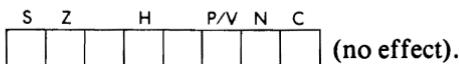


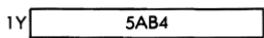
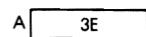
PROGRAMMING THE Z80

Flags:

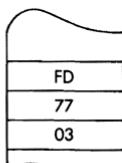
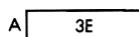


Example: LD (IY + 3), A

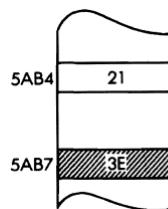
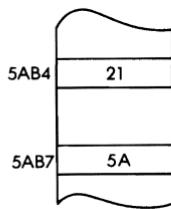
Before:



After:



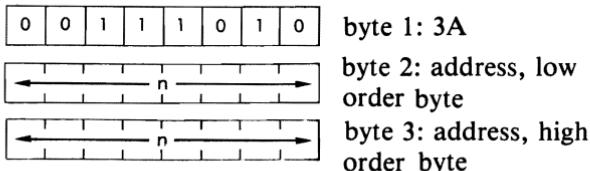
OBJECT CODE



LD A, (nn) Load accumulator from the memory location (nn).

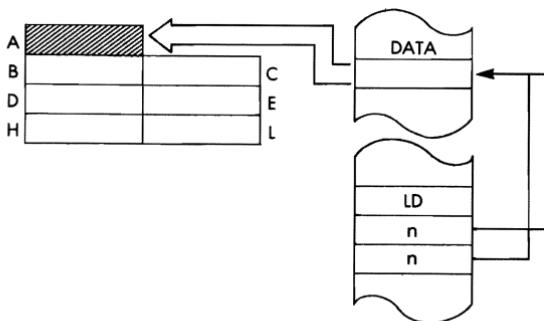
Function: $A \leftarrow (nn)$

Format:



Description: The contents of the memory location addressed by the contents of the 2 memory locations immediately following the opcode are loaded into the accumulator. The low byte of the address occurs immediately after the opcode.

Data Flow:

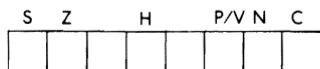


Timing: 4 M cycles; 13 T states; 6.5 usec @ 2 MHz

Addressing Mode: Direct.

PROGRAMMING THE Z80

Flags:



(no effect).

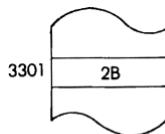
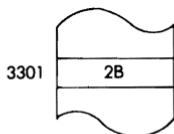
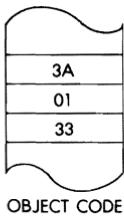
Example:

LD A, (3301)

Before:



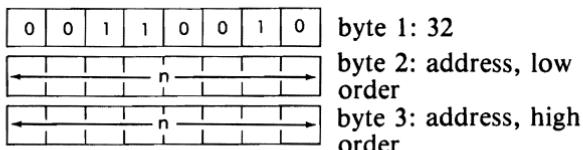
After:



LD (nn),A Load directly addressed memory location (nn) from accumulator.

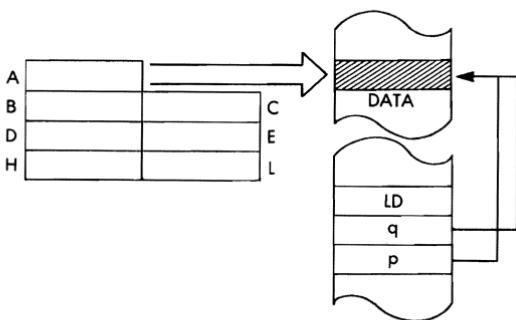
Function: $(nn) \leftarrow A$

Format:



Description: The contents of the accumulator are loaded into the memory location addressed by the contents of the memory locations immediately following the opcode. The low byte of the address immediately follows the opcode.

Data Flow:

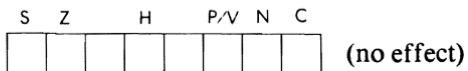


Timing: 4 M cycles; 13 T states; 6.5 usec @ 2 MHz

Addressing Mode: Direct.

PROGRAMMING THE Z80

Flags:



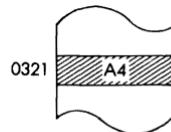
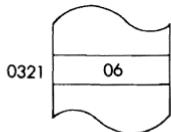
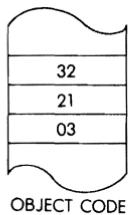
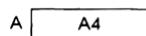
Example:

LD (0321), A

Before:



After:



LD (nn), dd

Load memory locations addressed by nn from register pair rr.

Function:

$(nn) \leftarrow dd_{low}; (nn + 1) \leftarrow dd_{high}$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

byte 1: ED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | d | d | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

byte 2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| n | n | n | n | n | n | n | n |
|---|---|---|---|---|---|---|---|

byte 3: address,
low order

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| n | n | n | n | n | n | n | n |
|---|---|---|---|---|---|---|---|

byte 4: address,
high order

Descriptions:

The contents of the low order of the specified register pair are loaded into the memory location addressed by the memory locations immediately following the opcode. The contents of the high order of the register pair are loaded into the memory location immediately following the one loaded from the low order. The low order of the nn address occurs immediately after the opcode. dd may be anyone of:

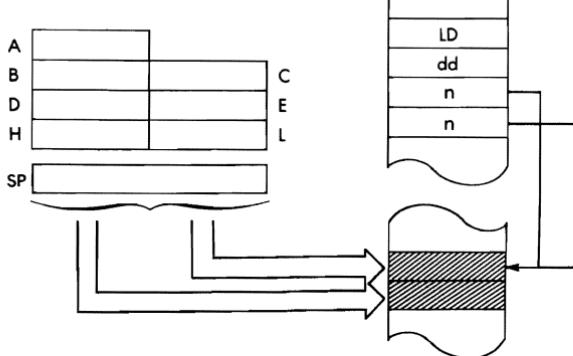
BC - 00

HL - 10

DE - 01

SP - 11

Data Flow:



PROGRAMMING THE Z80

Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

Byte Codes: dd: BC DE HL SP
ED-

| | | | |
|----|----|----|----|
| 43 | 53 | 63 | 73 |
|----|----|----|----|

Flags: S Z H P/V N C

| | | | | | | |
|--|--|--|--|--|--|--|
| | | | | | | |
|--|--|--|--|--|--|--|

 (no effect).

Example: LD (040B), BC

Before:

B

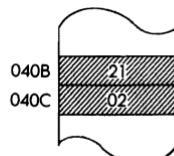
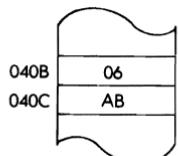
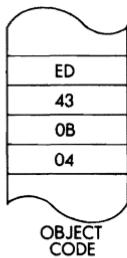
| |
|------|
| 0221 |
|------|

 C B

| |
|------|
| 0221 |
|------|

 C

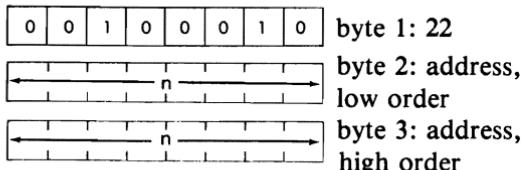
After:



LD (nn), HL Load the memory locations addressed by nn from HL.

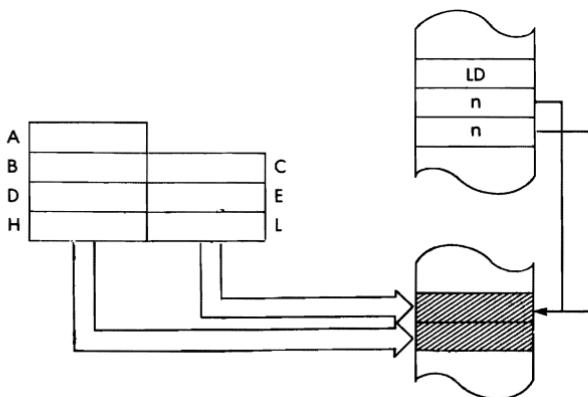
Function: $(nn) \leftarrow L; (nn + 1) \leftarrow H$

Format:



Description: The contents of the L register are loaded into the memory location addressed by the memory locations immediately following the opcode. The contents of the H register are loaded into the memory location immediately following the location loaded from the L register. The low order of the nn address occurs immediately after the opcode.

Data Flow:

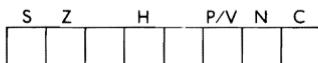


Timing: 5 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: Direct.

PROGRAMMING THE Z80

Flags:



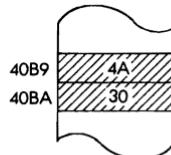
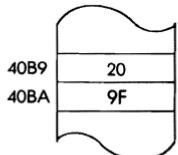
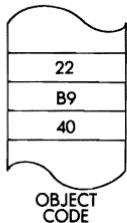
(no effect).

Example: LD (40B9), HL

Before:

H [] 304A [] L H [] 304A [] L

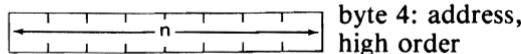
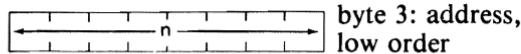
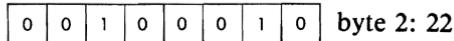
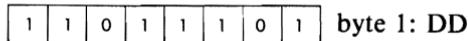
After:



LD (nn), IX Load memory locations addressed by nn from IX.

Function: $(nn) \leftarrow IX_{low}; (nn + 1) \leftarrow IX_{high}$

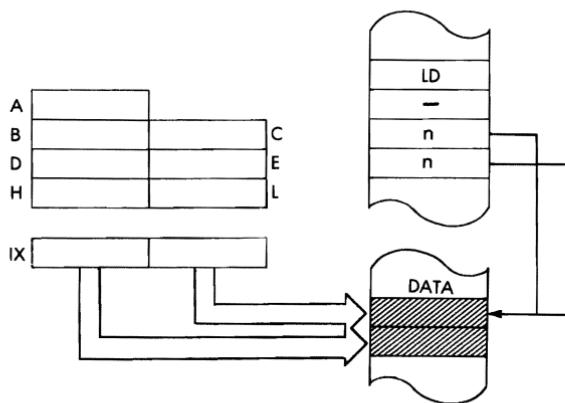
Format:



Description:

The contents of the low order of the IX register are loaded into the memory location addressed by the contents of the memory location immediately following the opcode. The contents of the high order of the IX register are loaded into the memory location immediately following the one loaded from the low order. The low order of the nn address occurs immediately after the op code.

Data Flow:

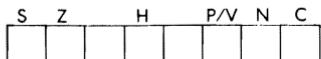


Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

PROGRAMMING THE Z80

Flags:



(no effect).

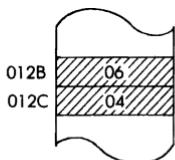
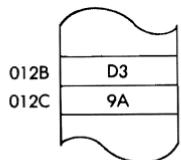
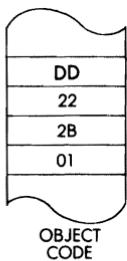
Example: LD (012B), IX

Before:

IX [] 0406

After:

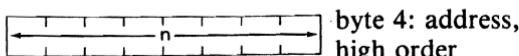
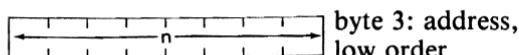
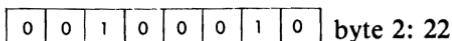
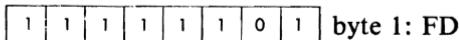
IX [] 0406



LD (nn), IY Load memory locations addressed by nn from IY.

Function: $(nn) \leftarrow IY_{low}; (nn + 1) \leftarrow IY_{high}$

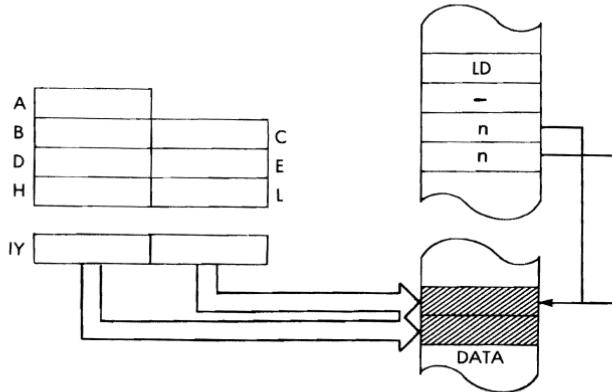
Format:



Description:

The contents of the low order of the IY register are loaded into the memory location addressed by the contents of the memory locations immediately following the opcode. The contents of the high order of the IY register are loaded into the memory location immediately following the one loaded from the low order. The low order of the nn address occurs immediately after the opcode.

Data Flow:

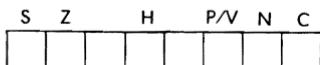


Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

PROGRAMMING THE Z80

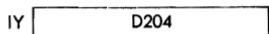
Flags:



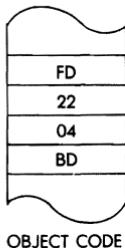
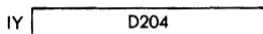
(no effect)

Example: LD (BD04), IY

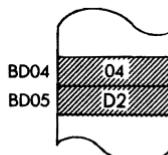
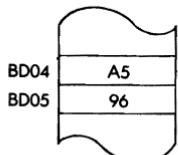
Before:



After:



OBJECT CODE



LD A, (BC)

Load accumulator from the memory location indirectly addressed by the BC register pair.

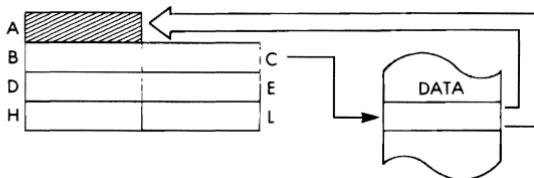
Function: $A \leftarrow (BC)$

Format:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0A |
|---|---|---|---|---|---|---|---|----|

Description: The contents of the memory location addressed by the contents of the BC register pair are loaded into the accumulator.

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|

(no effect).

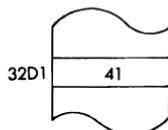
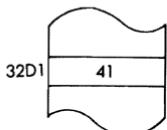
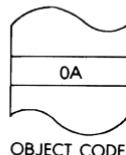
Example: LD A, (BC)

Before:

| | | |
|---|------|---|
| A | AB | C |
| B | 32D1 | C |

After:

| | | |
|---|------|---|
| A | 41 | C |
| B | 32D1 | C |



OBJECT CODE

PROGRAMMING THE Z80

LD A, (DE) Load the accumulator from the memory location indirectly addressed by the DE register pair.

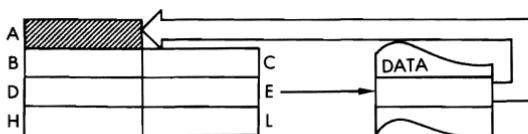
Function: $A \leftarrow (DE)$

Format:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----|
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1A |
|---|---|---|---|---|---|---|---|----|

Description: The contents of the memory location addressed by the contents of the DE register pair are loaded into the accumulator.

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags: S Z H P/V N C (No effect).

Example: LD A, (DE)

Before:

| | |
|---|----|
| A | D2 |
|---|----|

| | | |
|---|------|---|
| D | 6051 | E |
|---|------|---|

After:

| | |
|---|----|
| A | 09 |
|---|----|

| | | |
|---|------|---|
| D | 6051 | E |
|---|------|---|

OBJECT CODE

6051
09

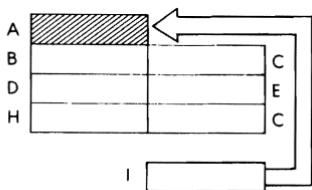
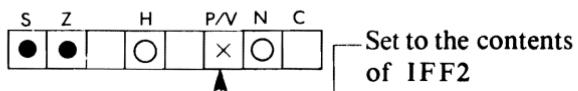
6051
09

LD A, I

Load accumulator from interrupt vector register I.

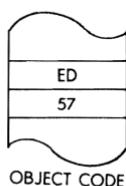
Function: $A \leftarrow I$ *Format:*

| | | | | | | | | | |
|--|---|---|---|---|---|---|---|---|------------|
| <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table> | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | byte 1: ED |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | | |
| <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> </table> | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | byte 2: 57 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | | |

Description: The contents of the interrupt vector register are loaded into the accumulator.*Data Flow:**Timing:* 2 M cycles; 9 T states; 4.5 usec @ 2 MHz*Addressing Mode:* Implicit.*Flags:**Example:* LD A, I

Before:

After:



PROGRAMMING THE Z80

LD I, A

Load Interrupt Vector register I from the accumulator.

Function: $I \leftarrow A$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

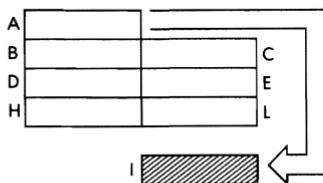
 byte 1: ED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: 47

Description: The contents of the accumulator are loaded into the Interrupt Vector register.

Data Flow:



Timing: 2 M cycles; 9 T states; 4.5 usec @ 2 MHz

Addressing Mode: Implicit.

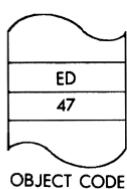
Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|

 (no effect)

Example: LD I, A

Before:



After:

| | | | | | | | |
|---|----|---|----|---|----|---|----|
| A | 06 | I | D2 | A | 06 | I | 06 |
|---|----|---|----|---|----|---|----|

LD A, R

Load accumulator from Memory Refresh register R.

Function: $A \leftarrow R$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

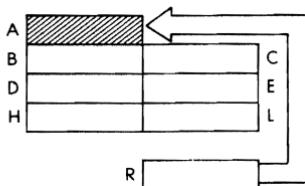
byte 1: ED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

byte 2: 5F

Description: The contents of the Memory Refresh register are loaded into the accumulator.

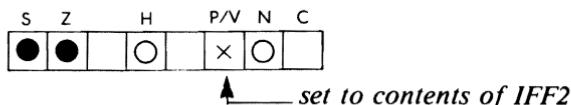
Data Flow:



Timing: 2 M cycles; 9 T states; 4.5 usec @ 2 MHz

Addressing Mode: Implicit.

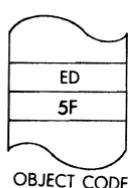
Flags:



Example: LD A, R

Before:

After:

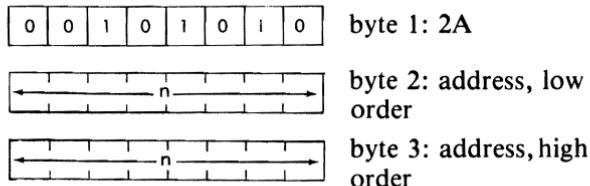


| | | | | | | | |
|---|----|---|----|---|----|---|----|
| A | 62 | R | 4A | A | 4A | R | 4A |
|---|----|---|----|---|----|---|----|

LD HL, (nn) Load HL register from memory locations addressed by nn.

