BYTE;
      DCL E1 LABEL;
       X = FALSE;
   E1: Z = X OR Y  AND  TRUE;
       DO POR$SIEMPRE;
          Y = Y + 1;
          IF Y   20  THEN  HALT ;
       END;
            . . .
  EOF
```

La primera declaración define abreviaciones para las palabras reservadas LITERALLY y DECLARE, las cuales pueden ser utilizadas en las proposiciones subsecuentes.

# 12.- VARIABLES PREDECLARADAS Y PROCEDURES INTERCONSTRUIDOS

## 12.1. Entrada y salida

La manera de hacer una entrada es mediante la forma general:

INPUT(número)

esta forma se utiliza como podría utilizarse una llamada a un 'procedure' de tipo BYTE, su valor será una cantidad de 8 bits, tomada del puerto de entrada especificado en el argumento. El argumento numérico debe estar en el rango 0 - 255 para el microprocesador 8080 y en el rango 0 - 7 para el 8008.

Para efectuar una salida, se utiliza la pseudovariable OUTPUT que deberá aparecer siempre en el lado izquierdo de una proposición de asignación; en cualquier otra posición es ilegal. Su forma es:

OUTPUT(número) = expresión;

donde el argumento numérico debe estar en el rango de 0 - 255 para el microprocesador 8080 y en el rango de 0 - 23 para el 8008. Su función es colocar el valor de 8 bits de la expresión en el puerto especificado por el argumento numérico.

## 12.2. LENGTH y LAST

PL/M tiene dos funciones interconstruidas que se basan en la dimensión declarada para los arreglos. Estas funciones tienen la forma

LENGTH(identificador)

LAST(identificador)

donde 'identificador' es cualquier nombre de arreglo o variable previamente declarado. Estas formas pueden aparecer en cualquier lugar de una expresión de un programa PL/M. Estas funciones evalúan la longitud declarada para una variable y el índice del último elemento de un arreglo respectivamente. El siguiente programa utiliza la función LAST para poner a ceros todos los elementos del vector VEC.

```
                DECLARE  VEC(50) BYTE;
                DECLARE K BYTE;
                 DO K = 0  TO  LAST(VEC);
                    VEC(K) = 0;
                 END;
                EOF
```

Para el ejemplo anterior, LENGTH(VEC) daría como resultado el número 50.

Para cualquier arreglo V;

$$\text{LENGTH(V)} = 1 + \text{LAST(V)}$$

12.3. Funciones LOW, HIGH y DOUBLE

Las funciones LOW y HIGH se emplean para convertir un valor tipo ADDRESS a tipo BYTE. Ambas toman como argumento una expre--- sión tipo ADDRESS y dan como resultado un valor tipo BYTE. Su forma es:

```
                    LOW( expresión )
                    HIGH( expresión )
```

LOW dá como resultado el byte menos significativo de su argumento; HIGH dá como resultado el byte más significativo de su argumento.

La función DOUBLE convierte un valor tipo BYTE a un valor - tipo ADDRESS mediante la concatenación de un byte puesto a ceros a la izquierda del valor BYTE.

Estos tres 'procedures' para conversión de tipo, se pueden llamar desde cualquier expresión válida de un programa pero nunca deben aparecer en el lado izquierdo de una proposición de asignación. Por ejemplo:

```
DECLARE  A  ADDRESS, (I,J) BYTE;
   A = 01FCH;
   I = LOW(A);
   J = HIGH(A);
   A = DOUBLE(J);
EOF
```

En este ejemplo, los valores que toman I, J y A son 0FCH, 01 y 0001H respectivamente.

## 12.4. Funciones de rotaciones y corrimientos

### 12.4.1. Funciones ROL y ROR

Estas dos funciones tienen la forma

```
ROL(expr-1, expr-2)
ROR(expr-1, expr-2)
```

donde 'expr-1' y 'expr-2' deben evaluarse como cantidades tipo BYTE; las dos funciones dan como resultado un valor tipo BYTE.

La función ROL efectúa 'expr-2' rotaciones a la izquierda al valor de 'expr-1'. ROR es similar, solo que las rotaciones son a la derecha. A continuación se muestran las funciones en forma esquemática.

Por ejemplo:

```
ROR(10011101B, 1) da como resultado   11001110B
ROL(10011101B, 2) da como resultado   01110110B
```

La ejecución de estas funciones afectan el bit de carry del CPU. El carry tomará el valor del último bit que sale por un extremo. Para el primer ejemplo, el carry quedará con un valor de 1 y en el segundo, con un valor de 0.

El valor de 'expr-2' no debe ser cero.

```
DECLARE  I ADDRESS, CRLF LITERALLY 'CR,LF',
    TIT DATA (CRLF,LF,LF,
'                     TABLA DE RAICES CUADRADAS',CRLF,LF,
' VALOR RAIZ VALOR RAIZ VALOR RAIZ VALOR RAIZ VALOR RAIZ',CRLF,LF);

    /*    PROGRAMA  PRINCIPAL              */

    DO I = 1    TO  1000;
       IF I MOD 5 = 1 THEN
       DO;
          IF I MOD 250  = 1  THEN
           CALL PRINT$STRING(.TIT, LENGTH(TIT));
          ELSE
           CALL PRINT$STRING(.(CR,LF), 2);
       END;
     CALL PRINT$NUM(I,10,6,TRUE);
     CALL PRINT$NUM(RAIZ(I),10,6,TRUE);
    END;