Function: $L \leftarrow (nn); H \leftarrow (nn + 1)$

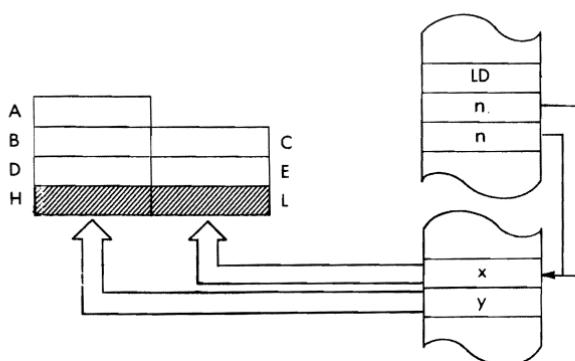
Format:



Description:

The contents of the memory location addressed by the memory locations immediately after the opcode are loaded into the L register. The contents of the memory location after the one loaded into the L register are loaded into the H register. The low byte of the nn address occurs immediately after the opcode.

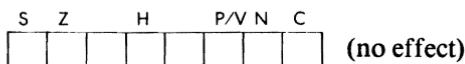
Data Flow:



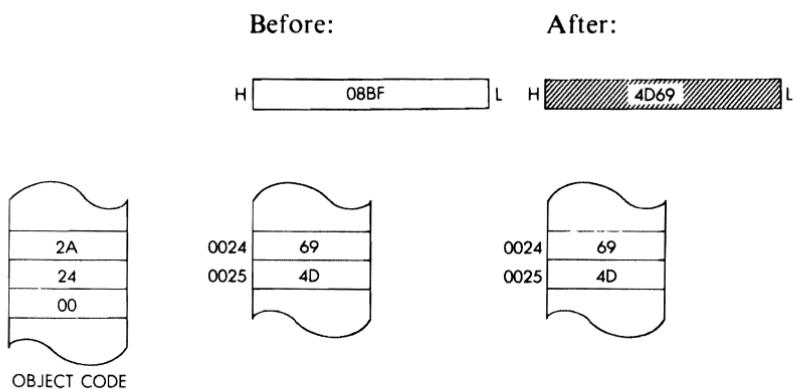
Timing: 5 M cycles, 16 T states; 8 usec @ 2 MHz

Addressing Mode: Direct.

Flags:

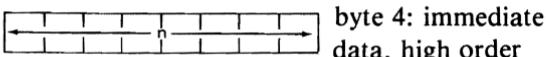
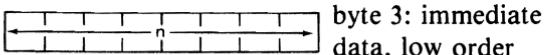
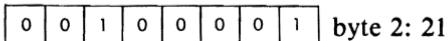
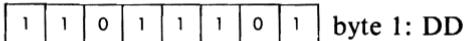


Example: LD HL, (0024)

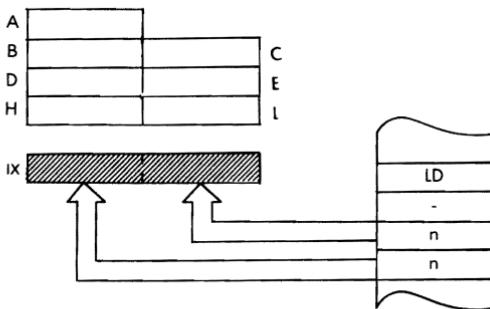
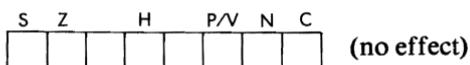


LD IX, nn

Load IX register with immediate data nn.

Function: $IX \leftarrow nn$ *Format:**Description:*

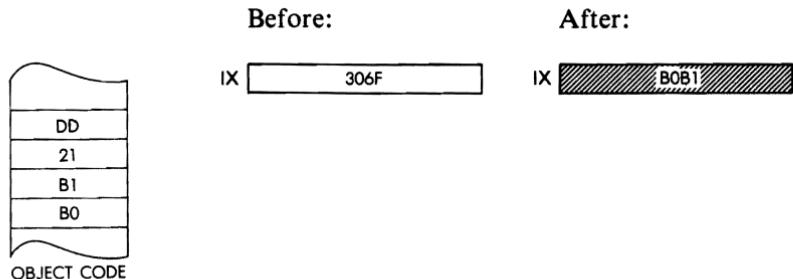
The contents of the memory locations immediately following the opcode are loaded into the IX register. The low order byte occurs immediately after the opcode.

Data Flow:*Timing:* 4 M cycles; 14 T states; 7 usec @ 2 MHz*Addressing Mode:* Immediate.*Flags:*

THE Z80 INSTRUCTION SET

Example:

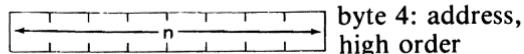
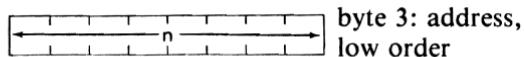
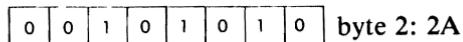
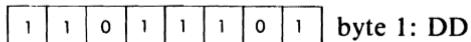
LD IX, B0B1



LD IX, (nn) Load IX register from memory locations addressed by nn.

Function: $\text{IX}_{\text{low}} \leftarrow (\text{nn}); \text{IX}_{\text{high}} \leftarrow (\text{nn} + 1)$

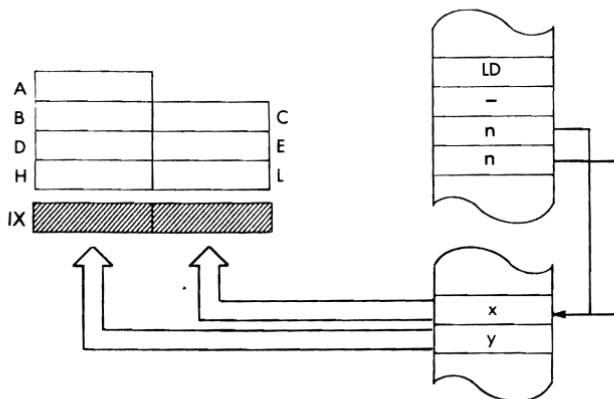
Format:



Descriptions:

The contents of the memory location addressed by the memory locations immediately following the opcode are loaded into the low order of the IX register. The contents of the memory location immediately following the one loaded into the low order are loaded into the high order of the IX register. The low order of the nn address immediately follows the opcode.

Data Flow:

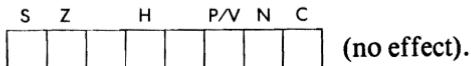


Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

THE Z80 INSTRUCTION SET

Flags:



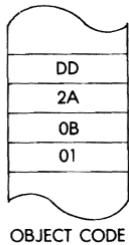
Example:

LD IX, (010B)

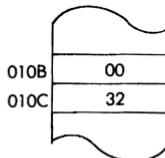
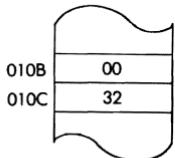
Before:



After:

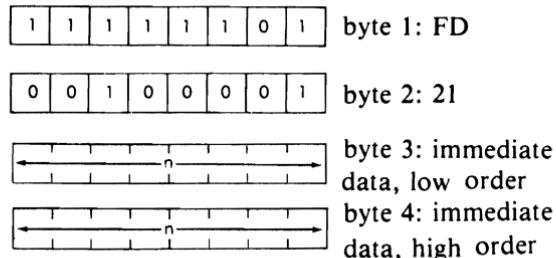


OBJECT CODE

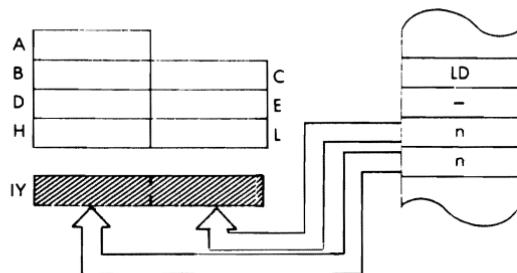


LD IY, nn

Load IY register with immediate data nn.

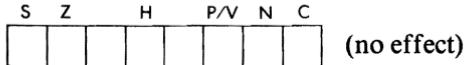
Function: $IY \leftarrow nn$ *Format:**Description:*

The contents of the memory locations immediately following the opcode are loaded into the IY register. The low order byte occurs immediately after the opcode.

Data Flow:*Timing:* 4 M cycles; 14 T states; 7 usec @ 2 MHz*Addressing Mode:* Immediate.

THE Z80 INSTRUCTION SET

Flags:



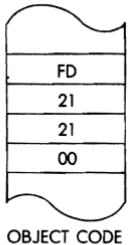
Example:

LD IY, 21

Before:

After:

IY [069B] IY [0021]



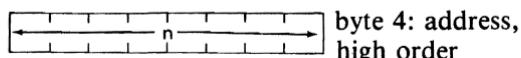
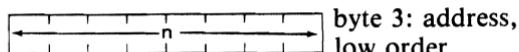
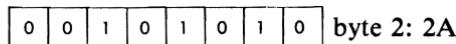
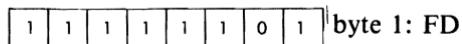
LD IY, (nn)

Load register IY from memory locations addressed by nn.

Function:

$IY_{low} \leftarrow (nn); IY_{high} \leftarrow (nn + 1)$

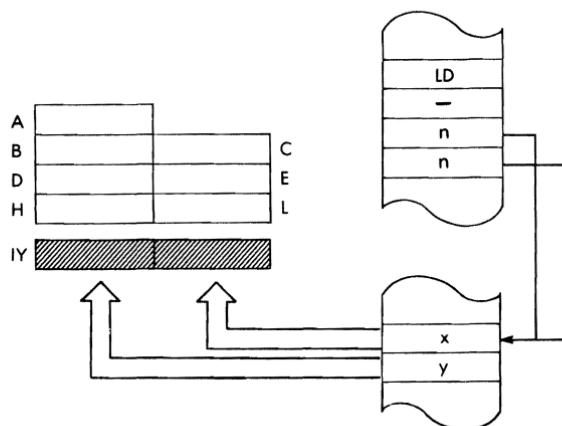
Format:



Description:

The contents of the memory location addressed by the memory locations immediately following the opcode are loaded into the low order of the IY register. The contents of the memory location immediately following the one loaded into the low order are loaded into the high order of the IY register. The low order of the nn address immediately follows the opcode.

Data Flow:

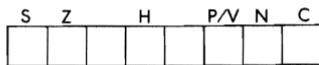


THE Z80 INSTRUCTION SET

Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

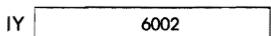
Flags:



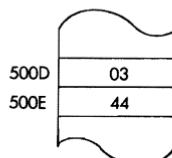
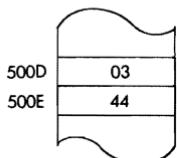
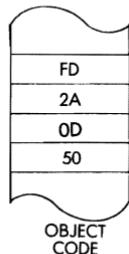
(no effect).

Example: LD IY, (500D)

Before:



After:



PROGRAMMING THE Z80

LD R,A

Load Memory Refresh register R from the accumulator.

Function:

$R \leftarrow A$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

byte 1: ED

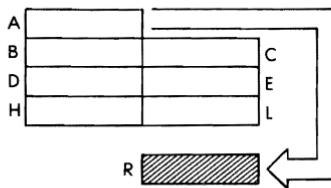
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

byte 2: 4F

Description:

The contents of the accumulator are loaded into the Memory Refresh register.

Data Flow:



Timing: 2 M cycles; 9 T states; 4.5 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| | | | | | |

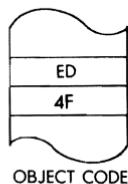
(no effect)

Example:

LD R, A

Before:

After:



| | | | | | | | |
|---|----|---|----|---|----|---|----|
| A | OF | R | 40 | A | OF | R | OF |
|---|----|---|----|---|----|---|----|

LD SP, HL Load stack pointer from HL.

Function: $SP \leftarrow HL$

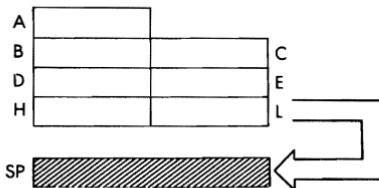
Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

F9

Description: The contents of the HL register pair are loaded into the stack pointer.

Data Flow:



Timing: 1 M cycles; 6 T states; 3 usec @ 2 MHz

Addressing Mode: Implicit.

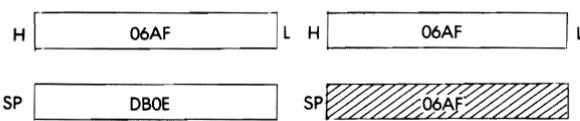
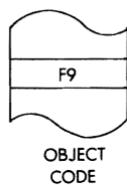
Flags: S Z H P/V N C

(no effect)

Example: LD SP, HL

Before:

After:

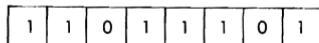


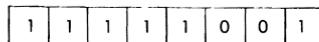
PROGRAMMING THE Z80

LD SP, IX Load stack pointer from IX register.

Function: $SP \leftarrow IX$

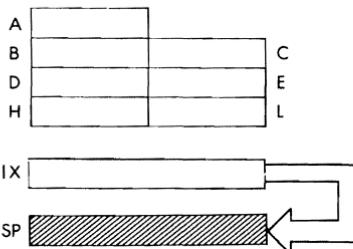
Format:

 byte 1: DD

 byte 2: F9

Description: The contents of the IX register are loaded into the stack pointer.

Data Flow:



Timing: 2 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Implicit.

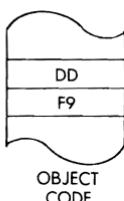
Flags:

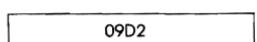
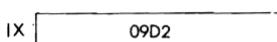
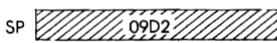
S Z H P/V N C
 (no effect)

Example: LD SP, IX

Before:

After:

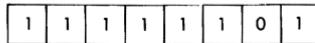


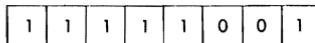
IX  IX 
SP  SP 

LD SP, IY Load stack pointer from IY register.

Function: $SP \leftarrow IY$

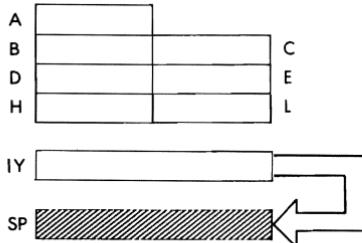
Format:

 byte 1: FD

 byte 2: F9

Description: The contents of the IY register are loaded into the stack pointer.

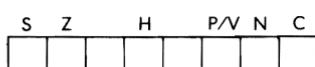
Data Flow:



Timing: 2 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Implicit.

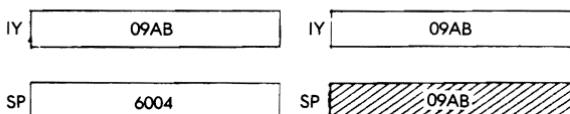
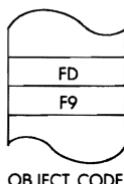
Flags:

 (no effect)

Example: LD SP, IY

Before:

After:



PROGRAMMING THE Z80

LDD

Block load with decrement.

Function:

$(DE) \leftarrow (HL); DE \leftarrow DE - 1; HL \leftarrow HL - 1;$
 $BC \leftarrow BC - 1$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

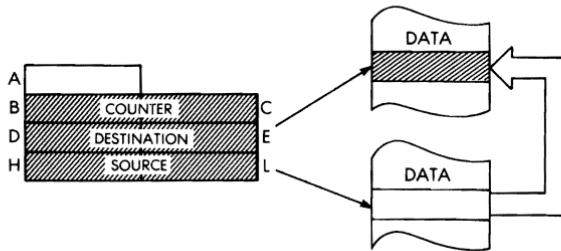
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

 byte 2: A8

Description:

The contents of the memory location addressed by HL are loaded into the memory location addressed by DE. Then BC, DE, and HL are all decremented.

Data Flow:



Timing: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Modes: Indirect.

Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| | | ○ | X | ○ | |

Reset if BC = 0 after execution, set otherwise.

Example:

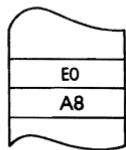
LDD

Before:

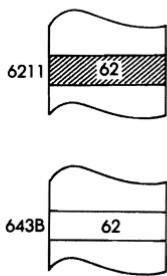
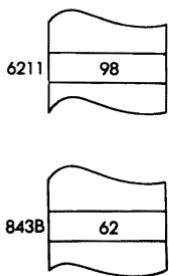
| | | |
|---|------|---|
| B | 0B04 | C |
| D | 6211 | E |
| H | 843B | L |

After:

| | | |
|---|------|---|
| B | 0B03 | C |
| D | 6210 | E |
| H | 843A | L |



OBJECT CODE



PROGRAMMING THE Z80

LDDR

Repeating block load with decrement.

Function:

$(DE) \leftarrow (HL); DE \leftarrow DE - 1; HL \leftarrow HL - 1;$
 $BC \leftarrow BC - 1; \text{Repeat until } BC = 0$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

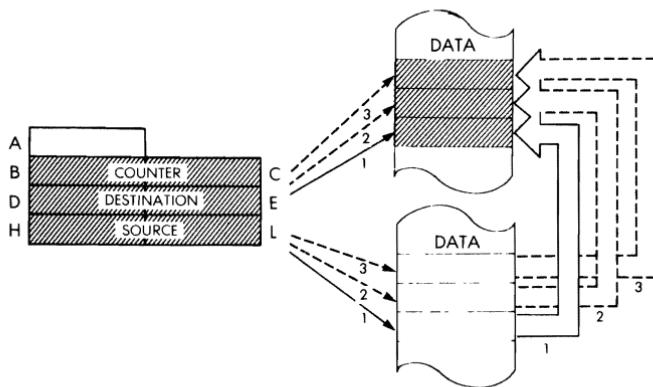
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

 byte 2: B8

Description:

The contents of the memory location addressed by HL are loaded into the memory location addressed by DE. Then DE, HL, and BC are all decremented. If $BC \neq 0$, then the program counter is decremented by 2 and the instruction re-executed.

Data Flow:



Timing: $BC \neq 0$: 5 M cycles; 21 T states; 10.5 usec @ 2 MHz.

$BC = 0$: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

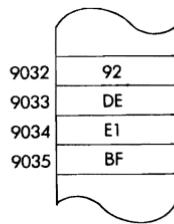
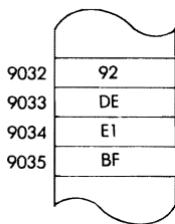
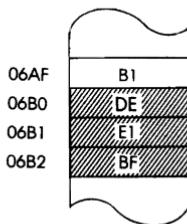
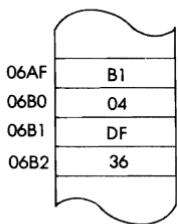
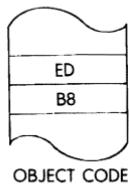
| | | | | | |
|-----|-----|-----|-----|-----|-----|
| S | Z | H | P/V | N | C |
| [] | [] | [O] | [O] | [O] | [] |

*Example:***LDDR****Before:**

| | |
|---|------|
| B | 0003 |
| D | 06B2 |
| H | 9035 |

After:

| | |
|---|------|
| C | 0000 |
| D | 06AF |
| H | 9032 |



LDI

Block load with increment.

Function:

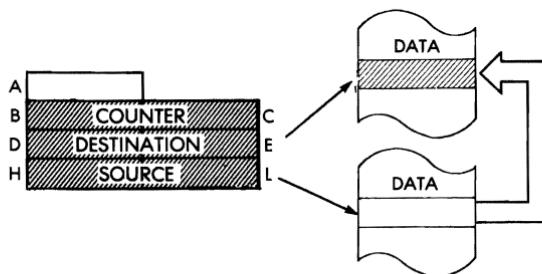
$$(DE) \leftarrow (HL); DE \leftarrow DE + 1; HL \leftarrow HL + 1;$$

$$BC \leftarrow BC - 1$$
Format:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|------------|
| <table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table> | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | byte 1: ED |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | | |
| <table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | byte 2: A0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | | |

Description:

The contents of the memory location addressed by HL are loaded into the memory location addressed by DE. Then both DE and HL are incremented, and the register pair BC is decremented.

Data Flow:*Timing:*

4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: Indirect.*Flags:*

| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|
| | | ○ | ✗ | ○ | |

Reset if BC = 0 after execution, set otherwise.

THE Z80 INSTRUCTION SET

Example:

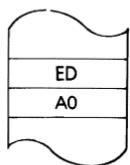
LDI

Before:

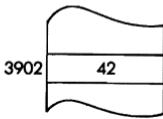
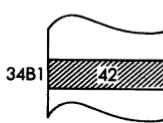
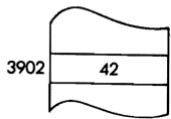
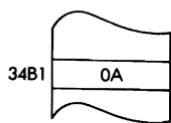
| | | |
|---|------|---|
| B | 0006 | C |
| D | 34B1 | E |
| H | 3902 | L |

After:

| | | |
|---|------|---|
| B | 0005 | C |
| D | 34B2 | E |
| H | 3903 | L |



OBJECT CODE



LDIR

Repeating block load with increment.

Function:

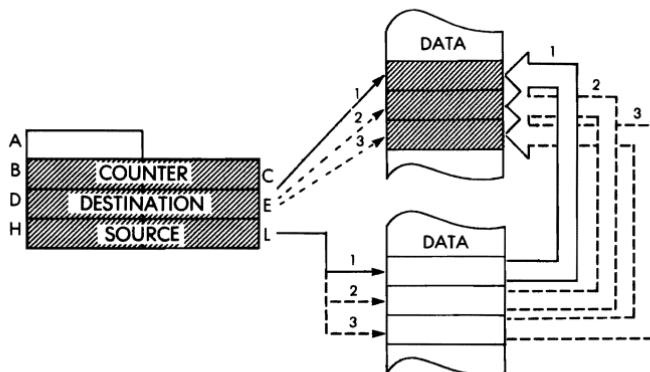
$$(DE) \leftarrow (HL); DE \leftarrow DE + 1; HL \leftarrow HL + 1;$$

$$BC \leftarrow BC - 1; \text{ Repeat until } BC = 0$$
Format:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|------------|
| <table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table> | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | byte 1: ED |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | | |
| <table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | byte 2: B0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | | |

Description:

The contents of the memory location addressed by HL are loaded into the memory location addressed by DE. Then both DE and HL are incremented. BC is decremented. If $BC \neq 0$, then the program counter is decremented by 2 and the instruction is re-executed.

Data Flow:*Timing:*

For $BC \neq 0$: 5M cycles; 21 T states; 10.5 usec @ 2 MHz.

For $BC = 0$: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: Indirect.

THE Z80 INSTRUCTION SET

Flags:



Example:

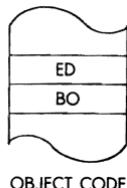
LDIR

Before:

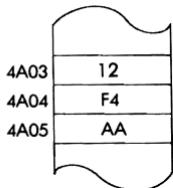
| | | |
|---|------|---|
| B | 0002 | C |
| D | 4A03 | E |
| H | 962A | L |

After:

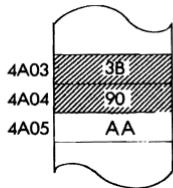
| | | |
|---|------|---|
| B | 0000 | C |
| D | 4A05 | E |
| H | 962C | L |



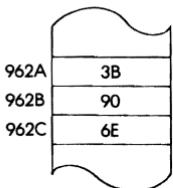
OBJECT CODE



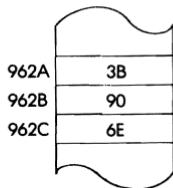
| | |
|------|----|
| 4A03 | 12 |
| 4A04 | F4 |
| 4A05 | AA |



| | |
|------|----|
| 4A03 | 3B |
| 4A04 | 90 |
| 4A05 | AA |



| | |
|------|----|
| 962A | 3B |
| 962B | 90 |
| 962C | 6E |



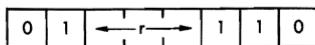
| | |
|------|----|
| 962A | 3B |
| 962B | 90 |
| 962C | 6E |

PROGRAMMING THE Z80

LD r, (HL) Load register r indirect from memory location (HL).

Function: $r \leftarrow (HL)$

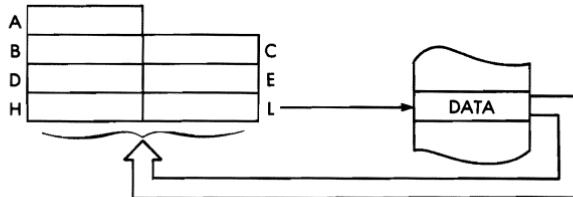
Format:



Description: The contents of the memory location addressed by HL are loaded into the specified register. r may be any one of:

| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

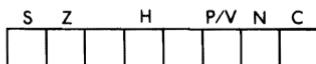
Addressing Mode: Indirect.

Byte Codes:

| r: | A | B | C | D | E | H | L |
|----|----|----|----|----|----|----|----|
| | 7E | 46 | 4E | 56 | 5E | 66 | 6E |

THE Z80 INSTRUCTION SET

Flags:

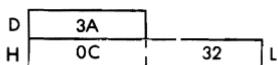


(no effect).

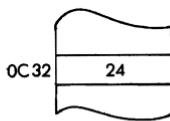
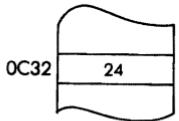
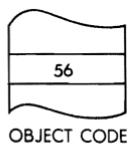
Example:

LD D, (HL)

Before:



After:



PROGRAMMING THE Z80

NEG

Negate accumulator.

Function: $A \leftarrow 0 - A$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

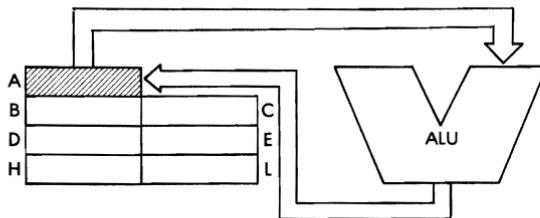
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

 byte 2: 44

Description:

The contents of the accumulator are subtracted from zero (two's complement) and the result is stored back in the accumulator.

Data Flow:



Timing: 2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

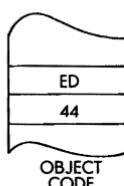
Flags:

| | | | | | |
|---|---|---|---------------|---|---|
| S | Z | H | \oplus/\vee | N | C |
| ● | ● | | ● | 1 | ● |

C will be set if A was 0 before the instruction.
P will be set if A was 80H.

Example:

NEG



Before:

A

| |
|----|
| 32 |
|----|

After:

A

| |
|-----|
| /CE |
|-----|

NOP No operation.

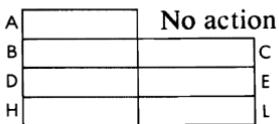
Function: Delay.

Format:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 |
|---|---|---|---|---|---|---|---|----|

Description: Nothing is done for 1 M cycle.

Data Flow:



Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit

Flags:

| | | | | | | | |
|---|---|---|-----|---|---|--|--|
| S | Z | H | P/V | N | C | | |
| | | | | | | | |

(no effect).

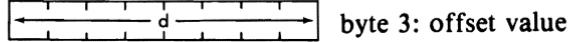
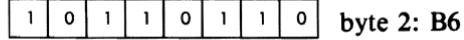
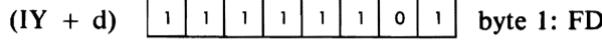
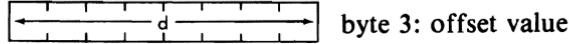
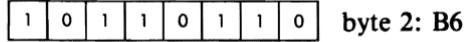
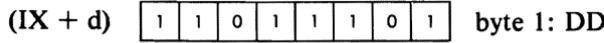
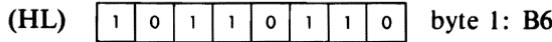
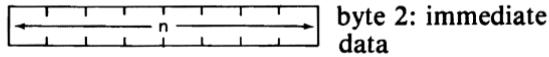
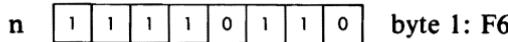
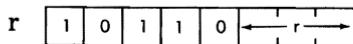
PROGRAMMING THE Z80

OR s

Logical or accumulator and operand s.

Function: $A \leftarrow A \vee s$

Format: s: may be r, n, (HL), (IX + d), or (IY + d)

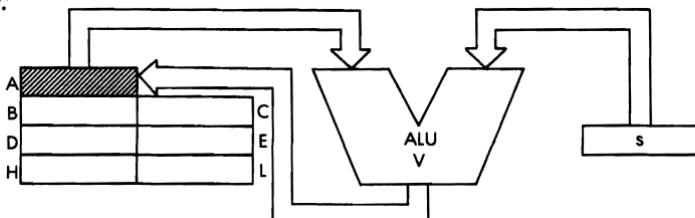


r may be any one of:

| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The accumulator and the specified operand are logically 'or'ed, and the result is stored in the accumulator. s is defined in the description of the similar ADD instructions.

Data Flow:*Timing:*

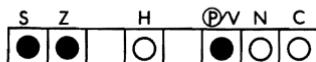
| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 1 | 4 | 4 |
| n | 2 | 7 | 3.5 |
| (HL) | 2 | 7 | 3.5 |
| (IX + d) | 5 | 19 | 9.5 |
| (IY + d) | 5 | 19 | 9.5 |

Addressing Mode: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

OR r

| | | | | | | |
|------|----|----|----|----|----|----|
| r: A | B | C | D | E | H | L |
| B7 | B0 | B1 | B2 | B3 | B4 | B5 |

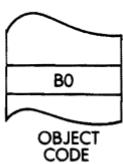
Flags:*Example:*

OR B

*Before:**After:*

| | |
|---|----|
| A | 06 |
| B | B9 |

| | |
|---|----|
| A | BF |
| B | B9 |



PROGRAMMING THE Z80

OTDR

Block output with decrement

Function:

$(C) \leftarrow (HL); B \leftarrow B - 1; HL \leftarrow HL - 1;$
Repeat until $B = 0$.

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

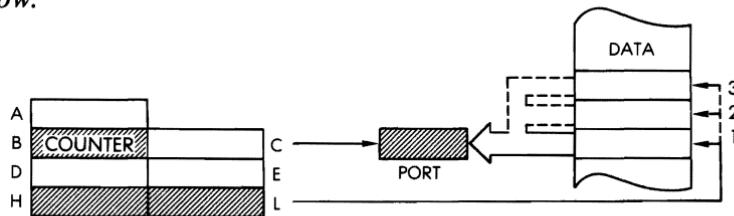
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: BB

Description:

The contents of the memory location addressed by the HL register pair are output to the peripheral device addressed by the contents of the C register. Both the B register and the HL register pair are then decremented. If $B \neq 0$, the program counter is decremented by 2 and the instruction is re-executed. C supplies bits A0 to A7 of the address bus. B supplies (after decrementation) bits A8 to A15.

Data Flow:



Timing:

$B = 0$: 4 M cycles; 16 T states; 8 usec @ 2 MHz.
 $B \neq 0$: 5 M cycles; 21 T states; 10.5 usec @ 2 MHz

Addressing Mode: External.

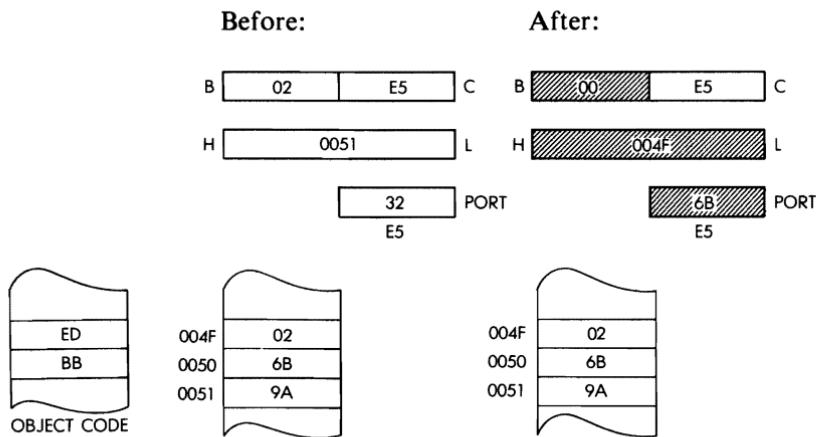
Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| ? | 1 | ? | ? | 1 | |

THE Z80 INSTRUCTION SET

Example:

OTDR



PROGRAMMING THE Z80

OTIR

Block output with increment.

Function: $(C) \leftarrow (HL); B \leftarrow B - 1; HL \leftarrow HL + 1;$ Repeat until $B = 0$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

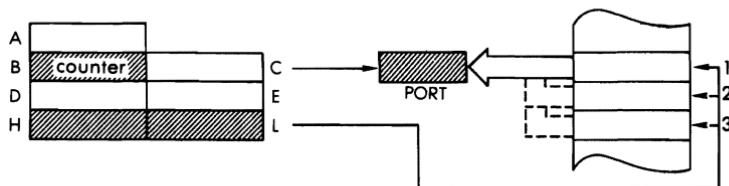
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: B3

Description:

The contents of the memory location addressed by the HL register pair are output to the peripheral device addressed by the contents of the C register. The B register is decremented and the HL register pair is incremented. If $B \neq 0$, the program counter is decremented by 2 and the instruction is re-executed. C supplies bits A0 to A7 of the address bus. B supplies (after decrementation) bits A8 to A15.

Data Flow:



Timing:

$B = 0$: 4 M cycles; 16 T states; 8 usec @ 2 MHz.
 $B \neq 0$: 5 M cycles; 21 T states; 10.5 usec @ 2 MHz

Addressing Mode: External.

Flags:

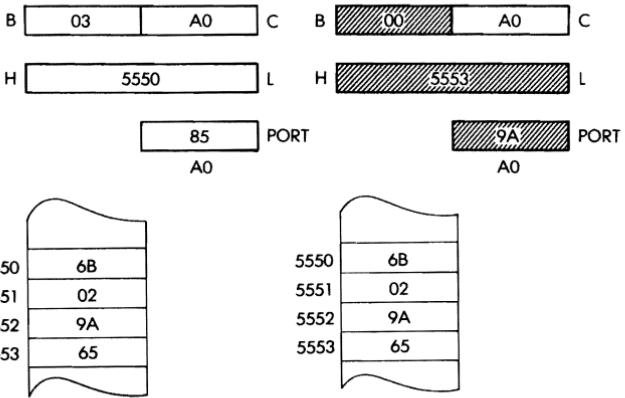
| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| ? | 1 | ? | ? | 1 | |

THE Z80 INSTRUCTION SET

Example:

OTIR

Before:



After:

PROGRAMMING THE Z80

OUT (C), r Output register r to port C.

Function: $(C) \leftarrow r$

Format:

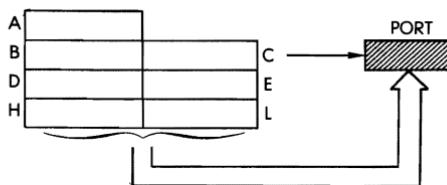
| | | | | | | | | |
|---|---|---|---|---|---|---|---|------------|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | byte 1: ED |
| 0 | 1 | r | | | 0 | 0 | 1 | byte 2 |

Description: The contents of the specified register are output to the peripheral device addressed by the contents of the C register. r may be any one of:

| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Register C supplies bits A0 to A7 of the address bus. Register B supplies bits A8 to A15.

Data Flow:



Timing: 3 M cycles; 12 T states; 6 usec @ 2 MHz

Addressing Mode: External.

Flags:

| | | | | | | |
|---|---|---|-----|---|---|--------------|
| S | Z | H | P/V | N | C | (no effect). |
| | | | | | | |

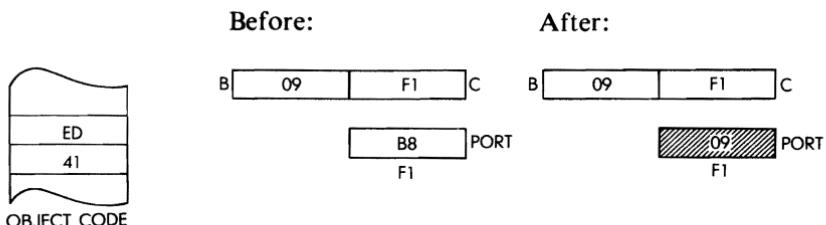
Byte Codes:

| | | | | | | | | |
|-----|----|----|----|----|----|----|----|----|
| ED- | r: | A | B | C | D | E | H | L |
| | | 79 | 41 | 49 | 51 | 59 | 61 | 69 |

THE Z80 INSTRUCTION SET

Example:

OUT (C), B

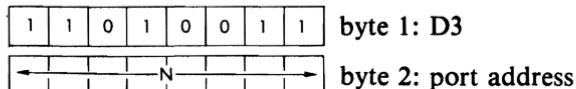


PROGRAMMING THE Z80

OUT (N), A Output accumulator to peripheral port N.

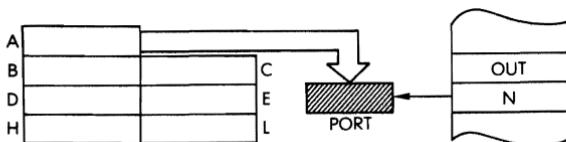
Function: $(N) \leftarrow A$

Format:



Description: The contents of the accumulator are output to the peripheral device addressed by the contents of the memory location immediately following the op-code.

Data Flow:



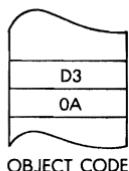
Timing: 3 M cycles, 11 T states; 5.5 usec @ 2 MHz

Addressing Mode: External.

Flags: S Z H P/V N C (no effect).

Example: OUT (0A), A

Before:



After:



OUTD Output with decrement.

Function: $(C) \leftarrow (HL); BC \leftarrow B - 1; HL \leftarrow HL - 1$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

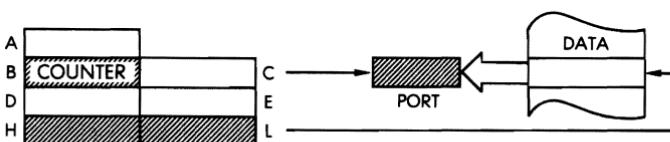
byte 1: ED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

byte 2: AB

Description: The contents of the memory location addressed by the HL register pair are output to the peripheral device addressed by the contents of the C register. Then both the B register and the HL register pair are decremented. C supplies bits A0 to A7 of the address bus. B supplies (after decrementation) A8 to A15.