EOF
```

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

# MICROPROCESADORES: TEORIA Y APLICACIONES

## SPECTRUM

MARZO, 1979

# The microcomputer invades the production line

## 'Mechanotechniques' yield to the 'silent revolution' of electronics

There's a quiet revolution going on in the manufacturing world that is slowly changing the look of the factory floor. Spearheaded by mini- and microcomputer technologies, electronics is controlling and monitoring processes far more efficiently and quietly than clangy electromechanical relays—and with minimum human supervision.

Although the application of electronics to manufacturing has been somewhat slow, despite the availability of the technology, distinct trends can be observed. These include:

- A proliferation of microcomputers in process-control applications, mainly brought on by their low prices, flexibility of reconfiguration, and suitability for dedicated functions.
- A new generation of robots for small-parts assembly, and the appearance on the production line of smarter robots. Some can make machining and processing decisions.
- An increase in the use of electronic tools, such as lasers and electron-beam guns, to treat materials.
- New electronic data-logging instruments that now monitor more manufacturing processes than scores of human operators formerly did.
- Greater use of computer software through computer-aided design, computer-aided manufacturing, and design automation to cut lengthy production times and increase throughput rates.

### A niche for microcomputers

The declining cost of microcomputer products has increased their use as industrial controllers. An 8-bit microcomputer system on a single printed-circuit (PC) board can now be purchased for $300 to $400—far lower than the $1500 to $2000 for a minicomputer to do the same task. And the microcomputer's use as an industrial controller is more efficient; a minicomputer has far more computing power than its application needs. In addition the microcomputer, with its smaller size—and hence less costly software program—is easier to reconfigure for changing applications.

Many microcomputers in industry are being used as dedicated controllers. Typically several microcomputers, each handling a specific function and all tied together by a minicomputer, can be found in a plant. The minicomputer performs the larger data-manipulation task and acts as an interface between the microcomputers and a monitoring station, which may be a CRT terminal (Fig. 1). A printer may also be included.

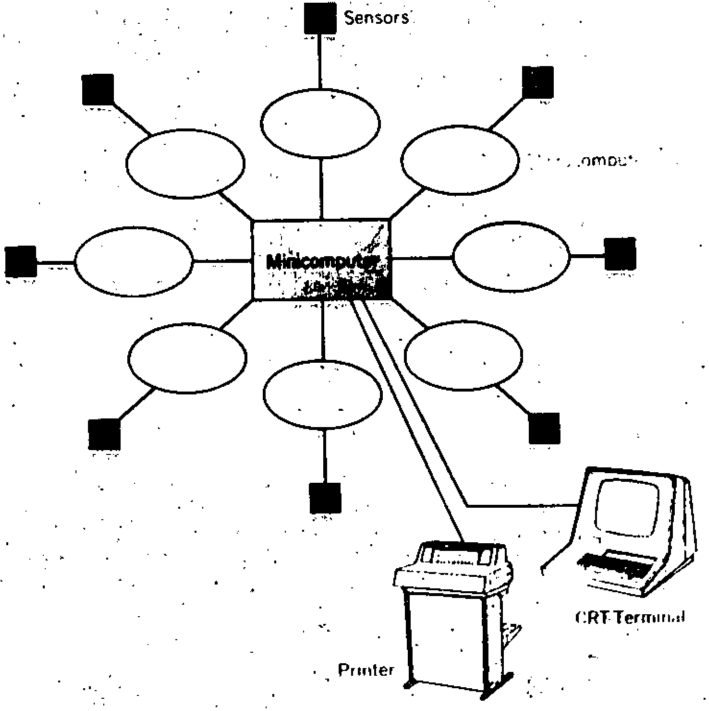
A high-volume application like automotive sheet-metal gauging illustrates the microcomputer's use. The conventional method is to use visual and manual inspection when sampling and checking sheet-metal parts for correct dimensions. This is done offline and only at infrequent intervals, because it takes a relatively long time. With a microcomputer, several gauging sensors can be set up to scan samples online. The job can be done faster and more frequently, and this ensures that fewer out-of-specification sheet-metal parts will get through. Inspections are also more accurate.

Roger Allan Associate Editor

The low cost of microcomputer PC boards has, in many cases, simplified the servicing of electronic control systems. A defective microcomputer board in a dedicated process-control system can simply be isolated and discarded and a replacement quickly put in. Larger and more expensive microcomputers, however, require troubleshooting and repair by service personnel in the plant; the PC boards are simply too costly to discard.

This poses a problem: plant service personnel are mostly house electricians and ill-equipped to repair complex electronic equipment. The shortage of technically proficient maintenance workers is part of a larger problem that the manufacturing industries face as the computer is applied in its many forms: production supervisors and

**[1] Because of its low cost and reprogramming flexibility, the microcomputer is being used in many dedicated industrial applications. Typically several microcomputers are tied into a minicomputer, which performs larger data-manipulation and analysis tasks.**



0018-9235/79/0100-0053$00.75©1979 IEEE

managers must be as familiar with computer technology (both hardware and software aspects) as they are with the processes they manage.

An indication of the growing use of mini- and microcomputers in manufacturing is the rise in sales of single-card I/O PC boards for interfacing with process variables. Typically such boards contain A/D converters, D/A converters, signal-conditioning components, and multiplexing circuitry. Some require little or no additional circuitry and can be interfaced to the sensor directly. Nearly all such PC boards are designed to plug into and interface directly with the mini- or microcomputer they're designed for; the link usually is made in the computer's card-cage housing.

Recently smart digital programming and controller instruments for process control have begun to appear. Typical of such equipment is the microcomputer-based DCP7700 from Honeywell, Fort Washington, Pa. It combines in one box a variable-setpoint vs time programmer with a three-mode controller. The programmer/controller has a keyboard through which an operator can enter and control process inputs. The unit can store up to nine separate programs consisting of up to 200 ramp or soak segments plus event switches.

Analog Devices in Norwood, Mass., has gone several steps further with a total microcomputer-based, closed-loop, real-time measurement and control system (Fig. 2). Known as MACSYM II, the system has a CRT and keyboard and is designed for scientific and industrial applications requiring the acquisition, storage, computation, reduction, presentation, and outputting of high-and low-level signals from universally used sensors (thermocouples, strain gauges, resistive temperature devices, etc.). The system makes use of a high-level language called MACBasic, an extension of Basic, to allow nontechnical operators to use it. As many as 256 channels of analog input data can be accommodated with plug-in PC cards that interface directly to nearly any known process variable.

**[2] This closed-loop system from Analog Devices is designed for total process measurement and control by an unskilled operator. The system, MACSYM II, interfaces with all popular industrial sensors.**

## More robots are appearing

New robots for parts assembly are proving more attractive for industrial batch assembly (nearly three-fourths of all assembly operations in the U.S. are of the batch-assembly variety). Until recently most robots were used for such operations as welding, parts transfer, die casting, forging, and the operation of punch presses. Recent robots have included versions with high intelligence. They can recognize poor metal welds and improperly positioned machine tools as well as do the work.

At the recent Third Industrial Robot Conference and Exhibition in Chicago, Unimation Inc. of Danbury, Conn., introduced the first commercially available microprocessor-controlled robot specifically designed for the assembly of small items, such as electronic components and hardware. Known as Puma (programmable universal manipulator for assembly), it was developed jointly by Unimation and General Motors in Detroit.

GM is using several Pumas for small-parts assembly. The small robot can repeatedly position an object, staying within a tolerance of 0.004 in (0.100 mm), which is only slightly thicker than a human hair. The heart of the 175-lb (79-kg) robot is a Digital Equipment Corp. LSI-11 microprocessor, which controls five other microprocessors, each dedicated to one of five robot arm axes. The motions correspond to waist rotation, shoulder rotation, elbow rotation, wrist bend, and hand rotation. The robot can lift up to 7.7 lb (3.5 kg), including the weight of its end manipulator. Under maximum load, its arm-tip velocity is 3.3 ft/s (101 cm/s). Arm and controller may be separated from each other by as much as 10 ft (3.3 meters) by means of a cable assembly.

Early last year the Westinghouse Electric Corp. Research and Development Center in Pittsburgh submitted to the National Science Foundation a second-phase proposal on programmable automation of batch-assembly operations. The first phase of this study, "Programmable assembly research technology transfer to industry," was completed in October 1977, and it led to the preliminary design of an automated system for small motors (Fig.3). Help for the study was supplied by SRI International of Menlo Park, Calif.; the Charles Stark Draper Laboratory at the Massachusetts Institute of Technology in Cambridge, and the University of Massachusetts in Amherst.

As part of the first-phase study for the National Science Foundation, Westinghouse conducted a worldwide review of programmable-assembly technology. It analyzed about 60 different Westinghouse product lines before deciding on small motors as the most adaptable to programmable automatic assembly. About 450 different motor styles were assembled experimentally. The average batch size was 600, and there were an average of 13 changeovers per shift. The pilot system, known as APAS (adaptable programmable assembly system), was tested over a three-month production schedule.

Richard G. Abraham, Westinghouse's manager for programmable automation, explains: "Although robots are important in the automation of batch-assembly operations, it is equally important to have low-cost, microprocessor-based visual-inspection systems for checking incoming parts and for ascertaining proper assembly steps and style changes. It is also important to stress the need for programmable-parts presentation equipment and software to manage the changeover required for different product styles."

Advances are being made in group control of robots. At the Leningrad Polytechnical Institute in the Soviet Union, researchers have devised a computer system that controls up to 20 sensing robots in real time. The researchers expect to increase this handling capacity to 40 robots as more sophisticated machines are developed with built-in controllers. In the present project, an operator at the system's control panel directs specific robots to carry out dedicated tasks and to take corrective actions, if necessary.

## The cell concept

Last year the Foundation of Scientific and Industrial Research at the University of Trondheim in Norway set up what it calls the first full-scale laboratory production line to use a cellular concept. Under this, manufacturing operations are broken down into "cells," each at a different plant. Each cell is responsible for the manufacture of specific subassemblies for a particular product. The cells are interconnected by a network of material and subassembly supply lines. The Norwegian project manufactured complex diesel-engine parts for Wickman Manufacturing of Norway.

In the cellular concept (Fig. 4) the output per manhour of labor is reported to be considerably higher than that of conventional manufacturing methods, even when the additional costs of the interconnecting material and subassembly supply lines are taken into account. Each cell has at its core one robot.

The cell system can be operated day and night, requiring worker participation only during the day. All that is needed for unattended night operation is proper planning, so the robot has an ample supply of materials and enough storage area for completed subassemblies. The robot is a six-arm model manufactured by Cincinnati Milacron, Cincinnati, Ohio.

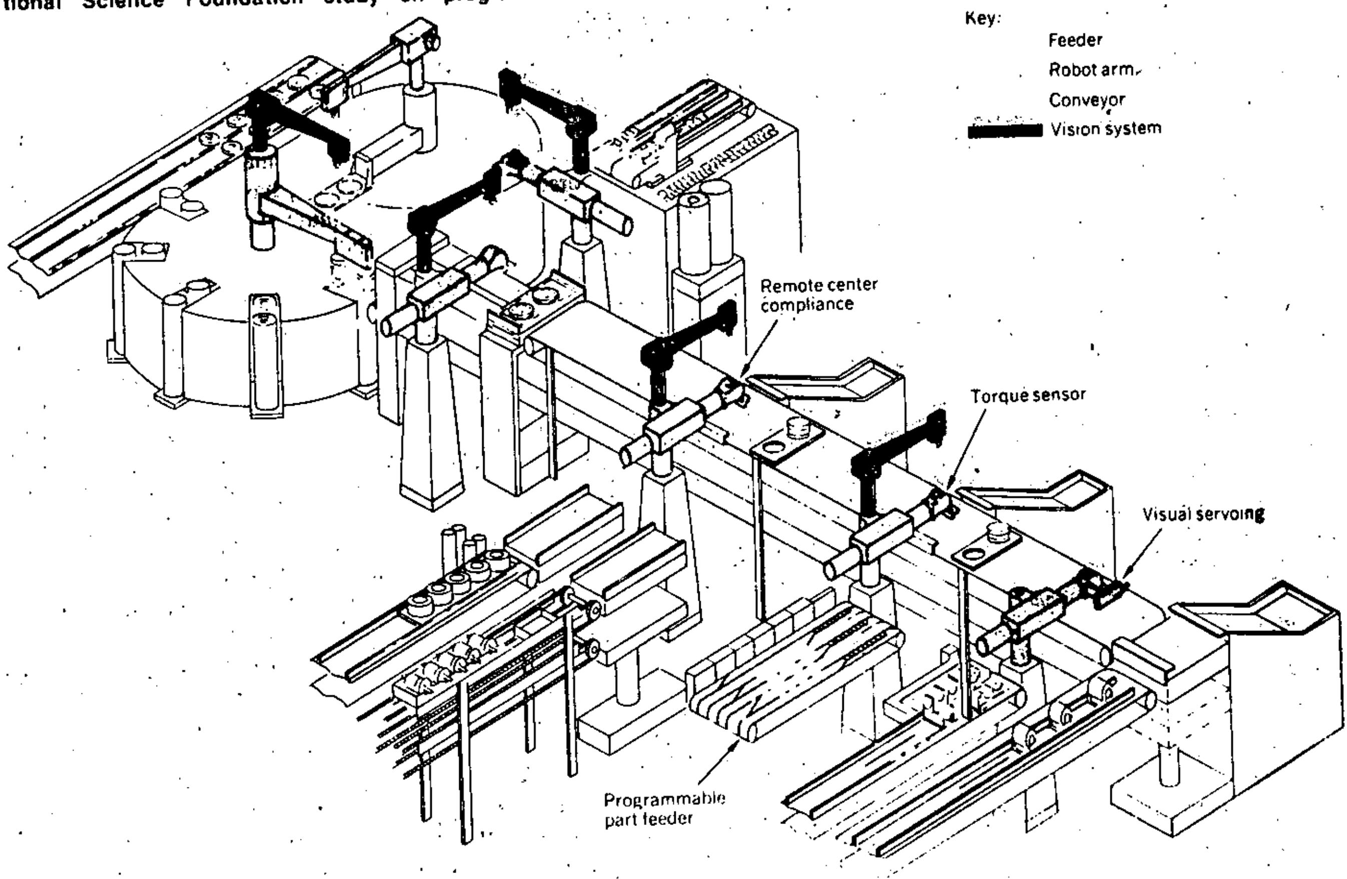
According to Prof. Oyvind Bjorke, head of the Norwegian foundation's production engineering laboratory, the cell concept, with its flexibility for manufacutring different parts, has proved particularly useful in Norway.

"This cell concept," he says, "has been quite beneficial to us in terms of increasing productivity, mainly due to our social and geographical conditions. Norway is a country that has a small and scattered population, which is desirable for national security reasons but is undesirable for manufacturing. We cannot get a large concentration of skilled workers in any one industrial center. This is like bringing the mountain to Mohammed if you can't bring Mohammed to the mountain."

## Lasers for materials treatment

Be it drilling, cutting, welding, treating, or the removal of materials, the laser can be found taking on more of these tasks in the plant as its precision and power are increased and its price drops further. Manufacturers are finding the new lasers more reliable, better designed for

**[3] A pilot automated assembly line for manufacturing small motors has been designed by Westinghouse Electric's Research and Development Center as part of a National Science Foundation study on programmable automation. Assisting have been SRI International, the Charles Stark Draper Laboratory at the Massachusetts Institute of Technology, and the University of Massachusetts.**



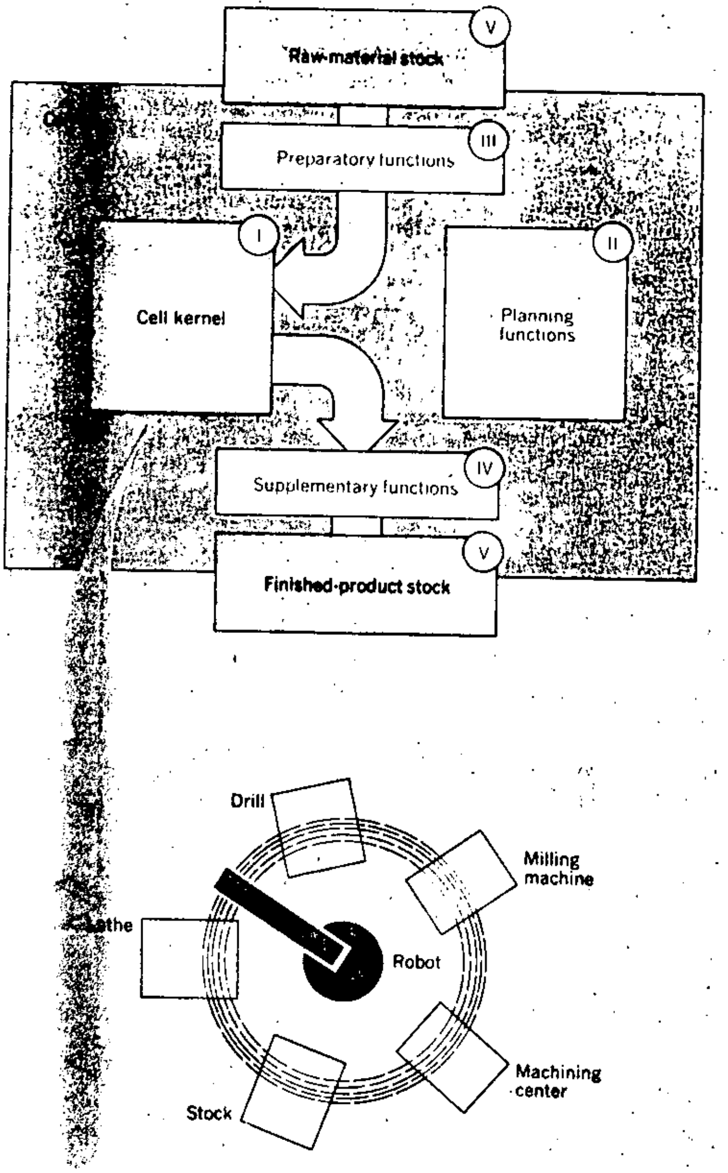
industry use, and simpler to operate than those previously available.

With recent improvements in output power, $CO_2$ lasers in the 5-15-kW range are becoming more popular for metal cutting and treating. In fact, laser treatment of materials is the fastest-growing segment of all laser sales, next to their applications in research.

One of the most dynamic and promising areas of industrial laser applications is in platemaking for printing, where faster printing turnarounds and lower costs are major advantages. Laser scanners are used to expose printing plates as large as 48 by 60 inches (122 by 152 cm) directly from the pasted-up page, thus eliminating several in-between steps required with conventional methods. Many medium- and small-circulation newspapers are using laser platemakers.

There also have been advances in computer control of lasers. At Western Electric's Engineering Research Center near Princeton, N.J., a high-speed, computer-controlled laser has been developed to spot-weld miniature-relay terminals. The laser's spot-welding speed of 20 times per second is four times faster than the resistance welding system it replaces. The system consists of a 200-watt (average power) pulsed Nd:YAG (neodymium:yttrium-aluminum-garnet) laser and an $X$-$Y$, positioning table, whose position accuracy is within 0.0001 in (0.00025 cm). The laser and positioning table are under the control of a microprocessor, which monitors position coordinates by use of linear encoders.

**[4] The cell concept of decentralized manufacturing has at its core a robot programmed to work day and night, attended by humans only during the day. The system increases throughputs in job-lot manufacturing. Proposed by the Foundation of Scientific and Industrial Research at the University of Trondheim in Norway, it is being used with a Cincinnati Milacron six-arm robot to manufacture complex diesel-engine parts.**



Another favorite and recent technological tool for welding and hardening metals is the electron-beam gun. Its use for selective heat treating of metals offers advantages over other techniques: it is faster to use than the laser, more accurate than induction-heating systems, and more selective in resolution than standard flame-hardening techniques. And it is more energy efficient than all three. These advantages, however, aren't without drawbacks. Electron-beam systems are still very expensive.

An electron-beam welding process was recently developed by Technical Materials Inc. of Lincoln, R.I., for welding formerly incompatible metals in strips of unlimited lengths. The process makes possible the welding of gold with copper, silver with copper, invar with stainless steel, and steel with copper. The key to the welding process is a triode electron gun that generates the electron beam. A high-stability alumina insulator is shielded from problem-causing vapors and is precisely positioned for minimal thermal expansion. Gun operation is accomplished with a 60-kV potential, which compares favorably with a potential of about 150 kV needed by most electron-beam welders. Advanced electronic controls for the gun's electro-optical focusing system are responsible for the lower potential.

## Keeping tabs on the process

Thanks to microprocessor technology, inexpensive data-logging instruments are making process control more scientific. This, in turn, has made possible better control over processes, because of the availability of more accurate and systematic data-collection and analysis techniques.

Data loggers can be found in nearly every industrial application, from food processing and materials treatment to papermaking and the generation of steam. The high speed and accurate data-collection advantages now possible have made obsolete more expensive and slower equipment, like pen-chart recorders.

A typical data-logging application is recorded in the files of the John Fluke Manufacturing Co. in Mountlake Terrace, Wash. To obtain more control over the final product, a producer of electrically conducting aluminum bus bars used a Fluke Model 2240A data logger to keep track of temperature and voltage data. The data were collected in point-for-point pairs along each bar to determine relative conductivity as a function of the bar's shape, alloy, and type. The reason for measuring temperature in addition to voltage was to provide better data correlation. The instrument also monitored the bus bars for dangerous overheating at five separate points every second. And it was equipped to notify operators of overheating and to turn the process off if necessary. All of the collected data were fed to a minicomputer for analysis.

## The challenge of technology transfer

"Many U.S. manufacturing experts say the technology to increase productivity has been here for some time. Knowing how to put it to practical use rapidly is really the problem, they add. The laser, for example, has been a research tool in the laboratory for well over a decade, but only in the last few years has it been applied successfully in manufacturing plants. Similarly the minicomputer was available long before it was applied in industry on a wide scale.

Perhaps the problem runs deeper than what is perceived. Prof. Gustav Olling, chairman of Bradley University's Department of Manufacturing Technology in Peoria, Ill., and a former practicing engineer with extensive industrial experience, puts it this way:

"U.S. academic researchers and more practical production/manufacturing individuals are not in tune with each other. Each group needs to get more involved with each other's experiences—the academician with the production manufacturing person's real-life experiences in the plant and the production person with the academician's theoretical contributions. This will also ensure an easier transition for engineering graduates into industry."

Prof. Olling cites foreign countries with more advanced industrial applications of electronics than the U.S.—Japan, Germany, and Norway. In these countries, he says, academic and industrial people work hand in hand to solve industry's problems. In many cases, promotions to professorships are based on the individual's industrial experience as well as scientific contributions. In some countries a professor who lectures in a university may also work in industry to solve automation problems. Prof. Oyvind Bjorke's work on cellular manufacturing, described in this report, is an example.

As part of its long-range plans, Computer Aided Manufacturing International, a Texas-based research organization, has an education/industry committee of which Prof. Olling is chairman. Its objective is to stimulate cooperation between industrial, educational, and professional groups through information exchanges, common research areas, faculty and student exchanges, and the development of educational programs in computer-aided design and computer-aided manufacturing.

The trend is to incorporate more intelligence in data-logging instruments and to provide them with wider ranges of options, since some of their users are not technically proficient and industrial applications differ widely.

## Greater use of CAD, CAM, and DA

Industry has slowly begun to make greater use of computer-aided design (CAD), computer-aided manufacturing (CAM), and design automation (DA) as manufacturing has become more complex and productivity has not kept pace with rising costs for labor and materials. In countries like Japan, where high industrial productivity is a national goal, CAD is used in nearly every industry. For example, the government-owned Nippon Telegraph and Telephone Corp. recently opened a service to the public that provides via telephone the supporting software for developing LSI microprocessor programs. The information comes from the data banks of a national computer.

Although total use of CAD and CAM—the completely automated factory—is still far off, manufacturing experts are emphasizing the potential of each technique. Computer Aided Manufacturing International of Arlington, Tex., a research organization composed of leading worldwide industrial companies, has been developing computer programs and manufacturing strategies to help automate production systems. These are intended to lead to higher manufacturing productivity.

One such program is CAPP (computer-aided process planning), a program that links CAD and CAM. CAPP features a parts-coding scheme that allows almost any part in a factory to be described from a few fundamental geometric shapes. With a CRT terminal and a keyboard, a designer in a plant using CAPP can call up at once, or even create instantly, a part needed for manufacturing. There is no need for time-consuming engineering drawings. All of the part's manufacturing characteristics are described in detail on the CRT screen.

One reason why more industrial manufacturers don't use more programs like CAPP is the high initial cost of software preparation and generation. Although CAPP has proved cost-effective for industrial applications, it is being used in few job-lot companies. They are skeptical of investing the $100 000 needed typically to build just a classifying data base with CAPP. For a 20 000-item job lot, 30 to 40 weeks are needed to build the classifying data base, and about one-third of this time is spent planning how to put such a data base together.

## Automating the design of PC boards

An area in manufacturing where application of the computer's power has made notable progress is in the design of PC boards. Here DA, design automation, is using the computer not only to generate precision design data (such as PC-board artwork) but also to lay out optimally the circuit components and interconnections on the board. With CAD, computer-aided design, component placement and interconnection are performed manually.

DA has been applied successfully in the design of dense PC boards with high-speed ICs like ECL (emitter-coupled logic), a job virtually impossible to achieve without the computer's aid. Given a logic diagram of such basic data as the components to be used, the size of the PC board, and the required wiring, a DA program can produce interconnection patterns for the most complex multilayered PC boards.

A leader in applying DA to PC-board design, Automated Systems Inc. in El Segundo, Calif., recently introduced a software module (Place II) that makes it possible to route the interconnections for large PC boards of high density, a job that previously could not be done automatically. For example, using the Place II module, a designer can lay out PC cards as large as 15 by 18 inches (38 by 46 cm) with 300 different devices. This is done by sophisticated techniques that allow the computer to group circuit elements according to similar functions. ◆

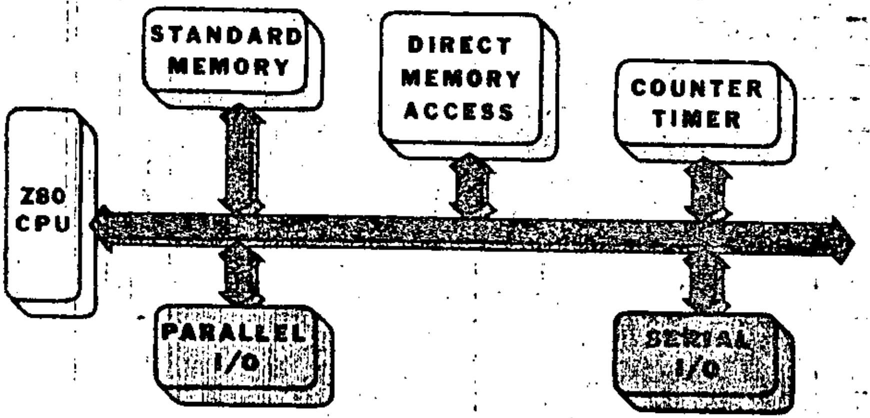
centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

# MICROPROCESADORES: TEORIA Y APLICACIONES

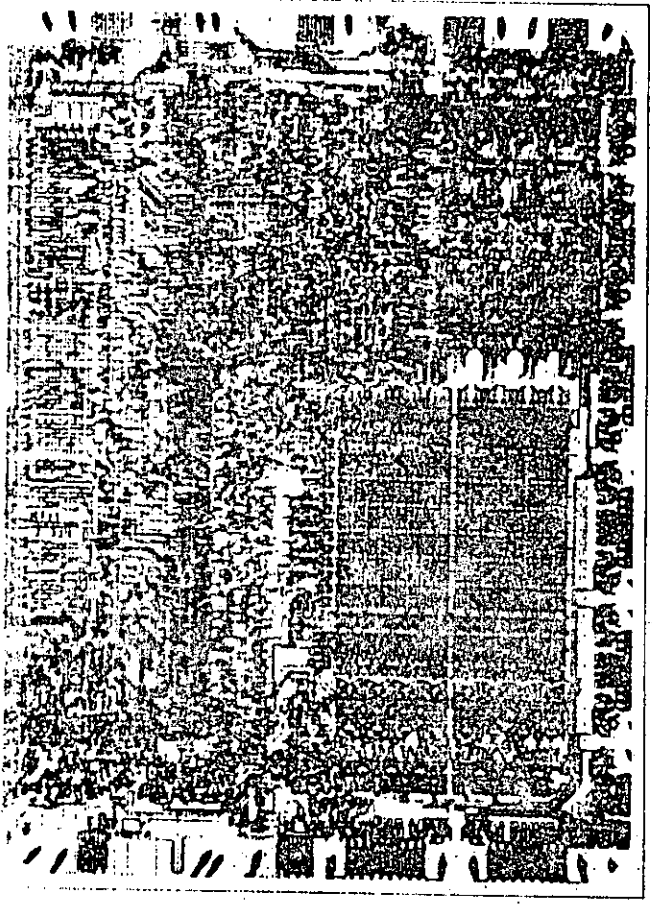
## Z80 MICROCOMPUTER FAMILY

MARZO, 1979

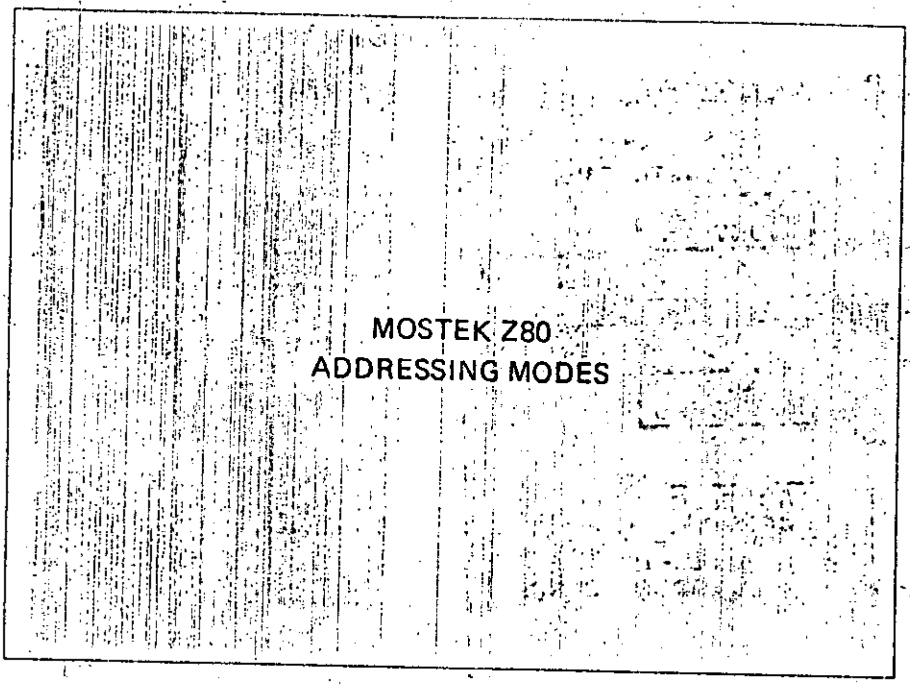
## MOSTEK Z80 FAMILY



NOTES:

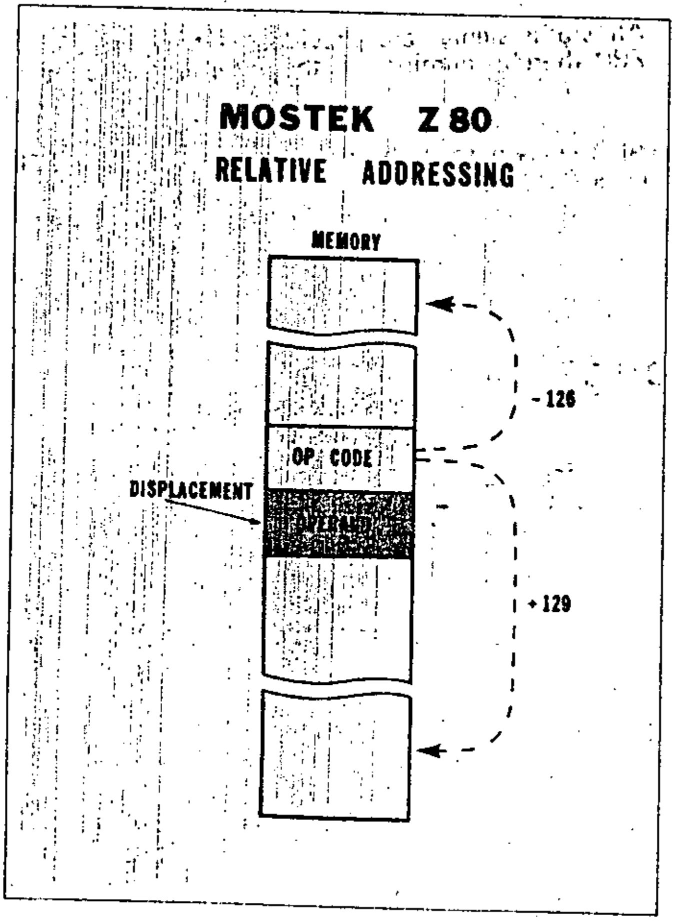


NOTES:



MOSTEK Z80
ADDRESSING MODES

NOTES:

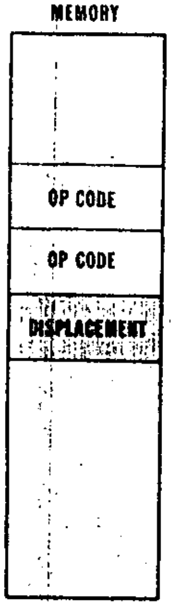


MOSTEK Z 80
RELATIVE ADDRESSING

□ The Relative Addressing Mode of the Z80 produces shorter programs because of the availability of two byte jumps rather than the normal three byte jumps.

NOTES:

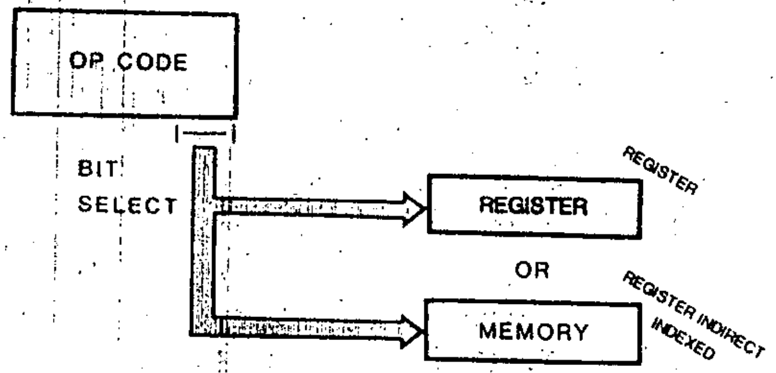
## MOSTEK Z80
### INDEXED ADDRESSING



(1) In addition to providing two additional 16 bit memory pointers the indexed addressing mode provides for efficient handling of blocks of data. The second index register allows efficient handling of two tables of data simultaneously.

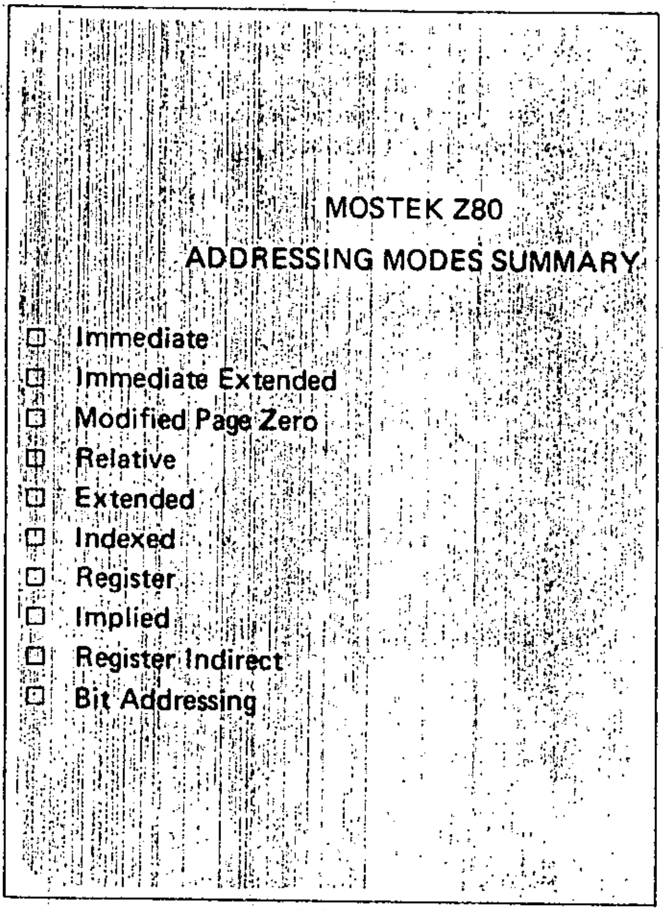
NOTES:

## MOSTEK Z80
### BIT ADDRESSING



NOTES:

The Bit Addressing Mode allows specifying an individual data bit anywhere in memory or in any CPU register. This avoids having to put data in the accumulator to mask out bits in order to modify or test an individual bit.

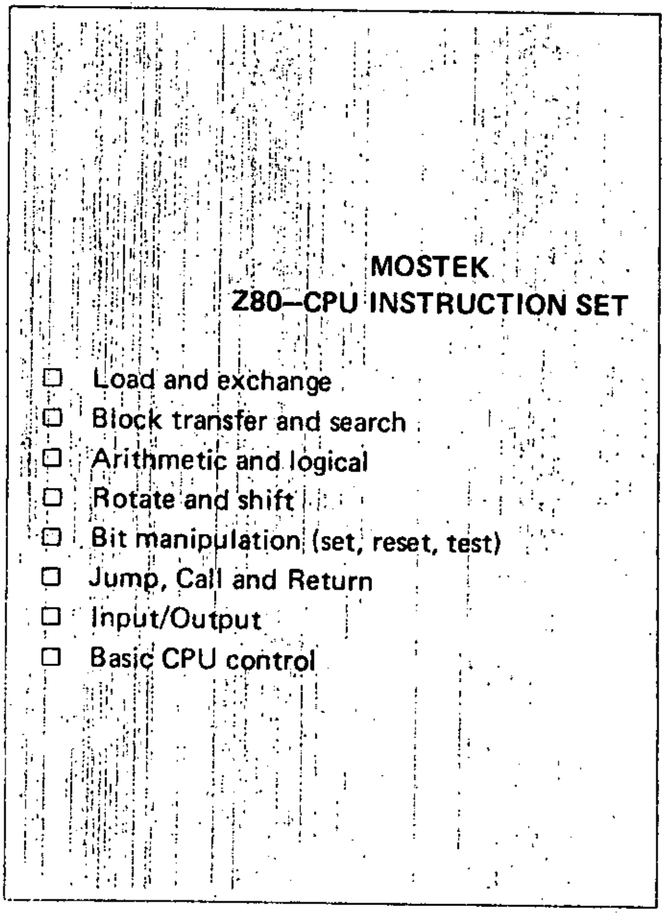


MOSTEK Z80

ADDRESSING MODES SUMMARY

- □ Immediate
- □ Immediate Extended
- □ Modified Page Zero
- □ Relative
- □ Extended
- □ Indexed
- □ Register
- □ Implied
- □ Register Indirect
- □ Bit Addressing

NOTES:



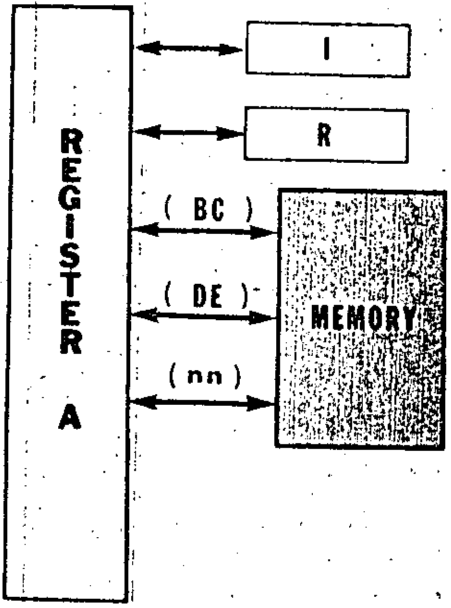
MOSTEK

Z80–CPU INSTRUCTION SET

- □ Load and exchange
- □ Block transfer and search
- □ Arithmetic and logical
- □ Rotate and shift
- □ Bit manipulation (set, reset, test)
- □ Jump, Call and Return
- □ Input/Output
- □ Basic CPU control

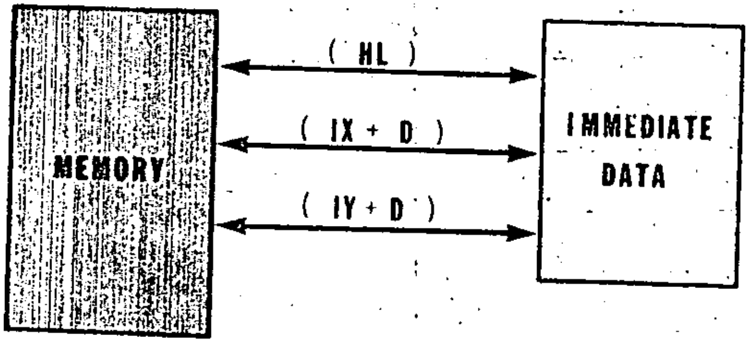
NOTES:

## 8-BIT LOAD GROUP



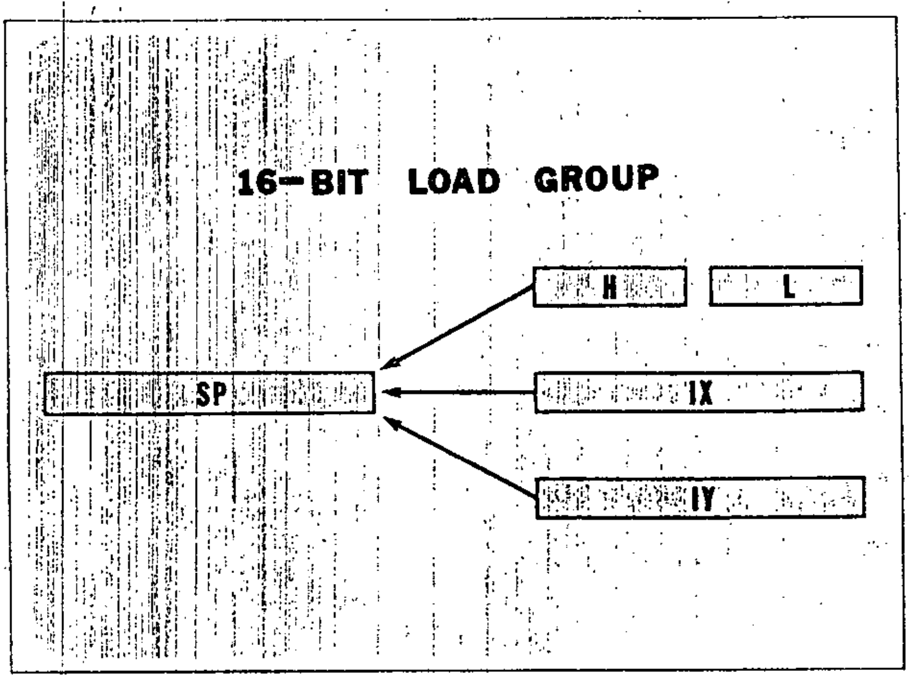
NOTES:

## 8-BIT LOAD GROUP



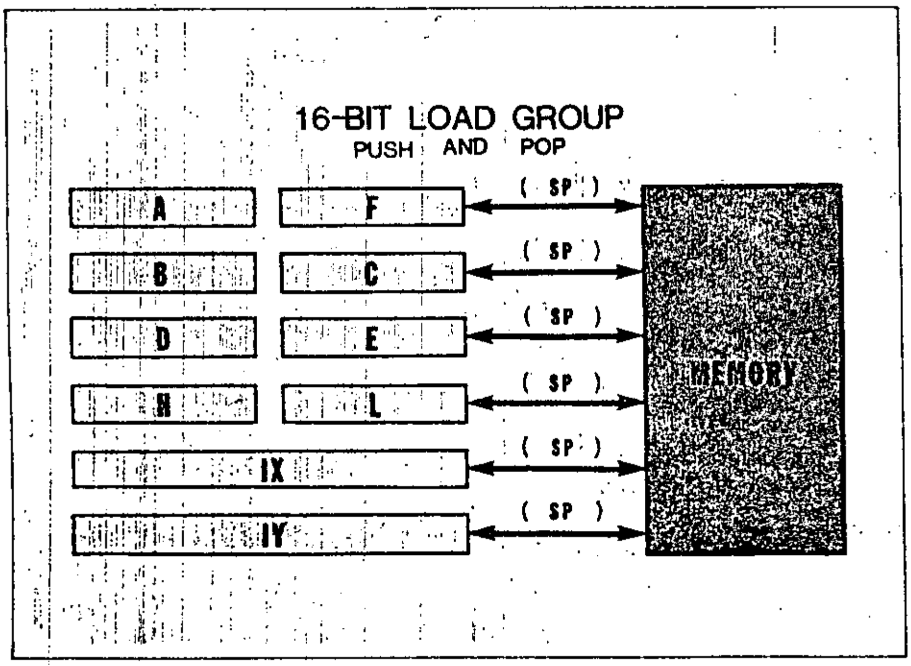
NOTES:

The index registers provide two additional memory pointers to transfer data between memory and the CPU's registers.



16—BIT LOAD GROUP

NOTES:



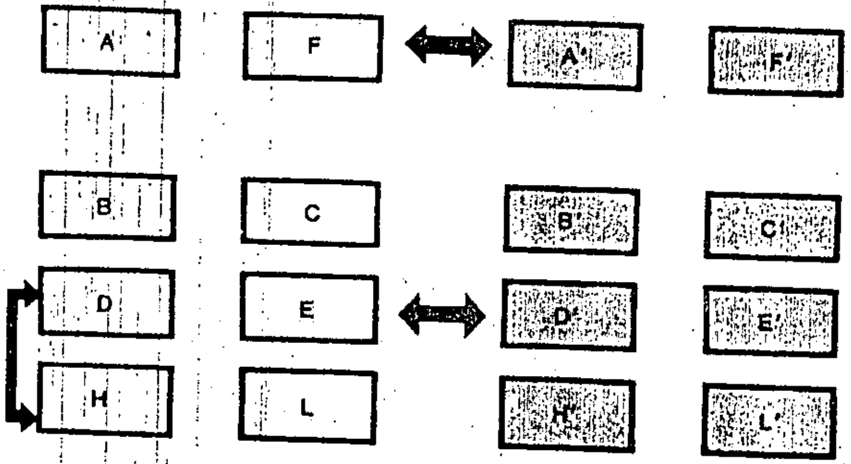
16-BIT LOAD GROUP

PUSH AND POP

NOTES:

□ All 16 bit registers can be PUSHed on or POPed off of the Stack thus allowing several tasks to use the same registers (re-entrent programs).

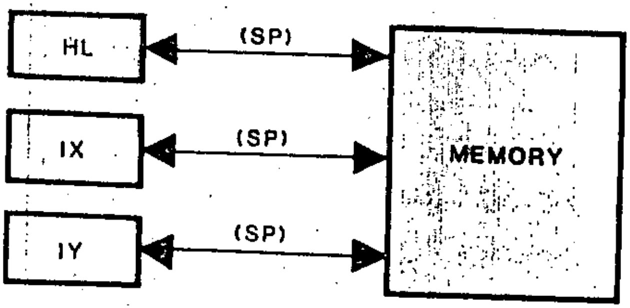
## EXCHANGE GROUP



The use of the exchange instructions in the Z80 can provide very fast interrupt service routines.

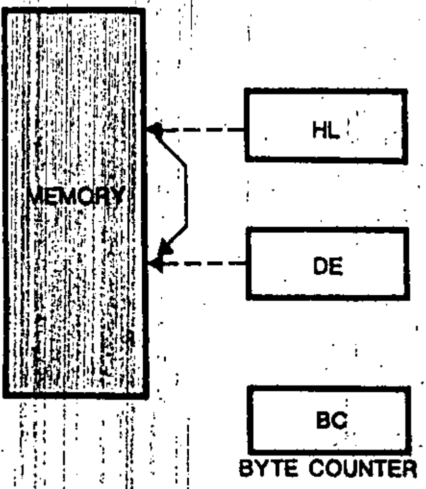
NOTES:

## EXCHANGE GROUP



NOTES:

## BLOCK TRANSFER GROUP



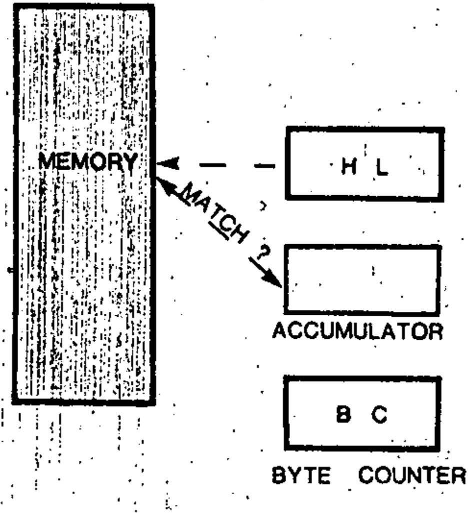
1. INC HL & DC. DEC BC
2. INC HL & DC. DEC BC. REPEAT UNTIL BC = 0
3. DEC HL & DE, DEC BC
4. DEC HL & DE, DEC BC. REPEAT UNTIL BC = 0

□ These instructions are very useful in most terminal applications where data movement is a large task for the microprocessor.

□ These single instructions in the Z80 are equivalent to subroutines in second generation architectures and can move data at a rate of 8.4 $\mu$sec/Byte.

NOTES:

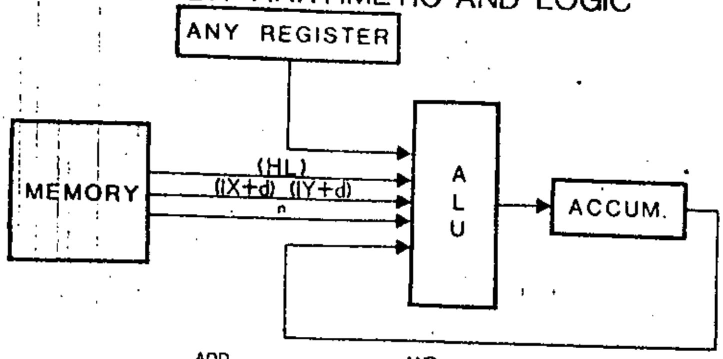
## BLOCK SEARCH GROUP



1. INC HL.DEC BC
2. INC HL.DEC BC.REPEAT UNTIL'MATCH' OR BC=0
3. DEC HL.DEC BC
4. DEC HL.DEC BC.REPEAT UNTIL'MATCH' OR BC=0

NOTES:

## 8 BIT ARITHMETIC AND LOGIC



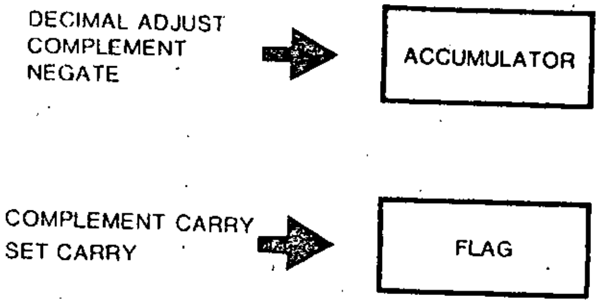
ADD
ADC
SUB
SBC

INC
DEC

AND
XOR
OR
CP

NOTES:

## GENERAL PURPOSE AF OPERATIONS



DECIMAL ADJUST
COMPLEMENT
NEGATE

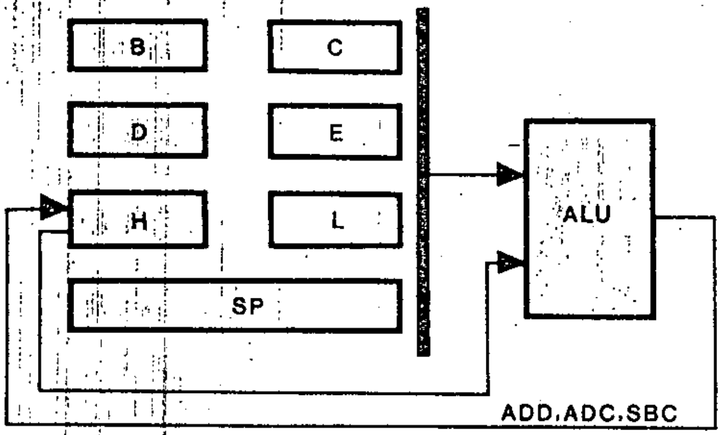
ACCUMULATOR

COMPLEMENT CARRY
SET CARRY

FLAG

NOTES:

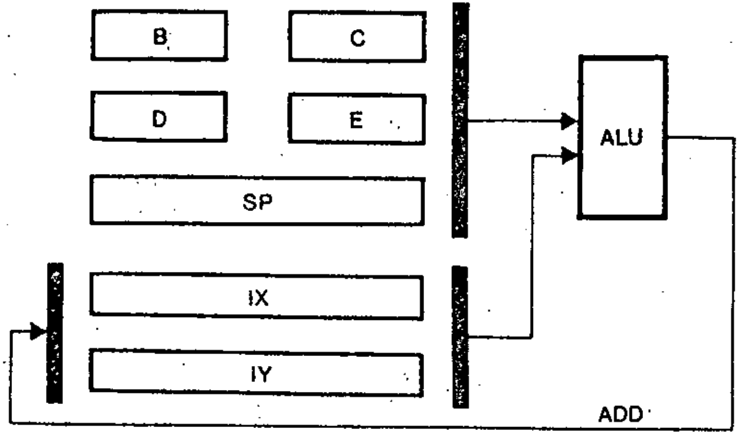
## 16 BIT ARITHMETIC



NOTES:

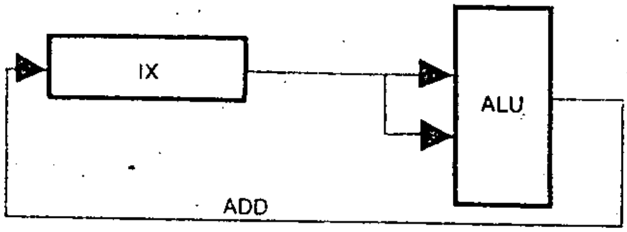
□ The addition of the ADC and SBC instructions in the Z80 allows more efficient multi-precision math routines by using the HL register pair as a 16 bit accumulator.

## 16-BIT ARITHMETIC



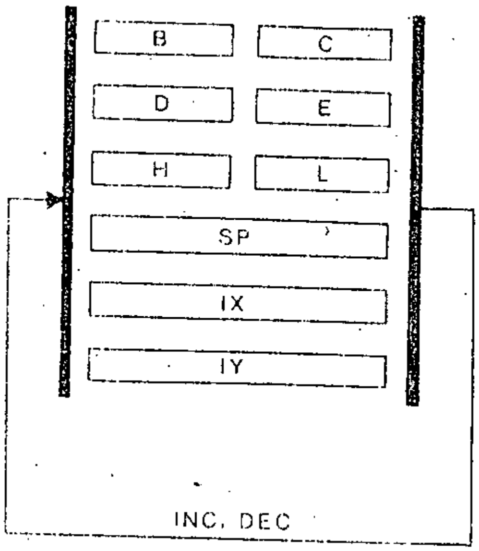
NOTES:

## 16-BIT ARITHMETIC

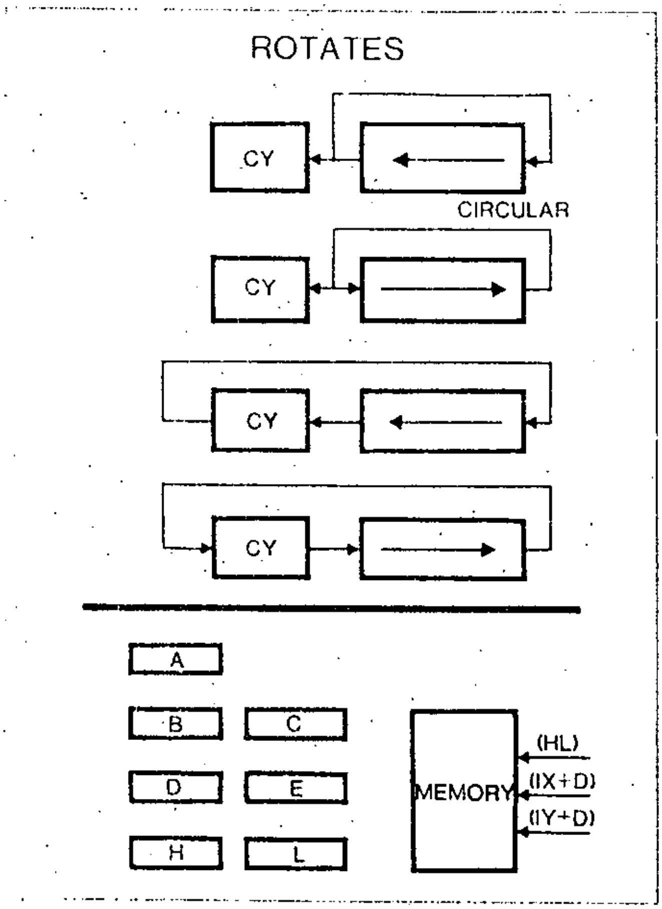


NOTES:

## 16-BIT ARITHMETIC

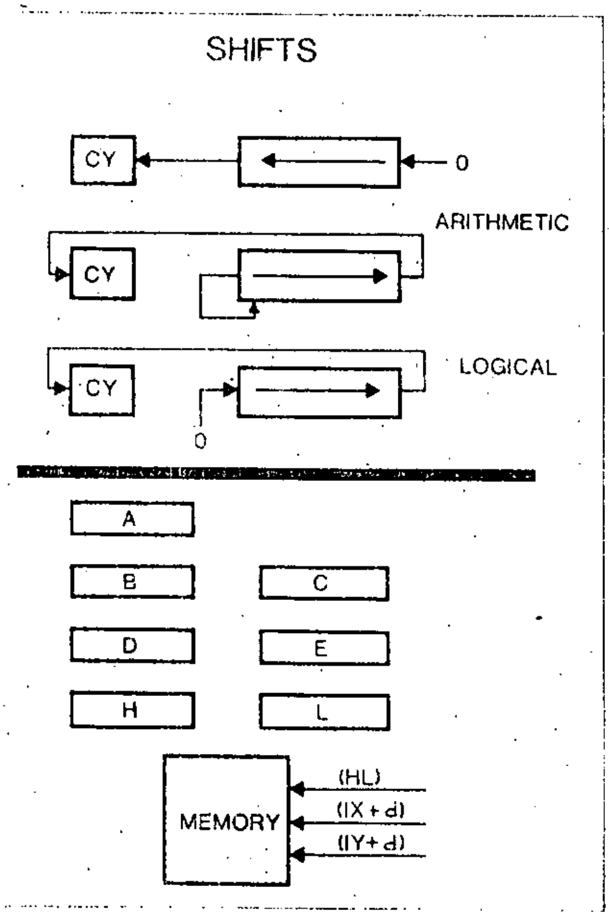


NOTES:

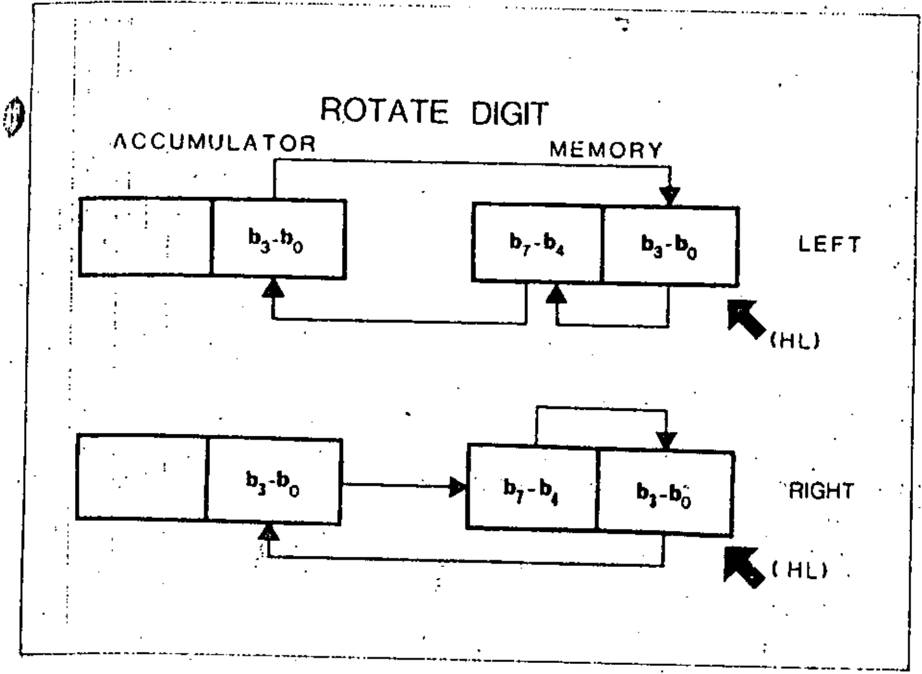


(1) The Z80 Rotate and Shift instructions operate on all memory locations as well as all internal registers.

NOTES:



NOTES:



ROTATE DIGIT

ACCUMULATOR

MEMORY

$b_3$-$b_0$

$b_7$-$b_4$

$b_3$-$b_0$

LEFT

(HL)

$b_3$-$b_0$

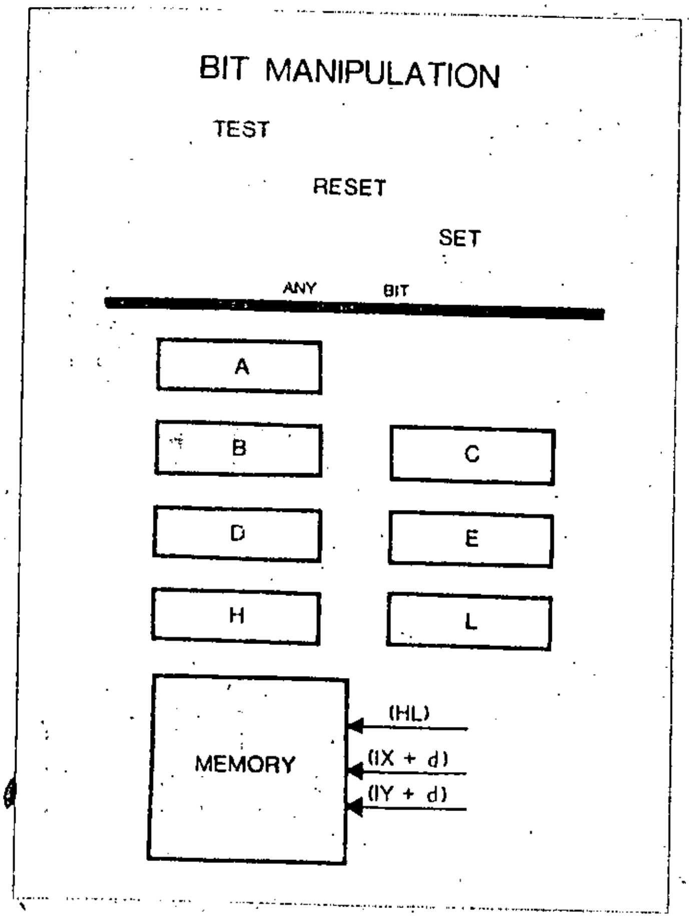
$b_7$-$b_4$

$b_3$-$b_0$

RIGHT

(HL)

NOTES:



BIT MANIPULATION

TEST

RESET

SET

ANY BIT

A

B

C

D

E

H

L

MEMORY

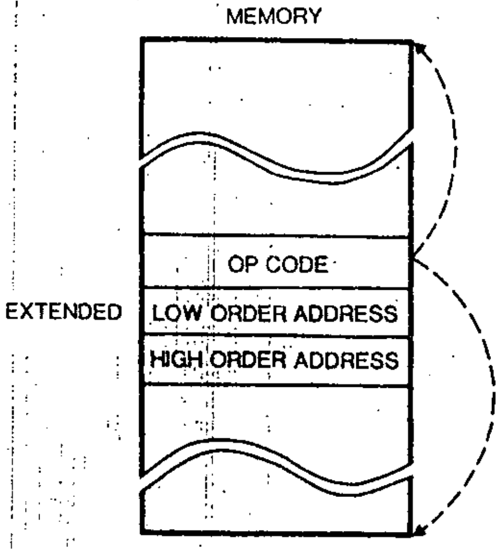
(HL)

(IX + d)

(IY + d)

NOTES:

## JUMP



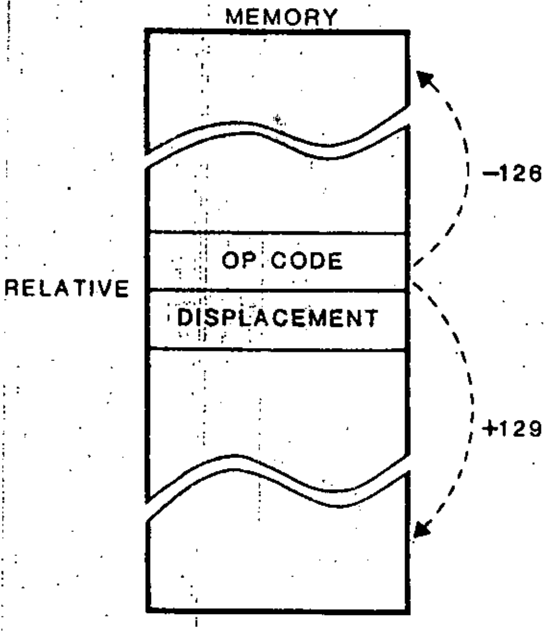
CONDITIONAL

CARRY
ZERO
SIGN
PARITY/OVERFLOW

UNCONDITIONAL

NOTES:

## JUMP



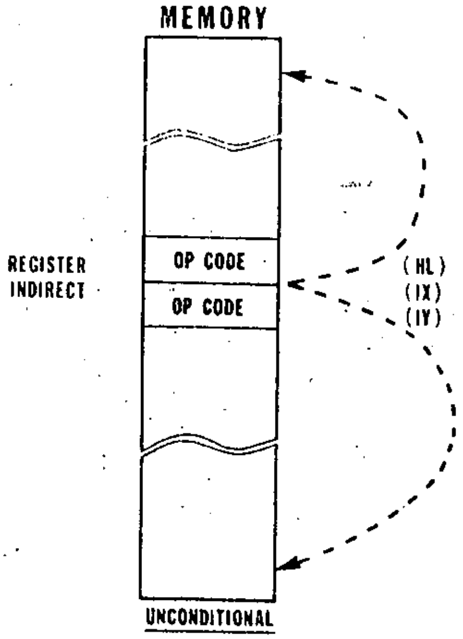
CONDITIONAL

CARRY
ZERO
DEC B JUMP IF NON ZERO

UNCONDITIONAL

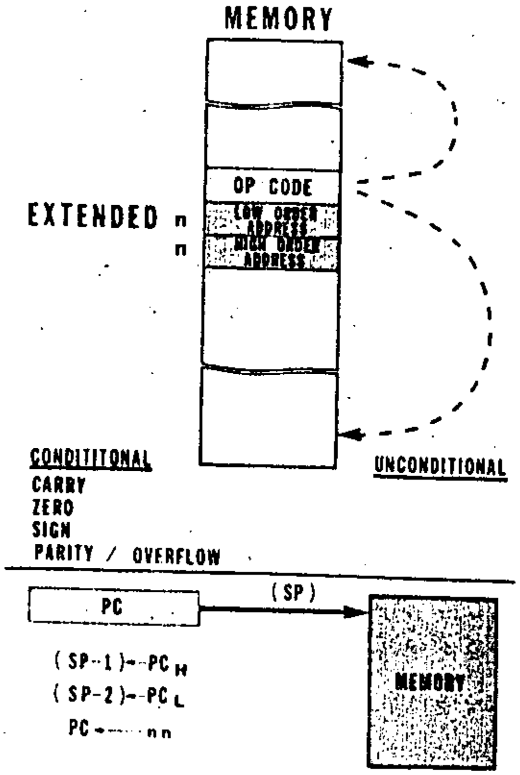
NOTES:

## JUMP



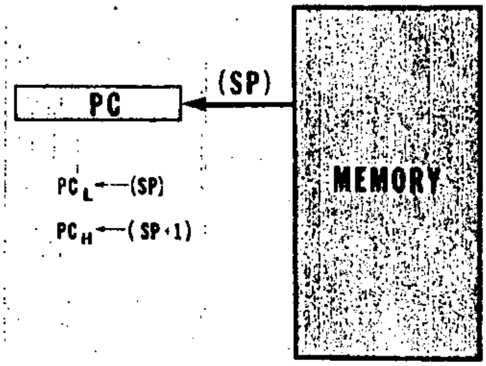
NOTES:

## CALL



NOTES:

## RETURN



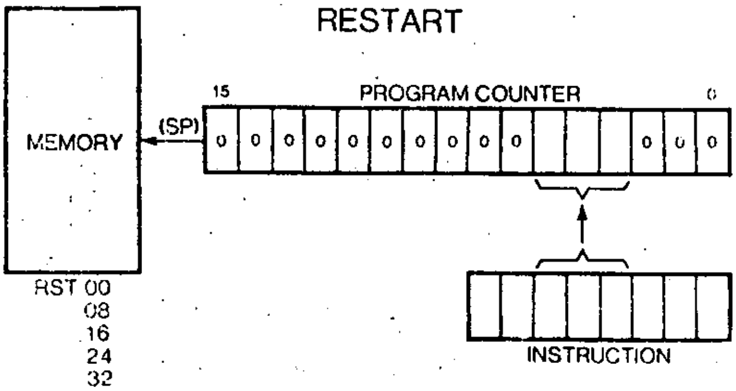
$PC_L \leftarrow (SP)$

$PC_H \leftarrow (SP+1)$

| CONDITIONAL | UNCONDITIONAL |
|---|---|
| CARRY | |
| ZERO | |
| SIGN | RETURN FROM INTERRUPT |
| PARITY / OVERFLOW | RETURN FROM NON-MASKABLE INTERRUPT |

NOTES:

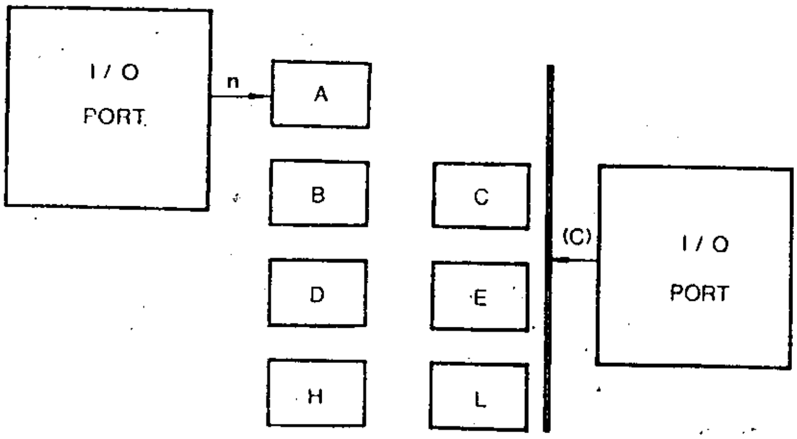
## RESTART



RST 00
08
16
24
32
40
48
$56_{10}$

NOTES:

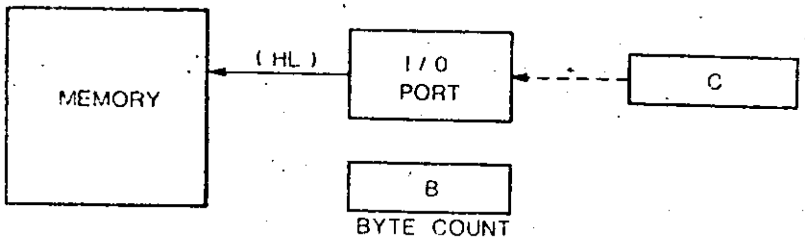
## INPUT GROUP



NOTES:

The Z80 can input data from any port to any eight bit CPU register thus eliminating the bottleneck in the Accumulator.

## BLOCK INPUT

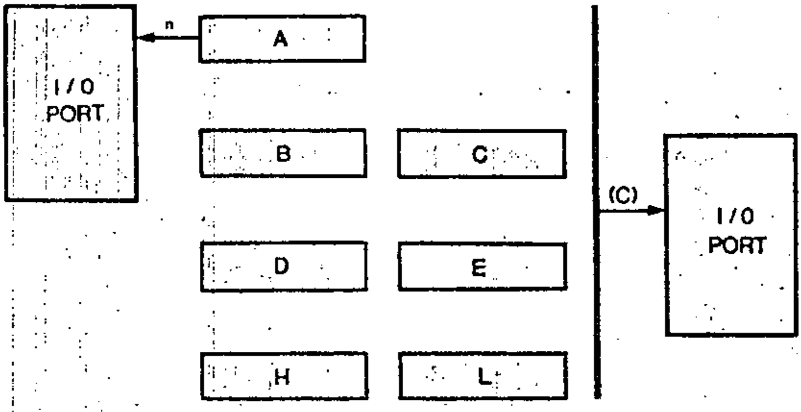


1. INC HL , DEC B
2. INC HL , DEC B , REPEAT IF B ≠ O
3. DEC HL , DEC B
4. DEC HL , DEC B, REPEAT IF B≠O

NOTES:

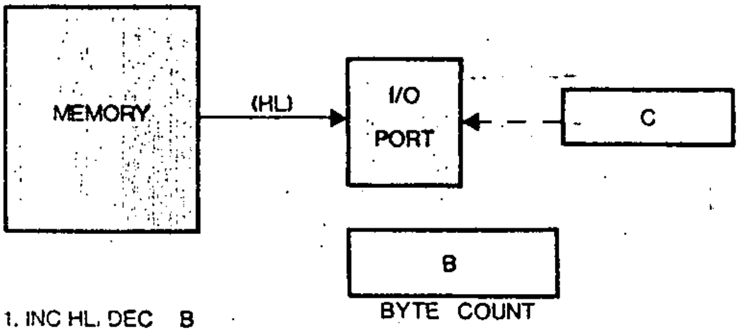
The Block Input or Output instructions can be thought of as a "Software" DMA. Data is moved from a peripheral to memory at a rate of 8 $\mu$s/Byte.

## OUTPUT GROUP



NOTES:

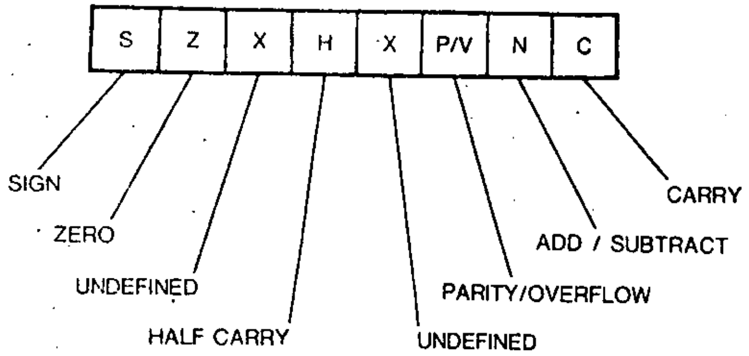
## BLOCK OUTPUT



1. INC HL, DEC B
2. INC HL, DEC B, REPEAT IF B≠O
3. DEC HL, DEC B
4. DEC HL DEC B, REPEAT IF B ≠ O

NOTES:

## Z80 FLAG REGISTER



S Z X H X P/V N C

SIGN

ZERO

UNDEFINED

HALF CARRY

UNDEFINED

PARITY/OVERFLOW

ADD / SUBTRACT

CARRY

NOTES:

## HOW TO SAVE MEMORY BYTES IN YOUR 8080 PROGRAM WITH THE Z80

□ Use Block Instructions for auto updating of memory pointers.

□ Exchange instructions speed up Interrupt Service Routines with fewer Bytes.

□ Index Register provides additional memory pointers plus indexed mode of addressing.

□ New 16 bit Arithmetic instructions perform multi-precision math with fewer bytes.

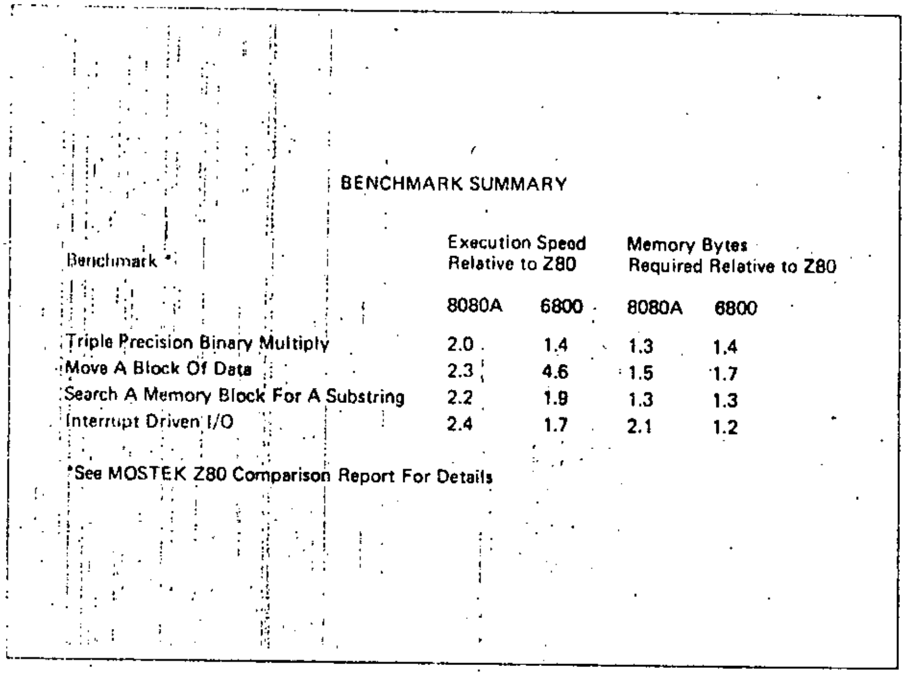
□ Rotates and Shifts can be done on all registers as well as external memory to avoid the Accumulator bottleneck.

□ Bit Addressing saves masking out of bits in the Accumulator.

□ Relative Addressing saves Bytes in short loops and allows relocatable programs.

□ In and Out instructions using an Indirect Port Address allows one I/O Routine to service multiple peripherals.

NOTES:

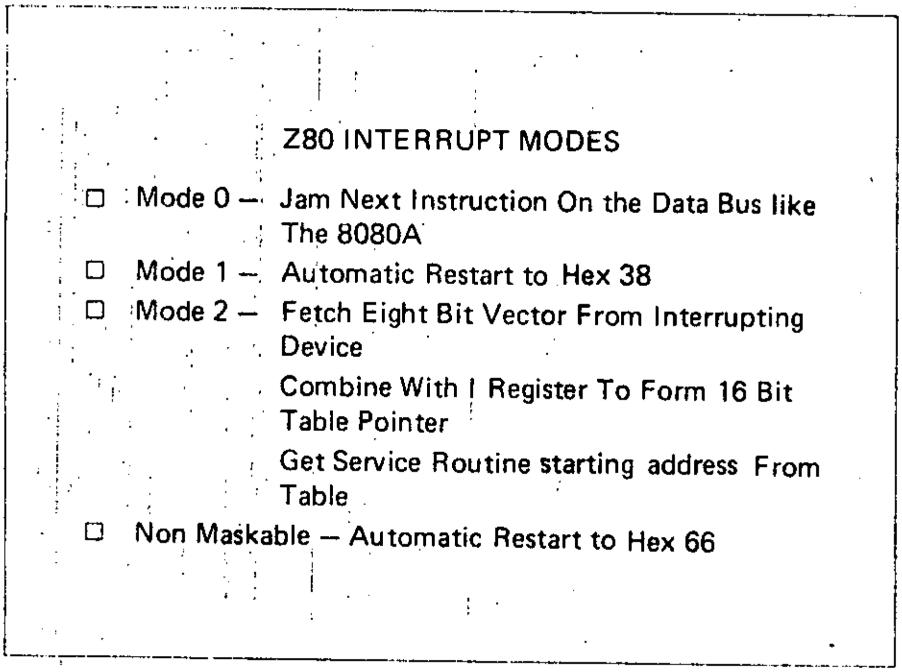


BENCHMARK SUMMARY

| Benchmark * | Execution Speed Relative to Z80 | | Memory Bytes Required Relative to Z80 | |
|---|---|---|---|---|
| | 8080A | 6800 | 8080A | 6800 |
| Triple Precision Binary Multiply | 2.0 | 1.4 | 1.3 | 1.4 |
| Move A Block Of Data | 2.3 | 4.6 | 1.5 | 1.7 |
| Search A Memory Block For A Substring | 2.2 | 1.9 | 1.3 | 1.3 |
| Interrupt Driven I/O | 2.4 | 1.7 | 2.1 | 1.2 |

*See MOSTEK Z80 Comparison Report For Details

NOTES:



Z80 INTERRUPT MODES

- □ Mode 0 — Jam Next Instruction On the Data Bus like The 8080A
- □ Mode 1 — Automatic Restart to Hex 38
- □ Mode 2 — Fetch Eight Bit Vector From Interrupting Device
  Combine With I Register To Form 16 Bit Table Pointer
  Get Service Routine starting address From Table
- □ Non Maskable — Automatic Restart to Hex 66

NOTES:

□ Mode 2 is used by the Z80 Peripheral devices to form a powerful interrupt structure while eliminating the need for an external interrupt controller.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# MSC-48 SINGLE COMPONENT MICROCOMPUTER

MARZO, 1979

## THE MCS-48™ FAMILY

Notes:

8048 — MICROCOMPUTER WITH ROM

8748 — MICROCOMPUTER WITH EPROM

8035 — MICROCOMPUTER WITHOUT ROM

8243 — I/O EXPANDER

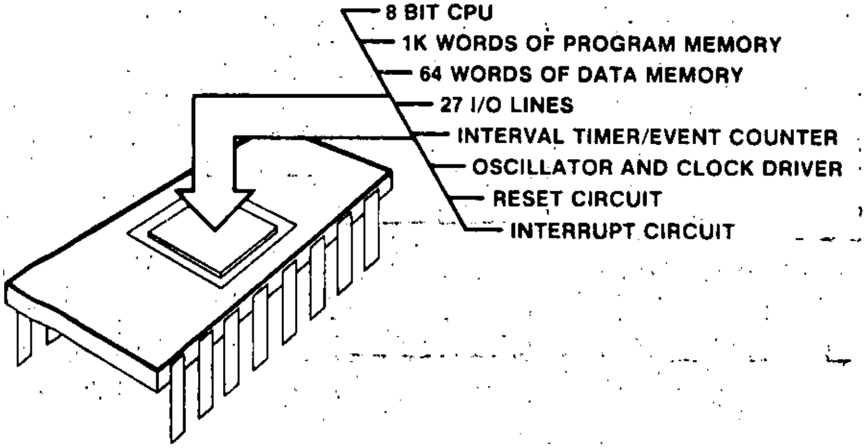
8355 — ROM PROGRAM MEMORY AND I/O EXPANDER

8755 — EPROM PROGRAM MEMORY AND I/O EXPANDER

8155 — DATA MEMORY AND I/O EXPANDER

- *The Basic Family will be expanded with additional I/O and processor elements.*
- *Most 8080 peripherals and standard memory products are directly compatible.*

## ON CHIP FEATURES



8 BIT CPU

1K WORDS OF PROGRAM MEMORY

64 WORDS OF DATA MEMORY

27 I/O LINES

INTERVAL TIMER/EVENT COUNTER

OSCILLATOR AND CLOCK DRIVER

RESET CIRCUIT

INTERRUPT CIRCUIT

*Totally self contained. All that is required is 5 Volts.*

## SPECIAL FEATURES

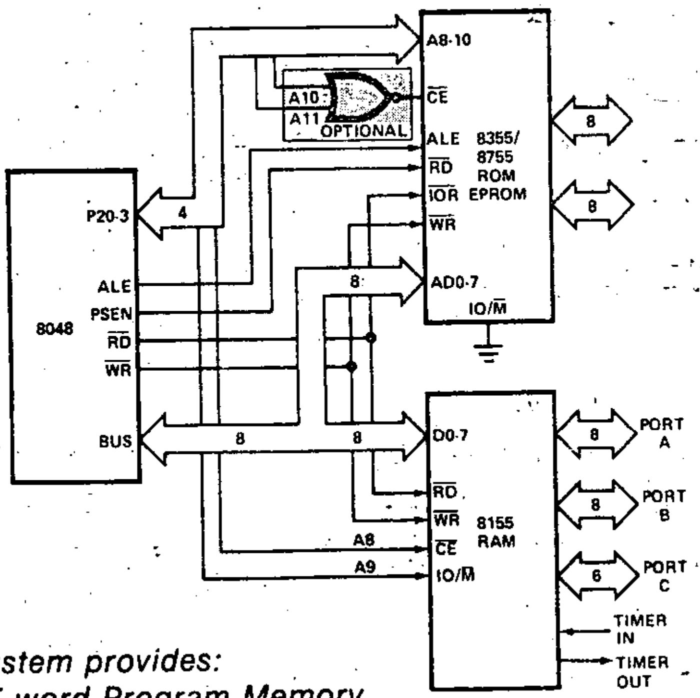
- SINGLE 5V SUPPLY
- 40 PIN DIP
- PIN COMPATIBLE ROM AND EPROM
- 2.5 μsec CYCLE
- ALL INSTRUCTIONS 1 OR 2 CYCLES
- SINGLE STEP
- 8 LEVEL STACK
- 2 WORKING REGISTER BANKS
- RC, XTAL, OR EXTERNAL FREQUENCY SOURCE
- ÷ 3 OR ÷ 15 CLOCK OUTPUT

- *All contained on the single chip.*
- *Low power standby on ROM version.*

## THE THREE COMPONENT MCS-48 SYSTEM

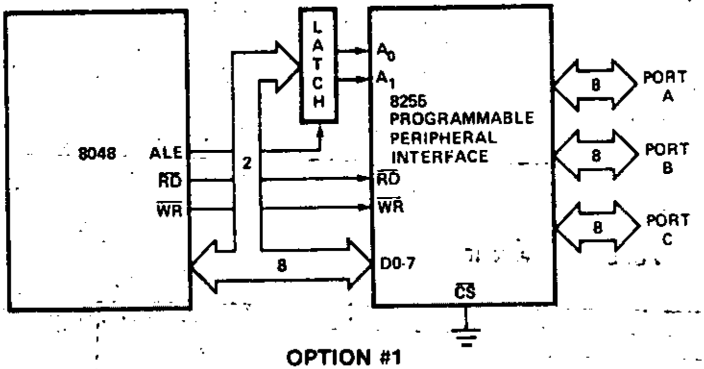
8155/8355

Notes:

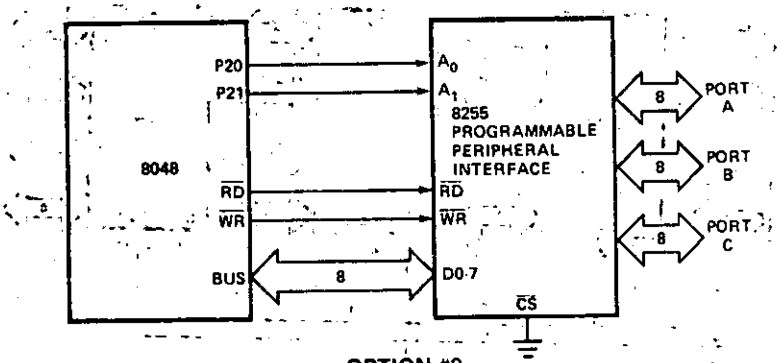


- *System provides:*
  *3K word Program Memory*
  *320 word Data Memory*
  *53 I/O Lines*
  *2 Timer/Counters*

## INTERFACE TO MCS-80 PERIPHERALS



OPTION #1



OPTION #2

- *OPTION #1 - Ports are addressed as data RAM*
- *OPTION #2 - Ports are addressed via output port*

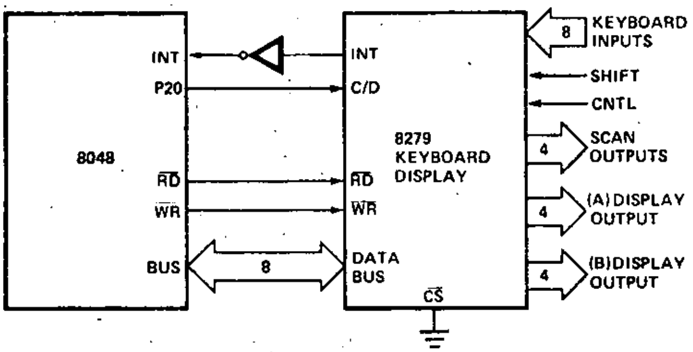
## SPECIAL PERIPHERALS

Notes:

| | |
|---|---|
| 8214 | Priority Interrupt |
| 8251 | USART |
| 8279 | Keyboard/Display |
| 8253 | Interval Timer |

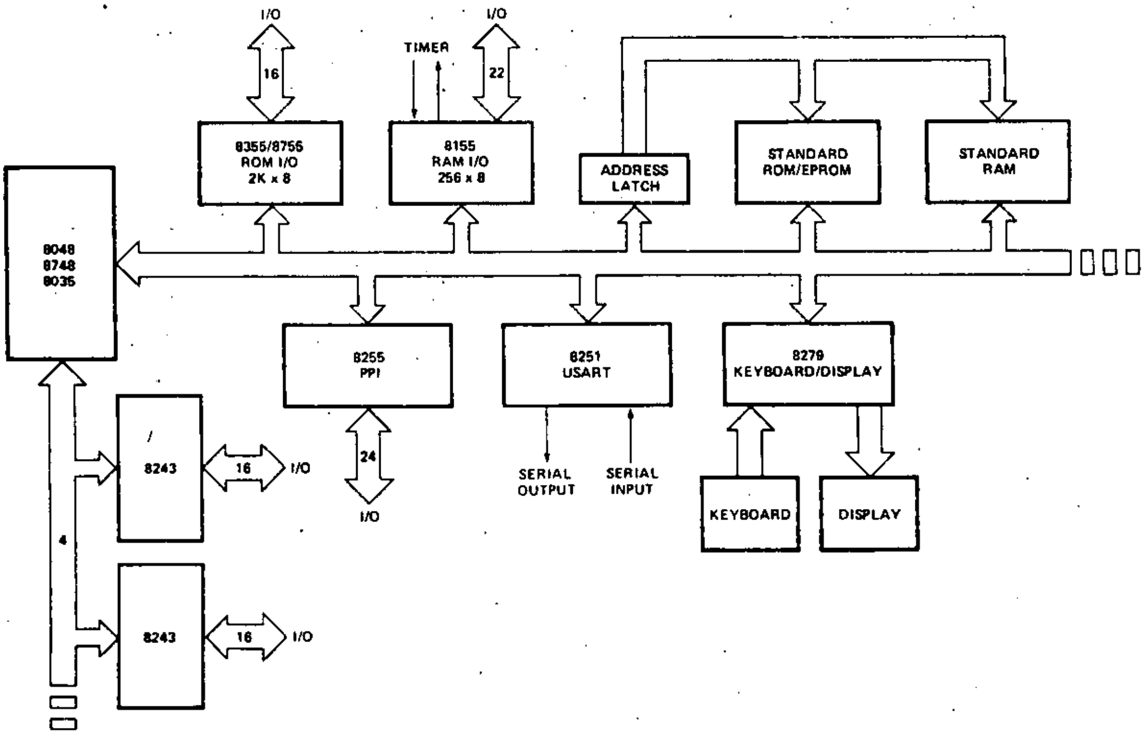
- *While most MCS-80 peripherals are MCS-48 compatible, the ones shown are the most popular.*
- *Peripherals reduce system cost by providing low component count and specialized interfaces.*

KEYBOARD/DISPLAY INTERFACE



- *Provides easy interface to a 64 key matrix and two 16 digit displays.*

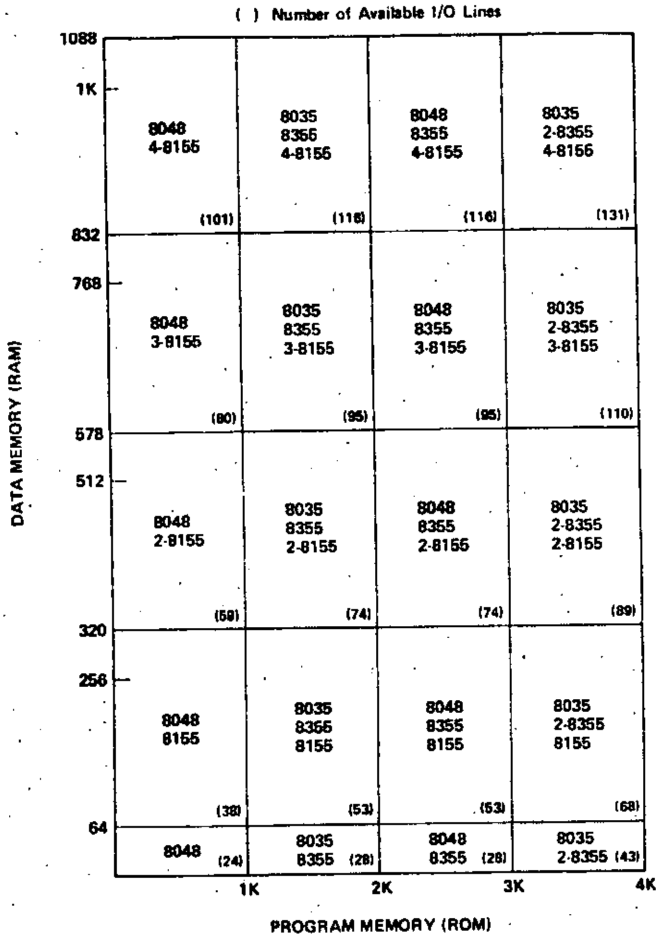
MCS-48 EXPANSION CAPABILITY



- *All modes of expansion may be used simultaneously.*

Notes:

THE EXPANDED MCS-48™ SYSTEM



- *The 8035 allows the user to match his program memory requirements exactly.*

## DEVELOPMENT SUPPORT

- **INTELLEC® Assembler**
- **UPP PROM Programmer**
- **PROMPT 48**
- **ICE-48™**
- **User's Library**
- **Application Engineers**
- **Training Courses**

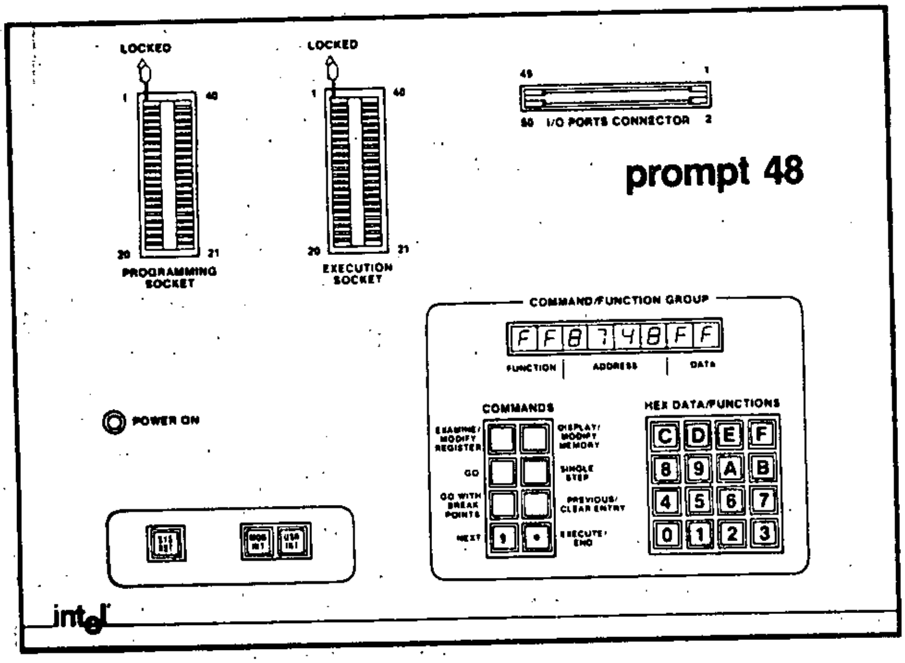
- *Development support is as important as components.*
- *MCS-48™ training is available and affords the best opportunity to learn the family.*

## PROMPT 48 FEATURES

Notes:

- 8748 EPROM Programmer
- Hex Keyboard/Display
- Real Time Execution
- Single Step
- 8 Break Points
- Examine/Modify all internal Registers
- TTY Interface (or RS232)
- Hex Tape Load/Dump (TTY)
- All I/O Ports available to User

- *Low cost.*
- *Self contained bench top enclosure.*



## INTELLEC® MICROCOMPUTER DEVELOPMENT SYSTEM FEATURES

- Resident MCS-48 Macro Assembler
- Universal PROM Programmer Module
- ICE-48™
- Disk Operating System
- High Speed Peripherals

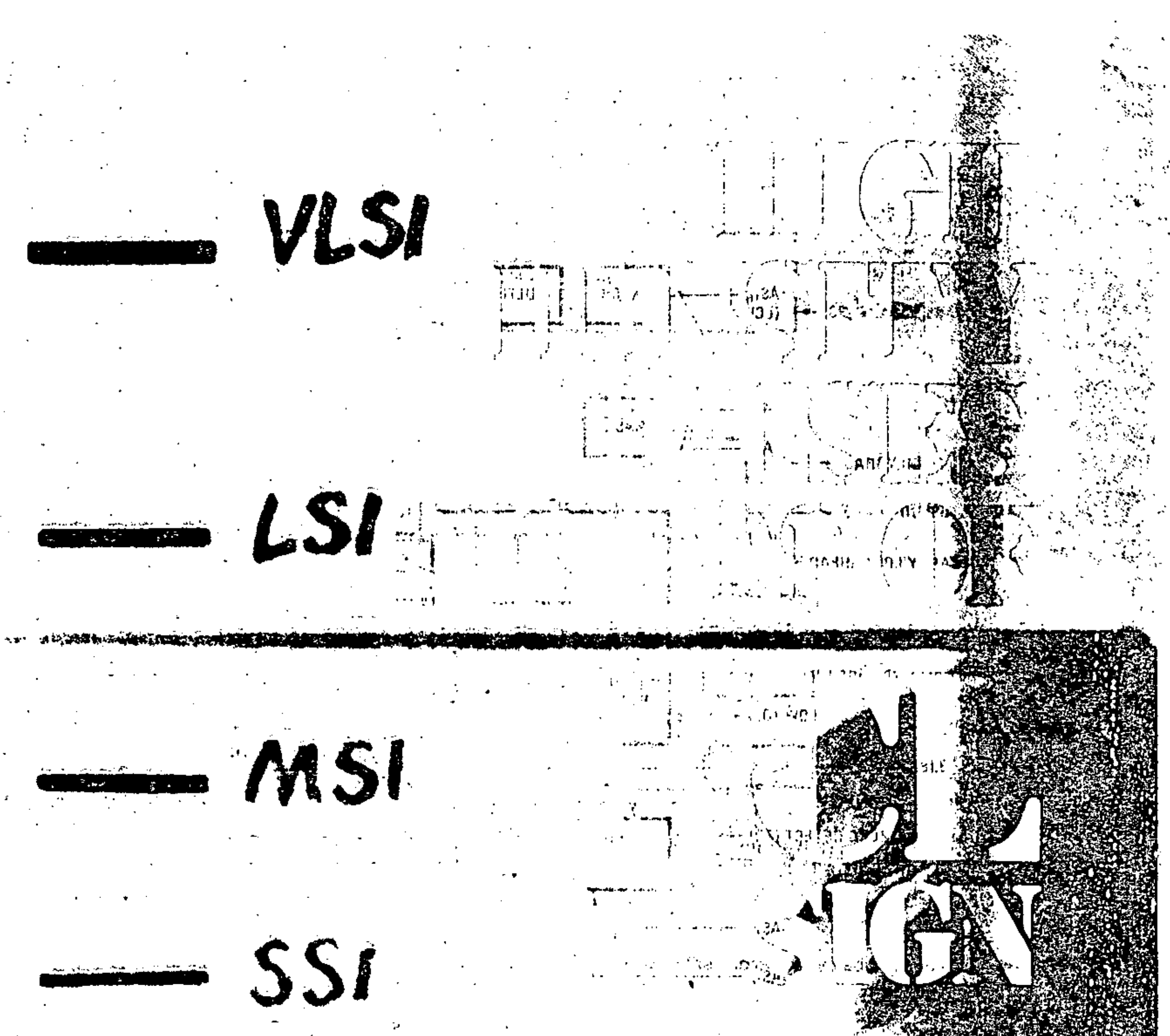
- *A self contained microcomputer development laboratory.*

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

# MICROPROCESADORES: TEORIA Y APLICACIONES

## ELECTRONICS
FEBRUARY

MARZO, 1979



by William R. Blood, Jr. *Motorola Inc., Integrated Circuits Division, Mesa, Ariz.*

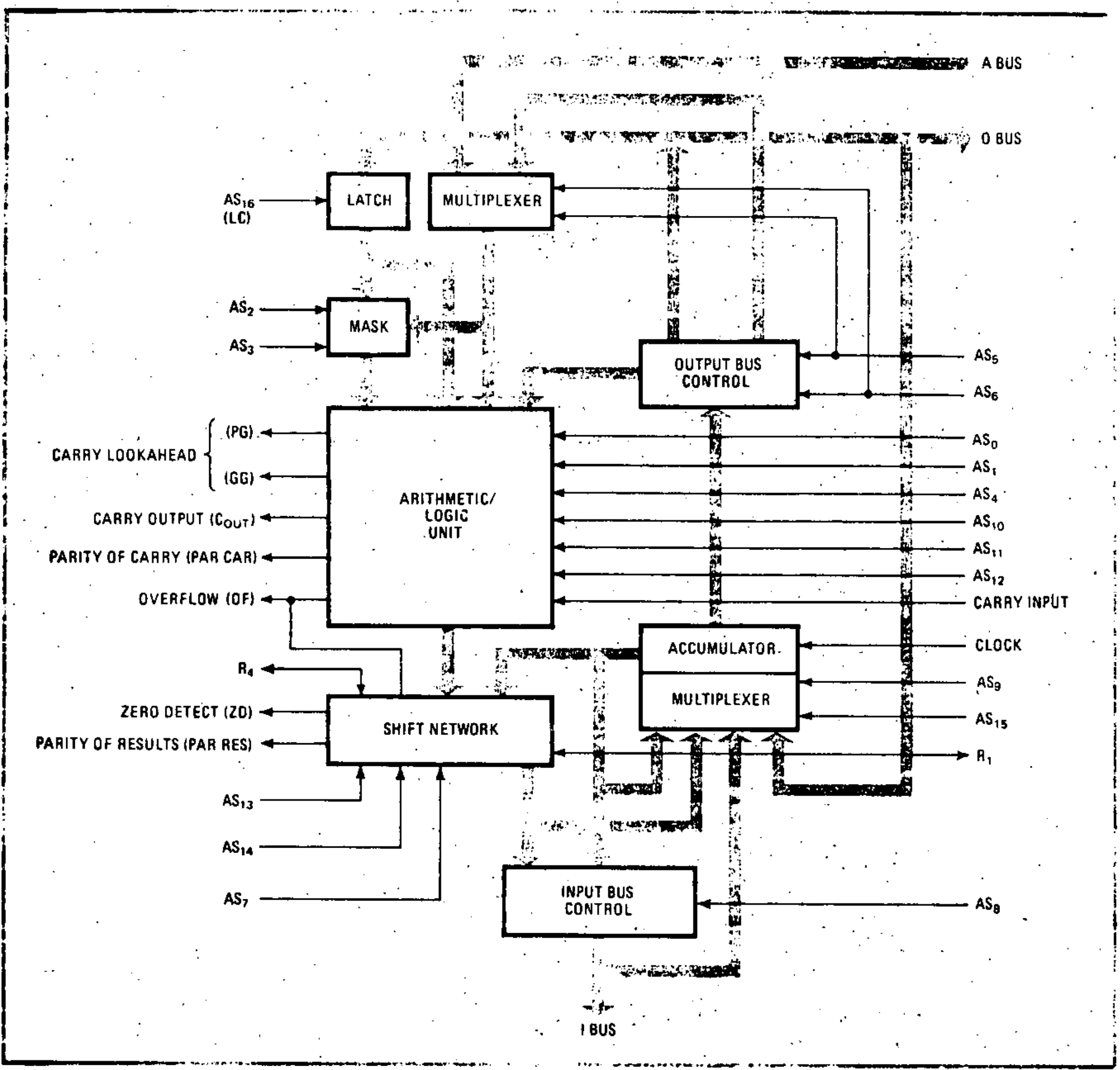
□ In the realm of microprocessors, bipolar large-scale integrated circuitry has evolved very differently from the metal-oxide-semiconductor LSI technologies. Its big selling point is its speed, which can only be optimized for any given microprocessor application if the designer has control of the processor's bus structure, word size, and instruction set. The need for such control has led to the bit-slice approach in bipolar LSI circuits, which is quite unlike the more general-purpose byte orientation of slower MOS microprocessors.

The fastest bipolar technology is emitter-coupled logic, and ECL is the basis for the M10800 family of standard bit-slice parts.

### The 10800 family

There are nine members in the 10800 ECL bit-slice family. Each handles 4-bit-wide data paths, but since each is designed around the slice concept, it can parallel itself to build a processor of any given word width. Moreover, each contains data ports for easy interconnection to other LSI circuits. The family includes:

- The MC10800 basic 4-bit arithmetic-and-logic unit.
- The MC10801 microprogram-control circuit.
- The MC10802 timing controller.



**1. Processor element.** The MC10800 4-bit arithmetic-and-logic unit slice is the heart of Motorola's family of high-speed emitter-coupled-logic circuits. Structured around three buses, two of which are bidirectional, the chip is capable of handling binary and binary-coded-decimal data.

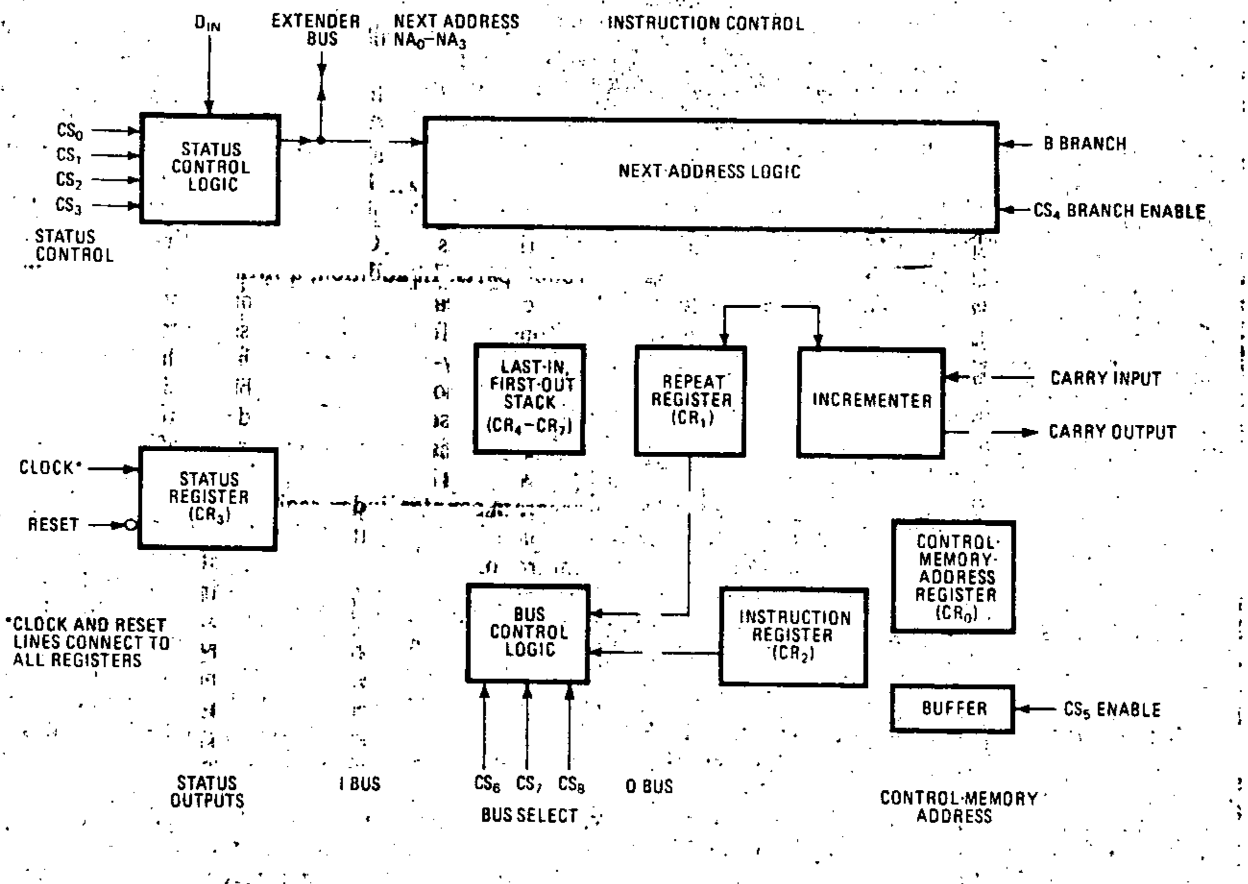
- The MC10803 memory-interface circuit.
- The MC10804 and MC10805, which are 4- and 5-bit level translators for hooking ECL to TTL.
- The MC10806, a dual-access buffer memory.
- The MC10807 5-bit bus transceiver.
- The MC10808 programmable multibit shifter.

The parts also hook onto compatible ECL memories, such as the MCM10146 1,024-bit random-access memory.

The speed of the family is mainly attributable to new circuit design techniques, rather than any breakthrough in integrated-circuit processing. An example is the internal logic that operates off a −2-volt supply, which is better suited than a 5-V supply to the operation of multiplexers, registers, and some other commonly used logic elements. Those elements can thus be easily integrated with those circuit elements like adders that are better built with the series-gated ECL structures powered by the conventional −5.2-V supply. Further, since the 10800 family of parts employs the same fabrication process as the MCM10146 high-speed 1,024-by-1-bit ECL RAM, they enjoy all the benefits of long-established, high-volume production.

## The ALU chip

At the heart of the family is the MC10800 4-bit arithmetic-and-logic-unit (ALU) slice, which was the first in the family of standard ECL products to be developed. The chip performs the logic, arithmetic, and shift functions required to execute various machine instructions. Because the part was the first built with the new −2-V logic design, circuit complexity was held to the equivalent of a conservative 350 gates. The area of the chip, which employs standard design rules and double-layer metalization, is less than 15,000 square mils.



**2. Controller.** The MC10801 microprogram-control chip, which is also a 4-bit-wide slice, generates the microprogram address and provides the logic for complete sequence control. Five address buses interface to microprogram memory, to other parts and to external test points.

The 10800 operates with three data ports, as shown in Fig. 1. The I and the O buses are both 4 bits wide and bidirectional. The third, the A bus, is a 4-bit-wide input-only port. Control of the ALU is by 17 select lines, $AS_0$ through $AS_{16}$. The select lines control all circuit functions and determine the source and destination for ALU data. A full set of condition-code outputs, which include parity, carry, overflow, and zero-detect, simplify branch testing. Unique among bit-slice processor elements is the 10800's ability to perform both binary and binary-coded-decimal (BCD) arithmetic with equal ease and speed. Direct BCD arithmetic is gaining in popularity in business computers, process controllers, and test systems where human interface is most often in a BCD format. The chip also features a signal overflow shift network that indicates when an arithmetic left shift has prompted a sign-bit change.

The MC10801 microprogram-control chip is the companion part to the processor element and carries out the sequencing of operations. The development of the 10801, which packs 550 equivalent gates onto a 25,000-square-mil chip, proceeded all the more confidently because of the high yields already obtained with the less complex 10800 part.

The 10801, also a 4-bit-wide slice, is shown in Fig. 2. The control-memory-address register $CR_0$ holds the microprogram memory address, while the remaining blocks in the figure provide logic for the sequencing operation. Register $CR_1$, called the repeat register, is a special feature that adds greatly to the 10801's speed and flexibility. Aside from its usefulness as a cycle counter, which allows single instructions or subroutines to be executed a specific number of times by automatically keeping track of loop count, testing for end count, and remaining in or leaving the loop on test result, register $CR_1$ further provides a return destination for microprogram interrupts.

Register $CR_2$ is set up to hold a machine instruction starting address or an interrupt vector. Register $CR_3$, however, is unique to the 10801 in that it interfaces microprogram control to external test points. The register can be loaded with any given bits of status information, whereupon it will test those bits for conditional microprogram jumps. Moreover, its contents can be set or cleared under program control to signal the processor's status. Finally, $CR_3$ can hold the page address in a word- or page-organized microprogram. In that case, memory address register $CR_0$ would hold only the microprogram word address.

Registers $CR_4$–$CR_7$ form a four-word last-in, first-out (LIFO) stack for nesting subroutines within a program. With the logic built into the 10801, operation of the LIFO is completely automatic. If needed, however, the LIFO can be extended or tested for full stack through the I bus

| | |
|---|---|
| INC | Increment |
| JMP | Jump to next address inputs |
| JIB | Jump to I bus |
| JIN | Jump to I bus and load $CR_2$ |
| JPI | Jump to primary instruction ($CR_2$) |
| JEP | Jump to external port (O bus) |
| JL2 | Jump to next address inputs and load $CR_2$ |
| JLA | Jump to next address inputs and load address into $CR_1$ |
| JSR | Jump to subroutine |
| RTN | Return from subroutine |
| RSR | Repeat subroutine (load $CR_1$ from next-address inputs) |
| RPI | Repeat instruction |
| BRC | Branch to next-address inputs on condition; otherwise increment |
| BSR | Branch to subroutine on condition; otherwise increment |
| ROC | Return from subroutine on condition; otherwise jump to next-address inputs |
| BRM | Branch and modify address with branch inputs (multiway branch) |

or the O bus ports. Two branch inputs—the branch (B) and extended branch ($\overline{XB}$)—supply status for conditional microprogram jumps.

The whole part is tied together with 16 instructions that are built into the next-address logic block. These instructions, listed in Table 1, control the source for each new microprogram word address and have been designed to save both microprogram memory size and development time. For example, an 8-bit shift in the ALU can be done with only two microprogram words—a repeat-subroutine instruction (RSR) to load the repeat number (8) into register $CR_1$, and a repeat instruction (RPI) to perform the eight shifts.

## Control of timing

Clock control, often one of the most complex segments of processor design, is implemented with a single chip—the MC10802. As shown in Fig. 3, the chip contains the logic to generate multiple phases, simplify system start and stop, and provide some diagnostic capability.

The 10802 takes a clock input, usually from a crystal oscillator, and splits the signal into separate phases with its four-phase shifter block. The number of different phases can be programmed for two, three, or four phases. The go/halt, run/maintenance, and start inputs control system start and stop operations. The single-cycle/single-phase input helps with diagnostics by advancing the system one clock phase or a complete cycle for each starting signal input. Finally, a synchronizer network built into the part eases interfacing to the start input.

Hooking the bit-slice processor to slower memory and peripherals calls for the MC10803 interface chip. The part is designed for maximum speed: it can simultaneously route data while addressing the memory or peripherals. With 600 equivalent gates packed into a 21,000-square-mil area, the 10803 is actually denser than the 10801 microcontroller. The difference is due to a less complex metalization pattern used in the memory interface chip.

The 10803 has its own ALU. Also organized as 4-bit slices, several 10803s may be connected in parallel to meet any particular system data and address requirements. As shown in Fig. 4, the circuit has data and address ports for interfacing to peripheral equipment, plus the I bus and O bus for connecting directly to other 10800 parts. In addition, a fifth port with pointer inputs to the ALU can be used as a source of address modifiers or constants for memory addressing.

A memory-address register holds the memory address while a memory-data register buffers incoming or outgoing data. A separate four-word register file stores information that is needed in the course of memory addressing, such as the page addresses or the value of the program counter, index register, or stack pointer.

Some select inputs to the data-interface logic have control of a total of 17 data-transfer operations between buses and registers. Other select inputs control the function of the ALU and determine the source and destination of data through microfunctions and destination-decoding logic. Although the ALU is normally used for memory addressing, it can perform seven basic functions—add, subtract, OR, AND, exclusive-OR, shift-left, and shift-right—on a wide variety of data sources.

Certain systems can take advantage of the 10803 to reduce parts count. A peripheral controller, for example, transfers and formats data and usually requires little arithmetic capability. Such a system uses the 10803 both for input/output (I/O) control and as its main ALU, eliminating the need for a 10800.

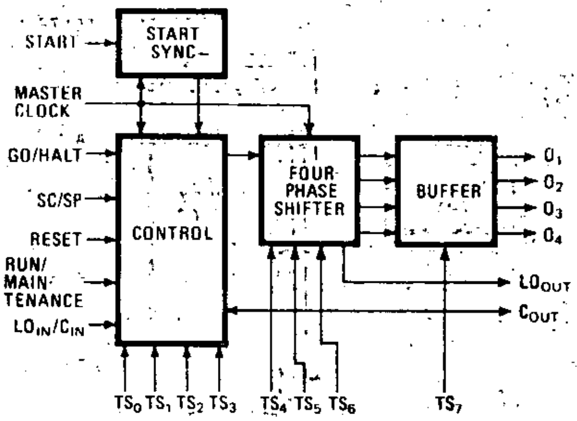
## Tying the parts into a system

The simplicity with which the parts of the 10800 bit-slice family can be assembled into a microprogrammable 16-bit minicomputer or signal processor is evident from Fig. 5. Although structures will vary depending on the application, the example illustrates several key features. Eleven LSI chips, together with a few medium-scale integrated parts and supporting high-speed memory devices, are all that is required.

Bus ports on the 10800, 10801, and 10803 directly interconnect. The 10801 microprogram controller supplies an address to microprogram memory, selecting one microprogram word. Each word is divided into groups of bits called fields (represented by the broad arrows at bottom). Each field independently controls a system section. Since all fields are present at the same time in each microprogram word, the various system sections can operate simultaneously for maximum system speed.

A system function performed by all the fields in one microprogram word is called a microinstruction. Several microinstructions may be required for one machine instruction. System performance, therefore, is determined by the number of microinstructions in a system and the speed of each microinstruction. Microinstruction cycle time for a 16-bit 10800-family-based system is about 100 nanoseconds.

The operation of the system in Fig. 5 can be explained



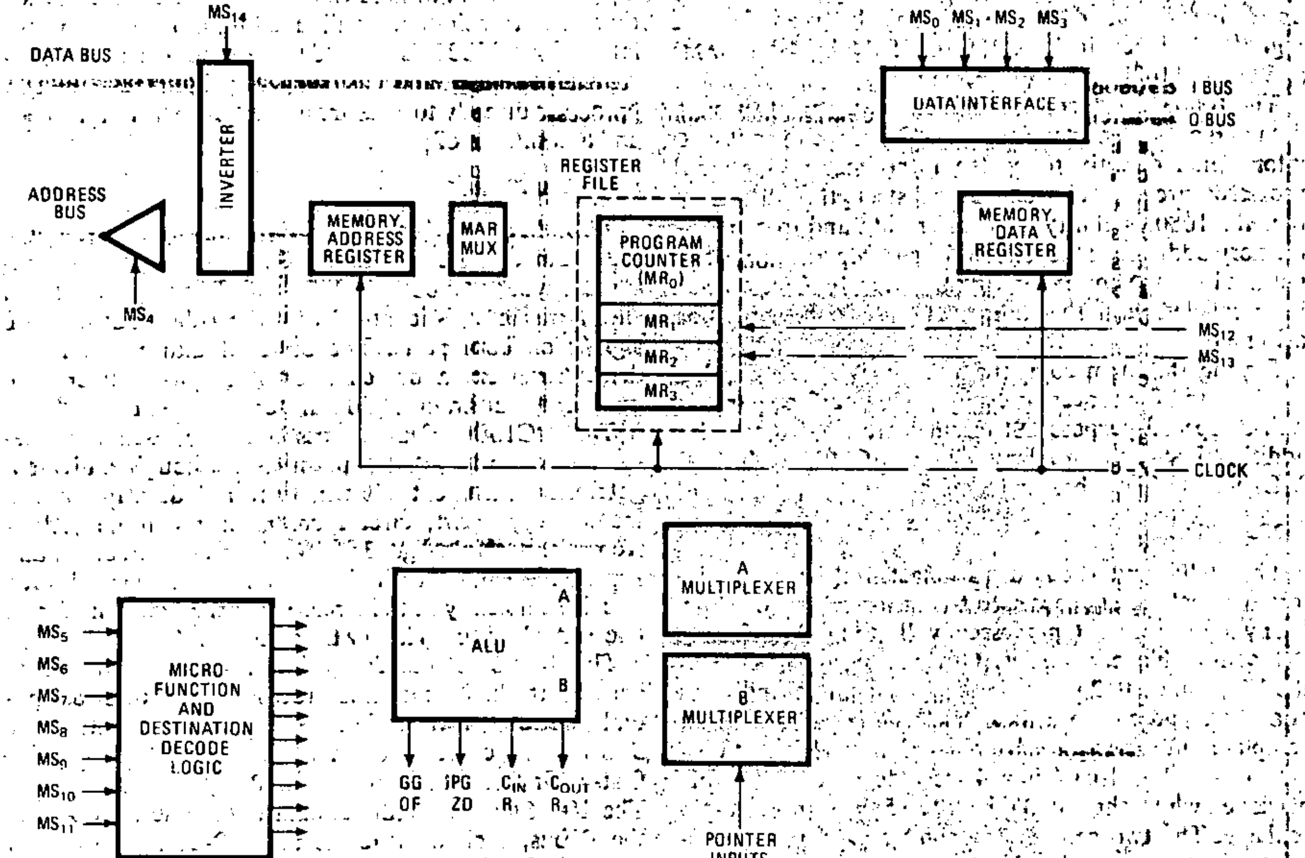
**3. Timer.** Many of the usual timing problems in bit-slice processor design are handled by the MC10802 timing function chip, including system start, stop, and clock control for diagnostics. A start synchronizer input simplifies interfacing to front-panel switches.

in terms of the relationship of microprogram fields to the LSI blocks. Two fields controlling the 10801 generate each new microprogram address. An instruction field selects one of the 16 program-flow instructions given in Table 1, and once selected, all logic needed to execute the instruction is contained in the 10801. However, some instructions require additional information. For example, a jump-to-next-address or jump-to-subroutine instruction requires a destination, which is supplied by the next-address microprogram field. An important feature of the 10801 is the ability to route next-address data through the O-bus port for ALU or memory interface constants, bit-mask patterns, and offsets when the field is not required for microprogram flow.

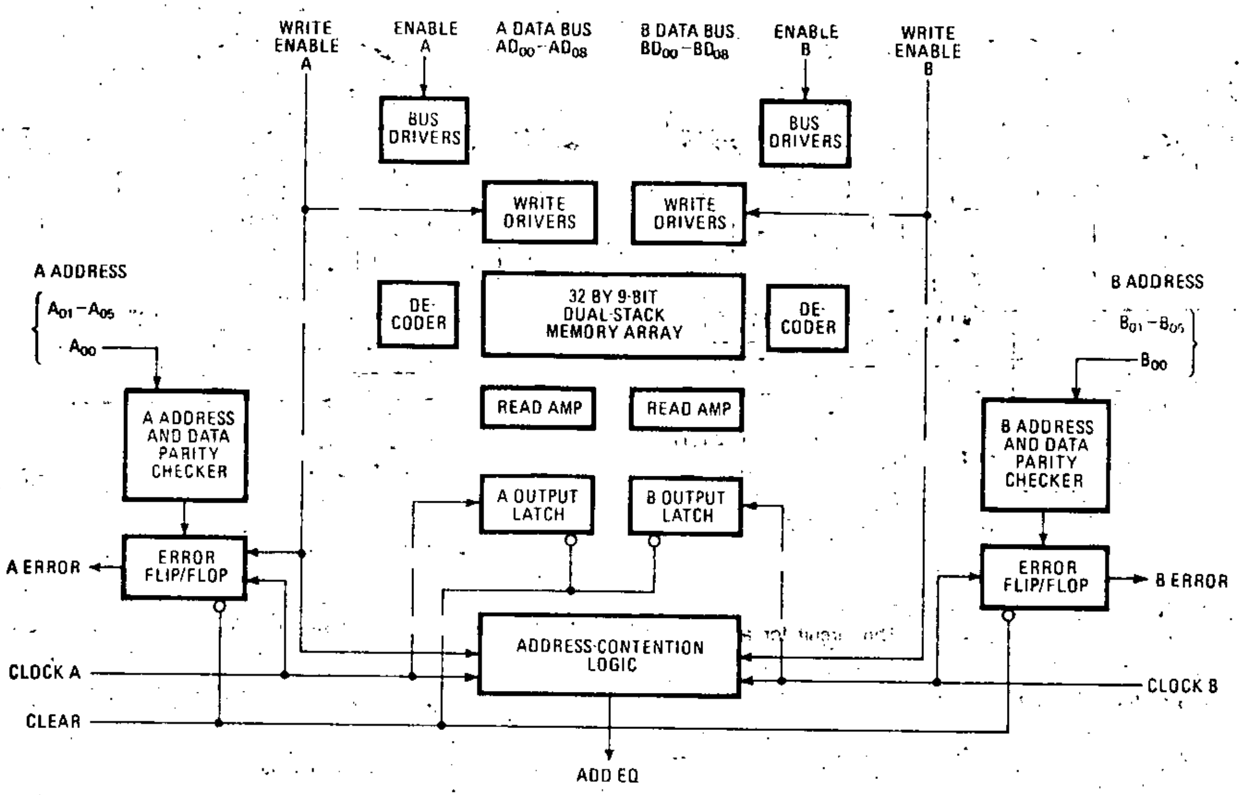
Branch control is a third field associated with microprogram addressing. Most programs have to make a large number of flow decisions either from ALU condition codes, such as zero-detection, overflow, and sign bit, or from external test points. Under command of the control field, those status signals are multiplexed into the 10801 branch inputs through control logic. Branch instructions that have been built into the 10801 include branch on condition, branch to subroutine, and branch and modify.

The 10800 performs arithmetic, logic, and shift operations on data within its ALU, register file, and/or memory interface. The bus structure of the processor in Fig. 5 also allows the ALU to generate microprogram addresses through the I bus, if required. Moreover, the ALU will operate in either binary or BCD data formats as controlled by the ALU microprogram field.

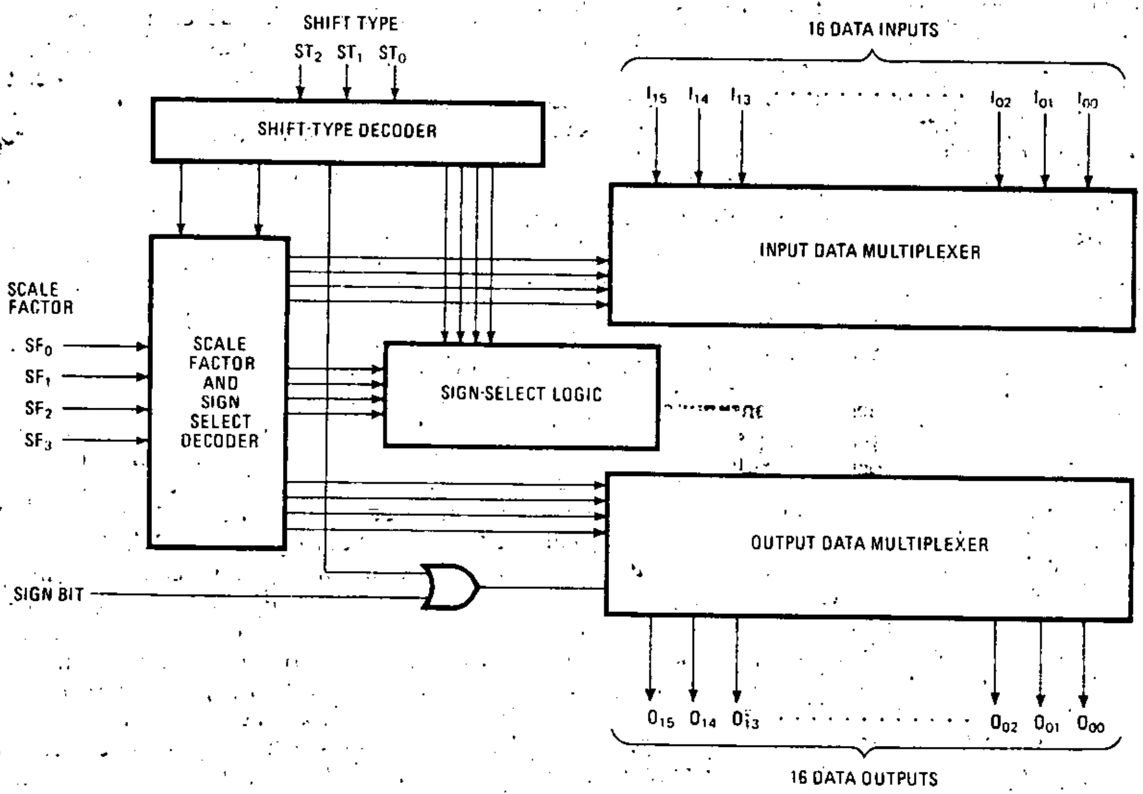
Unlike most other bipolar bit-slice families, the 10800 family leaves the register file as a separate block. The



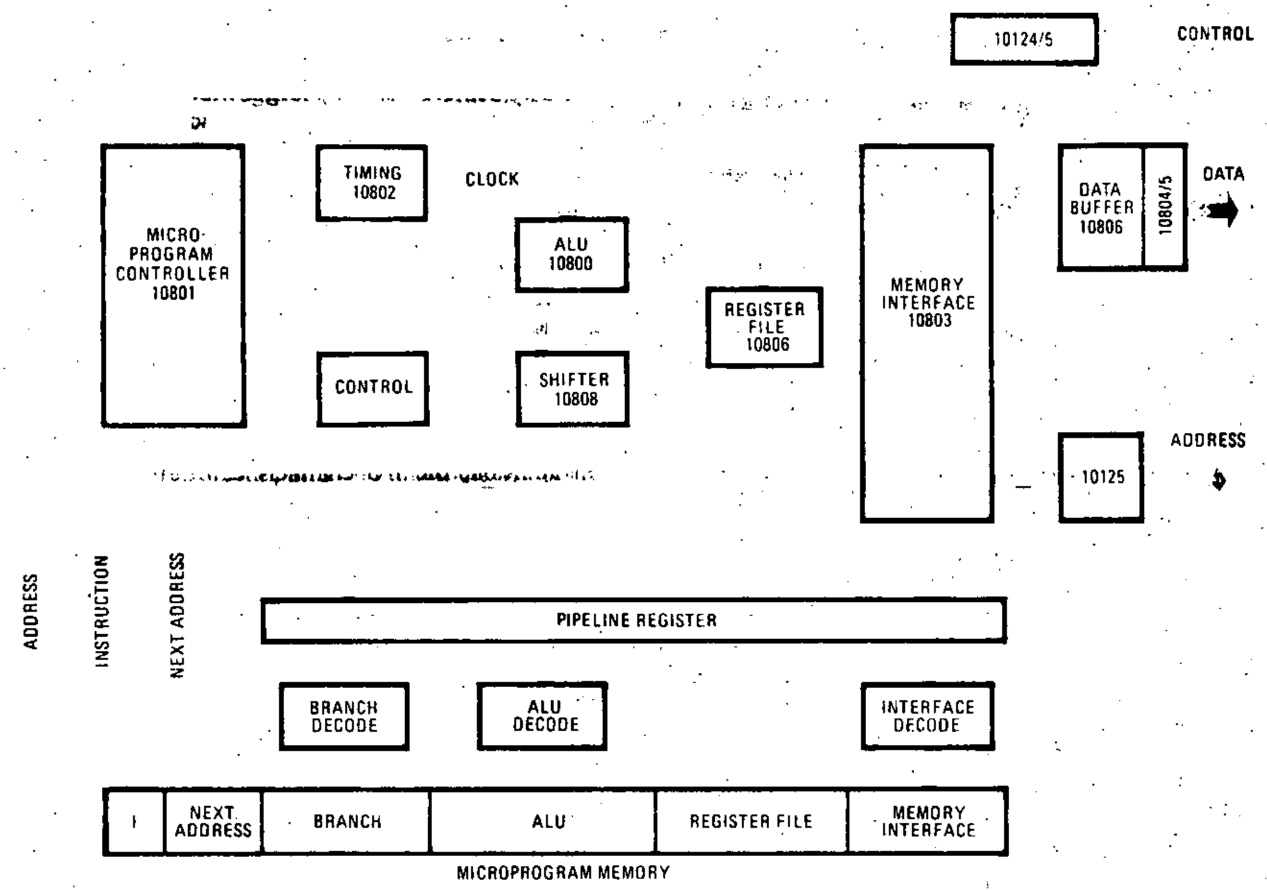
**4. Interface.** Interfacing the bit-slice processor to peripheral equipment through data and address buses, the MC10803 has separate select lines and provides for parallel data transfer and address generation within one microprogram cycle. It is expandable as necessary.



**8. Stacked.** The MC10806 dual-address stack has a 32-by-9-bit memory array with two independent read/write data ports. The dual-port structure combines with internal parity checking (which may be bypassed if unneeded) and solves many buffer-memory design problems.



**9. Shifter.** A programmable multiple-bit shifter, the MC10808 shifts 16 input bits from 0 to 15 places to the left, to the right, or in rotation. Cascadable like the other bit-slice parts, the circuit can be easily expanded to larger word sizes at no sacrifice in speed.



**10. Superprocessor.** Hooking the 10808 in series with the 10800 and supporting the 10806 with TTL translators creates an extremely powerful processor. The design also has a pipeline register to reduce cycle time, plus microprogram field decoding to cut word length.

[illegible]

| | | |
|---|---|---|
| ALS | – | ARITHMETIC SHIFT LEFT |
| ARS | – | ARITHMETIC SHIFT RIGHT |
| RLT | – | ROTATE LEFT |
| RRT | – | ROTATE RIGHT |
| SRC | – | SHIFT RIGHT - 2'S COMPLEMENT |
| SLC | – | SHIFT LEFT - 2'S COMPLEMENT |
| ODA | – | OUTPUT DISABLE |
| SBO | – | SIGN BIT AT ALL OUTPUTS |

operation between microprogram control and the rest of the processor. While the ALU, register file, and memory interface are executing one microinstruction, the 10801 is generating a new microprogram memory address. Pipelining is optional in a system built with the 10800 family—the 10801 interfaces directly to the microprogram in either case.

A final feature of the high-performance processor of Fig. 10 is the use of branch, ALU, and interface-decoding logic. Those blocks allow a relatively wide pipeline register feeding a large number of LSI control inputs to be driven from a narrow microprogram word. For example, a 6-bit microprogram field can select 1 of 64 ALU instructions. The ALU logic, however, may require 12 to 20 control inputs. The fanout is performed in decode logic commonly built with fast 10139 programmable read-only memories (PROMs). In addition to reducing microprogram size for cost reasons, decoding logic allows microprogram fields to be structured for easier programming. The decoding logic does not slow system performance since it is possible to go from clock to the 10801's address output through microprogrammed RAM or PROM and from decoding logic to pipeline register, all within the cycle time of one microinstruction.

## Future ECL LSI

Motorola has developed a MECL 10,000 Macrocell-array integrated circuit that is compatible with the 10800 parts and allows rapid development of high-speed LSI circuits with complexities of up to 750 equivalent gates. Although the Macrocell array will be used to develop specialized circuits for specific customers and systems, the advantages of this LSI concept will also lead to new standard products in the 10800 family.

The plans are thus to use new circuit developments to build on the 10800 family rather than around it. New functions under consideration include an advanced 8-bit arithmetic-and-logic and a very high-speed, expandable LSI array multiplier. These circuits, plus the Macrocell array concept, will be featured in an article on ECL LSI in the next issue. □

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# ELECTRONICS

DECEMBER

MARZO, 1979

# Two versions of 16-bit chip span microprocessor, minicomputer needs

Larger, 48-pin package addresses 8 megabytes of memory; regularity of instruction set makes programming easy

by Masatoshi Shima, *Zilog Inc., Cupertino, Calif.*

□ A microprocessor would gain ready acceptance if it could fit immediately into applications of current 8- and 16-bit microprocessors and at the same time have an advanced architecture that was expandable to ensure long product lifetime. The Z8000 from Zilog Inc. meets the first goal easily—and does so with 10 times the throughput of existing microprocessors. To meet the second goal, the Z8000 has departed from the traditional byte-oriented microprocessor design and moved toward the more regular architecture of minicomputers.

With many of the architectural features of minis and some pluses as well, the Z8000 is designed for minicomputer as well as microcomputer applications. To begin with, it handles seven data types, from bits to word strings, and offers eight selectable addressing modes. Its 81 distinct operation codes combine with the various data types and addressing modes to form a rich 414-instruction set more powerful than that of most minicomputers. Moreover, the set exhibits a high degree of regularity: more than 90% of the instructions can use any of five main addressing modes with 8-bit byte, 16-bit word, and 32-bit long-word data types.

Among its architectural resources are a large number of on-chip registers—24 16-bit registers in all—that dramatically reduce the number of memory references needed in programming. Sixteen of those registers are general-purpose, and all except one can be used as index registers without restrictions.

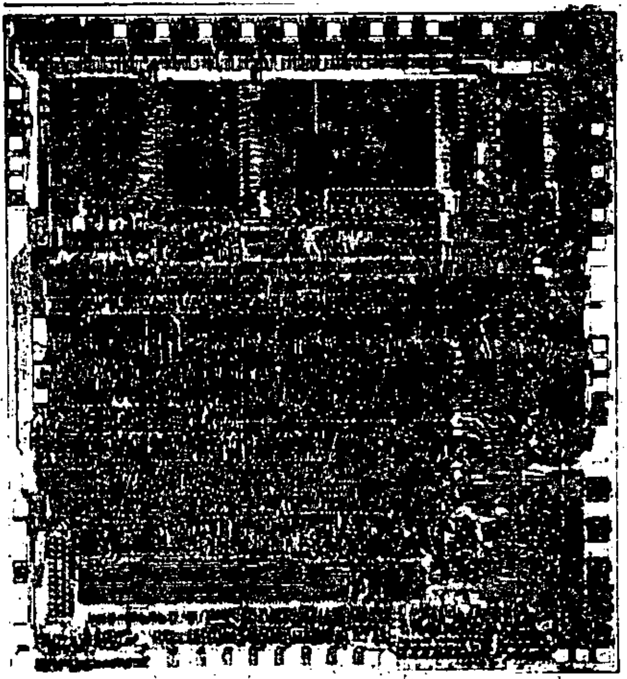
Also aiming at minicomputer applications is the Z8000's large direct-memory-addressing capability of 8 megabytes. Instead of treating it as linear space, however, the Z8000 organizes memory into a set of 128 segments of up to 65,536 bytes each. A segmented space is closer to the way the programmer uses memory—each procedure and data space, either local or global, resides in its own segment. To further facilitate use of all that space, a memory-management chip will work with the Z8000 in performing the dynamic relocation and memory protection needed in a large system.

## Two versions

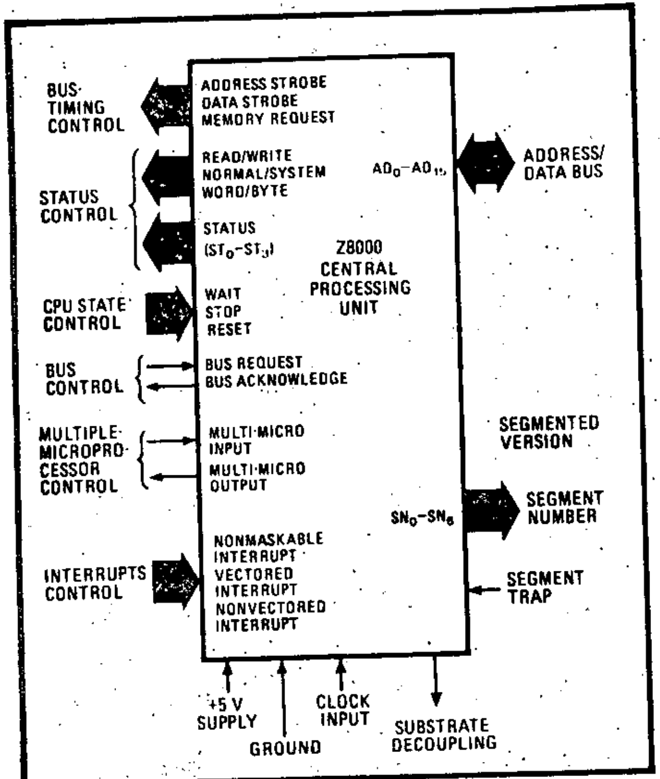
But because the Z8000 must satisfy existing microprocessor needs, two versions are offered. Besides the 48-pin memory-segmented version with 23 lines that addresses 8 megabytes, a 40-pin chip is offered with 16 lines to address 64 kilobytes—the equivalent of one segment. Expansion is guaranteed from the 40-pin to the 48-pin version: the segmented Z8000 can run any nonsegmented code in any one of its 128 segments using a load-program-status instruction.

Finally, the Z8000 boasts two operating modes, system and normal, that keep operating-system and applications programming separate, as in computer systems. Each mode has a separate stack, and the arrangement isolates global features like privileged instructions from normal programming.

An n-channel metal-oxide-semiconductor chip built with scaled-down depletion-load silicon-gate technology, the Z8000 squeezes about 17,500 transistors into an area of 238 by 256 mils. Its density—148 gates per square millimeter—surpasses that of previous microprocessors (see also "Genealogy of the Z8000," p. 83). The chip uses a 5-volt supply and requires a single-phase 4-megahertz (250-nanosecond) clock for timing. As at



**Dense.** The 16-bit Z8000 central processing unit is built with scaled n-channel depletion-load silicon-gate technology. The chip, which crams about 17,500 transistors into a 238-by-256-mil or 39.3-millimeter-square area, has a density of about 148 gates/mm².



**1. Two versions.** The Z8000 fits microprocessor sockets with its 40-pin nonsegmented version, which has 16 lines for directly addressing 64 kilobytes of memory. The minicomputer-like 48-pin version adds 7 segment-address lines; it can address 8 megabytes.

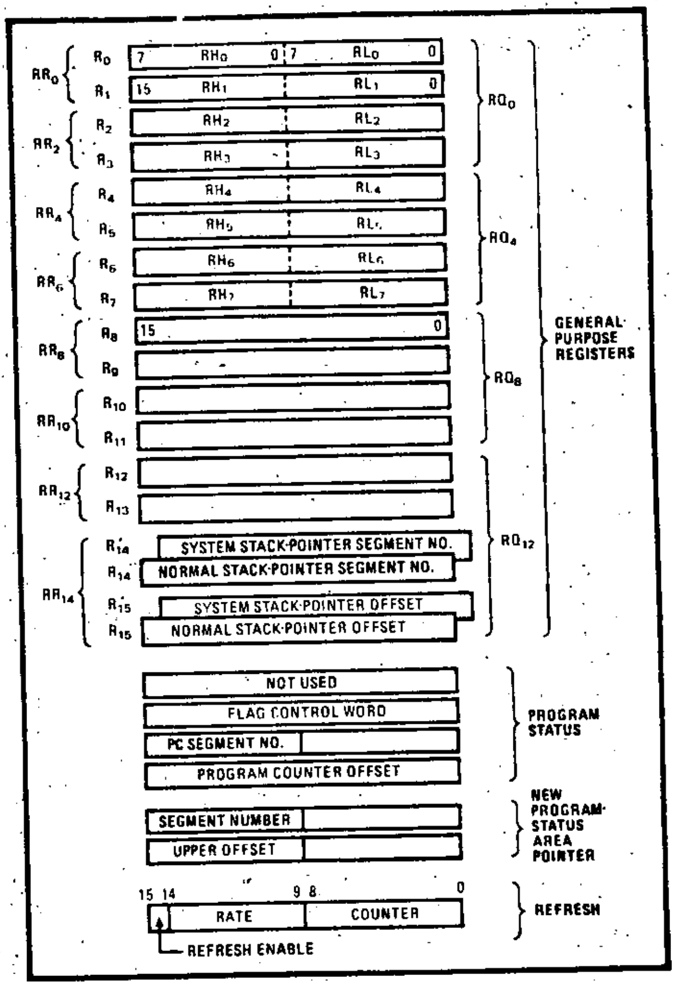
least three clock cycles in the central processing unit are required for one memory cycle, the Z8000 needs memory devices with a cycle time of 750 ns and an access time of 430 ns.

As shown in Fig. 1, the Z8000 has in addition to its address and data buses and clock and power-supply inputs six types of control buses: bus-timing, status, CPU-state, interrupt, bus, and multiple-microprocessor control. The three bus-timing control outputs coordinate the data flow over the chip's address/data lines. An address strobe signals that addresses are valid, and a data strobe times the window for valid data in and out of the CPU. The memory-request line is a timing signal that eases interfacing to dynamic memory.

## CPU status

The next bus provides information on the CPU's status. A read/write line gives early status of the forthcoming cycle, while a normal/system line indicates which of the two modes the CPU is in for the current cycle. A word/byte line indicates whether the CPU is accessing 16 bits of data or 8 bits. The four status-control lines form a 4-bit word that indicates several bus statuses, including memory-request, stack, first- and subsequent-word instruction fetch, interrupt acknowledgments, internal operation, and others.

The next of the control buses are three CPU-state inputs. The reset line initializes the CPU. A wait line signals the CPU that data transfer is not ready. The stop line halts internal CPU operation (although dynamic memory is still refreshed). The CPU can be stopped each time the first word of an instruction is fetched.



**2. Registers.** Sixteen 16-bit registers are organized into high and low bytes (RH and RL), 32-bit long-words (RR), and 64-bit quad-words (RQ). Four words, including the program counter, contain the program status and two more point to the new-program-status area.

A pair of lines governs the control of all the Z8000's buses. Driving the bus-request input low instructs the CPU to put all its address/data, bus-timing, and status-control lines into a high-impedance state so that other devices can use them. The CPU signals it has relinquished control with its bus-acknowledge output.

Another pair of lines is used with certain instructions to coordinate multiple-microprocessor systems. The multi-micro output line issues a request, while the input line recognizes outside requests. Thus any CPU in a multiple-microprocessor system can, for example, exclude all other asynchronous CPUs from being able to access a critical resource.

Finally, there are three interrupt inputs and, in the segmented version of the Z8000, a trap input. Interrupts are asynchronous events triggered typically by peripherals needing the CPU's attention, and traps are synchronous events resulting from the execution of specific instructions that occur each time the instruction is executed with the same set of data. The two are handled in a similar fashion by the Z8000.

The Z8000 is a register-oriented machine, placing little constraint on the use of its 16 general-purpose

## Genealogy of the Z8000

The changes in microprocessor architecture from the first-generation 8-bit devices to the fourth-generation Z8000 have been both swift and dramatic. Developments have been guided alternately by technology limitations and by the hardware and software demands of users.

Thus, the shortcomings of the first 8-bit microprocessor, the 8008 developed in 1971, were technological. Metal-oxide-semiconductor processing was relatively new and set limits to circuit complexity. Also, microprocessors were actually offshoots of calculator designs, being developed by semiconductor and not computer houses. So performance and features left much to be desired.

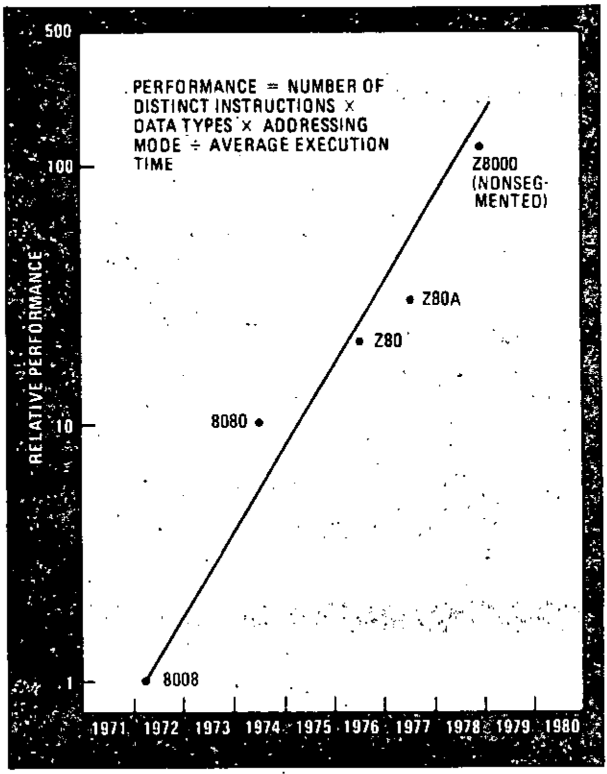
Advances in processing technology gave microprocessor designers a powerful tool with which to build the next generation of microprocessors—n-channel silicon-gate MOS that boosted circuit speeds by a factor of four over previous p-channel technology. Thus, the 8080, born in 1974, began the second generation of microprocessors.

By the time the third generation of microprocessors entered the design stage, users had become more sophisticated and were involved in high-level languages. Data-processing applications grew in popularity, and the disk-operating system was introduced. It was those software requirements that indicated the areas needing improvement, and the Z80 addressed the problem with software-oriented features. It added a large number of new instructions and a second register set, two index registers, and better interrupt handling. Still, because the Z80 maintains source-code compatibility with the 8080, many critical bottlenecks were inherited.

The Z80 marked the final exploitation of the original microprocessor structure and instruction format. Attempts to add capabilities would require two or three 8-bit instruction fetches—and exceedingly poor use of memory bandwidth and space. Moreover, the increasing popularity of high-level languages, plus a demand for much larger addressing space fueled by the plummeting costs of memory, outstripped the capabilities of an 8-bit microprocessor. The various trends toward large programs, complex distributed intelligent systems, and advanced memory management all pointed to a 16-bit architecture.

But it was Zilog Inc.'s conclusion that a chip with minicomputer performance could not last a decade without 32-bit operations and memory segmentation. Thus it chose the more advanced approach of the Z8000.

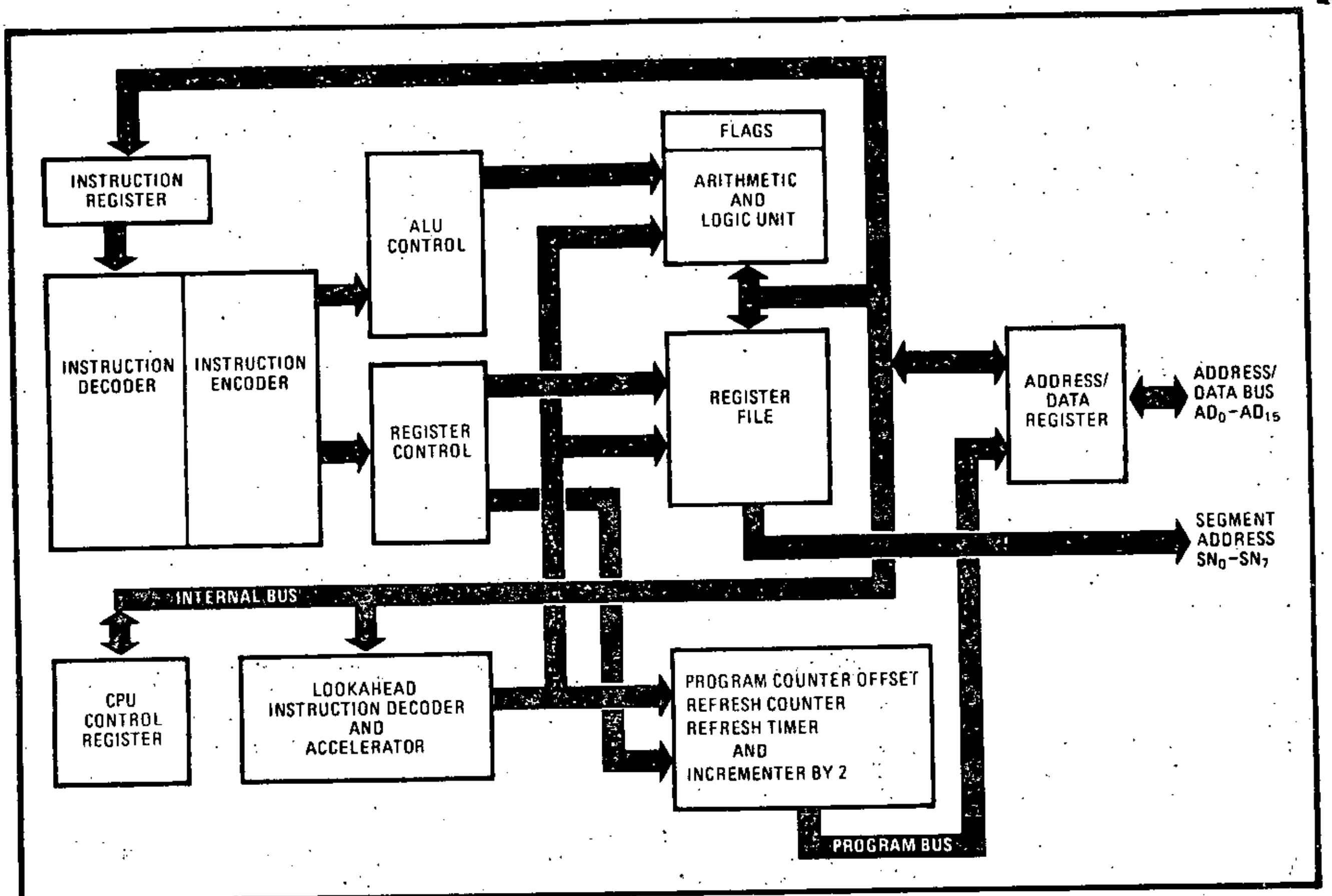
The table compares microprocessors. The companion graph indicates relative performance; though the equation provides no absolutes, it serves as an indicator for both hardware and software since it takes instructions, addressing, data types and speed into account.



COMPARISON OF MICROPROCESSOR CHARACTERISTICS

| | 8080 | Z80 | Z80A | Z8000 |
|---|---|---|---|---|
| Date of initial production | 1974 | 1976 | 1977 | 1978 |
| Power consumption (W) | 1.2 | 1.0 | | 1.5 |
| Number of transistors | 4,800 | 8,200 | | 17,500 |
| Number of gates | 1,600 | 2,733 | | 5,833 |
| Chip size (mm$^2$) | 22.3 | 27.1 | 22.4 | 39.3 |
| Density (gates/mm$^2$) | 72 | 101 | 122 | 148 |
| Number of distinct instructions* | 34 | 52 | | 81 |
| Combination of number of distinct instructions and data types* | 39 | 60 | | 149 |
| Combination of number of distinct instructions, data types and addressing modes* | 65 | 128 | | 414 |

*The numbers represent a conservative counting method. The user sees much larger number of instructions in assembly-language notation.



**3. Architecture.** The Z8000 boosts throughput with a look-ahead instruction decoder and accelerator on its internal bus. Thanks in part to the regularity of the instruction set, an instruction actually begins execution while it is entering the instruction register.

registers. Indeed, with but one exception (the stack pointer), no registers are ever implied in an instruction and none whatever have special restrictions. Bottlenecks found in early microprocessor designs, like dedicated accumulators, are thus avoided, so that programming is efficient and straightforward. All 16 of the 16-bit registers ($R_0$–$R_{15}$) can be used as accumulators. All except $R_0$ can be used as index registers, base registers, and as memory pointers for indirect addressing.

### A flexible register architecture.

As shown in Fig. 2, the flexibility of the registers is afforded by a unique arrangement of overlaps and pairs. The 16 8-bit registers ($RH_0$–$RH_7$ and $RL_0$–$RL_7$), all of which may be used as accumulators, are overlapped with the first eight 16-bit registers ($R_0$–$R_7$). The eight 32-bit long-word registers ($RR_0$–$RR_{14}$) are register pairs, and the four 64-bit quad-word registers ($RQ_0$–$RQ_{12}$), which are used by a few instructions such as multiply, divide, and extend sign, are register quadruples.

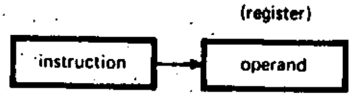
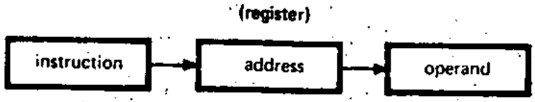
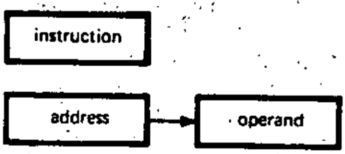
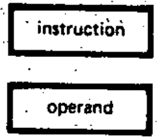
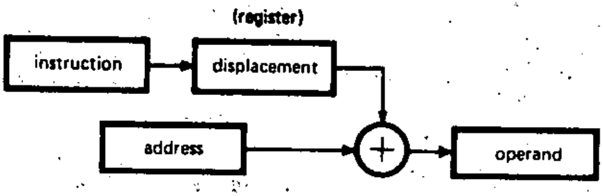
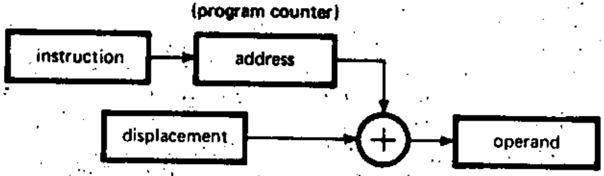
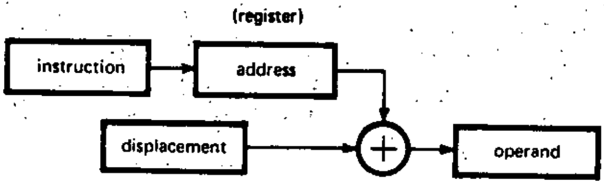
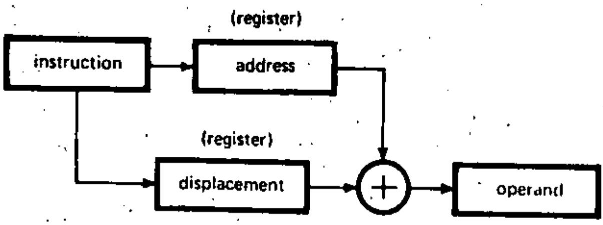
In the nonsegmented version of the chip, the last 16-bit general-purpose register, $R_{15}$, is the stack pointer. In the segmented version, the last two registers, $R_{14}$ and $R_{15}$ (or long-word register $RR_{14}$), are needed to hold the stack pointer, with $R_{14}$ storing the segment number while $R_{15}$ contains the offset. The only instructions that use the stack pointer exclusively are call, call relative, return, and return from interrupt; the push and pop instructions can use any register as a stack pointer. However, all instructions can manipulate the stack pointer, since it is in the general-purpose register group.

The two running modes of the Z8000 each have a copy of the stack pointer—one for the system mode and another for the normal mode—as implied by the primed registers $R_{14}'$ and $R_{15}'$ in Fig. 2. Although the stacks are separated, the normal stack registers can be accessed in the system mode by using the load-control-word instruction. Having two sets of stack pointers facilitates task-switching when interrupts or traps occur. The normal stack is always kept clear of system information, since the information saved on the occurrence of interrupts or traps is always pushed on the system stack before the new program status is loaded.

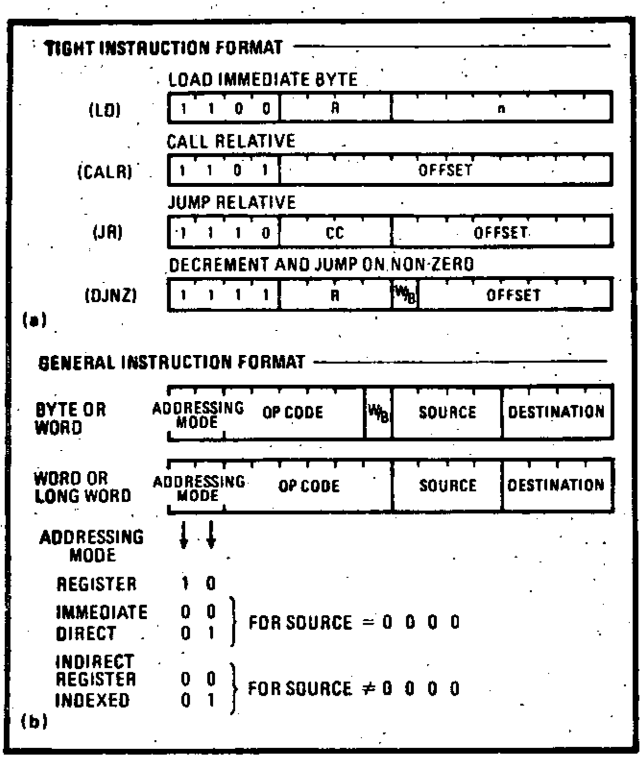
In addition to the general-purpose registers, there are the program-status registers, which contain the flags, control bits, and program counter. In the 40-pin nonsegmented version of the Z8000, the program status is held in two 16-bit registers: the first is the flag and control word, the second is the program counter. In the segmented version, program status is a full four words: the flag and control word, a two-word program counter, and a word reserved for future use.

Another register holds the pointer for the new-program-status area. It comprises two words in the segmented version and one word in the nonsegmented version. Lastly, a refresh register contains a 9-bit counter for automatic refresh of dynamic memories.

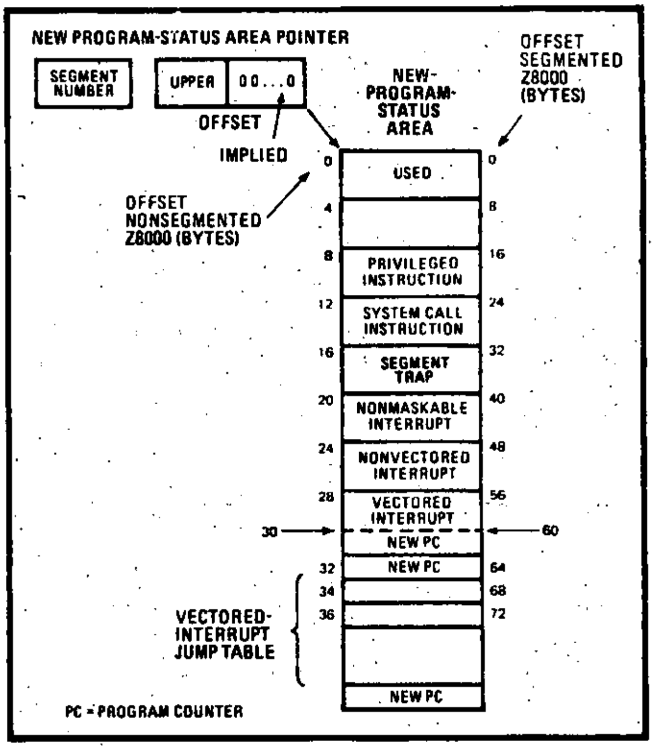
The Z8000 executes instructions by stepping through

| Z8000 ADDRESSING MODES | | |
|---|---|---|
| Mode | Diagram | Operand value |
| Register |  | the content of the register |
| Indirect register |  | the content of the location whose address is in the register |
| Direct address |  | the content of the location whose address is in the instruction |
| Immediate |  | in the instruction |
| Index |   (register)<br>instruction<br>displacement<br>address<br>+<br>operand | the content of the location whose address is the address in the instruction, offset by the content of the working register. |
| Relative address |   (program counter)<br>instruction<br>address<br>displacement<br>+<br>operand | the content of the location whose address is the content of the program counter, offset by the displacement in the instruction |
| Base address |   (register)<br>instruction<br>address<br>displacement<br>+<br>operand | the content of the location whose address is the address in the register, offset by the displacement in the instruction |
| Base index |   (register)<br>instruction<br>address<br>(register)<br>displacement<br>+<br>operand | the content of the location whose address is the address in the register, offset by the displacement in the register . |

**Many modes.** Over 90% of the Z8000's instructions work with any of five main addressing modes, proof of the chip's software regularity. A load-addressing instruction that accepts all eight modes accommodates any other operand-addressing scheme desired.



**4. Instruction formats.** Tight format (a) uses only one 16-bit word, is executed quickly, and saves memory. General format (b), used for bytes, words, or long words, specifies the addressing mode using the 2 address-mode bits and the source register number.



**5. Program status.** The new-program-status area pointer is two words long; the 7 most significant bits in the second word specify the beginning of an area in memory from which the new program status is fetched in response to interrupts and traps.

a set of the following basic machine cycles: memory read or write, input/output-device read or write, and internal data execution. Since the memory cycle uses three clock cycles to fetch the instruction or data from memory or to write data into memory, each machine cycle also uses a minimum of three clock cycles, though for complex operations it can extend to as many as eight.

## The matter of timing

Ideally, for optimum throughput, all instruction time should be memory-cycling time; no clock cycles should be wasted on other phases of the instruction cycle. Simulations of a wide variety of benchmark programs have shown that, on the average, the effective memory cycle time (also called the bus-utilization time or bus efficiency) of the Z8000 is 80% to 85% of the instruction time and up to 90% if jump instructions are excluded. This efficiency is a significant improvement over the 65% to 70% of the 8-bit Z80 microprocessor.

One reason for the high efficiency of the Z8000 is its look-ahead instruction decoder and accelerator, shown in the architectural block diagram of Fig. 3. Since the look-ahead is tied to the internal bus, and since the instruction set is very regular, an instruction can actually begin execution while it is still being stored in the instruction register. The look-ahead makes for a significant improvement in throughput, for example, in the case of direct and indexed memory addressing (the most frequently used addressing modes after register address), in which the Z8000 does not require any additional clock cycles to decode the instruction in deciding whether it is short or long offset. The load-register-to-register instruction has been optimized to require only the three clock cycles of its memory access. In most instructions, in fact, the data-manipulation time is fully overlapped with the fetching of the first word of the next instruction.

Throughout the design of the Z8000, meticulous attention was paid to accelerating and optimizing each instruction in proportion to its statistical importance. Some instructions and data references are aligned in a single word to speed execution, simplify logic, and get a larger range when the relative addressing mode is used.

To further increase execution speed, as well as to reduce memory usage, the most frequently used instructions in the Z8000 have been coded as one word. Among these are jump relative, decrement and jump on non-zero, load immediate byte, load immediate word, and call relative. Moreover, the sophisticated, interruptible, preprogrammed block and string instructions can execute memory-to-memory data manipulations as fast as 888,000 bytes per second.

## Extra instructions

A number of powerful instructions not found on previous microprocessors were added to the Z8000 repertoire. There are those that handle the new data types—instructions like multiply and divide that manipulate 32-bit long-words—and other instructions that load and store multiple words. And there are instructions that increment and decrement the contents of any register or memory location by any number from 1 to 16. Finally, multiple addressing modes for the push, pop, load, and store instructions enhance performance.

An important part of microprocessor design is the

## The Z8000 family

Even a minimum system based on a Z8000 microprocessor executes instructions fast and is easy to program. Soon to be available is a family of associated chips that will extend these advantages to complex Z8000-based computer systems and networks.

The members of the family include:

- The Z-MMU memory-management unit that takes care of memory segmentation and protection and address translation.
- The Z-UPC universal peripheral controller, a Z8 single-chip microcomputer used as a general-purpose programmable peripheral device with the Z8000.
- The Z-CIO counter and parallel-I/O chip, with three programmable 16-bit counters, two 8-bit bidirectional I/O ports, and a 4-bit I/O port.
- The Z-SIO serial-I/O circuit, which has two full-duplex channels and is capable of handling asynchronous and bisynchronous protocols at data rates of up to 880 kilobytes per second.
- The Z-MBU microprocessor buffer unit, a 128-by-8-bit first-in, first-out buffer that can be cascaded to any depth to connect asynchronous parallel processors to the Z8000 when multiprocessing.
- The Z-FIFO first-in, first-out buffer memory, also 128 by 8 bits, for expanding the Z-MBU depth or interfacing I/O ports to user equipment.
- The Z-bus RAMs, a pair of random-access memory chips—a 2,048-by-8-bit fully static RAM and a self-refreshing 4,096-by-8-bit pseudo-static RAM—for small local storage.

instruction format, since logic complexity (and hence chip size) depends heavily upon its complexity. Designing into the instruction set total software regularity (where all instructions can use all data types and addressing modes) is ideal, and that goal is one towards which the Z8000 has striven.

Of the eight selectable addressing modes (see table), the five main modes—register, indirect register, immediate, direct address, and indexed address—can be used with nearly all instructions, excepting a few such as rotate and shift instructions. The three other addressing modes—relative address, base address, and base indexed address—have been added to all load and store instructions. To save memory space, the relative addressing mode applies additionally to jump, call, and decrement and jump on non-zero instructions. Some instructions have built-in autoincrementing and auto-decrementing addressing modes. Finally, a load-address instruction, which can use all of the eight addressing modes, supports even the most sophisticated operand-addressing schemes.

### Instruction formats

The formats for Z8000 instructions are shown in Fig. 4. The 2 most significant bits in the instruction word determine whether the tight instruction format (a) or the general instruction format (b) is used. Use of the tightly coded instruction—a single word—reduces instruction-memory usage and speeds execution.

As long as the 2 most significant bits are not both 1s, the general instruction format applies. Those 2 bits in conjunction with the source-register field in the instruction are sufficient for specifying any of the five main addressing modes. As shown in Fig. 4b, an all-zero source specification distinguishes immediate or direct addressing from indirect and indexed addressing, both of which require a source register. Source and destination-register fields in the instruction format are 4 bits wide for addressing the 16 general-purpose registers.

The Z8000 does not have memory-to-memory arithmetic instructions. However, it performs memory-to-memory transfers on a sophisticated set of preprogrammed block-transfer and string-manipulation instructions and offers store immediate, push immediate, and compare immediate instructions. That arrangement provides a more compact instruction format with more op codes available for additional instructions than would be possible with the general memory-to-memory addressing mode used in the Digital Equipment Corp. PDP-11 minicomputer, which has two sets of addressing modes and register fields.

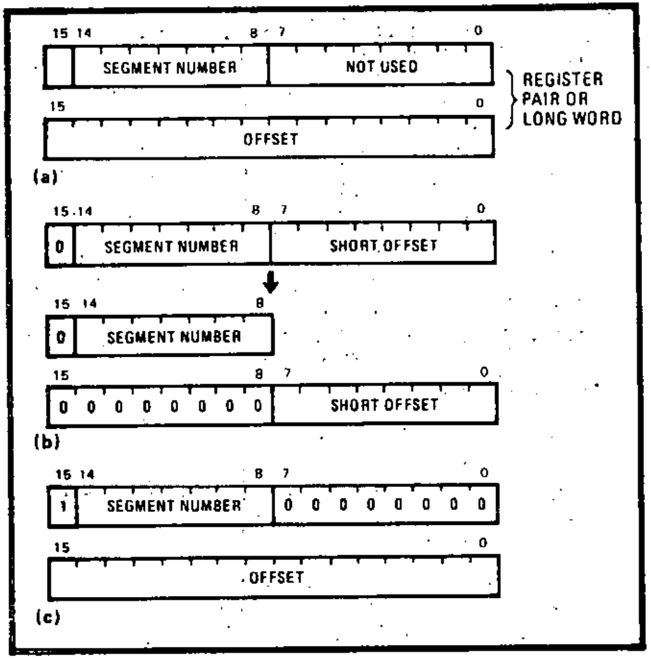
### Interrupts and traps

The Z8000's seven interrupts and traps, both internal and external, are arranged in priority. The three interrupts are all external inputs: nonmaskable interrupt, vectored interrupt, and nonvectored interrupt. The vectored and nonvectored interrupts are maskable. Of the four traps, the only external one is the segment input, which is found in only the 48-pin segmented version of the chip. The remaining three traps occur when certain instructions limited to the system mode are called in the normal mode, or for the system-call instruction, or for an illegal instruction. The descending priority order of the traps and interrupts is: internal traps, nonmaskable interrupts, segment trap, and vectored and nonvectored interrupt.

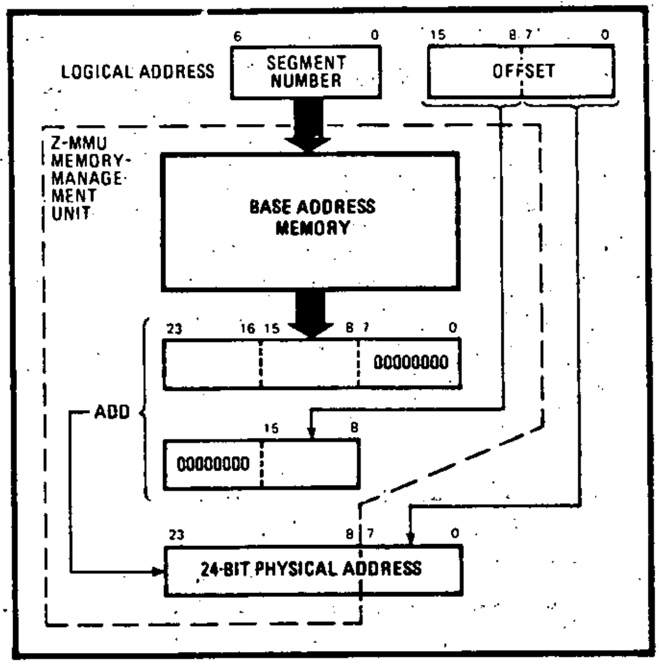
When an interrupt or trap occurs, the program status, which is contained in two 16-bit words in the nonsegmented version and three words in the segmented version, is pushed onto the system stack followed by an additional word. This extra word typically indicates the reason for the occurrence.

In the case of an internal trap, the reason word is the first word of the trapped instruction. In the case of the segment trap and for all interrupts, the reason is the vector on the data bus that is read by the CPU during the interrupt- or trap-acknowledge machine cycle.

The previous program status thus having been pushed on the system stack, a new program status is fetched from the new-program-status table (Fig. 5) that is specified by the new-program-status area pointer. As in Fig. 2, that pointer is the most significant byte in the new-program-status area pointer register. In the case of the segmented version of the Z8000, the pointer is two words in all—the segment number is specified by the 7 most significant bits of its second word. After the interrupt or the trap has terminated, a reset sequence is entered. A new program status is then fetched from a fixed location



**6. Address representation.** Segmented addresses appear as a long word (32 bits) when represented in a register or in memory (a). In an instruction, however, an address can be a single word (b) or a long word (c) if it is within the first 256 locations of a segment.



**7. Memory management.** The Z-MMU memory-management chip carries out the memory relocation from logical to physical address by adding the 16-bit offset value to a 24-bit base address associated with each segment. Two Z-MMUs handle all 128 segments.

in memory at the beginning of segment 0.

Facilitating the separation of operating-system programming from applications programming are the system and normal operating modes of the Z8000. The distinction is made by privileged instructions, which can only be executed in the system mode and are trapped when encountered in the instruction flow of normal-mode operation. Those instructions include all input/output instructions, halt, enable/disable interrupt, load control word, store control word, load new program status, return from interrupt, and all multiple-microprocessor instructions.

High-level languages, sophisticated operating systems, large programs and data bases, and decreasing memory prices are all accelerating the trend towards larger memory space in microcomputer systems. But even when it is available, questions are raised: how is it best accessed by a programmer? and what memory-management mechanism best allows the system to manage its memory on the user's behalf? In answer, the Z8000 proposes a segmented addressing scheme.

The segment number is an unsigned 7-bit integer ranging from 0 to 127; the offset is an unsigned 16-bit integer ranging from 0 to 65,535.

When represented in a register, a segmented address is always a register pair or long word (Fig. 6a). The two words may be manipulated separately or together by any of the word and long-word register operations. All segmented addresses exist in memory as a long word.

A segmented address in an instruction, however, has two different forms: either with a long offset (Fig. 6c), in which the address occupies two words, or with a short offset that is one word. The short offset, which, as shown in Fig. 6b, implies that the most significant 8 bits of the offset are all zero, can be used whenever the address is within the first 256 locations of a segment. That representation permits very dense encoding of addresses and is convenient not only for indexed addressing, but for direct addressing when short data segments are used or when subroutines start at the beginning of a segment.

## Memory-management chip

Those addresses manipulated by the programmer, used by the instructions, and appearing at the output of the Z8000 are called logical addresses. Transforming the logical addresses, which comprise the segment and offset concatenation, into a 24-bit physical address is the job of the Z-MMU memory-management unit (see "The Z8000 family," p. 87).

That transformation of logical address into a physical address, called relocation, is performed by this chip as shown in Fig. 7. A 24-bit origin or base is logically associated with each segment. To form the 24-bit physical address, the Z-MMU adds the 16-bit offset to the base for the given segment. (In operation, the Z8000 sends out the segment number half a clock period ahead of the 16-bit offset address to compensate for the time the unit needs to do this.) Thus the Z8000 can directly address half of a 16-megabyte physical memory space.