Data Flow:



Timing: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: External.

Flags:

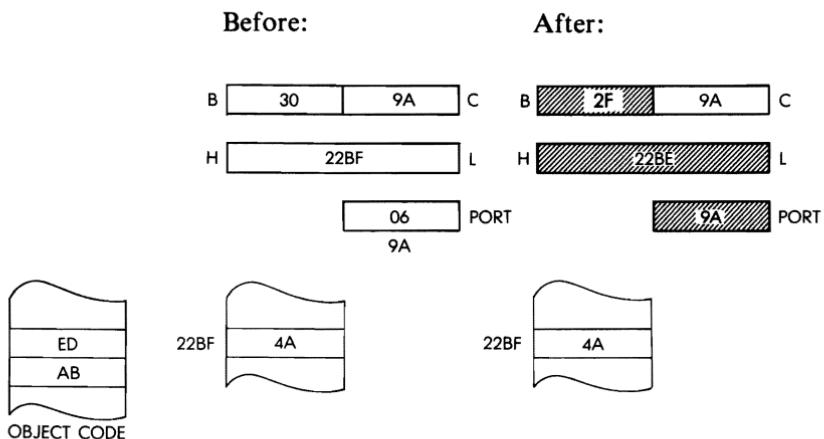
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|
| ? | X | ? | ? | 1 | |

Set if B = 0 after execution, reset otherwise.

PROGRAMMING THE Z80

Example:

OUTD



OUTI

Output with increment.

Function: $(C) \leftarrow (HL); B \leftarrow B - 1; HL \leftarrow HL + 1$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

byte 1: ED

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

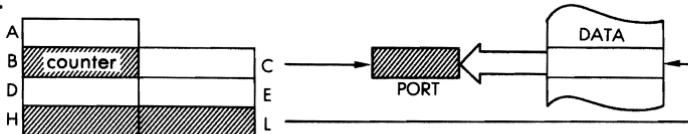
byte 2: A3

Description: The contents of the memory location addressed by the HL register pair are output to the peripheral device addressed by the C register. The B register is decremented and the HL register pair is incremented.

C supplies bits A0 to A7 of the address bus.

B (after decrementation) supplies bits A8 to A15.

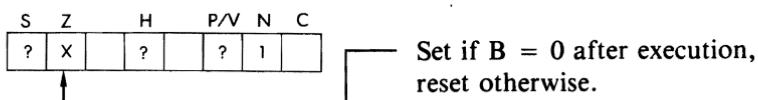
Data Flow:



Timing: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: External.

Flags:



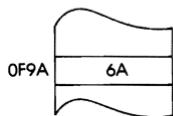
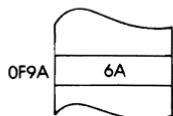
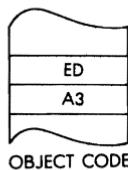
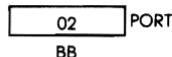
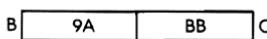
PROGRAMMING THE Z80

Example:

OUTI

Before:

After:



POP qq

Pop register pair qq from stack.

Function: $qq_{low} \leftarrow (SP); qq_{high} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | q | q | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

Description:

The contents of the memory location addressed by the stack pointer are loaded into the low order of the specified register pair and then the stack pointer is incremented. The contents of the memory location now addressed by the stack pointer are loaded into the high order of the register pair, and the stack pointer is again incremented. qq may be any one of:

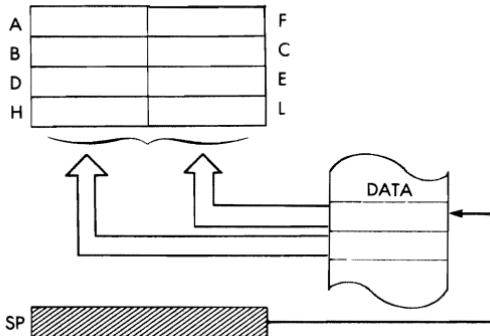
BC - 00

HL - 10

DE - 01

AF - 11

Data Flow:



Timing: 3 M cycles; 10 T states; 5 usec @ 2 MHz

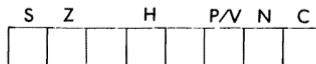
Addressing Mode: Indirect.

Byte Codes:

| | | | | |
|-----|----|----|----|----|
| qq: | BC | DE | HL | AF |
| | C1 | D1 | E1 | F1 |

PROGRAMMING THE Z80

Flags:



(no effect).

Example:

POP BC

Before:

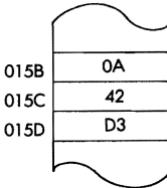
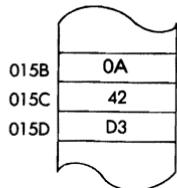
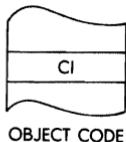
B [] B90A C

SP [] 015B

After:

B [] 420A C

SP [] 015D

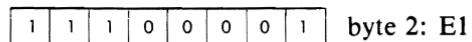
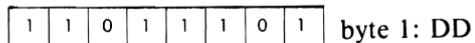


POP IX

POP IX register from stack.

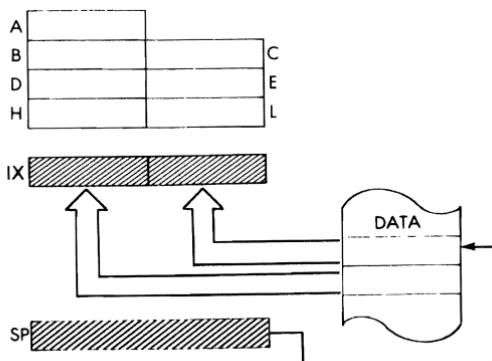
Function: $\text{IX}_{\text{low}} \leftarrow (\text{SP}); \text{IX}_{\text{high}} \leftarrow (\text{SP} + 1); \text{SP} \leftarrow \text{SP} + 2$

Format:



Description: The contents of the memory location addressed by the stack pointer are loaded into the low order of the IX register, and the stack pointer is incremented. The contents of the memory location now addressed by the stack pointer are loaded into the high order of the IX register, and the stack pointer is again incremented.

Data Flow:

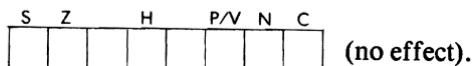


Timing: 4 M cycles; 14 T states; 7 usec @ 2 MHz

Addressing Mode: Indirect.

PROGRAMMING THE Z80

Flags:



Example:

POP IX

Before:

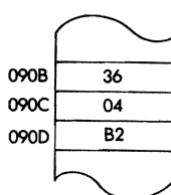
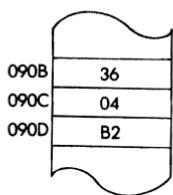
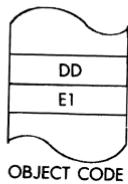
IX 0001

SP 090B

After:

IX 0435

SP 090D



POP IY

POP IY register from stack.

Function: $IY_{low} \leftarrow (SP); IY_{high} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

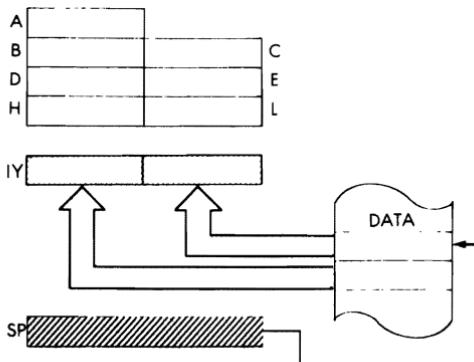
 byte 1: FD

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: E1

Description: The contents of the memory location addressed by the stack pointer are loaded into the low order of the IY register, and then the stack pointer is incremented. The contents of the memory location now addressed by the stack pointer are loaded into the high order of the IY register, and the stack pointer is again incremented.

Data Flow:



Timing: 4 M cycles; 14 T states; 2 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

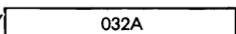
| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|

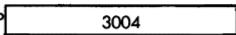
 (no effect).

PROGRAMMING THE Z80

Example: POP IY

Before:

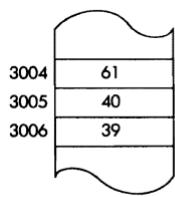
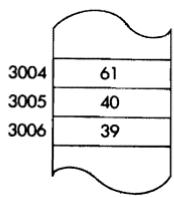
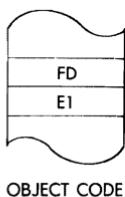
IY 

SP 

After:

IY 

SP 



PUSH qq Push register pair onto stack.

Function: $(SP - 1) \leftarrow qq\text{high}; (SP - 2) \leftarrow qq\text{low};$
 $SP \leftarrow SP - 2$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | q | q | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

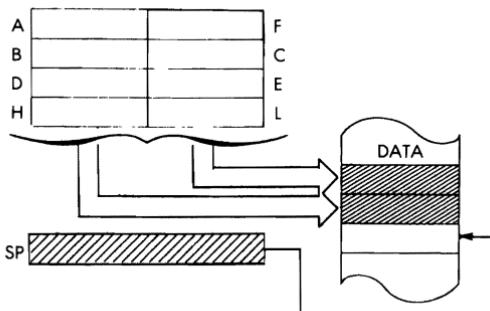
Description:

The stack pointer is decremented and the contents of the high order of the specified register pair are then loaded into the memory location addressed by the stack pointer. The stack pointer is again decremented and the contents of the low order of the register pair are loaded into the memory location currently addressed by the stack pointer. qq may be any one of:

BC – 00
DE – 01

HL – 10
AF – 11

Data Flow:



Timing: 3 M cycles; 11 T states; 6.5 usec @ 2 MHz

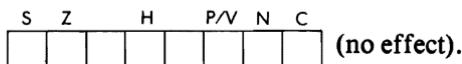
Addressing Mode: Indirect.

Byte Codes: qq: BC DE HL AF

| | | | |
|----|----|----|----|
| C5 | D5 | E5 | F5 |
|----|----|----|----|

PROGRAMMING THE Z80

Flags:



Example:

PUSH DE

Before:

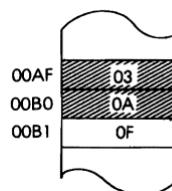
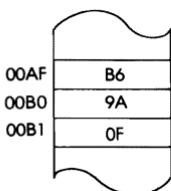
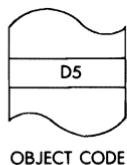
D [0A03] E

SP [00B1]

After:

D [.0A03] E

SP [00AF]



PUSH IX Push IX onto stack.

Function: $(SP - 1) \leftarrow IX_{high}; (SP - 2) \leftarrow IX_{low}; SP \leftarrow SP - 2$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

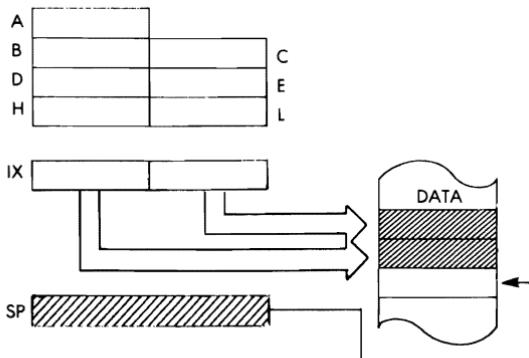
 byte 1: DD

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: E5

Description: The stack pointer is decremented, and the contents of the high order of the IX register are loaded into the memory location addressed by the stack pointer. The stack pointer is again decremented and then the contents of the low order of the IX register are loaded into the memory location addressed by the stack pointer.

Data Flow:



Timing: 4 M cycles; 15 T states; 7.5 usec @ 2 MHz

Addressing Mode: Indirect.

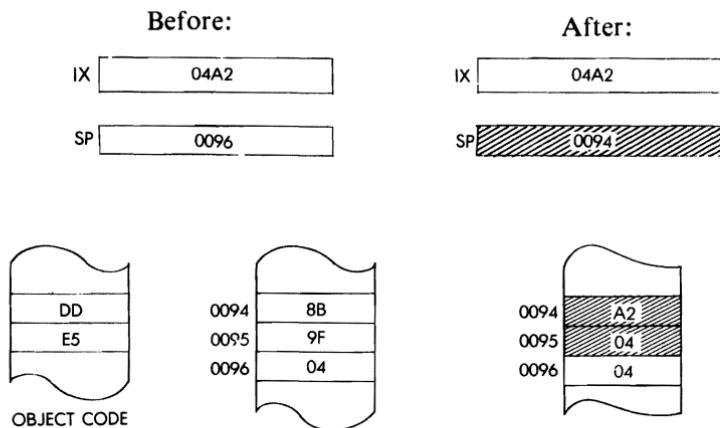
Flags:

| | | | | | | |
|---|---|---|--|-----|---|---|
| S | Z | H | | P/V | N | C |
|---|---|---|--|-----|---|---|

 (no effect)

PROGRAMMING THE Z80

Example: PUSH IX



PUSH IY

Push IY onto stack.

Function: $(SP - 1) \leftarrow IY_{high}; (SP - 2) \leftarrow IY_{low};$
 $SP \leftarrow SP - 2$

Format:

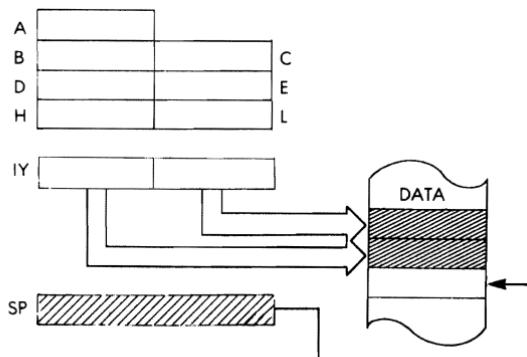
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: FD

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 2: E5
Description:

The stack pointer is decremented and the contents of the high order of the IY register are loaded into the memory location addressed by the stack pointer. The stack pointer is again decremented and the contents of the low order of the IY register are loaded into the memory location addressed by the stack pointer.

Data Flow:*Timing:* 3 M cycles; 15 T states; 7.5 usec @ 2 MHz*Addressing Mode:* Indirect.*Flags:*

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|

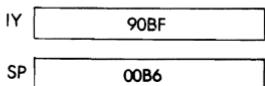
 (no effect)

PROGRAMMING THE Z80

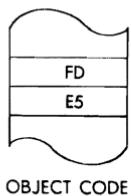
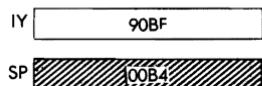
Example:

PUSH IY

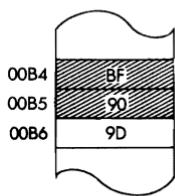
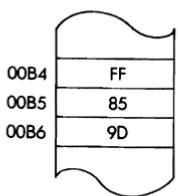
Before:



After:



OBJECT CODE



RES b, s Reset bit b of operand s.*Function:* $s_b \leftarrow 0$ *Format:* s:

| | | |
|----------|--|----------------------|
| r | | byte 1: CB |
| | | byte 2 |
| (HL) | | byte 1: CB |
| | | byte 2 |
| (IX + d) | | byte 1: DD |
| | | byte 2: CB |
| | | byte 3: offset value |
| | | byte 4 |
| (IY + d) | | byte 1: FD |
| | | byte 2: CB |
| | | byte 3: offset value |
| | | byte 4 |

b may be any one of:

| | |
|---------|---------|
| 0 – 000 | 4 – 100 |
| 1 – 001 | 5 – 101 |
| 2 – 010 | 6 – 110 |
| 3 – 011 | 7 – 111 |

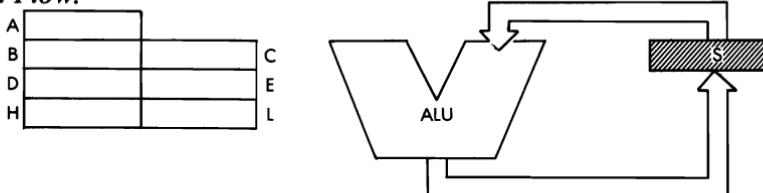
r may be any one of:

| | |
|---------|---------|
| A – 111 | E – 011 |
| B – 000 | H – 100 |
| C – 001 | L – 101 |
| D – 010 | |

PROGRAMMING THE Z80

Description: The specified bit of the location determined by s is reset. s is defined in the description of the similar BIT instructions.

Data Flow:



Timing:

| s: | M cycles: | T states: | usec @ 2 MHz: |
|----------|-----------|-----------|------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

RES b, r

| CB— | b: | r: A | B | C | D | E | H | L |
|-----|----|------|----|----|----|----|----|----|
| 0 | | 87 | 80 | 81 | 82 | 83 | 84 | 85 |
| 1 | | 8F | 88 | 89 | 8A | 8B | 8C | 8D |
| 2 | | 97 | 90 | 91 | 92 | 93 | 94 | 95 |
| 3 | | 9F | 98 | 99 | 9A | 9B | 9C | 9D |
| 4 | | A7 | A0 | A1 | A2 | A3 | A4 | A5 |
| 5 | | AF | A8 | A9 | AA | AB | AC | AD |
| 6 | | B7 | B0 | B1 | B2 | B3 | B4 | B5 |
| 7 | | BF | B8 | B9 | BA | BB | BC | BD |

RES b, (HL)

| CB— | b: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|----|----|----|----|----|----|----|----|----|
| | | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE |

THE Z80 INSTRUCTION SET

RES b, (IX + d) DDCB - b: 0 1 2 3 4 5 6 7
RES r/(HL) CB -
RES b, (IY + d) FDCB - [86 8E 96 9E A6 AE B6 BE]

Flags:

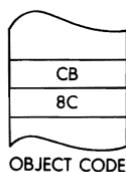
S Z H P/V N C
[] (No effect)

Examples:

RES 1, H

Before:

After:



H [42]

H [40]

PROGRAMMING THE Z80

RET

Return from subroutine

Function:

$\text{PC}_{\text{low}} \leftarrow (\text{SP})$; $\text{PC}_{\text{high}} \leftarrow (\text{SP} + 1)$; $\text{SP} \leftarrow \text{SP} + 2$

Format:

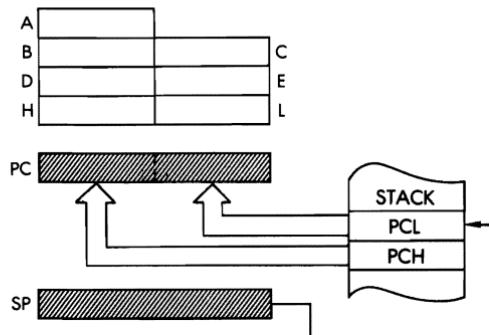
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 C9

Description:

The program counter is popped off the stack as described for the POP instructions. The next instruction fetched is from the location pointed to by PC.

Data Flow:



Timing:

3 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

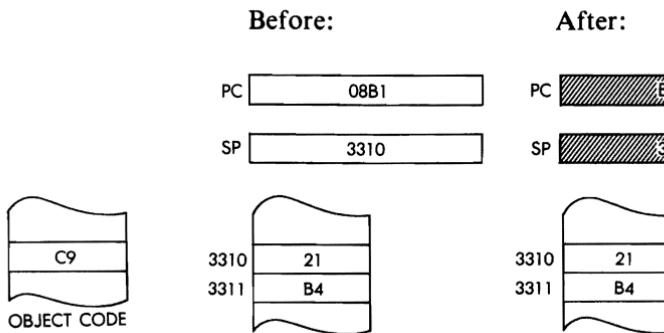
| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|

 (no effect)

THE Z80 INSTRUCTION SET

Example:

RET



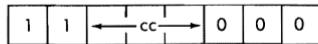
PROGRAMMING THE Z80

RET cc

Return from subroutine on condition.

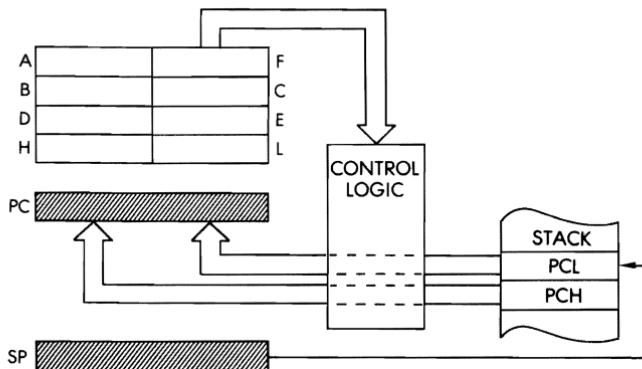
Function: If cc true: $PC_{low} \leftarrow (SP)$; $PC_{high} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Format:



Description: If the condition is met, the contents of the program counter are popped off the stack as described for the POP instructions. The next instruction is fetched from the address in PC. If the condition is not met, instruction execution continues in sequence.

Data Flow:



cc may be any one of:

| | |
|----------|----------|
| NZ - 000 | PO - 100 |
| Z - 001 | PE - 101 |
| NC - 010 | P - 110 |
| C - 011 | M - 111 |

Timing: Condition met: 3 M cycles; 11 T states; 6.5 usec @ 2 MHz.
Condition not met: 1 M cycle; 5 T states; 2.5 usec @ 2 MHz

Addressing Mode: Indirect.

THE Z80 INSTRUCTION SET

Byte Codes:

| | | | | | | | | |
|-----|----|----|----|----|----|----|----|----|
| CC: | NZ | Z | NC | C | P0 | PE | P | M |
| | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 |

Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| | | | | | |

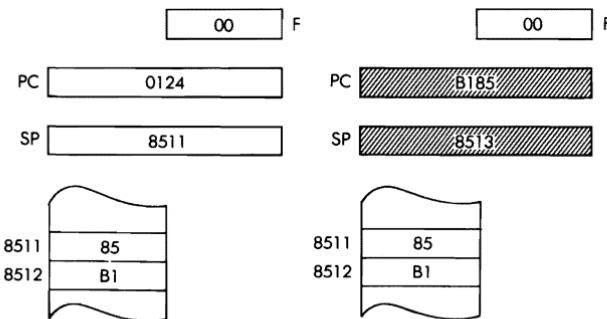
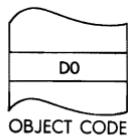
(no effect)

Example:

RET NC

Before:

After:



RETI

Return from interrupt.

Function:

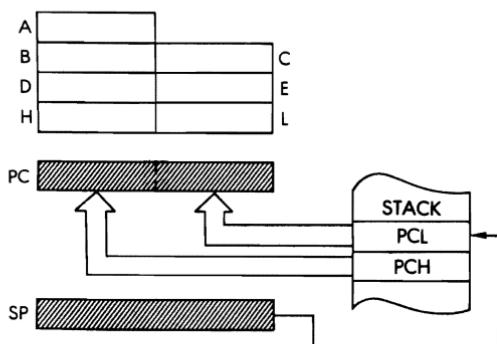
$$\text{PC}_{\text{low}} \leftarrow (\text{SP}); \text{PC}_{\text{high}} \leftarrow (\text{SP} + 1); \text{SP} \leftarrow \text{SP} + 2$$

Format:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|------------|
| <table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table> | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | byte 1: ED |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | | |
| <table border="1"> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table> | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | byte 2: 4D |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | | |

Description:

The program counter is popped off the stack as described for the POP instructions. This instruction is recognized by Zilog peripheral devices as the end of a peripheral service routine so as to allow proper control of nested priority interrupts. An EI instruction must be executed prior to RETI in order to re-enable interrupts.

Data Flow:*Timing:*

4 M cycles; 14 T states; 7 usec @ 2 MHz

Addressing Modes: *Indirect.**Flags:*

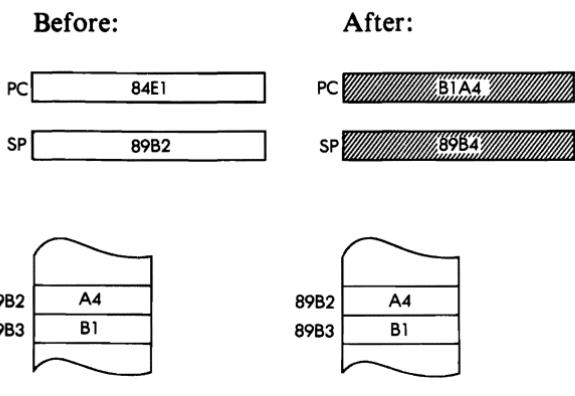
| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|
| | | | | | |

(no effect).

THE Z80 INSTRUCTION SET

Example:

RETI



RETN

Return from non-maskable interrupt.

Function: $\text{PC}_{\text{low}} \leftarrow (\text{SP}); \text{PC}_{\text{high}} \leftarrow (\text{SP} + 1); \text{SP} \leftarrow \text{SP} + 2; \text{IFF'1} \leftarrow \text{IFF2}$

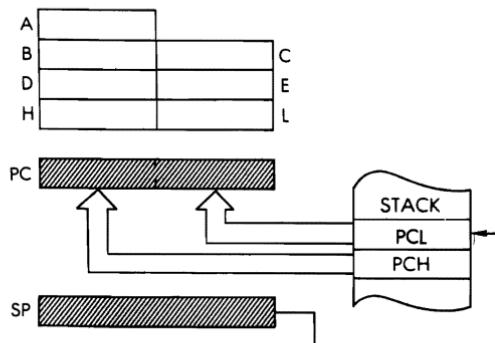
Format:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|--|------------|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | | byte 1: ED |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | | byte 2: 45 |

Description:

The program counter is popped off the stack as described for the POP instructions. Then the contents of the IFF2 (storage flip-flop) is copied back into the IFF1 to restore the state of the interrupt flag before the non-maskable interrupt.

Data Flow:

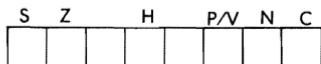


Timing: 4 M cycles; 14 T states; 7 usec @ 2 MHz

Addressing Mode: Indirect.

THE Z80 INSTRUCTION SET

Flags:

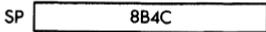
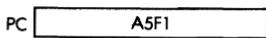


(no effect).

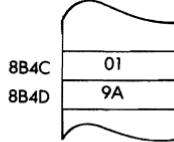
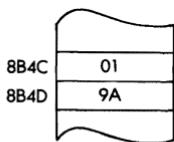
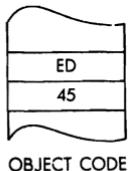
Example:

RETN

Before:

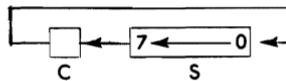


After:



RL s

Rotate left through carry operand s.

Function:*Format:**s:*

| | | | | | | | | | | |
|----------|---|---|-------|---|----------------------|---|-------|---|---|------------|
| r | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 1: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>← r →</td><td></td><td></td></tr></table> | 0 | 0 | 0 | 1 | 0 | ← r → | | | byte 2 |
| 0 | 0 | 0 | 1 | 0 | ← r → | | | | | |
| (HL) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 1: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | byte 2: 16 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | | | |
| (IX + d) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | byte 1: DD |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | | | |
| | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 2: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1"><tr><td>←</td><td>— d —</td><td>→</td></tr></table> | ← | — d — | → | byte 3: offset value | | | | | |
| ← | — d — | → | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | byte 4: 16 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | | | |
| (IY + d) | <table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | byte 1: FD |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | | | |
| | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 2: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1"><tr><td>←</td><td>— d —</td><td>→</td></tr></table> | ← | — d — | → | byte 3: offset value | | | | | |
| ← | — d — | → | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | byte 4: 16 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | | | |

r may be any one of:

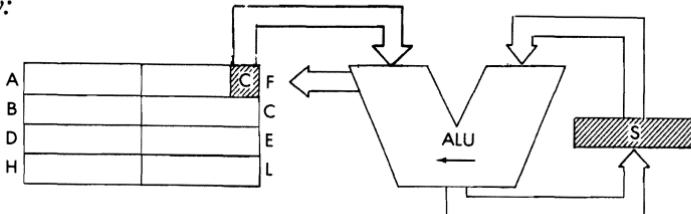
- | | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location of the specific operand are shifted left one bit place. The contents of the carry flag are moved to bit 0 and the contents of bit 7 are moved to the carry flag. The final result is stored back in the original location. s is defined in the description of the similar RLC instructions.

THE Z80 INSTRUCTION SET

Data Flow:



Timing:

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec</i> @ 2 MHz: |
|-----------|------------------|------------------|-------------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

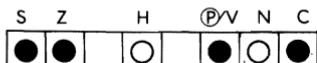
Byte Codes:

RL r

r:

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| A | B | C | D | E | H | L | |
| CB | 17 | 10 | 11 | 12 | 13 | 14 | 15 |

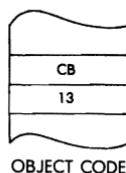
Flags:



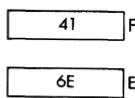
C is set by bit 7 of source.

Example:

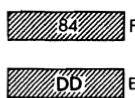
RL E



Before:



After:

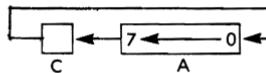


PROGRAMMING THE Z80

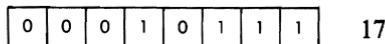
RLA

Rotate accumulator left through carry flag.

Function:



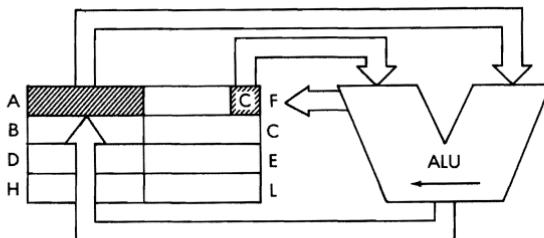
Format:



Description:

The contents of the accumulator are shifted left one bit position. The contents of the carry flag are moved into bit 0 and the original contents of bit 7 are moved into the carry flag. (9 bit rotation.)

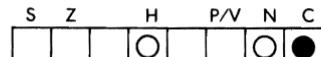
Data Flow:



Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:



C is set by bit 7 of A.

Example:

RLA

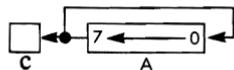
Before:

After:



RLCA

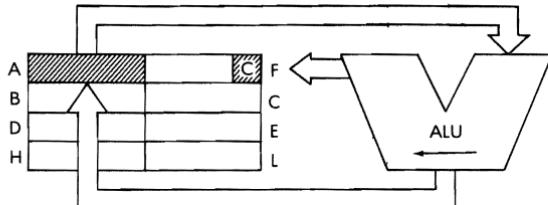
Rotate accumulator left with branch carry.

Function:*Format:*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 07 |
|---|---|---|---|---|---|---|---|----|

Description:

The contents of the accumulator are rotated left one bit position. The original contents of bit 7 is moved to the carry flag as well as to bit 0.

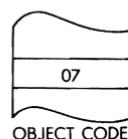
Data Flow:*Timing:* 1 M cycle; 4 T states; 2 usec @ 2 MHz*Addressing Mode:* Implicit.*Flags:*

| S | Z | H | P/V | N | C |
|---|---|---|-----|---|---|
| | | ○ | | ○ | ● |

C is set by bit 7 of A.

Example:

RLCA

*Before:*

| | | | |
|---|----|----|---|
| A | 6B | 01 | F |
|---|----|----|---|

After:

| | | | |
|---|----|----|---|
| A | D6 | 00 | F |
|---|----|----|---|

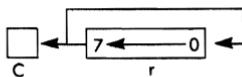
Note: This instruction is identical to RLC A, except for the flags. It is provided for compatibility with the 8080.

PROGRAMMING THE Z80

RLC r

Rotate register r left with branch carry.

Function:



Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: CB

| | | | | | |
|---|---|---|---|---|-------|
| 0 | 0 | 0 | 0 | 0 | ← r → |
|---|---|---|---|---|-------|

 byte 2

Description:

The contents of the specified register are rotated left. The original contents of bit 7 are moved to the carry flag as well as bit 0. r may be any one of:

A - 111

B - 000

C - 001

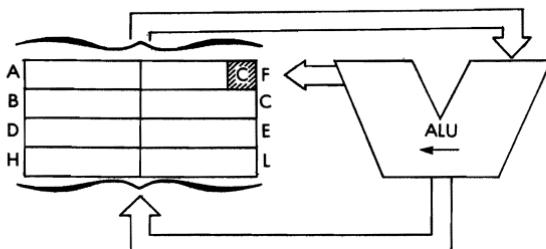
D - 010

E - 011

H - 100

L - 101

Data Flow:



Timing: 2 M cycles; 8 T states; 4 usec @ 2 MHz

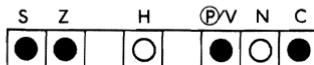
Addressing Mode: Implicit.

Byte Codes:

r: CB-

| | | | | | | |
|----|----|----|----|----|----|----|
| A | B | C | D | E | H | L |
| 07 | 00 | 01 | 02 | 03 | 04 | 05 |

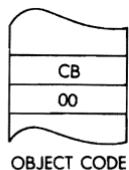
Flags:



C is set by bit 7 of source register.

Example:

RLC B



Before:

B 62 56 F

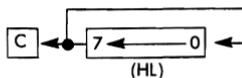
After:

B C4 80 F

RLC (HL)

Rotate left with branch carry memory location (HL).

Function:



Format:

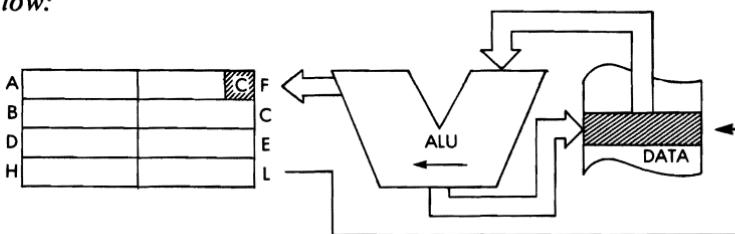
| | | | | | | | |
|------------|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| byte 1: CB | | | | | | | |

| | | | | | | | |
|------------|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| byte 2: 06 | | | | | | | |

Description:

The contents of the memory location addressed by the contents of the HL register pair are rotated left one bit position and the result is stored back at that location. The contents of bit 7 are moved to the carry flag as well as to bit 0.

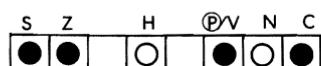
Data Flow:



Timing: 4 M cycles; 15 T states; 7.5 usec @ 2 MHz

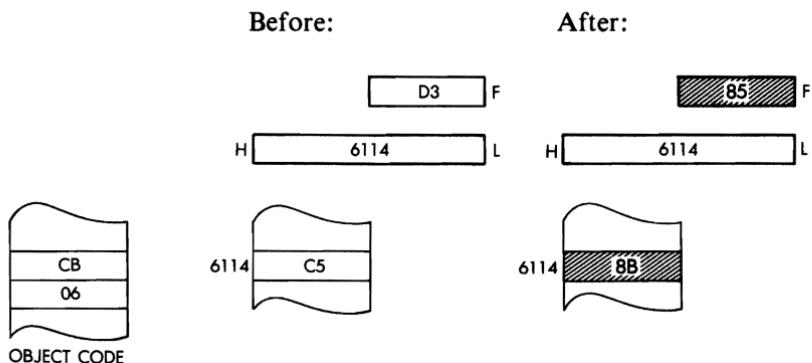
Addressing Mode: Indirect.

Flags:



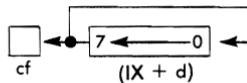
C is set by bit 7 of the memory location.

Example: RLC (HL)

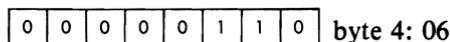
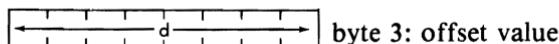
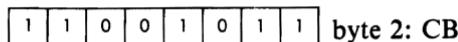
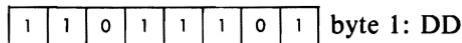


RLC (IX + d) Rotate left with branch carry memory location (IX + d)

Function:



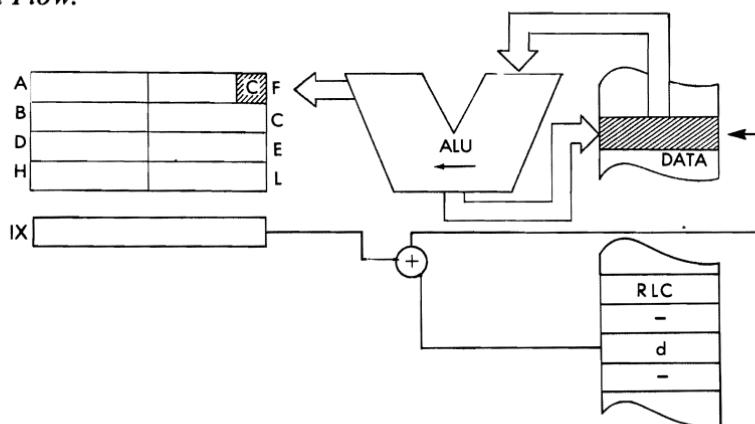
Format:



Description:

The contents of the memory location addressed by the contents of the IX register plus the given offset value are rotated left and the result is stored back at that location. The contents of bit 7 are moved to the carry flag as well as to bit 0.

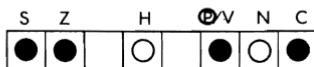
Data Flow:



Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

Addressing Mode: Indexed.

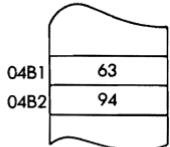
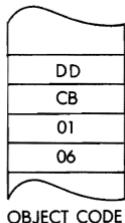
Flags:



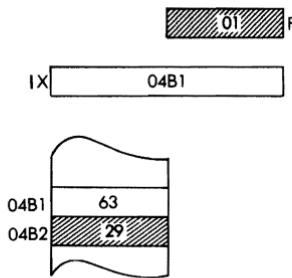
C is set by bit 7 of memory location.

Example: RLC (IX + 1)

Before:



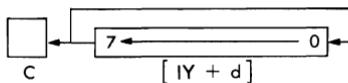
After:



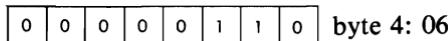
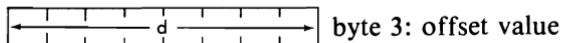
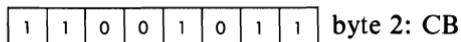
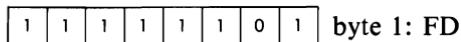
PROGRAMMING THE Z80

RLC (IY + d) Rotate left with carry memory location (IY + d).