In addition to relocation, the Z-MMU provides segment management and protection from undesired writeover. Each such unit stores 64 segment entries that consist of the segment base address and its attributes, size, and status. Segments can vary in size from 256 bytes to 64 kilobytes in increments of 256 bytes.

Using a pair of these units with the Z8000 accommodates all of the 128 segment numbers. Moreover, several Z-MMUs can be used together to accommodate several translation tables, although only a single pair may be enabled at any one time. □

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# COMPUTER

MARZO, 1979

# Architecture of a New Microprocessor

Bernard L. Peuto
Zilog, Inc.

*Increased capabilities, architectural compatibility, and clearly defined interfaces were the chief architectural goals of Zilog's new Z8000 microprocessor family. Here is an account of how those goals were met for two members of that family—the Z8000 CPU and the MMU.*

The Z8000 family is a new set of microprocessor components (CPU, CPU support chips, peripherals, and memories) which supports the Z8000 architecture. The account of how architectural goals were selected and achieved for two key members of this family—the Z8000 CPU and the memory management unit—illustrates how much of a challenge microprocessor architecture represents to the semiconductor industry. MOS technology shows enormous potential, but it is still difficult to use because of limitations on pin count, power dissipation, speed, and complexity.[1]

Since this discussion is restricted to technical issues, we will not allude to the many additional factors (marketing considerations, human considerations, self-imposed restrictions, etc.) which make architecture such a fascinating and difficult discipline. Furthermore, no attempt has been made to exhaustively describe the Z8000 architecture and components. Interested readers should consult the specific manuals for a more complete description.[2,3]

## The goals of the Z8000 architecture: increased capabilities, architectural compatibility, increased clarity

The primary reason for introducing a new system architecture is to significantly improve the control and processing capabilities of microprocessors while maintaining their price/performance advantages. Technical advances have permitted the implementation of substantially increased processor power, but the most significant motivation for a new component family is generality. Only through such a family could we provide for architecturally compatible growth over a wide range of processing power requirements.

Our approach was a staged system architecture which attempts to provide new components, enhanced features, and new functions, while protecting the user's investment in hardware and software. The Z8000 family supports a single unified architecture for all small, medium, and high-end user applications which are implemented using a mix of components within the same family.

The goals of the Z8000 architecture can be grouped into three categories: increased capabilities, architectural compatibility over a wide range of processing powers, and increased clarity. In all these cases the resulting architectural features apply either to the basic architecture (that seen by an applications programmer) or to system architecture (that seen by a system designer or an operating system programmer).

**Increased capabilities.** All existing 8-bit microprocessors and many 16-bit minicomputers suffer from having a small address space. So, one of our goals was to provide access to a large address space (8M bytes). A second goal was to provide more resources in terms of registers (16 general-purpose 16-bit registers), in terms of data types (from bits to 32 bits), and in terms of additional instructions compared to existing microprocessors (multiply and divide, multiple register saving instructions, specialized instructions for compiler support etc.).

To facilitate complex applications it was important to support multiprogramming with good hardware support of task switching, interrupts, traps, and two execution modes. Operating systems also required a good hardware protection system.

Finally, we wanted to increase overall system performance. This resulted in the choice of an implementation using a 16-bit-wide data path to memory.

0018-9162/79/0020-0010$00.75 © 1979 IEEE

**Architectural compatibility.** One of the important lessons learned from previous computer system designs is that the design of a new family architecture is a rare occurrence. One way to apply this lesson is to design a unified architecture compatible over a wide range of processing powers. If we anticipate user growth from small to large systems within a family architecture, then such an approach can significantly increase its life.

The two versions of the Z8000 (a 40-pin unsegmented and a 48-pin segmented version) are designed to achieve this goal, but many other features contribute indirectly to the family compatibility. For small aplications an unsegmented Z8000 with one or more 64K-byte address spaces can be used. For medium applications, a segmented Z8000 and one memory management unit allows direct access to 4M bytes of address space. For large applications a segmented Z8000 and multiple pairs of MMUs allow the use of several 8M-byte address spaces.

Since the segmented Z8000 can run in an unsegmented mode, both systems are compatible. Finally, to achieve even larger processing power through hardware replication, the architecture provides basic mechanisms for both multiprocessing and distributed processing.

**Clarity.** Clarity in an architecture is a measure of how well key interfaces are defined and specified. This is an elusive but important goal in a family where new and unforeseen components will be added during the life of its architecture.

**We felt bus protocols were so important that we developed an independent specification for the Z-bus along with the individual device manuals.**

Clarity in terms of the basic architecture means regularity and extendability of the instruction set, as well as the general and simple handling of the operating system interfaces. Clarity in terms of the system architecture means a well-defined method of communication between the various components. The key link between these components is the Z-bus, which is a shared system bus. In the section on communication with other devices, we describe some of the various types of bus protocols. At Zilog we felt this was so important that we developed an independent specification for the Z-bus along with the individual device manuals.[4]

## Comparison with other system architectures

We are convinced that the differences between microprocessor system architecture and large computer system architecture are not sufficient to require a different design approach, although they certainly influence the details of design compromises. The last section of this paper deals with implementation tradeoffs and illustrates some particular compromises. (In a few places we mix implementation considerations with descriptions of architectural tradeoffs. Despite the importance of separating an architecture from its implementation, we found that this separation is often absent during the actual creation of a new architecture.)

Two differences between conventional computer systems and microprocessor systems have the greatest impact: price structure and component boundary differences. For high-end LSI systems, it makes sense to have one unified architecture, but unlike their computer family counterparts (IBM 360/370, PDP-11) different implementations cannot be justified on a price/performance basis. Speed and performance are mainly dependent on the state of technology, and therefore, for a given application, a user will waste the speed willingly since another slower implementation would cost the same. This does not exclude different versions of one implementation, which reflect only different test and production criteria such as package type, functional temperature range, and even speed range.

Most computer systems have both external and internal interfaces. External interfaces which define system boundaries are often standardized (e.g., the IBM channel interface or the DEC unibus). The internal interfaces of most mini or large computer systems are essentially hidden. In contrast, the component boundaries of a microprocessor-based system represent actual interfaces, and most users must be familiar with them as well as with external interfaces. Because the component interfaces are more visible and often must be more general, the microprocessor-oriented system bus emerges as a key standardization link to allow a wider mix of components and designs.

## The basic architecture

**Address space considerations.** It is advantageous to have more than one address space, with each address space as large as possible. In the Z8000, memory references and I/O references are viewed as references to different address spaces. The I/O space is discussed in the section below on communication with other devices. Memory references may be instructions or data and stack accesses, with each type of access possible in either system or normal modes. The Z8000 distinguishes between each of these reference possibilities by using different combinations of its status lines. Separating the various address spaces can be used to increase the total number of addressable bytes and to achieve protection. The size of each address space depends on the versions of the Z8000 used. The 40-pin package version allows each address space to be at most 64K bytes, the 48-pin package version allows each address space to be at most 8000K bytes.

The 40-pin version is intended for systems, often used as dedicated systems, where the program and data spaces are small. In this case, relocation is not usually important. Using the different address spaces, one has a simple way to address in practice up to 4 x 64K bytes (with a maximum of 6 x 64K bytes). Some simple protection is achieved by separating these spaces in hardware.

The 48-pin version with one or more MMUs is intended for the medium to large applications where relocation and better memory protection are important.[3] In these cases, status information can also be used to separate between address spaces by using multiple MMUs. But it is also essential to achieve the detailed memory protection required. (It is possible to use the 48-pin version without an MMU.) For these high-end applications, the address spaces are so large that one is unlikely to exhaust them. Experience with large computers shows that 8M bytes is probably adequate. The current implementation of the Z8000 uses 8M-byte address spaces, but the architecture provides for 31-bit address (2147M bytes).

In both versions, the Z8000 allows direct access to each address space. Direct access means that the addresses used in instructions or registers have as many bits as the address space size requires. In other schemes the effective address is a combination of a shorter field in the instruction and other extension bits often found in an implied register. Despite the shorter address fields, we believe this "indirect access" does not save bytes, because extra instructions must be used to load and save the implied registers, which are typically in short supply.

**Registers.** The Z8000 is primarily a memory-to-register architecture. This characteristic does not entirely exclude other organizations, and mechanisms exist in the Z8000 to support them. For example, memory-to-memory operations are supported for strings, whereas stack operations are supported for procedure and process changes. This choice provides upward compatibility with the Z80. A register architecture also results in good performance, since register accesses are made at a greater speed than memory accesses in the current implementation.

Experience with register-oriented machines seems to confirm that four general-purpose registers are not enough and that a "proper" number is between eight and 32.[5] The Z8000 supports bytes, words (16-bit), and long words (32-bit); and a few instructions even use quadruple-word (64-bit) data elements. If we choose 16, 16-bit registers allow eight 32-bit registers as well as four 64-bit registers (Figure 1). Since addresses are 32 bits, the necessity of at least eight 32-bit registers was obvious. The impact of the 4-bit register field on the instruction format depends also on the number of address modes and operands. Sixteen registers allowed a reasonable tradeoff, whereas 32 registers would have resulted in too few one-word instructions.

With one minor restriction any register can be used by any instruction as an accumulator, source operand, index, or memory pointer. This regularity of the structure is so important that it is worthwhile to sacrifice any possible encoding improvements in instruction formats which could result from dedicating registers to special functions. Encoding improvements based on instruction frequency, so that frequent instructions use one word, are more effective in saving space without having a negative effect on the architecture.

**Why not have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another.**

Most applications dedicate the available registers to specific functions. For example, most high-level languages require a stack pointer and a stack frame pointer. Then why not, one might argue, have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another. If the architecture supplies only general-purpose registers, the user is free to dedicate them to specific usages for his application without restrictions. This is important in the context of microprocessors where user applications are not well known and where high-level languages are still used infrequently.

For example, the Z8000 allows software stacks to be implemented with any register. There are also two hardware supported stacks, but the registers used are still general-purpose and can participate in any operation. There is no allocated stack frame pointer, since any register can be used by means of the proper combination of addressing modes. The savings realized by register specialization are unattractive when the given function can still be performed simply. The loss that would result from restricting the applications would be too great. In contrast, significant savings result from excluding R0 from use as an index or memory pointer. This exclusion allows one to distinguish between the indexed and direct addressing modes which use the same combination of the instruction address mode field. The price is small, since R0 still can be an acumulator or source register and 15 others accumulator, index, and/or memory pointers are available. In this case the restriction made sense.

Another decision to be made about registers is their size. Since the architecture handles multiple data types we must have multiple data register sizes, which can hold each data type. The solution of the problem is implemented in the architecture by pairing registers, two 1-byte registers make a word register, two word registers make a long word register, etc.

**Data types.** Users would like to have as many directly implemented data types as possible. A data type is supported when it has a hardware representa-

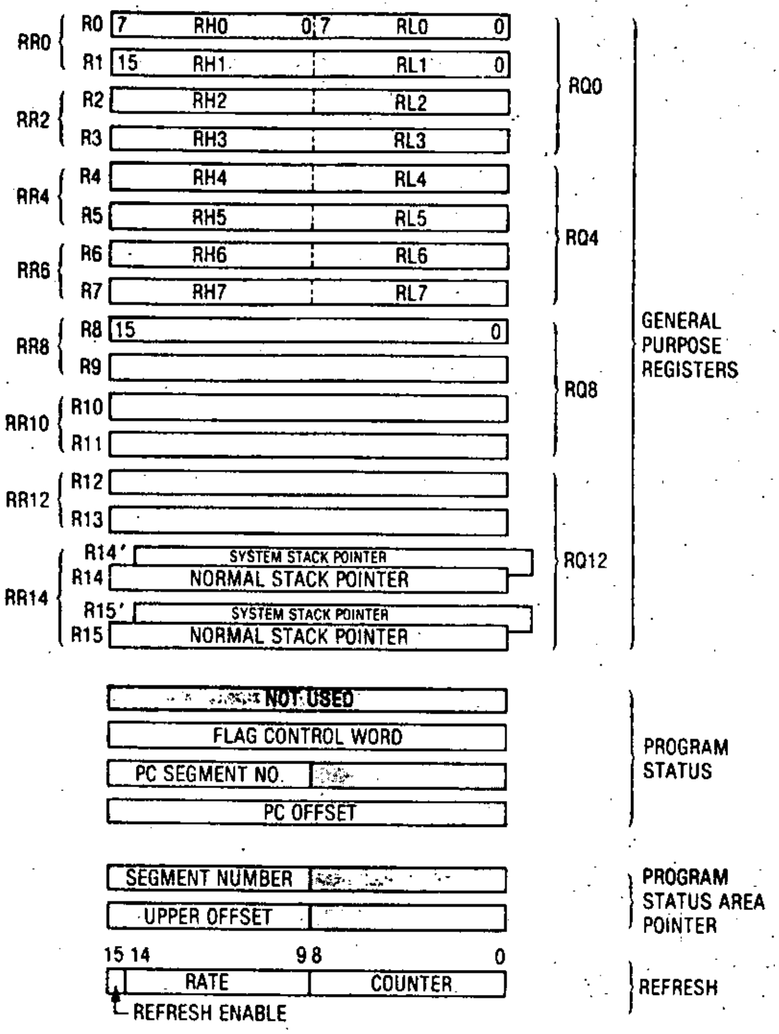


**Figure 1. CPU registers (segmented version).**

tion and instructions which directly apply to it. New data types can always be simulated in terms of basic data types, but hardware support provides faster and more convenient operations. At the same time, a proliferation of fully supported data types complicates the architecture and the implementations.

The Z8000 supports several primitive types in the architecture and provides expansion mechanisms. The basic data types are obviously the ones expected to be used most frequently. The extended data types are built using existing data types and manipulated using existing instructions.

The basic data type is the byte, which is also the basic addressable element. All other data types are referenced using their first byte address and their length in bytes. The architecture also supports the following data types: bytes (8 bits), words (16 bits), long words (32 bits), bytes, and word strings. In addition, bits are fully supported and addressed by number within a byte or word. BCD digits are supported and represented as two 4-bit digits in 1 byte. One consequence of this data type organization is that byte, word, and long-word registers are needed to support them. The Z8000 even provides quadruple register—another extension—used in long-word manipulation.

Other data types are supported by using one of the preceding data types; for example, addresses are manipulated as long words, and each element (segment number or offset) can be manipulated as a byte or a word. Instructions are one to five-word strings, the program status is four words, etc.

As the family grows, support for new data types will be added. The architecture will need to support them in its registers or in memory if they do not fit in registers (as strings are implemented today). But most important, the architecture will have to support the addition of new instructions to its repertoire.

**Instructions.** In designing an instruction format the architect must decide how to allocate a limited number of bits to the opcode field, address mode field, and other operand subfields. Instruction usage statistics are the best source of data to influence decisions about instruction set format.[1, 6, 7] Behind their usage lies a strong technical position: we do not

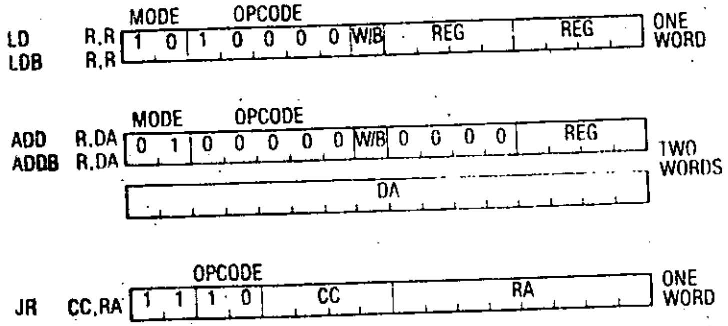


**Figure 2. Examples of instruction formats (nonsegmented version).**

believe that any one of the various instruction set structures—register oriented, memory oriented, stack oriented, symmetrical, or asymmetrical, etc.—are always better when used exclusively. Thus the task of the architect is to decide what his most important goals are, and for each of them adapt the best features of the various structures so that on the average, and for his set of goals, an optimum solution can be found. We do not believe that the optimum will be very sharp; it will be more like a range of applications for which the resulting composite structure works well. We decided to use a register structure for compatibility, multiple word instructions for speed, memory-to-memory instructions for strings, stack structure for process control and procedure support, "short" instruction for byte density improvement, etc.

*Instruction format consideration.* The Z8000 has over 110 distinct instruction types; several instruction formats are illustrated in Figure 2. The opcode field specifies the type of instruction (for example, ADD and LD). The mode field indicates the addressing modes (for example, Register (R), Direct Address (DA). The data element type (W/B) and register designator fields complete the basic instruction fields. Long word instructions use a different opcode value from their byte or word counterpart. Frequent instructions are encoded in a single word, and less frequent instructions which use more than two operands use two words. There are often additional fields for special elements such as immediate values or condition code descriptors (CC). Instructions can designate one, two, or three operands explicitly. The instruction TRANSLATE AND TEST is the only one with four operands and is also the only one with an implied register operand.

Several restraints can guide the proper choice of an instruction format. A large number of opcodes (used or reserved) is very important: having a given instruction implemented in hardware saves bytes and improves speed. But one usually needs to concentrate more on the completeness of the operations available on a particular data type rather than on adding more and more esoteric instructions which, if used frequently, will not significantly affect performance. Great care must be given to the problem of expanding the instruction set so, for example, new data types can be added.

*Addressing modes.* The Z8000 has eight addressing modes: *register* (R), *indirect register* (IR), *direct address* (DA), *indexed* (X), *immediate* (IM), *base address* (BA), *base indexed* (BX), and *relative address* (RA). Several other addressing modes are implied by specific instructions such as autoincrement or autodecrement.

Although a very large number of addressing modes is beneficial, usage statistics demonstrate that not all combinations of operands, address modes, and operators are meaningful.[6] The five basic addressing modes of R, IR, DA, X, and IM are the most frequently used and apply to most instructions with more than one address mode. For two-operand instructions, statistics show that most of the time the destination is a register. Other cases of addressing mode combinations and less basic addressing modes are associated with special instructions. Thus, the frequent combination of autodecrement for the destination operand with the five basic address modes for the source operand is provided by the PUSH instruction. The combination of autoincrement addressing modes for both source and destination operands is one of the block move instructions. In essence, the address mode field space has been traded for opcode field space. This allows more instructions and combinations while staying within a one-word format.

The price for this tradeoff is the infrequent occurrence of pairs or triples of instructions simulating a missing addressing mode. This situation occurs in most instruction sets in any case.

*Code density.* Because current microprocessors are restricted to primitive pipeline structures, their speed is largely dependent on the number of executed instruction words. Therefore, code density is not only important because of program size reduction but also because of speed improvement. One would like to encode in the smallest number of bits the most frequent instructions. The basic instruction size increment was chosen to be a word for reasons dealing with alignment, speed penalties, and hardware complexity. Thus the most frequent one and two-operand instructions take one word in their register or register-to-register forms. Less frequent instructions or instructions which use more than two operands use at least two words.

The Z8000 goes even further by selecting several special instructions as "short" instructions which take only one word, when normally they would take two words. These instructions, such as LOAD BYTE REGISTER IMMEDIATE and LOAD WORD REGISTER IMMEDIATE (for small immediate values), CALL RELATIVE, and JUMP RELATIVE, are so frequent statistically that they deserve such special treatment.

A one-word JUMP RELATIVE and DECREMENT AND JUMP ON NON-ZERO also have a very significant impact on speed. The short offset mechanism used by addresses (and described below) is also designed to allow one-word addresses. Compared to previous microprocessors, the largest reduction in size and increase in speed results from the Z8000's consistent and regular structure of the architecture and from its more powerful instruction set—which allows fewer instructions to accomplish a given task.

*High-level language support.* For microprocessor users, the transition from assembly language to high-level languages will allow greater freedom from architectural dependency and will improve ease of programming.[8] It is easy and tempting to adapt a computer architecture to execute a particular high-level language efficiently.[9] Most programming languages act as a filter and can be supported by a subset of available hardware with greater efficiency.[10] But efficiency for one particular high-level language is likely to lead to inefficiency for unrelated languages. The Z8000 will be used in a wide variety of applications, and we know that a large number of users will still be using assembly languages. Since the Z8000 is a general-purpose microprocessor, language support has been provided only through the inclusion of features designed to minimize typical compilation and code-generation problems. Among these is the regularity of the Z8000 addressing modes and data types. The addressing structure provided by segmentation should support procedures that result from structured programming. Access to parameters and local variables on the procedure stack is supported by index with short offset address mode as well as base address and base indexed address modes. In addition, address arithmetic is aided by the INCREMENT BY 1 TO 16 and DECREMENT BY 1 TO 16 instructions.

Testing of data, logical evaluation, initialization, and comparison of data are made possible by the instructions TEST, TEST CONDITION CODES, LOAD IMMEDIATE INTO MEMORY, and COMPARE IMMEDIATE WITH MEMORY. Compilers and assemblers manipulate character strings frequently, and the instructions TRANSLATE, TRANSLATE AND TEST, BLOCK COMPARE, and COMPARE STRING all result in dramatic speed improvements over software simulations of these important tasks, especially for certain types of languages. In addition, any register can be used as a stack pointer by the PUSH and POP instructions.

**Segmentation.** In order to provide for convenient code generation and data access, addresses must also be easy to manipulate. Architectures with direct access to memory typically use a linear address space, so that address arithmetic may be used on the entire address. In this case, addresses are manipulated as one of the data types of the same size. This removes the need to distinguish an address as a new data type. In contrast, the Z8000 has a non-linear address space. Addresses are made of two parts: a 7-bit segment number and a 16-bit offset. Only the offset participates in address arithmetic. The segment number is essentially a pointer to a part of the total address space, which can vary in size from 0 to 64K bytes. The hardware representation of a segmented address is a long word or a register pair (Figure 3), which allows the easy manipulation of each part of the address.

The segmented addresses are one of the key mechanisms used to support both large and small memory systems efficiently. The two versions of the Z8000 implementation, the 40-pin unsegmented and the 48-pin segmented, allow the maintenance of the architectural compatibility and ease the growth between these two application groups. The segmented address space guarantees that each 64K-byte address space of the 40-pin version becomes one of the segments of the 48-pin version. Each 40-pin version's 16-bit address becomes an offset within the segment, and a mode exists in the 48-pin package version in which 40-pin version code can be executed. Furthermore, compatibility with any current 8-bit microprocessor such as the Z80 is easy, and a new microcomputer such as the Z8 can address external data in a shared segment with the Z8000.

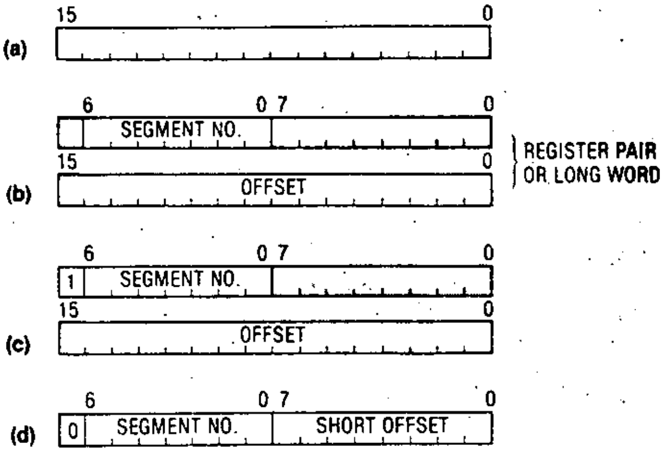


Figure 3. Hardware representation of segmented addresses. Any nonsegmented address is one word, whether it is in a register, memory, or an instruction (Figure 3a). Segmented addresses are always two words in a register or memory (Figure 3b); however, instructions can have one of two forms. The usual case (long offset) requires two words (Figure 3c); however, there is also a short offset form that uses only one word (Figure 3d).

The hardware performance of the Z8000 is also improved by address segmentation. Since a segment number does not participate in arithmetic, it can be put on the bus before the result of an address computation is available. This feature allows the use of MMUs with essentially no impact on memory access time by allowing it to function in parallel with the CPU. Indexing operations are also faster because only a 16-bit addition must be performed. Because of the distinction between the segment number and its offset, one can use shorter addresses without software constraints. Short addresses can use a short offset (fewer than 256 bytes) and thereby reduce program size (Figure 3).

Finally, it is very easy to associate with each of the 128 segments of the address space the protection and dynamic relocation features desirable for larger systems. Relocation allows a user to write his application using logical addresses independent of any physical addresses. Relocation is essential, for example, in a disk-based general data processing system with several users. Relocation is not essential for dedicated applications with code typically residing in

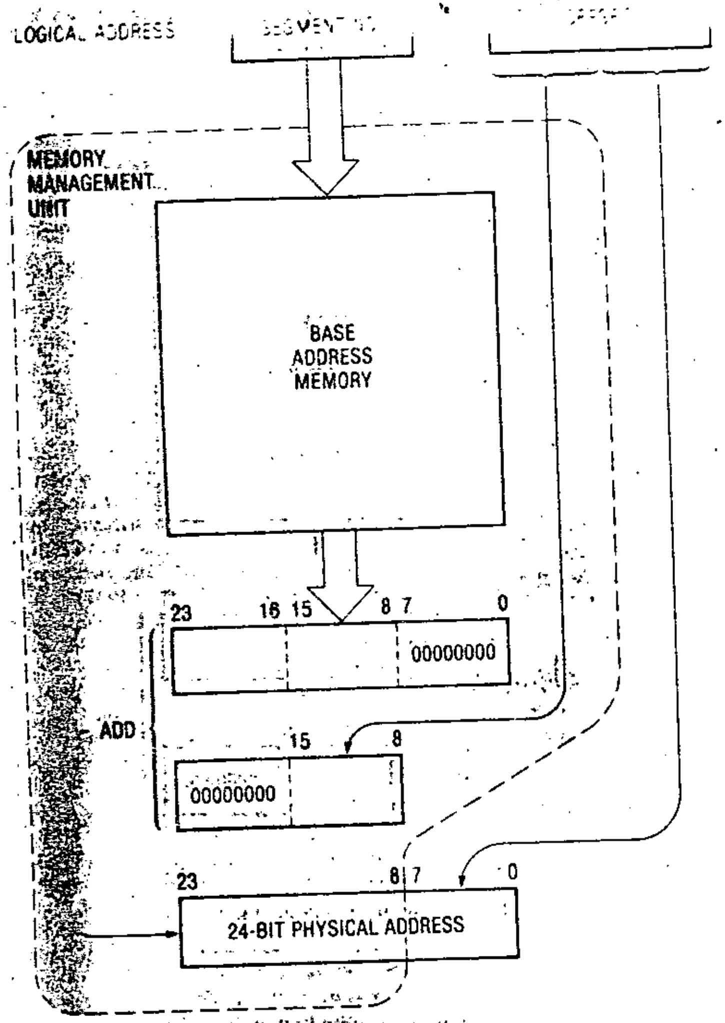


Figure 4. Logical to physical address translation.

ROM. Users whose total memory needs are small are also unlikely to need relocation.

In summary, the choice of a segmented address space has provided—at low cost and with few practical limitations—a powerful solution to the problem of user growth, relocation, and protection as well as virtual memory implementation. We believe that a linear address space could have achieved these results but at a considerably higher price.

## The system architecture

**Protection facilities.** The Z8000 protection facilities can be divided into system protection features and memory protection features. Experience with large computers has demonstrated the advantages of having at least two execution modes with different access rights to hardware facilities. The Z8000 provides the system and normal modes for this purpose. A simple protection system results from the presence of these two modes and their instructions is defined which deals with I/O, interrupts, traps, and mode changes. Programs in normal mode which attempt to execute a privileged instruction will cause a trap and a change to system mode. The switch from user to system mode can also be caused by the system call instruction. These mechanisms enforce protection and help in designing reliable and efficient operating systems with clean user interfaces. Several other traps are required to achieve a consistent system: segmentation trap, privileged instruction trap, and undefined instruction trap.

A desirable memory protection scheme is one for which protection information (read only, read write, execute only, system only, size of data or code, etc.) is easily associated with the data and code structures of a given application. It is also one for which a large number of different types of protection information can be verified.

The relocation and memory protection mechanisms described above are provided by an external device: the memory management unit.[3] To provide relocation and protection features directly on the Z8000 would have demanded too much simplification. The external MMU has the further advantage of providing for easier growth by the addition of components. The Z8000 40-pin package does not have to carry the burden of the unused advanced relocation and protection features, although some form of protection can be achieved by hardware separation of the different address spaces. With multiple MMUs, the 48-pin package user can control the relocation and protection complexity desired in his application.

**The memory management unit.** The MMU performs three functions: (1) address translation of logical address to physical address using dynamic relocation, (2) memory protection, and (3) segment management. The addresses manipulated by the programmer, used by the instructions, and output by the Z8000 are called logical addresses. The MMU uses these logical addresses, composed of a 7-bit segment number and 16-bit offset, and transforms them into a 24-bit physical address (Figure 4). A 24-bit origin or base is logically associated with each segment. To form a 24-bit physical address, the 16-bit offset is added to the base for the given segment. In effect, with the help of one memory management device, the Z8000 can address 8M bytes directly within a 16M-byte physical memory space. The reasons for the choice of a large physical address space include an expectation that large systems will want to use extra bits for complex resource management purposes.

Each segment is given a number of attributes when it is initially entered into the MMU. When a memory reference is made, the protection mechanism checks these attributes against the status information from the CPU. If a mismatch occurs, a trap is generated which interrupts the CPU. The CPU can then check the MMU status registers to determine the cause of the trap. Segment attributes include segment size and type (read only, system only, execute only, invalid DMA, invalid CPU, etc.) Other segment protection features include a write warning zone useful for stack operations.

When a memory protection violation is detected, a write inhibit line guarantees that memory will not be incorrectly changed. The invalid DMA and CPU bits indicate that the entry cannot be used by the DMA or CPU respectively, because either the segment number is illegal or the segment entry is not loaded. This fast feature, in conjunction with the segment history information (segment "changed" and segment "referenced" bits) and the segmentation trap mechanism, allows the implementation of a virtual segmented memory system.

The MMU comes in a 48-pin package (Figure 5). The chip inputs are the segment number, the upper 8 bits of the offset, and status information from the CPU. The outputs from the segment chip are the upper 16 bits of the 24-bit physical address and the segmentation trap line. Since the memory management device processes only the upper 8 bits of the offset, the lower 8 bits go directly to memory. This is equivalent to having zeros in the 8 lower bits of the 24-bit origin. Thus, the memory management device only needs to store the upper 16 bits of each base address. Segment limit protection is done in the memory management device, and thus segments can be protected in increments of 256 bytes.

Each MMU stores 64 segment entries that consist of the segment base address, its attributes, size, and status. A pair of MMUs support the 128 segments available in an address space. Additional MMUs can be used to accommodate multiple translation tables. Using the status information provided with each reference, pairs of MMUs can be enabled dynamically.

The memory management device functions constantly while memory references are made, but its translation and protection tables are loaded and unloaded as an I/O peripheral. To achieve this, the memory management device has chip select, address strobe, data strobe, and read/write lines. The Z8000 special byte I/O instructions that use the upper byte of the data bus can load or unload the memory management device.

**Mode switching: interrupt and trap handling.** From small users in dedicated process control applications to large users in general-purpose data processing applications, asynchronous events such as interrupts and synchronous events like traps must be handled. When these events occur, the state of any currently executing program must be saved during what is generally called a task switch or process switch. The users benefit from the availability of many interrupts and traps. They also benefit from a fast, easy, and uniform handling of process switching.

Peripherals using interrupts have widely varying constraints on interrupt processing time. To solve this problem, peripherals with the same characteristics are often associated with one of several interrupts. A priority enforced among the several interrupts allows the required processing time to be guaranteed. Enabling or disabling the various interrupts is the mechanism used to enforce this processing priority.

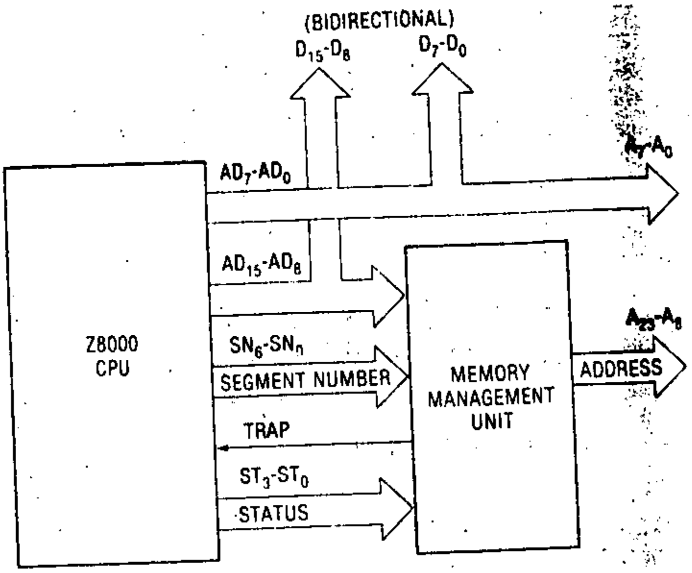


**Figure 5. Memory management device with Z8000 CPU.**

In the Z8000, we felt that three levels of interupts were sufficient. A *non-maskable interrupt* represents a catastrophic event which requires special handling to preserve system integrity. In addition there are two maskable interrupts: *non-vectored interrupts* and *vectored interrupts*, which correspond to a fixed mapping of interrupt processing routines and to a variable mapping of interrupt processing routines depending on the vector presented by the peripheral to the Z8000.

Both interrupts and traps result in similar process switches. Information related to the old process (its program status) is saved on a special system stack with a code describing the reason for the switch.. This allows recursive task switches to occur while leaving the normal stack undisturbed by system information. The state of the new process (its new program status) is loaded from a special area in memory—the program status area—designated by a pointer resident in the CPU (see Figure 6).

The use of the stack and of a pointer to the program status area are specific choices made to allow architectural compatibility if new interrupts or traps are added to the architecture. The choice of the two modes of execution has a strong impact on the design of clean user interfaces. Experience has shown that in large systems the normal mode instruction set and the user interfaces together constitute the most important element in achieving architectural compatibility.

**Communication with other devices: the Z-bus.** The Z-bus is the shared bus which links all the components of the Z8000 family.[4] The variety and performance requirements of the components are so different that in fact the Z-bus is composed of five buses:

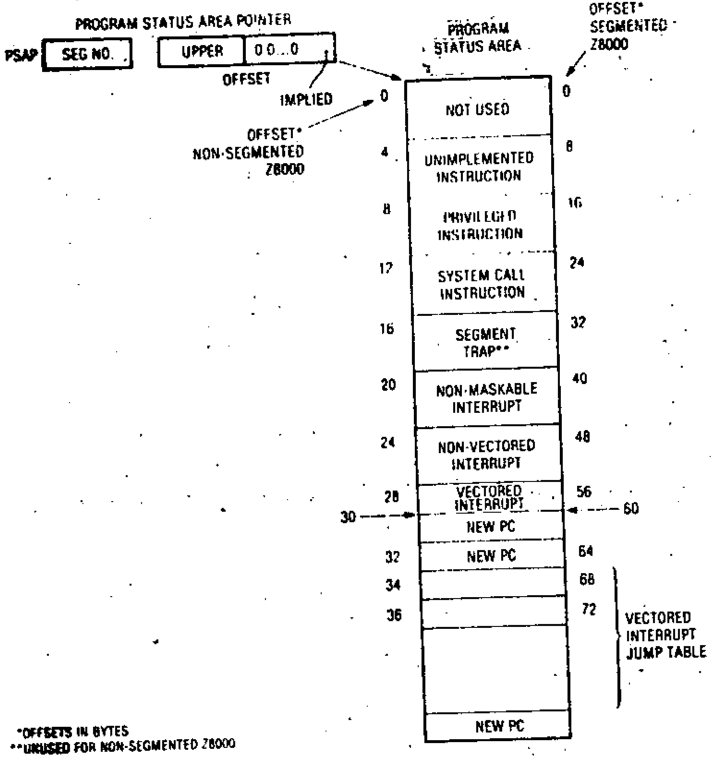


**Figure 6. Program status area.**

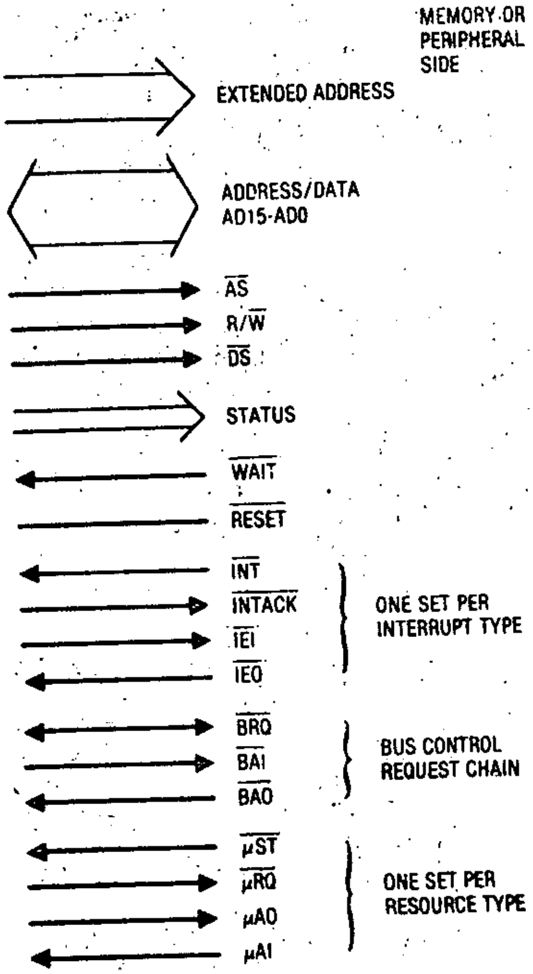


**Figure 7. Z-bus signals.**

a memory bus, an I/O bus, an interrupt bus, and two resource request buses (Figure 7).

The Z-bus is called a "shared" bus because several components can use it. A bus user is a CPU or a peripheral which can usually generate one or more bus transactions such as memory data request or an I/O request. Identical bus transactions cannot take place at the same time, but serialization mechanisms allow sequential use of the Z-bus. Architecturally, the buses can be grouped into two structures. The I/O structure uses the I/O bus and the interrupt bus. The memory structure uses the memory bus with or without address extensions. Both structures can use the resource request bus and the mastership request bus.

Each bus consists of a set of signals and the protocols which preside over the various types of transactions. Part of each protocol is the timing relationship between relevant signals. The Z8000 CPU provides most of these timing relations. The advantage of such a choice is the significant reduction in the number of components required to build such a system. One consequence is that bus transactions cannot be aborted or delayed freely since some devices, especially memory, have specific timing constraints. The most important consideration for the Z-bus is the need to interface to multiplexed address and data lines of the Z8000 CPU which must fit in 40- and 48-pin packages. The Z-bus maintains these multiplexed address and data lines. Very little speed could be gained by demultiplexing these lines for memory references since memories are themselves multiplexed. The most important advantage of a multiplexed Z-bus is the direct addressability of peripheral internal registers. This feature allows the construction of complex peripherals which maintain a simple program interface.

The Z-bus is known as a transparent or asynchronous bus. Z8000 components do not require that their clocks be synchronized with the CPU clock. The signals used by each transaction provide all the necessary timing. This concept is important: it allows, for example, I/O references to be independent of the speed and clock frequencies required by other Z-bus transactions.

*I/O bus versus memory bus.* The I/O and memory buses are the most important. The Z8000 family architecture distinguishes between memory and I/O spaces and thus requires specific I/O instructions. This architectural separation allows better protection and has a nicer potential for extension. The I/O and memory buses use a 16-bit address/data bus, which allows 16-bit I/O addresses and 8- or 16-bit data elements. Memory addresses are 16 bits for the 40-pin package or extended to 23 bits using the segmented version. Thus, the memory bus is in fact a logical address bus. The increased speed requirements of future microprocessors is likely to be achieved by tailoring memory and I/O references to their respective characteristic reference patterns and by using simultaneous I/O and memory referencing. These future possibilities require an architectural separation today. Memory-mapped I/O is still possible, but we feel the loss of protection and potential expandability are too severe to justify memory-mapped I/O by itself.

Both the I/O and memory buses need address, data, and control signals. One important implementation decision was to overlap the signals used by the memory and I/O buses on the same Z8000 CPU pins, with the obvious exception of the status signals used to distinguish between the two types of bus requests. For the current Z8000 implementation the resulting reduction in number of pins is significant. In contrast the impossibility of doing concurrent memory and I/O referencing is not very significant since their speeds are essentially the same.

In addition, memories and peripherals both benefit from the availability of early status information defining the bus transaction type (I/O versus memory, read versus write) ahead of the actual transaction so that bidirectional drivers and other hardware elements can be enabled before the reference. The status lines of the Z8000 CPU provide this type of early status.

*The I/O structure.* Since many peripherals are connected with one CPU, the I/O bus is shared and serialization must be provided. One solution involves using a master/slave protocol. The CPU is a master which can initiate an I/O transation at any time. The peripherals are slaves which participate in a transaction only when requested by the master. In order to find out if a peripheral needs to be serviced the master can poll each in turn. The Z-bus also provides a faster way of getting the attention of a master: an interrupt bus. In contrast, with the I/O transaction data bus, each peripheral sharing the interrupt bus may "try" to use it simultaneously. The interrupt bus uses an interrupt line, interrupt acknowledge line, and two more lines used to form a daisy chain. The daisy chain is an implementation of a distributed arbitration policy between the requests. Priority of processing is determined by the position in the daisy chain, and peripherals can be preempted. Interrupt vectors are used to determine the identity of the peripherals requesting service via an interrupt.

*Other buses.* The two resource request buses are used to request the control of the Z-bus from the CPU and to request control of any generalized resource.

The Z8000 CPU or any Z-bus compatible CPU does not need to request the bus to access it as a master, and is, therefore, the default master. Other devices can request bus mastership, but they must go through a non-preemptive distributed arbitration using another daisy chain. The CPU always relinquishes the bus at the end of its current bus transaction.

The resource request chain is a generalization of that concept in which each resource requestor has equal importance and can use the resource in a non-preemptive manner. This mechanism in the Z8000 CPU permits one to implement in software the kind of exclusion and serialization mechanisms needed for multiple distributed systems with critical resource sharing.

**Multiprocessing.** In the context of today's large mainframe systems characterized by multiple processes sharing one processor, one is tempted to design distributed processing systems with many low-cost microprocessors running dedicated processes. Such an approach distributes intelligence towards the peripherals, results in modularization, and permits easier development and growth. Unfortunately, in the past, the problem with such an approach has been software and not hardware. Thus one cannot be expected to provide detailed solutions in hardware to a software problem that has not been solved yet. However, some basic mechanisms have been provided to allow the sharing of address spaces: large segmented address spaces and the external MMU make this possible, and a resource request bus is provided which in conjunction with software provides the exclusion and serialization control of shared critical resources. These mechanisms and new peripherals like the Z-FIO have been designed to allow easy asynchronous communication between different CPUs.

## Implementation tradeoffs

**The key family decision: producibility.** Confronted with the problem of designing a new LSI-based system architecture, we could have ignored package size considerations by accepting packages with 64 or more pins, or we could have ignored mass production technology constraints by using die sizes larger than 260 mils square. Such solutions are often justified in the implementation of an existing computer system. The component boundaries, package limitations, and technological limitations are secondary to achieving the goal of exact membership in the computer family. But if one were to design a new system architecture with the same lack of constraints, the individual component would not be price-competitive—only the total system would be. A new system architecture based on this approach could only be used to design yet another traditional computer.

**The Z8000 family provides basic, general-purpose blocks out of which a system solution to most problems can be implemented.**

The Z8000 family market is intended to be much broader, and each component of the family must be economically viable. The staged introduction of components which are economically viable by themselves allows us to serve the market from very small configurations to very large configurations by using more components, in any combination. Not only do we believe that this approach does not restrict

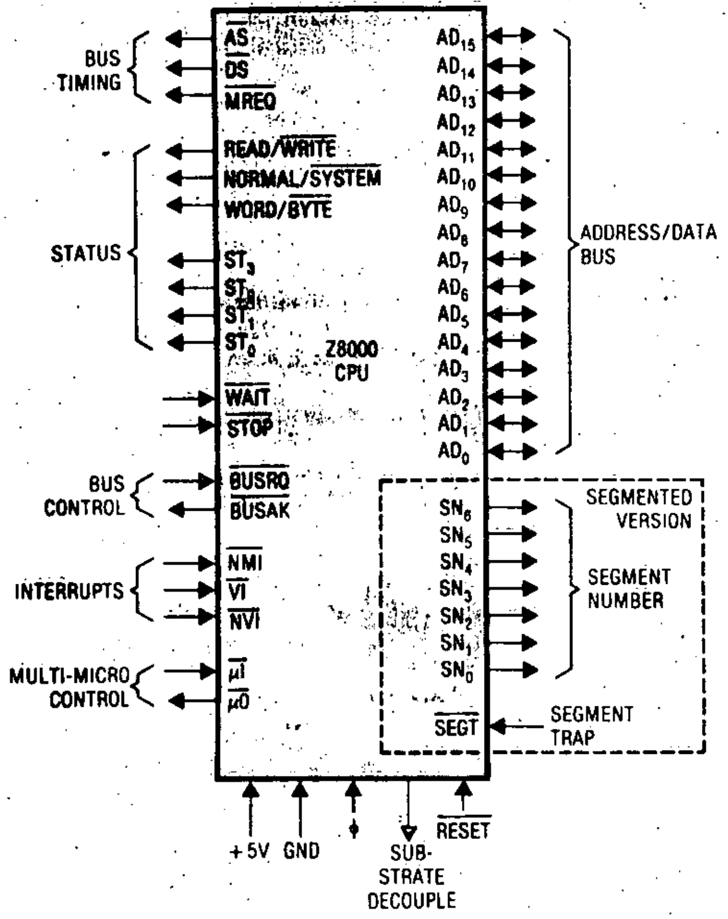


**Figure 8. Z8000 pin functions.**

system architectural possibilities, but we also believe that the family will be more effective because it will grow with its customer.

The Z8000 family does not always attempt to provide specific architectural solutions, often implemented in hardware, to all system architecture problems. Instead, it provides basic, general-purpose blocks out of which a system solution to most problems can be implemented. The multi-microprocessor and distributed system capabilities of the Z8000 family illustrate the use of open-ended mechanisms to solve a variety of architectural problems, while the memory management of address space illustrates a specific problem supported by a specific solution—the MMU. However, other solutions more appropriate to a particular problem can be used and an advance in the state of the art might be mapped into a new device for the family.

This vision of the family often results in components more powerful and complex than an application may require. The user should not take this as a cause for alarm, but rather as the reason his applications growth will be easier.

**Basic CPU implementation decisions.** The Z8000 currently uses a 16-bit data bus (Figure 8), an internal register array of 16-bit registers, and a 16-bit parallel ALU. These implementation decisions, which were guided by the technological and practical considerations, have a strong impact on performance.

To achieve good performance with the instruction format and data type envisioned for the Z8000, only a 16-bit bus seens adequate; a 32-bit bus would have necessitated using an unacceptable 56-pin or larger package. Optimal performace is obtained with this chosen bus width if the size of the frequently used register-to-register operations becomes one word. The choice of ALU and internal register widths is a tradeoff between speed of the most frequent operations and the chip area needed to implement a wider ALU or data path inside the CPU.

None of these implementation decisions should limit the architecture. Instructions are from one to five words long, and data types and addresses are not limited to 16 bits. For example, 32-bit words are one of the main data types of the machines, and addresses occupy two words. The *address* mechanism illustrates the strong distinction between an architecture and its implementation. The architectural address representation uses a 32-bit word of which 8 bits are reserved and 1 is a short format/long format descriptor. Thus, the Z8000 architecture provides up to 31-bit addresses, but only 23 are currently implemented and 23 pins of the current package are allocated to addresses.

**MMU tradeoffs.** The MMU and its relation to the Z8000 CPU illustrate tradeoffs that a microprocessor architect and designer team must make to ensure component manufacturability.

To achieve the goals of good architectural compatibility for high-end systems, it was necessry to include the protection and relocation mechanicms described above. But if all desired features were implemented as a one-chip CPU/MMU combination, it would have been too large and, therefore, uneconomical. And if a reduced set of features were implemented, it would have been architecturally too primitive. Thus, the choice was made to maintain all features and use two chips. This new organization has several significant advantages, such as a capability for multiple MMUs, and allows the access of a DMA device to the MMU.

Given the choice of an external MMU, the next set of decisions concerns package size and circuit speed. Having each relocated segment start on a word boundary would have required a 64-pin package and a very fast 24-bit adder (in fact, a 16-bit adder and 8 bits of carry propagation). In contrast, the decision to start segments on 256-byte boundaries allows the use of a 48-pin package, a fast 8-bit adder, and 8 bits of carry propagation. The latter solution is technically superior and places practically no restriction on the architecture. Segment granularity can be viewed as an implementation restriction and not as an architectural restriction.

Making the 8 low-order bits of the offset go directly to memory also significantly reduces memory access time. Since dynamic memories use these bits first, most of the MMU relocation time is hidden during a normal memory access. The availability of segment numbers earlier than the associated offset bits reinforces this advantage and allows the MMU to result in essentially no memory access speed reduction. Each MMU entry also requires 8 bits less for base and segment size value. This is important: it is desirable to pack as many entries as possible per MMU. With 64 entries a 2K-bit memory is needed, which is technologically difficult in view of the amount of logic surrounding this memory and the complexity of its organization.

The fact than an MMU is only connected to the upper byte of the data bus requires the use of special I/O instructions for its loading and obliges us to replace the possible use of an automatic demand loading of entries by explicit instruction loading. To compensate for the time penalty associated with the loading of potentially unused entries, multiple MMUs are used. They not only allow the implementation of 128 entries, but pairs of MMUs can be automatically enabled by the system and normal mode pins effecting a full environment switch at electronic speed.

We feel this example illustrates one important design approach: to compromise as little as possible on advanced architectural features but to accept compromises which result in implementation ease in order to achieve economical components.

## Conclusion

The architectural sophistication of the new 16-bit microprocessors is rapidly approaching the level of the minicomputer and large computer. Problems such as component families, large address spaces, bus standards, I/O structures, software investments, and architectural compatibility are being directly addressed. Some of the solutions to these problems are known, and therefore the transition from 8-bit microprocessors was relatively easy. But the challenges ahead—networks, distributed processing, new applications—are much harder. The impact of microprocessors is already enormous, but we feel they will achieve the often-predicted computer revolution only after these new problems are solved. ■

## Acknowledgements

The Z8000 family would not exist without the very talented and dedicated designers who contributed to and implemented the ideas described in this paper: Masatoshi Shima for the Z8000, Hiroshi Yonezawa for the MMU, and Ross Freeman for the peripheral devices. Judy Estrin made invaluable contributions to the architecture of the Z8000 and Z8. Many discussions with Charlie Bass, Leonard Shustek, and Forest Baskett have greatly influenced the Z8000. Leonard's instruction set measurements were especially valuable. Dennis Allison, Steve Meyer, Bruce Hunt, and many others must be thanked for their comments on early drafts of this paper.

## References

1. B. L. Peuto and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, Vol. 10, No. 2, Feb. 1977, pp. 20-25.
2. Zilog, *Z8000 Technical Manual*, Zilog, Inc., 1979.
3. Zilog, *MMU Technical Manual*, Zilog, Inc., 1979.
4. Zilog, *Z-Bus Specification*, Zilog, Inc., 1979.
5. A. Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *CACM*, Vol. 20, No. 3, Mar. 1977, pp. 143-152.
6. L. J. Shustek, *Analysis and Performance of Computer Instruction Sets*, PhD Dissertation, Dept. of Computer Science, Stanford University, Stanford, Calif., Jan. 1978.
7. B. L. Peuto and L. J. Shustek, "An Instruction Set Timing Model of CPU Performance," *Proc. Fourth Annual Symposium on Computer Architecture*, Mar. 23-25, 1977, pp. 165-178.
8. C. Bass, "PLZ: A Family of System Programming Languages for Microprocessors," *Computer*, Vol. 11, No. 3, Mar. 1978, pp. 34-39.
9. A. S. Tannenbaum, "Implications of Structured Programming for Machine Architecture," *CACM*, Vol. 21, No. 3, Mar. 1978, pp. 237-246.
10. N. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, Vol. 8, No. 11, Nov. 1975, pp. 41-46.

Bernard L. Peuto is one of the guest editors for this special section; his biography appears with the introduction on p. 9.

# The Intel MCS-48 Microcomputer Family: A Critique

John F. Wakerly,
Micro Systems Engineering

*A system designer and teacher, who has made liberal use of microcomputers in his own work and whose students have designed 8048 processors, reviews the capabilities and limitations of the MCS-48 family of microcomputers.*

The Intel MCS-48 family of single-chip microcomputers contains at least nine different microcomputer chips having a common instruction set but different amounts of on-chip read-only memory, read/write memory, and input/output (see Table 1). According to Intel, the MCS-48 family was originally aimed primarily at the "4-bit market"—users of Intel's 4040 and other low-cost microcontrollers. Recent entries into the family (the 8021, 8022, 8041, and 8741) are increasingly specialized for low-end microcontroller applications. The MCS-48 family has met this market very well.

Table 1.
MCS-48 microcomputers.

| PART # | PACKAGE SIZE (pins) | ON-CHIP PROGRAM MEMORY (bytes) | ON-CHIP DATA MEMORY (bytes) | I/O (lines) |
|---|---|---|---|---|
| 8048 | 40 | 1K ROM* | 64** | 27 |
| 8748 | 40 | 1K EPROM* | 64** | 27 |
| 8035 | 40 | none* | 64** | 27 |
| 8049 | 40 | 2K ROM* | 128** | 27 |
| 8039 | 40 | none* | 128** | 27 |
| 8021 | 28 | 1K ROM | 64 | 21 |
| 8022 | 40 | 2K ROM | 64 | 23 plus 2 8-bit A/D conv. |
| 8041 | 40 | 1K ROM | 64 | 18 plus master sys. intf. |
| 8741 | 40 | 1K EPROM | 64 | 18 plus master sys. intf. |

* Expandable to 4K with external chips
** Plus 256 bytes or more of external data memory with external chips

The MCS-48 family was also aimed at a second market—applications that require an expandable, single-chip, general-purpose microcomputer. As shown in Table 2, several expansion chips are available to provide an MCS-48 computer with up to 4K bytes of program ROM, 256 or more bytes of external RWM, and as many I/O bits as a designer would ever need. In addition, the external I/O bus of the MCS-48 family allows easy interfacing of standard 8080/8085-compatible peripheral chips. Nevertheless, the architecture of the MCS-48 family makes it difficult to use in many general-purpose applications, where a more capable 8-bit architecture is required.

Table 2.
MCS-48 expander chips.

| PART # | PACKAGE SIZE (pins) | ON-CHIP PROGRAM MEMORY (bytes) | ON-CHIP DATA MEMORY (bytes) | I/O (lines) |
|---|---|---|---|---|
| 8355 | 40 | 2K ROM | none | 16 |
| 8755 | 40 | 2K EPROM | none | 16 |
| 8155/56 | 40 | none | 256 | 22 plus timer/counter |
| 8243 | 24 | none | none | 16 |

## Basic architecture

Figure 1 shows the basic structure of an MCS-48 microcomputer chip. (Table 1 gives the facilities available for each of the microcomputers in the MCS-48 family that had been announced by late 1978.) The first member of the family was introduced in late 1976—the 8048 with 1K bytes of on-chip ROM, 64 bytes of RWM, timer/counter, and 27 I/O bits. A detailed description of the entire family can be found in the user's manual published by Intel.[1]

The MCS-48 is a single-accumulator architecture. Program memory and data memory are logically and physically separated (thus, the MCS-48 is not a von

0018-9162/79/0020-0022$00.75 © 1979 IEEE

Neumann machine!). The maximum program address space (including external ROM) supported by the architecture is 4K bytes. There are a maximum of 256 bytes of on-chip (internal) data memory, of which 128 bytes are implemented in the current family leader, the 8049. In addition to internal data memory, the MCS-48 directly supports 256 bytes of external data memory.

Most MCS-48 family members have 27 I/O pins, arranged as three 8-bit ports, two test inputs, and an interrupt input. Additional pins are provided for such functions as power-on reset, single-stepping, and memory and I/O expansion strobes. One 8-bit port and part of a second are used to form a multiplexed address and data bus for I/O and memory expansion.

The MCS-48 has a single-level interrupt system (only one interrupt in service at a time) and accepts interrupts from two sources—its internal timer/counter and an external interrupt input pin. Interrupt calls and returns automatically push and pop the program counter and certain internal status flags using a stack in the internal data memory.

## Program store and program control

The MCS-48 architecture supports a maximum of 4K bytes of program store, configured as shown in Figure 2. However, a close look at program-store organization shows that the MCS-48 was originally designed as a 2K-byte machine, with the second 2K-byte capability added as a clumsy afterthought. This creates two problems with the addressing mechanism.

First, the program counter is really only 11 bits and thus addresses instructions only within a 2K-byte bank of program store. Jump and subroutine call instructions likewise specify an 11-bit address. The problem, then, is how to provide a 12th address bit.

Intel's solution is as follows. Provide an internal flag, MB, that can be set and cleared by two instructions (SEL MB1 and SEL MB0, respectively). Whenever a jump or subroutine call is executed, take the 11 low-order PC bits from the instruction, and load the high-order bit from MB. On subroutine calls and returns, push and pop the entire 12-bit address.

There are some problems with this solution. First, in a general sequence of jumps and calls in a 4K system, we don't always know where we came from, and therefore we don't know the current value of MB. So in general, a SEL MB*i* instruction must precede every jump or call. Naturally the programmer can sometimes avoid this instruction on a case-by-case basis, but this is another thing to worry about.

Having solved the first problem, we think we understand the addressing mechanism until we write our first interrupt routine. Then we wake up in the middle of the night thinking, "Whoops! MB can't be read as part of the processor state PSW. But MB must be set to a new value in order to do jumps within the interrupt routine. How can the old value be restored on return?" We lie awake a few hours dreaming up possible solutions—don't use calls or jumps in in-

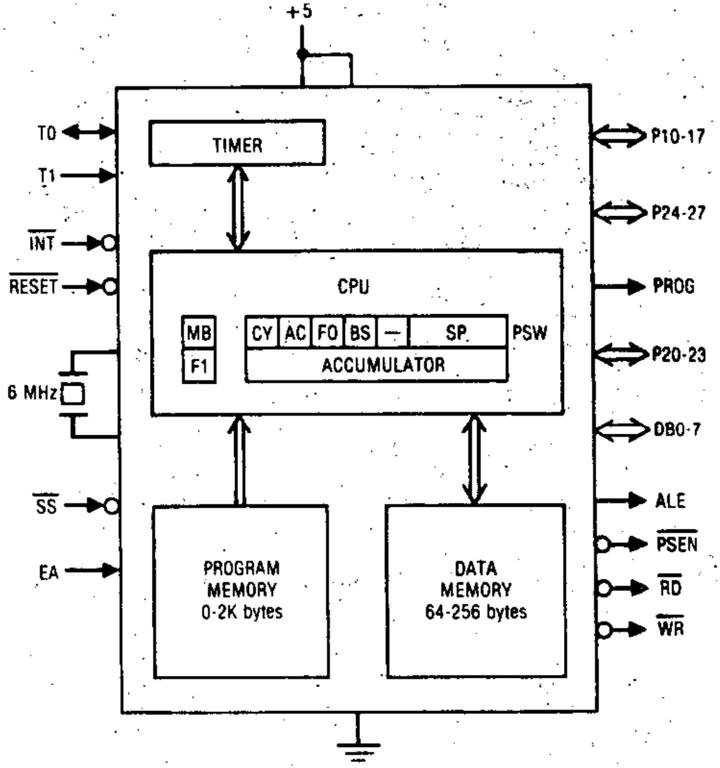


**Figure 1. Basic structure of a typical member of Intel's MCS-48 family of microcomputers.**

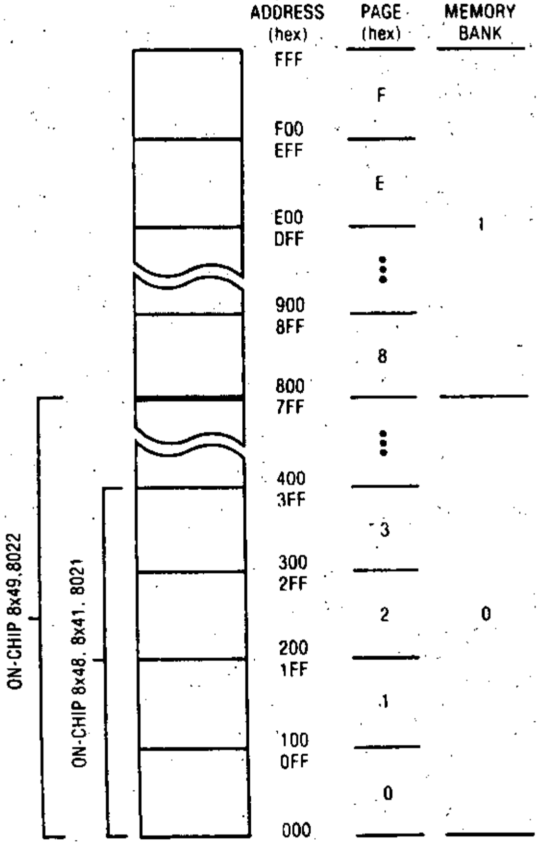


**Figure 2. MCS-48 program memory.**

## How to choose a microcomputer

There are at least six factors to consider in choosing a microcomputer (or microcomputer family) for a product application.

**Capability.** The μC must have enough ROM, RWM, I/O capability, and speed to satisfy the requirements of the application, plus a design margin. ROM, RWM, and I/O capability can be determined from the manufacturer's literature, while speed is best determined from benchmark programs tailored for the given application.

With some μCs, the amount of ROM, RWM, and I/O can be increased by using extra chips. This expandability helps if the job is initially underestimated or if the marketing department changes the requirements. If the application pushes the absolute memory limits of the μC, it becomes more difficult (and expensive) to develop the programs.

**Extensibility.** The designer must consider whether improved versions of the μC will be offered. A μC-based product designed in 1979, for example, might be redesigned in 1981 to reduce cost or to add features. It would then be desirable to eliminate extra ROM, RWM, and I/O chips (or avoid having to add them or pick a different μC) by using a new version of the original μC with the extra capability built in. Of course, many products do not undergo this evolution; but if product evolution is expected, the architectural limits of the selected μC should be examined in light of potential application requirements. One can expect lower hardware and software development costs and, probably, lower manufacturing costs if the enhanced product uses an upgraded version of the original μC rather than a completely new one.

**Cost.** In most areas of the private sector, minimizing cost is a goal, and minimizing μC cost usually means minimizing the number of IC packages. Cost is what prevents the designer from picking a Cray-1 in response to the first factor above, or an IBM 370 in response to the second factor.

If the job is well defined and no product enhancement is anticipated, it is relatively easy to find a minimum-cost μC that will do the job. Otherwise, there are many more tradeoffs to be considered. A simpler μC architecture usually implies a smaller IC die and lower chip cost, but it may also require more chips to support it later. (For example, a μC without a WAIT/READY line may be more difficult to interface to some types of peripherals or memory.) An expandable μC will facilitate later product evolution (if the product is successful), but may increase initial product cost because of instruction efficiency, memory size, I/O pins, or speed sacrificed by the chip designers to make expansion or enhancement possible.

**Availability.** Many manufacturing organizations require a second source for all components, both to ensure that parts will be available, even if some disaster befalls one source, and to enjoy the normal benefits of competition in a free market.

The designer of a new product is often tempted to select between a μC with one or two sources and one with *no* sources (yet—"We'll have samples in three months"). It is risky to commit to any part unless your purchasing department can order (and receive) 100 pieces from a distributor's shelf. Manufacturers *have* been known to slip schedules and even cancel parts.

On the other hand, marketing and cost factors can motivate the selection of a not-yet-available or very new μC. The new μC can give the product a competitive edge in features or performance. Although the new μC may be in short supply and costly initially, it may be cheaper in the long run because it allows a more efficient design with fewer IC packages.

Expected product lifetime should also be compared with the expected lifetime of the μC. Even if it is inexpensive currently, a μC that has been around for a few years may be a bad choice: production quantities may fall and prices rise in a few more years as newer chips are phased into new designs. Of course, this doesn't apply if your company alone is ordering 100,000 pieces per year.

**Support tools.** Hardware and software support tools are essential for timely development of a μC-based product. The support tools of a newly introduced μC cannot be expected to be as extensive or reliable as those of an established μC family. This encourages the use of an established μC if quick development is needed, or an extensible μC family with reusable tools if product evolution is expected.

Most single-chip μCs are programmed in assembly language, and a good macroassembler is a must. Most manufacturers supply software tools that run on their own development systems. However, if there are more than one or two programmers on the project, the need for good text editors, simulators, and documentation facilities makes it desirable to run all software support tools on a large central computing facility. The appropriate "cross assemblers" and simulators may or may not be offered by the chip manufacturer.

During product development it is obviously necessary to test and change programs running on the product hardware. Since most single-chip μCs ultimately use mask-programmable ROM to store their programs, another means is needed to store and change programs during development without making new masks. Some μCs have pin-compatible versions with on-chip EPROM instead of ROM that allows reuse of the μC chip with different programs. Many have provisions for using external EPROM chips instead of the on-chip ROM. If production quantities are low, or if software changes are expected after product introduction, EPROM versions may be essential.

Besides EPROM facilities, the main support tool provided by the chip manufacturer is the in-circuit emulator, which stores the software program in the RWM of a development system and emulates the μC through a cable and plug inserted in place of the μC in the product. An emulator is a useful tool for debugging both hardware and software. However, with new μCs, it may not be available as soon as the μC chips are, and even if it is, it may still have bugs.

**Specific technical factors.** Many specific technical factors can be examined in determining whether a μC will do the job at hand—power consumption, speed, TTL compatibility, package size, instruction set. However, once it is determined that the μC can do the job, the other factors above tend to equal or outweigh the technical "niceness" of the μC chip architecture.

terrupt routines; determine the value of MB experimentally by doing a jump to a fixed location and seeing whether it winds up in MB0 or MB1; keep a software copy of MB, updating it (with interrupts disabled, of course) every time we do a SEL MB*i*; make all the code fit in 2K, as we expected to do at the start of the project; and so on. The next morning we read the fine print to discover that the MCS-48 forces the most significant bit of the program counter to 0 during all interrupt routines. We should put all of our interrupt code in the bottom 2K of memory and not touch MB; in fact, we should forget that MB exists!

The requirement to put interrupt code in the bottom 2K makes the MCS-48 very difficult to use as a 4K machine in a real-time application. Not only must the basic interrupt service routine be in the bottom 2K but also any utility routine that might be called by it—that is, any code executed before an interrupt return instruction is executed. This could be well over half the code in an interrupt-driven environment.

But the main problem we find with MCS-48 program store, after writing half of our applications programs, is that the address space is just too small. With only two chips (and soon with just one, I'm sure) we can fill the entire 4K-byte address space of the MCS-48 with code for our original application, new features, diagnostics, and—of course—patches.

**Conditional jumps specify an 8-bit target address in the current page; it would be far more useful to have a signed offset from the current address.**

The annual halving of the cost of IC memory implies that every year we will need another address bit for the maximum-size application program (since most evolving products tend to use the decreased memory cost to increase features, not to reduce product cost). Clearly, then, a 4K limit is too low for any new architecture, even a single-chip microcomputer.

Besides the 2K memory banks, program store is also divided into 256-byte pages. Conditional jumps specify an 8-bit target address in the current page. It would be far more useful to have a signed offset from the current address; this would increase the likelihood of being able to use the short jump address, since most branch targets are within 128 bytes of the branch instruction.[2] More importantly, it would eliminate the partitioning problem created when many procedures must be packed into the memory space and split across page boundaries.

The only indirect jump instruction also uses an 8-bit target address in the current page. Very strangely, this instruction uses an 8-bit value in the accumulator not as the target address, but as a pointer to a program-store byte in the current page that contains the target address. So the page containing the indirect jump instruction must also contain all of the routines to be jumped to, as well as a silly little table that contains their starting addresses in the page. This not only wastes space and time, but, worse, makes it impossible to dynamically compute a target address after assembly time, since the jump table is in ROM. In my recent experience, three experienced programmers have coded MCS-48 indirect jumps improperly, believing "The manual must have a typo—the contents of the accumulator must be the target address itself." In any case, instructions supporting indirect jumps and calls anywhere in the program store (12-bit address) would be far more useful.

## Arithmetic and logical operations

The MCS-48 contains a single accumulator in which arithmetic and logical operations take place. Unary operations on the accumulator are as follows:

increment,
decrement,
clear,
one's complement,
decimal adjust,
swap nibbles,
rotate left,
rotate left with carry,
rotate right, and
rotate right with carry.

Binary operations combine the accumulator and an operand specified by one of the addressing modes described in the next section. The binary operations are:

add,
add with carry,
AND,
OR, and
exclusive OR.

There are also "data-move" operations that load or store the accumulator.

The main difficulty with MCS-48 operations is not the operations themselves but the lack of condition codes for testing their results. Only the accumulator can be tested for zero or negative, and an overflow bit is not provided, making comparisons of signed two's-complement numbers very frustrating.

## Operands

Most data moves and binary operations use an on-chip read/write internal data memory (see Figure 3) accessible by two addressing modes: REGISTER and INTERNAL REGISTER INDIRECT. The three other addressing modes are EXTERNAL REGISTER INDIRECT, IMMEDIATE, and ACCUMULATOR INDIRECT.

In REGISTER mode an operand is contained in a register specified by a 3-bit field in the instruction. A flag bit BS, set by a SEL RB*i* instruction, specifies one of two 8-byte register banks, corresponding to internal data memory locations 0-7 if BS is 0 and 24-31 if BS is 1. The specified register may be loaded with an immediate value, moved to or from the accumulator, combined with the accumulator by arithmetic or logical operations, incremented, decremented, or used as a loop counter.

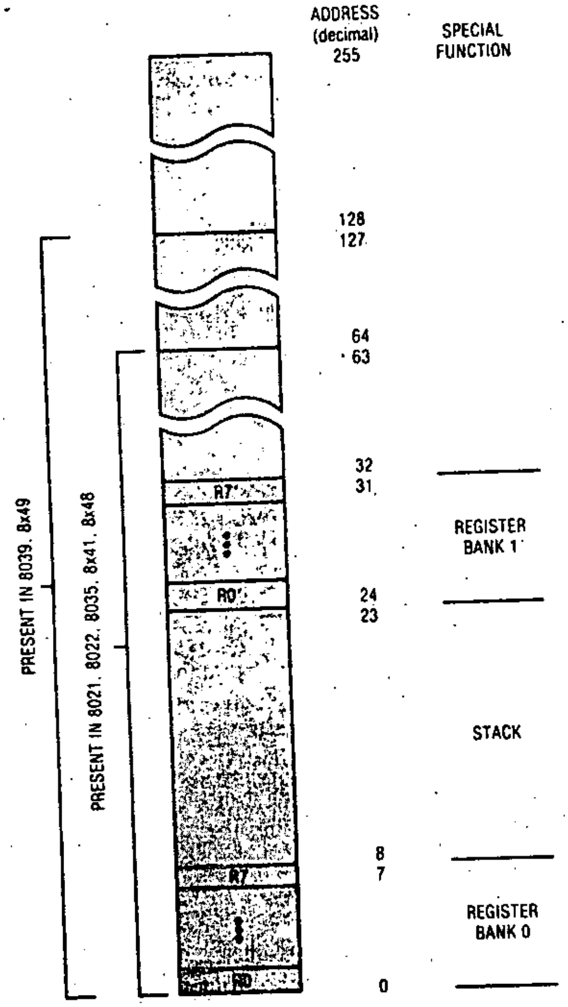


**Figure 3. MCS-48 internal data memory.**

INTERNAL REGISTER INDIRECT allows either R0 or R1 in the current register bank to be used as an 8-bit pointer to internal data memory. The addressed byte may be loaded with an immediate value, moved to or from the accumulator, combined with the accumulator, or incremented (but for some obscure reason not decremented, even though the necessary "hole" exists in the instruction set).

Locations 8-23 of the internal data memory are reserved for a return address stack (8 entries, 2 bytes per entry). These locations are written by interrupts and subroutine calls and read by interrupt and subroutine return instructions. The stack is too small, making it hard to write procedural code, which is important in larger programs ( 2K-4K bytes). The programmer must constantly worry about calling sequences and generally enable interrupts only at the top level of the program to avoid overflowing the stack.

There are no instructions to directly push or pop a byte. However, the stack can be rather inconveniently written or read by extracting the stack-pointer field from the PSW, building the appropriate address, and using INTERNAL REGISTER INDIRECT mode.

The architecture also supports up to 256 bytes of external data memory (which resides on a separate chip), accessed by EXTERNAL REGISTER INDIRECT mode. Either R0 or R1 in the current register bank may be used as an 8-bit pointer to external data memory; the addressed byte may be copied into the accumulator or written from the accumulator.

Since pointers are contained in 8-bit registers, the maximum amount of directly accessible data memory supported by the MCS-48 architecture is 256 bytes internal plus 256 bytes external. However, bank switching via I/O bits can be used to address any desired amount of additional external data memory.

The modes for reading operands from program store are rather limited. In IMMEDIATE mode an operand is contained in the byte following the instruction; immediate operands can either be loaded into or combined with the accumulator or be loaded into internal data memory with REGISTER or INTERNAL REGISTER INDIRECT modes.

In ACCUMULATOR INDIRECT mode the accumulator is used as an 8-bit pointer to an operand in either the current page or page 3 of program store; only one type of operation uses this mode, and it loads the accumulator with the specified operand.

A number of instructions specify some "special" operands implicitly, such as the program status word, I/O ports, timer/counter, carry bit, and two 1-bit flags, F0 and F1.

The MCS-48 addressing modes are simple, but they provide most of the facilities a program needs. Still, there are some deficiencies. The most serious problem is the way in which operands in program store are addressed. Since program store only in the current page and in page 3 can be read through a pointer, either lookup tables must all be located in page 3 or the code that reads each table must be in the same page as the table. This is inconvenient if more than one 256-byte translation table is needed. It also makes it difficult to do a ROM checksum self-test routine—a checksum subroutine would have to be placed in every page of program store (and since there is no indirect subroutine call, the main checksum program would have to contain a separate call instruction to each page's checksum routine).

For most programs, the method of indirectly addressing data memory through R0 and R1 is acceptable, but, for some data-structure manipulations, one wishes for one or two more registers that could be used as pointers.

The 256-byte limit on directly addressable internal data memory is too low. The 8049 already contains 128 bytes of RWM, and Intel should soon be able to provide the full 256 bytes of RWM on one chip. The architecture cannot make straightforward use of technology improvements for more RWM once this limit is reached.

## Input/output and interrupts

Most MCS-48 microcomputers have three 8-bit I/O ports, as shown in Figure 1. Two of the ports (1 and 2)

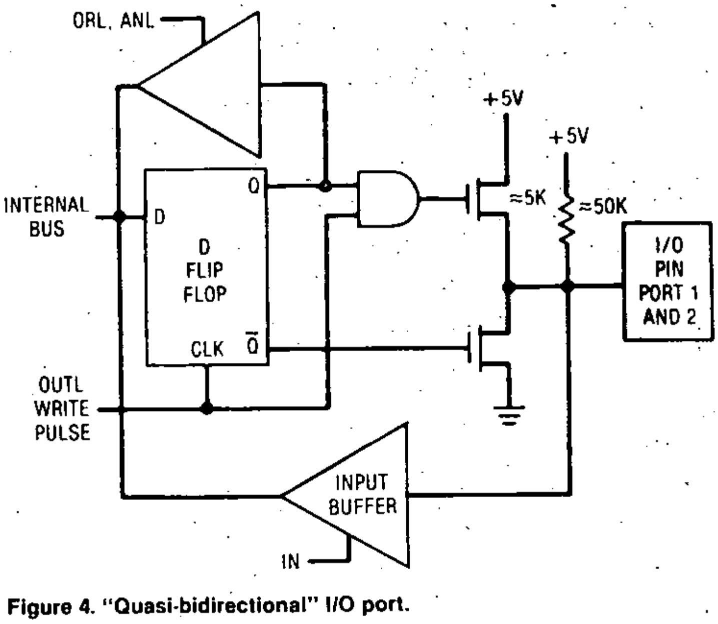


**Figure 4. "Quasi-bidirectional" I/O port.**

are "quasi-bidirectional," an interfacing arrangement shown in Figure 4. This type of I/O port was first introduced in the Fairchild F8.[3] In this arrangement, each I/O pin is both an open-drain output and an input pin with a high-impedance pullup to the logic 1 level. When a pin is used for output, the corresponding input buffer is unused except, possibly, for checking the output value. When a pin is used for input, the corresponding output bit must be set to logic 1 so that the I/O device drives only the high-impedance pullup. This can be contrasted with a tristate I/O port, which provides both active pullup and active pulldown in output mode and high impedance in input mode. Electrically, tristate I/O is more desirable, but it requires extra control bits to set the I/O direction for each port or bit. Intel has improved the quasi-bidirectional design by briefly providing active rather than passive pullup whenever a 1 is written to the port, which speeds up 0-to-1 transitions.

What quasi-bidirectional I/O means to the programmer is that input data on the port is logically ANDed with the current output. Ports 1 and 2 are set to all 1's at system reset, and the programmer must leave bits intended for inputs set at output value 1 at all times. The third port (bus) has conventional tristate outputs and can be used for eight strobed inputs, for eight strobed outputs, or for adding external program or data memory.

Four operations on the ports are available:

read input value into accumulator (IN),
load output latch from accumulator (OUTL),
logical AND output latch with immediate mask (ANL), and
logical OR output latch with immediate mask (ORL).

The logical operations allow a program to set or clear any bit or group of bits in one instruction. However, since the mask is an immediate value in program store, the bits to be set or cleared must be known at assembly time. Otherwise, a copy of the output value must be kept in data memory, combined with the mask by logical operations on the accumulator, and loaded into the port. (In general, the quasi-bidirectional interface prevents simply reading the port to get the old value of the output latch.)

A novel "expander-port" arrangement allows four external 4-bit I/O ports to be added to an MCS-48 using a five-wire interface. Again, four operations on the ports are available:

read input value into accumulator,
load output latch from accumulator,
logical AND output latch with accumulator, and
logical OR output latch with accumulator.

Only the low-order four bits of the accumulator are used in these operations. For these ports, dynamic selection of mask bits is possible because the mask is in the accumulator. On the other hand, dynamic selection takes more overhead because the accumulator must be loaded with the mask (and then possibly restored to its old value).

Both the on-chip and expander I/O port instructions contain the port number as an immediate value in the instruction; it is not possible to specify the port number dynamically in a register. This makes it impossible to write reusable I/O handlers for identical devices on different ports of the MCS-48 or of the same expander chip. The same problem exists in the MCS-48's older brother, the 8080. However, it is less serious in the MCS-48 for two reasons. First, the MCS-48 is intended for smaller applications less likely to employ many copies of the same I/O device. Second, the available I/O expansion modes do allow dynamic device selection when each device uses a separate I/O chip.

The processor architecture directly supports only 256 bytes of external data memory and four external 4-bit I/O ports. However, the amount of external data memory and I/O can be increased to any practical amount using on-chip I/O-port bits to implement program-controlled bank switching.

In addition to the I/O ports, an MCS-48 has three additional input pins that can be tested by conditional jump instructions. All are multipurpose pins—T0, which can be set up as a clock output under program control; T1, which can be used as the input to the on-chip timer/counter; and the external interrupt input.

The MCS-48 accepts interrupts from two sources—a level-sensitive input pin and an on-chip timer/counter. When an interrupt is serviced, the 12-bit PC and four status bits (carry, half carry, flag 0, register bank select) are pushed onto the internal stack. Depending on the source, a jump to either location 3 or location 7 is taken. The interrupt system is single-level; interrupt service routines cannot be interrupted. An interrupt return instruction restores the PC and status bits and allows further interrupts to be serviced.

At the time of this writing, the T1 interrupt input is generally useless for counting or timing asynchronous external events, because the current chip

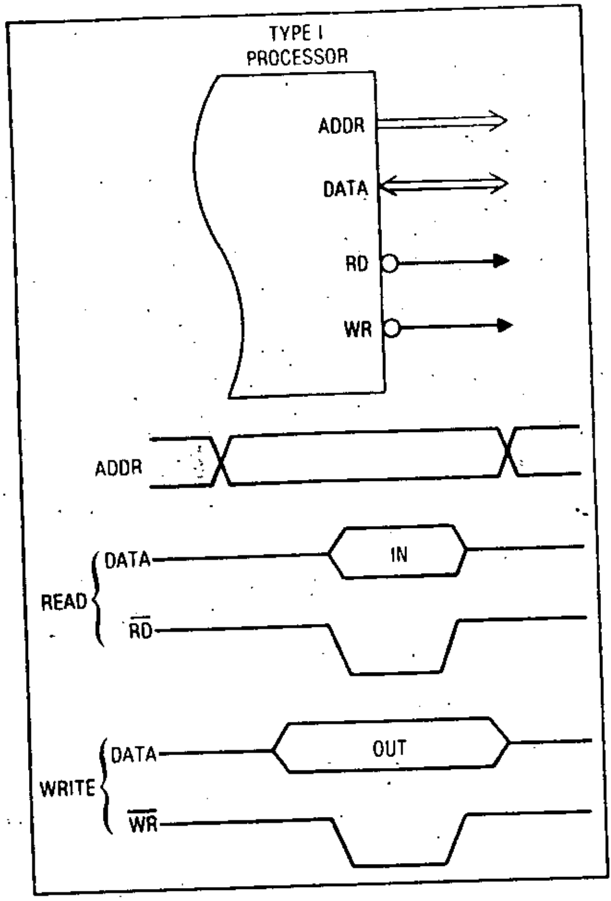


Figure 5. Type I bus control signals.

features a poorly designed synchronizer that sometimes misses input edges and hence skips counts. This is the second time in six months that I have seen an LSI chip whose designers were apparently unaware of problems in synchronizer design (the other was the Z80-SIO, which Zilog has since fixed). I would suggest that chip designers read some of the papers on the subject[4-6] and that academics warn their students of the increasing likelihood of synchronization problems in modern system design.

## Ease of programming

Compared to some of the older 4-bit and 8-bit microprocessors, the MCS-48 is a nice machine to program, but it leaves much to be desired compared with an M6801, a Z8, or even an 8085. The single-accumulator architecture, the lack of index registers, and the absence of even a direct data memory addressing mode means that the programmer must constantly be moving things back and forth between the accumulator, the two "pointer" registers R0 and R1, and the rest of the data memory (and keeping track of them!). One may write macros to ease the burden somewhat, at the expense of more inefficient code in the cramped address space. For example, one can write a macro to simulate a direct data-memory-addressing mode:

```
LDA     MACRO MEMADDR
        MOV R0, #MEMADDR
        MOV A,@R0
        ENDM
```

### A tale of two buses (or, different strobes for different 'phobes)

A microprocessor memory and I/O bus has many identifying characteristics—data and address word length, multiplexed or nonmultiplexed address and status, separate or memory-mapped I/O, and others.[3] It is interesting to look at two popular read/write clocking arrangements.

Let us call the first technique a Type I (or 1) interface, used by the Intel 8085, 8086, and MCS-48 families. As shown in Figure 5, there are two mutually exclusive control pulses, $\overline{RD}$ and $\overline{WR}$, that indicate a read or write operation.

We'll call the second technique Type Z (or 2), used by the Zilog Z8, Z80, Z8000, and also by the Motorola 6800 family. (Perhaps it should be Type M because the M6800 came first, but Z looks more like 2.) It is also used in principle by MCS-48 expander ports. As shown in Figure 6, there is a single control pulse, $\overline{REQ}$, and a level signal $\overline{R}/W$ that indicates which type of operation is to take place. The timing of $\overline{R}/W$ is similar to that of an address signal.

Figure 7 shows how to use a Type Z processor with a Type I peripheral chip. The decoding shown in the figure can be easily implemented with one-half of a TTL 74LS139 dual 2-to-4 decoder (this even leaves an extra control input for distinguishing between memory and I/O if desired). Assuming that processor and peripheral speeds are comparable, there should be no problem in satisfying the timing requirement of either the processor or the peripheral chip.

Figure 8 shows an attempt to use a Type I processor with a Type Z peripheral chip. The logical AND of $\overline{RD}$ and $\overline{WR}$ from the processor nicely produces $\overline{REQ}$ for the Type Z peripheral. $\overline{RD}$ has the correct logic value to serve as $\overline{R}/W$, but its timing is a problem. The Type Z peripheral expects $\overline{R}/W$ timing to be similar in character to an address signal, that is, it should be valid long before the $\overline{REQ}$ pulse appears. The only way we could ensure this would be to artificially delay $\overline{REQ}$ long enough for $\overline{RD}$ to satisfy the setup time of $\overline{R}/W$. Unfortunately, such a delay (unless highly asymmetric) would also delay the trailing edge of REQ until long after $\overline{R}/W$ ($\overline{RD}$) had gone away—again a problem.

The cleanest way to use a Type I processor with Type Z peripheral chips is to use an address line as $\overline{R}/W$. For example, the least significant bit of the I/O port address could be reserved as $\overline{R}/W$. Hardware decoding of actual port numbers would use the higher-order bits; then software would have to ensure that writes always used odd port addresses, and reads used even.

The conclusion is that it is simple to connect Type I peripherals to Type Z processors, but that the reverse can be difficult. Is this just the way it turned out or was there method to this madness?

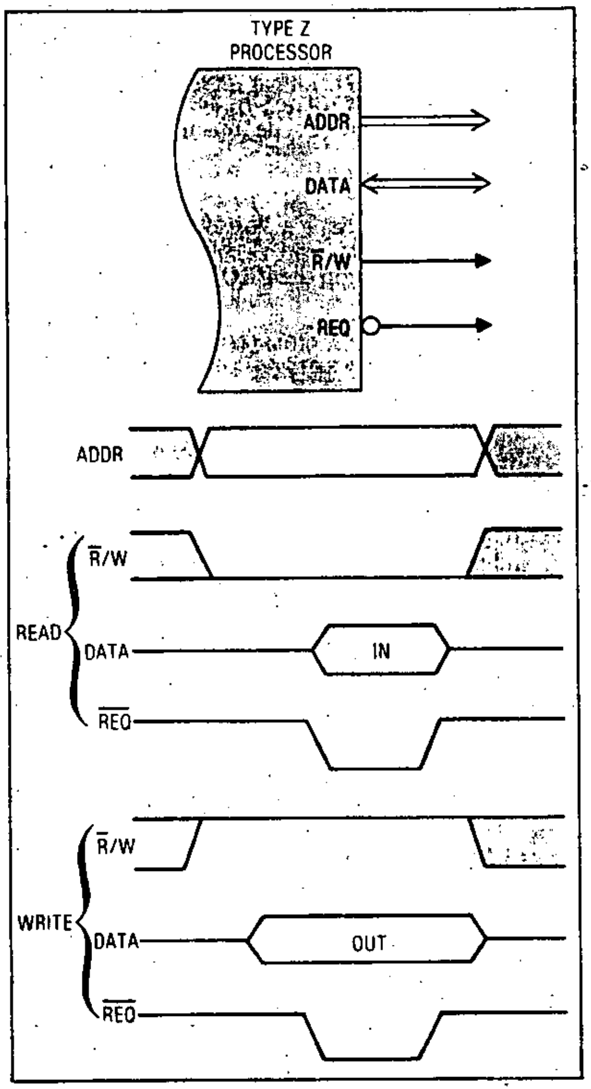


Figure 6. Type Z bus control signals.

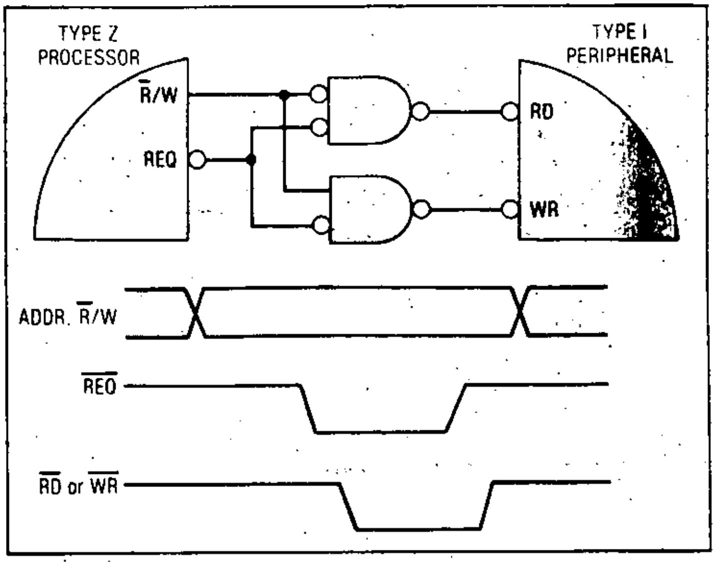


Figure 7. Acceptable Z-to-I interface.

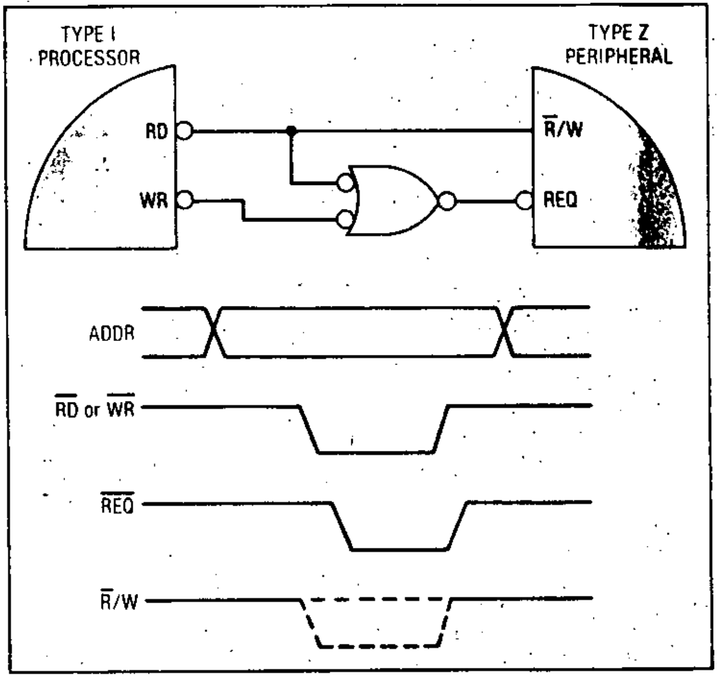


Figure 8. Unacceptable I-to-Z interface.

To use such macros, however, the programmer must give up some registers for use by the macros. In the above example, macros would use R0, leaving only R1 available to the program as a pointer variable. (Buffer copying and other routines requiring two or more pointers get to be a problem.)

The lack of symmetry in the instruction set also creates programming headaches. For example, why are there INC R0, DEC R0, and INC @R0 instructions, but not DEC @R0? Or, why can we conditionally jump on C, Z, T0, and T1 conditions true or false, but on F0, F1, TF, and accumulator bits only true? Except for accumulator bits false, the proper "holes" exist in the instruction set; in fact, it probably took more logic to turn the instructions off than to let them work. I have been told that these "unimportant" instructions leave room for future enhancements, but any worthwhile architectural enhancements would require changes more sweeping than a few special-purpose opcodes.

While nice programs can be written for the MCS-48, they take more effort than those written for a "general-purpose" architecture. Programming difficulty and expense also increase as we bump against the memory limits of the 8048. If we are writing one short (< 1K-byte) program and shipping 50,000 units with it, the programming expense is quite justifiable. If we are writing 10 different 2K- to 4K-byte programs, each of which will be shipped with 5000 units, we might be better off selecting a cleaner (albeit more expensive) machine.

## Some electrical characteristics

The MCS-48 uses Intel's reliable *n*-channel silicon-gate MOS process. Several second sources for the family have been announced—AMD, NEC, Signetics, Siemens, Intersil, RCA. Both Intersil and RCA have announced plans for CMOS versions of MCS-48 chips.

The MCS-48 chips use a single +5 volt supply and have logic input and output levels that are fully TTL compatible. As with all MOS microprocessors, the output drive is limited—typically four or five low-power Schottky (LSTTL) unit loads.

MCS-48 chips contain an oscillator to generate the processor clock (nominally 6 MHz) from an external crystal or RC circuit. It is also possible to connect an external clock directly to the oscillator input. The output of the oscillator feeds a divide-by-three counter whose output (nominally 2 MHz) controls the internal states of the processor. Since the divide-by-three counter cannot be synchronized externally, processor I/O cannot be referenced very well to the 6-MHz clock for tricky interfaces, nor can processors be run in lock-step from a common clock in a triplicated microcomputer (author's pet project).

**The MCS-48 family satisfied the usual Intel strategy of being the first in the marketplace with an imperfect but useful product.**

The MCS-48 chips have an active-low reset input pin connected to an internal high-impedance pullup resistor and Schmitt trigger. Thus, power-on reset can be accomplished by an external 1 mF capacitor. It is a little more difficult to add a logic-controlled reset (for example, by a watchdog timer), since open-collector or discrete transistor drive is required. And unless the driver circuit is sophisticated, a logic-commanded reset will disable the processor for a long time, due to the time constant of the power-on reset circuit—about 200 msec. In any case, reset destroys the state of the processor. It would be nice to have a nonmaskable interrupt (as in the 8085) that could be used for applications such as watchdog timers.

## Development tools

Intel supports two major development tools for the MCS-48 family—a cross assembler and an in-circuit emulator, ICE, both of which run on an Intel MDS microcomputer development system. Unfortunately, Intel does not support any MCS-48 assemblers or simulators that run on a large computing system, a necessity for any large development project. However, they can be obtained from independent software houses and consultants such as Microtec.

Intel MDS software for the MCS-48 lacks the consistency one expects from a good set of software tools. For example, there are at least three very different syntaxes that an engineer or programmer might use to change the value of a memory location in the MDS, depending on whether the monitor, ICE, or PROM programmer is being used. The monitor has a nice syntax that allows us to open a location, change it, and continue to the next location with a small number of keystrokes. In ICE, to read and change four locations, we must type (machine type underlined):