Function:



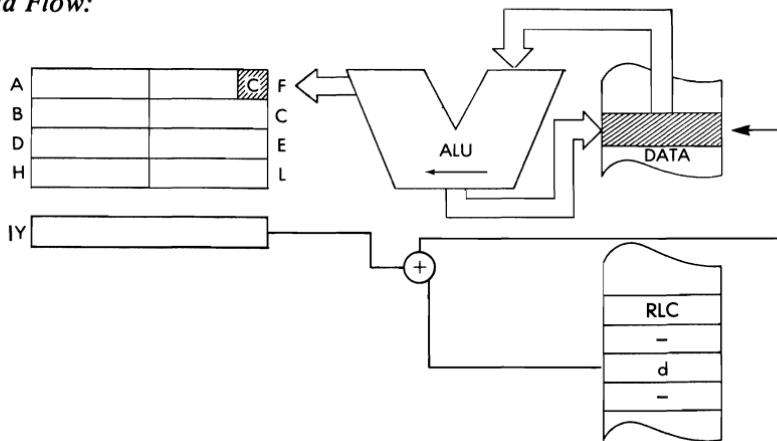
Format:



Description:

The contents of the memory location addressed by the contents of the IY register plus the given offset value are rotated left and the result is stored back at the location. The contents of bit 7 are moved to the carry flag as well as bit 0.

Data Flow:

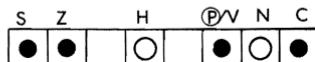


THE Z80 INSTRUCTION SET

Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

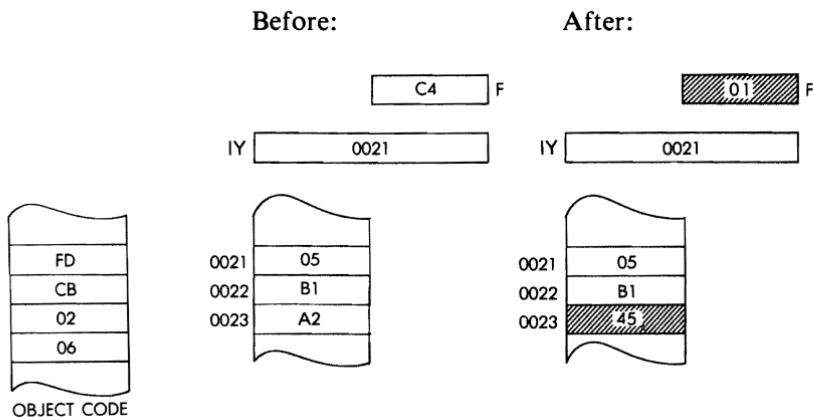
Addressing Mode: Indexed.

Flags:



C is set by bit 7 of memory location.

Example: RLC (IY + 2)



PROGRAMMING THE Z80

RLD

Rotate left decimal.

Function:



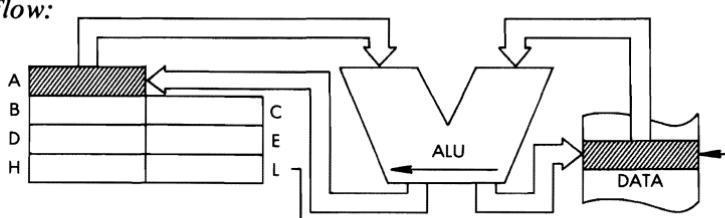
Format:

| | |
|-----------------|------------|
| 1 1 1 0 1 1 0 1 | byte 1: ED |
| 0 1 1 0 1 1 1 1 | byte 2: 6F |

Description:

The 4 low order bits of the memory location addressed by the contents of HL are moved to the high order bit positions of that same location. The 4 high order bits are moved to the 4 low order bits of the accumulator. The low order of the accumulator is moved to the 4 low order bits of the memory location originally specified. All of these operations occur simultaneously.

Data Flow:

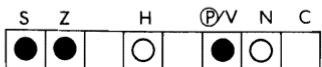


Timing: 5 M cycles; 18 T states; 9 usec @ 2 MHz

Addressing Mode: Indirect.

THE Z80 INSTRUCTION SET

Flags:



Examples:

RLD

Before:

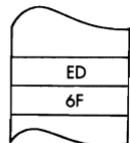
A 61

H B4F2 L

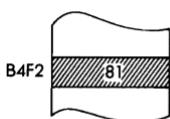
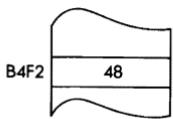
After:

A 64

H B4F2 L



OBJECT CODE

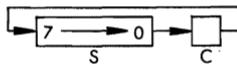


PROGRAMMING THE Z80

RR s

Rotate right s through carry.

Function:



Format:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|---|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|------------|---|---|--|---|---|---|---|---|---|---|---|---|---|---|---|--|---|---|---|---|---|---|---|--|
| r | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td><td></td><td></td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  | | byte 1: CB | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 |  |  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (HL) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td><td></td><td>0</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  | 0 | byte 2 | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 |  |  | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (IX + d) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td><td></td><td>0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td><td></td><td>0</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  | 0 | |  |  |  |  |  |  |  | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  | 0 | byte 1: CB byte 2: 1E byte 1: DD byte 2: CB byte 3: offset value |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 |  |  | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| |  |  |  |  |  |  |  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 |  |  | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (IY + d) | <table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table> | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | |  |  |  |  |  |  |  | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | byte 1: FD byte 2: CB byte 3: offset value byte 4: 1E | | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| |  |  |  |  |  |  |  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

r may be any one of:

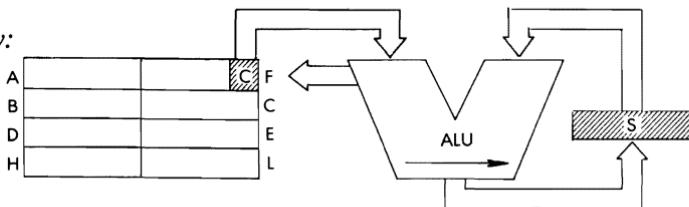
- | | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location determined by the specific operand are shifted right. The contents of the carry flag are moved to bit 7 and the contents of bit 0 are moved to the carry flag. The final result is stored back in the original location. s is defined in the description of the similar RLC instructions.

THE Z80 INSTRUCTION SET

Data Flow:



Timing:

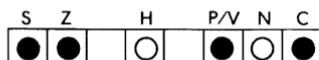
| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

| RR | r: | A | B | C | D | E | H | L |
|-----|----|----|----|----|----|----|----|---|
| CB- | 1F | 18 | 19 | 1A | 1B | 1C | 1D | |

Flags:

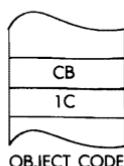


C is set by bit 0 of source data.

Example:

RR H

Before:



After:

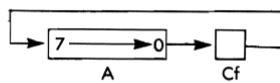


PROGRAMMING THE Z80

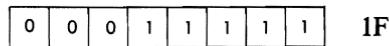
RRA

Rotate accumulator right through carry.

Function:



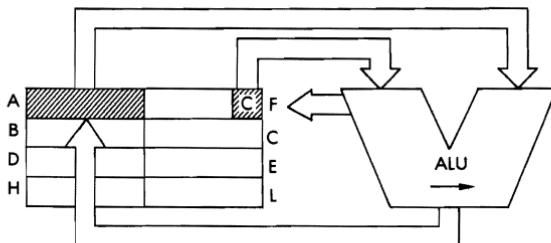
Format:



Description:

The contents of the accumulator are shifted right-one bit position. The contents of the carry flag are moved to bit 7 and the contents of bit 0 are moved to the carry flag (9-bit rotation).

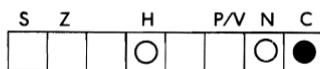
Data Flow:



Timing: 1 M cycle; 4 T states; 2 usec @ MHz

Addressing Mode: Implicit.

Flags:

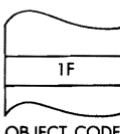


C is set by bit 0 of A.

Example:

RRA

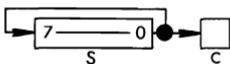
Before:



Note: This instruction is almost identical to RRA. It is provided for 8080 compatibility.

RRC s

Rotate right with branch carry s.

Function:*Format:*

s: s is any of r, (HL), (IX + d), (IY + d).

| | | | | | | | | | | |
|----------|--|-------|---|---|-------|---|-------|---|--------|----------------------|
| r | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 1: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="2" style="text-align: center;">← r →</td></tr></table> | 0 | 0 | 0 | 0 | 1 | ← r → | | byte 2 | |
| 0 | 0 | 0 | 0 | 1 | ← r → | | | | | |
| (HL) | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 1: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | byte 2: 0E |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | | | |
| (IX + d) | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | byte 1: DD |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 2: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td colspan="4" style="text-align: center;">← d →</td><td></td><td></td><td></td><td></td></tr></table> | ← d → | | | | | | | | byte 3: offset value |
| ← d → | | | | | | | | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | byte 4: 0E |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | | | |
| (IY + d) | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | byte 1: FD |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 2: CB |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td colspan="4" style="text-align: center;">← d →</td><td></td><td></td><td></td><td></td></tr></table> | ← d → | | | | | | | | byte 3: offset value |
| ← d → | | | | | | | | | | |
| | <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | byte 4: 0E |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | | | |

r may be any one of:

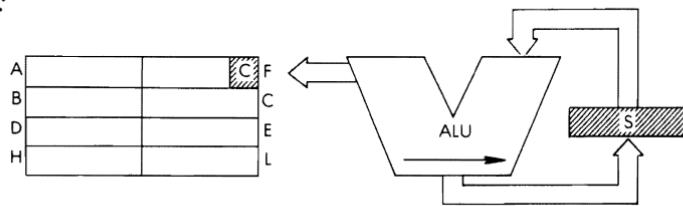
- | | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location determined by the specified operand are rotated right and the result is stored back in the original location. The contents of bit 0 are moved to the carry flag as well as to bit 7. s is defined in the description of the similar RLC instructions.

PROGRAMMING THE Z80

Data Flow:



Timing:

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte codes:

| RRC | r | r: | A | B | C | D | E | H | L |
|-----|---|-----|----|----|----|----|----|----|----|
| | | CB- | OF | 08 | 09 | 0A | 0B | 0C | 0D |

Flags:

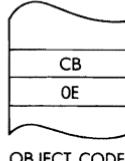


Example:

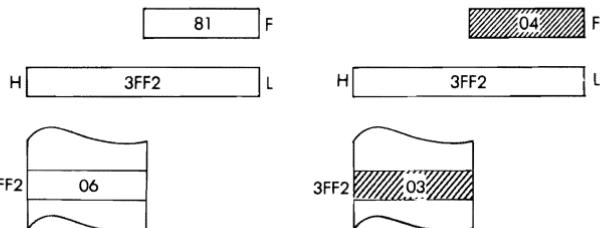
RRC (HL)

Before:

After:



OBJECT CODE



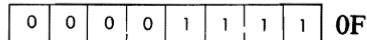
RRCA

Rotate accumulator right with branch carry.

Function:



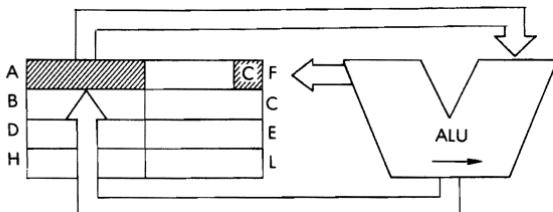
Format:



Description:

The contents of the accumulator are rotated right one bit position. The contents of bit 0 are moved to the carry flag as well as to bit 7.

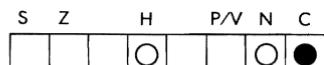
Data Flow:



Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:



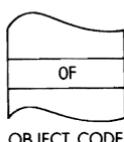
C is set by bit 0 of A.

Example:

RRCA

Before:

After:

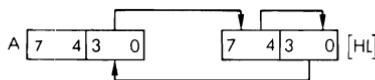


PROGRAMMING THE Z80

RRD

Rotate right decimal.

Function:



Format:

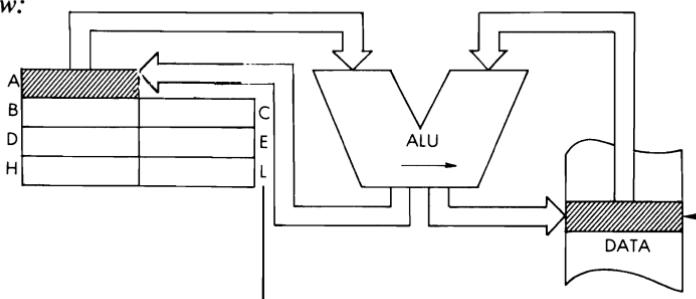
| | | | | | | | | |
|---|---|---|---|---|---|---|---|--|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |

byte 1: ED
byte 2: 67

Description:

The 4 high order bits of the memory location addressed by the contents of the HL register pair are moved to the low order 4 bits of that location. The 4 low order bits are moved to the 4 low order bits of the accumulator. The low order bits of the accumulator are moved to the 4 high order bit positions of the memory location originally specified. All of the above operations occur simultaneously.

Data Flow:

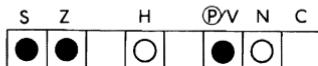


Timing: 5 M cycles; 18 T states; 9 usec @ 2 MHz

Addressing Mode: Indirect.

THE Z80 INSTRUCTION SET

Flags:



Example:

RRD

Before:

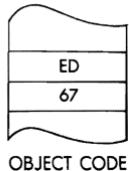
A 92

H FEB1 L

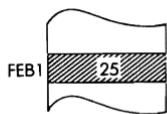
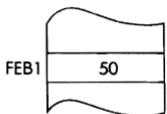
After:

A 90

H FEB1 L



OBJECT CODE



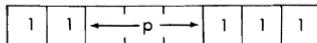
PROGRAMMING THE Z80

RST p

Restart at p.

Function: $(SP - 1) \leftarrow PC_{high}; (SP - 2) \leftarrow PC_{low}; SP \leftarrow SP - 2; PC_{high} \leftarrow 0; PC_{low} \leftarrow p$

Format:



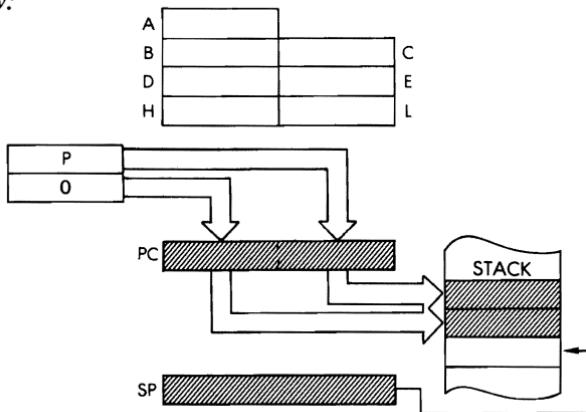
Description:

The contents of the program counter are pushed onto the stack as described for the PUSH instructions. The specified value for p is then loaded into the PC and the next instruction is fetched from this new address. p may be any one of:

| | |
|-----------|-----------|
| 00H – 000 | 20H – 100 |
| 08H – 001 | 28H – 101 |
| 10H – 010 | 30H – 110 |
| 18H – 011 | 38H – 111 |

This instruction performs a jump to any of eight starting addresses in low memory and requires only a single byte. It may be used as a fast response to an interrupt.

Data Flow:



THE Z80 INSTRUCTION SET

Timing: 3 M cycles; 11 T states; 5.5 usec @ 2 MHz

Addressing Mode: Indirect.

Byte Codes: p:

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 |
| C7 | CF | D7 | DF | E7 | EF | F7 | FF |

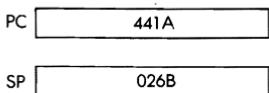
Flags:

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| S | Z | H | P/V | N | C |
| [] | [] | [] | [] | [] | [] |

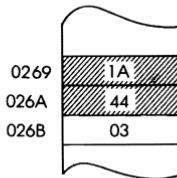
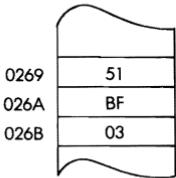
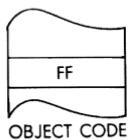
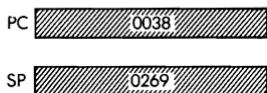
 (no effect).

Example: RST 38H

Before:



After:



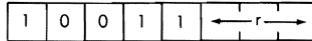
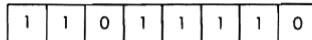
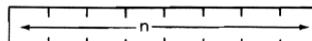
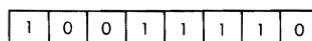
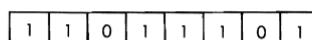
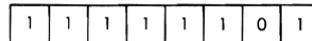
PROGRAMMING THE Z80

SBC A, s

Subtract with borrow accumulator and specified operand.

Function: $A \leftarrow A - s - C$

Format: $s:$ may be $r, n, (HL), (IX + d)$, or $(IY + d)$

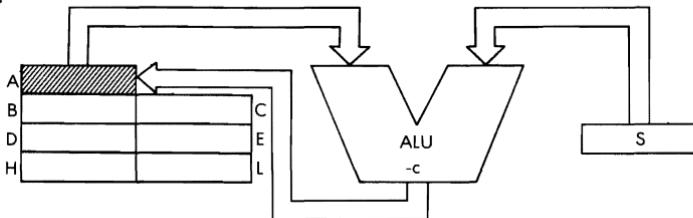
| | | |
|------------|---|--|
| r |  1 0 0 1 1 ← r → | |
| n |  1 1 0 1 1 1 1 0 ← n → | byte 1: DE |
| |  ← → | byte 2: immediate data |
| (HL) |  1 0 0 1 1 1 1 0 | byte 1: 9E |
| $(IX + d)$ |  1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 0 ← d → | byte 1: DD 9E byte 3: offset value |
| $(IY + d)$ |  1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0 ← d → | byte 1: FD byte 2: 9E byte 3: offset value |

r may be any one of:

| | |
|-----------|-----------|
| $A - 111$ | $E - 011$ |
| $B - 000$ | $H - 100$ |
| $C - 001$ | $L - 101$ |
| $D - 010$ | |

Description:

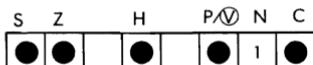
The specified operand s , summed with the contents of the carry flag, is subtracted from the contents of the accumulator, and the result is placed in the accumulator. s is defined in the description of the similar ADD instructions.

Data Flow:*Timing:*

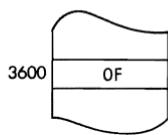
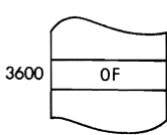
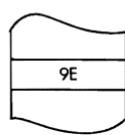
| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 1 | 4 | 2 |
| n | 2 | 7 | 3.5 |
| (HL) | 2 | 7 | 3.5 |
| (IX + d) | 5 | 19 | 9.5 |
| (IY + d) | 5 | 19 | 9.5 |

Addressing Mode: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed.*Byte Codes:*

| SBC A, r | r: A B C D E H L |
|----------|----------------------|
| | 9F 98 99 9A 9B 9C 9D |

Flags:*Example:*

SBC A, (HL)

Before:*After:*

PROGRAMMING THE Z80

SBC HL, ss Subtract with borrow HL and register pair ss.

Function: $HL \leftarrow HL - ss - C$

Format:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 byte 1: ED

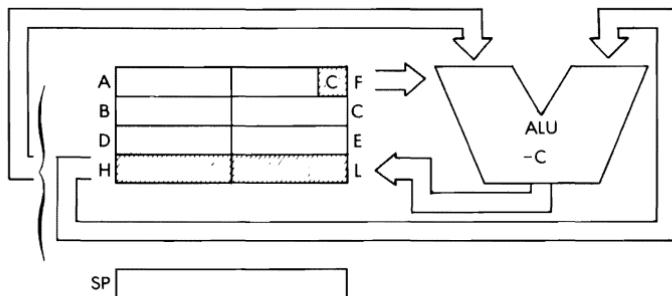
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | S | ' | S | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|

 byte 2

Description: The contents of the specified register pair plus the contents of the carry flag are subtracted from the contents of the HL register pair and the result is stored back in HL. ss may be any one of:

| | |
|---------|---------|
| BC - 00 | HL - 10 |
| DE - 01 | SP - 11 |

Data Flow:



Timing: 4 M cycles; 15 T states; 7.5 usec @ 2 MHz

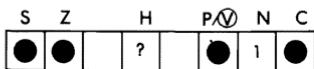
Addressing Mode: Implicit.

Byte Codes: SS: BC DE HL SP
ED ->

| | | | |
|----|----|----|----|
| 42 | 52 | 62 | 72 |
|----|----|----|----|

THE Z80 INSTRUCTION SET

Flags:



H is set if borrow from bit 12.

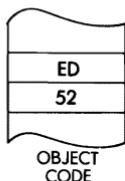
C is set if borrow.

Example:

SBC HL, DE

Before:

After:



66 F

06 F

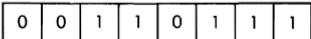
D 06B9 E
H 3142 L

D 06B9 E
H 2A69 L

PROGRAMMING THE Z80

SCF Set carry flag.

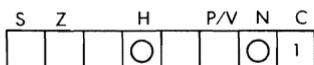
Function: $C \leftarrow 1$

Format:  37

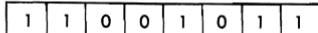
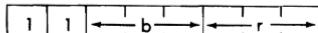
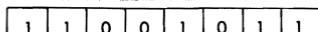
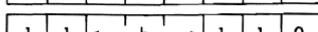
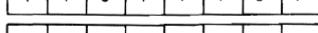
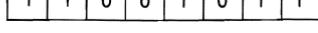
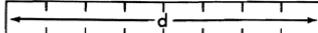
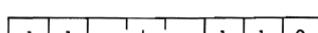
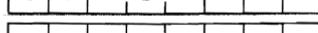
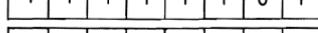
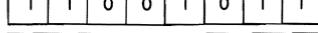
Description: The carry flag is set.

Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags: 

SET b, s Set bit b of operand s*Function:* $s_b \leftarrow 1$ *Format:* s:

| | | |
|----------|---|----------------------|
| r |  | byte 1: CB |
| |  | byte 2 |
| (HL) |  | byte 1: CB |
| |  | byte 2 |
| (IX + d) |  | byte 1: DD |
| |  | byte 2: CB |
| |  | byte 3: offset value |
| |  | byte 4 |
| (IY + d) |  | byte 1: FD |
| |  | byte 2: CB |
| |  | byte 3: offset value |
| |  | byte 4 |

r may be any one of:

| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

b may be any one of:

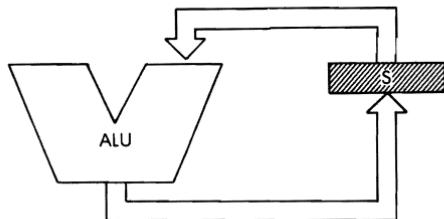
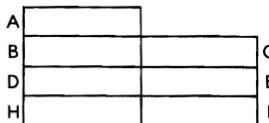
| | |
|---------|---------|
| 0 - 000 | 4 - 100 |
| 1 - 001 | 5 - 101 |
| 2 - 010 | 6 - 110 |
| 3 - 011 | 7 - 111 |

Description:

The specified bit of the location determined by s is set. s is defined in the description of the similar BIT instructions.

PROGRAMMING THE Z80

Data Flow:



Timing:

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|--------------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes: SET b, r

| | | b: r: A B C D E H L | | | | | | | |
|-----|---|---------------------|----|----|----|----|----|----|--|
| CB- | 0 | C7 | C0 | C1 | C2 | C3 | C4 | C5 | |
| | 1 | CF | C8 | C9 | CA | CB | CC | CD | |
| | 2 | D7 | D0 | D1 | D2 | D3 | D4 | D5 | |
| | 3 | DF | D8 | D9 | DA | DB | DC | DD | |
| | 4 | E7 | E0 | E1 | E2 | E3 | E4 | E5 | |
| | 5 | EF | E8 | E9 | EA | EB | EC | ED | |
| | 6 | F7 | F0 | F1 | F2 | F3 | F4 | F5 | |
| | 7 | FF | F8 | F9 | FA | FB | FC | FD | |

SET b, (HL)

SET b, (IX + d)

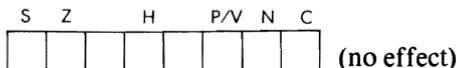
SET b, (IY + d)

}

| b: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|----|----|----|----|----|----|----|----|
| | C6 | CE | D6 | DE | E6 | EE | F6 | FE |

THE Z80 INSTRUCTION SET

Flags:

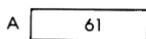
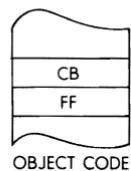


Example:

SET 7, A

Before:

After:

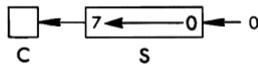


PROGRAMMING THE Z80

SLA s

Arithmetic shift left operand s.

Function:



Format:

s:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|---|---|---|---|---|---|---|---|----------------------|---|---|---|---|---|---|---|------------|--|--|--|--|--|--|--|------------|
| r | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>r</td><td></td><td></td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | r | | | byte 1: CB | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 0 | r | | | | | | | | | | | | | | | | | | | | | |
| (HL) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | byte 1: CB | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | |
| (IX + d) | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>d</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | d | | | | | | | | byte 1: DD |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | |
| d | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | byte 2: CB | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | byte 3: offset value | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | byte 4: 26 | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | |
| (IY + d) | <table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>d</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | d | | | | | | | | byte 1: FD |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | | | | |
| d | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | byte 2: CB | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | byte 3: offset value | | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | |
| | <table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table> | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | byte 4: 26 | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | |

r may be any one of:

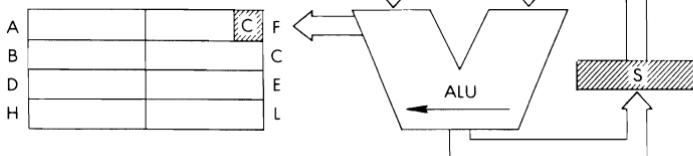
| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location determined by the specific operand are arithmetically shifted left with the contents of bit 7 being moved to the carry flag and a 0 being forced into bit 0. The final result is stored back in the original location. s is defined in the description of the similar RLC instructions.

THE Z80 INSTRUCTION SET

Data Flow:



Timing:

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

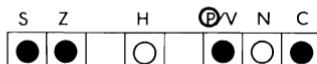
Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

SLA r

| r: | A | B | C | D | E | H | L |
|----|----|----|----|----|----|----|----|
| CB | 27 | 20 | 21 | 22 | 23 | 24 | 25 |

Flags:

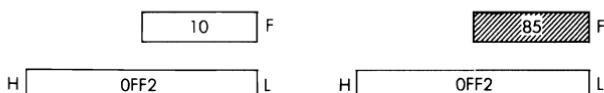


C is set by bit 7 of source data.

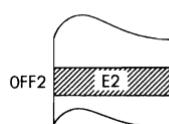
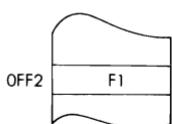
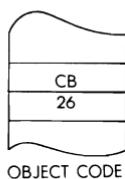
Example:

SLA (HL)

Before:



After:

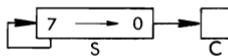


PROGRAMMING THE Z80

SRA s

Shift right arithmetic s.

Function:



Format:

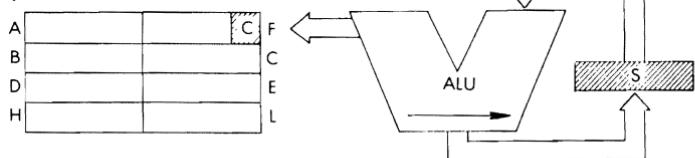
| s: | r | | |
|----------|-----------------|--|----------------------|
| | 1 1 0 0 1 0 1 1 | | byte 1: CB |
| | 0 0 1 0 1 ← r → | | byte 2 |
| (HL) | 1 1 0 0 1 0 1 1 | | byte 1: CB |
| | 0 0 1 0 1 1 1 0 | | byte 2: 2E |
| (IX + d) | 1 1 0 1 1 1 0 1 | | byte 1: DD |
| | 1 1 0 0 1 0 1 1 | | byte 2: CB |
| | ← d → | | byte 3: offset value |
| | 0 0 1 0 1 1 1 0 | | byte 4: 2E |
| (IY + d) | 1 1 1 1 1 1 0 1 | | byte 1: FD |
| | 1 1 0 0 1 0 1 1 | | byte 2: CB |
| | ← d → | | byte 3: offset value |
| | 0 0 1 0 1 1 1 0 | | byte 4: 2E |

r may be any one of:

- | | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location determined by the specific operand are arithmetically shifted right. The contents of bit 0 are moved to the carry flag and the contents of bit 7 remain unchanged. The final result is stored at the original location. s is defined in the description of the similar RLC instructions.

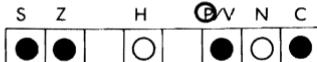
Data Flow:*Timing:*

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|--------------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

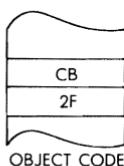
SRA r r: A B C D E H L
CB- [2F] [28] [29] [2A] [2B] [2C] [2D]

Flags:

C is set by bit 0 of source data.

Example:

SRA A

*Before:*

A [8B] [04] F

After:

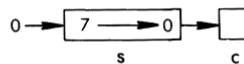
A [C5] [85] F

PROGRAMMING THE Z80

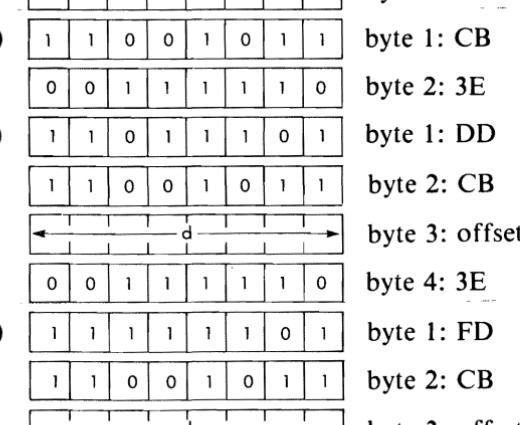
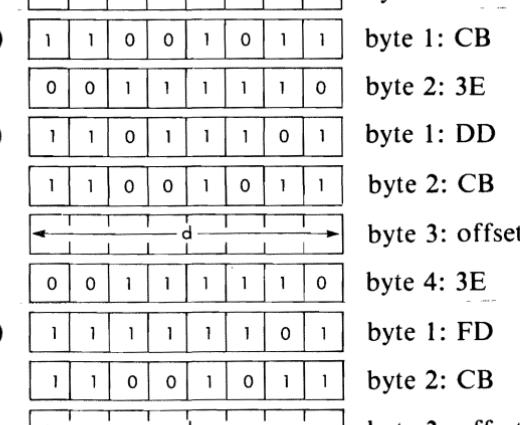
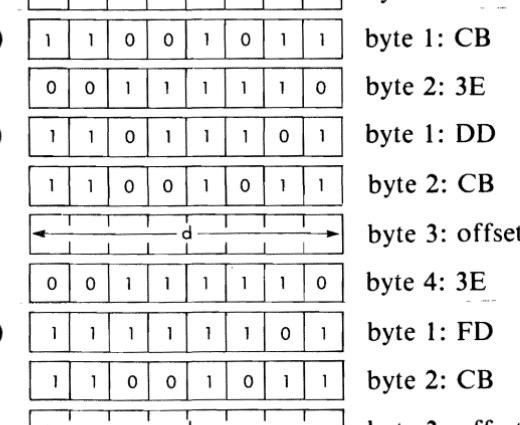
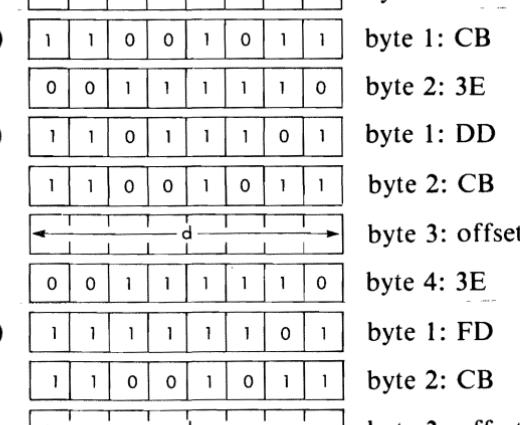
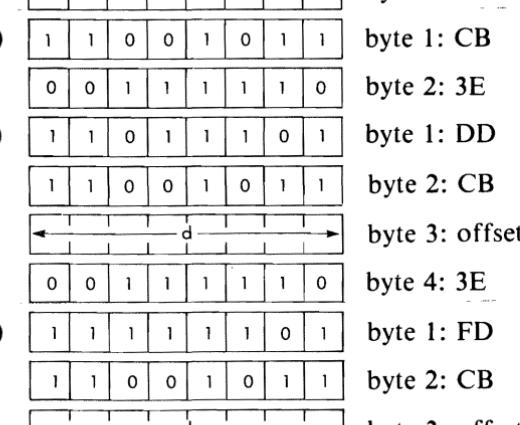
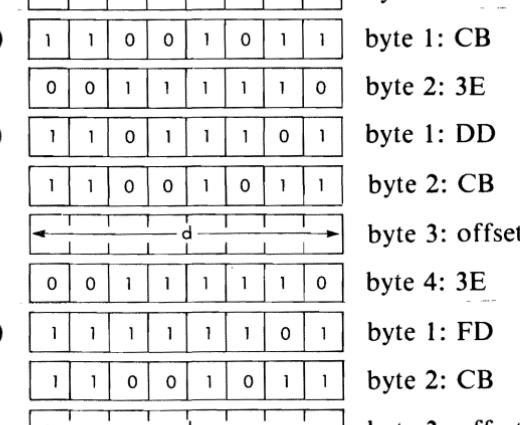
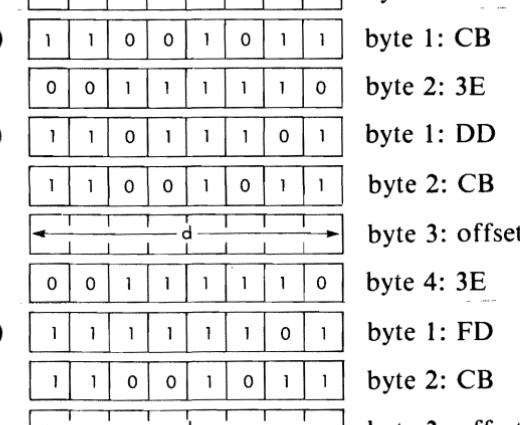
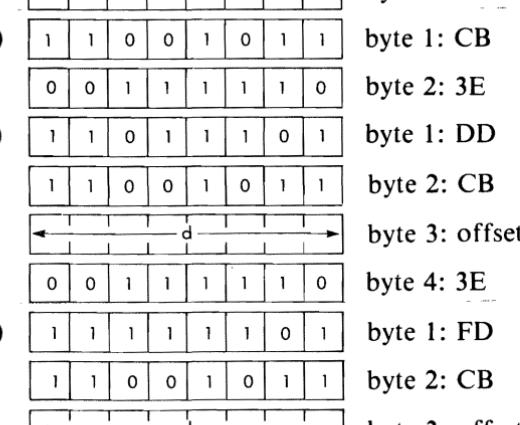
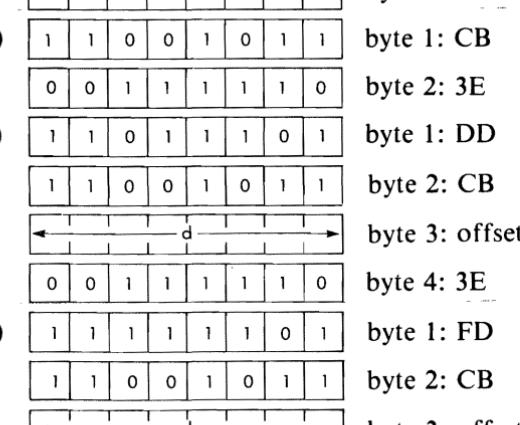
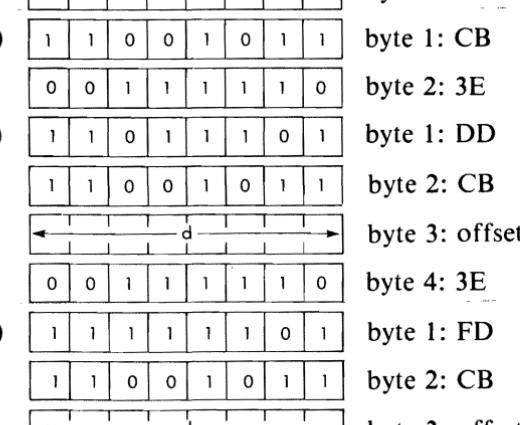
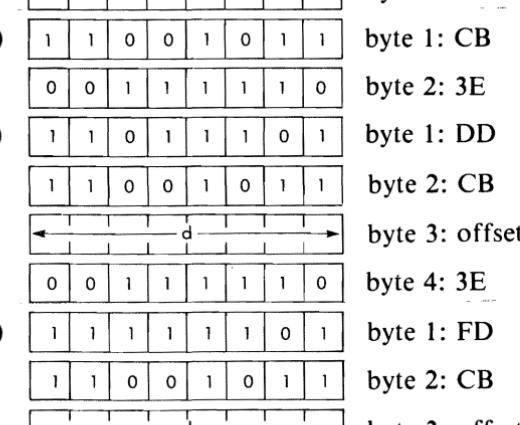
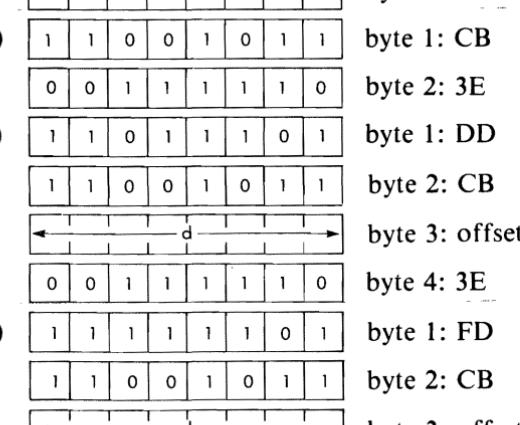
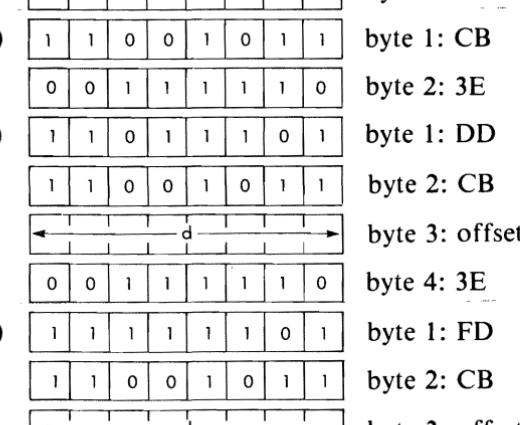
SRL S

Logical shift right s.