```
* CBYTE 144 TO 147
0144H=01H 3AH BFH 59H
* CBYTE 144=11,27,FF,6A
```

To do the same thing in the PROM programmer, the programmer types:

```
* DISPLAY FROM 144 TO 147
0090 01 3A BF 59
* CHANGE 144=11,27,FF,6A
```

In either syntax, one wastes keystrokes, and it's easy to lose track of the address in a long string. This isn't too terrible until we discover that ICE has interpreted and printed addresses and data in hex, while the PROM programmer has assumed inputs in decimal and printed outputs in hex (except for input FF, which the PROM programmer rejects because it looks like an assembler label!).

Another constant annoyance is that the MDS accepts only the RUB character for deleting characters (echoing the deleted character); backspace is not supported. It would have been easy enough to support both erase characters, making both teleprinter and CRT users happy.

## Conclusion

The MCS-48 microcomputer family was a reasonable contribution to the state of the art when it was introduced in 1976, in spite of its flaws. It achieved its design goals and satisfied the usual Intel strategy of being the first in the marketplace with an imperfect but useful product.

The MCS-48 is an acceptable choice for applications with initial estimated requirements of less than 1K bytes of program store, 64 bytes of data memory, and only one or two different programs to be developed. Designers whose applications require more memory or different programs in different chips should seek a more general-purpose architecture, such as the Z8 or 6801.

I have spent much of this article complaining that the MCS-48 is not a general-purpose 8-bit microcomputer. This may seem unfair, since some people at Intel claim the MCS-48 was never intended to go much beyond the old 4-bit market. So why criticize it on that basis? First, to help designers who might otherwise be tempted to use it in a larger application (Intel sales engineers frankly recommend their three-chip 8085 system in such cases). Second, to help designers who have already selected the MCS-48 for a larger application. Third, because discussion of general system requirements should benefit future system designers and chip designers alike. Finally, because Intel does not now offer a clean architecture suitable for the more general-purpose, single-chip, expandable microcomputer market, and I think they should. ■

## Acknowledgments

I appreciate the comments of my colleagues during the preparation of this article, especially those from Prem Jain, Paul Frantz, Ed Yarwood, and Dave Stearns. The comments of the reviewers were also very helpful. Thanks go to Dennis Allison for suggesting this article over a year ago, and to Bernard Peuto and Len Shustek for making me finish it. Of course, special thanks go to Intel for bringing us the MCS-48.

Some of this material was prepared while I was a lecturer at the Digital Systems Laboratory at Stanford University. Other material is adapted from my textbooks, *Microcomputers, Vol. 1: Architecture and Programming* and *Microcomputers, Vol. 2: System Hardware Design*, to be published by John Wiley & Sons in late 1979. All opinions expressed in this article are my own.

## References

1. Intel Corporation, *MCS-48 Family of Single Chip Microcomputers User's Manual*, Santa Clara, Calif., July 1978.
2. L. J. Shustek, "Analysis and Performance of Computer Instruction Sets," PhD dissertation, Stanford University, Jan. 1978, available from University Microfilms, Ann Arbor, Mich.
3. J. F. Wakerly, "Microcomputer Input/Output Architecture," *Computer*, Vol. 11, No. 2, Feb. 1977, pp. 26-33.
4. T. J. Chaney, S. M. Ornstein, and W. M. Littlefield, "Beware the Synchronizer," presented at COMPCON-72, IEEE Comput. Soc. Conf., San Francisco, Calif., Sept. 12-14, 1972.
5. T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE-TC* (Corresp.), Vol. C-22, No. 4, Apr. 1973, pp. 421-422.
6. M. Pechoucek, "Anomalous Response Times of Input Synchronizers," *IEEE-TC*, Vol. C-25, No. 2, Feb. 1976, pp. 133-139.

**John F. Wakerly** is an independent consultant and a lecturer at Stanford University, where from 1974 to 1976 he was an assistant professor of electrical engineering. He has published many technical articles in the areas of computer reliability, microprocessors, and digital systems education, and is the author of two textbooks, *Logic Design Projects Using Standard Integrated Circuits* (Wiley, 1976) and *Error-Detecting Codes, Self-Checking Circuits, and Applications* (Elsevier North-Holland, 1978).

Wakerly received the BEE degree from Marquette University in 1970 and the MSEE and PhD from Stanford University in 1971 and 1973, respectively. He was a Hertz Foundation Fellow during his studies at Stanford. He is a member of the IEEE Computer Society, ACM, Sigma Xi, Tau Beta Pi, and Eta Kappa Nu.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROCOMPUTADORES: TEORIA Y APLICACIONES

# SERIES 3000

# REFERENCE MANUAL

MARZO, 1979

# INTRODUCTION

### A family architecture

To reduce component count as far as practical, a multi-chip LSI microcomputer set must be designed as a complete, compatible family of devices. The omission of a bus or a latch or the lack of drive current can multiply the number of miscellaneous SSI and MSI packages to a dismaying extent—witness the reputedly LSI minicomputers now being offered which need over a hundred extra TTL packages on their processor boards to support one or two custom LSI devices. Successful integration should result in a minimum of extra packages, and that includes the interrupt and the input/output systems.

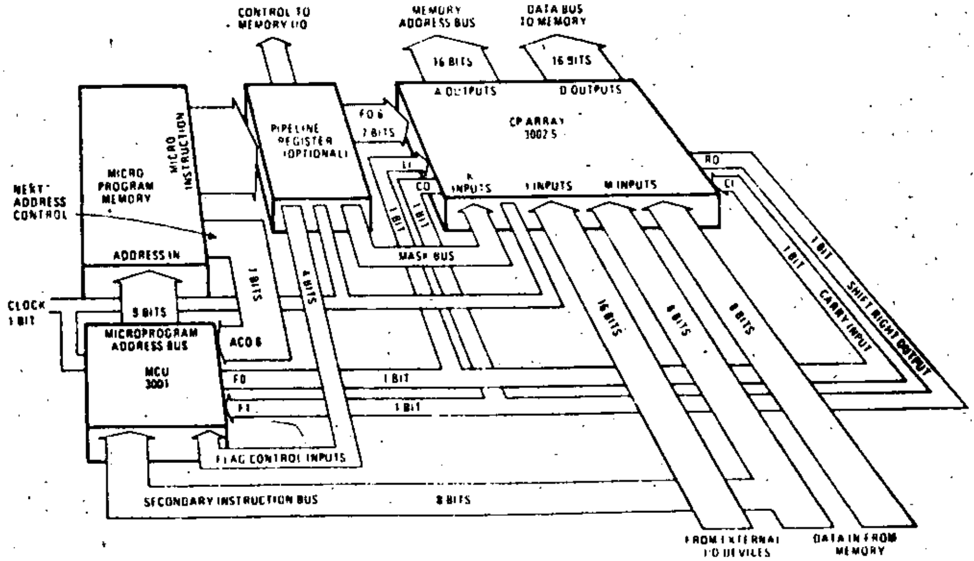
With this objective in mind, the Intel Schottky bipolar LSI microcomputer chip set was developed. Its two major components, the 3001 Microprogram Control Unit (MCU) and the 3002 Central Processing Element (CPE), may be combined by the digital designer with standard bipolar LSI memory to construct high-performance controller-processors (Fig. 1) with a minimum of ancillary logic.

Among the features that minimize package count and improve performance are: the multiple independent data and address busses that eliminate time multiplexing and the need for external latches; the three-state output buffers with high fanout that make bus drivers unnecessary except in the largest systems, and the separate output-enable logic that permits bidirectional busses to be formed simply by connecting inputs and outputs together.

Each CPE represents a complete two-bit slice through the data-processing section of a computer. Several CPEs may be arrayed in parallel to form a processor of any desired word length. The MCU, which together with the microprogram memory, controls the step-by-step operation of the processor, is itself a powerful micro-programed state sequencer.

Enhancing the performance and capabilities of these two components are a number of compatible computing elements. These include a fast look-ahead carry generator, a priority interrupt unit, and a multimode latch buffer. A complete summary of the first available members of this family of LSI computing elements and memories is given in the table on this page.

| | |
|---|---|
| 3001 | Microprogram control unit |
| 3002 | Central processing element |
| 3003 | Look-ahead carry generator |
| 3212 | Multimode latch buffer |
| 3214 | Priority interrupt unit |
| 3216 | Noninverting bidirectional bus driver |
| 3226 | Inverting bidirectional bus driver |
| 3601 | 256-by-4-bit programable read-only memory |
| 3604 | 512-by-8-bit programable read-only memory |
| 3301A | 256-by-4-bit read-only memory |
| 3304A | 512-by-8-bit read-only memory |



1. **Bipolar microcomputer.** Block diagram shows how to implement a typical 16-bit controller-processor with new family of bipolar computer elements. An array of eight central processing elements (CPEs) is governed by a microprogram control unit (MCU) through a separate read-only memory that carries the microinstructions for the various processing elements. This ROM may be a fast, off-the-shelf unit.

### CPEs form a processor

Each CPE (Fig. 2) carries two bits of five independent busses. The three input busses can be used in several different ways. Typically, the K-bus is used for microprogram mask or literal (constant) value input, while the other two input busses, M and I, carry data from external memory or input/output devices. D-bus outputs are connected to the CPE accumulator; A-bus outputs are connected to the CPE memory address register. As the CPEs are wired together, all the data paths, registers, and busses expand accordingly.

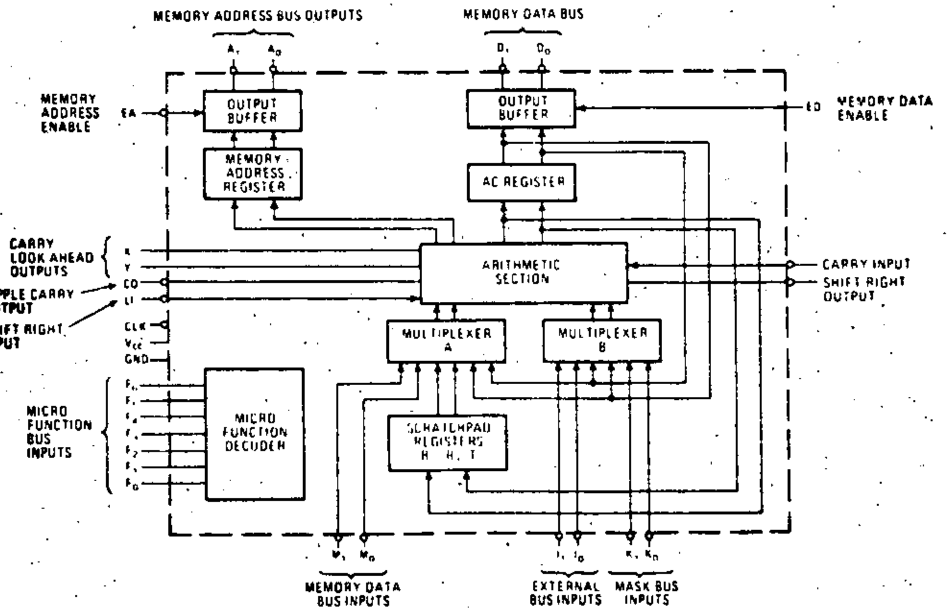
Certain data operations can be performed simply by connecting the busses in a particular fashion. For example, a byte exchange operation, often used in data-communications processors, may be carried out by wiring the D-bus outputs back to the I-bus inputs, exchanging the high-order outputs and low-order inputs. Several other discretionary shifts and rotates can be accomplished in this manner.

A sixth CPE bus, the seven-line microfunction bus, controls the internal operation of the CPE by selecting the operands and the operation to be performed. The arithmetic function section, under control of the microfunction bus decoder, performs over 40 Boolean and binary functions, including 2's complement arithmetic and logical AND, OR, NOT, and exclusive-NOR. It increments, decrements, shifts left or right, and tests for zero.

Unlike earlier MSI arithmetic-logic units, which contain many functions that are rarely used, the microfunction decoder selects only useful CPE operations. Standard carry look-ahead outputs, X and Y, are generated by the CPE for use with available look-ahead devices or the Intel 3003 Look-ahead Carry Generator. Independent carry input, carry output, shift input, and shift output lines are also available.

What's more, since the K-bus inputs are always ANDed with the B-multiplexer outputs into the arithmetic function section, a number of useful functions that in conventional MSI ALUs would require several cycles are generated in a single CPE microcycle. The type of bit masking frequently done in computer control systems can be performed with the mask supplied to the K-bus directly from the microinstruction.

Placing the K-bus in either the all-one or all-zero state will, in most cases, select or deselect the accumulator in the operation, respectively. This toggling effect of the K-bus on the accumulator nearly doubles the CPE's repertoire of microfunctions. For instance, with the K-bus in the all-zero state, the data on the M-bus may be complemented and loaded into the CPE's accumulator. The same function selected with the K-bus in the all-one state will exclusive-NOR the data on the M-bus with the accumulator contents.



**2. Central processing element.** This element contains all the circuits representing a two-bit-wide slice through a small computer's central processor. To build a processor of word width N, all that's necessary is to connect an array of N/2 CPEs together.

### Three Innovations

The power and versatility of the CPE are increased by three rather novel techniques. The first of these is the use of the carry lines and logic during non-arithmetic operations for bit testing and zero detection. The carry circuits during these operations perform a word-wide logical OR (ORing adjacent bits) of a selected result from the arithmetic section. The value of the OR, called the carry OR, is passed along the carry lines to be ORed with the result of an identical operation taking place simultaneously in the adjacent higher-order CPE.

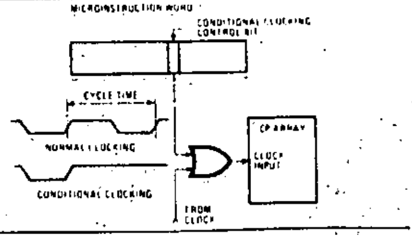
Obviously, the presence of at least one bit in the logical 1 state will result in a true carry output from the highest-order CPE. This output, as explained later, can be used by the MCU to determine which microprogram sequence to follow. With the ability to mask any desired bit, or set of bits, via the K-bus inputs included in the carry OR, a powerful bit-testing and zero-detection facility is realized.

The second novel CPE feature is the use of three-state outputs on the shift right output (RO) and carry output (CO) lines. During a right shift operation, the CO line is placed in the high-impedance (Z) state, and the shift data is active on the RO line. In all other CPE operations, the RO line is placed in the Z state, and the carry data is active on the CO line. This permits the CO and RO lines to be tied together and sent as a single rail input to the MCU for testing and branching. Left shift operations utilize the carry lines, rather than the shift lines, to propagate data.

The third novel CPE capability, called conditional clocking, saves microcode and microcycles by reducing the number of microinstructions required to perform a given test. One extra bit is used in the microinstruction to selectively control the gating of the clock pulse to the central processor (CP) array. Momentarily freezing the clock (Fig. 3) permits the CPE microfunction to be performed, but stops the results from being clocked into the specified registers. The carry or shift data that results from the operation is available because the arithmetic section is combinatorial, rather than sequential. The data can be used as a jump condition by the MCU and in this way permits a variety of nondestructive tests to be performed on register data.

### Microprogram control

The classic form of microprogram control incorporates a next-address field in each microinstruction—any other approach would require some type of program counter. To simplify its logic, the MCU (Fig. 4) uses the classic approach and requires address control information from each microinstruction. This information is not, however, simply the next microprogram address. Rather, it is a highly encoded specification of the next address and one of a set of conditional tests on the MCU bus inputs and registers.



**3. Conditional clock.** This feature permits an extra bit in microinstruction to selectively control gating of clock pulse to CP array. Carry or shift data thus made available permits tests to be performed on data with fewer microinstructions.

The next-address logic and address control functions of the MCU are based on a unique scheme of memory addressing. Microprogram addresses are organized as a two-dimensional array or matrix. Unlike in ordinary memory, which has linearly sequenced addresses, each microinstruction is pinpointed by its row and column address in the matrix. The 9-bit microprogram address specifies the row address in the upper 5 bits and the column address in the lower 4 bits. The matrix can therefore contain up to 32 row addresses and 16 column addresses for a total of 512 microinstruction addresses.

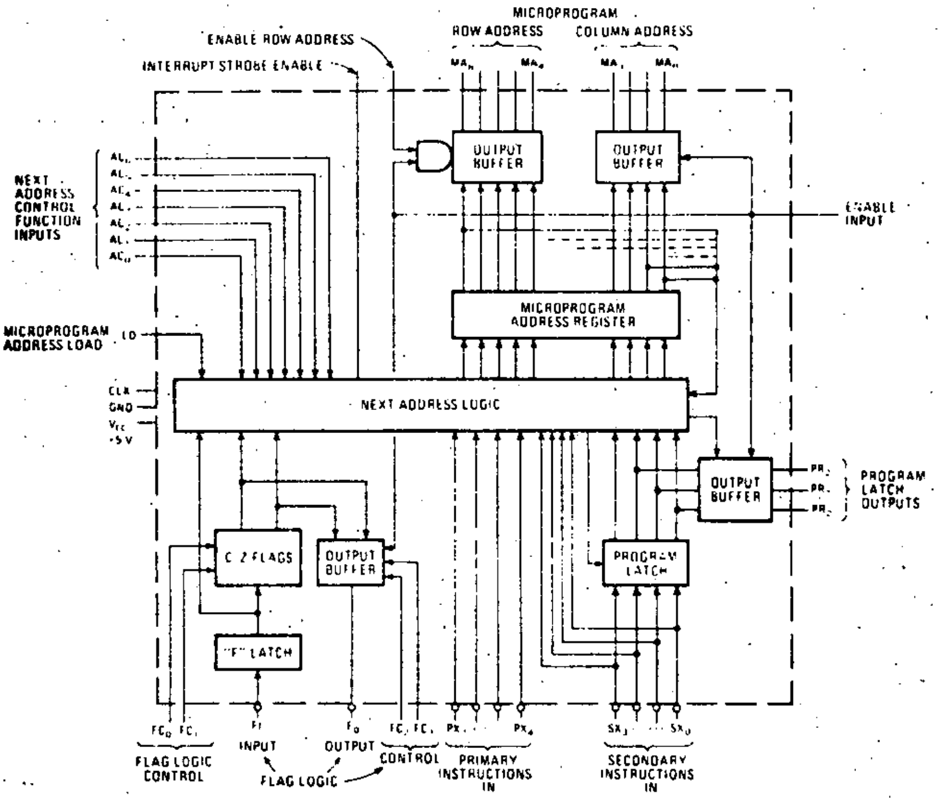
The next-address logic of the MCU makes extensive use of this addressing scheme. For example, from a particular row or column address, it is possible to jump either unconditionally to any other location in that row or column or conditionally to other specified locations, all in one operation. For a given location in the matrix there is a fixed subset of microprogram addresses that may be selected as the next address. These are referred to as a jump set, and each type of MCU address control jump function has a jump set associated with it.

Incorporating a jump operation in every microinstruction improves performance by allowing processing functions to be executed in parallel with program branches. Reductions in microcode are also obtained because common microprogram sequences can be shared without the time-space penalty usually incurred by conditional branching.

Independently controlled flag logic in the MCU is available for latching and controlling the value of the carry and shift inputs to the CP array. Two flags, called C and Z, are used to save the state of the flag input line. Under microprogram control, the flag logic simultaneously sets the state of the flag output line, forcing the line to logical 0, logical 1, or the value of the C or Z flag.

The jump decisions are made by the next-address logic on the basis of: the MCU's current microprogram address; the address control function on the accumulator inputs; and the data that's on the macroinstruction (X) bus or in the program latch or in the flags. Jump decisions may also be based on the instantaneous state of the flag input line without loading the value in one of the flags. This feature eliminates many extra microinstructions that would be required if only the flag flip-flop could be tested.

Microinstruction sequences are normally selected by the operation codes (op codes) supplied by the microinstructions, such as control commands or user instructions in main memory. The MCU decodes these commands by using their bit patterns to determine which is to be the next microprogram address. Each decoding results in a 16-way program branch to the desired microinstruction sequence.



4. **Microprogram control unit.** The MCU's two major control functions include controlling the sequence of microprograms fetched from the microprogram memory, and keeping track of the carry inputs and outputs of the CP array by means of the flag logic control.

### Cracking the op codes

For instance, the MCU can be microprogramed to directly decode conventional 8-bit op codes. In these op codes the upper 4 bits specify one of up to 16 instruction classes or address modes, such as register, indirect, or indexed. The remaining bits specify the particular subclass such as ADD, SKIP IF ZERO, and so on. If a set of op codes is required to be in a different format, as may occur in a full emulation, an external pre-decoder, such as ROM, can be used in series with the X-bus to reformat the data for the MCU.

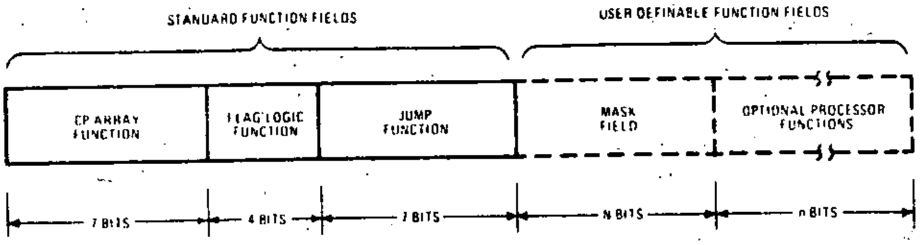
In rigorous decoding situations where speed or space is critical, the full 8-bit macroinstruction bus can be used for a single 256-way branch. Pulling down the load line of the MCU forces the 8 bits of data on the X-bus (typically generated by a predecoder) directly into the microprogram address register.

The data thus directly determines the next microprogram address which should be the start of the desired microprogram sequence. The load line may also be used by external logic to force the MCU, at power-up, into the system re-initialization sequence.

From time to time, a microprocessor must examine the state of its interrupt system to determine whether an interrupt is pending. If one is, the processor must suspend its normal execution sequence and enter an interrupt sequence in the microprogram. This requirement is handled by the MCU in a simple but elegant manner.

When the microprogram flows through address row 0 and column 15, the interrupt strobe enable line of the MCU is raised. The interrupt system, an Intel 3214 Interrupt Control Unit, responds by disabling the row address outputs of the MCU via the enable row address line, and by forcing the row entry address of the microprogram interrupt sequence onto the row address bus. The operation is normally performed just before the macroinstruction fetch cycle, so that a macroprogram is interrupted between, not during, macroinstructions.

The 9-bit microprogram address register and address bus of the MCU directly address 512 microinstructions. This is about twice as many as required by the typical 16-bit disk-controller or central processor.



**5. Microinstruction format.** Only a generalized microinstruction format can be shown since allocation of bits for the mask field and optional processor functions depends on the wishes of the designer and the tradeoffs he decides to make.

Moreover, multiple 512 microinstruction memory planes can easily be implemented simply by adding an extra address bit to the microinstruction each time the number of extra planes is doubled. Incidentally, as the number of bits in the microinstruction is increased, speed is not reduced. The additional planes also permit program jumps to take place in three address dimensions instead of two.

Because of the tremendous design flexibility offered by the Intel computing elements, it is impossible to describe every microinstruction format exactly. But generally speaking, the formats all derive from the one in Fig. 5. The minimum width is 18 bits: 7 bits for the address control functions, plus 4 bits for the flag logic control; plus 7 bits for the CPE microfunction control.

More bits can be added to the microinstruction format to provide such functions as mask field input to the CP array, external memory control, conditional clocking, and so on. Allocation of these bits is left to the designer who organizes the system. He is free to trade off memory costs, support logic, and microinstruction cycles to meet his cost/performance objectives.

### Microprograming technology

- **Microprogram:** A type of program that directly controls the operation of each functional element in a microprocessor.
- **Microinstruction:** A bit pattern that is stored in a microprogram memory word and specifies the operation of the individual LSI computing elements and related subunits, such as main memory and input/output interfaces.
- **Microinstruction sequence:** The series of microinstructions that the microprogram control unit (MCU) selects from the microprogram to execute a single macroinstruction or control command. Microinstruction sequences can be shared by several macroinstructions.
- **Macroinstruction:** Either a conventional computer instruction (e.g. ADD MEMORY TO REGISTER, INCREMENT, and SKIP, etc.) or device controller command (e.g., SEEK, READ, etc.).

### The cost/performance spectrum

The total flexibility of the Intel LSI computing elements is demonstrated by the broad cost/performance spectrum of the controllers and processors that can be constructed with them. These include:

- High-speed controllers, built with a stand-alone ROM-MCU combination that sequences at up to 10 megahertz; it can be used without any CPEs as a system state controller.
- Pipelined look-ahead carry controller-processors, where the overlapped microinstruction fetch/execute cycles and fast-carry logic reduce the 16-bit add time to less than 125 nanoseconds.
- Ripple-carry controller processors (a 16-bit design adds the contents of two registers in 300 nanoseconds).
- Multiprocessors, or networks of any of the above controllers and processors, to provide computation, interrupt supervision, and peripheral control.

These configurations represent a range of microinstruction execution rates of from 3 million to 10 million instructions per second, or up to two orders of magnitude faster, for example, than p-channel microprocessors. Moreover, the increases in processor performance are achieved with relative simplicity. A ripple-carry 16-bit processor uses one MCU, eight CPEs, plus microprogram memory. One extra computing element, the 3003 Look-ahead Carry Generator, enhances the processor with fast carry. Increasing speed further by pipelining, the overlap of microinstruction fetch and execute cycles, requires a few D-type MSI flip-flops.

At the multiprocessor level, the microprogram memory, MCU, or CPE devices can be shared. A 16-bit processor, complete with bus control and microprogram memory, requires some 20 bipolar LSI packages and half that many small-scale ICs. In this configuration, it replaces an equivalent MSI TTL system having more than 200 packages.

Furthermore, systems built with this large-scale integrated circuitry are much smaller and less costly and consume less energy than equivalent designs using lower levels of transistor-transistor-logic integration. Even allowing for ancillary logic circuits, the new bipolar computing elements cut 60% to 80% off the package count in realizing most of today's designs made with small- or medium-scale-integrated TTL.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# 8080 MICROCOMPUTER SYSTEM USER MANUAL

MARZO, 1979

# CHAPTER 2
# THE 8080 CENTRAL PROCESSOR UNIT

The 8080 is a complete 8-bit parallel, central processor unit (CPU) for use in general purpose digital computer systems. It is fabricated on a single LSI chip (see Figure 2-1). using Intel's n-channel silicon gate MOS process. The 8080 transfers data and internal state information via an 8-bit, bidirectional 3-state Data Bus ($D_0$-$D_7$). Memory and peripheral device addresses are transmitted over a separate 16-bit 3-state Address Bus ($A_0$-$A_{15}$). Six timing and control outputs (SYNC, DBIN, WAIT, $\overline{\text{WR}}$, HLDA and INTE) emanate from the 8080, while four control inputs (READY, HOLD, INT and RESET), four power inputs (+12v, +5v, -5v, and GND) and two clock inputs ($\phi_1$ and $\phi_2$) are accepted by the 8080.

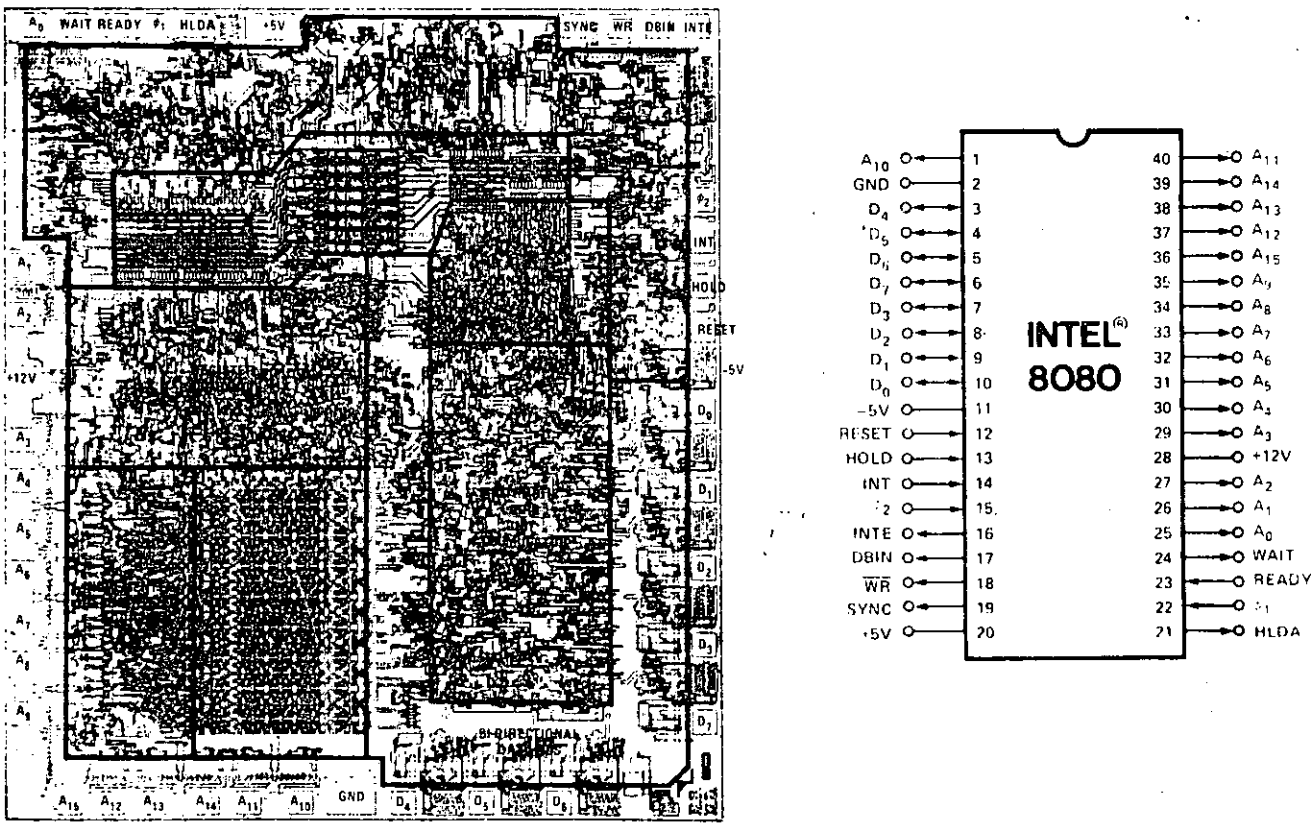


**Figure 2-1. 8080 Photomicrograph With Pin Designations**

## ARCHITECTURE OF THE 8080 CPU

The 8080 CPU consists of the following functional units:

- Register array and address logic
- Arithmetic and logic unit (ALU)
- Instruction register and control section
- Bi-directional, 3-state data bus buffer

Figure 2-2 illustrates the functional blocks within the 8080 CPU.

### Registers:

The register section consists of a static RAM array organized into six 16-bit registers:

- Program counter (PC)
- Stack pointer (SP)
- Six 8-bit general purpose registers arranged in pairs, referred to as B,C; D,E; and H,L
- A temporary register pair called W,Z

The program counter maintains the memory address of the current program instruction and is incremented automatically during every instruction fetch. The stack pointer maintains the address of the next available stack location in memory. The stack pointer can be initialized to use any portion of read-write memory as a stack. The stack pointer is decremented when data is "pushed" onto the stack and incremented when data is "popped" off the stack (i.e., the stack grows "downward").

The six general purpose registers can be used either as single registers (8-bit) or as register pairs (16-bit). The temporary register pair, W,Z, is not program addressable and is only used for the internal execution of instructions.

Eight-bit data bytes can be transferred between the internal bus and the register array via the register-select multiplexer. Sixteen-bit transfers can proceed between the register array and the address latch or the incrementer/decrementer circuit. The address latch receives data from any of the three register pairs and drives the 16 address output buffers ($A_0$-$A_{15}$), as well as the incrementer/decrementer circuit. The incrementer/decrementer circuit receives data from the address latch and sends it to the register array. The 16-bit data can be incremented or decremented or simply transferred between registers.

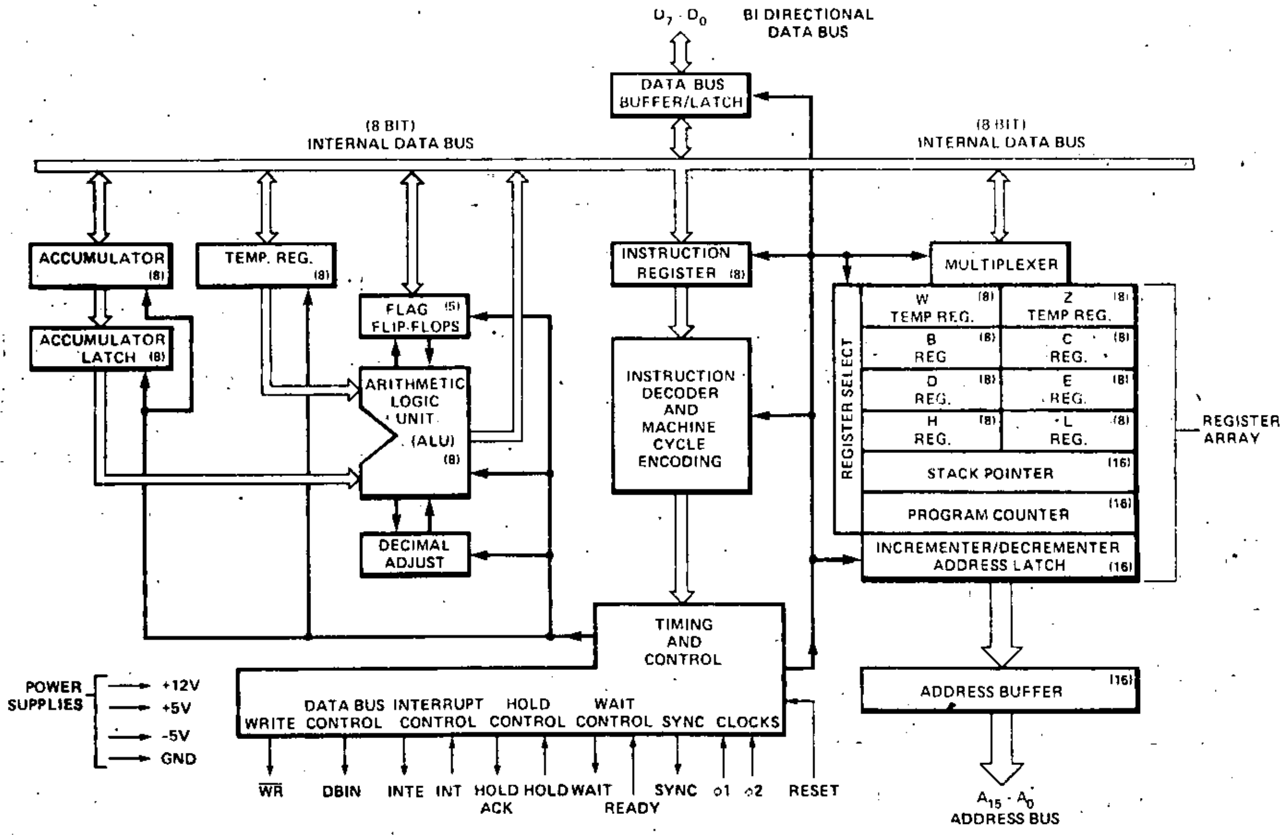


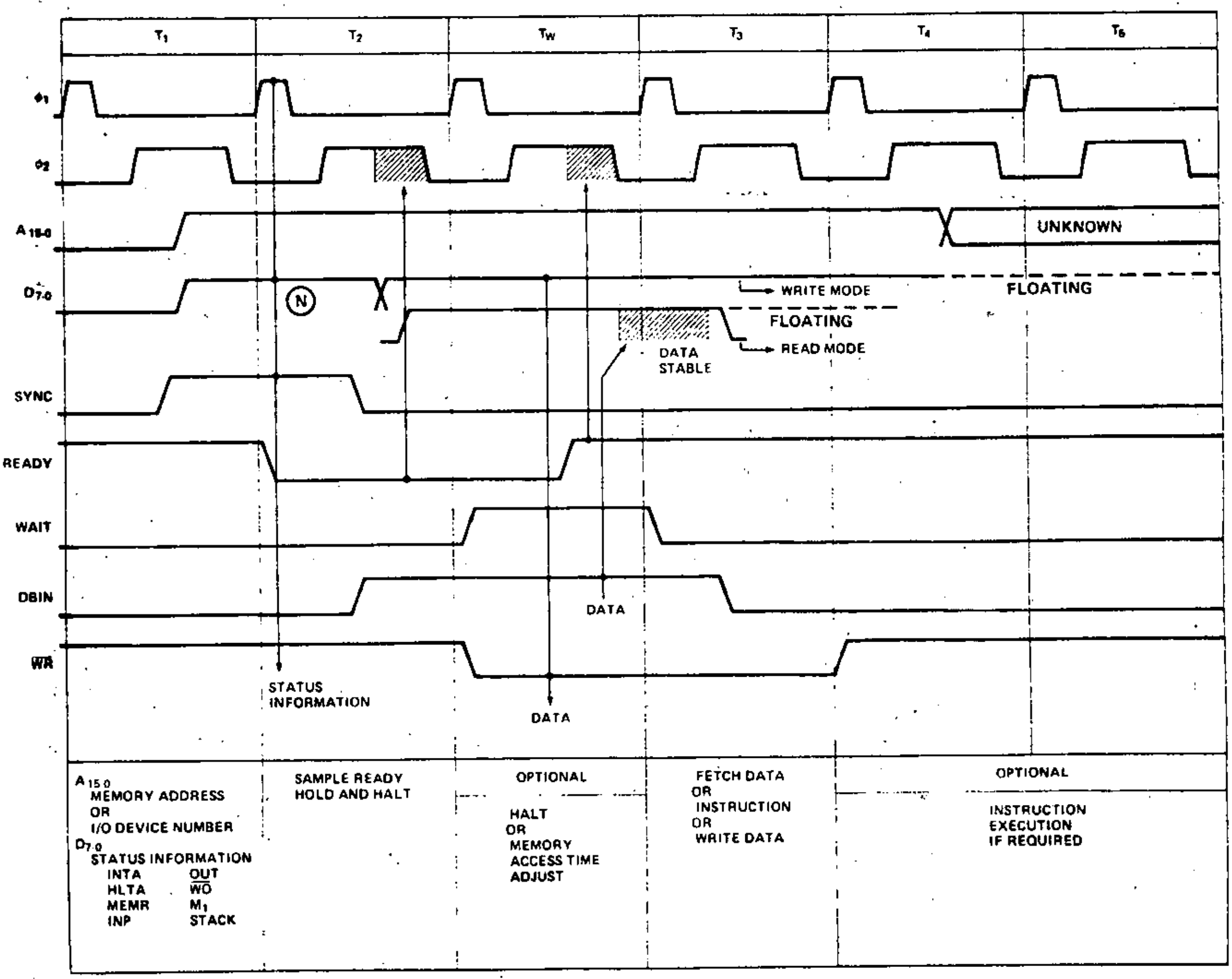
**Figure 2-2. 8080 CPU Functional Block Diagram**

The events that take place during the $T_3$ state are determined by the kind of machine cycle in progress. In a FETCH machine cycle, the processor interprets the data on its data bus as an instruction. During a MEMORY READ or a STACK READ, data on this bus is interpreted as a data word. The processor outputs data on this bus during a MEMORY WRITE machine cycle. During I/O operations, the processor may either transmit or receive data, depending on whether an OUTPUT or an INPUT operation is involved.

Figure 2-6 illustrates the timing that is characteristic of a data input operation. As shown, the low-to-high transition of $\phi_2$ during $T_2$ clears status information from the processor's data lines, preparing these lines for the receipt of incoming data. The data presented to the processor must have stabilized prior to both the "$\phi_1$–data set-up" interval ($t_{DS1}$), that precedes the falling edge of the $\phi_1$ pulse defining state $T_3$, and the "$\phi_2$–data set-up" interval ($t_{DS2}$), that precedes the rising edge of $\phi_2$ in state $T_3$. This same data must remain stable during the "data hold" interval ($t_{DH}$) that occurs following the rising edge of the $\phi_2$ pulse. Data placed on these lines by memory or by other external devices will be sampled during $T_3$.

During the input of data to the processor, the 8080 generates a DBIN signal which should be used externally to enable the transfer. Machine cycles in which DBIN is available include: FETCH, MEMORY READ, STACK READ, and INTERRUPT. DBIN is initiated by the rising edge of $\phi_2$ during state T2 and terminated by the corresponding edge of $\phi_2$ during $T_3$. Any $T_W$ phases intervening between $T_2$ and $T_3$ will therefore extend DBIN by one or more clock periods.

Figure 2-7 shows the timing of a machine cycle in which the processor outputs data. Output data may be destined either for memory or for peripherals. The rising edge of $\phi_2$ within state $T_2$ clears status information from the CPU's data lines, and loads in the data which is to be output to external devices. This substitution takes place within the



NOTE: Ⓝ Refer to Status Word Chart on Page 2-6.

**Figure 2-5. Basic 8080 Instruction Cycle**

"data output delay" interval ($t_{DD}$) following the $\phi_2$ clock's leading edge. Data on the bus remains stable throughout the remainder of the machine cycle, until replaced by updated status information in the subsequent $T_1$ state. Observe that a READY signal is necessary for completion of an OUTPUT machine cycle. Unless such an indication is present, the processor enters the $T_W$ state, following the $T_2$ state. Data on the output lines remains stable in the interim, and the processing cycle will not proceed until the READY line again goes high.

The 8080 CPU generates a $\overline{WR}$ output for the synchronization of external transfers, during those machine cycles in which the processor outputs data. These include MEMORY WRITE, STACK WRITE, and OUTPUT. The negative-going leading edge of $\overline{WR}$ is referenced to the rising edge of the first $\phi_1$ clock pulse following $T_2$, and occurs within a brief delay ($t_{DC}$) of that event. $\overline{WR}$ remains low until re-triggered by the leading edge of $\phi_1$ during the state following $T_3$. Note that any $T_W$ states intervening between $T_2$ and $T_3$ of the output machine cycle will necessarily extend $\overline{WR}$, in much the same way that DBIN is affected during data input operations.

All processor machine cycles consist of at least three states: $T_1$, $T_2$, and $T_3$ as just described. If the processor has to wait for a response from the peripheral or memory with which it is communicating, then the machine cycle may also contain one or more $T_W$ states. During the three basic states, data is transferred to or from the processor.

After the $T_3$ state, however, it becomes difficult to generalize. $T_4$ and $T_5$ states are available, if the execution of a particular instruction requires them. But not all machine cycles make use of these states. It depends upon the kind of instruction being executed, and on the particular machine cycle within the instruction cycle. The processor will terminate any machine cycle as soon as its processing activities are completed, rather than proceeding through the $T_4$ and $T_5$ states every time. Thus the 8080 may exit a machine cycle following the $T_3$, the $T_4$, or the $T_5$ state and proceed directly to the $T_1$ state of the next machine cycle.

| STATE | ASSOCIATED ACTIVITIES |
|---|---|
| $T_1$ | A memory address or I/O device number is placed on the Address Bus ($A_{15-0}$); status information is placed on Data Bus ($D_{7-0}$). |
| $T_2$ | The CPU samples the READY and HOLD inputs and checks for halt instruction. |
| TW<br>(optional) | Processor enters wait state if READY is low or if HALT instruction has been executed. |
| T3 | An instruction byte (FETCH machine cycle), data byte (MEMORY READ, STACK READ) or interrupt instruction (INTERRUPT machine cycle) is input to the CPU from the Data Bus; or a data byte (MEMORY WRITE, STACK WRITE or OUTPUT machine cycle) is output onto the data bus. |
| T4<br>T5<br>(optional) | States $T_4$ and $T_5$ are available if the execution of a particular instruction requires them; if not, the CPU may skip one or both of them. $T_4$ and $T_5$ are only used for internal processor operations. |

Table 2-2. State Definitions

## INSTRUCTION SET

The accumulator group instructions include arithmetic and logical operators with direct, indirect, and immediate addressing modes.

Move, load, and store instruction groups provide the ability to move either 8 or 16 bits of data between memory, the six working registers and the accumulator using direct, indirect, and immediate addressing modes.

The ability to branch to different portions of the program is provided with jump, jump conditional, and computed jumps. Also the ability to call to and return from subroutines is provided both conditionally and unconditionally. The RESTART (or single byte call instruction) is useful for interrupt vector operation.

Double precision operators such as stack manipulation and double add instructions extend both the arithmetic and interrupt handling capability of the 8080A. The ability to increment and decrement memory, the six general registers and the accumulator is provided as well as extended increment and decrement instructions to operate on the register pairs and stack pointer. Further capability is provided by the ability to rotate the accumulator left or right through or around the carry bit.

Input and output may be accomplished using memory addresses as I/O ports or the directly addressed I/O provided for in the 8080A instruction set.

The following special instruction group completes the 8080A instruction set: the NOP instruction, HALT to stop processor execution and the DAA instructions provide decimal arithmetic capability. STC allows the carry flag to be directly set, and the CMC instruction allows it to be complemented. CMA complements the contents of the accumulator and XCHG exchanges the contents of two 16-bit register pairs directly.

### Data and Instruction Formats

Data in the 8080A is stored in the form of 8-bit binary integers. All data transfers to the system data bus will be in the same format.

| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ |
|---|

DATA WORD

The program instructions may be one, two, or three bytes in length. Multiple byte instructions must be stored in successive words in program memory. The instruction formats then depend on the particular operation executed.

| | | TYPICAL INSTRUCTIONS |
|---|---|---|
| **One Byte Instructions** | | |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | OP CODE | Register to register, memory reference, arithmetic or logical, rotate, return, push, pop, enable or disable Interrupt instructions |
| **Two Byte Instructions** | | |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | OP CODE | |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | OPERAND | Immediate mode or I/O instructions |
| **Three Byte Instructions** | | |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | OP CODE | Jump, call or direct load and store instructions |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | LOW ADDRESS OR OPERAND 1 | |
| $D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ | HIGH ADDRESS OR OPERAND 2 | |

For the 8080A a logic "1" is defined as a high level and a logic "0" is defined as a low level.

# Schottky Bipolar 8212

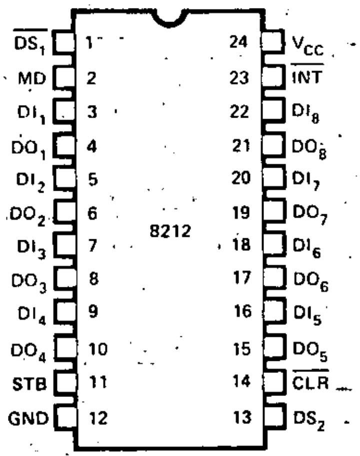
## EIGHT-BIT INPUT/OUTPUT PORT

- **Fully Parallel 8-Bit Data Register and Buffer**
- **Service Request Flip-Flop for Interrupt Generation**
- **Low Input Load Current — .25 mA Max.**
- **Three State Outputs**
- **Outputs Sink 15 mA**
- **3.65V Output High Voltage for Direct Interface to 8080 CPU or 8008 CPU**
- **Asynchronous Register Clear**
- **Replaces Buffers, Latches and Multiplexers in Microcomputer Systems**
- **Reduces System Package Count**

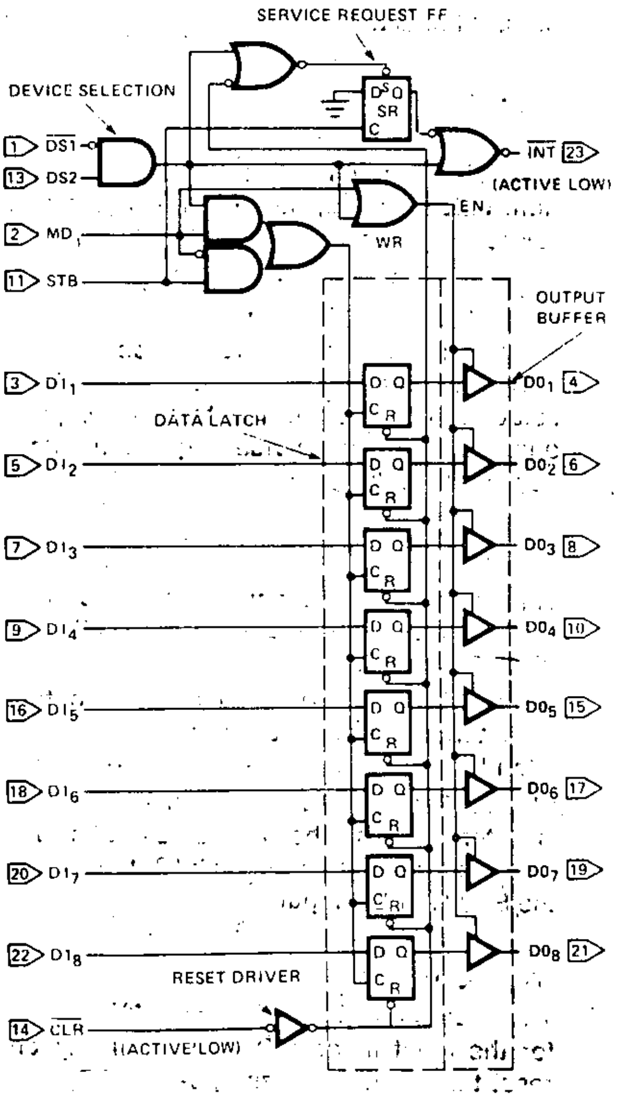
The 8212 input/output port consists of an 8-bit latch with 3-state output buffers along with control and device selection logic. Also included is a service request flip-flop for the generation and control of interrupts to the microprocessor.

The device is multimode in nature. It can be used to implement latches, gated buffers or multiplexers. Thus, all of the principal peripheral and input/output functions of a microcomputer system can be implemented with this device.

### PIN CONFIGURATION



### LOGIC DIAGRAM



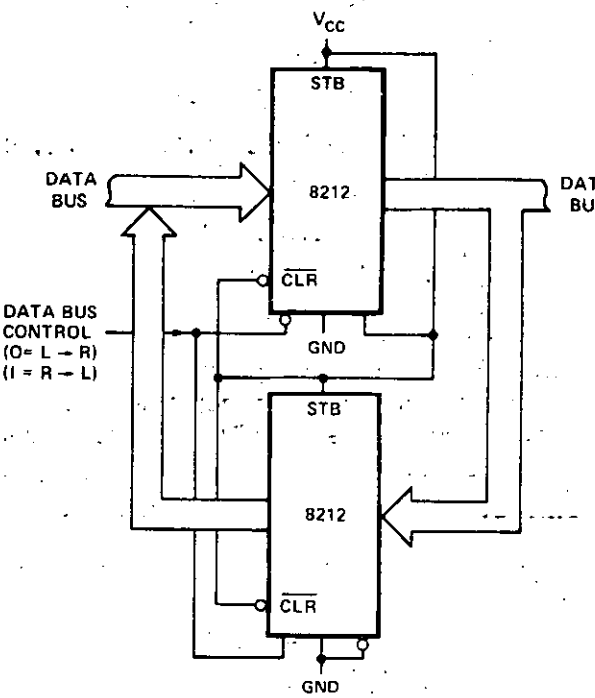
### PIN NAMES

| | |
|---|---|
| $DI_1$-$DI_8$ | DATA IN |
| $DO_1$-$DO_8$ | DATA OUT |
| $\overline{DS_1}$-$DS_2$ | DEVICE SELECT |
| MD | MODE |
| STB | STROBE |
| $\overline{INT}$ | INTERRUPT (ACTIVE LOW) |
| $\overline{CLR}$ | CLEAR (ACTIVE LOW) |

## III. Bi-Directional Bus Driver

A pair of 8212's wired (back-to-back) can be used as a symmetrical drive, bi-directional bus driver. The devices are controlled by the data bus input control which is connected to $\overline{DS1}$ on the first 8212 and to DS2 on the second. One device is active, and acting as a straight through buffer the other is in 3-state mode. This is a very useful circuit in small system design.

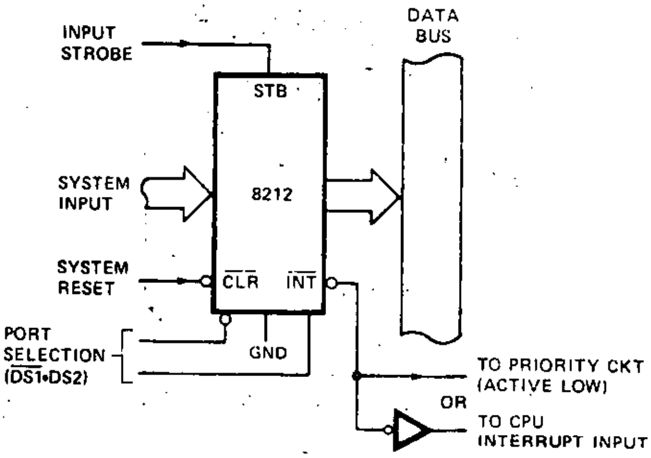
BI-DIRECTIONAL BUS DRIVER



## IV. Interrupting Input Port

This use of an 8212 is that of a system input port that accepts a strobe from the system input source, which in turn clears the service request flip-flop and interrupts the processor. The processor then goes through a service routine, identifies the port, and causes the device selection logic to go true — enabling the system input data onto the data bus.

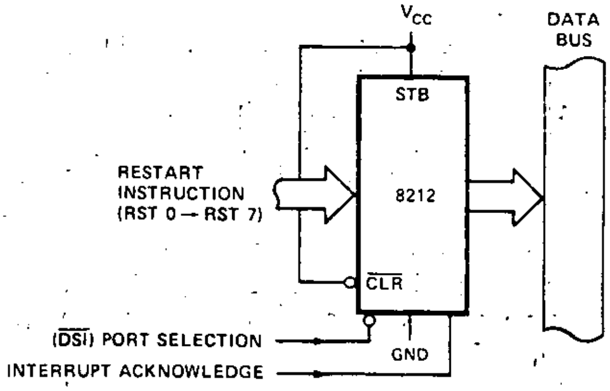
INTERRUPTING INPUT PORT



## V. Interrupt Instruction Port

The 8212 can be used to gate the interrupt instruction, normally RESTART instructions, onto the data bus. The device is enabled from the interrupt acknowledge signal from the microprocessor and from a port selection signal. This signal is normally tied to ground. ($\overline{DS1}$ could be used to multiplex a variety of interrupt instruction ports onto a common bus).

INTERRUPT INSTRUCTION PORT

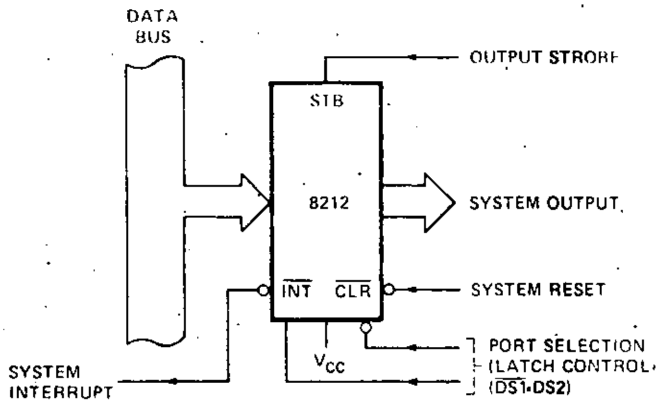


$V_{CC}$
DATA BUS
STB
RESTART INSTRUCTION (RST 0 — RST 7)
8212
$\overline{CLR}$
($\overline{DS1}$) PORT SELECTION
GND
INTERRUPT ACKNOWLEDGE

## VI. Output Port (With Hand-Shaking)

The 8212 can be used to transmit data from the data bus to a system output. The output strobe could be a hand-shaking signal such as "reception of data" from the device that the system is outputting to. It in turn, can interrupt the system signifying the reception of data. The selection of the port comes from the device selection logic. ($\overline{DS1}$ • DS2)

OUTPUT PORT (WITH HAND-SHAKING)



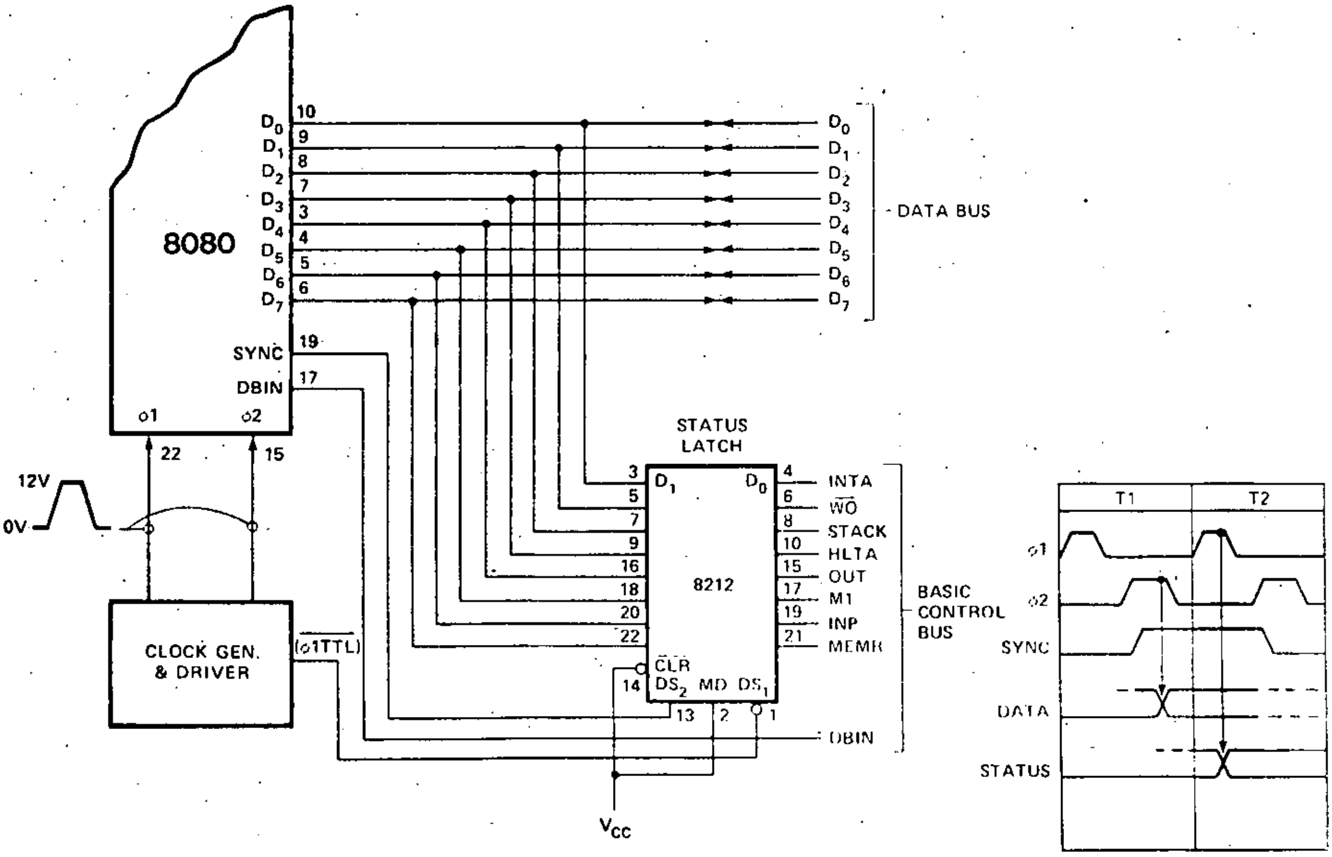
## VII. 8080 Status Latch

Here the 8212 is used as the status latch for an 8080 microcomputer system. The input to the 8212 latch is directly from the 8080 data bus. Timing shows that when the SYNC signal is true, which is connected to the DS2 input and the phase 1 signal is true, which is a TTL level coming from the clock generator; then, the status data will be latched into the 8212.

Note: The mode signal is tied high so that the output on the latch is active and enabled all the time.

It is shown that the two areas of concern are the bidirectional data bus of the microprocessor and the control bus.

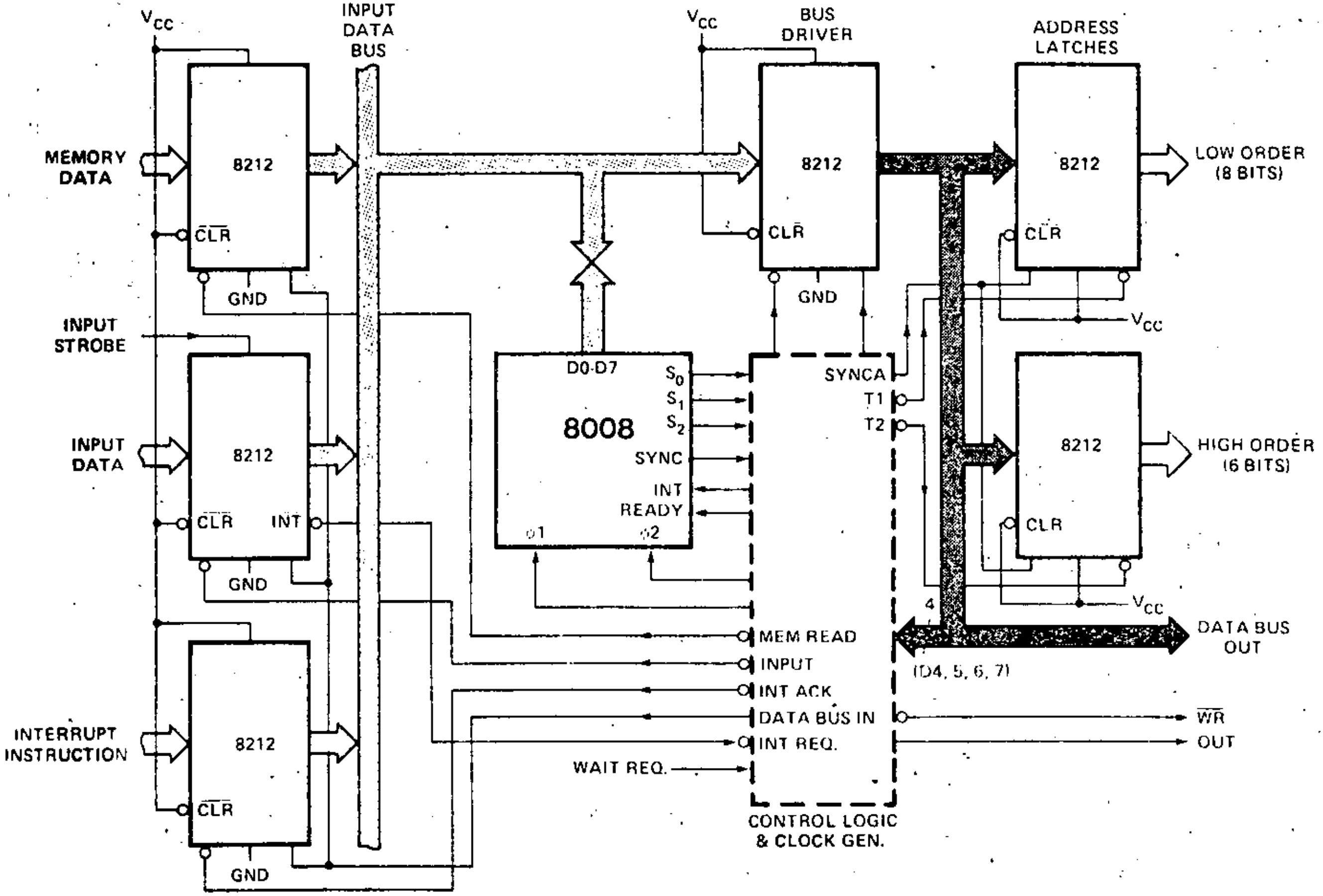
8080 STATUS LATCH



## VIII. 8008 System

This shows the 8212 used in an 8008 microcomputer system. They are used to multiplex the data from three different sources onto the 8008 input data bus. The three sources of data are: memory data, input data, and the interrupt instruction. The 8212 is also used as the uni-directional bus driver to provide a proper drive to the address latches (both low order and high order are also 8212's) and to provide adequate drive to the output data bus. The control of these six 8212's in the 8008 system is provided by the control logic and clock generator circuits. These circuits consist of flip-flops, decoders, and gates to generate the control functions necessary for 8008 microcomputer systems. Also note that the input data port has a strobe input. This allows the processor to be interrupted from the input port directly. The control of the input bus consists of the data bus input signal, control logic, and the appropriate status signal for bus discipline whether memory read, input, or interrupt acknowledge. The combination of these four signals determines which one of these three devices will have access to the input data bus. The bus driver, which is implemented in an 8212, is also controlled by the control logic and clock generator so it can be 3-stated when necessary and also as a control transmission device to the address latches. Note: The address latches can be 3-stated for DMA purposes and they provide 15 milli amps drive, sufficient for large bus systems.

8008 SYSTEM



## IX. 8080 System

This drawing shows the 8212 used in the I/O section of an 8080 microcomputer system. The system consists of 8 input ports, 8 output ports, 8 level priority systems, and a bidirectional bus driver. (The data bus within the system is darkened for emphasis). Basically, the operation would be as follows: The 8 ports, for example, could be connected to 8 keyboards, each keyboard having its own priority level. The keyboard could provide a strobe input of its own which would clear the service request flip-flop. The $\overline{\text{INT}}$ signals are connected to an 8 level priority encoding circuit. This circuit provides a positive true level to the central processor (INT) along with a three-bit code to the interrupt instruction port for the generation of RESTART instructions. Once the processor has been interrupted and it acknowledges the reception of the interrupt, the Interrupt Acknowledge signal is generated. This signal transfers data in the form of a RESTART instruction onto the buffered data bus. When the DBIN signal is true this RESTART instruction is gated into the microcomputer, in this case, the 8080 CPU. The 8080 then performs a software controlled interrupt service routine, saving the status of its current operation in the push-down stack and performing an INPUT instruction. The INPUT instruction thus sets the INP status bit, which is common to all input ports.

Also present is the address of the device on the 8080 address bus which in this system is connected to an 8205, one out of eight decoder with active low outputs. These active low outputs will enable one of the input ports, the one that interrupted the processor, to put its data onto the buffered data bus to be transmitted to the CPU when the data bus input signal is true. The processor can also output data from the 8080 data bus to the buffered data bus when the data bus input signal is false. Using the same address selection technique from the 8205 decoder and the output status bit, we can select with this system one of eight output ports to transmit the data to the system's output device structure.

Note: This basic I/O configuration for the 8080 can be expanded to 256 input devices and 256 output devices all using 8212 and, of course, the appropriate decoding.

Note that the 8080 is a 3.3-volt minimum high input requirement and that the 8212 has a 3.65-volt minimum high output providing the designer with a 350 milli volt noise margin worst case for 8080 systems when using the 8212.

# Silicon Gate MOS 8255

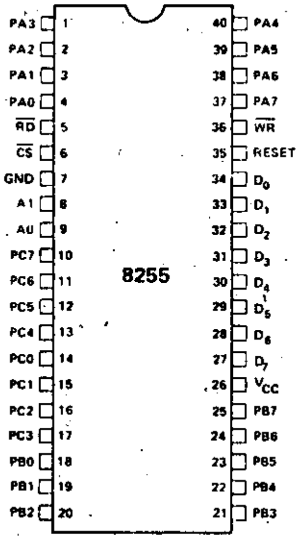
## PROGRAMMABLE PERIPHERAL INTERFACE

- **24 Programmable I/O Pins**
- **Completely TTL Compatible**
- **Fully Compatible with MCS™ -8 and MCS™ -80 Microprocessor Families**
- **Direct Bit Set/Reset Capability Easing Control Application Interface**
- **40 Pin Dual In-Line Package**
- **Reduces System Package Count**

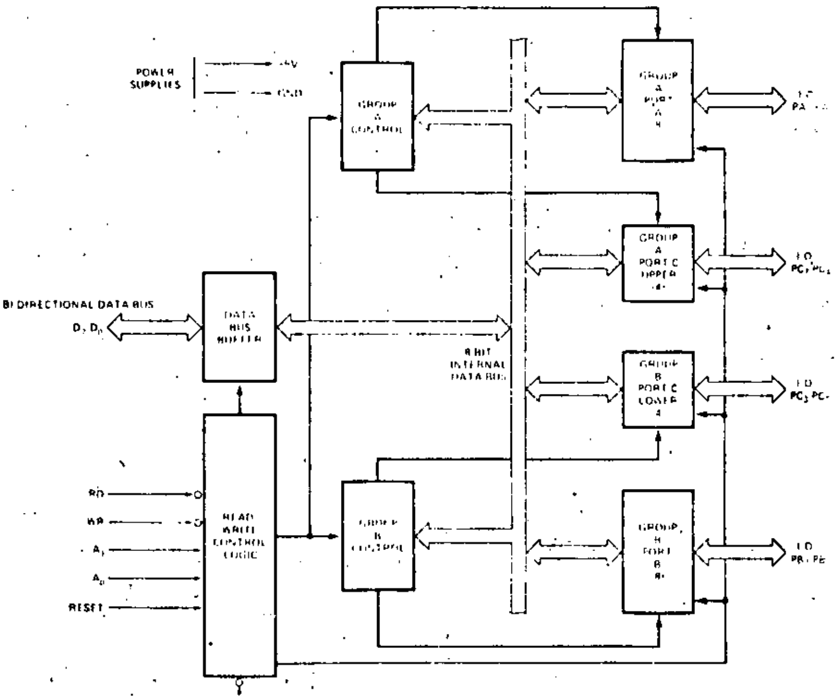
The 8255 is a general purpose programmable I/O device designed for use with both the 8008 and 8080 microprocessors. It has 24 I/O pins which may be individually programmed in two groups of twelve and used in three major modes of operation. In the first mode (Mode 0), each group of twelve I/O pins may be programmed in sets of 4 to be input or output. In Mode 1, the second mode, each group may be programmed to have 8 lines of input or output. Of the remaining four pins three are used for handshaking and interrupt control signals. The third mode of operation (Mode 2) is a Bidirectional Bus mode which uses 8 lines for a bidirectional bus, and five lines, borrowing one from the other group, for handshaking.

Other features of the 8255 include bit set and reset capability and the ability to source 1mA of current at 1.5 volts. This allows darlington transistors to be directly driven for applications such as printers and high voltage displays.

### PIN CONFIGURATION



### 8255 BLOCK DIAGRAM



### PIN NAMES

| | |
|---|---|
| $D_7$–$D_0$ | DATA BUS (BI-DIRECTIONAL) |
| RESET | RESET INPUT |
| $\overline{CS}$ | CHIP SELECT |
| $\overline{RD}$ | READ INPUT |
| $\overline{WR}$ | WRITE INPUT |
| A0, A1 | PORT ADDRESS |
| PA7-PA0 | PORT A (BIT) |
| PB7-PB0 | PORT B (BIT) |
| PC7-PC0 | PORT C (BIT) |
| Vcc | +5 VOLTS |
| GND | Ø VOLTS |

## 8255 BASIC FUNCTIONAL DESCRIPTION

### General

The 8255 is a Programmable Peripheral Interface (PPI) device designed for use in 8080 Microcomputer Systems. Its function is that of a general purpose I/O component to interface peripheral equipment to the 8080 system bus. The functional configuration of the 8255 is programmed by the system software so that normally no external logic is necessary to interface peripheral devices or structures.

### Data Bus Buffer

This 3-state, bi-directional, eight bit buffer is used to interface the 8255 to the 8080 system data bus. Data is transmitted or received by the buffer upon execution of INput or OUTput instructions by the 8080 CPU. Control Words and Status information are also transferred through the Data Bus buffer.

### Read/Write and Control Logic

The function of this block is to manage all of the internal and external transfers of both Data and Control or Status words. It accepts inputs from the 8080 CPU Address and Control busses and in turn, issues commands to both of the Control Groups.

### ($\overline{CS}$)

**Chip Select:** A "low" on this input pin enables the communication between the 8255 and the 8080 CPU.

### ($\overline{RD}$)

**Read:** A "low" on this input pin enables the 8255 to send the Data or Status information to the 8080 CPU on the Data Bus. In essence, it allows the 8080 CPU to "read from" the 8255.

### ($\overline{WR}$)

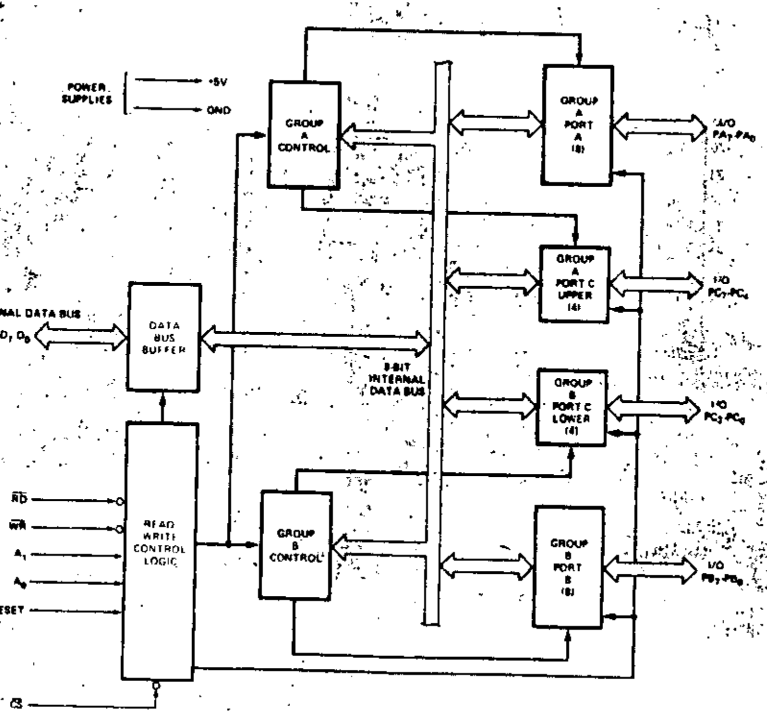
**Write:** A "low" on this input pin enables the 8080 CPU to write Data or Control words into the 8255.

### ($A_0$ and $A_1$)

**Port Select 0 and Port Select 1:** These input signals, in conjunction with the $\overline{RD}$ and $\overline{WR}$ inputs, control the selection of one of the three ports or the Control Word Register. They are normally connected to the least significant bits of the Address Bus ($A_0$ and $A_1$).

## 8255 BASIC OPERATION

| $A_1$ | $A_0$ | $\overline{RD}$ | $\overline{WR}$ | $\overline{CS}$ | INPUT OPERATION (READ) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | PORT A ⇒ DATA BUS |
| 0 | 1 | 0 | 1 | 0 | PORT B ⇒ DATA BUS |
| 1 | 0 | 0 | 1 | 0 | PORT C ⇒ DATA BUS |
| | | | | | **OUTPUT OPERATION (WRITE)** |
| 0 | 0 | 1 | 0 | 0 | DATA BUS ⇒ PORT A |
| 0 | 1 | 1 | 0 | 0 | DATA BUS ⇒ PORT B |
| 1 | 0 | 1 | 0 | 0 | DATA BUS ⇒ PORT C |
| 1 | 1 | 1 | 0 | 0 | DATA BUS ⇒ CONTROL |
| | | | | | **DISABLE FUNCTION** |
| X | X | X | X | 1 | DATA BUS ⇒ 3-STATE |
| 1 | 1 | 0 | 1 | 0 | ILLEGAL CONDITION |



8255 Block Diagram

## (RESET)

Reset: A "high" on this input clears all internal registers including the Control Register and all ports (A, B, C) are set to the input mode.

## Group A and Group B Controls

The functional configuration of each port is programmed by the systems software. In essence, the 8080 CPU "outputs" a control word to the 8255. The control word contains information such as "mode", "bit set", "bit reset" etc. that initializes the functional configuration of the 8255.

Each of the Control blocks (Group A and Group B) accepts "commands" from the Read/Write Control Logic, receives "control words" from the internal data bus and issues the proper commands to its associated ports.

Control Group A – Port A and Port C upper (C7-C4)
Control Group B – Port B and Port C lower (C3-C0)

The Control Word Register can Only be written into. No Read operation of the Control Word Register is allowed.

## Ports A, B, and C

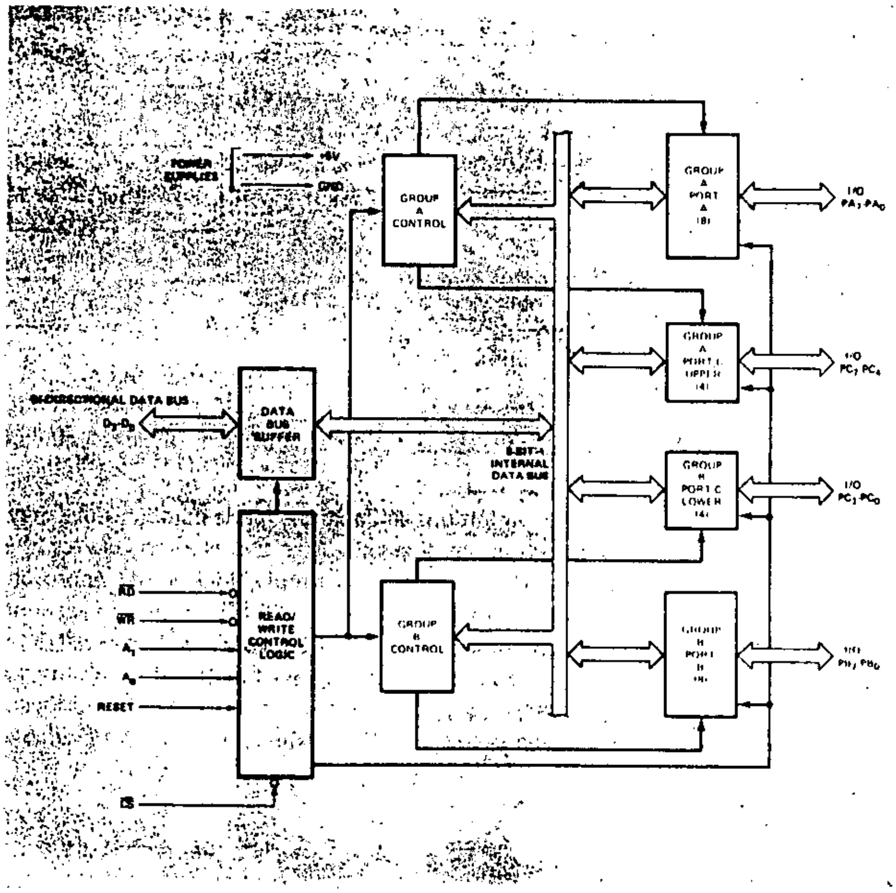
The 8255 contains three 8-bit ports (A, B, and C). All can be configured in a wide variety of functional characteristics by the system software but each has its own special features or "personality" to further enhance the power and flexibility of the 8255.

**Port A:** One 8-bit data output latch/buffer and one 8-bit data input latch.

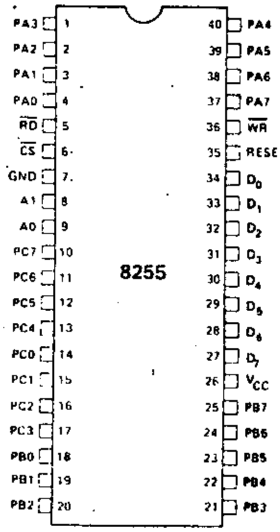
**Port B:** One 8-bit data input/output latch/buffer and one 8-bit data input buffer.

**Port C:** One 8-bit data output latch/buffer and one 8-bit data input buffer (no latch for input). This port can be divided into two 4-bit ports under the mode control. Each 4-bit port contains a 4-bit latch and it can be used for the control signal outputs and status signal inputs in conjunction with Ports A and B.

### 8255 BLOCK DIAGRAM



### PIN CONFIGURATION



### PIN NAMES

| | |
|---|---|
| $D_7$-$D_0$ | DATA BUS (BI-DIRECTIONAL) |
| RESET | RESET INPUT |
| CS | CHIP SELECT |
| RD | READ INPUT |
| WR | WRITE INPUT |
| A0, A1 | PORT ADDRESS |
| PA7-PA0 | PORT A (BIT) |
| PB7-PB0 | PORT B (BIT) |
| PC7-PC0 | PORT C (BIT) |
| $V_{CC}$ | +5 VOLTS |
| GND | 0 VOLTS |

## 8255 DETAILED OPERATIONAL DESCRIPTION

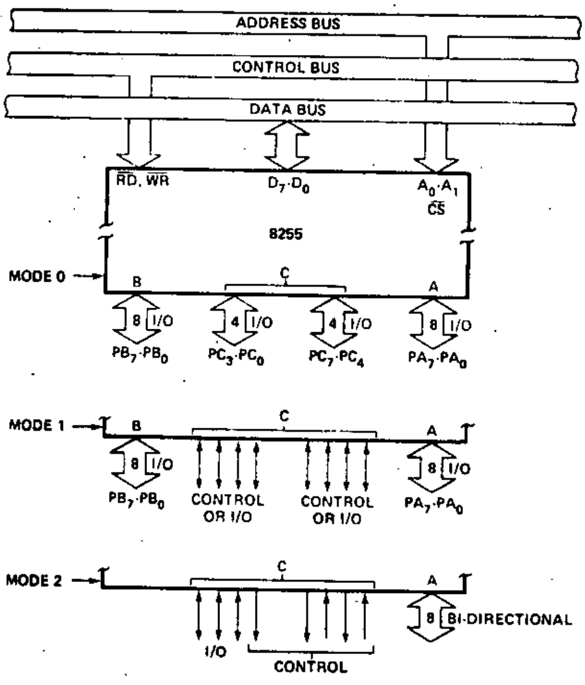
### Mode Selection

There are three basic modes of operation that can be selected by the system software:

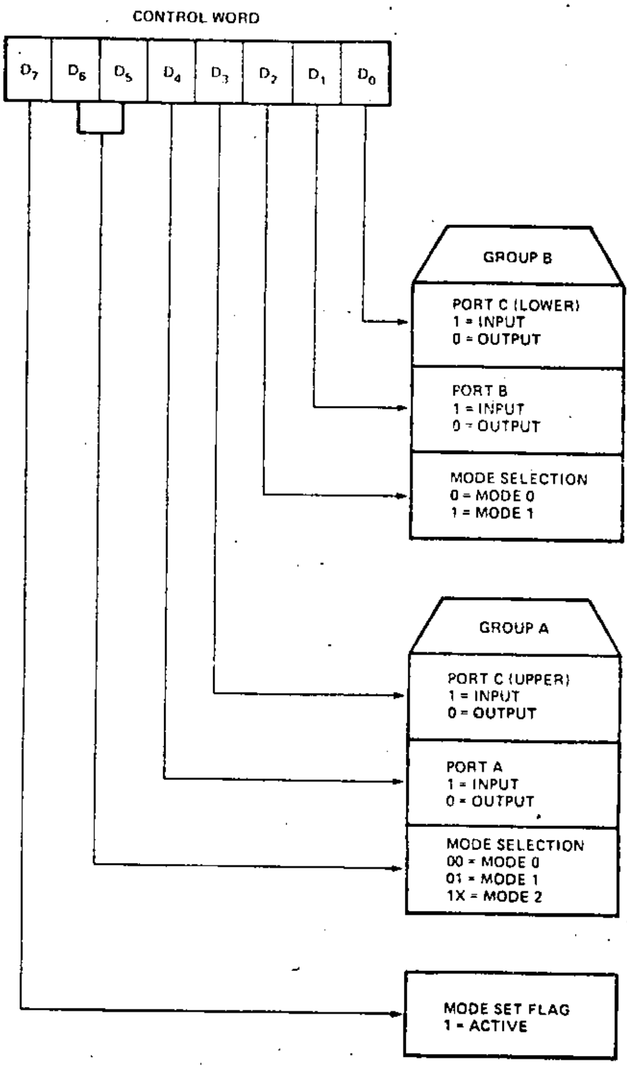
Mode 0 — Basic Input/Output
Mode 1 — Strobed Input/Output
Mode 2 — Bi-Directional Bus

When the RESET input goes "high" all ports will be set to the Input mode (i.e., all 24 lines will be in the high impedance state). After the RESET is removed the 8255 can remain in the Input mode with no additional initialization required. During the execution of the system program any of the other modes may be selected using a single OUTput instruction. This allows a single 8255 to service a variety of peripheral devices with a simple software maintenance routine.

The modes for Port A and Port B can be separately defined, while Port C is divided into two portions as required by the Port A and Port B definitions. All of the output registers, including the status flip-flops, will be reset whenever the mode is changed. Modes may be combined so that their functional definition can be "tailored" to almost any I/O structure. For instance; Group B can be programmed in Mode 0 to monitor simple switch closings or display computational results, Group A could be programmed in Mode 1 to monitor a keyboard or tape reader on an interrupt-driven basis.



**Basic Mode Definitions and Bus Interface**

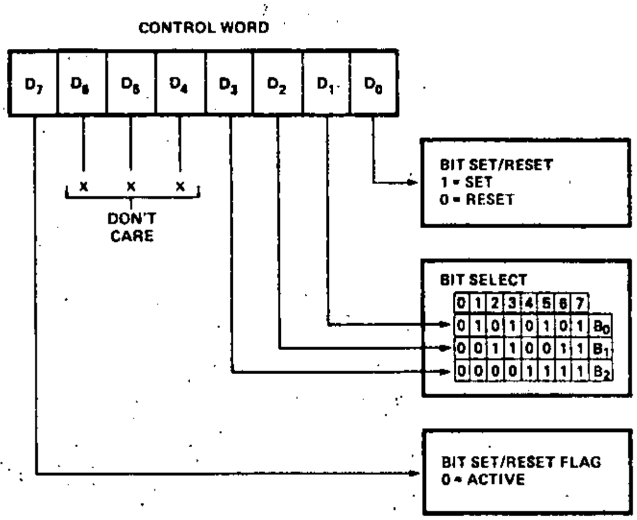


**Mode Definition Format**

The Mode definitions and possible Mode combinations may seem confusing at first but after a cursory review of the complete device operation a simple, logical I/O approach will surface. The design of the 8255 has taken into account things such as efficient PC board layout, control signal definition vs PC layout and complete functional flexibility to support almost any peripheral device with no external logic. Such design represents the maximum use of the available pins.

### Single Bit Set/Reset Feature

Any of the eight bits of Port C can be Set or Reset using a single OUTput instruction. This feature reduces software requirements in Control-based applications.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $B_0$ |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | $B_1$ |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | $B_2$ |

**Bit Set/Reset Format**

When Port C is being used as status/control for Port A or B, these bits can be set or reset by using the Bit Set/Reset operation just as if they were data output ports.

### Interrupt Control Functions

When the 8255 is programmed to operate in Mode 1 or Mode 2, control signals are provided that can be used as interrupt request inputs to the CPU. The interrupt request signals, generated from Port C, can be inhibited or enabled by setting or resetting the associated INTE flip-flop, using the Bit set/reset function of Port C.

This function allows the Programmer to disallow or allow a specific I/O device to interrupt the CPU without effecting any other device in the interrupt structure.

INTE flip-flop definition:

(BIT-SET) – INTE is SET – Interrupt enable
(BIT-RESET) – INTE is RESET – Interrupt disable

Note: All Mask flip-flops are automatically reset during mode selection and device Reset.

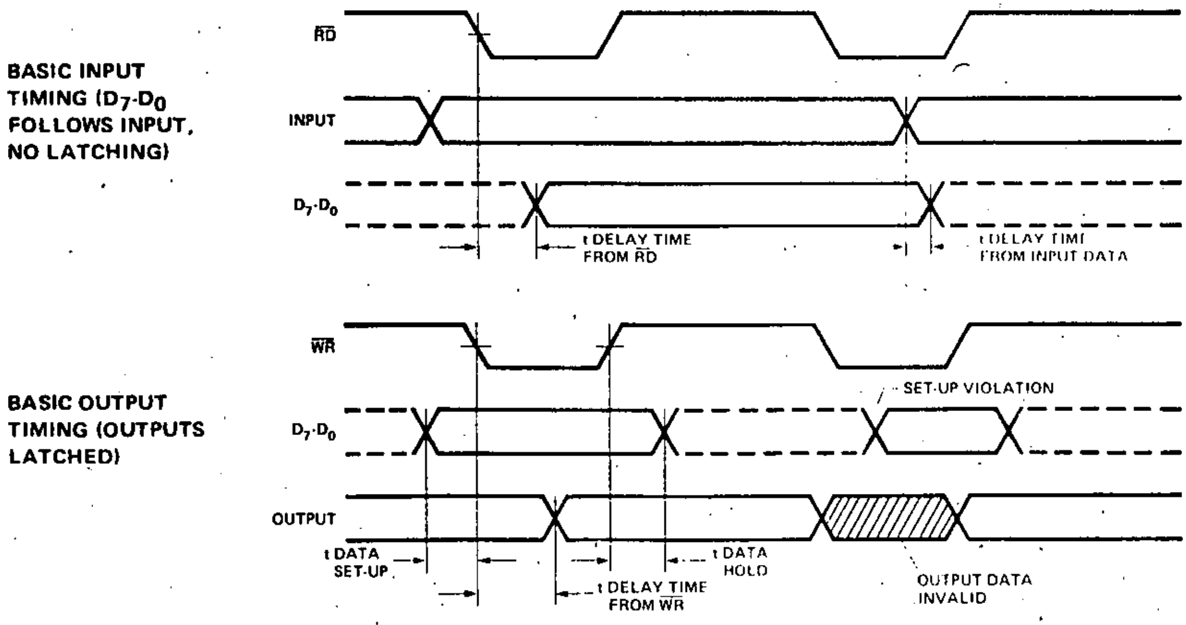
## Operating Modes

### Mode 0 (Basic Input/Output)

This functional configuration provides simple Input and Output operations for each of the three ports. No "handshaking" is required, data is simply written to or read from a specified port.

Mode 0 Basic Functional Definitions:

- Two 8-bit ports and two 4-bit ports.
- Any port can be input or output.
- Outputs are latched.
- Inputs are not latched.
- 16 different Input/Output configurations are possible in this Mode.



**Mode 0 Timing**

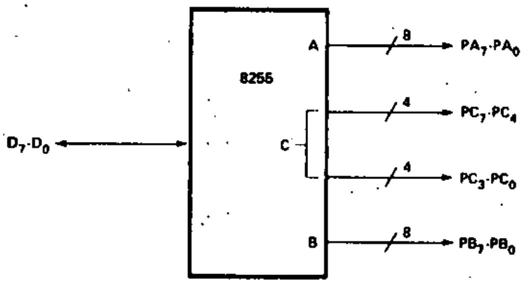
## MODE 0 PORT DEFINITION CHART

| A | | B | | GROUP A | | | GROUP B | |
|---|---|---|---|---|---|---|---|---|
| $D_4$ | $D_3$ | $D_1$ | $D_0$ | PORT A | PORT C (UPPER) | # | PORT B | PORT C (LOWER) |
| 0 | 0 | 0 | 0 | OUTPUT | OUTPUT | 0 | OUTPUT | OUTPUT |
| 0 | 0 | 0 | 1 | OUTPUT | OUTPUT | 1 | OUTPUT | INPUT |
| 0 | 0 | 1 | 0 | OUTPUT | OUTPUT | 2 | INPUT | OUTPUT |
| 0 | 0 | 1 | 1 | OUTPUT | OUTPUT | 3 | INPUT | INPUT |
| 0 | 1 | 0 | 0 | OUTPUT | INPUT | 4 | OUTPUT | OUTPUT |
| 0 | 1 | 0 | 1 | OUTPUT | INPUT | 5 | OUTPUT | INPUT |
| 0 | 1 | 1 | 0 | OUTPUT | INPUT | 6 | INPUT | OUTPUT |
| 0 | 1 | 1 | 1 | OUTPUT | INPUT | 7 | INPUT | INPUT |
| 1 | 0 | 0 | 0 | INPUT | OUTPUT | 8 | OUTPUT | OUTPUT |
| 1 | 0 | 0 | 1 | INPUT | OUTPUT | 9 | OUTPUT | INPUT |
| 1 | 0 | 1 | 0 | INPUT | OUTPUT | 10 | INPUT | OUTPUT |
| 1 | 0 | 1 | 1 | INPUT | OUTPUT | 11 | INPUT | INPUT |
| 1 | 1 | 0 | 0 | INPUT | INPUT | 12 | OUTPUT | OUTPUT |
| 1 | 1 | 0 | 1 | INPUT | INPUT | 13 | OUTPUT | INPUT |
| 1 | 1 | 1 | 0 | INPUT | INPUT | 14 | INPUT | OUTPUT |
| 1 | 1 | 1 | 1 | INPUT | INPUT | 15 | INPUT | INPUT |

## MODE 0 CONFIGURATIONS

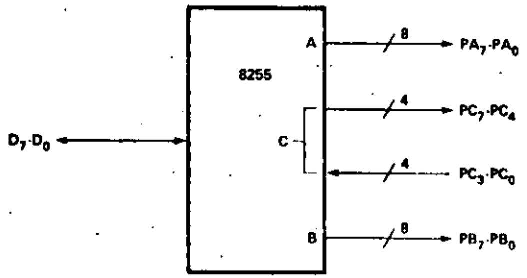
CONTROL WORD #0

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



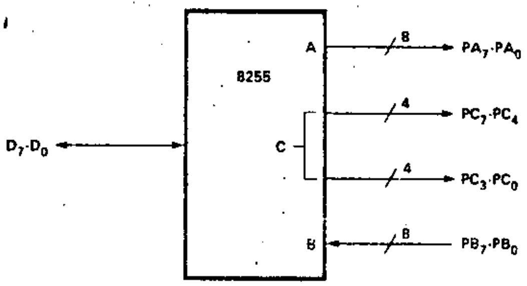
CONTROL WORD #1

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



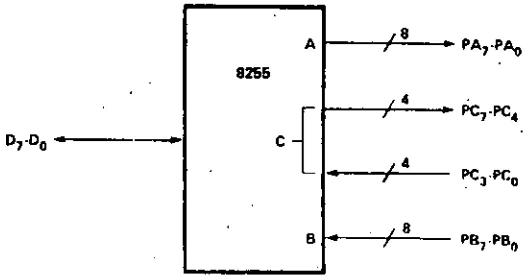
CONTROL WORD #2

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |



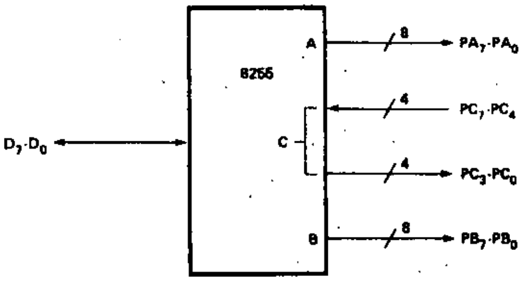
CONTROL WORD #3

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |



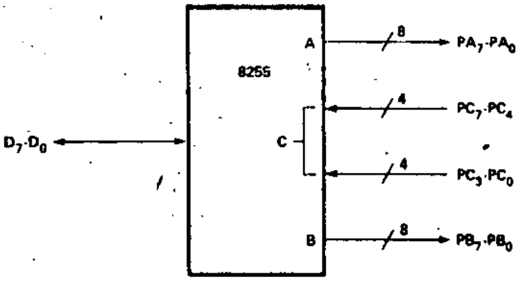
CONTROL WORD #4

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |



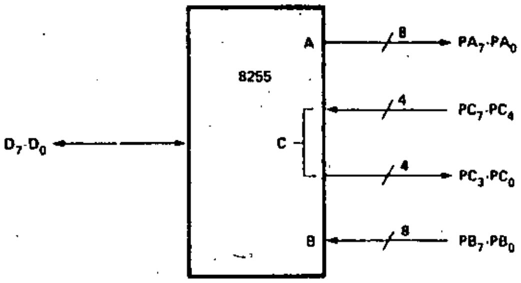
CONTROL WORD #5

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |



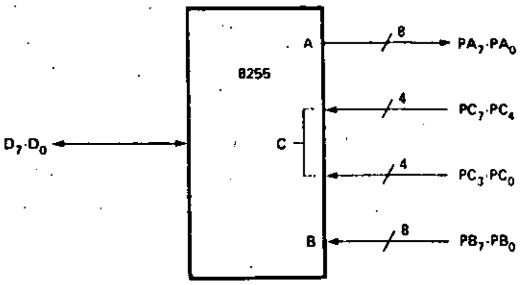
CONTROL WORD #6

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |



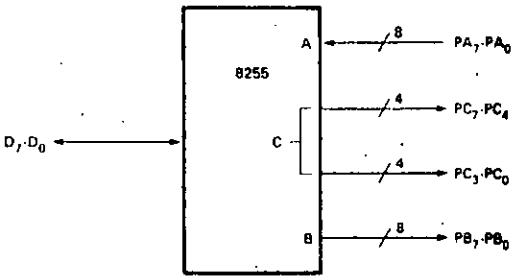
CONTROL WORD #7

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |



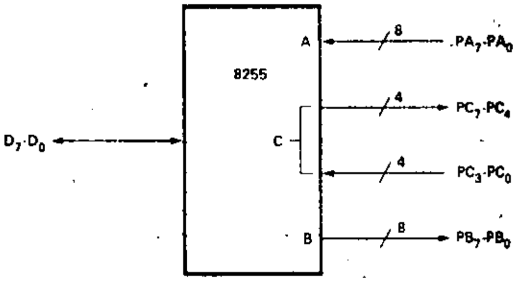
CONTROL WORD #8

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |



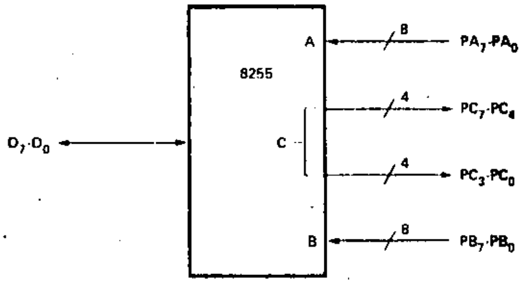
CONTROL WORD #9

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |



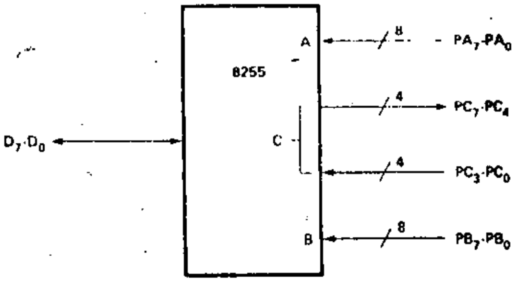
CONTROL WORD #10

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |



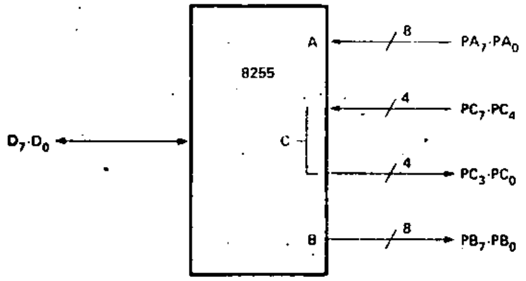
CONTROL WORD #11

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |



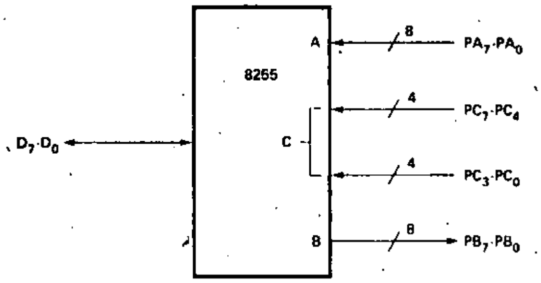
CONTROL WORD #12

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |



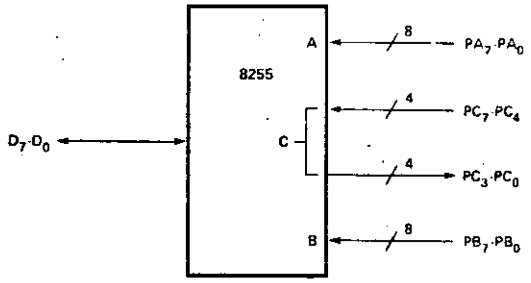
CONTROL WORD #13

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |



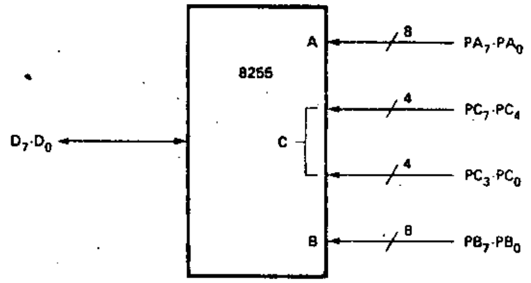
CONTROL WORD #14

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |



CONTROL WORD #15

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |



## Operating Modes

### Mode 1 (Strobed Input/Output)

This functional configuration provides a means for transferring I/O data to or from a specified port in conjunction with strobes or "handshaking" signals. In Mode 1, Port A and Port B use the lines on Port C to generate or accept these "handshaking" signals.

Mode 1 Basic Functional Definitions:

- Two Groups (Group A and Group B)
- Each group contains one 8-bit data port and one 4-bit control/data port.
- The 8-bit data port can be either input or output. Both inputs and outputs are latched.
- The 4-bit port is used for control and status of the 8-bit data port.

## Input Control Signal Definition

**$\overline{\text{STB}}$ (Strobe Input)**

A "low" on this input loads data into the input latch.

### IBF (Input Buffer Full F/F)

A "high" on this output indicates that the data has been loaded into the input latch; in essence, an acknowledgement. IBF is set by the falling edge of the STB input and is reset by the rising edge of the RD input.

### INTR (Interrupt Request)

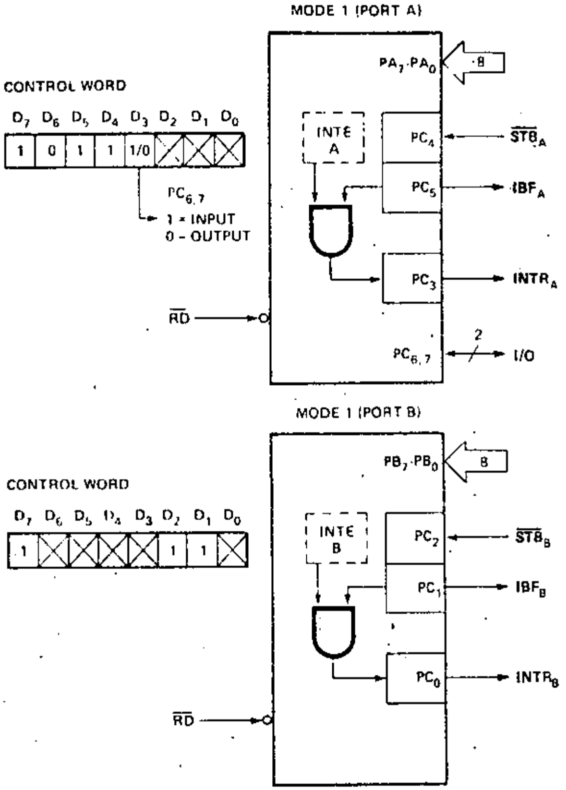
A "high" on this output can be used to interrupt the CPU when an input device is requesting service. INTR is set by the rising edge of $\overline{\text{STB}}$ if IBF is a "one" and INTE is a "one". It is reset by the falling edge of $\overline{\text{RD}}$. This procedure allows an input device to request service from the CPU by simply strobing its data into the port.

**INTE A**

Controlled by bit set/reset of $PC_4$.

**INTE B**

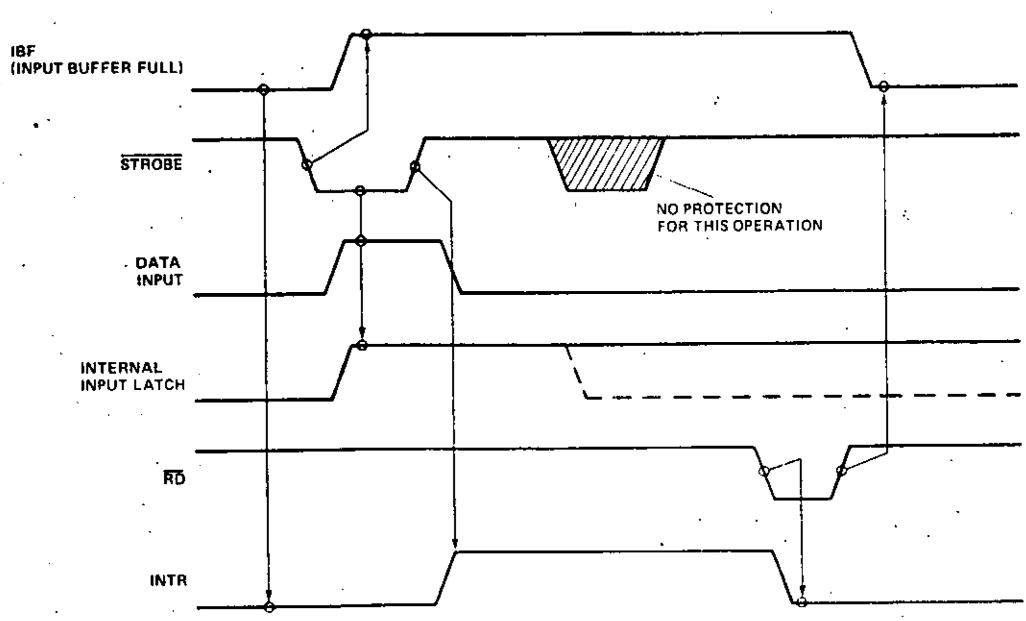
Controlled by bit set/reset of $PC_2$.



Mode 1 Input

MODE 1 (STROBED INPUT)

BASIC TIMING



Basic Timing Input

## Output Control Signal Definition

### $\overline{\text{OBF}}$ (Output Buffer Full F/F)

The $\overline{\text{OBF}}$ output will go "low" to indicate that the CPU has written data out to the specified port. The OBF F/F will be set by the rising edge of the WR input and reset by the falling edge of the $\overline{\text{ACK}}$ input signal.

### $\overline{\text{ACK}}$ (Acknowledge Input)

A "low" on this input informs the 8255 that the data from Port A or Port B has been accepted. In essence, a response from the peripheral device indicating that it has received the data output by the CPU.

### INTR (Interrupt Request)

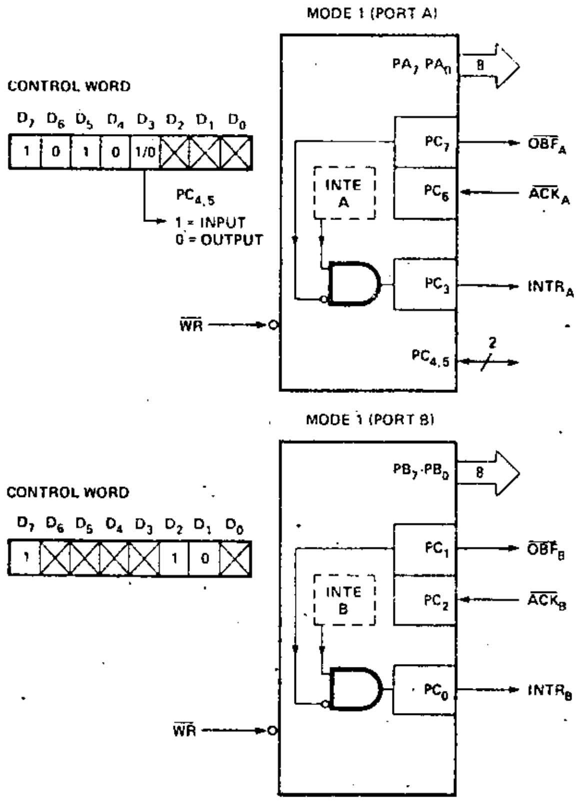
A "high" on this output can be used to interrupt the CPU when an output device has accepted data transmitted by the CPU. INTR is set by the rising edge of $\overline{\text{ACK}}$ if $\overline{\text{OBF}}$ is a "one" and INTE is a "one". It is reset by the falling edge of $\overline{\text{WR}}$.

**INTE A**

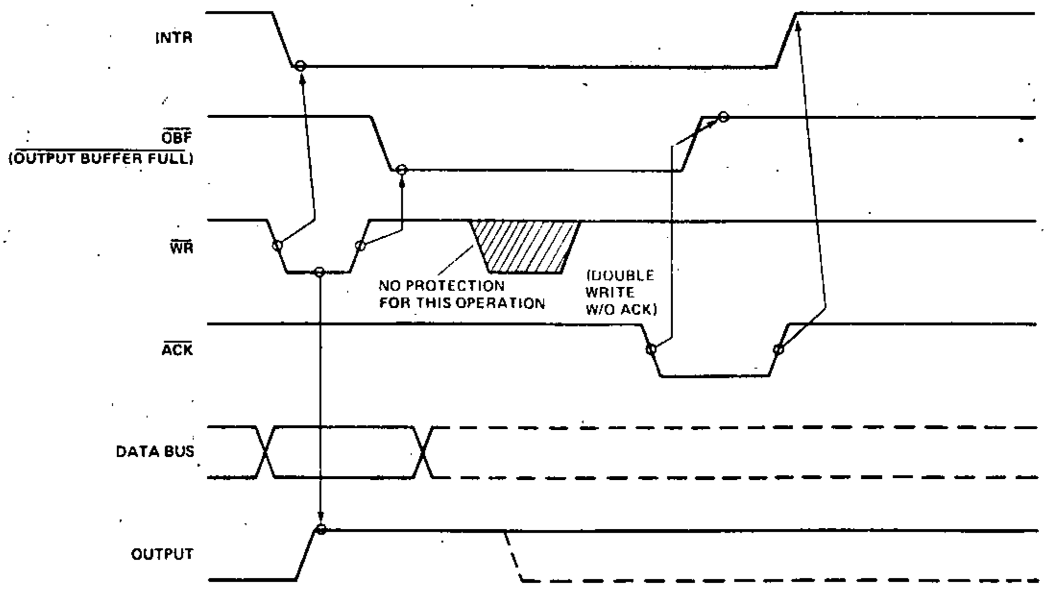
Controlled by bit set/reset of $PC_6$.

**INTE B**

Controlled by bit set/reset of $PC_2$.



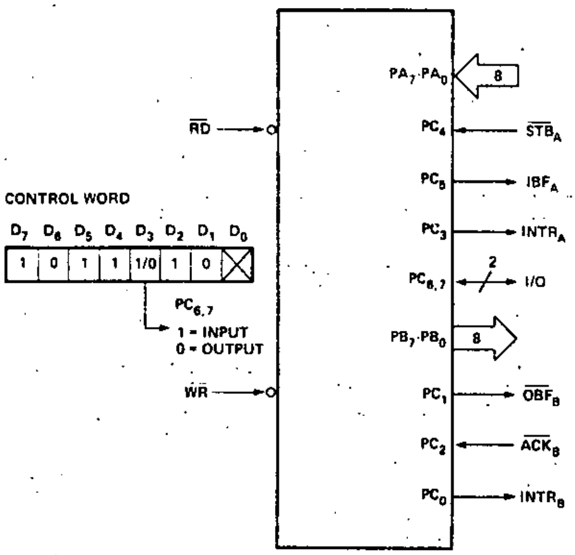
**Mode 1 Output**



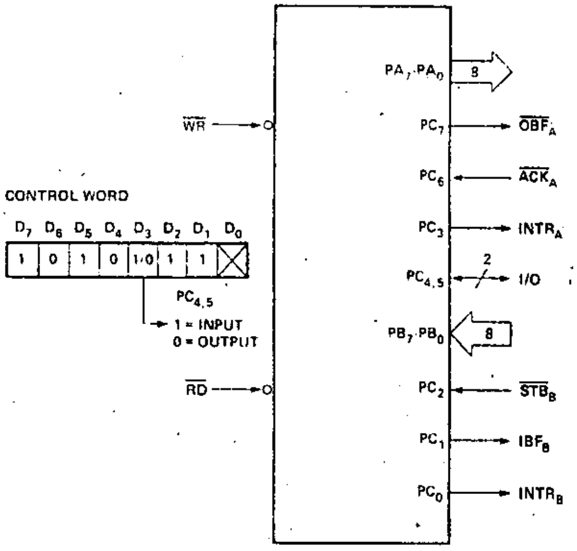
**Basic Timing Output**

## Combinations of Mode 1

Port A and Port B can be individually defined as input or output in Mode 1 to support a wide variety of strobed I/O applications.



PORT A – (STROBED INPUT)
PORT B – (STROBED OUTPUT)



PORT A – (STROBED OUTPUT)
PORT B – (STROBED INPUT)

## Operating Modes

### Mode 2 (Strobed Bi-Directional Bus I/O)

This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bi-directional bus I/O). "Handshaking" signals are provided to maintain proper bus flow discipline in a similar manner to Mode 1. Interrupt generation and enable/disable functions are also available.

Mode 2 Basic Functional Definitions:

- Used in Group A only.
- One 8-bit, bi-directional bus Port (Port A) and a 5-bit control Port (Port C).
- Both inputs and outputs are latched.
- The 5-bit control port (Port C) is used for control and status for the 8-bit, bi-directional bus port (Port A).

## Bi-Directional Bus I/O Control Signal Definition

### INTR (Interrupt Request)

A high on this output can be used to interrupt the CPU for both input or output operations.

## Output Operations

### $\overline{\text{OBF}}$ (Output Buffer Full)

The $\overline{\text{OBF}}$ output will go "low" to indicate that the CPU has written data out to Port A.

### $\overline{\text{ACK}}$ (Acknowledge)

A "low" on this input enables the tri-state output buffer of Port A to send out the data. Otherwise, the output buffer will be in the high-impedance state.

### INTE 1 (The INTE Flip-Flop associated with $\overline{\text{OBF}}$)

Controlled by bit set/reset of $PC_6$.

## Input Operations

### $\overline{\text{STB}}$ (Strobe Input)

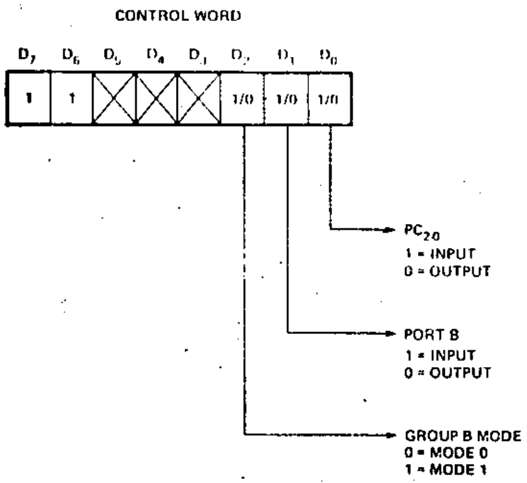
A "low" on this input loads data into the input latch.

### IBF (Input Buffer Full F/F)

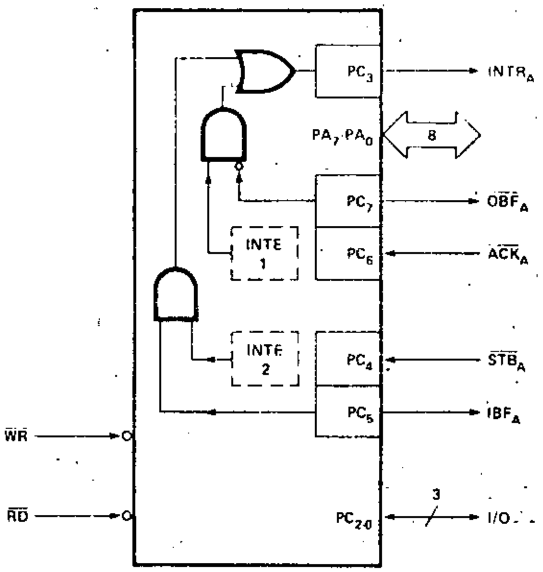
A "high" on this output indicates that data has been loaded into the input latch.

### INTE 2 (The INTE Flip-Flop associated with IBF)

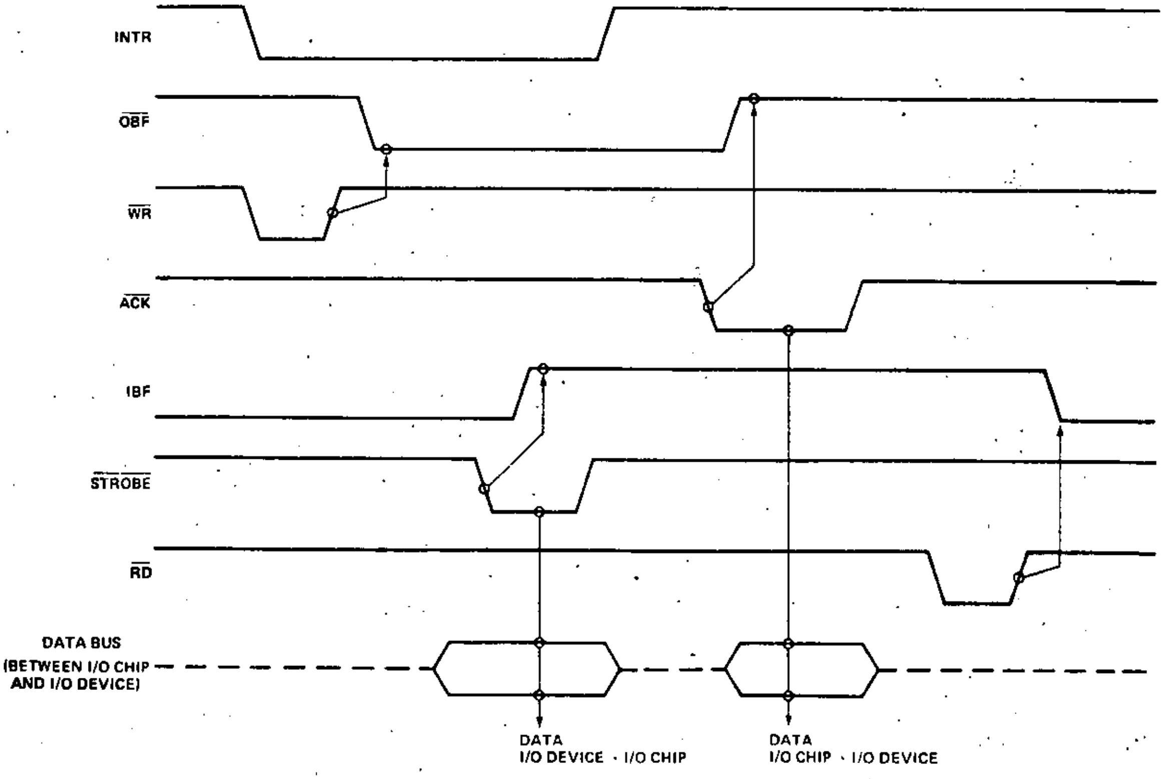
Controlled by bit set/reset of $PC_4$.



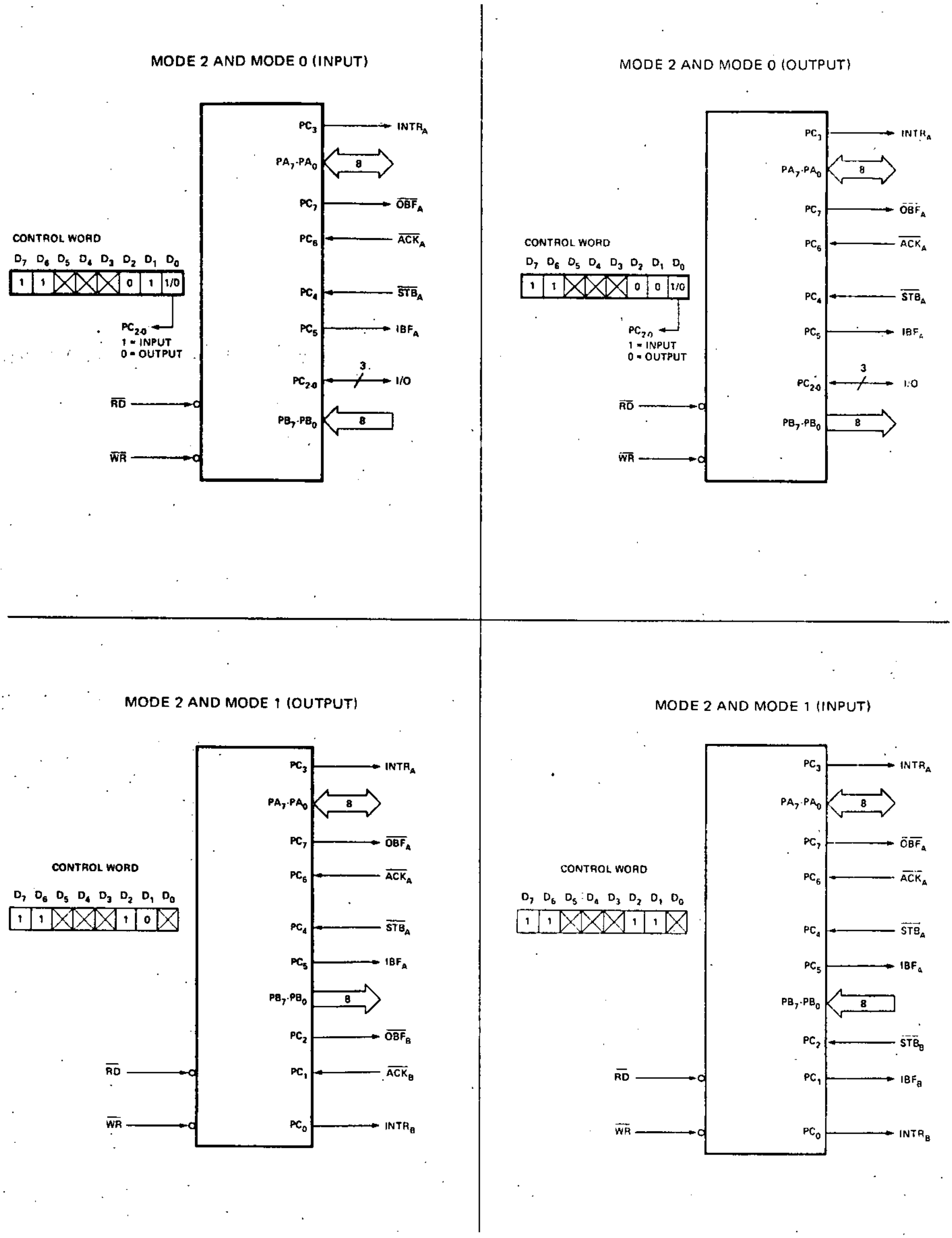
Mode 2 Control Word



Mode 2



Mode 2 (Bi-directional) Timing



MODE 2 AND MODE 0 (INPUT)

MODE 2 AND MODE 0 (OUTPUT)

MODE 2 AND MODE 1 (OUTPUT)

MODE 2 AND MODE 1 (INPUT)

**Mode 2 Combinations**

## MODE DEFINITION SUMMARY TABLE

| | MODE 0 | | MODE 1 | | MODE 2 |
|---|---|---|---|---|---|
| | IN | OUT | IN | OUT | GROUP A ONLY |
| $PA_0$ | IN | OUT | IN | OUT | ↔ |
| $PA_1$ | IN | OUT | IN | OUT | ↔ |
| $PA_2$ | IN | OUT | IN | OUT | ↔ |
| $PA_3$ | IN | OUT | IN | OUT | ↔ |
| $PA_4$ | IN | OUT | IN | OUT | ↔ |
| $PA_5$ | IN | OUT | IN | OUT | ↔ |
| $PA_6$ | IN | OUT | IN | OUT | ↔ |
| $PA_7$ | IN | OUT | IN | OUT | ↔ |
| $PB_0$ | IN | OUT | IN | OUT | — |
| $PB_1$ | IN | OUT | IN | OUT | — |
| $PB_2$ | IN | OUT | IN | OUT | — |
| $PB_3$ | IN | OUT | IN | OUT | — |
| $PB_4$ | IN | OUT | IN | OUT | — |
| $PB_5$ | IN | OUT | IN | OUT | — |
| $PB_6$ | IN | OUT | IN | OUT | — |
| $PB_7$ | IN | OUT | IN | OUT | — |
| $PC_0$ | IN | OUT | $INTR_B$ | $INTR_B$ | I/O |
| $PC_1$ | IN | OUT | $IBF_B$ | $\overline{OBF}_B$ | I/O |
| $PC_2$ | IN | OUT | $\overline{STB}_B$ | $\overline{ACK}_B$ | I/O |
| $PC_3$ | IN | OUT | $INTR_A$ | $INTR_A$ | $INTR_A$ |
| $PC_4$ | IN | OUT | $\overline{STB}_A$ | I/O | $\overline{STB}_A$ |
| $PC_5$ | IN | OUT | $IBF_A$ | I/O | $IBF_A$ |
| $PC_6$ | IN | OUT | I/O | $\overline{ACK}_A$ | $\overline{ACK}_A$ |
| $PC_7$ | IN | OUT | I/O | $\overline{OBF}_A$ | $\overline{OBF}_A$ |

MODE 0 OR MODE 1 ONLY

### Special Mode Combination Considerations

There are several combinations of modes when not all of the bits in Port C are used for control or status. The remaining bits can be used as follows:

If Programmed as Inputs –
All input lines can be accessed during a normal Port C read.

If Programmed as Outputs –
Bits in C upper ($PC_7$-$PC_4$) must be individually accessed using the bit set/reset function.

Bits in C lower ($PC_3$-$PC_0$) can be accessed using the bit set/reset function or accessed as a threesome by writing into Port C.

### Source Current Capability on Port B and Port C

Any set of eight output buffers, selected randomly from Ports B and C can source 1mA at 1.5 volts. This feature allows the 8255 to directly drive Darlington type drivers and high-voltage displays that require such source current.

### Reading Port C Status

In Mode 0, Port C transfers data to or from the peripheral device. When the 8255 is programmed to function in Modes 1 or 2, Port C generates or accepts "hand-shaking" signals with the peripheral device. Reading the contents of Port C allows the programmer to test or verify the "status" of each peripheral device and change the program flow accordingly.

There is no special instruction to read the status information from Port C. A normal read operation of Port C is executed to perform this function.

INPUT CONFIGURATION

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| I/O | I/O | $IBF_A$ | $INTE_A$ | $INTR_A$ | $INTE_B$ | $IBF_B$ | $INTR_B$ |
| GROUP A | | | | | GROUP B | | |

OUTPUT CONFIGURATION

| $D_7$ | $D_6$ | $D_5$ | $D_1$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| $\overline{OBF}_A$ | $INTE_A$ | I/O | I/O | $INTR_A$ | $INTE_B$ | $\overline{OBF}_B$ | $INTR_B$ |
| GROUP A | | | | | GROUP B | | |

**Mode 1 Status Word Format**

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| $\overline{OBF}_A$ | $INTE_1$ | $IBF_A$ | $INTE_2$ | $INTR_A$ | ☒ | ☒ | ☒ |
| GROUP A | | | | | GROUP B | | |

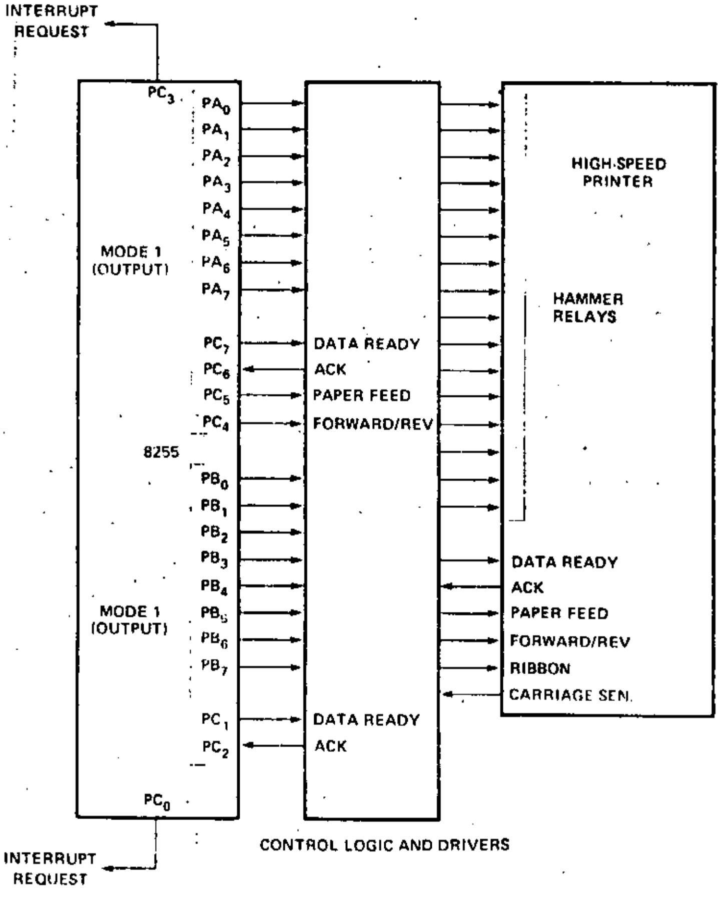
(DEFINED BY MODE 0 OR MODE 1 SELECTION)

**Mode 2 Status Word Format**

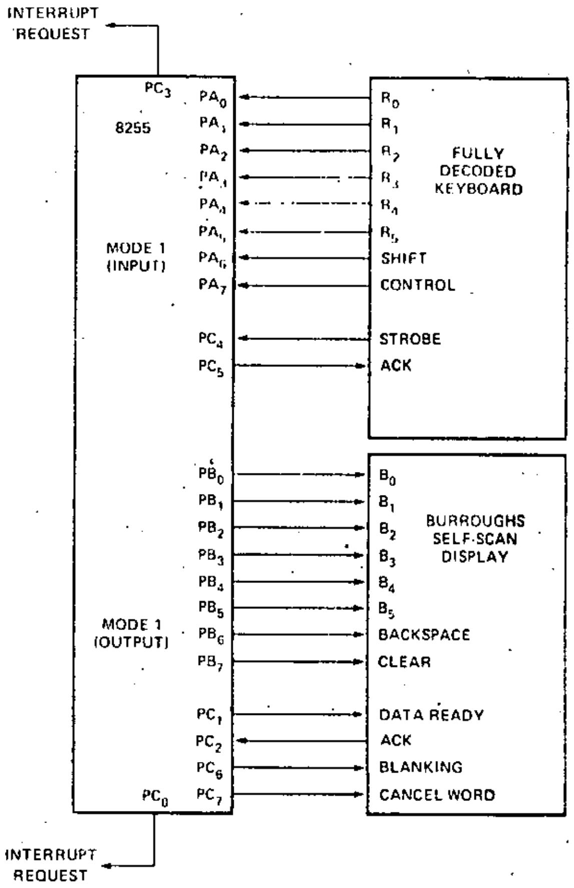
## APPLICATIONS OF THE 8255

The 8255 is a very powerful tool for interfacing peripheral equipment to the 8080 microcomputer system. It represents the optimum use of available pins and is flexible enough to interface almost any I/O device without the need for additional external logic.

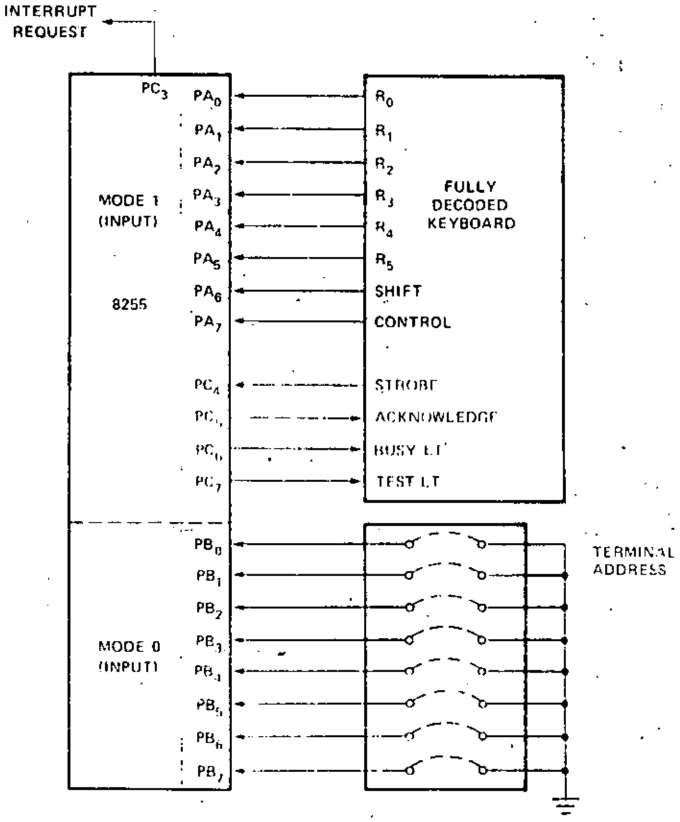
Each peripheral device in a Microcomputer system usually has a "service routine" associated with it. The routine manages the software interface between the device and the CPU. The functional definition of the 8255 is programmed by the I/O service routine and becomes an extension of the systems software. By examining the I/O devices interface characteristics for both data transfer and timing, and matching this information to the examples and tables in the Detailed Operational Description, a control word can easily be developed to initialize the 8255 to exactly "fit" the application. Here are a few examples of typical applications of the 8255.



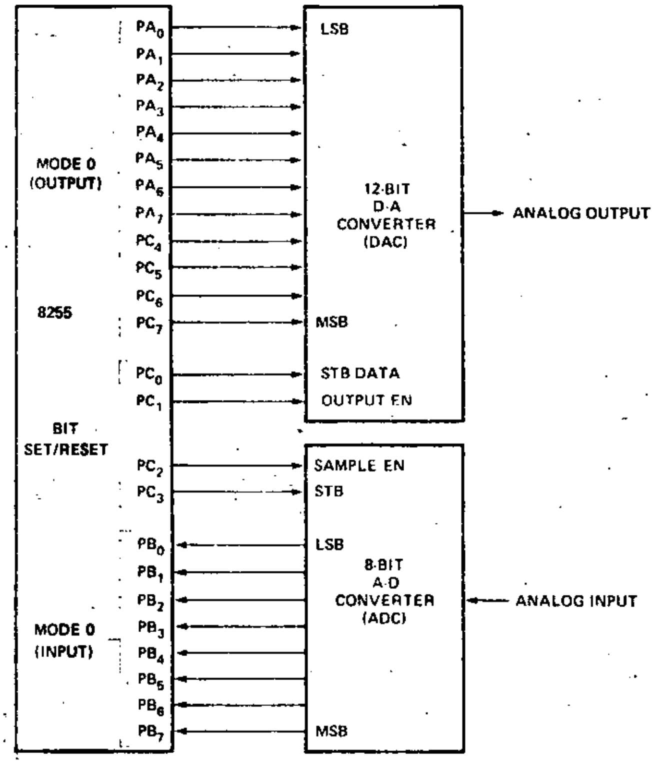
Printer Interface



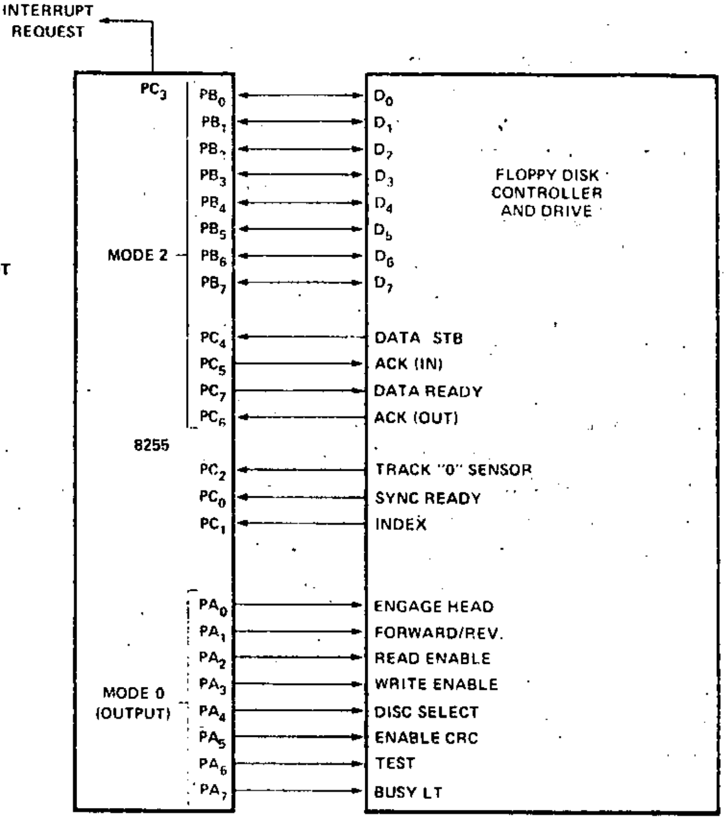
Keyboard and Display Interface



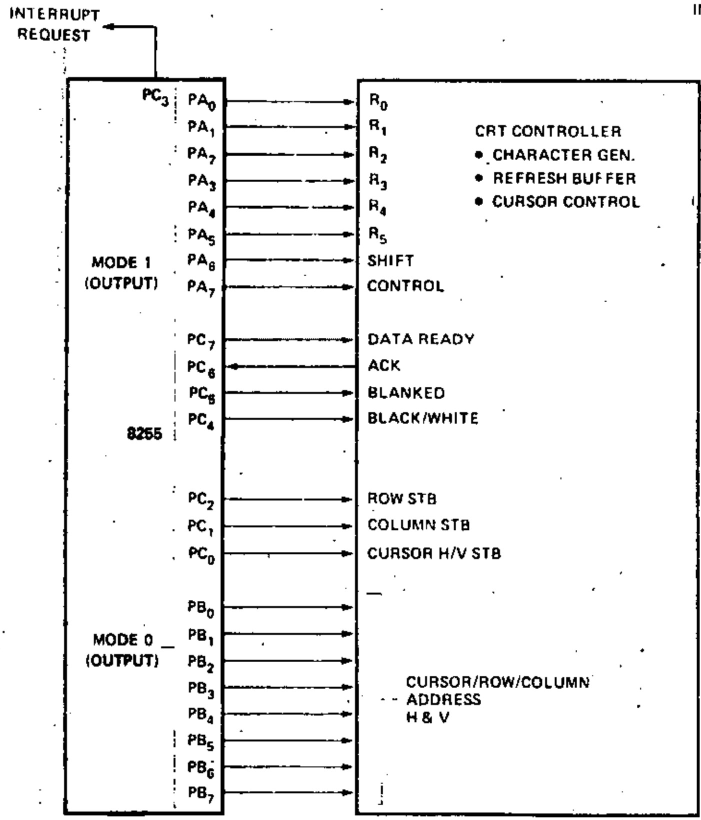
Keyboard and Terminal Address Interface



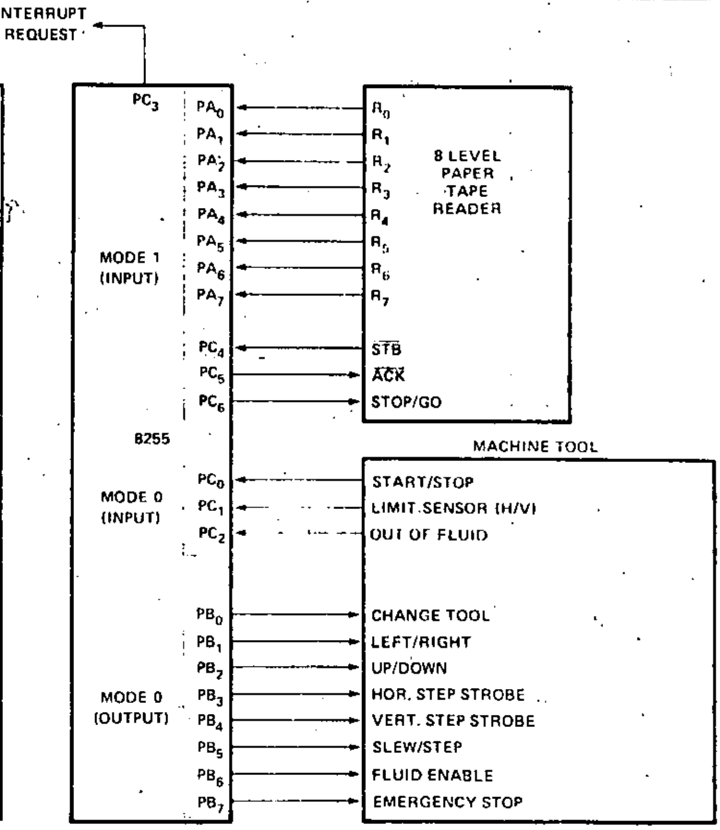
Digital to Analog, Analog to Digital



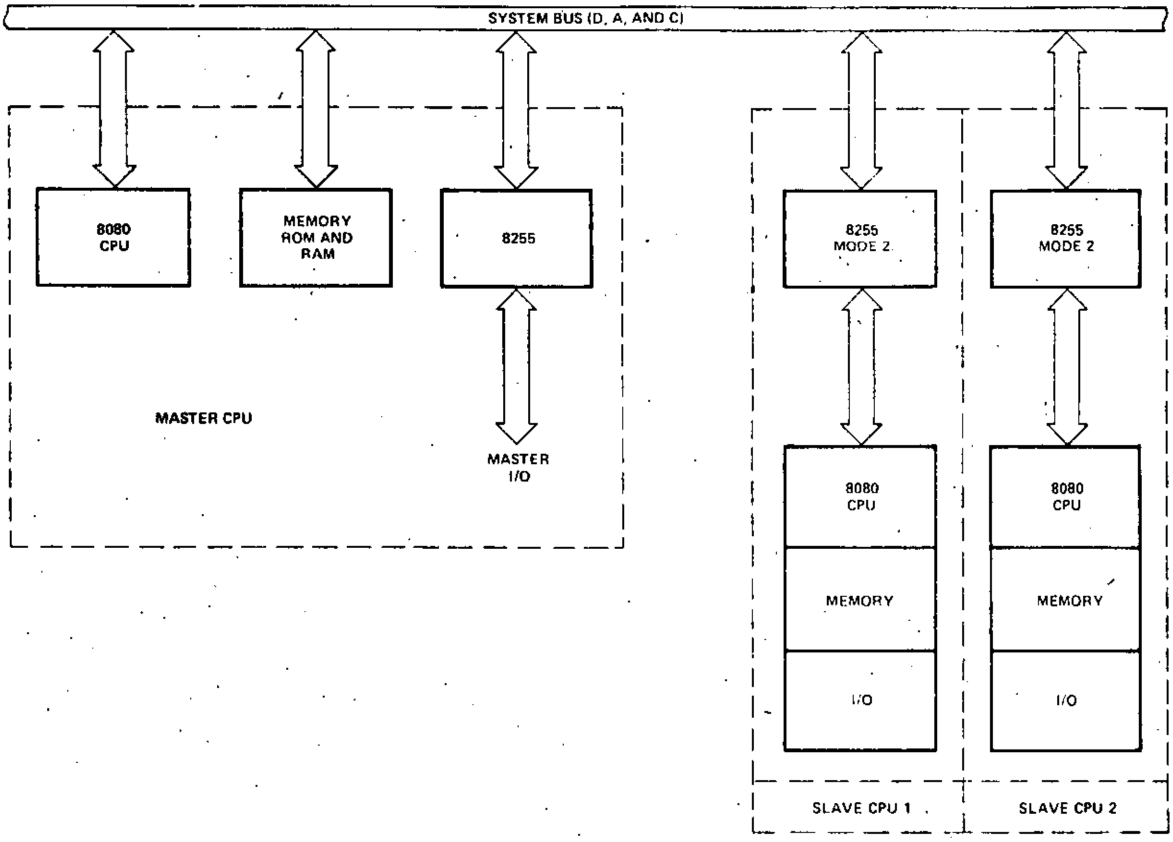
Basic Floppy Disc Interface



Basic CRT Controller Interface



Machine Tool Controller Interface



**Distributed Intelligence Multi-Processor Interface**

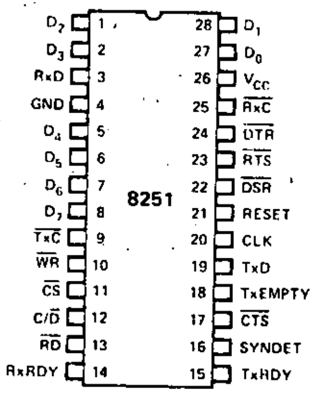
# Silicon Gate MOS 8251

## PROGRAMMABLE COMMUNICATION INTERFACE

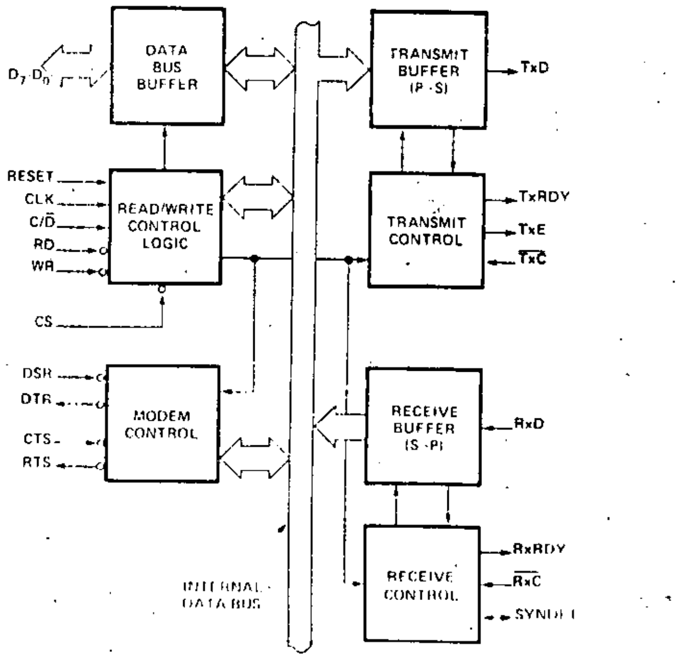
- **Synchronous and Asynchronous Operation**
  - **Synchronous:**
    **5-8 Bit Characters**
    **Internal or External Character Synchronization**
    **Automatic Sync Insertion**
  - **Asynchronous:**
    **5-8 Bit Characters**
    **Clock Rate — 1,16 or 64 Times Baud Rate**
    **Break Character Generation**
    **1, 1½, or 2 Stop Bits**
    **False Start Bit Detection**
- **Baud Rate —DC to 56k Baud (Sync Mode)**
  **DC to 9.6k Baud (Async Mode)**
- **Full Duplex, Double Buffered, Transmitter and Receiver**
- **Error Detection — Parity, Overrun, and Framing**
- **Fully Compatible with 8080 CPU**
- **28-Pin DIP Package**
- **All Inputs and Outputs Are TTL Compatible**
- **Single 5 Volt Supply**
- **Single TTL Clock**

The 8251 is a Universal Synchronous/Asynchronous Receiver/Transmitter (USART) Chip designed for data communications in microcomputer systems. The USART is used as a peripheral device and is programmed by the CPU to operate using virtually any serial data transmission technique presently in use (including IBM Bi-Sync). The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream for transmission. Simultaneously it can receive serial data streams and convert them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the complete status of the USART at any time. These include data transmission errors and control signals such as SYNDET, TxEMPT. The chip is constructed using N-channel silicon gate technology.

PIN CONFIGURATION



BLOCK DIAGRAM



| Pin Name | Pin Function |
|---|---|
| $D_7$-$D_0$ | Data Bus (8 bits) |
| C/$\overline{D}$ | Control or Data is to be Written or Read |
| $\overline{RD}$ | Read Data Command |
| $\overline{WR}$ | Write Data or Control Command |
| $\overline{CS}$ | Chip Enable |
| CLK | Clock Pulse (TTL) |
| RESET | Reset |
| $\overline{TxC}$ | Transmitter Clock |
| TxD | Transmitter Data |
| $\overline{RxC}$ | Receiver Clock |
| RxD | Receiver Data |
| RxRDY | Receiver Ready (has character for 8080) |
| TxRDY | Transmitter Ready (ready for char. from 8080) |

| Pin Name | Pin Function |
|---|---|
| $\overline{DSR}$ | Data Set Ready |
| $\overline{DTR}$ | Data Terminal Ready |
| SYNDET | Sync Detect |
| $\overline{RTS}$ | Request to Send Data |
| $\overline{CTS}$ | Clear to Send Data |
| TxE | Transmitter Empty |
| $V_{CC}$ | +5 Volt Supply |
| GND | Ground |

## 8251 BASIC FUNCTIONAL DESCRIPTION

### General

The 8251 is a Universal Synchronous/Asynchronous Receiver/Transmitter designed specifically for the 8080 Microcomputer System. Like other I/O devices in the 8080 Microcomputer System its functional configuration is programmed by the systems software for maximum flexibility. The 8251 can support virtually any serial data technique currently in use (including IBM "bi-sync").

In a communication environment an interface device must convert parallel format system data into serial format for transmission and convert incoming serial format data into parallel system data for reception. The interface device must also delete or insert bits or characters that are functionally unique to the communication technique. In essence, the interface should appear "transparent" to the CPU, a simple input or output of byte-oriented system data.

### Data Bus Buffer

This 3-state, bi-directional, 8-bit buffer is used to interface the 8251 to the 8080 system Data Bus. Data is transmitted or received by the buffer upon execution of INput or OUTput instructions of the 8080 CPU. Control words, Command words and Status information are also transferred through the Data Bus Buffer.

### Read/Write Control Logic

This functional block accepts inputs from the 8080 Control bus and generates control signals for overall device operation. It contains the Control Word Register and Command Word Register that store the various control formats for device functional definition.

### RESET (Reset)

A "high" on this input forces the 8251 into an "Idle" mode. The device will remain at "Idle" until a new set of control words is written into the 8251 to program its functional definition.

### CLK (Clock)

The CLK input is used to generate internal device timing and is normally connected to the Phase 2 (TTL) output of the 8224 Clock Generator. No external inputs or outputs are referenced to CLK but the frequency of CLK must be greater than 30 times the Receiver or Transmitter clock inputs for synchronous mode (4.5 times for asynchronous mode).

### $\overline{WR}$ (Write)

A "low" on this input informs the 8251 that the CPU is outputting data or control words, in essence, the CPU is writing out to the 8251.

### $\overline{RD}$ (Read)

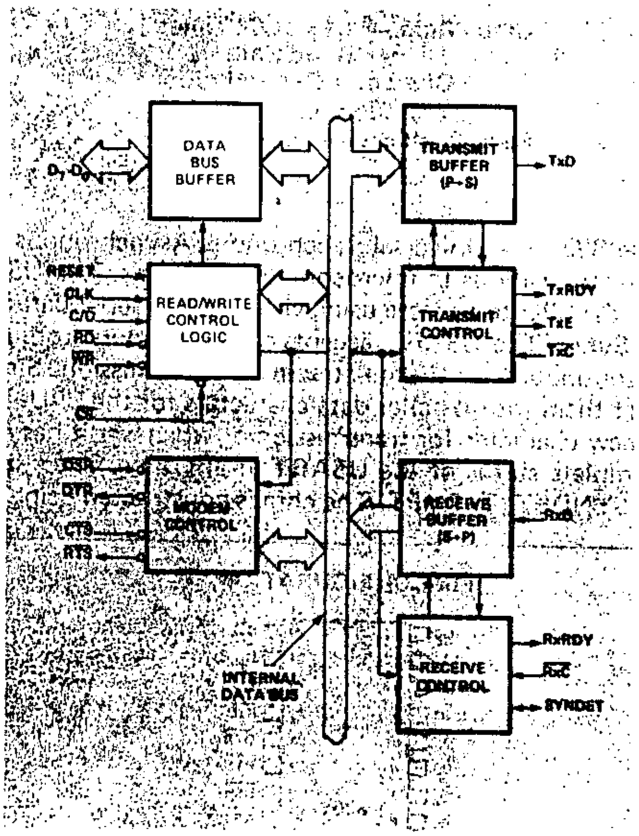
A "low" on this input informs the 8251 that the CPU is inputting data or status information, in essence, the CPU is reading from the 8251.

### C/$\overline{D}$ (Control/Data)

This input, in conjunction with the $\overline{WR}$ and $\overline{RD}$ inputs in forms the 8251 that the word on the Data Bus is either a data character, control word or status information.
1 = CONTROL 0 = DATA

### $\overline{CS}$ (Chip Select)

A "low" on this input enables the 8251. No reading or writing will occur unless the device is selected.



| C/$\overline{D}$ | $\overline{RD}$ | $\overline{WR}$ | $\overline{CS}$ | |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 8251 ⇒ DATA BUS |
| 0 | 1 | 0 | 0 | DATA BUS ⇒ 8251 |
| 1 | 0 | 1 | 0 | STATUS ⇒ DATA BUS |
| 1 | 1 | 0 | 0 | DATA BUS ⇒ CONTROL |
| X | X | X | 1 | DATA BUS ⇒ 3-STATE |

### Modem Control

The 8251 has a set of control inputs and outputs that can be used to simplify the interface to almost any Modem. The modem control signals are general purpose in nature and can be used for functions other than Modem control, if necessary.

### $\overline{\text{DSR}}$ (Data Set Ready)

The $\overline{\text{DSR}}$ input signal is general purpose in nature. Its condition can be tested by the CPU using a Status Read operation. The $\overline{\text{DSR}}$ input is normally used to test Modem conditions such as Data Set Ready.

### $\overline{\text{DTR}}$ (Data Terminal Ready)

The $\overline{\text{DTR}}$ output signal is general purpose in nature. It can be set "low" by programming the appropriate bit in the Command Instruction word. The $\overline{\text{DTR}}$ output signal is normally used for Modem control such as Data Terminal Ready or Rate Select.

### $\overline{\text{RTS}}$ (Request to Send)

The $\overline{\text{RTS}}$ output signal is general purpose in nature. It can be set "low" by programming the appropriate bit in the Command Instruction word. The $\overline{\text{RTS}}$ output signal is normally used for Modem control such as Request to Send.

### $\overline{\text{CTS}}$ (Clear to Send)

A "low" on this input enables the 8251 to transmit data (serial) if the Tx EN bit in the Command byte is set to a "one."

### Transmitter Buffer

The Transmitter Buffer accepts parallel data from the Data Bus Buffer, converts it to a serial bit stream, inserts the appropriate characters or bits (based on the communication technique) and outputs a composite serial stream of data on the TxD output pin.

### Transmitter Control

The Transmitter Control manages all activities associated with the transmission of serial data. It accepts and issues signals both externally and internally to accomplish this function.

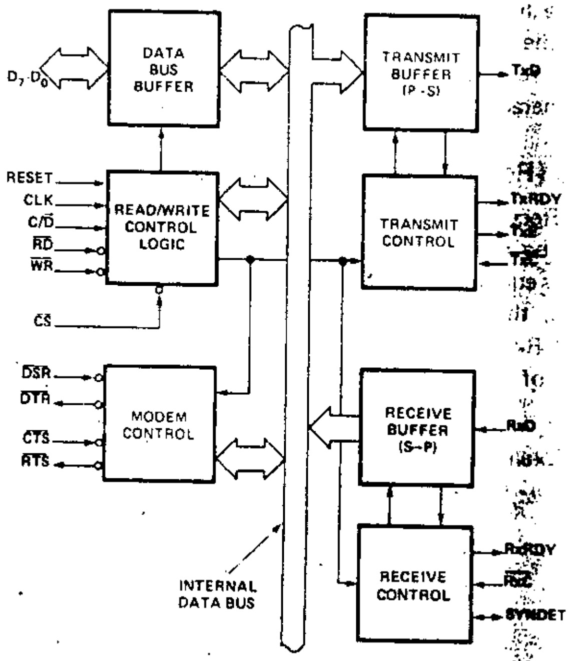
### TxRDY (Transmitter Ready)

This output signals the CPU that the transmitter is ready to accept a data character. It can be used as an interrupt to the system or for the Polled operation the CPU can check TxRDY using a status read operation. TxRDY is automatically reset when a character is loaded from the CPU.

### TxE (Transmitter Empty)

When the 8251 has no characters to transmit, the TxE output will go "high". It resets automatically upon receiving a character from the CPU. TxE can be used to indicate the end of a transmission mode, so that the CPU "knows" when to "turn the line around" in the half-duplexed operational mode.

In SYNChronous mode, a "high" on this output indicates that a character has not been loaded and the SYNC character or characters are about to be transmitted automatically as "fillers".



### $\overline{\text{TxC}}$ (Transmitter Clock)

The Transmitter Clock controls the rate at which the character is to be transmitted. In the Synchronous transmission mode, the frequency of $\overline{\text{TxC}}$ is equal to the actual Baud Rate (1X). In Asynchronous transmission mode, the frequency of $\overline{\text{TxC}}$ is a multiple of the actual Baud Rate. A portion of the mode instruction selects the value of the multiplier; it can be 1x, 16x or 64x the Baud Rate.

For Example:

If Baud Rate equals 110 Baud,  
TxC equals 110 Hz (1x)  
TxC equals 1.76 kHz (16x)  
TxC equals 7.04 kHz (64x).  
If Baud Rate equals 9600 Baud,  
$\overline{\text{TxC}}$ equals 614.4 kHz (64x).

The falling edge of $\overline{\text{TxC}}$ shifts the serial data out of the 8251.

### Receiver Buffer

The Receiver accepts serial data, converts this serial input to parallel format, checks for bits or characters that are unique to the communication technique and sends an "assembled" character to the CPU. Serial data is input to the RxD pin.

### Receiver Control

This functional block manages all receiver-related activities.

### RxRDY (Receiver Ready)

This output indicates that the 8251 contains a character that is ready to be input to the CPU. RxRDY can be connected to the interrupt structure of the CPU or for Polled operation the CPU can check the condition of RxRDY using a status read operation. RxRDY is automatically reset when the character is read by the CPU.

### $\overline{\text{RxC}}$ (Receiver Clock)

The Receiver Clock controls the rate at which the character is to be received. In Synchronous Mode, the frequency of $\overline{\text{RxC}}$ is equal to the actual Baud Rate (1x). In Asynchronous Mode, the frequency of $\overline{\text{RxC}}$ is a multiple of the actual Baud Rate. A portion of the mode instruction selects the value of the multiplier; it can be 1x, 16x or 64x the Baud Rate.

For Example: If Baud Rate equals 300 Baud,
$\overline{\text{RxC}}$ equals 300 Hz (1x)
$\overline{\text{RxC}}$ equals 4800 Hz (16x)
$\overline{\text{RxC}}$ equals 19.2 kHz (64x).
If Baud Rate equals 2400 Baud,
$\overline{\text{RxC}}$ equals 2400 Hz (1x)
$\overline{\text{RxC}}$ equals 38.4 kHz (16x)
$\overline{\text{RxC}}$ equals 153.6 kHz (64x).

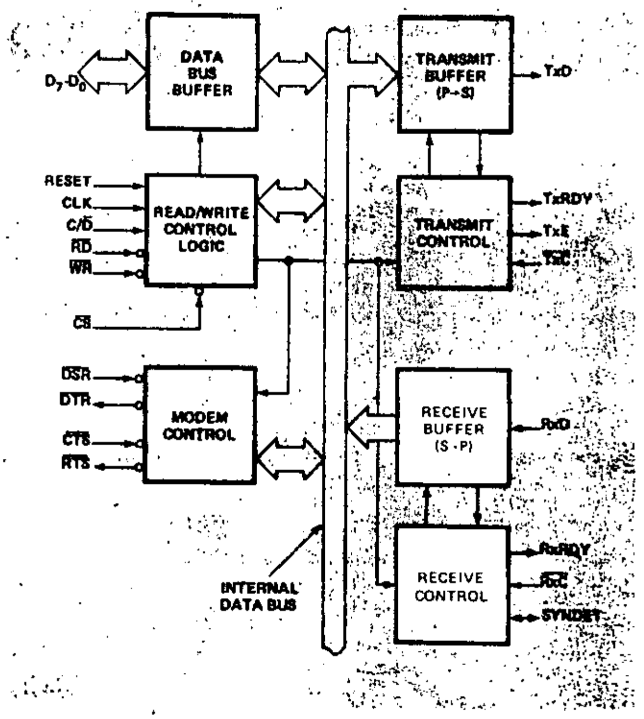
Data is sampled into the 8251 on the rising edge of $\overline{\text{RxC}}$.

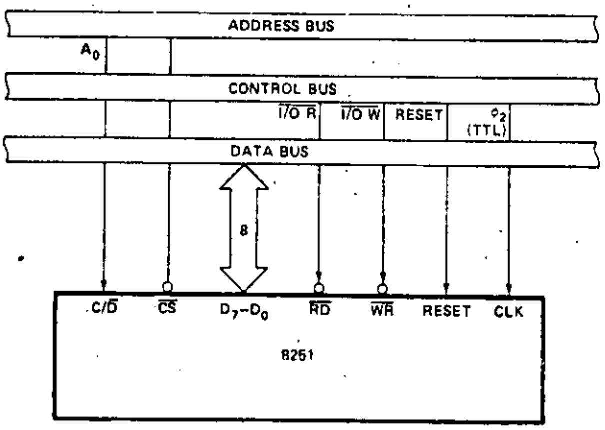
NOTE: In most communications systems, the 8251 will be handling both the transmission and reception operations of a single link. Consequently, the Receive and Transmit Baud Rates will be the same. Both $\overline{\text{TxC}}$ and $\overline{\text{RxC}}$ will require identical frequencies for this operation and can be tied together and connected to a single frequency source (Baud Rate Generator) to simplify the interface.

### SYNDET (SYNC Detect)

This pin is used in SYNChronous Mode only. It is used as either input or output, programmable through the Control Word. It is reset to "low" upon RESET. When used as an output (internal Sync mode), the SYNDET pin will go "high" to indicate that the 8251 has located the SYNC character in the Receive mode. If the 8251 is programmed to use double Sync characters (bi-sync), then SYNDET will go "high" in the middle of the last bit of the second Sync character. SYNDET is automatically reset upon a Status Read operation.

When used as an input, (external SYNC detect mode), a positive going signal will cause the 8251 to start assembling data characters on the falling edge of the next $\overline{\text{RxC}}$. Once in SYNC, the "high" input signal can be removed. The duration of the high signal should be at least equal to the period of $\overline{\text{RxC}}$.





8251 Interface to 8080 Standard System Bus

## DETAILED OPERATION DESCRIPTION

### General

The complete functional definition of the 8251 is programmed by the systems software. A set of control words must be sent out by the CPU to initialize the 8251 to support the desired communications format. These control words will program the: BAUD RATE, CHARACTER LENGTH, NUMBER OF STOP BITS, SYNCHRONOUS or ASYNCHRONOUS OPERATION, EVEN/ODD PARITY etc. In the Synchronous Mode, options are also provided to select either internal or external character synchronization.

Once programmed, the 8251 is ready to perform its communication functions. The TxRDY output is raised "high" to signal the CPU that the 8251 is ready to receive a character. This output (TxRDY) is reset automatically when the CPU writes a character into the 8251. On the other hand, the 8251 receives serial data from the MODEM or I/O device, upon receiving an entire character the RxRDY output is raised "high" to signal the CPU that the 8251 has a complete character ready for the CPU to fetch. RxRDY is reset automatically upon the CPU read operation.

The 8251 cannot begin transmission until the TxEN(Transmitter Enable) bit is set in the Command Instruction and it has received a Clear To Send (CTS) input. The TxD output will be held in the marking state upon Reset.

### Programming the 8251

Prior to starting data transmission or reception, the 8251 must be loaded with a set of control words generated by the CPU. These control signals define the complete functional definition of the 8251 and must immediately follow a Reset operation (internal or external).

The control words are split into two formats:

1. Mode Instruction
2. Command Instruction

### Mode Instruction

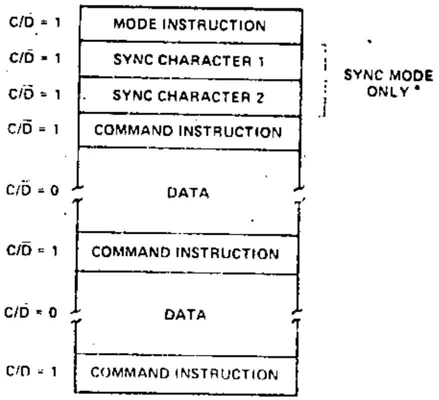
This format defines the general operational characteristics of the 8251. It must follow a Reset operation (internal or external). Once the Mode instruction has been written into the 8251 by the CPU, SYNC characters or Command instructions may be inserted.

### Command Instruction

This format defines a status word that is used to control the actual operation of the 8251.

Both the Mode and Command instructions must conform to a specified sequence for proper device operation. The Mode Instruction must be inserted immediately following a Reset operation, prior to using the 8251 for data communication.

All control words written into the 8251 after the Mode Instruction will load the Command Instruction. Command Instructions can be written into the 8251 at any time in the data block during the operation of the 8251. To return to the Mode Instruction format a bit in the Command Instruction word can be set to initiate an internal Reset operation which automatically places the 8251 back into the Mode Instruction format. Command Instructions must follow the Mode Instructions or Sync characters.



*The second SYNC character is skipped if MODE instruction has programmed the 8251 to single character Internal SYNC Mode. Both SYNC characters are skipped if MODE instruction has programmed the 8251 to ASYNC mode.

Typical Data Block

## Mode Instruction Definition

The 8251 can be used for either Asynchronous or Synchronous data communication. To understand how the Mode Instruction defines the functional operation of the 8251 the designer can best view the device as two separate components sharing the same package. One Asynchronous the other Synchronous. The format definition can be changed "on the fly" but for explanation purposes the two formats will be isolated.

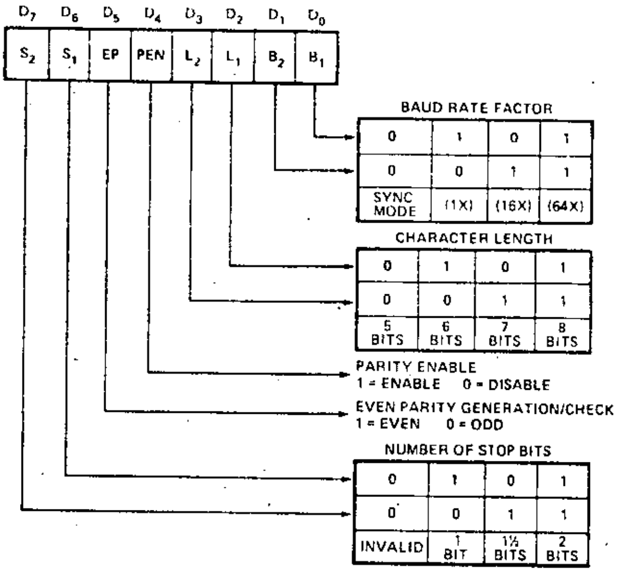
## Asynchronous Mode (Transmission)

Whenever a data character is sent by the CPU the 8251 automatically adds a Start bit (low level) and the programmed number of Stop bits to each character. Also, an even or odd Parity bit is inserted prior to the Stop bit(s), as defined by the Mode Instruction. The character is then transmitted as a serial data stream on the TxD output. The serial data is shifted out on the falling edge of $\overline{TxC}$ at a rate equal to 1, 1/16, or 1/64 that of the $\overline{TxC}$, as defined by the Mode Instruction. BREAK characters can be continuously sent to the TxD if commanded to do so.

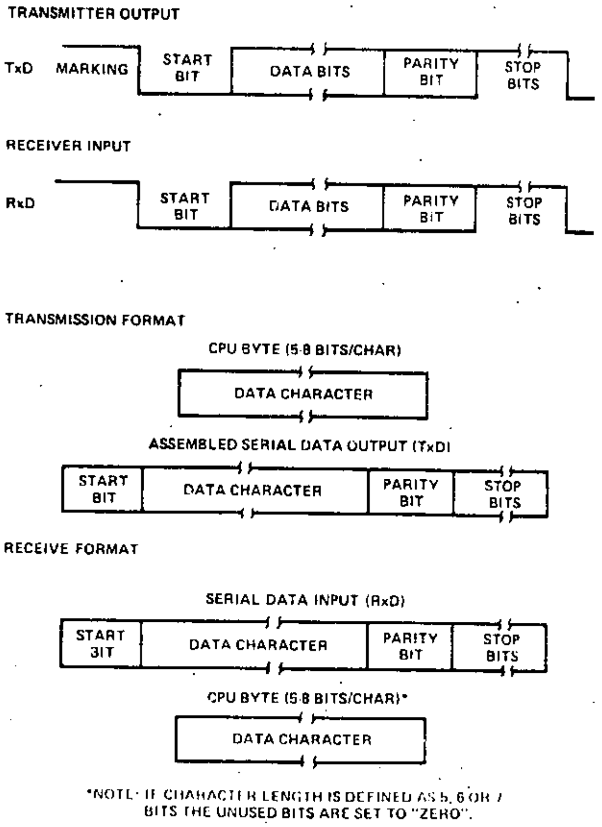
When no data characters have loaded into the 8251 the TxD output remains "high" (marking) unless a Break (continuously low) has been programmed.

## Asynchronous Mode (Receive)

The RxD line is normally high. A falling edge on this line triggers the beginning of a START bit. The validity of this START bit is checked by again strobing this bit at its nominal center. If a low is detected again, it is a valid START bit, and the bit counter will start counting. The bit counter locates the center of the data bits, the parity bit (if it exists) and the stop bits. If parity error occurs, the parity error flag is set. Data and parity bits are sampled on the RxD pin with the rising edge of $\overline{RxC}$. If a low level is detected as the STOP bit, the Framing Error flag will be set. The STOP bit signals the end of a character. This character is then loaded into the parallel I/O buffer of the 8251. The RxRDY pin is raised to signal the CPU that a character is ready to be fetched. If a previous character has not been fetched by the CPU, the present character replaces it in the I/O buffer, and the OVERRUN flag is raised (thus the previous character is lost). All of the error flags can be reset by a command instruction. The occurrence of any of these errors will not stop the operation of the 8251.


$D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$
$S_2$ $S_1$ EP PEN $L_2$ $L_1$ $B_2$ $B_1$
BAUD RATE FACTOR
0 1 0 1
0 0 1 1
SYNC MODE (1X) (16X) (64X)
CHARACTER LENGTH
0 1 0 1
0 0 1 1
5 BITS 6 BITS 7 BITS 8 BITS
PARITY ENABLE
1 = ENABLE 0 = DISABLE
EVEN PARITY GENERATION/CHECK
1 = EVEN 0 = ODD
NUMBER OF STOP BITS
0 1 0 1
0 0 1 1
INVALID 1 BIT 1½ BITS 2 BITS

Mode Instruction Format, Asynchronous Mode



*NOTE: IF CHARACTER LENGTH IS DEFINED AS 5, 6 OR 7 BITS THE UNUSED BITS ARE SET TO "ZERO".

Asynchronous Mode

## Synchronous Mode (Transmission)

The TxD output is continuously high until the CPU sends its first character to the 8251 which usually is a SYNC character. When the $\overline{\text{CTS}}$ line goes low, the first character is serially transmitted out. All characters are shifted out on the falling edge of $\overline{\text{TxC}}$. Data is shifted out at the same rate as the $\overline{\text{TxC}}$.

Once transmission has started, the data stream at TxD output must continue at the $\overline{\text{TxC}}$ rate. If the CPU does not provide the 8251 with a character before the 8251 becomes empty, the SYNC characters (or character if in single SYNC word mode) will be automatically inserted in the TxD data stream. In this case, the TxEMPTY pin is raised high to signal that the 8251 is empty and SYNC characters are being sent out. The TxEMPTY pin is internally reset by the next character being written into the 8251.

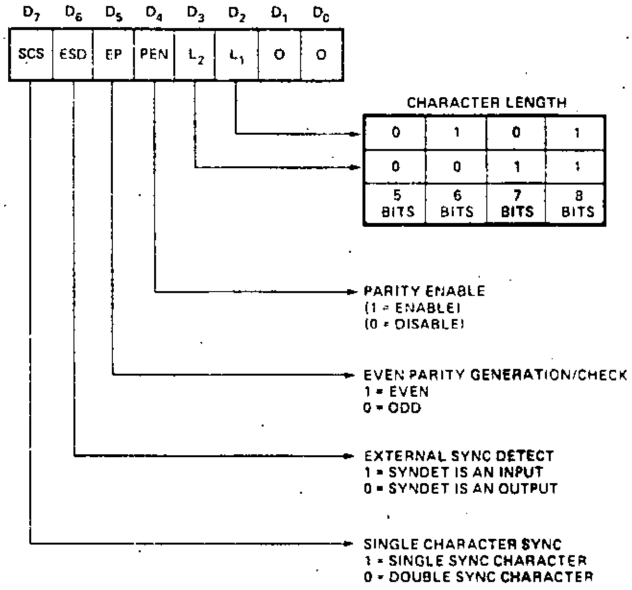
## Synchronous Mode (Receive)

In this mode, character synchronization can be internally or externally achieved. If the internal SYNC mode has been programmed, the receiver starts in a HUNT mode. Data on the RxD pin is then sampled in on the rising edge of $\overline{\text{RxC}}$. The content of the Rx buffer is continuously compared with the first SYNC character until a match occurs. If the 8251 has been programmed for two SYNC characters, the subsequent received character is also compared; when both SYNC characters have been detected, the USART ends the HUNT mode and is in character synchronization. The SYNDET pin is then set high, and is reset automatically by a STATUS READ.

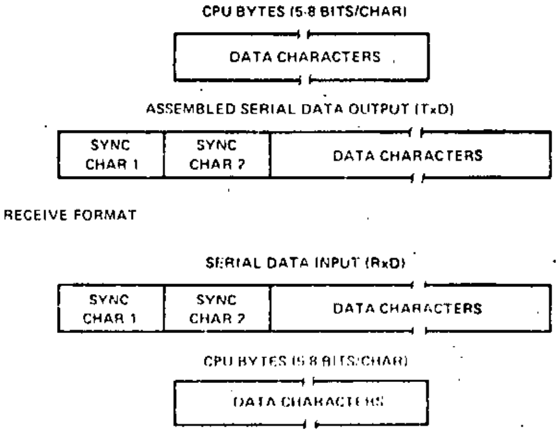
In the external SYNC mode, synchronization is achieved by applying a high level on the SYNDET pin. The high level can be removed after one $\overline{\text{RxC}}$ cycle.

Parity error and overrun error are both checked in the same way as in the Asynchronous Rx mode.

The CPU can command the receiver to enter the HUNT mode if synchronization is lost.



Mode Instruction Format, Synchronous Mode

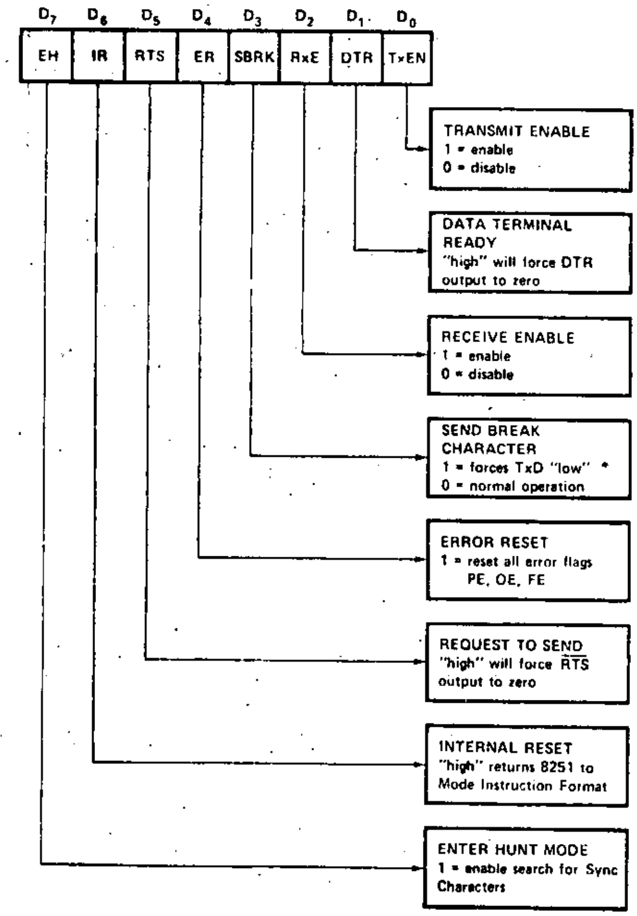


Synchronous Mode, Transmission Format

## COMMAND INSTRUCTION DEFINITION

Once the functional definition of the 8251 has been programmed by the Mode Instruction and the Sync Characters are loaded (if in Sync Mode) then the device is ready to be used for data communication. The Command Instruction controls the actual operation of the selected format. Functions such as: Enable Transmit/Receive, Error Reset and Modem Controls are provided by the Command Instruction.

Once the Mode Instruction has been written into the 8251 and Sync characters inserted, if necessary, then all further "control writes" (C/$\overline{D}$ = 1) will load the Command Instruction. A Reset operation (internal or external) will return the 8251 to the Mode Instruction Format.



$D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$
EH
IR
RTS
ER
SBRK
RxE
DTR
TxEN
TRANSMIT ENABLE 1 = enable 0 = disable
DATA TERMINAL READY "high" will force $\overline{DTR}$ output to zero
RECEIVE ENABLE 1 = enable 0 = disable
SEND BREAK CHARACTER 1 = forces TxD "low" 0 = normal operation
ERROR RESET 1 = reset all error flags PE, OE, FE
REQUEST TO SEND "high" will force $\overline{RTS}$ output to zero
INTERNAL RESET "high" returns 8251 to Mode Instruction Format
ENTER HUNT MODE 1 = enable search for Sync Characters

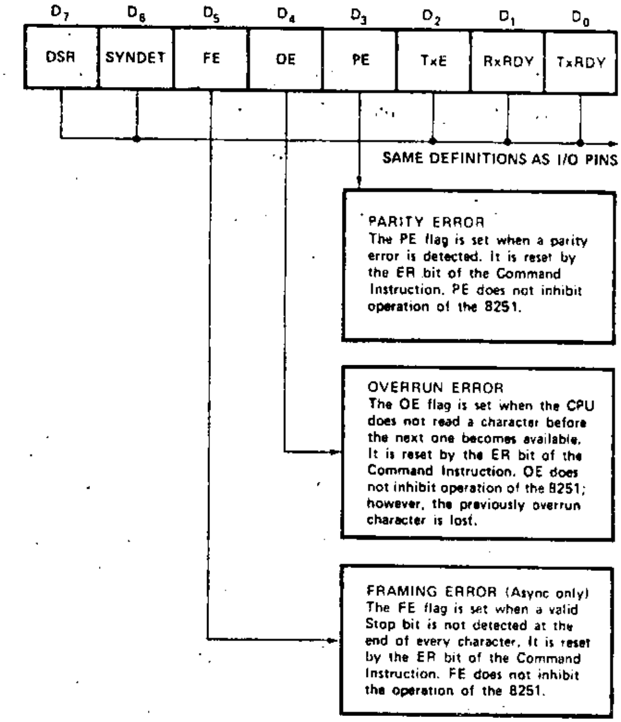
Command Instruction Format

## STATUS READ DEFINITION

In data communication systems it is often necessary to examine the "status" of the active device to ascertain if errors have occurred or other conditions that require the processor's attention. The 8251 has facilities that allow the programmer to "read" the status of the device at any time during the functional operation.

A normal "read" command is issued by the CPU with the C/D input at one to accomplish this function.

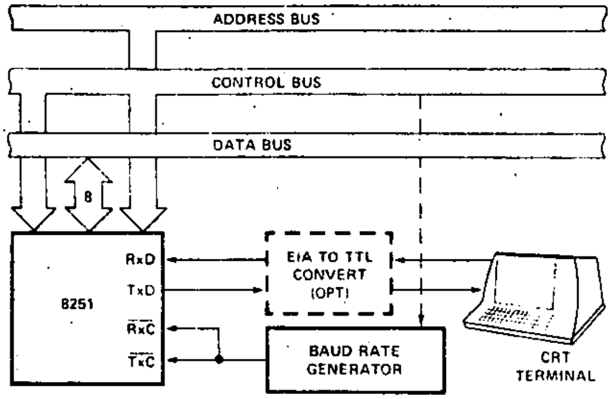
Some of the bits in the Status Read Format have identical meanings to external output pins so that the 8251 can be used in a completely Polled environment or in an interrupt driven environment.



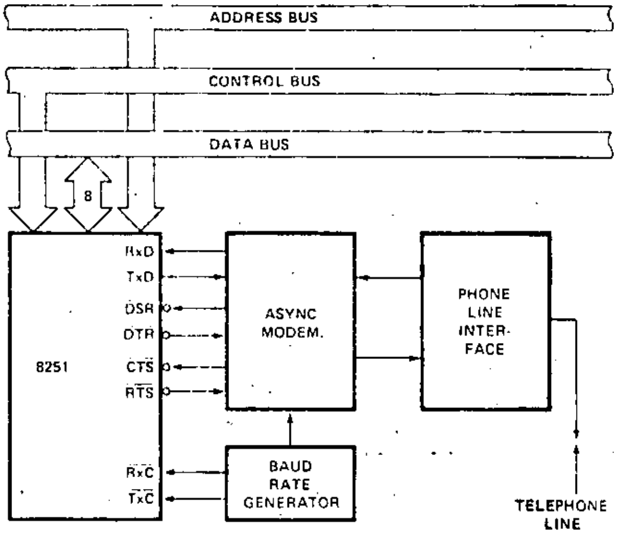
$D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$
DSR
SYNDET
FE
OE
PE
TxE
RxRDY
TxRDY
SAME DEFINITIONS AS I/O PINS
PARITY ERROR The PE flag is set when a parity error is detected. It is reset by the ER bit of the Command Instruction. PE does not inhibit operation of the 8251.
OVERRUN ERROR The OE flag is set when the CPU does not read a character before the next one becomes available. It is reset by the ER bit of the Command Instruction. OE does not inhibit operation of the 8251; however, the previously overrun character is lost.
FRAMING ERROR (Async only) The FE flag is set when a valid Stop bit is not detected at the end of every character. It is reset by the ER bit of the Command Instruction. FE does not inhibit the operation of the 8251.

Status Read Format

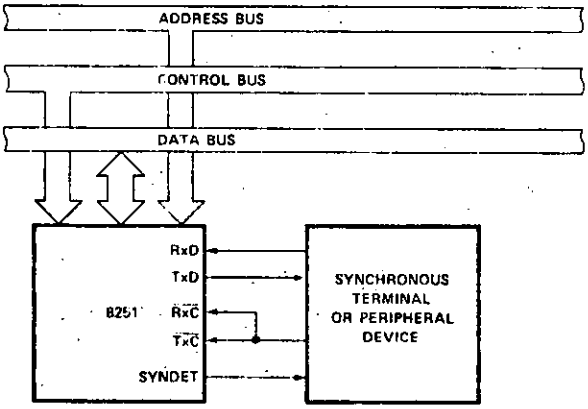
## APPLICATIONS OF THE 8251



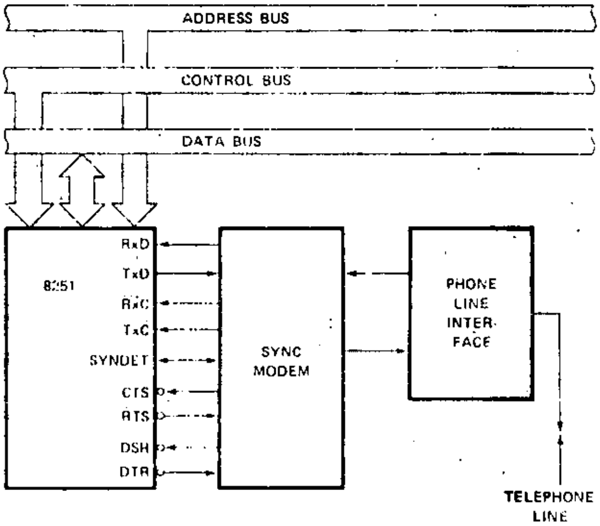
**Asynchronous Serial Interface to CRT Terminal, DC-9600 Baud**



**Asynchronous Interface to Telephone Lines**



**Synchronous Interface to Terminal or Peripheral Device**



**Synchronous Interface to Telephone Lines**

# Schottky Bipolar 8214

## PRIORITY INTERRUPT CONTROL UNIT

- **Eight Priority Levels**
- **Current Status Register**
- **Priority Comparator**
- **Fully Expandable**
- **High Performance (50ns)**
- **24-Pin Dual In-Line Package**

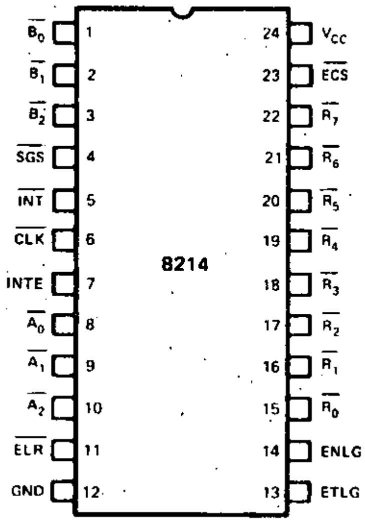
The 8214 is an eight level priority interrupt control unit designed to simplify interrupt driven microcomputer systems.

The PICU can accept eight requesting levels; determine the highest priority, compare this priority to a software controlled current status register and issue an interrupt to the system along with vector information to identify the service routine.

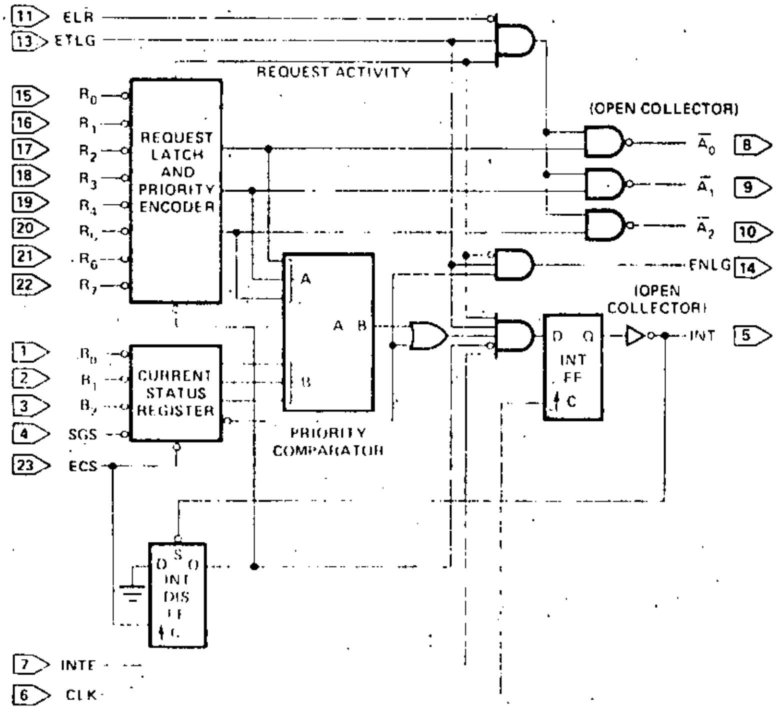
The 8214 is fully expandable by the use of open collector interrupt output and vector information. Control signals are also provided to simplify this function.

The PICU is designed to support a wide variety of vectored interrupt structures and reduce package count in interrupt driven microcomputer systems.

### PIN CONFIGURATION

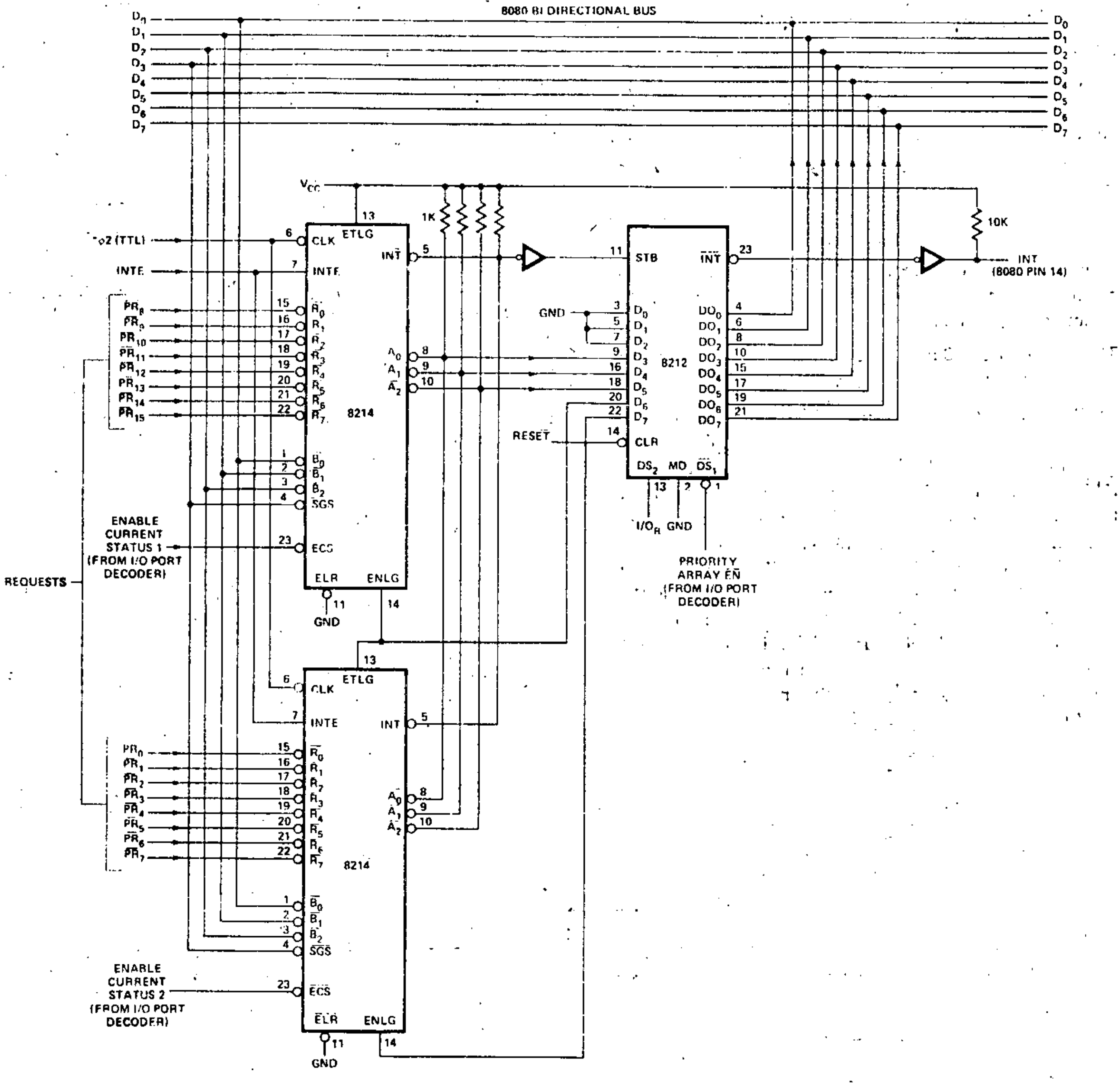


### LOGIC DIAGRAM



### PIN NAMES

| | | |
|---|---|---|
| INPUTS | | |
| $\overline{R_0}$-$\overline{R_7}$ | REQUEST LEVELS ($R_7$ HIGHEST PRIORITY) | |
| $\overline{B_0}$-$\overline{B_2}$ | CURRENT STATUS | |
| $\overline{SGS}$ | STATUS GROUP SELECT | |
| $\overline{ECS}$ | ENABLE CURRENT STATUS | |
| INTE | INTERRUPT ENABLE | |
| $\overline{CLK}$ | CLOCK (INT F-F) | |
| $\overline{ELR}$ | ENABLE LEVEL READ | |
| ETLG | ENABLE THIS LEVEL GROUP | |
| OUTPUTS: | | |
| $\overline{A_0}$-$\overline{A_2}$ | REQUEST LEVELS | OPEN COLLECTOR |
| $\overline{INT}$ | INTERRUPT (ACT. LOW) | |
| ENLG | ENABLE NEXT LEVEL GROUP | |



16 Level Controller

# Silicon Gate MOS 8257

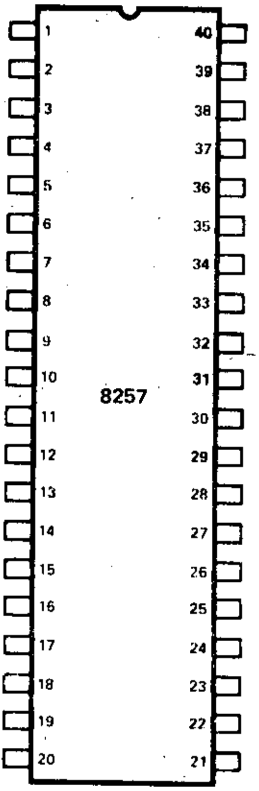
## PROGRAMMABLE DMA CONTROLLER

- Four Channel DMA Controller
- Priority DMA Request Logic
- Channel Inhibit Logic
- Terminal and Modulo 256/128 Outputs
- Auto Load Mode
- Single TTL Clock (Ø2/TTL)
- Single +5V Supply
- Expandable
- 40 Pin Dual-in-Line Package

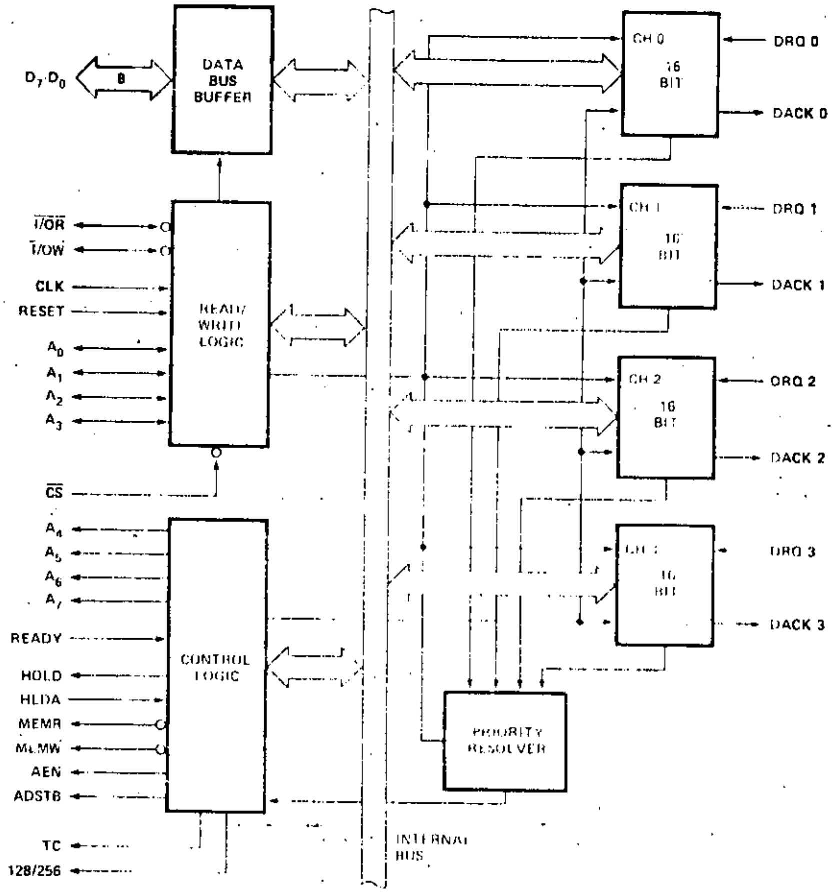
The 8257 is a Direct Memory Access (DMA) Chip which has four channels for use in 8080 microcomputer systems. Its primary function is to generate, upon a peripheral request, a sequential memory address which will allow the peripheral to access or deposit data directly from or to memory. It uses the Hold feature of the 8080 to acquire the system bus. It also keeps count of the number of DMA cycles for each channel and notifies the peripheral when a programmable terminal count has been reached. Other features that it has are two mode priority logic to resolve the request among the four channels, programmable channel inhibit logic, an early write pulse option, a modulo 256/128 Mark output for sectored data transfers, an automatic load mode, a terminal count status register, and control signal timing generation during DMA cycles. There are three types of DMA cycles: Read DMA Cycle, Write DMA Cycle and Verify DMA Cycle.

The 8257 is a 40-pin, N-channel MOS chip which uses a single +5V supply and the o2 (TTL) clock of the 8080 system. It is designed to work in conjunction with a single 8212 8-bit, three-state latch chip. Multiple DMA chips can be used to expand the number of channels with the aid of the 8214 Priority Interrupt Chip.

PIN CONFIGURATION



BLOCK DIAGRAM



## 8257 PRELIMINARY FUNCTIONAL DESCRIPTION

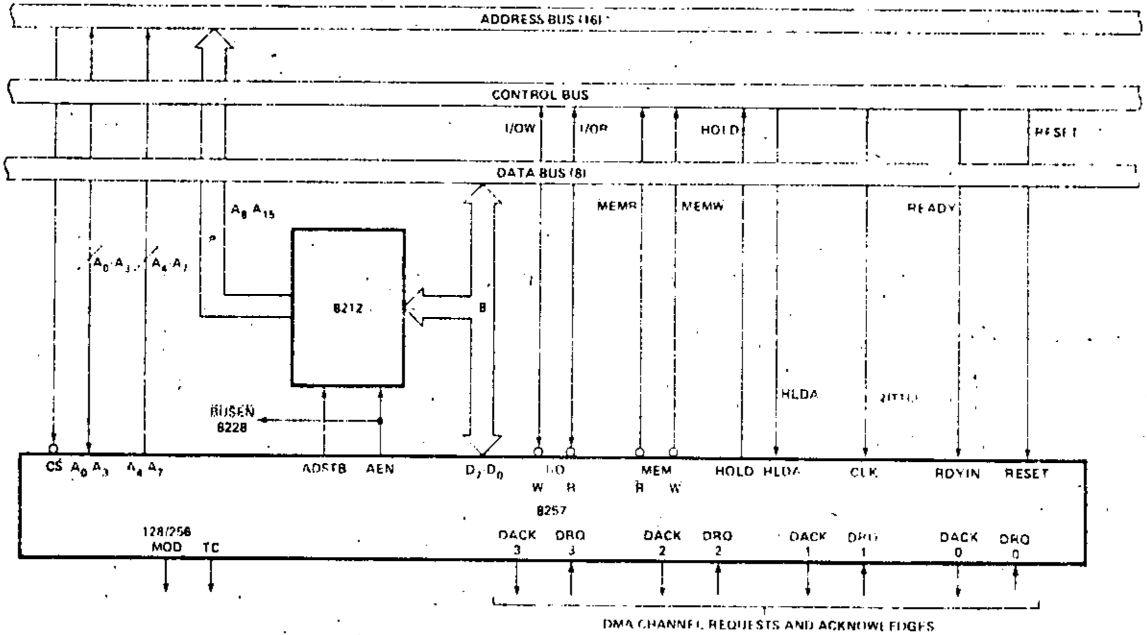
The transfer of data between a mass storage device such as a floppy disk or mag cassette and system RAM memory is often limited by the speed of the microprocessor. Removing the processor during such a transfer and letting an auxillary device manage the transfer in a more efficient manner would greatly improve the speed and make mass storage devices more attractive, even to the small system designer.

The transfer technique is called DMA (Direct Memory Access); in essence the CPU is idled so that it no longer has control of the system bus and a DMA controller takes over to manage the transfer.

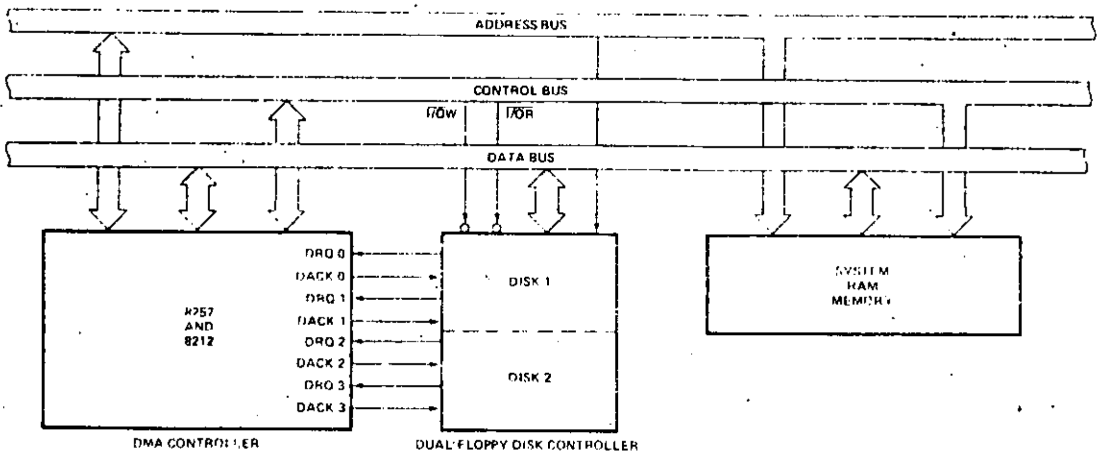
The 8257 Programmable DMA Controller is a single chip, four channel device that can efficiently manage DMA activities. Each channel is assigned a priority level so that if multi-DMA activities are required each mass storage device can be serviced, based on its importance in the system. In operation, a request is made from a peripheral device for access to the system bus. After its priority is accepted a HOLD command is ussued to the CPU, the CPU issues a HLDA and that DMA channel has complete control of the system bus. Transfers can be made in blocks, suspending the processors operation during the entire transfer or, the transfer can be made a few bytes at a time, hidden in the execution states of each instruction cycle, (cycle-stealing).

The modes and priority resolving are maintained by the system software as well as initializing each channel as to the starting address and length of transfer.

The system interface is similar to the other peripherals of the MCS-80 but an additional 8212 is necessary to control the entire address bus. A special control signal $\overline{\text{BUSEN}}$ is connected directly to the 8228 so that the data bus and control bus will be released at the proper time.



**System Interface 8257.**



**System Application of 8257.**

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES : TEORIA Y APLICACIONES

# THE 8080 INSTRUCTION SET

ING. MARIO RODRIGUEZ

MARZO,1979.

CHAPTER 2
THE 8080
INSTRUCTION SET

This section describes the 8080 assembly language instruction set.

For the reader who understands assembly language programming, Appendix A provides a complete summary of the 8080 instructions.

For the reader who is not completely familiar with assembly language, Chapter 2 describes individual instructions with examples and machine code equivalents.

## ASSEMBLY LANGUAGE

### How Assembly Language is Used

Upon examining the contents of computer memory, a program would appear as a sequence of hexadecimal digits, which are interpreted by the CPU as instruction codes, addresses, or data. It is possible to write a program as a sequence of digits (just as they appear in memory), but that is slow and expensive. For example, many instructions reference memory to address either a data byte or another instruction:

| Hexadecimal Memory Address | |
|---|---|
| 1432 | 7E |
| 1433 | C3 |
| 1434 | C4 |
| 1435 | 14 |
| 1436 | |
| • | |
| • | |
| 14C3 | FF |
| 14C4 | 2E |
| 14C5 | 36 |
| 14C6 | 77 |

Assuming that registers H and L contain 14H and C3H respectively, the program operates as follows:

Byte 1432 specifies that the accumulator is to be loaded with the contents of byte 14C3.

Bytes 1433 through 1435 specify that execution is to continue with the instruction starting at byte 14C4.

Bytes 14C4 and 14C5 specify that the L register is to be loaded with the number 36H.

Byte 14C6 specifies that the contents of the accumulator are to be stored in byte 1436.

Now suppose that an error discovered in the program logic necessitates placing an extra instruction after byte 1432. Program code would have to change as follows:

| Hexadecimal Memory Address | Old Code | New Code |
|---|---|---|
| 1432 | 7E | 7E |
| 1433 | C3 | New Instruction |
| 1434 | C4 | C3 |
| 1435 | 14 | C5 |
| 1436 | • | 14 |
| 1437 | • | • |
| 14C3 | FF | • |
| 14C4 | 2E | FF |
| 14C5 | 36 | 2E |
| 14C6 | 77 | 37 |
| 14C7 | | 77 |

Most instructions have been moved and as a result many must be changed to reflect the new memory addresses of instructions or data. The potential for making mistakes is very high and is aggravated by the complete unreadability of the program.

Writing programs in assembly language is the first and most significant step towards economical programming; it provides a readable notation for instructions, and separates the programmer from a need to know or specify absolute memory addresses.

Assembly language programs are written as a sequence of instructions which are converted to executable hexadecimal code by a special program called an ASSEMBLER. Use of the 8080 assembler is described in its operator's manual.

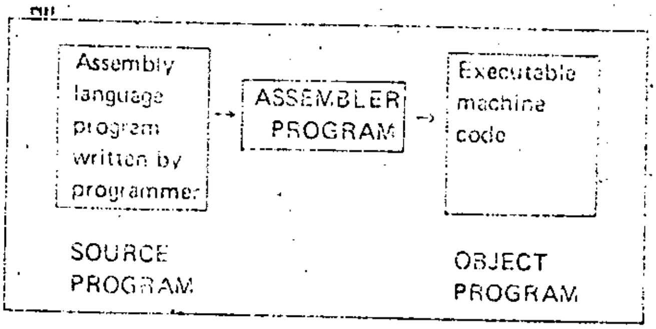


**Figure 2-1. Assembler Program Converts Assembly Language Source Program to Object Program**

As illustrated in Figure 2-1, the assembly language program generated by a programmer is called a SOURCE PROGRAM. The assembler converts the SOURCE PROGRAM into an equivalent OBJECT PROGRAM, which consists of a sequence of binary codes that can be loaded into memory and executed.

For example:

```
                                        One Possible
                                        Version of the
Source Program                          Object Program

NOW: MOV   A,B                           78
     CPI   'C'   → is converted      →   FE43
     JZ    LER     by the                CA7C3D
      :            Assembler              :
LER: MOV   M,A     to                    77
```

NOTE: In this and subsequent examples, it is not necessary to understand the operations of the individual instructions. They are presented only to illustrate typical assembly language statements. Individual instructions are described later in this chapter.

Now if a new instruction must be added, only one change is required. Even the reader who is not yet familiar with assembly language will see how simple the addition is:

```
NOW:        MOV          A,B
       (New instruction inserted here)
            CPI          'C'
            JZ           LER

LER         MOV          M,A
```

The assembler takes care of the fact that a new instruction will shift the rest of the program in memory.

## Statement Syntax

Assembly language instructions must adhere to a fixed set of rules as described in this section. An instruction has four separate and distinct parts or fields.

Field 1 is the LABEL field. It is a name used to reference the instruction's address.

Field 2 is the CODE field. It specifies the operation that is to be performed.

Field 2 is the OPERAND field. It provides any address or data information needed by the CODE field.

Field 4 is the COMMENT field. It is present for the programmer's convenience and is ignored by the assembler. The programmer uses comment fields to describe the operation and thus make the program more readable.

The assembler uses free fields; that is, any number of blanks may separate fields.

Before describing each field in detail, here are some general examples:

```
Label      Code    Operand
HERE:      MVI     C,0      ; Load the C register with 0
THERE:     DB      3AH      ; Create a one-byte data
                            ; constant
LOOP:      ADD     E        ; Add contents of E register
                              to the accumulator
           RLC              ; Rotate the accumulator left
```

NOTE: These examples and the ones which follow are intended to illustrate how the various fields appear in complete assembly language statements. It is not necessary at this point to understand the operations which the statements perform.

## Label Field

This is an optional field, which, if present, may be from 1 to 5 characters long. The first character of the label must be a letter of the alphabet or one of the special characters @ (at sign) or ? (question mark). A colon (:) must follow the last character. (The operation codes, pseudo-instruction names, and register names are specially defined within the assembler and may not be used as labels. Operation codes and pseudo-instructions are given later in this chapter and Appendix A.

Here are some examples of valid label fields:

```
LABEL:
F14F:
@HERE:
?ZERO:
```

Here are some invalid label fields:

| | |
|---|---|
| 123: | begins with a decimal digit' |
| LABEL | is not followed by a colon |
| ADD: | is an operation code |
| END: | is a pseudo-instruction |

The following label has more than five characters; only the first five will be recognized:

```
INSTRUCTION:    will be read as INSTR:
```

Since labels serve as instruction addresses, they cannot be duplicated. For example, the sequence:

```
HERE:      JMP      THERE
           ---
           ---
THERE:     MOV      C,D
           ---
           ---
THERE:     CALL     SUB
```

is ambiguous; the assembler cannot determine which address is to be referenced by the JMP instruction.

One instruction may have more than one label, however. The following sequence is valid:

```
LOOP1:                          ; First label
LOOP2:     MOV      C,D         ; Second label
           ---
           JMP      LOOP1
           ---
           JMP      LOOP2
```

Each JMP instruction will cause program control to be transferred to the same MOV instruction.

## Code Field

This field contains a code which identifies the machine operation (add, subtract, jump, etc.) to be performed: hence the term operation code or op code. The instructions described later in this chapter are each identified by a mnemonic label which must appear in the code field. For example, since the "jump" instruction is identified by the letters "JMP," these letters must appear in the code field to identify the instruction as "jump."

There must be at least one space following the code field. Thus,

```
HERE:      JMP      THERE
```

is legal, but:

```
HERE       JMPTHERE
```

is illegal.

## Operand Field

This field contains information used in conjunction with the code field to define precisely the operation to be performed by the instruction. Depending upon the code field, the operand field may be absent or may consist of one item or two items separated by a comma.

There are four types of information [(a) through (d) below] that may be requested as items of an operand field, and the information may be specified in nine ways [(1) through (9) below], as summarized in the following table and described in detail in the subsequent examples.

OPERAND FIELD INFORMATION

| Information required | Ways of specifying |
|---|---|
| (a) Register | (1) Hexadecimal Data |
| (b) Register Pair | (2) Decimal Data |
| (c) Immediate Data | (3) Octal Data |
| (d) 16-bit Memory Address | (4) Binary Data |
| | (5) Program Counter ($) |
| | (6) ASCII Constant |
| | (7) Labels assigned values |
| | (8) Labels of instructions |
| | (9) Expressions |

The nine ways of specifying information are as follows:

(1) Hexadecimal data. Each hexadecimal number must be followed by a letter 'H' and *must* begin with a numeric digit (0-9),

Example:

```
Label     Code   Operand    Comment
HERE:     MVI    C,0BAH     ; Load register C with the
                            ; hexadecimal number BA
```

(2) Decimal data. Each decimal number may optionally be followed by the letter 'D,' or may stand alone.

Example:

```
Label     Code   Operand    Comment
ABC:      MVI    E,105      ; Load register E with 105
```

(3) Octal data. Each octal number must be followed by one of the letters 'O' or 'Q.'

Example:

```
Label     Code   Operand    Comment
LABEL:    MVI    A,72O      ; Load the accumulator with
                            ; the octal number 72
```

(4) Binary data. Each binary number must be followed by the letter 'B.'

Example:

```
Label   Code  Operand                 Comment
TWO:    MVI   10B,11110110B           ; Load register two
                                      ; (the D register) with
                                      ; 0F6H
JUMP:   JMP   0010111011111010B       ; Jump to
                                      ; memory
                                      ; address 2EFA
```

(5) The current program counter. This is specified as the character '$' and is equal to the address of the current instruction.

Example:

```
Label        Code         Operand
GO:          JMP          $+6
```

The instruction above causes program control to be transferred to the address 6 bytes beyond where the JMP instruction is loaded.

(6) An ASCII constant. This is one or more ASCII characters enclosed in single quotes. Two successive single quotes must be used to represent one single quote within an ASCII constant. Appendix D contains a list of legal ASCII characters and their hexadecimal representations.

Example:

```
Label   Code  Operand  Comment
CHAR:   MVI   E,'*'    ;Load the E register with the
                       ; eight-bit ASCII representa-
                       ; tion of an asterisk
```

(7) Labels that have been assigned a numeric value by the assembler. The following assignments are built into the assembler and are therefore always active:

| | | | | | |
|---|---|---|---|---|---|
| B | assigned to | 0 | representing | register | B |
| C | " | " 1 | " | " | C |
| D | " | " 2 | " | " | D |
| E | " | " 3 | " | " | E |
| H | " | " 4 | " | " | H |
| L | " | " 5 | " | " | L |
| M | " | " 6 | " | a memory reference | |
| A | " | " 7 | " | register | A |

Example:

Suppose VALUE has been equated to the hexadecimal number 9FH. Then the following instructions all load the D register with 9FH:

```
Label         Code          Operand
A1:           MVI           D, VALUE
A2:           MVI           2, 9FH
A3:           MVI           2, VALUE
```

(8) Labels that appear in the label field of another instruction.

Example:

```
Label    Code   Operand   Comment
HERE:    JMP    THERE     ; Jump to instruction
                          ; at THERE
         ---
         ---
THERE:   MVI    D, 9FH
```

(9) Arithmetic and logical expressions involving data types (1) to (8) above connected by the arithmetic operators (+) (addition), - (unary minus and subtraction), * (multiplication), / (division), MOD (modulo), the logical operators NOT, AND, OR, XOR, SHR (shift right), SHL (shift left), and left and right parentheses.

All operators treat their arguments as 15-bit quantities, and generate 16-bit quantities as their result.

The operator + produces the arithmetic sum of its operands.

The operator - produces the arithmetic difference of its operands when used as subtraction, or the arithmetic negative of its operand when used as unary minus.

The operator * produces the arithmetic product of its operands.

The operator / produces the arithmetic integer quotient of its operands, discarding any remainder.

The operator MOD produces the integer remainder obtained by dividing the first operand by the second.

The operator NOT complements each bit of its operand.

The operator AND produces the bit-by-bit logical AND of its operands.

The operator OR produces the bit-by-bit logical OR of its operands.

The operator XOR produces the bit-by-bit logical EXCLUSIVE-OR of its operands.

The SHR and SHL operators are linear shifts which shift their first operands right or left, respectively, by the number of bit positions specified by their second operands. Zeros are shifted into the high-order or low-order bits, respectively, of their first operands.

The programmer must insure that the result generated by any operation fits the requirements of the operation being coded. For example, the second operand of an MVI

Example: What is the value of 86H interpreted as a signed two's complement number? The high-order bit is set, indicating that this is a negative number. To obtain its value, again complement each bit and add one.

86H = 1 0 0 0 0 1 1 0 B
Complement each bit : 0 1 1 1 1 0 0 1 B
Add one : 0 1 1 1 1 0 1 0 B

Thus, the value of 86H is -7AH = -122D.

The range of negative numbers that can be represented in signed two's complement notation is from -1 to -128.

-1 = 1 1 1 1 1 1 1 1 B = FFH
-2 = 1 1 1 1 1 1 1 0 B = FEH
•
•
•
-127D = 1 0 0 0 0 0 0 1 B = 81H
-128D = 1 0 0 0 0 0 0 0 B = 80H

To perform the subtraction 1AH-0CH, the following operations are performed:

Take the two's complement of 0CH=F4H

Add the result to the minuend:

1AH = 0 0 0 1 1 0 1 0
+(-0CH) = F4H = 1 1 1 1 0 1 0 0
0 0 0 0 1 1 1 0 = 0EH the correct answer

When a byte is interpreted as an unsigned two's complement number, its value is considered positive and in the range 0 to $255_{10}$:

0 = 0 0 0 0 0 0 0 0 B = 0H
1 = 0 0 0 0 0 0 0 1 B = 1H
•
•
127D = 0 1 1 1 1 1 1 1 B = 7FH
128D = 1 0 0 0 0 0 0 0 B = 80H
•
•
255D = 1 1 1 1 1 1 1 1 B = FFH

Two's complement arithmetic is still valid. When performing an addition operation, the Carry bit is set when the result is greater than 255D. When performing subtraction, the Carry bit is reset when the result is positive. If the Carry bit is set, the result is negative and present in its two's complement form. *Thus, the Carry bit when set indicates the occurrence of a "borrow."*

Example: Subtract 98D from 197D using unsigned two's complement arithmetic.

197D = 1 1 0 0 0 1 0 1 = C5H
-98D = 1 0 0 1 1 1 1 0 = 9EH
carry out → 1 0 1 1 0 0 0 1 1 = 63H = 99D

Since the carry out of bit 7 = 1, indicating that the answer is correct and positive, the subtract operation will reset the Carry bit to 0.

Example: Subtract 15D from 12D using unsigned two's complement arithmetic.

12D = 0 0 0 0 1 1 0 0 = 0CH
-15D = 1 1 1 1 0 0 0 1 = 0F1H
carry out → 0 1 1 1 1 1 1 0 1 = -3D

Since the carry out of bit 7 = 0, indicating that the answer is negative and in its two's complement form, the subtract operation will set the Carry bit indicating that a "borrow" occurred.

NOTE: The 8080 instructions which perform the subtraction operation are SUB, SUI, SBB, SBI, CMP, and CMI. Although the same result will be obtained by addition of a complemented number or subtraction of an uncomplemented number, the resulting Carry bit will be different.

EXAMPLE: If the result -3 is produced by performing an "ADD" operation on the numbers +12D and -15D, the Carry bit will be reset; if the same result is produced by performing a "SUB" operation on the numbers +12D and +15D, the Carry bit will be set. Both operations indicate that the result is negative; the programmer must be aware which operations set or reset the Carry bit.

"ADD" +12D and -15D

+12D = 0 0 0 0 1 1 0 0
+(-15D) = 1 1 1 1 0 0 0 1
0 1 1 1 1 1 1 0 1 = -3D
causes carry to be reset

"SUB" +15D from +12D

+12D = 0 0 0 0 1 1 0 0
-(+15D) = 1 1 1 1 0 0 0 1
1 1 1 1 1 1 1 0 1 = -3D
causes carry to be set

## DB Define Byte(s) of Data

| Label | Code | Operand |
|---|---|---|
| oplab: | DB | list |

"list" is a list of either:

(1) Arithmetic and logical expressions involving any of the arithmetic and logical operators, which evaluate to eight-bit data quantities

(2) Strings of ASCII characters enclosed in quotes

Description: The eight-bit value of each expression, or the eight-bit ASCII representation of each character is stored in the next available byte of memory starting with the byte addressed by "oplab." (The most significant bit of each ASCII character is always = 0).

Example:

| Instruction | Assembled Data (hex) |
|---|---|
| HERE: DB 0A3H | A3 |
| WORD1: DB 5*2, 2FH-0AH | 0A25 |
| WORD2: DB 5ABCH SHR 8 | 5A |
| STR: DB 'STRINGSpl' | 535452494E472031 |
| MINUS: DB -03H | FD |

NOTE: In the first example above, the hexadecimal value A3 must be written as 0A3 since hexadecimal numbers must start with a decimal digit.

## DW Define Word (Two Bytes) of Data

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | DW | list |

"list" is a list of expressions which evaluate to 16 bit data quantities.

Description: The least significant 8 bits of the expression are stored in the lower address memory byte (oplab), and the most significant 8 bits are stored in the next higher addressed byte (oplab +1). This reverse order of the high and low address bytes is normally the case when storing addresses in memory. This statement is usually used to create address constants for the transfer-of-control instructions; thus LIST is usually a list of one or more statement labels appearing elsewhere in the program.

Examples:

Assume COMP address memory location 3B1CH and FILL addresses memory location 3EB4H.

| Instruction | | | Assembled Data (hex) |
|---|---|---|---|
| ADD1: | DW | COMP | 1C3B |
| ADD2: | DW | FILL | B43E |
| ADD3: | DW | 3C01H, 3CAEH | 013CAE3C |

Note that in each case, the data are stored with the least significant 8 bits first.

## DS Define Storage (Bytes)

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | DS | exp |

"exp" is a single arithmetic or logical expression.

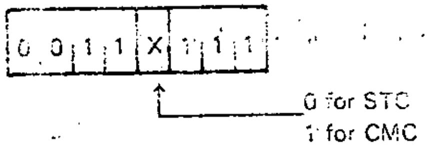
Description: The value of EXP specifies the number of memory bytes to be reversed for data storage. No data values are assembled into these bytes: in particular the programmer should not assume that they will be zero, or any other value. The next instruction will be assembled at memory location oplab+EXP (oplab+10 or oplab+16 in the example below).

Examples:

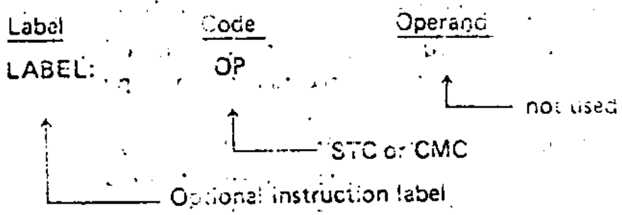
```
HERE:    DS     10    ; Reserve the next 10 bytes
         DS     10H   ; Reserve the next 16 bytes
```

## CARRY BIT INSTRUCTIONS

This section describes the instructions which operate directly upon the Carry bit. Instructions in this class occupy one byte as follows:



The general assembly language format is:



### CMC Complement Carry

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | CMC | — |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

Description: If the Carry bit = 0, it is set to 1. If the Carry bit = 1, it is reset to 0.

Condition bits affected: Carry

### STC Set Carry

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | STC | — |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |

Description: The Carry bit is set to one.

Condition bits affected: Carry

## SINGLE REGISTER INSTRUCTIONS

This section describes instructions which operate on a single register or memory location. If a memory reference is specified, the memory byte addressed by the H and L registers is operated upon. The H register holds the most significant 8 bits of the address while the L register holds the least significant 8 bits of the address.

## OUT Output

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | OUT | exp |

| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | exp |
|---|---|---|---|---|---|---|---|---|

Description: The contents of the accumulator are sent to output device number exp.

Condition bits affected: None

Example:

| Label | Code | Operand | Comment |
|---|---|---|---|
| | OUT | 10 | ; Write the contents of the<br>; accumulator to output<br>; device # 10 |
| | OUT | 1FH | ; Write the contents of the<br>; accumulator to output<br>; device # 31 |

## HLT HALT INSTRUCTION

This section describes the HLT instruction, which occupies one byte.

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | HLT | |

not used

| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

Description: The program counter is incremented to the address of the next sequential instruction. The CPU then enters the STOPPED state and no further activity takes place until an interrupt occurs.

## PSEUDO — INSTRUCTIONS

This section describes pseudo-instructions recognized by the assembler. A pseudo-instruction is written in the same fashion as the machine instructions described earlier in this chapter, but does not cause any object code to be generated. It acts merely to provide the assembler with information to be used subsequently while generating object code.

The general assembly language format of a pseudo instruction is:

| Label | Code | Operand | Comment |
|---|---|---|---|
| name | op | opnd | |

Operand, may be optional

ORG,EQU,SET,END,IF,ENDIF,MACRO, ENDM

name may be required, option, or illegal

NOTE: Names on pseudo-instructions are not followed by a colon, as are labels. *Names* are required in the label field of MACRO, EQU, and SET pseudo-instructions. The label fields of the remaining pseudo-instructions may contain optional labels, exactly like the labels on machine instructions. In this case, the label refers to the memory location immediately following the last previously assembled machine instruction.

## ORG Origin

Format:

| Label | Code | Operand |
|---|---|---|
| oplab: | ORG | exp |

A 16-bit address

Description: The assembler's location counter is set to the value of exp, which must be a valid 16-bit memory address. The next machine instruction or data byte(s) generated will be assembled at address exp, exp+1, etc.

If no ORG appears before the first machine instruction or data byte in the program, assembly will begin at location 0.

Example 1:

| Hex Memory Address | Label | Code | Operand | Assembled Data |
|---|---|---|---|---|
| | | ORG | 1000H | |
| 1000 | | MOV | A,C | 79 |
| 1001 | | ADI | 2 | C602 |
| 1003 | | JMP | NEXT | C35010 |
| | HERE: | ORG | 1050H | |
| 1050 | NEXT: | XRA | A | AF |

The first ORG pseudo-instruction informs the assembler that the object program will begin at memory address 1000H. The second ORG tells the assembler to set its location counter to 1050H and continue assembling machine instructions or data bytes from that point. The label HERE refers to memory location 1006H, since this is the address immediately following the jump instruction. Note that the range of memory from 1006H to 104FH is still included in the object program, but does not contain assembled data. In particular, the programmer should not assume that these locations will contain zero, or any other value.

Example 2:

The ORG pseudo-instruction can perform a function equivalent to the DS (define storage) instruction (see the section on DS earlier in this chapter). The following two sections of code are exactly equivalent:

| Memory Address | Label | Code | Operand | Label | Code | Operand | Asmbl. Data |
|---|---|---|---|---|---|---|---|
| 2C00 | | MOV | A,C | | MOV | A,C | 79 |
| 2C01 | | JMP | NEXT | | JMP | NEXT | C3102C |
| 2C04 | | DS | 12 | | ORG | S+12 | |
| 2C10 | NEXT: | XRA | A | NEXT: | XRA | A | AF |

## EQU Equate

Format:

| Label | Code | Operand |
|---|---|---|
| name | EQU | exp |

An expression

*Required* name

Description: The symbol "name" is assigned the value by EXP by the assembler. Whenever the symbol "name" is encountered subsequently in the assembly, this value will be used.

NOTE: A symbol may appear in the name field or only one EQU pseudo-instruction; i.e., an EQU symbol may not be redefined.

Example:

| Label | Code | Operand | Assembled Data |
|---|---|---|---|
| PTO | EQU | 8 | |
| | | . | |
| | | . | |
| | | . | |
| | OUT | PTO | D308 |

The OUT instruction in this example is equivalent to the statement:

OUT 8

If at some later time the programmer wanted the name PTO to refer to a different output port, it would be necessary only to change the EQU statement, not every OUT statement.

## SET

Format:

| Label | Code | Operand |
|---|---|---|
| name | SET | exp |

An expression

*Required* name

Description: The symbol "name" is assigned the value of exp by the assembler. Whenever the symbol "name" is encountered subsequently in the assembly, this value will be used unless changed by another SET instruction.

This is identical to the EQU equation, except that symbols may be defined more than once.

Example 1:

| Label | Code | Operand | Assembled Data |
|---|---|---|---|
| IMMED | SET | 5 | |
| | ADI | IMMED | C605 |
| IMMED | SET | 10H-6 | |
| | ADI | IMMED | C60A |

Example 2:

Before every assembly, the assembler performs the following SET statements:

| Label | Code | Operand |
|---|---|---|
| B | SET | 0 |
| C | SET | 1 |
| D | SET | 2 |
| E | SET | 3 |
| H | SET | 4 |
| L | SET | 5 |
| M | SET | 6 |
| A | SET | 7 |

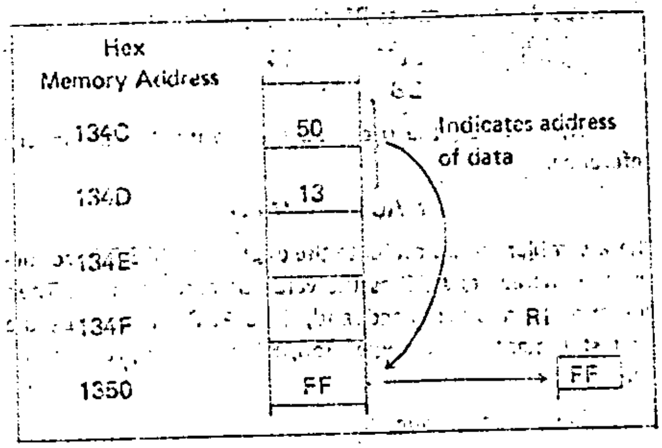
If this were not done, a statement like:

MOV D,A

would be invalid, forcing the programmer to write:

MOV 2,7

Example:



If the address of INADD is 134CH, register RI will be loaded from the address held in memory locations 134CH and 134DH, which is 1350H.

Macro definition:

```
Label   Code    Operand     Comment
LIND    MACRO   RI, INADD
        LHLD    INADD       ; Load indirect address
                            ; into H and L registers
        MOV     RI, M       ; Load data into RI
        ENDM
```

Macro reference:

```
Label           Code            Operand
; Load register C indirect with the contents of memory
; location LABEL.
                LIND            C, LABEL
```

Macro expansion:

```
Label           Code            Operand
                LHLD            LABEL
                MOV             C, M
```

## Other Indirect Addressing Macros

Refer to the LIND macro definition in the last section. Only the MOV RI,M instruction need be altered to create any other indirect addressing macro. For example, substituting MOV M,RI will create a "store indirect" macro. Providing RI is the accumulator, substituting ADD M will create an "add to accumulator indirect" macro.

As an alternative to having load-indirect, store-indirect, and other such indirect macros, we could have a "create indirect address" macro, followed by selected instructions. This alternative approach is illustrated for indexed addressing in the next section.

## Create Indexed Address Macro

The following macro, IXAD, loads registers H and L with the base address ESADD, plus the 16-bit index formed by register pair RP (RP=B,D,H, or SP).

Macro definition:

```
Label   Code    Operand     Comment
IXAD    MACRO   RP, BSADD
        LXI     H, BSADD    ; Load the base address,
        DAD     RP          ; Add index to base
                            ; address
        ENDM
```

Macro reference:

```
Label           Code            Operand
; The address created in H and L by the following macro
; call will be Label +012EH
                MVI             D, 1
                MVI             E, 2EH
                IXAD            D, LABEL
```

Macro expansion:

```
Label           Code            Operand
                MVI             D, 1
                MVI             E, 2EH
                LXI             H, BSADD
                DAD             D
```

# CHAPTER 5
# INTERRUPTS

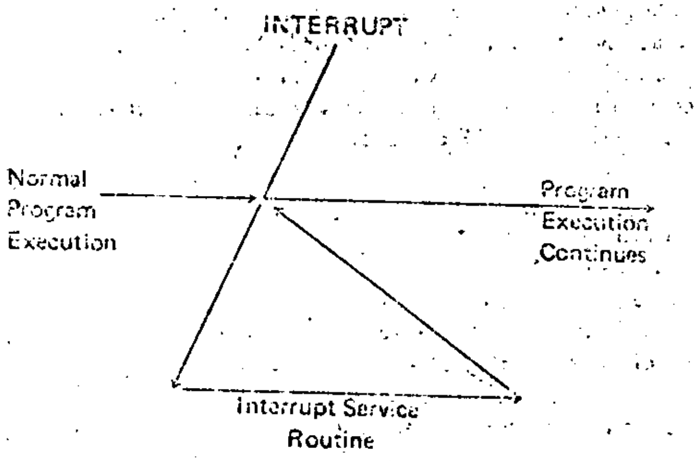
Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is receiving a string of 80 characters from the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

(a) A program could be written which inputs the first character, stalls until the next character is ready (e.g., executes a timeout by incrementing a sufficiently large counter), then inputs the next character, and proceeds in this fashion until the entire 80 character string has been received.

This method is referred to as programmed Input/Output

(b) The device controller could interrupt the CPU when a character is ready to be input, forcing a branch from the executing program to a special interrupt service routine.

The interrupt sequence may be illustrated as follows:



The 8080 contains a bit named INTE which may be set or reset by the instructions EI and DI described in Chapter 2. Whenever INTE is equal to 0, the entire interrupting system is disabled, and no interrupts will be accepted.

When the CPU recognizes an interrupt request from an external device, the following actions occur:

(1) The instruction currently being executed is completed.

(2) The interrupt enable bit, INTE, is reset = 0.

(3) The interrupting device supplies, via hardware, one instruction which the CPU executes. This instruction does not appear anywhere in memory, and the programmer has no control over it, since it is a function of the interrupting device's controller design. The program counter is not incremented before this instruction.

The instruction supplied by the interrupting device is normally an RST instruction (see Chapter 2), since this is an efficient one byte call to one of 8 eight-byte subroutines located in the first 64 words of memory. For instance, the teletype may supply the instruction:

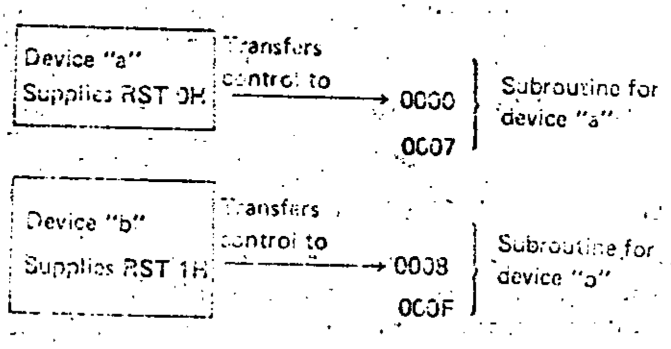
RST 0H

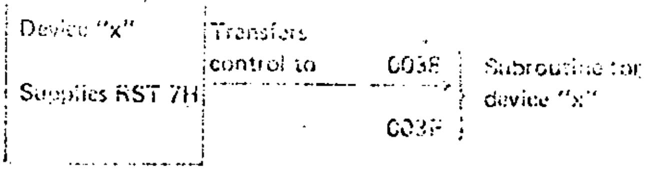
with each teletype input interrupt. Then the subroutine which processes data transmitted from the teletype to the CPU will be called into execution via an eight-byte instruction sequence at memory locations 0000H to 0007H.

A digital input device may supply the instruction:

RST 1H

Then the subroutine that processes the digital input signals will be called via a sequence of instructions occupying memory locations 0008H to 000FH.

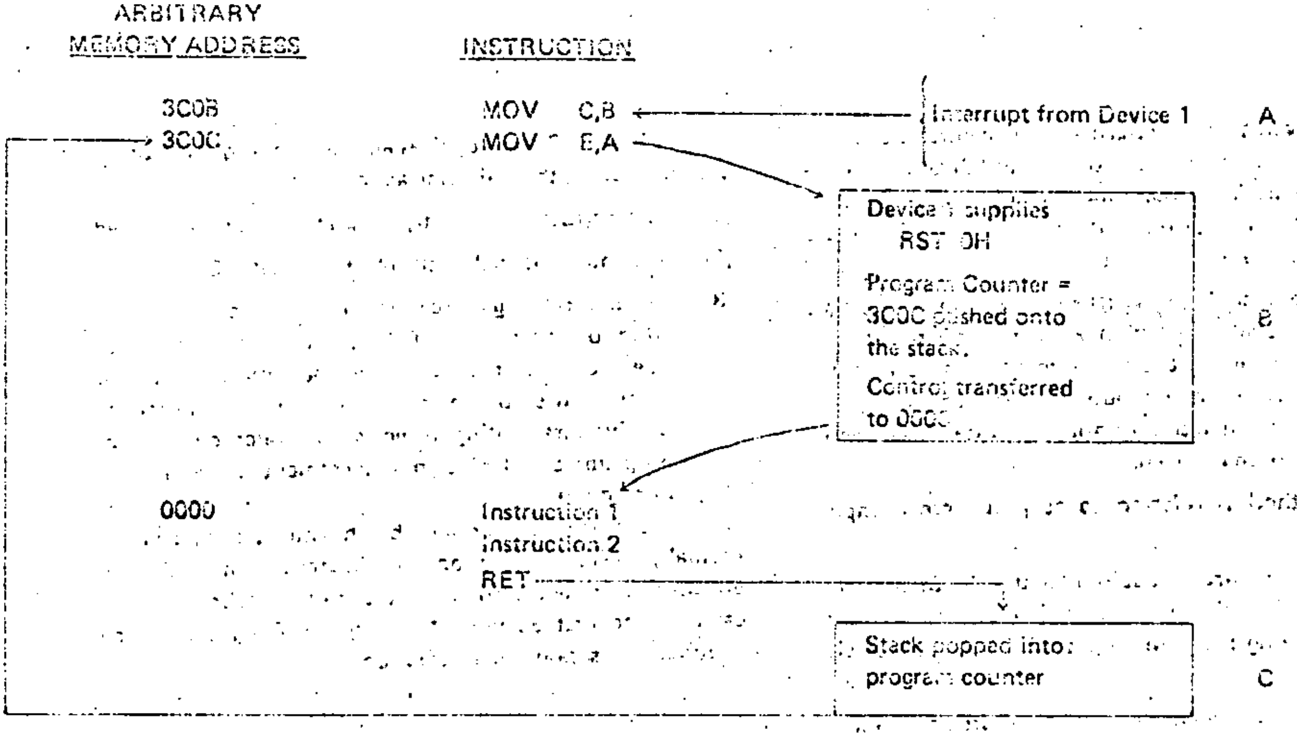




Note that any of these 8-byte subroutines may in turn call longer subroutines to process the interrupt, if necessary.

Any device may supply an RST instruction (and indeed may supply any 8080 instruction).

The following is an example of an Interrupt sequence:



Device 1 signals an interrupt as the CPU is executing the instruction at 3C0B. This instruction is completed. The program counter remains set to 3C0C, and the instruction RST 0H supplied by device 1 is executed. Since this is a call to location zero, 3C0C is pushed onto the stand and program control is transferred to location 0000H. (This subroutine may perform jumps, calls, or any other operation.) When the RETURN is executed, address 3C0C is popped off the stack and replaces the contents of the program counter, causing execution to continue at the instruction following the point where the interrupt occurred.

## WRITING INTERRUPT SUBROUTINES

In general, any registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program, or errors will occur.

For example, suppose a program is interrupted just prior to the instruction:

```
JC LOC
```

and the carry bit equal 1. If the interrupt subroutine happens to zero the carry bit just before returning to the interrupted program, the jump to LOC which should have occurred will not, causing the interrupted program to produce erroneous results.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a RETURN operation. (The obvious and most convenient way to do this is to save the data in the stack, using PUSH and POP operations.)

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an EI instruction somewhere to permit detection and handling of future interrupts. Any time after an EI is executed, the interrupt subroutine may itself be interrupted. This process may continue to any level, but as long as all pertinent data are saved and restored, correct program execution will continue automatically.

A typical interrupt subroutine, then, could appear as follows:

```
Code  Operand                Comment
PUSH   PSW     ; Save condition bits and accumulator
EI             ; Re-enable interrupts
  .            ;
  .            ; Perform necessary actions to service
  .            ; the interrupt
  .
  .
POP    PSW     ; Restore machine status
RET            ; Return to interrupted program
```

Appendix C, PL/M Special Characters

| SYMBOL | NAME | USE |
|---|---|---|
| $ | dollar sign | compiler toggles,<br>number and identifier spacer |
| = | equal sign | relational test operator,<br>assignment operator |
| := | assign | imbedded assignment operator |
| . | dot | address operator |
| / | slash | division operator |
| /* | | left comment delimiter |
| */ | | right comment delimiter |
| ( | left paren | left delimiter of lists,<br>subscripts, and expressions |
| ) | right paren | right delimiter of lists,<br>subscripts, and expressions |
| + | plus | addition operator |
| - | minus | subtraction operator |
| ' | apostrophe | string delimiter |
| * | asterisk | multiplication operator |
| < | less than | relational test operator |
| > | greater than | relational test operator |
| <= | less or equal | relational test operator |
| >= | greater or equal | relational test operator |
| <> | not equal | relational test operator |
| : | colon | label delimiter |
| ; | semicolon | statement delimiter |
| , | comma | list element delimiter |

Appendix D; PL/M Reserved Words

| RESERVED WORD | USE |
|---|---|
| IF | conditional tests and alternative execution |
| THEN | |
| ELSE | |
| DO | statement grouping and procedure definition |
| PROCEDURE | |
| INTERRUPT | |
| END | |
| DECLARE | data declarations |
| BYTE | |
| ADDRESS | |
| LABEL | |
| INITIAL | |
| DATA | |
| LITERALLY | |
| BASED | |
| GO | unconditional branching and loop control |
| TO | |
| BY | |
| GOTO | |
| CASE | |
| WHILE | |
| CALL | procedure call |
| RETURN | procedure return |
| HALT | machine stop |
| ENABLE | interrupt enable |
| DISABLE | interrupt disable |
| OR | boolean operators |
| AND | |
| XOR | |
| NOT | |
| MOD | remainder after division |
| PLUS | add with carry |
| MINUS | subtract with borrow |
| EOF | end of input file (compiler control) |

CARRY
DEC
DOUBLE
HIGH
INPUT
LAST
LENGTH
LOW
MEMORY
OUTPUT
PARITY
ROL
ROR
SCL
SCR
SHL
SHR
SIGN
STACKPTR
TIME
ZERO

# APPENDIX C ASCII TABLE

The 8080 uses a seven-bit ASCII code, which is the normal 8 bit ASCII code with the parity (high-order) bit always reset.

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| NULL | 00 |
| SOM | 01 |
| EOA | 02 |
| EOM | 03 |
| EOT | 04 |
| WRU | 05 |
| RU | 06 |
| BELL | 07 |
| FE | 08 |
| H. Tab | 09 |
| Line Feed | 0A |
| V. Tab | 0B |
| Form | 0C |
| Return | 0D |
| SO | 0E |
| SI | 0F |
| DCO | 10 |
| X-On | 11 |
| Tape Aux. On | 12 |
| X-Off | 13 |
| Tape Aux. Off | 14 |
| Error | 15 |
| Sync | 16 |
| LEM | 17 |
| S0 | 18 |
| S1 | 19 |
| S2 | 1A |
| S3 | 1B |
| S4 | 1C |
| S5 | 1D |
| S6 | 1E |
| S7 | 1F |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| ACK | 7C |
| Alt. Mode | 7D |
| Rubout | 7F |
| ! | 21 |
| " | 22 |
| # | 23 |
| $ | 24 |
| % | 25 |
| & | 26 |
| ' | 27 |
| ( | 28 |
| ) | 29 |
| * | 2A |
| + | 2B |
| , | 2C |
| - | 2D |
| . | 2E |
| / | 2F |
| : | 3A |
| ; | 3B |
| < | 3C |
| = | 3D |
| > | 3E |
| ? | 3F |
| [ | 5B |
| \ | 5C |
| ] | 5D |
| ↑ | 5E |
| ← | 5F |
| @ | 40 |
| blank | 20 |
| 0 | 30 |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| 1 | 31 |
| 2 | 32 |
| 3 | 33 |
| 4 | 34 |
| 5 | 35 |
| 6 | 36 |
| 7 | 37 |
| 8 | 38 |
| 9 | 39 |
| A | 41 |
| B | 42 |
| C | 43 |
| D | 44 |
| E | 45 |
| F | 46 |
| G | 47 |
| H | 48 |
| I | 49 |
| J | 4A |
| K | 4B |
| L | 4C |
| M | 4D |
| N | 4E |
| O | 4F |
| P | 50 |
| Q | 51 |
| R | 52 |
| S | 53 |
| T | 54 |
| U | 55 |
| V | 56 |
| W | 57 |
| X | 58 |
| Y | 59 |
| Z | 5A |

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

# MICROPROCESADORES: TEORIA Y APLICACIONES

## MICROCOMPUTADORA 8080

ABRIL, 1979

intel®

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

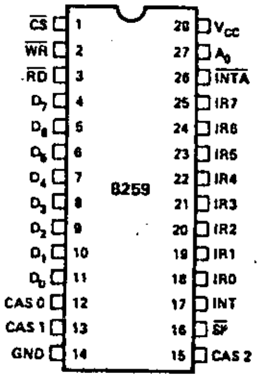
# 8259
# PROGRAMMABLE INTERRUPT CONTROLLER

- Eight Level Priority Controller
- Expandable to 64 Levels
- Programmable Interrupt Modes (Algorithms)
- Individual Request Mask Capability
- Single +5V Supply (No Clocks)
- 28 Pin Dual-In-Line Package
- Fully Compatible with Intel CPUs

The 8259 handles up to eight vectored priority interrupts for microprocessors. It is cascadable for up to 64 vectored priority interrupts, without additional circuitry. It will be packaged in a 28-pin plastic DIP, uses nMOS technology and requires a single +5V supply. Circuitry is static, requiring no clock input.

The 8259 is designed to minimize the software and real time overhead in handling multi-level priority interrupts. It has several modes, permitting optimization for a variety of system requirements.

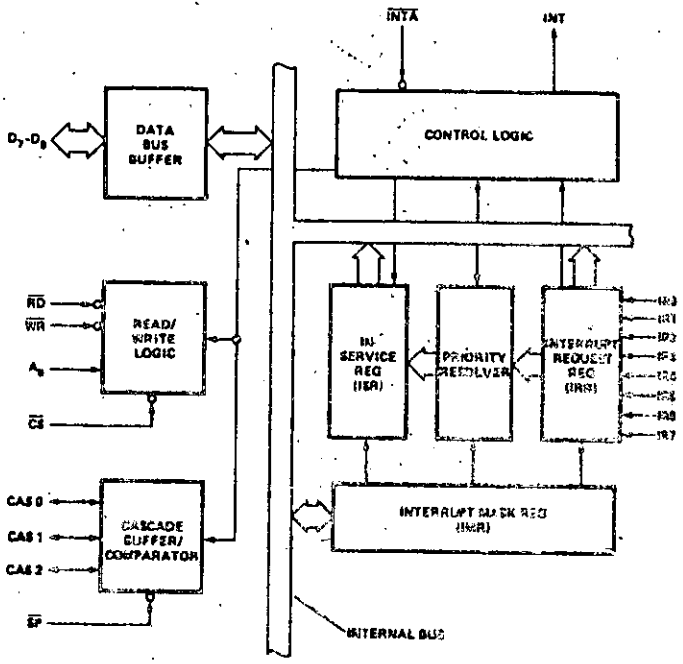
## PIN CONFIGURATION



## PIN NAMES

| | |
|---|---|
| $D_7$-$D_0$ | DATA BUS (BI-DIRECTIONAL) |
| $\overline{RD}$ | READ INPUT |
| $\overline{WR}$ | WRITE INPUT |
| $A_0$ | COMMAND SELECT ADDRESS |
| $\overline{CS}$ | CHIP SELECT |
| CAS1-CAS0 | CASCADE LINES |
| $\overline{SP}$ | SLAVE PROGRAM INPUT |
| INT | INTERRUPT OUTPUT |
| $\overline{INTA}$ | INTERRUPT ACKNOWLEDGE INPUT |
| IR0-IR7 | INTERRUPT REQUEST INPUTS |

## BLOCK DIAGRAM



## INTERRUPTS IN MICROCOMPUTER SYSTEMS

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors and other components receive servicing in an efficient method so that large amounts of the total system tasks can be assumed by the microcomputer with little or no effect on throughput.

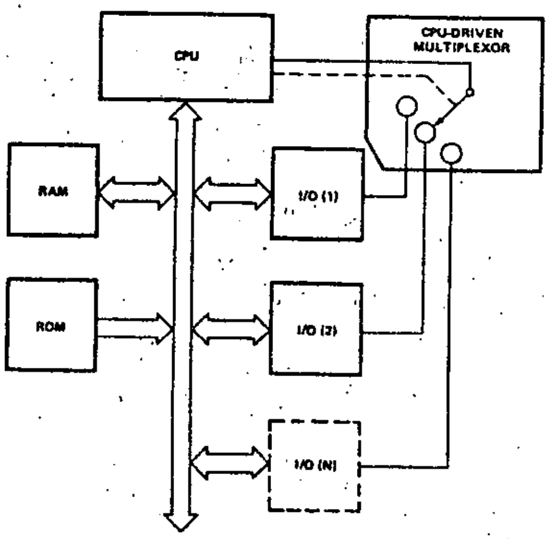
The most common method of servicing such devices is the Polled approach. This is where the processor must test each device in sequence and in effect "ask" each one if it needs servicing. It is easy to see that a large portion of the main program is looping through this continuence polling cycle and that such a method would have a serious, detrimental effect on system throughput thus limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices.

A more desireable method would be one that would allow the microprocessor to be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete however the processor would resume exactly where it left off.

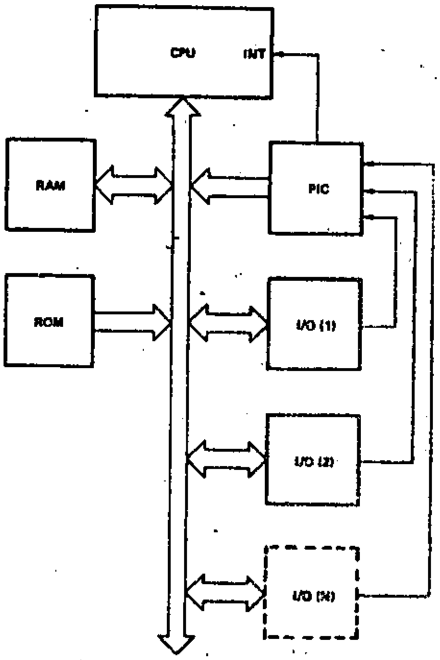
This method is called Interrupt. It is easy to see that system throughput would drastically increase, and thus more tasks could be assumed by the microcomputer to further enhance its cost effectiveness.

The Programmable Interrupt Controller (PIC) functions as an overall manager in an Interrupt-Driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced and issues an Interrupt to the CPU based on this determination.

Each peripheral device or structure usually has a special program or "routine" that is associated with its specific functional or operational requirements; this is referred to as a "service routine". The PIC, after issuing an Interrupt to the CPU, must somehow input information into the CPU that can "point" the Program Counter to the service routine associated with the requesting device. The PIC does this by providing the CPU with a 3-byte CALL instruction.



POLLED METHOD



INTERRUPT METHOD

## 8259 BASIC FUNCTIONAL DESCRIPTION

### General

The 8259 is a device specifically designed for use in real time, interrupt driven, microcomputer systems. It manages eight levels or requests and has built-in features for expandability to other 8259s (up to 64 levels). It is programmed by the system's software as an I/O peripheral. A selection of priority algorithms is available to the programmer so that the manner in which the requests are processed by the 8259 can be configured to match his system requirements. The priority assignments and algorithms can be changed or reconfigured dynamically at any time during the main program. This means that the complete interrupt structure can be defined as required, based on the total system environment.

### Interrupt Request Register (IRR) and In-Service Register (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all the interrupt levels which are requesting service; and the ISR is used to store all the interrupt levels which are being serviced.

The IRR bit is set and INT line is raised high whenever there is a positive going edge at the IR input. However, the IR input must be held high until the 1st $\overline{\text{INTA}}$ pulse has arrived. More than one bit of the IRR can be set at once as long as they are not masked. The IRR is reset by the $\overline{\text{INTA}}$ sequence.

The ISR bit is set by the $\overline{\text{INTA}}$ pulse (at the same time the selected IRR bit is reset). This bit remains set during the subroutine until an EOI (End of Interrupt) command is received by the 8259.

The return from the subroutine to the main program may look like this:

```
DI
OUT     OCW2     (Send EOI command)
POP     PSW
EI
RET
```

### Priority Resolver

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during $\overline{\text{INTA}}$ pulse.

### INT (Interrupt)

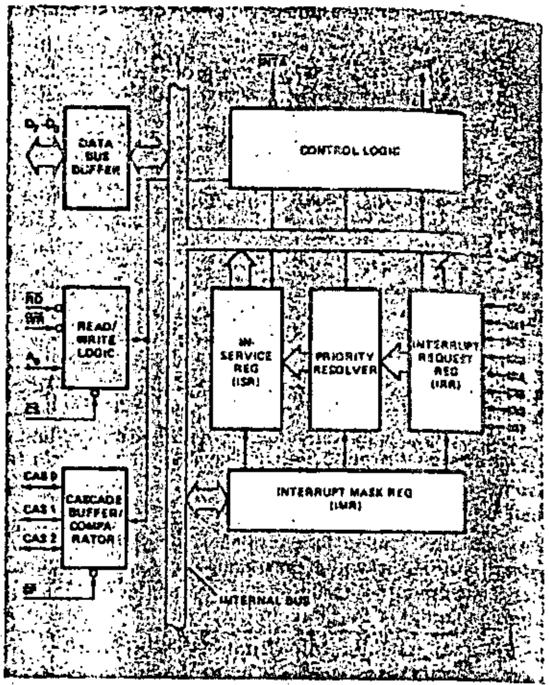
This output goes directly to the 8080 INT input. The $V_{OH}$ level on this line is designed to be fully compatible with the 8080 input level.

### $\overline{\text{INTA}}$ (Interrupt Acknowledge)

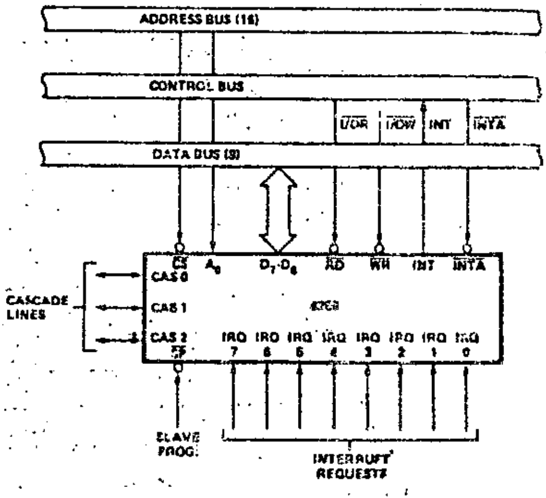
This input generally comes from the 8228 of the CPU group. The 8228 will produce 3 distinct $\overline{\text{INTA}}$ pulses. The 3 $\overline{\text{INTA}}$ pulses will cause the 8259 to release a 3-byte CALL instruction onto the Data Bus.

### Interrupt Mask Register (IMR)

The IMR stores the bits of the interrupt lines to be masked. The IMR operates on both the IRR and the ISR. Masking of a higher priority bit will not affect the interrupt request lines of lower priority.



8259 BLOCK DIAGRAM



8259 INTERFACE TO 8080 STANDARD SYSTEM BUS

### Data Bus Buffer

This 3-state, bi-directional, 8-bit buffer is used to interface the 8259 to the 8080 system Data Bus. Control words and status information are transferred through the Data Bus Buffer.

### Read/Write Control Logic

The function of this block is to accept OUTput commands from the 8080. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the 8259 to be transferred onto the 8080 Data Bus.

### $\overline{CS}$ (Chip Select)

A "low" on this input enables the 8259. No reading or writing of the chip will occur unless the device is selected.

### $\overline{WR}$ (Write)

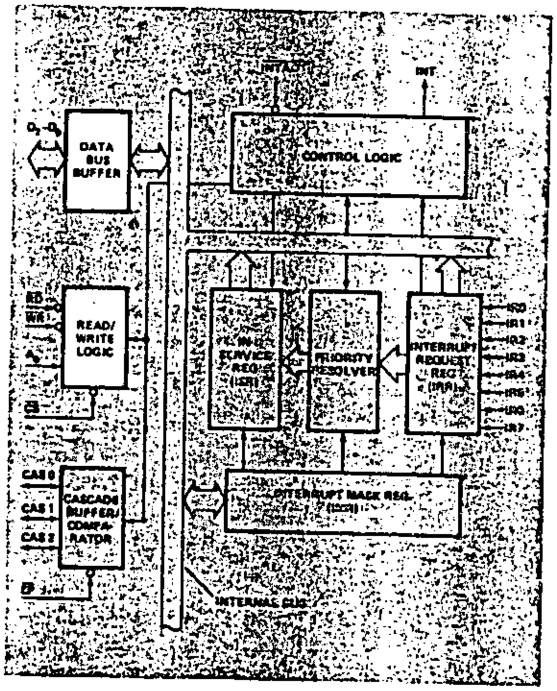
A "low" on this input enables the 8080 CPU to write control words (ICWs and OCWs) to the 8259.

### $\overline{RD}$ (Read)

A "low" on this input enables the 8259 to send the status of the Interrupt Request Register (IRR), In Service Register (ISR), the Interrupt Mask Register (IMR) or the BCD of the Interrupt level on to the Data Bus.

### A0

This input signal is used in conjunction with $\overline{WR}$ and $\overline{RD}$ signals to write commands into the various command registers as well as reading the various status registers of the chip. This line can be tied directly to one of the 8080 address lines.



8259 BLOCK DIAGRAM

8259 BASIC OPERATION

| $A_0$ | $D_4$ | $D_3$ | $\overline{RD}$ | $\overline{WR}$ | $\overline{CS}$ | INPUT OPERATION (READ) |
|---|---|---|---|---|---|---|
| 0 | | | 0 | 1 | 0 | IRR, ISR or Interrupting Level ⇒ DATA BUS (Note 1) |
| 1 | | | 0 | 1 | 0 | IMR ⇒ DATA BUS |
| | | | | | | OUTPUT OPERATION (WRITE) |
| 0 | 0 | 0 | 1 | 0 | 0 | DATA BUS ⇒ OCW2 |
| 0 | 0 | 1 | 1 | 0 | 0 | DATA BUS ⇒ OCW3 |
| 0 | 1 | X | 1 | 0 | 0 | DATA BUS ⇒ ICW1 |
| 1 | X | X | 1 | 0 | 0 | DATA BUS ⇒ OCW1, ICW2, ICW3 (Note 2) |
| | | | | | | DISABLE FUNCTION |
| X | X | X | 1 | 1 | 0 | DATA BUS ⇒ 3-STATE |
| X | X | X | X | X | 1 | DATA BUS ⇒ 3-STATE |

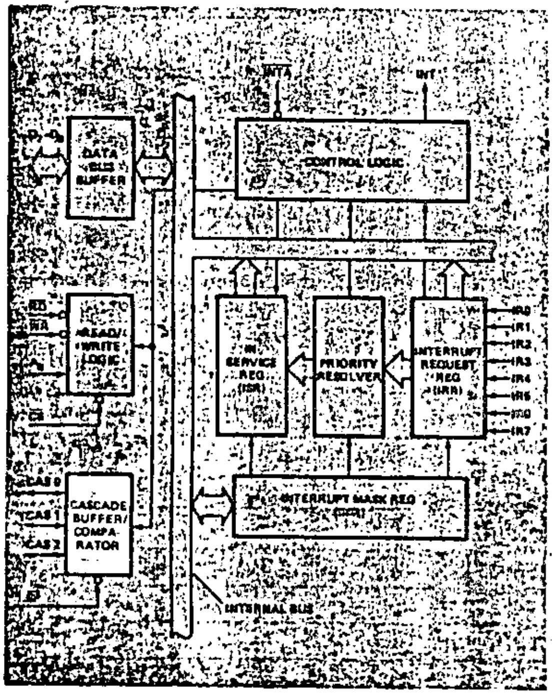
Note 1: Selection of IRR, ISR or Interrupting Level is based on the content of OCW3 written before the READ operation.
Note 2: On-chip sequencer logic queues these commands into proper sequence.

### $\overline{SP}$ (Slave Program)

More than one 8259 can be used in the system to expand the priority interrupt scheme up to 64 levels. In such case, one 8259 acts as the master, and the others act as slaves. A "high" on the $\overline{SP}$ pin designates the 8259 as the master, a "low" designates it as a slave.

### The Cascade Buffer/Comparator

This function block stores and compares the IDs of all 8259 used in the system. The associated three I/O pins (CAS0-2) are outputs when the 8259 is used as a master ($\overline{SP}$ = 1), and are inputs when the 8259 is used as a slave ($\overline{SP}$ = 0). As a master, the 8259 sends the ID of the interrupting slave device onto the CAS0-2 lines. The slave thus selected will send its preprogrammed subroutine addressed onto the Data Bus during next two consecutive $\overline{INTA}$ pulses. (See section "Cascading the 8259".)



8259 BLOCK DIAGRAM

## 8259 DETAILED OPERATIONAL SUMMARY

### General

The powerful features of the 8259 in the 8080 microcomputer system are its programmability and its utilization of the 8080 CALL instruction to jump into any address in the memory map. The normal sequence of events that the 8259 interacts with the CPU is as follows:

1. One or more of the INTERRUPT REQUEST lines (IR7-0) are raised high signaling the 8259 that the peripheral equipment(s) are demanding service.
2. The 8259 accepts these requests, resolves the priorities, and sends an INT to the 8080 CPU.
3. The 8080 CPU acknowledges the INT and responds with an $\overline{INTA}$ pulse.
4. Upon receiving the $\overline{INTA}$ from the CPU group (8228), the 8259 will release a CALL instruction code (11001101) onto the 8-bit Data Bus through its D7-0 pins.
5. This CALL instruction will initiate two more $\overline{INTA}$ pulses to be sent to the 8259 from the CPU group (8228).
6. These two $\overline{INTA}$ pulses allow the 8259 to release its preprogrammed subroutine address onto the Data Bus. The lower 8-bit address is released at the first $\overline{INTA}$ pulse and the higher 8-bit address is released at the second $\overline{INTA}$ pulse.
7. This completes the 3-byte CALL instruction released by the 8259. The In-Service Register (ISR) is not reset until the end of the subroutine when an EOI (End of interrupt) command is issued to the 8259.

### Programming The 8259

The 8259 accepts two types of command words generated by the CPU:

1. Initialization Command Words (ICWs):
   Before normal operation can begin, each 8259 in the system must be brought to a starting point — by a sequence of 2 or 3 bytes timed by $\overline{WR}$ pulses. This sequence is described in Figure 1.
2. Operation Command Words (OCWs):
   These are the command words which command the 8259 to operate in various interrupt modes. These modes are:
   a. Fully nested mode
   b. Rotating priority mode
   c. Special mask mode
   d. Polled mode

   The OCWs can be written into the 8259 at anytime during operation.

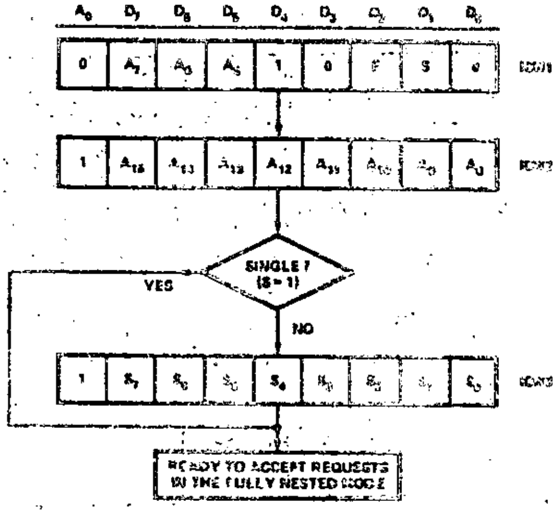


FIGURE 1. INITIALIZATION SEQUENCE

### Operation Command Words (OCWs)

After the Initialization Command Words (ICWs) are programmed into the 8259, the chip is ready to accept interrupt requests at its input lines. However, during the 8259 operation, a selection of algorithms can command the 8259 to operate in various modes through the Operation Command Words (OCWs). These various modes and their associated OCWs are described below.

#### Interrupt Masks

Each Interrupt Request input can be masked individually by the Interrupt Masked Register (IMR) programmed through OCW1.

The IMR will operate on both the Interrupt Request Register and the In-Service Register. Note that if an interrupt is already acknowledged by the 8259 (an $\overline{\text{INTA}}$ pulse has occurred), then the Interrupting level, although masked, will inhibit the lower priorities. To enable these lower priority interrupts, one can do one of the two things: (1) Write an End of Interrupt (EOI) command (OCW2) to reset the ISR bit or (2) Set the special mask mode using OCW3 (as will be explained later in the special mask mode.)

#### Fully Nested Mode

The 8259 will operate in the fully nested mode after the execution of the initialization sequence without any OCW being written. In this mode, the interrupt requests are ordered in priorities from 0 through 7. When an interrupt is acknowledged, the highest priority request is determined and its address vector placed on the bus. In addition, a bit of the Interrupt service register (IS 7-0) is set. This bit remains set until the 8080 issues an End of Interrupt (EOI) command immediately before returning from the service routine. While the IS bit is set, all further interrupts of lower priority are inhibited, while higher levels will be able to generate an interrupt (which will only be acknowledged if the 8080 has enabled its own interrupt input through software).

After the Initialization sequence, IR0 has the highest priority and IR7 the lowest. Priorities can be changed, as will be explained in the rotating priority mode.

#### Rotating Priority Modes

The Rotating Priority Modes of the 8259 serves in application of interrupting devices of equal priority such as communication channels. There are two variations of the rotating priority mode: the auto mode and the specific mode.

1. Auto Mode — In this mode, a device after being serviced receives the lowest priority, so a device requesting an interrupt will have to wait, in the worst case, until 7 other devices are serviced at most once each. i.e., if the priority and "in service" status is:

| BEFORE ROTATE | IS7 | IS6 | IS5 | IS4 | IS3 | IS2 | IS1 | IS0 |
|---|---|---|---|---|---|---|---|---|
| "IS" STATUS | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| | LOWEST PRIORITY | | | | | | | HIGHEST PRIORITY |
| PRIORITY STATUS | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| AFTER ROTATE | IS7 | IS6 | IS5 | IS4 | IS3 | IS2 | IS1 | IS0 |
|---|---|---|---|---|---|---|---|---|
| "IS" STATUS | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | LOWEST PRIORITY | | | | | | | HIGHEST PRIORITY |
| PRIORITY STATUS | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 |

In this example, the In-Service FF corresponding to line 4 (the highest priority FF set) was reset and line 4 became the lowest priority, while all the other priorities rotated correspondingly.

The Rotate command is issued in OCW2, where: R = 1, EOI = 1, SEOI = 0.

2. Specific Mode — The programmer can change priorities by programming the bottom priority, and by doing this, to fix the highest priority: i.e., if IR5 is programmed as the bottom priority device, the IR6 will have the highest one.

   The Rotate command is issued in OCW2 where: R = 1, SEOI = 1. L2, L1, L0 are the BCD priority level codes of the bottom priority device.

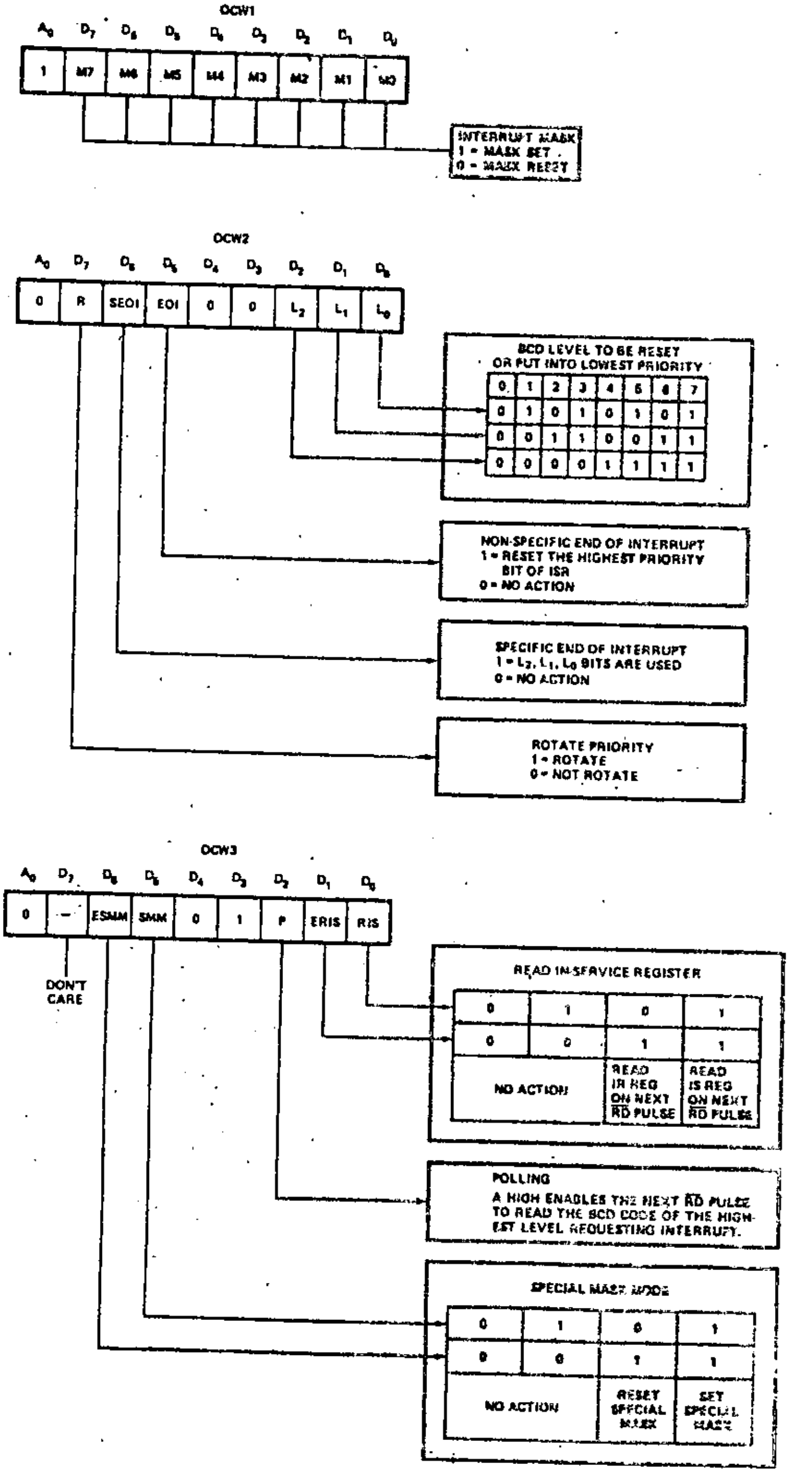
   Observe that this mode is independent of the End of Interrupt Command and priority changes can be executed during EOI command or independently from the EOI command.

#### End of Interrupt (EOI) and Specific End of Interrupt (SEOI)

An End of Interrupt command word must be issued to the 8259 before returning from a service routine, to reset the appropriate IS bit.

There are two forms of EOI command: Specific and non-Specific. When the 8259 is operated in modes which preserve the fully nested structure, it can determine which IS bit to reset on EOI. When a non-Specific EOI command is issued the 8259 will automatically reset the highest IS bit of those that are set, since in the nested mode, the highest IS level was necessarily the last level acknowledged and will necessarily be the next routine level returned from.

However, when a mode is used which may disturb the fully nested structure, such as in the rotating priority case, the 8259 may no longer be able to determine the last level acknowledged. In this case, a specific EOI (SEOI) must be issued which includes the IS level to be reset as part of the command. The End of the Interrupt is issued whenever EOI = "1" in OCW2. For specific EOI, SEOI = "1", and EOI = 1. L2, L1, L0 is then the BCD level to be reset. As explained in the Rotate Mode earlier, this can also be the bottom priority code. Note that although the Rotate command can be issued during an EOI = 1, it is not necessarily tied to it.


OCW1
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 M7 M6 M5 M4 M3 M2 M1 M0
INTERRUPT MASK
1 = MASK SET
0 = MASK RESET
OCW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 R SEOI EOI 0 0 L2 L1 L0
BCD LEVEL TO BE RESET OR PUT INTO LOWEST PRIORITY
NON-SPECIFIC END OF INTERRUPT
1 = RESET THE HIGHEST PRIORITY BIT OF ISR
0 = NO ACTION
SPECIFIC END OF INTERRUPT
1 = L2, L1, L0 BITS ARE USED
0 = NO ACTION
ROTATE PRIORITY
1 = ROTATE
0 = NOT ROTATE
OCW3
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 — ESMM SMM 0 1 P ERIS RIS
DON'T CARE
READ IN-SERVICE REGISTER
POLLING
A HIGH ENABLES THE NEXT RD PULSE TO READ THE BCD CODE OF THE HIGHEST LEVEL REQUESTING INTERRUPT.
SPECIAL MASK MODE

BCD LEVEL TO BE RESET OR PUT INTO LOWEST PRIORITY

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

READ IN-SERVICE REGISTER

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| NO ACTION | | READ IR REG ON NEXT $\overline{RD}$ PULSE | READ IS REG ON NEXT $\overline{RD}$ PULSE |

SPECIAL MASK MODE

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| NO ACTION | | RESET SPECIAL MASK | SET SPECIAL MASK |

OPERATION COMMAND WORD FORMAT

**Special Mask Mode (SMM)**

This mode is useful when some bit(s) are set (masked) by the Interrupt Mask Register (IMR) through OCW1. If, for some reason, we are currently in a subroutine which is masked (this could happen when the subroutine intentionally masks itself off). It is still possible to enable the lower priority lines by setting the Special Mask mode. In this mode the lower priority lines are enabled until the SMM is reset. The higher priorities are not affected.

The special mask mode FF is set by OCW3 where ESMM = 1, SMM = 1, and reset where:. ESSM = 1 and SMM = 0.

**Polled Mode**

In this mode, the 8080 disables its interrupt input. Service to devices is achieved by programmer initiative by a Poll command.

The poll command is issued by setting P = "1" in OCW3 during a $\overline{WR}$ pulse.

The 8259 treats the next $\overline{RD}$ pulse as an interrupt acknowledge, sets the appropriate IS Flip-flop, if there is a request, and reads the priority level.

The word enabled onto the data bus during $\overline{RD}$ is:

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| I | – | – | – | – | W2 | W1 | W0 |

W0 — 2: BCD code of the highest priority level requesting service.

I: Equal to a "1" if there is an interrupt.

This mode is useful if there is a routine command common to several levels — so that the $\overline{INTA}$ sequence is not needed (and this saves ROM space). Another application is to use the poll mode to expand the number of priority levels to more than 64.

**SUMMARY OF OPERATION COMMAND WORD PROGRAMMING**

| | A0 | D4 | D3 | | |
|---|---|---|---|---|---|
| OCW1 | 1 | | | M7-M0 | IMR (Interrupt Mask Register). $\overline{WR}$ will load it while status can be read with $\overline{RD}$. |
| OCW2 | 0 | 0 | 0 | R SEOI EOI | |
| | | | | 0 0 0 | No Action. |
| | | | | 0 0 1 | Non-specific End of Interrupt. |
| | | | | 0 1 0 | No Action. |
| | | | | 0 1 1 | Specific End of Interrupt. L2, L1, L0 is the BCD level to be reset. |
| | | | | 1 0 0 | No Action. |
| | | | | 1 0 1 | Rotate priority at EOI. (Auto Mode) |
| | | | | 1 1 0 | Rotate priority, L2, L1, L0 becomes bottom priority without Ending of Interrupt. |
| | | | | 1 1 1 | Rotate priority at EOI (Specific Mode), L2, L1, L0 becomes bottom priority, and its corresponding IS FF is reset. |
| OCW3 | 0 | 1 | 0 | ESMM SMM | |
| | | | | 0 0 | Special Mask not Affected. |
| | | | | 0 1 | |
| | | | | 1 0 | Reset Special Mask. |
| | | | | 1 1 | Set Special Mask. |
| | | | | ERIS RIS | |
| | | | | 0 0 | No Action. |
| | | | | 0 1 | |
| | | | | 1 0 | Read IR Register Status. |
| | | | | 1 1 | Read IS Register Status. |

Note: The 8080 INT input must be disabled during:

1. Initialization sequence for all the 8259 in the system.
2. Any control command execution.

### Reading 8259 Status

The input status of several internal registers can be read to update the user information on the system. The following registers can be read by issuing a suitable OCW and reading with $\overline{RD}$ for the data bus lines:

Interrupt Requests Register (IRR): 8-bit register which contains the priority levels requesting an interrupt to be acknowledged. The highest request level is reset from the IRR when an interrupt is acknowledged.

In Service Register (ISR): 8-bit register which contains the priority levels that are being serviced. The ISR is updated when an End of Interrupt command is issued.

Interrupt Mask Register: 8-bit register which contains the interrupt request lines which are masked.

The IRR can be read when prior to the $\overline{RD}$ pulse, an $\overline{WR}$ pulse is issued with OCW3, and ERIS = 1, RIS = 0.

The ISR can be read in a similar mode, when ERIS = 1, RIS = 1.

There is no need to write an OCW3 before every status read operation as long as the status read corresponds with the previous one, i.e. the 8259 "remembers" whether the IRR or ISR has been previously selected by the OCW3. On the other hand, for polling operation, an OCW3 must be written before every read.

For reading the IMR, a $\overline{WR}$ pulse is not necessary to preceed the $\overline{RD}$. The output data bus will contain the IMR whenever $\overline{RD}$ is active and A0 = 1.

The IMR can be loaded through the data bus when $\overline{WR}$ is active and A0 = 1.

Polling overrides status read when P = 1, ERIS = 1 in OCW3.

### Cascading

The 8259 can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels.

A typical system is shown in Figure 2. The master controls, through the 3 line cascade bus, which one of the slaves will release the corresponding address.

As shown in Figure 2, the slaves interrupt outputs are connected to the master interrupt request inputs. When a slave request line is activated and afterwards acknowledged, the master will release the 8080 CALL code during byte 1 of $\overline{INTA}$ and will enable the corresponding slave to release the device routine address during bytes 2 and 3 of $\overline{INTA}$.

The cascade bus lines are normally low and will contain the slave address code from the trailing edge of the first $\overline{INTA}$ pulse to the trailing edge of the third pulse. It is obvious that each 8259 in the system must follow a separate initialization sequence and can be programmed to work in a different mode. An EOI command must be issued twice: once for the master and once for the corresponding slave. An address decoder is required to activate the Chip Select ($\overline{CS}$) input of each 8259. The slave program pin ($\overline{SP}$) must be at a "low" level for a slave (and then the cascade lines are inputs) and at a "high" level for a master (and then the cascade lines are outpus).

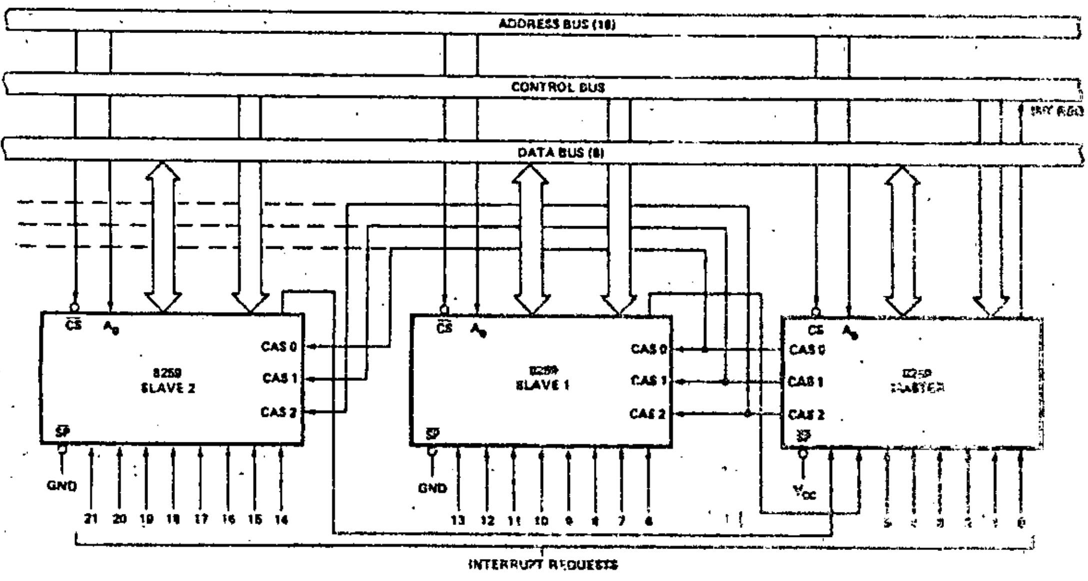


FIGURE 2. CASCADING THE 8259

## 8259 INSTRUCTION SET

| INST. NO. | MNEMONIC | A0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | OPERATION DESCRIPTION |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ICW1 A | 0 | A7 | A6 | A5 | 1 | 0 | 1 | 1 | 0 | Byte 1 initialization, format = 4, single. |
| 2 | ICW1 B | 0 | A7 | A6 | A5 | 1 | 0 | 1 | 0 | 0 | Byte 1 initialization, format = 4, not single. |
| 3 | ICW1 C | 0 | A7 | A6 | A5 | 1 | 0 | 0 | 1 | 0 | Byte 1 initialization, format = 8, single. |
| 4 | ICW1 D | 0 | A7 | A6 | A5 | 1 | 0 | 0 | 0 | 0 | Byte 1 initialization, format = 8, not single. |
| 6 | ICW2 | 1 | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | Byte 2 initialization (Address No. 2) |
| 6 | ICW3 M | 1 | S7 | S6 | S5 | S4 | S3 | S2 | S1 | S0 | Byte 3 initialization — master. |
| 7 | ICW3 S | 1 | 0 | 0 | 0 | 0 | 0 | S2 | S1 | S0 | Byte 3 initialization — slave. |
| 8 | OCW1 | 1 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 | Load mask reg, read mask reg. |
| 9 | OCW2 E | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Non specific EOI. |
| 10 | OCW2 SE | 0 | 0 | 1 | 1 | 0 | 0 | L2 | L1 | L0 | Specific EOI. L2, L1, L0 code of IS FF to be reset. |
| 11 | OCW2 RE | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Rotate at EOI (Auto Mode). |
| 12 | OCW2 RSE | 0 | 1 | 1 | 1 | 0 | 0 | L2 | L1 | L0 | Rotate at EOI (Specific Mode). L2, L1, L0, code of line to be reset and selected as bottom priority. |
| 13 | OCW2 RS | 0 | 1 | 1 | 0 | 0 | 0 | L2 | L1 | L0 | L2, L1, L0 code of bottom priority line. |
| 14 | OCW3 P | 0 | — | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Poll mode. |
| 15 | OCW3 RIS | 0 | — | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Read IS register. |
| 16 | OCW3 RR | 0 | — | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Read requests register. |
| 17 | OCW3 SM | 0 | — | 1 | 1 | 0 | 1 | 0 | 0 | 0 | Set special mask mode. |
| 18 | OCW3 RSM | 0 | — | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Reset special mask mode. |

Notes:
1. In the master mode $\overline{SP}$ pin = 1, in slave mode $\overline{SP}$ = 0.
2. (—) = do not care.

10-223

## ABSOLUTE MAXIMUM RATINGS*

| | |
|---|---|
| Ambient Temperature Under Bias | 0°C to 70°C |
| Storage Temperature | −65°C to +150°C |
| Voltage On Any Pin With Respect to Ground | −0.5 V to +7 V |
| Power Dissipation | 1 Watt |

*COMMENT:
*Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.*

## D.C. CHARACTERISTICS ($T_A$ = 0°C to 70°C; $V_{CC}$ = 5V ±5%)

| SYMBOL | PARAMETER | MIN. | MAX. | UNITS | TEST CONDITIONS |
|---|---|---|---|---|---|
| $V_{IL}$ | Input Low Voltage | −.5 | .8 | V | |
| $V_{IH}$ | Input High Voltage | 2.0 | $V_{CC}$+.5V | V | |
| $V_{OL}$ | Output Low Voltage | | .45 | V | $I_{OL}$ = 2 mA |
| $V_{OH}$ | Output High Voltage | 2.4 | | V | $I_{OH}$ = −400 μA |
| $V_{OH-INT}$ | Interrupt Output High Voltage | 2.4<br>3.5 | | V<br>V | $I_{OH}$ = −400 μA<br>$I_{OH}$ = −50 μA |
| $I_{IL(IR_{0-7})}$ | Input Leakage Current for $IR_{0-7}$ | | -300<br>10 | μA<br>μA | $V_{IN}$ = 0V<br>$V_{IN}$ = $V_{CC}$ |
| $I_{IL}$ | Input Leakage Current for Other Inputs | | 10 | μA | $V_{IN}$ = $V_{CC}$ to 0V |
| $I_{LOL}$ | Output Leakage Current | | -10 | μA | $V_{OUT}$ = 0.45V |
| $I_{LOH}$ | Output Leakage Current | | 10 | μA | $V_{OUT}$ = $V_{CC}$ |
| $I_{CC}$ | $V_{CC}$ Supply Current | | 85 | mA | |

## CAPACITANCE $T_A$ = 25°C; $V_{CC}$ = GND = 0V

| SYMBOL | PARAMETER | MIN. | TYP. | MAX. | UNIT | TEST CONDITIONS |
|---|---|---|---|---|---|---|
| $C_{IN}$ | Input Capacitance | | | 10 | pF | fc = 1 MHz |
| $C_{I/O}$ | I/O Capacitance | | | 20 | pF | Unmeasured pins returned to $V_{SS}$ |

10-224

## .C. CHARACTERISTICS ($T_A$ = 0°C to 70°C; $V_{CC}$ = +5V ±5%, GND = 0V)

### US PARAMETERS

### READ

| SYMBOL | PARAMETER | MIN. | MAX. | UNIT | TEST CONDITIONS |
|---|---|---|---|---|---|
| $t_{AR}$ | $\overline{\text{CS}}/\text{A}_0$ Stable before $\overline{\text{RD}}$ or $\overline{\text{INTA}}$ | 50 | | ns | |
| $t_{RA}$ | $\overline{\text{CS}}/\text{A}_0$ Stable after $\overline{\text{RD}}$ or $\overline{\text{INTA}}$ | 50 | | ns | |
| $t_{RR}$ | $\overline{\text{RD}}$ Pulse Width | 420 | | ns | |
| $t_{RD}$ | Data Valid from $\overline{\text{RD}}/\overline{\text{INTA}}$ | | 300 | ns | CL = 100 pF |
| $t_{DF}$ | Data Float after $\overline{\text{RD}}/\overline{\text{INTA}}$ | <br>20 | 200 | ns | CL = 100 pF<br>CL = 20 pF |

### WRITE

| SYMBOL | PARAMETER | MIN. | MAX. | UNIT | TEST CONDITIONS |
|---|---|---|---|---|---|
| $t_{AW}$ | $\text{A}_0$ Stable before $\overline{\text{WR}}$ | 50 | | ns | |
| $t_{WA}$ | $\text{A}_0$ Stable after $\overline{\text{WR}}$ | 20 | | ns | |
| $t_{CW}$ | $\overline{\text{CS}}$ Stable before $\overline{\text{WR}}$ | 50 | | ns | |
| $t_{WC}$ | $\overline{\text{CS}}$ Stable after $\overline{\text{WR}}$ | 20 | | ns | |
| $t_{WW}$ | $\overline{\text{WR}}$ Pulse Width | 400 | | ns | |
| $t_{DW}$ | Data Valid to $\overline{\text{WR}}$ (T.E.) | 300 | | ns | |
| $t_{WD}$ | Data Valid after $\overline{\text{WR}}$ | -40 | | ns | |

### OTHER TIMINGS

| SYMBOL | PARAMETER | MIN. | MAX. | UNIT | TEST CONDITIONS |
|---|---|---|---|---|---|
| $t_{IW}$ | Width of Interrupt Request Pulse | 100 | | ns | |
| $t_{INT}$ | INT ↑ after IR ↑ | 250 | | ns | |
| $t_{IC}$ | Cascade Line Stable after $\overline{\text{INTA}}$ ↑ | 300 | | ns | |

# Programmer's Guide

## 5.1 Introduction to CDOS System Calls

To a programmer, system calls are the single most important feature of CDOS. The user writing assembly language programs to run under CDOS should become familiar with the use of system calls.

A system call is a call to the operating system which initiates a function, usually involving one of the I/O devices. The most important system calls perform I/O with, and manipulate, the floppy disk drives. CDOS also has system calls to perform device I/O with CRTs, printers, punches, and readers. System calls are available to perform such special purpose functions as storing and reading the date or time of day and multiplying and dividing integers.

Because CDOS is designed to handle all I/O functions in a manner which is independent of system size and configuration, programmers should ordinarily do all I/O through the operating system by means of system calls.

A system call is executed by loading the C register with the number of the call (refer to Section 5.1.5) and loading any entry parameters into the specified registers. Upon execution of a CALL 5 instruction, CDOS will perform the desired function. When CDOS has finished, it will return to the user program with a RET instruction.

All Z8Ø registers will be preserved by system calls except the F (Flag) register and those containing Return Parameters. The Z8Ø set of Primed Registers are not used by system calls and hence programs may safely use these registers for temporary storage. Entry Parameters are preserved by system calls unless otherwise noted in sections 5.1.2 through 5.1.4.

For more detailed information on how system calls index into the operating system, refer to the CDOS Memory Allocation section (5.3). For a summary of the system calls available with CDOS (listed in numerical order), refer to Section 5.1.5. The system calls are grouped under three general headings and described individually in detail in this chapter. They are also cross-referenced in the Index.

### 5.1.1 Detailed List of CDOS System Calls

The following is a detailed description of the CDOS system calls. They are sub-divided into three sections.

Section 5.1.2 covers Input/Output from/to all devices except disk drives. This includes the system console(s), printer, punch, and reader.

The next section (5.3.3) covers the system calls used to access disk files. This includes functions to search for, create, rename, delete, open and close disk files. Also included are routines to read or write data, and a number of functions designed to manipulate the disk drives.

Section 5.1.4 covers a number of additional calls which fit no specific category and include such functions as integer multiply and divide; set/read the date/time of day; and system abort and program link functions.

The system calls given below are in numerical order in each of the three sections.

All device and disk input and output should be done through the CDOS system calls. This allows user programs to be independent of physical devices or port assignments and assures that the program will be able to run on other Cromemco machines regardless of how I/O devices are connected to those machines. If a change needs to be made in a device driver, it has only to be done once in the system drivers and this change becomes effective in all programs which access that driver through the system calls.

To use one of these routines the C register must be set to the function number given with the title of each instruction. The other registers are set up as that system call requires (for example, the E or DE registers usually contain the entry parameter passed). A CALL 5 instruction is then executed to carry out the function. Remember that CDOS initializes location 5 with a jump instruction. This is done so that the location of CDOS in memory is transparent to a user program. A program using the CDOS system functions does not therefore need to do a CALL to a particular address in CDOS.

### 5.1.2 CDOS Device System Calls

The system calls of this section involve device I/O with all devices except disk drives. The number given preceding each CDOS function is the number which should be loaded into the C register prior to the CALL 5 instruction. This number is given first in decimal and then in hexadecimal in parentheses.

1 - READ CONSOLE (with echo)

This call is used to retrieve a single character (one byte) from the console. The byte will be returned in the A register with the parity bit (Bit 7) reset. CDOS does not return control to the user program until a character has been read and echoed back to the CRT. No entry parameters are required other than the value in the C register.

Note that a CTRL-Z (^Z) character is usually to be considered by a user program as an end of file mark. Also, most other control characters will *not* be echoed back to the CRT and some have special meanings for the operating system. For example, CTRL-J (LF), CTRL-M (CR), and CTRL-G (BEL) are echoed directly, CTRL-I (TAB) is echoed as expanded spaces (see WRITE CONSOLE below), and CTRL-P will toggle on/off a line printer but will not be echoed.

## 2 - WRITE CONSOLE

This call is used to write a single ASCII character (one byte) to the CRT. The character is placed in the E register before the call. CDOS will wait until the console is ready to receive the character and then print it. No parameters are returned by the call.

After CTRL-P (^P) is typed while CDOS is outputting characters with this system call, all subsequent characters are sent to both the console and the printer until CTRL-P is depressed a second time (thus CTRL-P acts as a toggle switch). CTRL-W (^W) also causes subsequent characters to be sent to both the console and the printer but must be encountered in a file to do so. CTRL-T (^T) in a file cancels the effect of either the CTRL-W or the CTRL-P and causes characters to be sent only to the console. CTRL-W and CTRL-T may be edited into a file so when that file is being typed out on the console, it can stop and start the printer at the appropriate places.

CTRL-I is the tab character and is converted to spaces as it is typed out so that the cursor is positioned at one of the standard tab stops: column 1, 9, 17, 25, 33, 41, 49, 57, 65, or 73. However, the tab is still stored internally in a file as a single ASCII character (9).

## 3 - READ READER

This call will read one character from a paper tape or card reader. All 8 bits are read and returned in the A register (i.e., the parity bit is not masked). No entry parameters are required other than the value in the C register. Since no card or paper tape reader is connected to a standard Cromemco computer system, the port assignments and method of interface (default is serial) for this system call are set up initially with the console as a reader.

Also note that console status is checked during the read for the CTRL-S (^S) toggle, enabling the user to stop/start the reading process at will. This is useful for pausing during a paper tape jam, for example.

### 4 - WRITE PUNCH

This call will punch one character on a paper tape punch. All 8 bits are punched (i.e., including parity). The character to be punched is placed in the E register before the call. CDOS will then wait until the punch is ready to receive the character. No parameters are returned by this call.

Also note that console status is checked during the read for CTRL-S (^S), enabling the user to stop/start the punching process. This is useful for pausing during a paper tape jam, for example.

### 5 - WRITE LIST

This call will print a single character (one byte) on the printer. The character to be printed is placed in the E-register before the call, and CDOS will wait for the printer to be ready before returning to a user program. No parameters are returned by this call.

Tabs are not expanded, and control characters which do not have meaning to the printer will be transmitted anyway. Cromemco printers will ignore such control characters. A useful control character for the Cromemco Model 3703 Printer is CTRL-N (^N), which, when present in a line of printer output, will cause that line to be printed in double width characters.

Also note that console status is checked during the print out for the CTRL-S (^S) character, enabling the user to stop/start the listing. This is useful for pausing to start a new box of line printer paper.

### 7 - GET I/O BYTE

For extra I/O devices, an IOBYTE has been provided. This byte is not currently used by CDOS, but it is provided for the user's programs. This system call returns the IOBYTE in the A register. The format of the byte is:

| Bit: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|---|
| Device: | PRN | PUNCH | | READER | | CONSOLE | | |

Thus up to eight consoles can be designated, four each of paper-tape punch and reader, and one printer.

### 8 - SET I/O BYTE

This call allows the user program to set the IOBYTE. The E register contains the byte prior to the call. See GET I/O BYTE for the format of the byte.

### 9 - PRINT BUFFERED LINE

This call will print a string of ASCII characters which has been terminated with the dollar sign ($) character. The DE register pair is set up with the address of the beginning of the string before the call is made to CDOS. If the printer toggle is on, the line will also be sent to the printer.

### 1Ø (ØAH) - INPUT BUFFERED LINE

This call will read an input line from the console. The DE register must be pointing to an available buffer before the call is made to CDOS. The first byte of the buffer must contain the maximum length of the buffer. On return from this call the second byte of the buffer will contain the actual length entered. The line that is input will be stored beginning at the third byte. If the buffer is not full, the byte at the end of the line will contain a zero.

When the line is being entered, the following characters will have special meaning:

CTRL-C (^C) Abort. Warm boot back to CDOS.

CTRL-E (^E) Physical CR-LF. The line is not terminated and nothing is entered into the buffer. This character is used to enter a line longer than can be entered on the console.

CTRL-P (^P) Toggle printer/console link. When this character is first typed, the link is toggled ON. All characters will then be sent to the console and the printer. The next time the character is typed, the toggle will be turned off. All characters will then be sent only to the console.

CTRL-R (^R) Repeat what has been typed so far on the line.

CTRL-U (^U) Delete the entered line and go back to beginning of buffer for new line.

CTRL-X (^X) Delete the previous character and echo the deleted character (used for hard-copy terminals).

RUBout Delete the previous character and back up the cursor (used for CRT terminals).

DEL Same as RUBout.

Underscore Same as RUBout.

Backspace (^H) Same as RUBout.

### 11 (ØBH) - TEST CONSOLE READY

The console is tested to see if a character has been typed. If a character has been typed, ØFFH is returned in the A register. If no character has been typed, Ø is returned in the A register.

### 128 (8ØH) - READ CONSOLE (without echo)

This call is the same as READ CONSOLE (with echo) except that it does not echo the character after it is read. The byte is returned in the A register.

### 142 (8EH) - SET SPECIAL CRT FUNCTION

This call is used to perform special functions on the system console terminal. The call is designed to be very broad and include as many of the special features available in present-day intelligent terminals as possible. In particular it allows the programmer to take full advantage of the features available in Cromemco Model 31ØØ and Model 31Ø1 CRT terminals. The call is executed by first loading the correct entry parameters into the DE register-pair. These values are summarized in the following table. No parameters are returned by this call.

| Mnemonic | | Function | D | E |
|---|---|---|---|---|
| Address | * | address cursor on screen | 1-8Ø | 1-24 |
| Clear | * | clear CRT screen | Ø | Ø |
| Home | * | home cursor without clearing | 1 | Ø |
| Back | * | move cursor to left one character position | 2 | Ø |
| Forward | * | move cursor to right one character position | 3 | Ø |
| Up | * | move cursor up one line | 4 | Ø |
| Down | * | move cursor down one line | 5 | Ø |
| Clear-EOL | * | clear from cursor position to end of line | 6 | Ø |
| Clear-EOS | * | clear from cursor position to end of screen | 7 | Ø |
| Highlight | | set intensity to high-light | 8 | Ø |
| Lowlight | | set intensity to low-light | 9 | Ø |
| Normlight | | set intensity to normal-light | 1Ø | Ø |
| Keybd-on | * | enable keyboard | 11 | Ø |
| Keybd-off | * | disable keyboard | 12 | Ø |
| Curpd-on | | enable cursor pad | 13 | Ø |
| Curpd-off | | disable cursor pad | 14 | Ø |
| Prtec-on | * | begin protected field | 15 | Ø |
| Prtec-off | * | end protected field | 16 | Ø |
| Blink-on | * | begin blinking characters | 17 | Ø |

| | | | | |
|---|---|---|---|---|
| Blink-off | * | end blinking characters | 18 | Ø |
| Line-send | * | send from cursor position to end of line | 19 | Ø |
| Page-send | * | send from cursor position to end of screen | 2Ø | Ø |
| Aux-send | * | transmit screen out auxiliary port | 21 | Ø |
| Del-char | * | delete character at present cursor position | 22 | Ø |
| Insr-char | | insert character at present cursor position | 23 | Ø |
| Del-line | | delete line at present cursor position | 24 | Ø |
| Insr-line | | insert line at present cursor position | 25 | Ø |
| Formt-on | * | turn on formatted screen | 26 | Ø |
| Formt-off | * | turn off formatted screen | 27 | Ø |
| Rever-on | | begin reverse background field | 28 | Ø |
| Rever-off | | end reverse background field | 29 | Ø |

Those features marked with an asterisk (*) above are all standard features of a Cromemco Model 31Ø1 System Terminal. Also note from the above chart that the E register is always loaded with Ø to select any special CRT function except cursor addressing. For cursor addressing the D register should contain the column address (1 through 8Ø for Cromemco CRTs) and the E register should contain the row address (1 through 24 for Cromemco CRTs) of the desired cursor position. The system call will generate no error if these values are exceeded. Note: On some CRTs addressing the cursor at a non-existent location may cause it to disappear from the screen.

For reference, the location (1,1) is considered to be the upper left-hand corner and the location (8Ø,24) the lower right-hand corner of the screen.

### 5.1.3 CDOS Disk System Calls

CDOS divides the disk into regions called files. Files are referenced through file control blocks (FCBs). FCBs are 33 bytes long and have the following format, where each of the numbers below stands for one byte:

| | | | |
|---|---|---|---|
| FCBDK | Disk descriptor | Ø | (Ø=current disk, 1=drive-A, 2=B, 3=C, 4=D) |
| FCBFN | File name | 1...8 | (right-filled with blanks) |
| FCBFT | File type (extension) | 9...11 | (right-filled with blanks) |
| FCBEX | File entry or extent | 12 | (initially Ø; is incremented by one in every new entry of 16 Kbytes) |
| | Reserved | 13...14 | |
| FCBRC | Record count | 15 | (total number of 128-byte sectors or records) |
| FCBMP | Cluster allocation map | 16...31 | (allocated clusters 2 through 24Ø) |
| FCBNR | Next record | 32 | (next record to be read or written; has the value Ø through 127) |

Note: The directory entries on the disk consist of 32 byte FCBs. The last byte, FCBNR, which points to the next record, is omitted.

12 (ØCH) - DESELECT CURRENT DISK

The current disk is deselected. The CDOS disk driver can be changed to perform any desired function at this time to deselect the disk. Currently the driver outputs a Ø to port 34H when this function is selected.

### 13 (ØDH) - RESET CDOS AND SELECT DRIVE A

CDOS is initialized, all disks are logged-off, and drive A is selected as the current drive. The other disks will be logged-on again as soon as they are accessed.

### 14 (ØEH) - SELECT DISK DRIVE

The disk drive number in the E register is selected as the current disk. The drive number in the E register is Ø for drive A, 1 for drive B, 2 for drive C, or 3 for drive D.

### 15 (ØFH) - OPEN DISK FILE

The FCB pointed to by the DE register pair is opened to allow reading or writing to the file whose name is specified in the FCB. The A register returns with -1 (ØFFH or 255D) if the file is not found, or the directory block number if the file is found. Block numbers start at Ø and there is one block number for every four directory entries. The DE register pair returns pointing to the directory entry in memory.

### 16 (1ØH) - CLOSE DISK FILE

The file described by the File Control Block pointed to by the DE register pair is closed, and the disk directory is updated (i.e., the FCB containing updated cluster information is written to the disk). The A register is returned with -1 (FFH) if the file is not found on the drive, or the directory block number if the file is found on the stated drive. The file described by the FCB should have been previously opened or created. A file to which bytes have just been written must be closed using this function or the entire last entry (or extent) will be unable to be read (i.e., no cluster information will be present for this entry in the directory).

## 17 (11H) - SEARCH DIRECTORY FOR FILENAME

The directory is searched for the first occurrence of the file specified in the FCB pointed to by the DE register pair. ASCII question mark (? - 3FH) in the FCB matches any character. The block number (see description of directory block numbers in ØFH - Open Disk File, above) is returned in the A register if found, if the file is not found -1 (ØFFH or 255D) is returned in A. HL is returned pointing to the directory entry in memory. An important point to note about this call and the one following (12H) is that they will get the directory entry whether it has been erased or not; i.e., these calls do not check to see if a file has been erased. Files are erased by placing a ØE5H in the first byte (FCBDK); the rest of the FCB is left unchanged.

## 18 (12H) - FIND NEXT DIRECTORY ENTRY

This call is the same as 11H (17) above except that it finds the NEXT occurrence of the filename in the directory. This may be either the next entry of a file occupying several entries (extents), or another filename if the question mark match character (?) is used in the FCB. This call is made after system call 17 and no other disk system function can be executed between these calls.

## 19 (13H) - DELETE FILE

The file specified by the FCB pointed to by the DE register pair is deleted from the disk directory. ASCII question mark (?) in the FCB matches any character. The number of directory entries deleted is returned in the A register.

### 2Ø (14H) - READ NEXT RECORD

The DE register pair points to a successfully OPENED FCB. The next record (128 bytes) is read into the current disk buffer. The FCBNR in the FCB is incremented to read the next record. One of the following codes is returned in the A register:

Ø - read completed

1 - end of file

2 - read attempted on unwritten cluster (random access files only)

### 21 (15H) - WRITE NEXT RECORD

The DE register pair points to a successfully OPENED FCB. The next record (128 bytes) is written into the file from the current disk buffer. The FCBNR in the FCB is incremented to be ready to write the next record. One of the following codes is returned in the A register:

Ø - write completed

1 - entry error (attempted to close an unopened entry)

2 - out of disk space (limited to 81K for small; 241K for large)

-1 - (or FFH) out of directory space (limited to 64 entries)

### 22 (16H) - CREATE FILE

The file specified in the FCB pointed to by the DE register pair is created on the disk. The A register is returned containing the block number of the directory entry (see ØFH - Open Disk File), or -1 (ØFFH or 255) if no more directory space is available.

### 23 (17H) - RENAME FILE

This call will rename a disk file. The DE register pair points to the FCB to be renamed. The old file name and file type are in the first 16 bytes and the new file name and file type are in the second 16 bytes of the FCB. ASCII question mark (?) in the FCB will match with any character. The A register returns containing the number of directory entries renamed.

### 24 (18H) - GET DISK LOG-IN VECTOR

The A register is returned specifying the disks that are logged in. Each bit represents one disk drive logged in. If the bit is a one, then it is logged in; else it is off-line. The least significant bit is the A drive, next most significant (Bit 1) is drive B, etc. Since there would be no more than four drives, the upper four bits are Ø's.

### 25 (19H) - GET CURRENT DISK

The number of the current disk drive is returned in the A register. Ø = drive A, 1 = drive B, 2 = drive C, 3 = drive D.

### 26 (1AH) - SET DISK BUFFER

The buffer pointed to by the DE register pair is used for disk I/O. When a program is loaded, the disk buffer is initially located at 8ØH.

## 27 (1BH) - GET DISK CLUSTER ALLOCATION MAP

The BC register pair returns pointing to a bit map that corresponds to the allocated clusters on the disk. The DE register pair returns containing the capacity of the current disk in number of clusters. The A register returns containing the number of records or sectors per cluster (8). This system call is used by the STATus utility program.

## 131 (83H) - READ LOGICAL BLOCK

This system call will read a logical block from the disk without any attention to the files it may contain (i.e., no FCB is specified). A block is defined to be one sector or record of 128 bytes. When this function is called, the DE register pair should contain the block number and the B register should contain the disk number (Ø for current drive, 1-4 for A-D). The high bit of the B register contains a 1 for an interleaved and a Ø for a non-interleaved read. Interleaved means the block which is read is found in the order CDOS stores it (every fifth sector for small disks and every sixth sector for large disks). Non-interleaved means the block which is read is found in sequential order, the order it is physically stored on the disk. The A register is returned with the status of the read according to the following:

Ø - OK
1 - I/O error
2 - illegal request
3 - illegal block

An example will help to illustrate the use of these parameters. CDOS makes use of 716 sectors on the small floppy disks. The block numbers which can legally be loaded into the DE register are Ø through 715 decimal, or Ø through 2CBH. Suppose that DE is loaded with the value 2 and the B register with Ø (current disk, non-interleaved read). Thus, since the sectors are numbered beginning with 1, sector 3 would be read into memory in the disk buffer (located at 8ØH if it has not been changed). The same read with the B register loaded with 8ØH (current disk, interleaved read) would read sector ØBH (the third sector when they are read every fifth one).

### 132 (84H) - WRITE LOGICAL BLOCK

This system call will write a logical block or sector to the disk without any attention to the file there (no FCB is specified). The registers are set up and returned in the same way as they are for the Read Logical Block system call.

### 134 (86H) - FORMAT NAME TO FILE CONTROL BLOCK

This system call will build a File Control Block. The HL register pair points to the start of the input line. The DE register points to the place in memory where the FCB is to be built. The input line is of the format:

```
d:filename.ext
```

where d stands for one of A-D, the filename is up to 8 letters with a 3 letter extension. The FCB is then built from this input line, converting lower case to upper case. The input line is terminated by an ASCII slash (/) or any character with an ASCII value less than 21H (such as a space or carriage return).

On return the HL register pair points to the terminator that ended the build operation. The DE register pair points to the start of the new FCB.

### 135 (87H) - UPDATE DIRECTORY ENTRY

The last disk I/O function called must have been system call 17 or 18, Search Directory or Find Next Entry. The DE register pair points to the FCB used in the system call 17 or 18. The directory entry is then updated on the disk; this means that the entry is written back to the disk without the user having to specify a block. The user merely specifies a filename when calling 17 or 18. This is useful if it is desired to change a directory entry and write it back to the disk.

## 139 (8BH) - HOME DRIVE HEAD

The disk drive specified in the B register (Ø for current drive and 1-4 for drives A-D) is sent a command to HOME the head. The disk drive head will return to track Ø.

## 14Ø (8CH) - EJECT DISKETTE

This call will eject the disk whose number is given in the E register (Ø for current drive and 1-4 for drives A-D, respectively), only if the disk drive is a CROMEMCO Dual Disk Drive System, Model PFD with the eject option. Otherwise, the call will have no effect.

## 5.1.4 Miscellaneous System Calls

A number of miscellaneous CDOS system calls have been added for the programmer's convenience. These calls are explained in this section.

### Ø - PROGRAM ABORT

This call will abort the current program and return control to CDOS. This call has the same effect as jumping to location Ø.

### 129 (81H) - GET USER-REGISTER POINTER

This call is provided for expansion of CDOS to a multiprogramming system. The BC register pair returns pointing to the user register pointers.

### 13Ø (82H) - SET USER CONTROL-C ABORT

When CTRL-C (^C) is typed, the system normally aborts and returns control to CDOS. This call allows the programmer to change the address to which control is transferred when CTRL-C is typed (i.e., a user may assign a new function to CTRL-C). The address is given in the DE register pair. Note that if DE contains a zero, the system abort is reset. Jumping to location Ø at any time still causes a return to CDOS, also with the CTRL-C being restored to its original function.

## 136 (88H) - LINK TO NEW PROGRAM

This enables one command program to call another. The default command-line buffer and default FCBs for the new program must be set up prior to this call if that program expects to be able to use them. The DE register pair should contain the address of the FCB of the new program (which must have an extension of COM). If the new program is NOT found, the A register returns containing -1 (ØFFH or 255). In this case the first 8ØH bytes (from 1ØØH to 17FH) will be destroyed because this is used in reading the directory. If the program is found execution begins at 1ØØH and no return is made to the original program.

## 137 (89H) - MULTIPLY INTEGERS

This system call provides a 16 bit multiply. The HL and DE register pairs contain the two 16-bit factors, and the answer is returned in register DE (i.e., DE = DE*HL).

## 138 (8AH) - DIVIDE INTEGERS

This system call provides a 16-bit divide. The HL register pair should contain the dividend, and the DE register pair, the divisor. The quotient is returned in HL, and the remainder in DE (i.e., HL = HL/DE with DE = remainder).

## 141 (8DH) - GET CDOS VERSION AND RELEASE NUMBERS

This call will return the version number of CDOS in the B register and the release number in the C register.

### 143 (8FH) - SET CALENDAR DATE

This call is used to store the date (day/mon/yr) in CDOS. Upon entry to this call, the B register contains the day, the D register the month, and the E register the year minus 19ØØ. These values will be stored in locations in CDOS where they may be accessed by user programs (through system call 144) and thus added to listings or other output. No parameters are returned by this call.

The operating system makes no check for the correctness or plausibility of the incoming values; thus, it is up to the user to supply this error-checking. Also, the date is not stored on the disk and is thus volatile (will be lost if the user reboots or turns off the power).

### 144 (9ØH) - READ CALENDAR DATE

This call is used to retrieve the date (day/mon/yr) stored in CDOS by system call 143. The day is returned in the A register, the month in the B register, and the year minus 19ØØ in the C register. No entry parameters are required other than the value in the C register. Note that the C register is changed by this call unlike most other system calls which preserve C.

This is the function which should be used by a program to recover the last previously stored date from the operating system. Note that if Set Date has not yet been used, Read Date will return the values ØØ/ØØ/ØØ.

### 145 (91H) - SET TIME OF DAY

This call is used to store the time of day (sec/min/hr) in CDOS for use by a hardware clock or user program. Upon entry to the call, the B register contains the seconds, the D register the minutes, and the E register the hours in 24-hour time. These values will be stored in locations in CDOS where they may either be accessed and updated by user programs or may in turn be stored in registers of an electronic clock. No parameters are returned by this call.

The operating system makes no check for the correctness or plausibility of the incoming values. It is up to the user to supply this error checking. Note in the I/O Device Drivers that a dummy routine is supplied to Start Clock. This dummy routine is called by the operating system during the Set Time function; thus, users may substitute their own routine in the drivers to initialize a hardware clock.

### 146 (92H) - READ TIME OF DAY

This call is used to retrieve the time of day (sec/min/hr) stored in CDOS by system call 145. The seconds are returned in the A register, the minutes in the B register, and hours (24 hour time) in the C register. Note that the C register is changed by this call unlike most other system calls which preserve C.

This is the function which should be used by a program to recover the last previously stored time from the operating system. Note that if Set Time has not yet been used, Read Time will return the values ØØ/ØØ/ØØ.

The I/O Device Drivers contain a dummy routine to Read Clock. This dummy routine is called by CDOS during the Read Time system call. Thus, users may substitute their own routine in the drivers to read the time from a hardware clock and store it in the time registers also supplied in the drivers.

### 15Ø (96H) - TURN DRIVE MOTORS OFF

This call is used to turn off the disk drive motors. It may be used by any program which will perform its primary function in memory over a long period of time during which there will be few disk accesses (e.g., an editor or interpreter). No parameters are required on entry or given on return from this call other than the value in the C register.

Note that there is no corollary call to turn the motors on. This will be performed automatically by the operating system the next time any disk operation is attempted. CDOS will also pause for approximately 1 second after turning on the motors and before accessing the disk only if the Motor Off call has been issued. This is to allow the motors to come up to speed before the disk is accessed.

## 151 (97H) - SET BOTTOM OF CDOS IN RAM

This call is used to set the bottom address of CDOS to a lower value than the one at which CDOS was originally loaded when it was booted up. The high byte of the address of the new bottom is placed into the E register prior to executing the call. The low byte is assumed Ø; thus, the bottom of CDOS can never be located on any address other than a 256 byte boundary. If the value is -1 (FFH) or any other value greater than the high byte of the original bottom when booting up, CDOS will restore this original bottom address. No parameters are returned by the call.

This function will change the system call jump at locations 5, 6, and 7. Programs using the address at locations 6 and 7 to determine the size of the present User Area will find this area to be reduced in size. A second set of jumps (9 bytes) will be loaded at the new bottom of CDOS which points to the old bottom so that system calls will still execute correctly. Note that CDOS is in no way relocated by this function and will reside in the same memory space as it did previously. The purpose of the call is to make it possible to attach a permanent patch space to CDOS for programs which are to become a permanent part of the operating system for as long as it resides in memory. The only way the patch space may be removed is by a second Set Bottom call.

### 5.1.5 Summary of CDOS System Calls

Following is a summary table listing all the system calls described in Chapter 5 along with their entry and return parameters. The system calls are listed in numerical order, i.e., by order of the number which is loaded into the C register to achieve the desired function.

| NUMBER | FUNCTION | ENTRY PARAMETERS | RETURN PARAMETERS |
|---|---|---|---|
| Ø | PROGRAM ABORT | none | none |
| 1 | READ CONSOLE (with echo) | none | A = character (parity bit reset) |
| 2 | WRITE CONSOLE | E = character | none |
| 3 | READ READER | none | A = character |
| 4 | WRITE PUNCH | E = character | none |
| 5 | WRITE LIST | E = character | none |
| 6 | (not used presently - reserved for expansion) | | |
| 7 | GET I/O BYTE | none | A = I/O byte |
| 8 | SET I/O BYTE | E = I/O byte | none |
| 9 | PRINT BUFFERED LINE | DE = buffer address | none |
| 1Ø (ØAH) | INPUT BUFFERED LINE | DE = buffer address | none |
| 11 (ØBH) | TEST CONSOLE READY | none | A = -1 (FFH) if ready<br>A = Ø if not ready |
| 12 (ØCH) | DESELECT CURRENT DISK | none | none |
| 13 (ØDH) | RESET CDOS AND SELECT DRIVE A | none | none |
| 14 (ØEH) | SELECT CURRENT DISK | E = disk drive no. | none |
| 15 (ØFH) | OPEN DISK FILE | DE = FCB address | A = directory block<br>A = -1 (FFH) if not found |
| 16 (1ØH) | CLOSE DISK FILE | DE = FCB address | A = directory block<br>A = -1 (FFH) if not found |
| 17 (11H) | SEARCH DIRECTORY FOR FILENAME | DE = FCB address | A = directory block<br>A = -1 (FFH) if not found |
| 18 (12H) | FIND NEXT ENTRY IN DIRECTORY | DE = FCB address | A = directory block<br>A = -1 (FFH) if not found |
| 19 (13H) | DELETE FILE | DE = FCB address | A = number of entries deleted |
| 2Ø (14H) | READ NEXT RECORD | DE = FCB address | A = Ø if OK<br>A = 1 if end of file<br>A = 2 if tried to read unwritten records |

| | | | | |
|---|---|---|---|---|
| 21 | (15H) | WRITE NEXT RECORD | DE = FCB address | A = Ø if OK<br>A = 1 if entry error<br>A = 2 if out of disk space<br>A = -1 (FFH) if out of directory space |
| 22 | (16H) | CREATE FILE | DE = FCB address | A = directory block<br>A = -1 (FFH) if out of directory space |
| 23 | (17H) | RENAME FILE | DE = FCB address | A = number of entries renamed |
| 24 | (18H) | GET DISK LOG-IN VECTOR | none | A = those disks currently logged-in |
| 25 | (19H) | CURRENT DISK | none | A = disk drive number |
| 26 | (1AH) | SET DISK BUFFER | DE = buffer address | none |
| 27 | (1BH) | DISK CLUSTER ALLOCATION MAP | none | BC = address of bitmap<br>DE = number of clusters<br>A = sectors/cluster |
| 128 | (8ØH) | READ CONSOLE (with no echo) | none | A = character |
| 129 | (81H) | GET USER REGISTER POINTER | none | BC = pointer to user register pointers |
| 130 | (82H) | SET USER CTRL-C ABORT | DE = address of ^C handler (Ø to reset; -1 to disable) | none |
| 131 | (83H) | READ LOGICAL BLOCK | DE = block number<br>B = drive number<br>B top bit = 1 if interleaved | A = Ø if OK<br>A = 1 if I/O error<br>A = 2 if illegal request<br>A = 3 if illegal block |
| 132 | (84H) | WRITE LOGICAL BLOCK | DE = block number<br>B = drive number<br>B top bit = 1 if interleaved | A = Ø if OK<br>A = 1 if I/O error<br>A = 2 if illegal request<br>A = 3 if illegal block |
| 133 | (85H) | (not used presently - reserved for expansion) | | |
| 134 | (86H) | FORMAT NAME TO FILE CONTROL BLOCK | HL = address of string<br>DE = FCB address | HL = address of terminator<br>DE = FCB address |
| 135 | (87H) | UPDATE DIRECTORY ENTRY | DE = FCB address | none |
| 136 | (88H) | LINK TO PROGRAM | DE = FCB address | A = -1 (FFH) if error; else execute at 1ØØH |
| 137 | (89H) | MULTIPLY INTEGERS | DE = factor 1<br>HL = factor 2 | DE = product |
| 138 | (8AH) | DIVIDE INTEGERS | HL = dividend<br>DE = divisor | HL = quotient<br>DE = remainder |

| | | | | |
|---|---|---|---|---|
| 139 | (8BH) | HOME DRIVE | B = drive number | none |
| 14Ø | (8CH) | EJECT DISKETTE | E = drive number | none |
| 141 | (8DH) | GET VERSION OF OPERATING SYSTEM | none | B = version-number<br>C = release-number |
| 142 | (8EH) | SET SPECIAL CRT FUNCTION | D = column address/ special function<br>E = row address/Ø | none |
| 143 | (8FH) | SET DATE | B = day<br>D = month<br>E = year-19ØØ | none |
| 144 | (9ØH) | READ DATE | none | A = day<br>B = month<br>C = year-19ØØ |
| 145 | (91H) | SET TIME OF DAY | B = seconds<br>D = minutes<br>E = hours (24 hr. time) | none |
| 146 | (92H) | READ TIME OF DAY | none | A = seconds<br>B = minutes<br>C = hours (24 hr. time) |
| 147 | (93H) | SET PROGRAM RETURN CODE | E = return code for next program | A = previously set return code |
| 148 | (94H) | SET FILE ATTRIBUTES | DE = FCB address<br>B = new attributes | none |
| 149 | (95H) | READ DISK LABEL | none | DE = FCB address |
| 15Ø | (96H) | TURN MOTORS OFF | none | none |
| 151 | (97H) | SET BOTTOM OF CDOS IN RAM | E = high byte of address of bottom of CDOS | none |

## 5.2 CDOS-CP/M Compatibility

The Cromemco Disk Operating System (CDOS)* is an original product designed and written in Z80 machine code by Cromemco, Inc. for its own line of microcomputers. However, due to the large number of programs currently available to run under the CP/M** operating system, CDOS was designed to be upwards CP/M-compatible. Cromemco is licensed by Digital Research, the originator of CP/M, for use of the CP/M data structures and user interface. This means that most programs written for CP/M (versions up to and including 1.3) will run without modification under CDOS. This also means that programs written for CDOS will not generally run under CP/M.

There are several advantages to end-users which result from this compatibility. First, users of Cromemco machines are able to draw on the large library of existing CP/M and CP/M compatible programs available on the market. Second, users familiar with CP/M can easily move up to CDOS taking advantage of the many additional features available with CDOS.

The enhancements contained in CDOS but not CP/M are primarily visible in the system calls. CDOS has added a number of new system calls to allow the user even more flexible means of device and disk I/O. System calls are in all cases executed by first loading the C register with the number of the call. In the case of the non-CP/M calls, the parity bit of the 8-bit quantity in C is set to allow for a new range of 128. This is apparent in the Summary of CDOS System Calls, Section 5.1.5.

* CDOS is a Trademark of Cromemco, Inc.
Mountain View, California

**CP/M is a Trademark of Digital Research, Inc.
Pacific Grove, California

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

MICROPROCESADORES: TEORIA Y APLICACIONES

# SOFT Z-80

ABRIL, 1979

Palacio de Minería Calle de Tacuba 5, primer piso. México 1, D. F.

## 5.3 INSTRUCTION OP CODES

This section describes each of the Z-80 instructions and provides tables listing the OP codes for every instruction. In each of these tables the OP codes in bold type are identical to those offered in the 8080A CPU. Also shown is the assembly language mnemonic that is used for each instruction. All instruction OP codes are listed in hexadecimal notation. Single byte OP codes require two hex characters while double byte OP codes require four hex characters. The conversion from hex to binary is repeated here for convenience.

| Hex | | Binary | | Decimal | Hex | | Binary | | Decimal |
|---|---|---|---|---|---|---|---|---|---|
| 0 | = | 0000 | = | 0 | 8 | = | 1000 | = | 8 |
| 1 | = | 0001 | = | 1 | 9 | = | 1001 | = | 9 |
| 2 | = | 0010 | = | 2 | A | = | 1010 | = | 10 |
| 3 | = | 0011 | = | 3 | B | = | 1011 | = | 11 |
| 4 | = | 0100 | = | 4 | C | = | 1100 | = | 12 |
| 5 | = | 0101 | = | 5 | D | = | 1101 | = | 13 |
| 6 | = | 0110 | = | 6 | E | = | 1110 | = | 14 |
| 7 | = | 0111 | = | 7 | F | = | 1111 | = | 15 |

Z-80 instruction mnemonics consist of an OP code and zero, one or two operands. Instructions in which the operand is implied have no operand. Instructions which have only one logical operand or those in which one operand is invariant (such as the Logical OR instruction) are represented by a one operand mnemonic. Instructions which may have two varying operands are represented by two operand mnemonics.

### LOAD AND EXCHANGE

Table 5.3-1 defines the OP code for all of the 8-bit load instructions implemented in the Z-80 CPU. Also shown in this table is the type of addressing used for each instruction. The source of the data is found on the top horizontal row while the destination is specified by the left hand column. For example, load register C from register B uses the OP code 48H. In all of the tables the OP code is specified in hexadecimal notation and the 48H (=0100 1000 binary) code is fetched by the CPU from the external memory during M1 time, decoded and then the register transfer is automatically performed by the CPU.

The assembly language mnemonic for this entire group is LD, followed by the destination followed by the source (LD DEST., SOURCE). Note that several combinations of addressing modes are possible. For example, the source may use register addressing and the destination may be register indirect; such as load the memory location pointed to by register HL with the contents of register D. The OP code for this operation would be 72. The mnemonic for this load instruction would be as follows:

LD (HL), D

The parentheses around the HL means that the contents of HL are used as a pointer to a memory location. In all Z-80 load instruction mnemonics the destination is always listed first, with the source following. The Z-80 assembly language has been defined for ease of programming. Every instruction is self documenting and programs written in Z-80 language are easy to maintain.

Note in table 5.3-1 that some load OP codes that are available in the Z-80 use two bytes. This is an efficient method of memory utilization since 8, 16, 24 or 32 bit instructions are implemented in the Z-80. Thus often utilized instructions such as arithmetic or logical operations are only 8-bits which results in better memory utilization than is achieved with fixed instruction sizes such as 16-bits.

All load instructions using indexed addressing for either the source or destination location actually use three bytes of memory with the third byte being the displacement d. For example a load register E with the operand pointed to by IX with an offset of +8 would be written:

LD E, (IX + 8)

The instruction sequence for this in memory would be:

| | | |
|---|---|---|
| Address A | DD | OP Code |
| A+1 | 5E | |
| A+2 | 08 | Displacement operand |

The two extended addressing instructions are also three byte instructions. For example the instruction to load the accumulator with the operand in memory location 6F32H would be written:

LD A, (6F 32H)

and its instruction sequence would be:

| | | |
|---|---|---|
| Address A | 3A | OP Code |
| A+1 | 32 | low order address |
| A+2 | 6F | high order address |

Notice that the low order portion of the address is always the first operand.

The load immediate instructions for the general purpose 8-bit registers are two-byte instructions. The instruction load register H with the value 36H would be written:

LD H, 36H

and its sequence would be:

| | | |
|---|---|---|
| Address A | 26 | OP Code |
| A+1 | 36 | Operand |

Loading a memory location using indexed addressing for the destination and immediate addressing for the source requires four bytes. For example:

LD (IX - 15), 21H

would appear as:

| | | |
|---|---|---|
| Address A | DD | OP Code |
| A+1 | 36 | |
| A+2 | F1 | displacement (-15 in signed two's complement) |
| A+3 | 21 | operand to load |

Notice that with any indexed addressing the displacement always follows directly after the OP code.

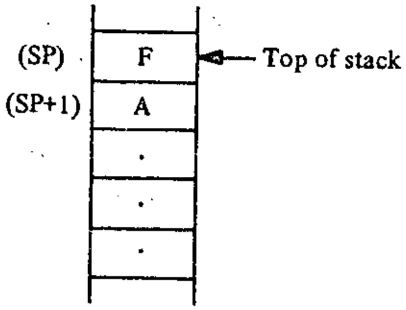
Table 5.3-2 specifies the 16-bit load operations. This table is very similar to the previous one. Notice that the extended addressing capability covers all register pairs. Also notice that register indirect operations specifying the stack pointer are the PUSH and POP instructions. The mnemonic for these instructions is "PUSH" and "POP." These differ from other 16-bit loads in that the stack pointer is automatically decremented and incremented as each byte is pushed onto or popped from the stack respectively. For example the instruction:

PUSH AF

is a single byte instruction with the OP code of F5H. When this instruction is executed the following sequence is generated:

Decrement SP
LD (SP), A
Decrement SP
LD (SP), F

Thus the external stack now appears as follows:



| | | SOURCE | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | IMPLIED | | REGISTER | | | | | | | REG INDIRECT | | | INDEXED | | EXT. ADDR. | IMME. |
| | | I | R | A | B | C | D | E | H | L | (HL) | (BC) | (DE) | (IX + d) | (IY + d) | (nn) | n |
| DESTINATION<br>REGISTER | A | ED<br>57 | ED<br>5F | 7F | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 0A | 1A | DD<br>7E<br>d | FD<br>7E<br>d | 3A<br>n<br>n | 3E<br>n |
| | B | | | 47 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | | | DD<br>46<br>d | FD<br>46<br>d | | 06<br>n |
| | C | | | 4F | 48 | 49 | 4A | 4B | 4C | 4D | 4E | | | DD<br>4E<br>d | FD<br>4E<br>d | | 0E<br>n |
| | D | | | 57 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | | | DD<br>56<br>d | FD<br>56<br>d | | 16<br>n |
| | E | | | 5F | 58 | 59 | 5A | 5B | 5C | 5D | 5E | | | DD<br>5E<br>d | FD<br>5E<br>d | | 1E<br>n |
| | H | | | 67 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | | | DD<br>66<br>d | FD<br>66<br>d | | 26<br>n |
| | L | | | 6F | 68 | 69 | 6A | 6B | 6C | 6D | 6E | | | DD<br>6E<br>d | FD<br>6E<br>d | | 2E<br>n |
| REG INDIRECT | (HL) | | | 77 | 70 | 71 | 72 | 73 | 74 | 75 | | | | | | | 36<br>n |
| | (BC) | | | 02 | | | | | | | | | | | | | |
| | (DE) | | | 12 | | | | | | | | | | | | | |
| INDEXED | (IX+d) | | | DD<br>77<br>d | DD<br>70<br>d | DD<br>71<br>d | DD<br>72<br>d | DD<br>73<br>d | DD<br>74<br>d | DD<br>75<br>d | | | | | | | DD<br>36<br>d<br>n |
| | (IY+d) | | | FD<br>77<br>d | FD<br>70<br>d | FD<br>71<br>d | FD<br>72<br>d | FD<br>73<br>d | FD<br>74<br>d | FD<br>75<br>d | | | | | | | FD<br>36<br>d<br>n |
| EXT. ADDR | (nn) | | | 32<br>n<br>n | | | | | | | | | | | | | |
| IMPLIED | I | | | ED<br>47 | | | | | | | | | | | | | |
| | R | | | ED<br>4F | | | | | | | | | | | | | |

**8 BIT LOAD GROUP**
**'LD'**
**TABLE 5.3–1**

The POP instruction is the exact reverse of a PUSH. Notice that all PUSH and POP instructions utilize a 16-bit operand and the high order byte is always pushed first and popped last. That is a:

PUSH BC is PUSH B then C<br>
PUSH DE is PUSH D then E<br>
PUSH HL is PUSH H then L<br>
POP HL is POP L then H

The instruction using extended immediate addressing for the source obviously requires 2 bytes of data following the OP code. For example:

LD DE, 0659H

will be:

| | | |
|---|---|---|
| Address A | 11 | OP Code |
| A+1 | 59 | Low order operand to register E |
| A+2 | 06 | High order operand to register D |

In all extended immediate or extended addressing modes, the low order byte always appears first after the OP code.

Table 5.3-3 lists the 16-bit exchange instructions implemented in the Z-80. OP code 08H allows the programmer to switch between the two pairs of accumulator flag registers while D9H allows the programmer to switch between the duplicate set of six general purpose registers. These OP codes are only one byte in length to absolutely minimize the time necessary to perform the exchange so that the duplicate banks can be used to effect very fast interrupt response times.

## BLOCK TRANSFER AND SEARCH

Table 5.3-4 lists the extremely powerful block transfer instructions. All of these instructions operate with three registers.

HL points to the source location.<br>
DE points to the destination location.<br>
BC is a byte counter.

After the programmer has initialized these three registers, any of these four instructions may be used. The LDI (Load and Increment) instruction moves one byte from the location pointed to by HL to the location pointed to by DE. Register pairs HL and DE are then automatically incremented and are ready to point to the following locations. The byte counter (register pair BC) is also decremented at this time. This instruction is valuable when blocks of data must be moved but other types of processing are required between each move. The LDIR (Load, increment and repeat) instruction is an extension of the LDI instruction. The same load and increment operation is repeated until the byte counter reaches the count of zero. Thus, this single instruction can move any block of data from one location to any other.

Note that since 16-bit registers are used, the size of the block can be up to 64K bytes (1K = 1024) long and it can be moved from any location in memory to any other location. Furthermore the blocks can be overlapping since there are absolutely no constraints on the data that is used in the three register pairs.

The LDD and LDDR instructions are very similar to the LDI and LDIR. The only difference is that register pairs HL and DE are decremented after every move so that a block transfer starts from the highest address of the designated block rather than the lowest.

|  |  |  | SOURCE |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | REGISTER |  |  |  |  |  | IMM. EXT. | EXT. ADDR. | REG. INDIR. |
|  |  |  | AF | BC | DE | HL | SP | IX | IY | nn | (nn) | (SP) |
| DESTINATION | REGISTER | AF |  |  |  |  |  |  |  |  |  | F1 |
|  |  | BC |  |  |  |  |  |  |  | 01<br>n<br>n | ED<br>4B<br>n<br>n | C1 |
|  |  | DE |  |  |  |  |  |  |  | 11<br>n<br>n | ED<br>5B<br>n<br>n | D1 |
|  |  | HL |  |  |  |  |  |  |  | 21<br>n<br>n | 2A<br>n<br>n | E1 |
|  |  | SP |  |  |  | F9 |  | DD<br>F9 | FD<br>F9 | 31<br>n<br>n | ED<br>7B<br>n<br>n |  |
|  |  | IX |  |  |  |  |  |  |  | DD<br>21<br>n<br>n | DD<br>2A<br>n<br>n | DD<br>E1 |
|  |  | IY |  |  |  |  |  |  |  | FD<br>21<br>n<br>n | FD<br>2A<br>n<br>n | FD<br>E1 |
|  | EXT. ADDR. | (nn) |  | ED<br>43<br>n<br>n | ED<br>53<br>n<br>n | 22<br>n<br>n | ED<br>73<br>n<br>n | DD<br>22<br>n<br>n | FD<br>22<br>n<br>n |  |  |  |
| PUSH INSTRUCTIONS → | REG. IND. | (SP) | F5 | C5 | D5 | E5 |  | DD<br>E5 | FD<br>E5 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | ↑<br>POP INSTRUCTIONS |

NOTE: The Push & Pop Instructions adjust the SP after every execution

**16 BIT LOAD GROUP**
**'LD'**
**'PUSH' AND 'POP'**
**TABLE 5.3–2**

|  |  | IMPLIED ADDRESSING |  |  |  |  |
|---|---|---|---|---|---|---|
|  |  | AF' | BC', DE' & HL' | HL | IX | IY |
| IMPLIED | AF | 08 |  |  |  |  |
|  | BC,<br>DE<br>&<br>HL |  | D9 |  |  |  |
|  | DE |  |  | EB |  |  |
| REG. INDIR. | (SP) |  |  | E3 | DD<br>E3 | FD<br>E3 |

**EXCHANGES**
**'EX' AND 'EXX'**
**TABLE 5.3–3**

| | | | SOURCE | |
|---|---|---|---|---|
| | | | REG. INDIR. | |
| | | | (HL) | |
| DESTINATION | REG. INDIR. | (DE) | ED<br>A0 | 'LDI' – Load (DE)←—(HL)<br>Inc HL & DE, Dec BC |
| | | | ED<br>B0 | 'LDIR,' – Load (DE)←—(HL)<br>Inc HL & DE, Dec BC, Repeat until BC = 0 |
| | | | ED<br>A8 | 'LDD' – Load (DE)←—(HL)<br>Dec HL & DE, Dec BC |
| | | | ED<br>B8 | 'LDDR' – Load (DE)←—(HL)<br>Dec HL & DE, Dec BC, Repeat until BC = 0 |

| | | |
|---|---|---|
| Reg | HL | points to source |
| Reg | DE | points to destination |
| Reg | BC | is byte counter |

BLOCK TRANSFER GROUP
TABLE 5.3–4

Table 5.3-5 specifies the OP codes for the four block search instructions. The first, CPI (compare and increment) compares the data in the accumulator, with the contents of the memory location pointed to by register HL. The result of the compare is stored in one of the flag bits (see section 6.0 for a detailed explanation of the flag operations) and the HL register pair is then incremented and the byte counter (register pair BC) is decremented.

The instruction CPIR is merely an extension of the CPI instruction in which the compare is repeated until either a match is found or the byte counter (register pair BC) becomes zero. Thus, this single instruction can search the entire memory for any 8-bit character.

The CPD (Compare and Decrement) and CPDR (Compare, Decrement and Repeat) are similar instructions, their only difference being that they decrement HL after every compare so that they search the memory in the opposite direction. (The search is started at the highest location in the memory block).

It should be emphasized again that these block transfer and compare instructions are extremely powerful in string manipulation applications.

## ARITHMETIC AND LOGICAL

Table 5.3-6 lists all of the 8-bit arithmetic operations that can be performed with the accumulator, also listed are the increment (INC) and decrement (DEC) instructions. In all of these instructions, except INC and DEC, the specified 8-bit operation is performed between the data in the accumulator and the source data specified in the table. The result of the operation is placed in the accumulator with the exception of compare (CP) that leaves the accumulator unaffected. All of these operations affect the flag register as a result of the specified operation. (Section 6.0 provides all of the details on how the flags are affected by any instruction type). INC and DEC instructions specify a register or a memory location as both source and destination of the result. When the source operand is addressed using the index registers the displacement must follow directly. With immediate addressing the actual operand will follow directly. For example the instruction:

AND 07H

would appear as:

| | | |
|---|---|---|
| Address A | E6 | OP Code |
| A+1 | 07 | Operand |

SEARCH LOCATION

| REG. INDIR. | |
|---|---|
| (HL) | |
| ED<br>A1 | 'CPI'<br>Inc HL, Dec BC |
| ED<br>B1 | 'CPIR', Inc HL, Dec BC<br>repeat until BC = 0 or find match |
| ED<br>A9 | 'CPD' Dec HL & BC |
| ED<br>B9 | 'CPDR' Dec HL & BC<br>Repeat until BC = 0 or find match |

HL points to location in memory
to be compared with accumulator
contents
BC is byte counter

BLOCK SEARCH GROUP
TABLE 5.3–5

Assuming that the accumulator contained the value F3H the result of03H would be placed in the accumulator:

| | |
|---|---|
| Acc before operation | 1111 0011 = F3H |
| Operand | 0000 0111 = 07H |
| Result to Acc | 0000 0011 = 03H |

The Add instruction (ADD) performs a binary add between the data in the source location and the data in the accumulator. The subtract (SUB) does a binary subtraction. When the add with carry is specified (ADC) or the subtract with carry (SBC), then the carry flag is also added or subtracted respectively. The flags and decimal adjust instruction (DAA) in the Z-80 (fully described in section 6.0) allow arithmetic operations for:

multiprecision packed BCD numbers

multiprecision signed or unsigned binary numbers

multiprecision two's complement signed numbers

Other instructions in this group are logical and (AND), logical or (OR), exclusive or (XOR) and compare (CP).

There are five general purpose arithmetic instructions that operate on the accumulator or carry flag. These five are listed in table 5.3-7. The decimal adjust instruction can adjust for subtraction as well as addition, thus making BCD arithmetic operations simple. Note that to allow for this operation the flag N is used. This flag is set if the last arithmetic operation was a subtract. The negate accumulator (NEG) instruction forms the two's complement of the number in the accumulator. Finally notice that a reset carry instruction is not included in the Z-80 since this operation can be easily achieved through other instructions such as a logical AND of the accumulator with itself.

Table 5.3-8 lists all of the 16-bit arithmetic operations between 16-bit registers. There are five groups of instructions including add with carry and subtract with carry. ADC and SBC affect all of the flags. These two groups simplify address calculation operations or other 16-bit arithmetic operations.

SOURCE

| | REGISTER ADDRESSING | | | | | | | REG. INDIR. | INDEXED | | IMMED. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) | n |
| 'ADD' | 87 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | DD<br>86<br>d | FD<br>86<br>d | C6<br>n |
| ADD w CARRY 'ADC' | 8F | 88 | 89 | 8A | 8B | 8C | 8D | 8E | DD<br>8E<br>d | FD<br>8E<br>d | CE<br>n |
| SUBTRACT 'SUB' | 97 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | DD<br>96<br>d | FD<br>96<br>d | D6<br>n |
| SUB w CARRY 'SBC' | 9F | 98 | 99 | 9A | 9B | 9C | 9D | 9E | DD<br>9E<br>d | FD<br>9E<br>d | DE<br>n |
| 'AND' | A7 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | DD<br>A6<br>d | FD<br>A6<br>d | E6<br>n |
| 'XOR' | AF | A8 | A9 | AA | AB | AC | AD | AE | DD<br>AE<br>d | FD<br>AE<br>d | EE<br>n |
| 'OR' | B7 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | DD<br>B6<br>d | FD<br>B6<br>d | F6<br>n |
| COMPARE 'CP' | BF | B8 | B9 | BA | BB | BC | BD | BE | DD<br>BE<br>d | FD<br>BE<br>d | FE<br>n |
| INCREMENT 'INC' | 3C | 04 | 0C | 14 | 1C | 24 | 2C | 34 | DD<br>34<br>d | FD<br>34<br>d | |
| DECREMENT 'DEC' | 3D | 05 | 0D | 15 | 1D | 25 | 2D | 35 | DD<br>35<br>d | FD<br>35<br>d | |

8 BIT ARITHMETIC AND LOGIC
TABLE 5.3–6

| | |
|---|---|
| Decimal Adjust Acc, 'DAA' | 27 |
| Complement Acc, 'CPL' | 2F |
| Negate Acc, 'NEG'<br>(2's complement) | ED<br>44 |
| Complement Carry Flag, 'CCF' | 3F |
| Set Carry Flag, 'SCF' | 37 |

GENERAL PURPOSE AF OPERATIONS
TABLE 5.3-7

|  |  |  | SOURCE |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  | BC | DE | HL | SP | IX | IY |
| DESTINATION | 'ADD' | HL | 09 | 19 | 29 | 39 |  |  |
|  |  | IX | DD<br>09 | DD<br>19 |  | DD<br>39 | DD<br>29 |  |
|  |  | IY | FD<br>09 | FD<br>19 |  | FD<br>39 |  | FD<br>29 |
|  | ADD WITH CARRY AND SET FLAGS 'ADC' | HL | ED<br>4A | ED<br>5A | ED<br>6A | ED<br>7A |  |  |
|  | SUB WITH CARRY AND SET FLAGS 'SBC' | HL | ED<br>42 | ED<br>52 | ED<br>62 | ED<br>72 |  |  |
|  | INCREMENT 'INC' |  | 03 | 13 | 23 | 33 | DD<br>23 | FD<br>23 |
|  | DECREMENT 'DEC' |  | 0B | 1B | 2B | 3B | DD<br>2B | FD<br>2B |

16 BIT ARITHMETIC
TABLE 5.3–8

## ROTATE AND SHIFT

A major capability of the Z-80 is its ability to rotate or shift data in the accumulator, any general purpose register, or any memory location. All of the rotate and shift OP codes are shown in table 5.3-9. Also included in the Z-80 are arithmetic and logical shift operations. These operations are useful in an extremely wide range of applications including integer multiplication and division. Two BCD digit rotate instructions (RRD and RLD) allow a digit in the accumulator to be rotated with the two digits in a memory location pointed to by register pair HL. (See figure 5.3-9). These instructions allow for efficient BCD arithmetic.

## BIT MANIPULATION

The ability to set, reset and test individual bits in a register or memory location is needed in almost every program. These bits may be flags in a general purpose software routine, indications of external control conditions or data packed into memory locations to make memory utilization more efficient.

The Z-80 has the ability to set, reset or test any bit in the accumulator, any general purpose register or any memory location with a single instruction. Table 5.3-10 lists the 240 instructions that are available for this purpose. Register addressing can specify the accumulator or any general purpose register on which the operation is to be performed. Register indirect and indexed addressing are available to operate on external memory locations. Bit test operations set the zero flag (Z) if the tested bit is a zero. (Refer to section 6.0 for further explanation of flag operation).

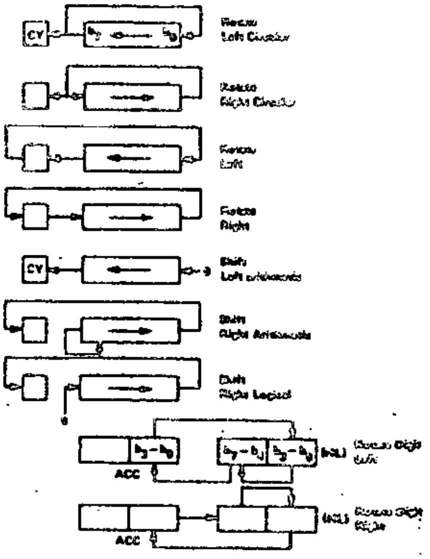
## JUMP, CALL AND RETURN

Figure 5.3-11 lists all of the jump, call and return instructions implemented in the Z-80 CPU. A jump is a branch in a program where the program counter is loaded with the 16-bit value as specified by one of the three available addressing modes (Immediate Extended, Relative or Register Indirect). Notice that the jump group has several different conditions that can be specified to be met before the jump will be made. If these conditions are not met, the program merely continues with the next sequential instruction. The conditions are all dependent on the data in the flag register. (Refer to section 6.0 for details on the flag register). The immediate extended addressing is used to jump to any location in the memory. This instruction requires three bytes (two to specify the 16-bit address) with the low order address byte first followed by the high order address byte.

Source and Destination

| TYPE OF ROTATE OR SHIFT | A | B | C | D | E | H | L | (HL) | (IX + d) | (IY + d) |
|---|---|---|---|---|---|---|---|---|---|---|
| 'RLC' | CB<br>07 | CB<br>00 | CB<br>01 | CB<br>02 | CB<br>03 | CB<br>04 | CB<br>05 | CB<br>06 | DD<br>CB<br>d<br>06 | FD<br>CB<br>d<br>06 |
| 'RRC' | CB<br>0F | CB<br>08 | CB<br>09 | CB<br>0A | CB<br>0B | CB<br>0C | CB<br>0D | CB<br>0E | DD<br>CB<br>d<br>0E | FD<br>CB<br>d<br>0E |
| 'RL' | CB<br>17 | CB<br>10 | CB<br>11 | CB<br>12 | CB<br>13 | CB<br>14 | CB<br>15 | CB<br>16 | DD<br>CB<br>d<br>16 | FD<br>CB<br>d<br>16 |
| 'RR' | CB<br>1F | CB<br>18 | CB<br>19 | CB<br>1A | CB<br>1B | CB<br>1C | CB<br>1D | CB<br>1E | DD<br>CB<br>d<br>1E | FD<br>CB<br>d<br>1E |
| 'SLA' | CB<br>27 | CB<br>20 | CB<br>21 | CB<br>22 | CB<br>23 | CB<br>24 | CB<br>25 | CB<br>26 | DD<br>CB<br>d<br>26 | FD<br>CB<br>d<br>26 |
| 'SRA' | CB<br>2F | CB<br>28 | CB<br>29 | CB<br>2A | CB<br>2B | CB<br>2C | CB<br>2D | CB<br>2E | DD<br>CB<br>d<br>2E | FD<br>CB<br>d<br>2E |
| 'SRL' | CB<br>3F | CB<br>38 | CB<br>39 | CB<br>3A | CB<br>3B | CB<br>3C | CB<br>3D | CB<br>3E | DD<br>CB<br>d<br>3E | FD<br>CB<br>d<br>3E |
| 'RLD' |  |  |  |  |  |  |  | ED<br>6F |  |  |
| 'RRD' |  |  |  |  |  |  |  | ED<br>67 |  |  |

|  | A |
|---|---|
| RLCA | 07 |
| RRCA | 0F |
| RLA | 17 |
| RRA | 1F |



## ROTATES AND SHIFTS
## TABLE 5.3–9

For example an unconditional Jump to memory location 3E32H would be:

| Address A | C3 | OP Code |
|---|---|---|
| A+1 | 32 | Low order address |
| A+2 | 3E | High order address |

The relative jump instruction uses only two bytes, the second byte is a signed two's complement displacement form the existing PC. This displacement can be in the range of +129 to -126 and is measured from the address of the instruction OP code.

Three types of register indirect jumps are also included. These instructions are implemented by loading the register pair HL or one of the index registers IX or IY directly into the PC. This capability allows for program jumps to be a function of previous calculations.

A call is a special form of a jump where the address of the byte following the call instruction is pushed onto the stack before the jump is made. A return instruction is the reverse of a call because the data on the top of the stack is popped directly into the PC to form a jump address. The call and return instructions allow for simple subroutine and interrupt handling. Two special return instructions have been included in the Z-80 family of components. The return from interrupt instruction (RETI) and the return from non maskable interrupt (RETN) are treated in the CPU as an unconditional return identical to the OP code C9H. The difference is that (RETI) can be used at the end of an interrupt routine and all Z-80 peripheral chips will recognize the execution of this instruction for proper control of nested priority interrupt handling. This instruction coupled with the Z-80 peripheral devices implementation simplifies the normal return from nested interrupt. Without this feature the following software sequence would be necessary to inform the interrupting device that the interrupt routine is completed:

|  |  | REGISTER ADDRESSING |  |  |  |  |  |  | REG. INDIR. | INDEXED |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  | BIT | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) |
| TEST 'BIT' | 0 | CB<br>47 | CB<br>40 | CB<br>41 | CB<br>42 | CB<br>43 | CB<br>44 | CB<br>45 | CB<br>46 | DD<br>CB<br>d<br>46 | FD<br>CB<br>d<br>46 |
|  | 1 | CB<br>4F | CB<br>48 | CB<br>49 | CB<br>4A | CB<br>4B | CB<br>4C | CB<br>4D | CB<br>4E | DD<br>CB<br>d<br>4E | FD<br>CB<br>d<br>4E |
|  | 2 | CB<br>57 | CB<br>50 | CB<br>51 | CB<br>52 | CB<br>53 | CB<br>54 | CB<br>55 | CB<br>56 | DD<br>CB<br>d<br>56 | FD<br>CB<br>d<br>56 |
|  | 3 | CB<br>5F | CB<br>58 | CB<br>59 | CB<br>5A | CB<br>5B | CB<br>5C | CB<br>5D | CB<br>5E | DD<br>CB<br>d<br>5E | FD<br>CB<br>d<br>5E |
|  | 4 | CB<br>67 | CB<br>60 | CB<br>61 | CB<br>62 | CB<br>63 | CB<br>64 | CB<br>65 | CB<br>66 | DD<br>CB<br>d<br>66 | FD<br>CB<br>d<br>66 |
|  | 5 | CB<br>6F | CB<br>68 | CB<br>69 | CB<br>6A | CB<br>6B | CB<br>6C | CB<br>6D | CB<br>6E | DD<br>CB<br>d<br>6E | FD<br>CB<br>d<br>6E |
|  | 6 | CB<br>77 | CB<br>70 | CB<br>71 | CB<br>72 | CB<br>73 | CB<br>74 | CB<br>75 | CB<br>76 | DD<br>CB<br>d<br>76 | FD<br>CB<br>d<br>76 |
|  | 7 | CB<br>7F | CB<br>78 | CB<br>79 | CB<br>7A | CB<br>7B | CB<br>7C | CB<br>7D | CB<br>7E | DD<br>CB<br>d<br>7E | FD<br>CB<br>d<br>7E |
| RESET BIT 'RES' | 0 | CB<br>87 | CB<br>80 | CB<br>81 | CB<br>82 | CB<br>83 | CB<br>84 | CB<br>85 | CB<br>86 | DD<br>CB<br>d<br>86 | FD<br>CB<br>d<br>86 |
|  | 1 | CB<br>8F | CB<br>88 | CB<br>89 | CB<br>8A | CB<br>8B | CB<br>8C | CB<br>8D | CB<br>8E | DD<br>CB<br>d<br>8E | FD<br>CB<br>d<br>8E |
|  | 2 | CB<br>97 | CB<br>90 | CB<br>91 | CB<br>92 | CB<br>93 | CB<br>94 | CB<br>95 | CB<br>96 | DD<br>CB<br>d<br>96 | FD<br>CB<br>d<br>96 |
|  | 3 | CB<br>9F | CB<br>98 | CB<br>99 | CB<br>9A | CB<br>9B | CB<br>9C | CB<br>9D | CB<br>9E | DD<br>CB<br>d<br>9E | FD<br>CB<br>d<br>9E |
|  | 4 | CB<br>A7 | CB<br>A0 | CB<br>A1 | CB<br>A2 | CB<br>A3 | CB<br>A4 | CB<br>A5 | CB<br>A6 | DD<br>CB<br>d<br>A6 | FD<br>CB<br>d<br>A6 |
|  | 5 | CB<br>AF | CB<br>A8 | CB<br>A9 | CB<br>AA | CB<br>AB | CB<br>AC | CB<br>AD | CB<br>AE | DD<br>CB<br>d<br>AE | FD<br>CB<br>d<br>AE |
|  | 6 | CB<br>B7 | CB<br>B0 | CB<br>B1 | CB<br>B2 | CB<br>B3 | CB<br>B4 | CB<br>B5 | CB<br>B6 | DD<br>CB<br>d<br>B6 | FD<br>CB<br>d<br>B6 |
|  | 7 | CB<br>BF | CB<br>B8 | CB<br>B9 | CB<br>BA | CB<br>BB | CB<br>BC | CB<br>BD | CB<br>BE | DD<br>CB<br>d<br>BE | FD<br>CB<br>d<br>BE |
| SET BIT 'SET' | 0 | CB<br>C7 | CB<br>C0 | CB<br>C1 | CB<br>C2 | CB<br>C3 | CB<br>C4 | CB<br>C5 | CB<br>C6 | DD<br>CB<br>d<br>C6 | FD<br>CB<br>d<br>C6 |
|  | 1 | CB<br>CF | CB<br>C8 | CB<br>C9 | CB<br>CA | CB<br>CB | CB<br>CC | CB<br>CD | CB<br>CE | DD<br>CB<br>d<br>CE | FD<br>CB<br>d<br>CE |
|  | 2 | CB<br>D7 | CB<br>D0 | CB<br>D1 | CB<br>D2 | CB<br>D3 | CB<br>D4 | CB<br>D5 | CB<br>D6 | DD<br>CB<br>d<br>D6 | FD<br>CB<br>d<br>D6 |
|  | 3 | CB<br>DF | CB<br>D8 | CB<br>D9 | CB<br>DA | CB<br>DB | CB<br>DC | CB<br>DD | CB<br>DE | DD<br>CB<br>d<br>DE | FD<br>CB<br>d<br>DE |
|  | 4 | CB<br>E7 | CB<br>E0 | CB<br>E1 | CB<br>E2 | CB<br>E3 | CB<br>E4 | CB<br>E5 | CB<br>E6 | DD<br>CB<br>d<br>E6 | FD<br>CB<br>d<br>E6 |
|  | 5 | CB<br>EF | CB<br>E8 | CB<br>E9 | CB<br>EA | CB<br>EB | CB<br>EC | CB<br>ED | CB<br>EE | DD<br>CB<br>d<br>EE | FD<br>CB<br>d<br>EE |
|  | 6 | CB<br>F7 | CB<br>F0 | CB<br>F1 | CB<br>F2 | CB<br>F3 | CB<br>F4 | CB<br>F5 | CB<br>F6 | DD<br>CB<br>d<br>F6 | FD<br>CB<br>d<br>F6 |
|  | 7 | CB<br>FF | CB<br>F8 | CB<br>F9 | CB<br>FA | CB<br>FB | CB<br>FC | CB<br>FD | CB<br>FE | DD<br>CB<br>d<br>FE | FD<br>CB<br>d<br>FE |

BIT MANIPULATION GROUP
TABLE 5.3–10

|  |  |
|---|---|
| Disable Interrupt | – prevent interrupt before routine is exited. |
| LD A, n<br>OUT n, A | – notify peripheral that service routine is complete |
| Enable Interrupt |  |
| Return |  |

This seven byte sequence can be replaced with the two byte RETI instruction in the Z-80. This is important since interrupt service time often must be minimized.

To facilitate program loop control the instruction DJNZ e can be used advantageously. This two byte, relative jump instruction decrements the B register and the jump occurs if the B register has not been decremented to zero. The relative displacement is expressed as a signed two's complement number. A simple example of its use might be:

| Address | Instruction | Comments |
|---|---|---|
| N, N + 1 | LD B, 7 | ; set B register to count of 7 |
| N + 2 to N + 9 | (Perform a sequence of instructions) | ; loop to be performed 7 times |
| N + 10, N + 11 | DJNZ –8 | ; to jump from N + 12 to N + 2 |
| N + 12 | (Next Instruction) |  |

|  |  |  | CONDITION |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | UN-COND. | CARRY | NON CARRY | ZERO | NON ZERO | PARITY EVEN | PARITY ODD | SIGN NEG | SIGN POS | REG B≠0 |
| JUMP 'JP' | IMMED. EXT. | nn | C3<br>n<br>n | DA<br>n<br>n | D2<br>n<br>n | CA<br>n<br>n | C2<br>n<br>n | EA<br>n<br>n | E2<br>n<br>n | FA<br>n<br>n | F2<br>n<br>n |  |
| JUMP 'JR' | RELATIVE | PC+e | 18<br>e-2 | 38<br>e-2 | 30<br>e-2 | 28<br>e-2 | 20<br>e-2 |  |  |  |  |  |
| JUMP 'JP' | REG. INDIR. | (HL) | E9 |  |  |  |  |  |  |  |  |  |
| JUMP 'JP' |  | (IX) | DD<br>E9 |  |  |  |  |  |  |  |  |  |
| JUMP 'JP' |  | (IY) | FD<br>E9 |  |  |  |  |  |  |  |  |  |
| 'CALL' | IMMED. EXT. | nn | CD<br>n<br>n | DC<br>n<br>n | D4<br>n<br>n | CC<br>n<br>n | C4<br>n<br>n | EC<br>n<br>n | E4<br>n<br>n | FC<br>n<br>n | F4<br>n<br>n |  |
| DECREMENT B, JUMP IF NON ZERO 'DJNZ' | RELATIVE | PC+e |  |  |  |  |  |  |  |  |  | 10<br>e-2 |
| RETURN 'RET' | REGISTER INDIR. | (SP)<br>(SP+1) | C9 | D8 | D0 | C8 | C0 | E8 | E0 | F8 | F0 |  |
| RETURN FROM INT 'RETI' | REG. INDIR. | (SP)<br>(SP+1) | ED<br>4D |  |  |  |  |  |  |  |  |  |
| RETURN FROM NON MASKABLE INT 'RETN' | REG. INDIR. | (SP)<br>(SP+1) | ED<br>45 |  |  |  |  |  |  |  |  |  |

NOTE–CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

JUMP, CALL and RETURN GROUP
TABLE 5.3–11

Table 5.3-12 lists the eight OP codes for the restart instruction. This instruction is a single byte call to any of the eight addresses listed. The simple mnemonic for these eight calls is also shown. The value of this instruction is that frequently used routines can be called with this instruction to minimize memory usage.

| CALL ADDRESS | OP CODE | |
|---|---|---|
| $0000_H$ | C7 | 'RST 0' |
| $0008_H$ | CF | 'RST 8' |
| $0010_H$ | D7 | 'RST 16' |
| $0018_H$ | DF | 'RST 24' |
| $0020_H$ | E7 | 'RST 32' |
| $0028_H$ | EF | 'RST 40' |
| $0030_H$ | F7 | 'RST 48' |
| $0038_H$ | FF | 'RST 56' |

RESTART GROUP
TABLE 5.3–12

## INPUT/OUTPUT

The Z-80 has an extensive set of Input and Output instructions as shown in table 5.3-13 and table 5.3-14. The addressing of the input or output device can be either absolute or register indirect, using the C register. Notice that in the register indirect addressing mode data can be transferred between the I/O devices and any of the internal registers. In addition eight block transfer instructions have been implemented. These instructions are similar to the memory block transfers except that they use register pair HL for a pointer to the memory source (output commands) or destination (input commands) while register B is used as a byte counter. Register C holds the address of the port for which the input or output command is desired. Since register B is eight bits in length, the I/O block transfer command handles up to 256 bytes.

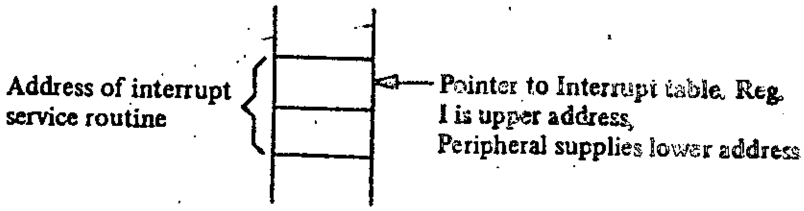
In the instructions IN A, n and OUT n, A the I/O device address n appears in the lower half of the address bus ($A_0$-$A_7$) while the accumulator content is transferred in the upper half of the address bus. In all register indirect input output instructions, including block I/O transfers the content of register C is transferred to the lower half of the address bus (device address) while the content of register B is transferred to the upper half of the address bus.

|  |  |  |  | SOURCE<br>PORT ADDRESS |  |
|---|---|---|---|---|---|
|  |  |  |  | IMMED. | REG.<br>INDIR. |
|  |  |  |  | (n) | (C) |
| INPUT DESTINATION | INPUT 'IN' | REG ADDRESSING | A | DB<br>n | ED<br>78 |
|  |  |  | B |  | ED<br>40 |
|  |  |  | C |  | ED<br>48 |
|  |  |  | D |  | ED<br>50 |
|  |  |  | E |  | ED<br>58 |
|  |  |  | H |  | ED<br>60 |
|  |  |  | L |  | ED<br>68 |
| BLOCK INPUT COMMANDS | 'INI' – INPUT & Inc HL, Dec B | REG. INDIR | (HL) |  | ED<br>A2 |
|  | 'INIR'– INP, Inc HL, Dec B, REPEAT IF B≠0 |  |  |  | ED<br>B2 |
|  | 'IND'– INPUT & Dec HL, Dec B |  |  |  | ED<br>AA |
|  | 'INDR'– INPUT, Dec HL, Dec B, REPEAT IF B≠0 |  |  |  | ED<br>BA |

INPUT GROUP
TABLE 5.3–13

## CPU CONTROL GROUP

The final table, table 5.3-15 illustrates the six general purpose CPU control instructions. The NOP is a do-nothing instruction. The HALT instruction suspends CPU operation until a subsequent interrupt is received, while the DI and EI are used to lock out and enable interrupts. The three interrupt mode commands set the CPU into any of the three available interrupt response modes as follows. If mode zero is set the interrupting device can insert any instruction on the data bus and allow the CPU to execute it. Mode 1 is a simplified mode where the CPU automatically executes a restart (RST) to location 0038H so that no external hardware is required. (The old PC content is pushed onto the stack). Mode 2 is the most powerful in that it allows for an indirect call to any location in memory. With this mode the CPU forms a 16-bit memory address where the upper 8-bits are the content of register I and the lower 8-bits are supplied by the interrupting device. This address points to the first of two sequential bytes in a table where the address of the service routine is located. The CPU automatically obtains the starting address and performs a CALL to this address.


Address of interrupt
service routine
Pointer to Interrupt table. Reg.
I is upper address,
Peripheral supplies lower address

| | | | SOURCE | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | REGISTER | | | | | | | REG. IND. |
| | | | A | B | C | D | E | H | L | (HL) |
| 'OUT' | IMMED. | (n) | D3<br>n | | | | | | | |
| | REG. IND. | (C) | ED<br>79 | ED<br>41 | ED<br>49 | ED<br>51 | ED<br>59 | ED<br>61 | ED<br>69 | |
| 'OUTI' – OUTPUT Inc HL, Dec b | REG. IND. | (C) | | | | | | | | ED<br>A3 |
| 'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0 | REG. IND. | (C) | | | | | | | | ED<br>B3 |
| 'OUTD' – OUTPUT Dec HL & B | REG. IND. | (C) | | | | | | | | ED<br>AB |
| 'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0 | REG. IND. | (C) | | | | | | | | ED<br>BB |

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

OUTPUT GROUP
TABLE 5.3–14

| | | |
|---|---|---|
| 'NOP' | 00 | |
| 'HALT' | 76 | |
| DISABLE INT '(DI)' | F3 | |
| ENABLE INT '(EI)' | FB | |
| SET INT MODE 0<br>'IM0' | ED<br>46 | 8080A MODE |
| SET INT MODE 1<br>'IM1' | ED<br>56 | CALL TO LOCATION $0038_H$ |
| SET INT MODE 2<br>'IM2' | ED<br>5E | INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER. |

MISCELLANEOUS CPU CONTROL
TABLE 5.3–15

## 6.0 FLAGS

Each of the two Z-80 CPU Flag registers contains six bits of information which are set or reset by various CPU operations. Four of these bits are testable; that is, they are used as conditions for jump, call or return instructions. For example a jump may be desired only if a specific bit in the flag register is set. The four testable flag bits are:

1) Carry Flag (C) – This flag is the carry from the highest order bit of the accumulator. For example, the carry flag will be set during an add instruction where a carry from the highest bit of the accumulator is generated. This flag is also set if a borrow is generated during a subtraction instruction. The shift and rotate instructions also affect this bit.

2) Zero Flag (Z) – This flag is set if the result of the operation loaded a zero into the accumulator. Otherwise it is reset.

3) Sign Flag (S) – This flag is intended to be used with signed numbers and it is set if the result of the operation was negative. Since bit 7 (MSB) represents the sign of the number (A negative number has a 1 in bit 7), this flag stores the state of bit 7 in the accumulator.

4) Parity/Overflow Flag (P/V) – This dual purpose flag indicates the parity of the result in the accumulator when logical operations are performed (such as AND A, B) and it represents overflow when signed two's complement arithmetic operations are performed. The Z-80 overflow flag indicates that the two's complement number in the accumulator is in error since it has exceeded the maximum possible (+127) or is less than the minimum possible (-128) ~~number~~ than can be represented in two's complement notation. For example consider adding:

```
    +120 =     0111 1000
    +105 =     0110 1001
       C = 0   1110 0001  =  -95 (wrong) Overflow has occured
```

Here the result is incorrect. Overflow has occurred and yet there is no carry to indicate an error. For this case the overflow flag would be set. Also consider the addition of two negative numbers:

```
      -5 =     1111 1011
     -16 =     1111 0000
       C = 1   1110 1011  =  -21 correct
```

Notice that the answer is correct but the carry is set so that this flag can not be used as an overflow indicator. In this case the overflow would not be set.

For logical operations (AND, OR, XOR) this flag is set if the parity of the result is even and it is reset if it is odd.

There are also two non-testable bits in the flag register. Both of these are used for BCD arithmetic. They are:

1) Half carry (H) – This is the BCD carry or borrow result from the least significant four bits of operation. When using the DAA (Decimal Adjust Instruction) this flag is used to correct the result of a previous packed decimal add or subtract.

2) Subtract Flag (N) – Since the algorithm for correcting BCD operations is different for addition or subtraction, this flag is used to specify what type of instruction was executed last so that the DAA operation will be correct for either addition or subtraction.

The Flag register can be accessed by the programmer and its format is as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | Z | X | H | X | P/V | N | C |

X means flag is indeterminate.

Table 6.0-1 lists how each flag bit is affected by various CPU instructions. In this table a '•' indicates that the instruction does not change the flag, an 'X' means that the flag goes to an indeterminate state, a '0' means that it is reset, a '1' means that it is set and the symbol '↕' indicates that it is set or reset according to the previous discussion. Note that any instruction not appearing in this table does not affect any of the flags.

Table 6.0-1 includes a few special cases that must be described for clarity. Notice that the block search instruction sets the Z flag if the last compare operation indicated a match between the source and the accumulator data. Also, the parity flag is set if the byte counter (register pair BC) is not equal to zero. This same use of the parity flag is made with the block move instructions. Another special case is during block input or output instructions, here the Z flag is used to indicate the state of register B which is used as a byte counter. Notice that when the I/O block transfer is complete, the zero flag will be reset to a zero (i.e. B=0) while in the case of a block move command the parity flag is reset when the operation is complete. A final case is when the refresh or I register is loaded into the accumulator, the interrupt enable flip flop is loaded into the parity flag so that the complete state of the CPU can be saved at any time.

| Instruction | C | Z | P/V | S | N | H | Comments |
|---|---|---|---|---|---|---|---|
| ADD A, s; ADC A,s | ↕ | ↕ | V | ↕ | 0 | ↕ | 8-bit add or add with carry |
| SUB s; SBC A, s, CP s, NEG | ↕ | ↕ | V | ↕ | 1 | ↕ | 8-bit subtract, subtract with carry, compare and negate accumulator |
| AND s | 0 | ↕ | P | ↕ | 0 | 1 | Logical operations |
| OR s; XOR s | 0 | ↕ | P | ↕ | 0 | 0 | And set's different flags |
| INC s | ● | ↕ | V | ↕ | 0 | ↕ | 8-bit increment |
| DEC m | ● | ↕ | V | ↕ | 1 | ↕ | 8-bit decrement |
| ADD DD, ss | ↕ | ● | ● | ● | 0 | X | 16-bit add |
| ADC HL, ss | ↕ | ↕ | V | ↕ | 0 | X | 16-bit add with carry |
| SBC HL, ss | ↕ | ↕ | V | ↕ | 1 | X | 16-bit subtract with carry |
| RLA; RLCA, RRA, RRCA | ↕ | ● | ● | ● | 0 | 0 | Rotate accumulator |
| RL m; RLC m; RR m; RRC m<br>SLA m; SRA m; SRL m | ↕ | ↕ | P | ↕ | 0 | 0 | Rotate and shift location s |
| RLD, RRD | ● | ↕ | P | ↕ | 0 | 0 | Rotate digit left and right |
| DAA | ↕ | ↕ | P | ↕ | ● | ↕ | Decimal adjust accumuiator |
| CPL | ● | ● | ● | ● | 1 | 1 | Complement accumulator |
| SCF | 1 | ● | ● | ● | 0 | 0 | Set carry |
| CCF | ↕ | ● | ● | ● | 0 | X | Complement carry |
| IN r, (C) | ● | ↕ | P | ↕ | 0 | 0 | Input register indirect |
| INI; IND; OUTI; OUTD | ● | ↕ | X | X | 1 | X | Block input and output |
| INIR; INDR; OTIR; OTDR | ● | 1 | X | X | 1 | X | Z = 0 if B ≠ 0 otherwise Z = 1 |
| LDI, LDD | ● | X | ↕ | X | 0 | 0 | Block transfer instructions |
| LDIR, LDDR | ● | X | 0 | X | 0 | 0 | P/V = 1 if BC ≠ 0, otherwise P/V = 0 |
| CPI, CPIR, CPD, CPDR | ● | ↕ | ↕ | X | 1 | X | Block search instructions<br>Z = 1 if A = (HL), otherwise Z = 0<br>P/V = 1 if BC ≠ 0, otherwise P/V = 0 |
| LD A, I; LD A, R | ● | ↕ | IFF | ↕ | 0 | 0 | The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag |
| BIT b, s | ● | ↕ | X | X | 0 | 1 | The state of bit b of location s is copied into the Z flag |
| NEG | ↕ | ↕ | V | ↕ | 1 | ↕ | Negate accumulator |

The following notation is used in this table:

| Symbol | Operation |
|---|---|
| C | Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result. |
| Z | Zero flag. Z=1 if the result of the operation is zero. |
| S | Sign flag. S=1 if the MSB of the result is one. |
| P/V | Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow. |
| H | Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator. |
| N | Add/Subtract flag. N=1 if the previous operation was a subtract. |
|  | H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format. |
| ↕ | The flag is affected according to the result of the operation. |
| ● | The flag is unchanged by the operation. |
| 0 | The flag is reset by the operation. |
| 1 | The flag is set by the operation. |
| X | The flag is a "don't care." |
| V | P/V flag affected according to the overflow result of the operation. |
| P | P/V flag affected according to the parity result of the operation. |
| r | Any one of the CPU registers A, B, C, D, E, H, L. |
| s | Any 8-bit location for all the addressing modes allowed for the particular instruction. |
| ss | Any 16-bit location for all the addressing modes allowed for that instruction. |
| ii | Any one of the two index registers IX or IY. |
| R | Refresh counter. |
| n | 8-bit value in range <0, 255> |
| nn | 16-bit value in range <0, 65535> |
| m | Any 8-bit location for all the addressing modes allowed for the particular instruction. |

SUMMARY OF FLAG OPERATION
TABLE 6.0-1

## 7.0 SUMMARY OF OP CODES AND EXECUTION TIMES

The following section gives a summary of the Z-80 instructions set. The instructions are logically arranged into groups as shown on tables 7.0-1 through 7.0-11. Each table shows the assembly language mnemonic OP code, the actual OP code, the symbolic operation, the content of the flag register following the execution of each instruction, the number of bytes required for each instruction as well as the number of memory cycles and the total number of T states (external clock periods) required for the fetching and execution of each instruction. Care has been taken to make each table self-explanatory without requiring any cross reference with the test or other tables.

| Mnemonic | Symbolic Operation | Flags<br>C | <br>Z | <br>P/V | <br>S | <br>N | <br>H | OP-Code<br>76 543 210 | No. of Bytes | No. of M Cycles | No. of T Cycles | Comments | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD r, r' | r←r' | ● | ● | ● | ● | ● | ● | 01 r r' | 1 | 1 | 4 | r, r' | Reg. |
| LD r, n | r←n | ● | ● | ● | ● | ● | ● | 00 r 110 | 2 | 2 | 7 | 000 | B |
| | | | | | | | | ← n → | | | | 001 | C |
| LD r, (HL) | r←(HL) | ● | ● | ● | ● | ● | ● | 01 r 110 | 1 | 2 | 7 | 010 | D |
| LD r, (IX+d) | r←(IX+d) | ● | ● | ● | ● | ● | ● | 11 011 101 | 3 | 5 | 19 | 011 | E |
| | | | | | | | | 01 r 110 | | | | 100 | H |
| | | | | | | | | ← d → | | | | 101 | L |
| LD r, (IY+d) | r←(IY+d) | ● | ● | ● | ● | ● | ● | 11 111 101 | 3 | 5 | 19 | 111 | A |
| | | | | | | | | 01 r 110 | | | | | |
| | | | | | | | | ← d → | | | | | |
| LD (HL), r | (HL)←r | ● | ● | ● | ● | ● | ● | 01 110 r | 1 | 2 | 7 | | |
| LD (IX+d), r | (IX+d)←r | ● | ● | ● | ● | ● | ● | 11 011 101 | 3 | 5 | 19 | | |
| | | | | | | | | 01 110 r | | | | | |
| | | | | | | | | ← d → | | | | | |
| LD (IY+d), r | (IY+d)←r | ● | ● | ● | ● | ● | ● | 11 111 101 | 3 | 5 | 19 | | |
| | | | | | | | | 01 110 r | | | | | |
| | | | | | | | | ← d → | | | | | |
| LD (HL), n | (HL)←n | ● | ● | ● | ● | ● | ● | 00 110 110 | 2 | 3 | 10 | | |
| | | | | | | | | ← n → | | | | | |
| LD (IX+d), n | (IX+d)←n | ● | ● | ● | ● | ● | ● | 11 011 101 | 4 | 5 | 19 | | |
| | | | | | | | | 00 110 110 | | | | | |
| | | | | | | | | ← d → | | | | | |
| | | | | | | | | ← n → | | | | | |
| LD (IY+d), n | (IY+d)←n | ● | ● | ● | ● | ● | ● | 11 111 101 | 4 | 5 | 19 | | |
| | | | | | | | | 00 110 110 | | | | | |
| | | | | | | | | ← d → | | | | | |
| | | | | | | | | ← n → | | | | | |
| LD A, (BC) | A←(BC) | ● | ● | ● | ● | ● | ● | 00 001 010 | 1 | 2 | 7 | | |
| LD A, (DE) | A←(DE) | ● | ● | ● | ● | ● | ● | 00 011 010 | 1 | 2 | 7 | | |
| LD A, (nn) | A←(nn) | ● | ● | ● | ● | ● | ● | 00 111 010 | 3 | 4 | 13 | | |
| | | | | | | | | ← n → | | | | | |
| | | | | | | | | ← n → | | | | | |
| LD (BC), A | (BC)←A | ● | ● | ● | ● | ● | ● | 00 000 010 | 1 | 2 | 7 | | |
| LD (DE), A | (DE)←A | ● | ● | ● | ● | ● | ● | 00 010 010 | 1 | 2 | 7 | | |
| LD (nn), A | (nn)←A | ● | ● | ● | ● | ● | ● | 00 110 010 | 3 | 4 | 13 | | |
| | | | | | | | | ← n → | | | | | |
| | | | | | | | | ← n → | | | | | |
| LD A, I | A←I | ● | ↕ | IFF | ↕ | 0 | 0 | 11 101 101 | 2 | 2 | 9 | | |
| | | | | | | | | 01 010 111 | | | | | |
| LD A, R | A←R | ● | ↕ | IFF | ↕ | 0 | 0 | 11 101 101 | 2 | 2 | 9 | | |
| | | | | | | | | 01 011 111 | | | | | |
| LD I, A | I←A | ● | ● | ● | ● | ● | ● | 11 101 101 | 2 | 2 | 9 | | |
| | | | | | | | | 01 000 111 | | | | | |
| LD R, A | R←A | ● | ● | ● | ● | ● | ● | 11 101 101 | 2 | 2 | 9 | | |
| | | | | | | | | 01 001 111 | | | | | |

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

8-BIT LOAD GROUP
TABLE 7.0-1

| Mnemonic | Symbolic Operation | Flags C | Z | P/V | S | N | H | Op-Code 76 543 210 | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD dd, nn | $dd \leftarrow nn$ | • | • | • | • | • | • | 00 dd0 001<br>← n →<br>← n → | 3 | 3 | 10 | dd Pair<br>00 BC<br>01 DE<br>10 HL<br>11 SP |
| LD IX, nn | $IX \leftarrow nn$ | • | • | • | • | • | • | 11 011 101<br>00 100 001<br>← n →<br>← n → | 4 | 4 | 14 | |
| LD IY, nn | $IY \leftarrow nn$ | • | • | • | • | • | • | 11 111 101<br>00 100 001<br>← n →<br>← n → | 4 | 4 | 14 | |
| LD HL, (nn) | $H \leftarrow (nn+1)$<br>$L \leftarrow (nn)$ | • | • | • | • | • | • | 00 101 010<br>← n →<br>← n → | 3 | 5 | 16 | |
| LD dd, (nn) | $dd_H \leftarrow (nn+1)$<br>$dd_L \leftarrow (nn)$ | • | • | • | • | • | • | 11 101 101<br>01 dd1 011<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD IX, (nn) | $IX_H \leftarrow (nn+1)$<br>$IX_L \leftarrow (nn)$ | • | • | • | • | • | • | 11 011 101<br>00 101 010<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD IY, (nn) | $IY_H \leftarrow (nn+1)$<br>$IY_L \leftarrow (nn)$ | • | • | • | • | • | • | 11 111 101<br>00 101 010<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD (nn), HL | $(nn+1) \leftarrow H$<br>$(nn) \leftarrow L$ | • | • | • | • | • | • | 00 100 010<br>← n →<br>← n → | 3 | 5 | 16 | |
| LD (nn), dd | $(nn+1) \leftarrow dd_H$<br>$(nn) \leftarrow dd_L$ | • | • | • | • | • | • | 11 101 101<br>01 dd0 011<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD (nn), IX | $(nn+1) \leftarrow IX_H$<br>$(nn) \leftarrow IX_L$ | • | • | • | • | • | • | 11 011 101<br>00 100 010<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD (nn), IY | $(nn+1) \leftarrow IY_H$<br>$(nn) \leftarrow IY_L$ | • | • | • | • | • | • | 11 111 101<br>00 100 010<br>← n →<br>← n → | 4 | 6 | 20 | |
| LD SP, HL | $SP \leftarrow HL$ | • | • | • | • | • | • | 11 111 001 | 1 | 1 | 6 | |
| LD SP, IX | $SP \leftarrow IX$ | • | • | • | • | • | • | 11 011 101<br>11 111 001 | 2 | 2 | 10 | |
| LD SP, IY | $SP \leftarrow IY$ | • | • | • | • | • | • | 11 111 101<br>11 111 001 | 2 | 2 | 10 | |
| PUSH qq | $(SP-2) \leftarrow qq_L$<br>$(SP-1) \leftarrow qq_H$ | • | • | • | • | • | • | 11 qq0 101 | 1 | 3 | 11 | qq Pair<br>00 BC<br>01 DE<br>10 HL<br>11 AF |
| PUSH IX | $(SP-2) \leftarrow IX_L$<br>$(SP-1) \leftarrow IX_H$ | • | • | • | • | • | • | 11 011 101<br>11 100 101 | 2 | 4 | 15 | |
| PUSH IY | $(SP-2) \leftarrow IY_L$<br>$(SP-1) \leftarrow IY_H$ | • | • | • | • | • | • | 11 111 101<br>11 100 101 | 2 | 4 | 15 | |
| POP qq | $qq_H \leftarrow (SP+1)$<br>$qq_L \leftarrow (SP)$ | • | • | • | • | • | • | 11 qq0 001 | 1 | 3 | 10 | |
| POP IX | $IX_H \leftarrow (SP+1)$<br>$IX_L \leftarrow (SP)$ | • | • | • | • | • | • | 11 011 101<br>11 100 001 | 2 | 4 | 14 | |
| POP IY | $IY_H \leftarrow (SP+1)$<br>$IY_L \leftarrow (SP)$ | • | • | • | • | • | • | 11 111 101<br>11 100 001 | 2 | 4 | 14 | |

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
E.g. $BC_L = C$, $AF_H = A$

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ flag is affected according to the result of the operation.

## 16-BIT LOAD GROUP
## TABLE 7.0-2

| Mnemonic | Symbolic Operation | C | Z | P/V | S | N | H | 76 543 210 | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EX DE, HL | DE ↔ HL | • | • | • | • | • | • | 11 101 011 | 1 | 1 | 4 | |
| EX AF, AF' | AF ↔ AF' | • | • | • | • | • | • | 00 001 000 | 1 | 1 | 4 | |
| EXX | $\begin{pmatrix}BC\\DE\\HL\end{pmatrix} \leftrightarrow \begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$ | • | • | • | • | • | • | 11 011 001 | 1 | 1 | 4 | Register bank and auxiliary register bank exchange |
| EX (SP), HL | H ↔ (SP+1)<br>L ↔ (SP) | • | • | • | • | • | • | 11 100 011 | 1 | 5 | 19 | |
| EX (SP), IX | $IX_H$ ↔ (SP+1)<br>$IX_L$ ↔ (SP) | • | • | • | • | • | • | 11 011 101<br>11 100 011 | 2 | 6 | 23 | |
| EX (SP), IY | $IY_H$ ↔ (SP+1)<br>$IY_L$ ↔ (SP) | • | • | • | • | • | • | 11 111 101<br>11 100 011 | 2 | 6 | 23 | |
| LDI | (DE) ← (HL)<br>DE ← DE+1<br>HL ← HL+1<br>BC ← BC-1 | • | • | ①<br>↕ | • | 0 | 0 | 11 101 101<br>10 100 000 | 2 | 4 | 16 | Load (HL) into (DE), increment the pointers and decrement the byte counter (BC) |
| LDIR | (DE) ← (HL)<br>DE ← DE+1<br>HL ← HL+1<br>BC ← BC-1<br>Repeat until<br>BC = 0 | • | • | 0 | • | 0 | 0 | 11 101 101<br>10 110 000 | 2<br>2 | 5<br>4 | 21<br>16 | If BC ≠ 0<br>If BC = 0 |
| LDD | (DE) ← (HL)<br>DE ← DE-1<br>HL ← HL-1<br>BC ← BC-1 | • | • | ①<br>↕ | • | 0 | 0 | 11 101 101<br>10 101 000 | 2 | 4 | 16 | |
| LDDR | (DE) ← (HL)<br>DE ← DE-1<br>HL ← HL-1<br>BC ← BC-1<br>Repeat until<br>BC = 0 | • | • | 0 | • | 0 | 0 | 11 101 101<br>10 111 000 | 2<br>2 | 5<br>4 | 21<br>16 | If BC ≠ 0<br>If BC = 0 |
| CPI | A - (HL)<br>HL ← HL+1<br>BC ← BC- 1 | • | ②<br>↕ | ①<br>↕ | ↕ | 1 | ↕ | 11 101 101<br>10 100 001 | 2 | 4 | 16 | |
| CPIR | A - (HL)<br>HL ← HL+1<br>BC ← BC-1<br>Repeat until<br>A = (HL) or<br>BC = 0 | • | ②<br>↕ | ①<br>↕ | ↕ | 1 | ↕ | 11 101 101<br>10 110 001 | 2<br>2 | 5<br>4 | 21<br>16 | If BC ≠ 0 and A ≠ (HL)<br>If BC = 0 or A = (HL) |
| CPD | A - (HL)<br>HL ← HL-1<br>BC ← BC-1 | • | ②<br>↕ | ①<br>↕ | ↕ | 1 | ↕ | 11 101 101<br>10 101 001 | 2 | 4 | 16 | |
| CPDR | A - (HL)<br>HL ← HL-1<br>BC ← BC-1<br>Repeat until<br>A = (HL) or<br>BC = 0 | • | ②<br>↕ | ①<br>↕ | ↕ | 1 | ↕ | 11 101 101<br>10 111 001 | 2<br>2 | 5<br>4 | 21<br>16 | If BC ≠ 0 and A ≠ (HL)<br>If BC = 0 or A = (HL) |

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

## EXCHANGE GROUP AND BLOCK TRANSFER AND SEARCH GROUP
## TABLE 7.0-3

| Mnemonic | Symbolic Operation | Flags |  |  |  |  |  | Op-Code | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | C | Z | P/V | S | N | H | 76 543 210 |  |  |  |  |
| ADD A, r | A ← A + r | ↕ | ↕ | V | ↕ | 0 | ↕ | 10 $\boxed{000}$ r | 1 | 1 | 4 | r Reg.<br>000 B<br>001 C<br>010 D<br>011 E<br>100 H<br>101 L<br>111 A |
| ADD A, n | A ← A + n | ↕ | ↕ | V | ↕ | 0 | ↕ | 11 $\boxed{000}$ 110<br>← n → | 2 | 2 | 7 |  |
| ADD A, (HL) | A ← A + (HL) | ↕ | ↕ | V | ↕ | 0 | ↕ | 10 $\boxed{000}$ 110 | 1 | 2 | 7 |  |
| ADD A, (IX+d) | A←A + (IX+d) | ↕ | ↕ | V | ↕ | 0 | ↕ | 11 011 101<br>10 $\boxed{000}$ 110<br>← d → | 3 | 5 | 19 |  |
| ADD A, (IY+d) | A←A+(IY+d) | ↕ | ↕ | V | ↕ | 0 | ↕ | 11 111 101<br>10 $\boxed{000}$ 110<br>← d → | 3 | 5 | 19 |  |
| ADC A, s | A ← A + s + CY | ↕ | ↕ | V | ↕ | 0 | ↕ | $\boxed{001}$ |  |  |  |  |
| SUB s | A ← A − s | ↕ | ↕ | V | ↕ | 1 | ↕ | $\boxed{010}$ |  |  |  | s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction |
| SBC A, s | A ← A − s − CY | ↕ | ↕ | V | ↕ | 1 | ↕ | $\boxed{011}$ |  |  |  |  |
| AND s | A ← A ∧ s | 0 | ↕ | P | ↕ | 0 | 1 | $\boxed{100}$ |  |  |  |  |
| OR s | A ← A ∨ s | 0 | ↕ | P | ↕ | 0 | 0 | $\boxed{110}$ |  |  |  |  |
| XOR s | A ← A ⊕ s | 0 | ↕ | P | ↕ | 0 | 0 | $\boxed{101}$ |  |  |  | The indicated bits replace the 000 in the ADD set above. |
| CP s | A − s | ↕ | ↕ | V | ↕ | 1 | ↕ | $\boxed{111}$ |  |  |  |  |
| INC r | r ← r + 1 | ● | ↕ | V | ↕ | 0 | ↕ | 00 r $\boxed{100}$ | 1 | 1 | 4 |  |
| INC (HL) | (HL) ← (HL)+1 | ● | ↕ | V | ↕ | 0 | ↕ | 00 110 $\boxed{100}$ | 1 | 3 | 11 |  |
| INC (IX+d) | (IX+d) ← (IX+d)+1 | ● | ↕ | V | ↕ | 0 | ↕ | 11 011 101<br>00 110 $\boxed{100}$<br>← d → | 3 | 6 | 23 |  |
| INC (IY+d) | (IY+d) ← (IY+d) + 1 | ● | ↕ | V | ↕ | 0 | ↕ | 11 111 101<br>00 110 $\boxed{100}$<br>← d → | 3 | 6 | 23 |  |
| DEC m | m←m−1 | ● | ↕ | V | ↕ | 1 | ↕ | $\boxed{101}$ |  |  |  | m is any of r, (HL), (IX+d), (IY+d) as shown for INC. Same format and states as INC. Replace 100 with 101 in OP code. |

**Notes:** The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

**Flag Notation:** ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

8-BIT ARITHMETIC AND LOGICAL GROUP
TABLE 7.0-4

|  |  | Flags |  |  |  |  |  | Op-Code |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnemonic | Symbolic Operation | C | Z | P/V | S | N | H | 76 543 210 | No. of Bytes | No. of M Cycles | No. of T States | Comments |
| DAA | Converts acc. content into packed BCD following add or subtract with packed BCD operands | ‡ | ‡ | P | ‡ | • | ‡ | 00 100 111 | 1 | 1 | 4 | Decimal adjust accumulator |
| CPL | A ← $\overline{A}$ | • | • | • | • | 1 | 1 | 00 101 111 | 1 | 1 | 4 | Complement accumulator (one's complement) |
| NEG | A ← 0 − A | ‡ | ‡ | V | ‡ | 1 | ‡ | 11 101 101<br>01 000 100 | 2 | 2 | 8 | Negate acc. (two's complement) |
| CCF | CY ← $\overline{CY}$ | ‡ | • | • | • | 0 | X | 00 111 111 | 1 | 1 | 4 | Complement carry flag |
| SCF | CY ← 1 | 1 | • | • | • | 0 | 0 | 00 110 111 | 1 | 1 | 4 | Set carry flag |
| NOP | No operation | • | • | • | • | • | • | 00 000 000 | 1 | 1 | 4 |  |
| HALT | CPU halted | • | • | • | • | • | • | 01 110 110 | 1 | 1 | 4 |  |
| DI | IFF ← 0 | • | • | • | • | • | • | 11 110 011 | 1 | 1 | 4 |  |
| EI | IFF ← 1 | • | • | • | • | • | • | 11 111 011 | 1 | 1 | 4 |  |
| IM 0 | Set interrupt mode 0 | • | • | • | • | • | • | 11 101 101<br>01 000 110 | 2 | 2 | 8 |  |
| IM 1 | Set interrupt mode 1 | • | • | • | • | • | • | 11 101 101<br>01 010 110 | 2 | 2 | 8 |  |
| IM2 | Set interrupt mode 2 | • | • | • | • | • | • | 11 101 101<br>01 011 110 | 2 | 2 | 8 |  |

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

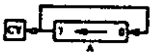
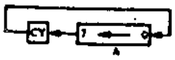
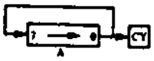
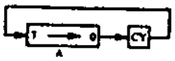
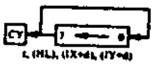
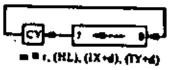
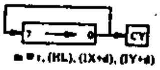
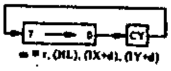
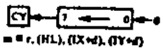
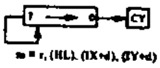
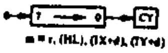
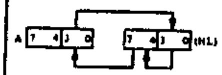
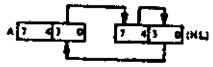
GENERAL PURPOSE ARITHMETIC AND CPU CONTROL GROUPS
TABLE 7.0-5

| Mnemonic | Symbolic Operation | Flags | | | | | | Op-Code | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C | Z | P/V | S | N | H | 76 543 210 | | | | |
| ADD HL, ss | HL ← HL+ss | ↕ | ● | ● | ● | 0 | X | 00 ss1 001 | 1 | 3 | 11 | ss Reg.<br>00 BC<br>01 DE<br>10 HL<br>11 SP |
| ADC HL, ss | HL←HL+ss+CY | ↕ | ↕ | V | ↕ | 0 | X | 11 101 101<br>01 ss1 010 | 2 | 4 | 15 | |
| SBC HL, ss | HL←HL-ss-CY | ↕ | ↕ | V | ↕ | 1 | X | 11 101 101<br>01 ss0 010 | 2 | 4 | 15 | |
| ADD IX, pp | IX ← IX + pp | ↕ | ● | ● | ● | 0 | X | 11 011 101<br>00 pp1 001 | 2 | 4 | 15 | pp Reg.<br>00 BC<br>01 DE<br>10 IX<br>11 SP |
| ADD IY, rr | IY←IY+rr | ↕ | ● | ● | ● | 0 | X | 11 111 101<br>00 rr1 001 | 2 | 4 | 15 | rr Reg.<br>00 BC<br>01 DE<br>10 IY<br>11 SP |
| INC ss | ss ← ss + 1 | ● | ● | ● | ● | ● | ● | 00 ss0 011 | 1 | 1 | 6 | |
| INC IX | IX ← IX + 1 | ● | ● | ● | ● | ● | ● | 11 011 101<br>00 100 011 | 2 | 2 | 10 | |
| INC IY | IY ← IY + 1 | ● | ● | ● | ● | ● | ● | 11 111 101<br>00 100 011 | 2 | 2 | 10 | |
| DEC ss | ss ← ss - 1 | ● | ● | ● | ● | ● | ● | 00 ss1 011 | 1 | 1 | 6 | |
| DEC IX | IX ← IX - 1 | ● | ● | ● | ● | ● | ● | 11 011 101<br>00 101 011 | 2 | 2 | 10 | |
| DEC IY | IY ← IY - 1 | ● | ● | ● | ● | ● | ● | 11 111 101<br>00 101 011 | 2 | 2 | 10 | |

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

## 16-BIT ARITHMETIC GROUP
## TABLE 7.0-6

| Mnemonic | Symbolic Operation | Flags | | | | | | Op-Code | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C | Z | P/V | S | N | H | 76 543 210 | | | | |
| RLCA |  | ↕ | • | • | • | 0 | 0 | 00 000 111 | 1 | 1 | 4 | Rotate left circular accumulator |
| RLA |  | ↕ | • | • | • | 0 | 0 | 00 010 111 | 1 | 1 | 4 | Rotate left accumulator |
| RRCA |  | ↕ | • | • | • | 0 | 0 | 00 001 111 | 1 | 1 | 4 | Rotate right circular accumulator |
| RRA |  | ↕ | • | • | • | 0 | 0 | 00 011 111 | 1 | 1 | 4 | Rotate right accumulator |
| RLC r | | ↕ | ↕ | P | ↕ | 0 | 0 | 11 001 011<br>00 $\boxed{000}$ r | 2 | 2 | 8 | Rotate left circular register r |
| RLC (HL) | | ↕ | ↕ | P | ↕ | 0 | 0 | 11 001 011<br>00 $\boxed{000}$ 110 | 2 | 4 | 15 | r Reg.<br>000 B<br>001 C<br>010 D<br>011 E<br>100 H<br>101 L<br>111 A |
| RLC (IX+d) |  | ↕ | ↕ | P | ↕ | 0 | 0 | 11 011 101<br>11 001 011<br>← d →<br>00 $\boxed{000}$ 110 | 4 | 6 | 23 | |
| RLC (IY+d) | | ↕ | ↕ | P | ↕ | 0 | 0 | 11 111 101<br>11 001 011<br>← d →<br>00 $\boxed{000}$ 110 | 4 | 6 | 23 | |
| RL m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{010}$ | | | | Instruction format and states are as shown for RLC,m. To form new OP-code replace $\boxed{000}$ of RLC,m with shown code |
| RRC m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{001}$ | | | | |
| RR m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{011}$ | | | | |
| SLA m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{100}$ | | | | |
| SRA m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{101}$ | | | | |
| SRL m |  | ↕ | ↕ | P | ↕ | 0 | 0 | $\boxed{111}$ | | | | |
| RLD |  | • | ↕ | P | ↕ | 0 | 0 | 11 101 101<br>01 101 111 | 2 | 5 | 18 | Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected |
| RRD |  | • | ↕ | P | ↕ | 0 | 0 | 11 101 101<br>01 100 111 | 2 | 5 | 18 | |

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

## ROTATE AND SHIFT GROUP
## TABLE 7.0-7

| Mnemonic | Symbolic Operation | C | Z | P/V | S | N | H | Op-Code<br>76 543 210 | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT b, r | $Z \leftarrow \overline{r}_b$ | • | ↕ | X | X | 0 | 1 | 11 001 011<br>01 b r | 2 | 2 | 8 | r — Reg.<br>000 B<br>001 C<br>010 D<br>011 E<br>100 H<br>101 L<br>111 A |
| BIT b, (HL) | $Z \leftarrow \overline{(HL)}_b$ | • | ↕ | X | X | 0 | 1 | 11 001 011<br>01 b 110 | 2 | 3 | 12 | |
| BIT b, (IX+d) | $Z \leftarrow \overline{(IX+d)}_b$ | • | ↕ | X | X | 0 | 1 | 11 011 101<br>11 001 011<br>← d →<br>01 b 110 | 4 | 5 | 20 | |
| BIT b, (IY+d) | $Z \leftarrow \overline{(IY+d)}_b$ | • | ↕ | X | X | 0 | 1 | 11 111 101<br>11 001 011<br>← d →<br>01 b 110 | 4 | 5 | 20 | b — Bit Tested<br>000 0<br>001 1<br>010 2<br>011 3<br>100 4<br>101 5<br>110 6<br>111 7 |
| SET b, r | $r_b \leftarrow 1$ | • | • | • | • | • | • | 11 001 011<br>$\boxed{11}$ b r | 2 | 2 | 8 | |
| SET b, (HL) | $(HL)_b \leftarrow 1$ | • | • | • | • | • | • | 11 001 011<br>$\boxed{11}$ b 110 | 2 | 4 | 15 | |
| SET b, (IX+d) | $(IX+d)_b \leftarrow 1$ | • | • | • | • | • | • | 11 011 101<br>11 001 011<br>← d →<br>$\boxed{11}$ b 110 | 4 | 6 | 23 | |
| SET b, (IY+d) | $(IY+d)_b \leftarrow 1$ | • | • | • | • | • | • | 11 111 101<br>11 001 011<br>← d →<br>$\boxed{11}$ b 110 | 4 | 6 | 23 | |
| RES b, m | $s_b \leftarrow 0$<br>$m \equiv r, (HL), (IX+d), (IY+d)$ | | | | | | | $\boxed{10}$ | | | | To form new OP-code replace $\boxed{11}$ of SET b,m with $\boxed{10}$. Flags and time states for SET instruction |

Notes: The notation $s_b$ indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

## BIT SET, RESET AND TEST GROUP
## TABLE 7.0-8

|  |  | Flags |  |  |  |  |  | Op-Code |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnemonic | Symbolic Operation | C | Z | P/V | S | N | H | 76 543 210 | No. of Bytes | No. of M Cycles | No. of T States | Comments |
| JP nn | PC ← nn | ● | ● | ● | ● | ● | ● | 11 000 011<br>← n →<br>← n → | 3 | 3 | 10 |  |
| JP cc, nn | If condition cc is true PC ←nn, otherwise continue | ● | ● | ● | ● | ● | ● | 11 cc 010<br>← n →<br>← n → | 3 | 3 | 10 | cc Condition<br>000 NZ non zero<br>001 Z zero<br>010 NC non carry<br>011 C carry<br>100 PO parity odd<br>101 PE parity even<br>110 P sign positive<br>111 M sign negative |
| JR e | PC ← PC + e | ● | ● | ● | ● | ● | ● | 00 011 000<br>← e-2 → | 2 | 3 | 12 |  |
| JR C, e | If C = 0, continue | ● | ● | ● | ● | ● | ● | 00 111 000<br>← e-2 → | 2 | 2 | 7 | If condition not met |
|  | If C = 1, PC ← PC+e |  |  |  |  |  |  |  | 2 | 3 | 12 | If condition is met |
| JR NC, e | If C = 1, continue | ● | ● | ● | ● | ● | ● | 00 110 000<br>← e-2 → | 2 | 2 | 7 | If condition not met |
|  | If C = 0, PC ← PC + e |  |  |  |  |  |  |  | 2 | 3 | 12 | If condition is met |
| JR Z, e | If Z = 0 continue | ● | ● | ● | ● | ● | ● | 00 101 000<br>← e-2 → | 2 | 2 | 7 | If condition not met |
|  | If Z = 1, PC ← PC + e |  |  |  |  |  |  |  | 2 | 3 | 12 | If condition is met |
| JR NZ, e | If Z = 1, continue | ● | ● | ● | ● | ● | ● | 00 100 000<br>← e-2 → | 2 | 2 | 7 | If condition not met |
|  | If Z = 0, PC ← PC + e |  |  |  |  |  |  |  | 2 | 3 | 12 | If condition met |
| JP (HL) | PC ← HL | ● | ● | ● | ● | ● | ● | 11 101 001 | 1 | 1 | 4 |  |
| JP (IX) | PC ← IX | ● | ● | ● | ● | ● | ● | 11 011 101<br>11 101 001 | 2 | 2 | 8 |  |
| JP (IY) | PC ← IY. | ● | ● | ● | ● | ● | ● | 11 111 101<br>11 101 001 | 2 | 2 | 8 |  |
| DJNZ,e | B ← B-1<br>If B = 0, continue | ● | ● | ● | ● | ● | ● | 00 010 000<br>← e-2 → | 2 | 2 | 8 | If B = 0 |
|  | If B ≠ 0, PC ← PC + e |  |  |  |  |  |  |  | 2 | 3 | 13 | IF B ≠ 0 |

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

## JUMP GROUP
## TABLE 7.0-9

| Mnemonic | Symbolic Operation | Flags | | | | | | Op-Code | | | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C | Z | P/V | S | N | H | 76 | 543 | 210 | | | | |
| CALL nn | (SP-1)←$PC_H$ | ● | ● | ● | ● | ● | ● | 11 | 001 | 101 | 3 | 5 | 17 | |
| | (SP-2)←$PC_L$ | | | | | | | ← | n | → | | | | |
| | PC←nn | | | | | | | ← | n | → | | | | |
| CALL cc, nn | If condition<br>cc is false | ● | ● | ● | ● | ● | ● | 11 | cc | 100 | 3 | 3 | 10 | If cc is false |
| | continue, | | | | | | | ← | n | → | | | | |
| | otherwise<br>same as<br>CALL nn | | | | | | | ← | n | → | 3 | 5 | 17 | If cc is true |
| RET | $PC_L$←(SP)<br>$PC_H$←(SP+1) | ● | ● | ● | ● | ● | ● | 11 | 001 | 001 | 1 | 3 | 10 | |
| RET cc | If condition<br>cc is false<br>continue, | ● | ● | ● | ● | ● | ● | 11 | cc | 000 | 1 | 1 | 5 | If cc is false |
| | otherwise<br>same as<br>RET | | | | | | | | | | 1 | 3 | 11 | If cc is true |
| RETI | Return from interrupt | ● | ● | ● | ● | ● | ● | 11 | 101 | 101 | 2 | 4 | 14 | |
| | | | | | | | | 01 | 001 | 101 | | | | |
| RETN | Return from non maskable interrupt | ● | ● | ● | ● | ● | ● | 11 | 101 | 101 | 2 | 4 | 14 | |
| | | | | | | | | 01 | 000 | 101 | | | | |
| RST p | (SP-1)←$PC_H$ | ● | ● | ● | ● | ● | ● | 11 | t | 111 | 1 | 3 | 11 | |
| | (SP-2)←$PC_L$ | | | | | | | | | | | | | |
| | $PC_H$←0 | | | | | | | | | | | | | |
| | $PC_L$←P | | | | | | | | | | | | | |

| cc | Condition | |
|---|---|---|
| 000 | NZ | non zero |
| 001 | Z | zero |
| 010 | NC | non carry |
| 011 | C | carry |
| 100 | PO | parity odd |
| 101 | PE | parity even |
| 110 | P | sign positive |
| 111 | M | sign negative |

| t | P |
|---|---|
| 000 | 00H |
| 001 | 08H |
| 010 | 10H |
| 011 | 18H |
| 100 | 20H |
| 101 | 28H |
| 110 | 30H |
| 111 | 38H |

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
↕ = flag is affected according to the result of the operation.

CALL AND RETURN GROUP
TABLE 7.0-10

| Mnemonic | Symbolic Operation | Flags | | | | | | Op-Code | No. of Bytes | No. of M Cycles | No. of T States | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C | Z | P/V | S | N | H | 76 543 210 | | | | |
| IN A, (n) | A ← (n) | ● | ● | ● | ● | ● | ● | 11 011 011<br>← n → | 3 | 3 | 11 | n to $A_0$ ~ $A_7$<br>Acc to $A_8$ ~ $A_{15}$ |
| IN r, (C) | r ← (C)<br>if r = 110 only the flags will be affected | ● | ↕ | P | ↕ | 0 | ↕ | 11 101 101<br>01 r 000 | 2 | 3 | 12 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| INI | (HL) ← (C)<br>B ← B - 1<br>HL ← HL + 1 | ● | ①<br>↕ | X | X | 1 | X | 11 101 101<br>10 100 010 | 2 | 4 | 16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| INIR | (HL) ← (C)<br>B ← B - 1<br>HL ← HL + 1<br>Repeat until B = 0 | ● | 1 | X | X | 1 | X | 11 101 101<br>10 110 010 | 2<br>3 | 5<br>(If B ≠ 0)<br>4<br>(If B = 0) | 21<br>16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| IND | (HL) ← (C)<br>B ← B - 1<br>HL ← HL - 1 | ● | ①<br>↕ | X | X | 1 | X | 11 101 101<br>10 101 010 | 2 | 4 | 16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| INDR | (HL) ← (C)<br>B ← B - 1<br>HL ← HL -1<br>Repeat until B = 0 | ● | 1 | X | X | 1 | X | 11 101 101<br>10 111 010 | 2<br>2 | 5<br>(If B ≠ 0)<br>4<br>(If B = 0) | 21<br>16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| OUT (n), A | (n) ← A | ● | ● | ● | ● | ● | ● | 11 010 011<br>← n → | 2 | 3 | 11 | n to $A_0$ ~ $A_7$<br>Acc to $A_8$ ~ $A_{15}$ |
| OUT (C), r | (C) ← r | ● | ● | ● | ● | ● | ● | 11 101 101<br>01 r 001 | 2 | 3 | 12 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| OUTI | (C) ← (HL)<br>B ← B - 1<br>HL ← HL + 1 | ● | ①<br>↕ | X | X | 1 | X | 11 101 101<br>10 100 011 | 2 | 4 | 16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| OTIR | (C) ← (HL)<br>B ← B - 1<br>HL ← HL + 1<br>Repeat until B = 0 | ● | 1 | X | X | 1 | X | 11 101 101<br>10 110 011 | 2<br>2 | 5<br>(If B ≠ 0)<br>4<br>(If B = 0) | 21<br>16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| OUTD | (C) ← (HL)<br>B ← B - 1<br>HL ← HL - 1 | ● | ①<br>↕ | X | X | 1 | X | 11 101 101<br>10 101 011 | 2 | 4 | 16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |
| OTDR | (C) ← (HL)<br>B ← B - 1<br>HL ← HL -1<br>Repeat until B = 0 | ● | 1 | X | X | 1 | X | 11 101 101<br>10 111 011 | 2<br>2 | 5<br>(If B ≠ 0)<br>4<br>(If B = 0) | 21<br>16 | C to $A_0$ ~ $A_7$<br>B to $A_8$ ~ $A_{15}$ |

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwize it is reset.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

**INPUT AND OUTPUT GROUP**
**TABLE 7.0-11**

DIRECTORIO DE PROFESORES DEL CURSO MICROPROCESADORES: TEORIA Y APLICACIONES 1979.

1. RAMIRO ANAYA ORTIZ
   INST. DE INVESTG.ELEC.
   INTERIOR INTERNADO PALMIRA
   CUERNAVACA, MOR.
   TEL.4.13.60

   Nisperos 2
   Lomas de Cuernavaca
   Cuernavaca, Mor.
   Tel.4.29.41

2. TEODORO AVENDAÑO MARTINEZ
   UNIVERSIDAD MICHOACANA
   ESC. DE ING. ELEC.
   CIUDAD UNIVERSITARIA, EDIF. C P.A.
   MORELIA, MICH.
   TEL.2.77.76

   M. Faraday 72
   Col. Viveros
   Morelia, Mich.

3. FRANCISCO AVILA PUERTO
   TOLEDO SCALE CO. DE MEX. S.A. DE C.V.
   PINO 350
   MEXICO 4, D.F.
   TEL.545.57.00

   Sn. Antonio 50
   Col. Sn. Agustín
   Naucalpan, Edo. de Méx.

4. JESUS BARRIOS ROMANO
   REAL DEL MONTE 236
   COL. INDUSTRIAL
   MEXICO 14,D.F.

5. AGUSTIN BETANCOURT DEL RIO
   SECRETARIA DE PROGRAMACION Y PRESUPUESTO
   CALZ. GALVAN 255
   COLIMA, COL.
   TEL.2.60.43

   Filomeno Medina 135
   Colima, Col.

6. EDMUNDO CACERES ACERETO
   TRANSDATA S.A.
   ENRIQUE REBSAMEN 413-4
   MEXICO 12, D.F.
   TEL.543.50.54

   Av.Churubusco 154
   Col. Prado
   México 13, D.F.

7. NICOLAS CALVA TAPIA
   ENEP ARAGON
   AV. CENTRAL Y RANCHO SECO
   COL. ARAGON
   MEXICO,D.F.

   Nicolás Bravo 17-1
   Col. Martín Carrera
   México 14, D.F.
   Tel.577.84.39

8. VICTOR M. CASTILLO DOMINGUEZ
   C. F. E.
   PLAZA DE LA REP. 26-5°
   MEXICO, D.F.
   TEL.546.24.21

   Div. del Nte. Mza. a No.19
   Col. B.Domínguez
   México 22, D.F.
   Tel. 677.24.35

9. SILVERIO CASTILLO SALCEDO
UNIONDE PROFESORES
FACULTAD DE INGENIERIA
UNAM
TEL.550.52.15 EXT.4485

Sabinos 34
Col. Sta. Anita
México 8, D.F.
Tel. 530.92.56

10. ODON DE BUEN LOZANO
FACULTAD DE INGENIERIA
UNAM
TEL.548.99.58

Cerro del Cubilete.
México ,D.F.
Tel.549.07.41

11. JUAN ANTONIO DELVALLE DOMINGUEZ
S.C. T.
DIR. GRAL. DE TEEGRAFOS NACIONALES
XOLA Y AV. UNIVERSIDAD 8°
MEXICO 12, D.F.
TEL.521.14.23

Peten 336-16
México 12, D.F.
Tel.536.48.99

12. PABLO DE URQUIJO NIEMBRO
YUCA 209
COL. NVA. STA. MARIA
MEXICO,D.F.
TEL.556.13.16

13. JUAN VALENTE ESTRADA ROBLES
S.C.T.
TELEGRAFOS NACIONALES
DPTO. DE AUTOMATIZACION
TORREON, COAH.
TEL.2.38.45

Calle 6ta. No. 10
Ampliasión losAngeles
TORREON, COAH.

14. JUAN MANUEL GARCIA RODRIGUEZ
PEMEX
MARINA NAL.329
MEXICO 17, D.F.
TEL.531.63.53

LAGO FONDO 185-5
Col. Pensil
México 17, D.F.
Tel.545.45.62

15. J.G. RUBEN GOMEZ S.
PRODUCTORA NACIONALIE RADIO Y T.V.
ATLETAS 2
COL. COUNTRY CLUB
MEXICO 21, D.F.
TEL.544.95.80

Apdo. Postal 76060
México 21, D.F.
Tel.590.94.02

16. NOE LEOVIGILDO GONZALEZ RODRIGUEZ
S. A. R.H.
P. DE LA REFORMA 69-9°
MEXICO 1, D.F.
TEL.535.65.95

Río Carmen 15
Fracc. del Moral
PASEOS DE CHURUBUSCO
MEXICO 13, D.F.

17. ENRIQUE GONZALEZ R.
UNIVERSIDAD DE LA AMERICAS
DOMICILIO CONOCIDO
CHOLULA, PUE.
TEL.47.06.55 EXT.123

18. JUMBERTO GONZALEZ RUIZ
MELCHOR OCAMPO 312
CUERNAVACA, MOR.
TEL.317.34

19.ANTONIO HERRERA MEJIA
ESC. NAC. EST. PROFESIONALES CUAUTITLAN
UNAM
CAMPO 3 CUAUTITLAN IZCALLI
ESTADO DE MEXICO

Lago Urmiah 20-9
Col. Pensil
México 17, D.F.
Tel.527.77.24

20. JOSE G. JURADO GONZALEZ
PRODUCTORA NAL. DE RADIO Y T.V.
CALLE ATLETAS 2
COL. COUNTRY CLUB
MEXICO 21, D.F.
TEL.544.23.09

Lago Mask 113-2
México 17, D.F.
Tel.544.23.09

21. JOSE LANDEROS VALDEPEÑA
ESC. NAL. DE EST. PROFESIONALES
CUAUTITLAN IZCALLI,EDO. DE MEX.
TEL.91591-331.11

Calle Dos No.41
Col. Indpendencia
México 13, D.F.
Tel.539.77.84

22. LEOPOLDO LADRON DE GUEVARA
INST. MEX. DEL PETROLEO
AV. CIEN METROS 152
MEXICO 4, D.F.
TEL. 567.54.76

Calle 647 No. 5
U.C.T.M. ARAGON
MEXICO 14, D.F.
Tel.794.09.40

23. ENRIQUE LOPEZ PATIÑO
FACULTAD DE ING. UNAM
Tel.548.99.58

Navegantes 32
Cda. Satélite,Edo. de Méx.
Tel.562.50.96

24. Jaime Antonio Machuca González
PRODUCTOS ESPECIALIZADOS DE ACERO S.A.
PTE. 134 # 854
COL. IND. VALLEJO
MEXICO 16, D.F.
TEL. 567.70.22

Bosques de Chiahuahua 56
Col. Sta. Mónica
Atizapán, México
Tel. 397.11.97

25. MANUEL MARTINEZ HERNANDEZ
S.A.H.O P.
P. DE LA REFORMA 77-7°
MEXICO, D.F.
TEL.

Av. La Teja 40 A -302
Villa Coapa
México 22, D.F.
Tel.594.12.60

26. MIGUEL MERCADO HERNANDEZ
CALLE 1523 y 416
SECC. VI, VII
SN. J. DE ARAGON
MEXICO 14, D.F.

Martín Carrera 22-18
México 14, D.F.

27. ARMANDO MELENDEZ LOZANO
INDUSTRIAS XOROGRAFICAS S.A.
AV. INDUSTRIA 43 y 45
FRACC. IND. SN. PABLO XALPA
TLALNEPANTLA,EDO. DE MEX.
TEL.392.04.80 EXT.144

Hda.de la Purísima 69
Prados del Rosario
México16, D.F.
Tel.352.05.24

28. JOSE E. MENDOZA PASCUE
TOLEDO SCALE CO. DE MEXICO S.A.
DE C.V.
PINO 350
MEXICO 4, D.F.
TEL. 547.57.00

Cerro Sn. Miguel 119
Fracc. Rincón del Valle
Tlalnepantla, Edo. de Méx.
Tel.379.19.72

29. CARLOS MILLER FARFAN
INST. MEX. DEL PETROLEO
AV. DE LOS 100 METROS NO.152
MEXICO, D.F.
TE.567.54.76

Sta. Ana. 58
Fracc. Capistrano
Atizapán, Edo. de Méx.
Tel.397.32.45

30. JAIME D. MORENO JIMENEZ
ESC. NAL. DE EST. PROFESIONALES
CUAUTITLAN, IZCALLI,EDO. DE MEX.

Camino Real de Toluca
México
Tel. 271.16.79

31. IRENE I. NAVARRO GONZALEZ
INST. DE INGENIERIA
UNAM
MEXICO 20, D.F.

Div. del Nte. 124-5
México 12, D.F.
Tel.687.07.57

32. DOMINGO OLIVA GUTIERREZ
S.A.H.O.P.
P. DE LA REFORMA 77-10°
MEXICO 4, D.F.
TEL.591.18.69

Lago Tus 4-1
México 17, D.F.
Tel.399.56.40

33. GUSTAVO ADOLFO PASTRANA ANGELES.
PEMEX
AV.MARINA NAL. 329-6°EDIF. B-1.
MEXICO 17, D.F.
TEL. 545.74.60 EXT.3429

Petirrojos No. 2 Secc. 1ra.
Lomas Verdes, Edo. de Méx.
Tel.572.74.06

34. ALEJANDRO PEREZ MARTINEZ
PEMEX
Marina NAL. 329
MEXICO 17, D.F.
TEL.531.63.53

Chichimecas lote 4,Manzana 81
Col. Ajusco
México 22, D.F.
Tel.677.64.78

35. PRUDENCIO JORGE RIVERA MENCHACA
PHILCO, S.A.
CLAVEL 157
MEXICO 4, D.F.
TEL.547.46.00

Vainilla 405-2
Col. Granjas México
México 8, D.F.
Tel.657.85.38

36. ALEJANDRO ROMAY MUÑOZ DE COTE
MICROBYTE S.A.DE C.V.
B. CALIFORNIA 375-5
MEXICO 11, D.F.
TEL.516.20.38

Amores 1065
México 12, D.F.
Tel.575.11.07

37. GILBERTO SANTOS ARAOZ
S.A.H.O.P.
XOLA ESQ. AV. UNIVERSIDAD
MEXICO 12, d.f.
TEL.519.51.34

Costa Rica 44-17
México 2, D.F.
Tel.529.19.84

38. ANTONIO SANTOS MORENO
UNIVERSIDAD DE LAS AMERICAS
ST. CATARINA MARTIR
COLULA, PUEBLA.

29 Sur 714
La Paz, Puebla
Puebla ,Pue.
Tel.48.12.37

39. MIGUEL SERRANO RICAÑO
S. C. T.
CENTRO SCOP
MEXICO,D.F.
TEL.579.31.50

Tomás Vázquez 169
Iztacalco
México 13, D.F.
Tel.579.12.89

40. TENORIO GUILLEN ENRIQUE ANDRES
MICROMEX
MEXICO,D.F.
TEL.575.79.31

Cerro Macuiltepec 301-401
México 21, D.F.
Tel.544.45.44

41. AUSTREBERTO VAZQUEZ ZAPATA
CALLE NTE.94 # 8312
LA ESMERALDA
MEXICO 14, D.F.

42. LUIS FRANCISCO VERDIGUEL ALVAREZ
CENTRO DE INSTRUMENTOS
UNAM
MEXICO 20, D.F.

I. LA CATOLICA 289-1
COL. OBRERA
MEXICO 8,D.F.
Tel.588.39.25

43. HECTOR YAÑEZ
PHILCO S.A.
CLAVEL 157
MEXICO 4, D.F.
TEL.547.46.00

Ciencias 54
Col. Escandón
México, D.F.
Tel.515.14.63

44. L. ENRIQUE ZARATE LEYVA
S.A.R.H.
P. DE LA REFORMA 45-11°
MEXICO ,D.F.
TEL.592.00.90

Acuario 60
Fracc. Prados
Tult,Edo. de México