Function:



Format:

| | |
|-----------------|---|
| S: |  |
| r |  |
| | byte 1: CB |
| |  |
| | byte 2: 3E |
| (HL) |  |
| | byte 1: CB |
| |  |
| | byte 2: CB |
| (IX + d) |  |
| | byte 1: DD |
| |  |
| | byte 2: 3E |
| |  |
| | byte 3: offset value |
| |  |
| | byte 4: 3E |
| (IY + d) |  |
| | byte 1: FD |
| |  |
| | byte 2: CB |
| |  |
| | byte 3: offset value |
| |  |
| | byte 4: 3E |

r may be any one of:

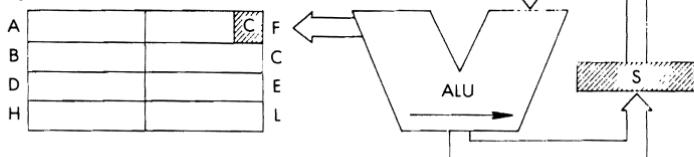
| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The contents of the location determined by the specific operand are logically shifted right. A zero is moved into bit 7 and the contents of bit 0 are moved into the carry flag. The final result is stored back in the original location.

THE Z80 INSTRUCTION SET

Data Flow:



Timing:

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|--------------------------|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7.5 |
| (IX + d) | 6 | 23 | 11.5 |
| (IY + d) | 6 | 23 | 11.5 |

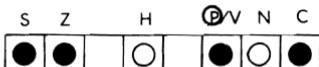
Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

SRL r

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| r | A | B | C | D | E | H | L |
| CB | 3F | 38 | 39 | 3A | 3B | 3C | 3D |

Flags:



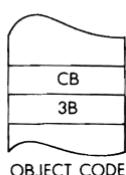
C is set by bit 0 of source data.

Example:

SRL E

Before:

After:



| | |
|----|---|
| 01 | F |
|----|---|

| | |
|----|---|
| 00 | F |
|----|---|

| | |
|----|---|
| 02 | E |
|----|---|

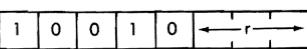
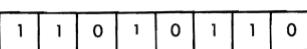
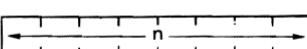
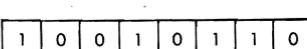
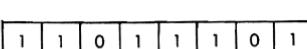
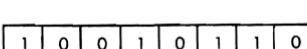
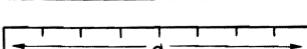
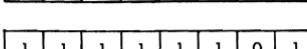
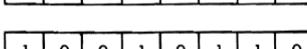
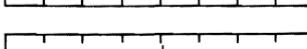
| | |
|----|---|
| 01 | E |
|----|---|

PROGRAMMING THE Z80

SUB s Subtract operand s from accumulator.

Function: $A \leftarrow A - s$

Format: s: may be r, n, (HL), (IX + d) or (IY + d)

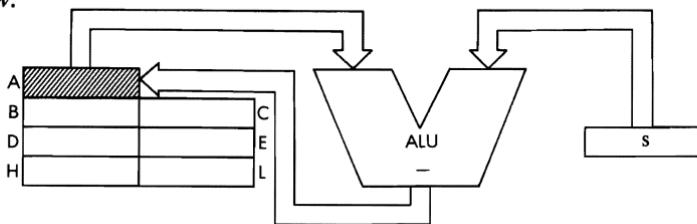
| | | |
|----------|--|------------------------|
| r |  | |
| n |  | byte 1: D6 |
| |  | byte 2: immediate data |
| (HL) |  | 96 |
| (IX + d) |  | byte 1: DD |
| |  | byte 2: 96 |
| |  | byte 3: offset value |
| (IY + d) |  | byte 1: FD |
| |  | byte 2: 96 |
| |  | byte 3: offset value |

r may be any one of:

| | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The specified operand s is subtracted from the accumulator and the result is stored in the accumulator. The operand s is defined in the description of the similar ADD instructions.

Data Flow:**Timing:**

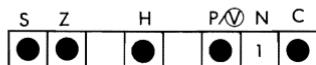
| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec</i> @ 2 MHz |
|-----------|------------------|------------------|------------------------|
| r | 1 | 4 | 2 |
| n | 2 | 7 | 3.5 |
| (HL) | 2 | 7 | 3.5 |
| (IX + d) | 5 | 19 | 9.5 |
| (IY + d) | 5 | 19 | 9.5 |

Addressing Mode: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed

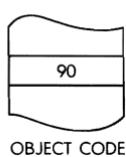
Byte Codes:

SUB r

| <i>r:</i> | A | B | C | D | E | H | L |
|-----------|----|----|----|----|----|----|----|
| | 97 | 90 | 91 | 92 | 93 | 94 | 95 |

Flags:**Example:**

SUB B

**Before:**

| | |
|---|----|
| A | 80 |
| B | 31 |

After:

| | |
|---|----|
| A | 4F |
| B | 31 |

PROGRAMMING THE Z80

XOR s

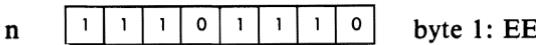
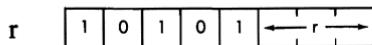
Exclusive or accumulator and s.

Function:

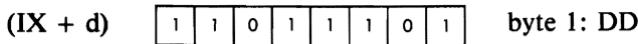
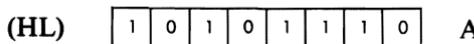
$$A \leftarrow A \text{ XOR } s$$

Format:

s: may be r, n, (HL), (IX + d), or (IY + d)

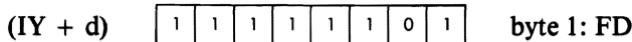


byte 2: immediate data



byte 2: AE

byte 3: offset value



byte 2: AE

byte 3: offset value

r may be any one of:

A - 111 E - 011

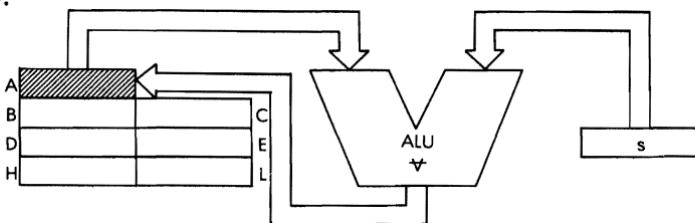
B - 000 H - 100

C - 001 L - 101

D - 010

Description:

The accumulator and the specified operand s are exclusive 'or'ed, and the result is stored in the accumulator. s is defined in the description of the similar ADD instructions.

Date Flow:*Timing:*

| <i>s:</i> | <i>M cycles:</i> | <i>T states:</i> | <i>usec @ 2 MHz:</i> |
|-----------|------------------|------------------|----------------------|
| r | 1 | 4 | 2 |
| n | 2 | 7 | 3.5 |
| (HL) | 2 | 7 | 3.5 |
| (IX + d) | 5 | 19 | 9.5 |
| (IY + d) | 5 | 19 | 9.5 |

Addressing Modes: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed

Byte Codes:

XOR r

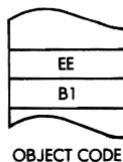
| | | | | | | |
|----|----|----|----|----|----|----|
| A | B | C | D | E | H | L |
| AF | A8 | A9 | AA | AB | AC | AD |

Flags:

| | | | | | |
|---|---|---|-----|---|---|
| S | Z | H | P/V | N | C |
| ● | ● | ○ | ● | ○ | ○ |

Example:

XOR B1H



Before:

A [36]

After:

A [87]

5

ADDRESSING TECHNIQUES

INTRODUCTION

This chapter will present the general theory of addressing and the various techniques which have been developed to facilitate the retrieval of data. In a second section, the specific addressing modes available in the Z80 will be reviewed, along with their advantages and limitations. Finally, in order to familiarize the reader with the various trade-offs possible, an applications section will demonstrate possible trade-offs between the various addressing techniques by studying specific application programs.

Because the Z80 has several 16-bit registers, in addition to the program counter, which can be used to specify an address, it is important that the Z80 user understand the various addressing modes, and in particular, the use of the index registers. Complex retrieval modes may be omitted at the beginning stage. However, all the addressing modes are useful in developing programs for this microprocessor. Let us now study the various alternatives available.

POSSIBLE ADDRESSING MODES

Addressing refers to the specification, within an instruction, of the location of the operand on which the instruction will operate. The main addressing methods will now be examined. They are all illustrated in Figure 5.1.

Implicit Addressing (or "Implied," or "Register")

Instructions which operate exclusively on registers normally use *implicit addressing*. This is illustrated in Figure 5.1. An implicit instruc-

tion derives its name from the fact that it does not specifically contain the address of the operand on which it operates. Instead, its opcode specifies one or more registers, usually the accumulator, or else any other register(s). Since internal registers are usually few in number (commonly eight), this will require a small number of bits. As an example, three bits within the instruction will point to one out of eight internal registers. Such instructions can, therefore, normally be encoded within eight bits. This is an important advantage, since an eight-bit instruction normally executes faster than any two- or three-byte instruction.

An example of an implicit instruction is:

LD A, B

which specifies “transfer the contents of B into A” (Load A from B.)

Immediate Addressing

Immediate addressing is illustrated in Figure 5.1. The eight-bit opcode is followed by an 8- or 16-bit literal (a constant). This type of instruction is needed, for example, to load an eight-bit value in an eight-bit register. Since the microprocessor is equipped with 16-bit registers, it may also be necessary to load 16-bit literals. An example of an immediate instruction is:

ADD A, 0H

The second word of this instruction contains the literal “0”, which is added to the accumulator.

Absolute Addressing

Absolute addressing usually refers to the way in which data is retrieved from or placed in memory, in which an opcode is followed by a 16-bit address. Absolute addressing, therefore, requires three-byte instructions. An example of absolute addressing is:

LD (1234H), A

It specifies that the contents of the accumulator are to be stored at memory location “1234” hexadecimal.

The disadvantage of absolute addressing is to require a three-byte instruction. In order to improve the efficiency of the microprocessor, another addressing mode may be made available, whereby only one word is used for the address: direct addressing.

PROGRAMMING THE Z80

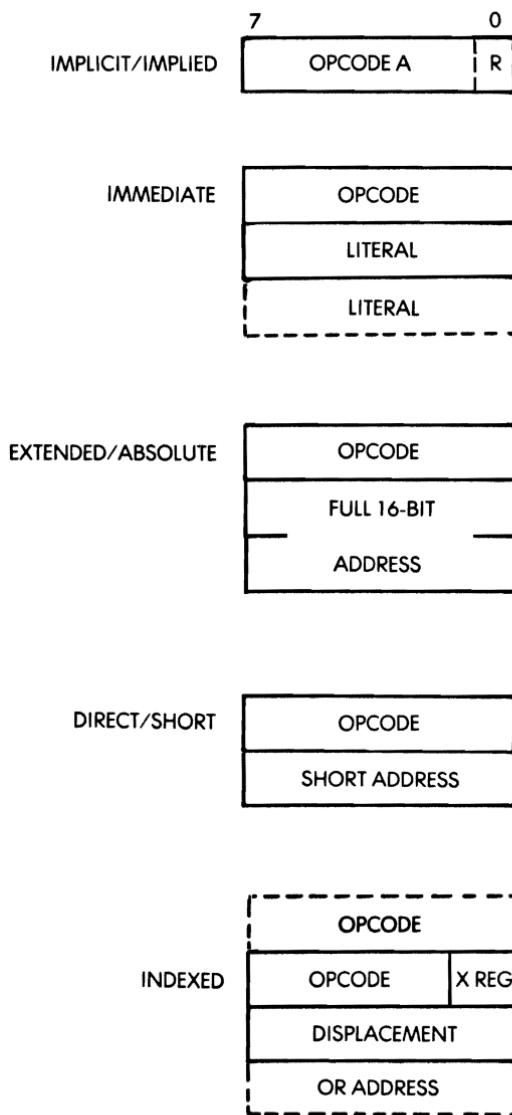


Fig. 5.1: Basic Addressing Modes

Direct Addressing (or “Short,” or “Relative”)

In this addressing mode, the opcode is followed by an eight-bit address. This is also illustrated in Figure 5.1. The advantage of this approach is to require only two bytes instead of three for absolute addressing. The disadvantage is to limit all addressing within this mode to addresses 0 to 255 or else -128 to +127. When using 0 to 255 (“page zero”), this is also called short addressing, or 0-page addressing. Whenever short addressing is available, absolute addressing is often called *extended addressing* by contrast. The range -128 to +127 is used with branch instructions. This is called relative addressing.

Relative Addressing

Normal jump or branch instructions require eight bits for the opcode, plus the 16-bit address to which the program has to jump. Just as in the preceding example, this mode has the disadvantage of requiring three words, i.e., three memory cycles. To provide more efficient branching, *relative addressing* uses only a two-word format. The first word is the branch specification, usually along with the test it is implementing. The second word is a displacement. Since the displacement must be positive or negative, a relative branching instruction allows a branch forward to 127 locations (seven-bits) or a branch backwards to 128 locations (usually +129 or -126, since PC will have been incremented by 2). Because most loops tend to be short, relative branching can be used most of the time and results in significantly improved performance for such short routines. As an example, we have already used the instruction JR NC, which specifies a “jump if no carry” to a location within 127 words of the branch instruction (more precisely +129 to -126).

The two advantages of relative addressing are improved performance (fewer bytes used) and program relocatability (independence from absolute addresses).

Indexed Addressing

Indexed addressing is a technique used to access the elements of a block or of a table successively. This will be illustrated by examples later in this chapter. The principle of indexed addressing is that the instruction specifies both an index register and an address. The contents of the register are added to the address to provide the final address. In this way, the address could be the beginning of a table in the memory.

The index register would then be used to access all the elements of a table successively in an efficient way. (This requires the availability of increment/decrement instructions for the index register). In practice, restrictions often exist which may limit the size of the index register, or the size of the address or displacement field.

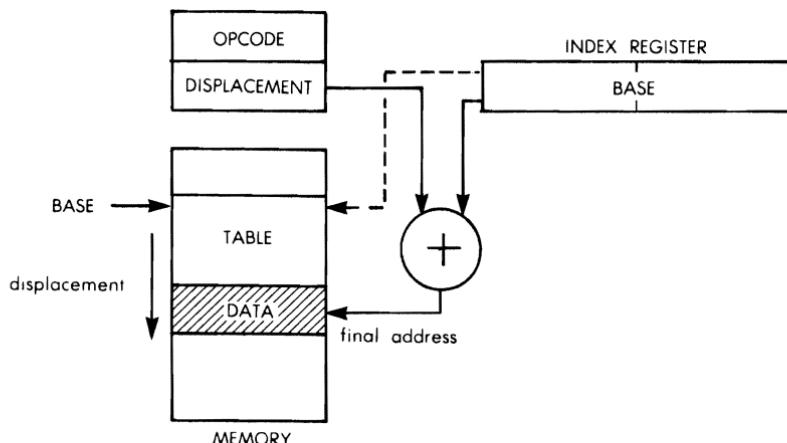


Fig. 5.2: Addressing (Pre-indexing)

Pre-Indexing and Post-Indexing

Two modes of indexing may be distinguished. Pre-indexing is the usual indexing mode in which the final address is the sum of a displacement or address and of the contents of the index register. It is shown in Figure 5.2, assuming an 8-bit displacement field and a 16-bit index register.

Post-indexing treats the contents of the displacement field like the *address* of the actual displacement, rather than the displacement itself. This is illustrated in Figure 5.3. In post-indexing, the final address is the sum of the contents of the index register plus the contents of the memory word *designated by the displacement field*. This feature utilizes, in fact, a combination of indirect addressing and pre-indexing. But we have not defined indirect addressing yet. Let us do that.

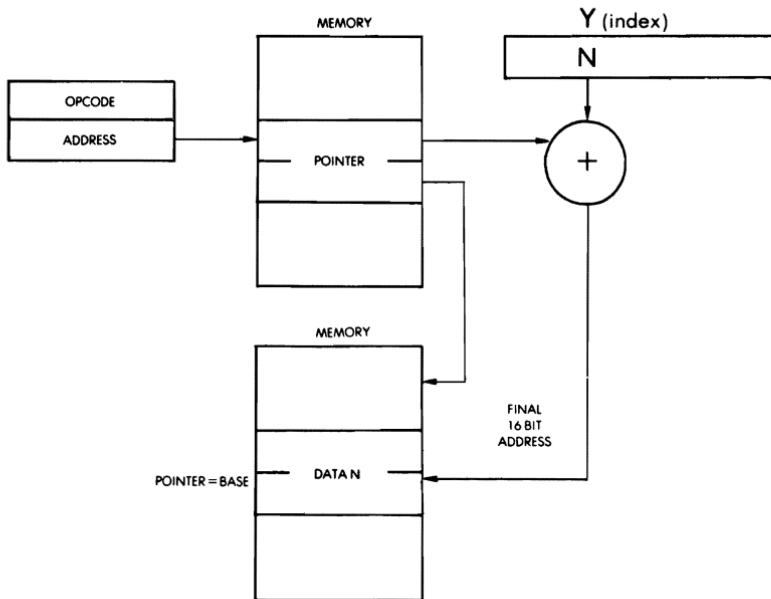


Fig. 5.3: Indirect Indexed Addressing (Post-Indexing)

Indirect Addressing

We have already seen that two subroutines may wish to exchange a large quantity of data stored in the memory. More generally, several programs, or several subroutines, may need to access a common block of information. To preserve the generality of the program, it is desirable not to keep such a block at a fixed memory location. In particular, the size of this block might grow or shrink dynamically, and it may have to reside in various areas of the memory, depending on its size. It would, therefore, be impractical to try to access this block using absolute addresses, that is without rewriting the program every time.

The solution to this problem lies in depositing the starting address of the block at a fixed memory location. This is analogous to a situation in which several persons need to get into a house, and only one key exists. By convention, the key to the house will be hidden under the mat. Every user will then know where to look (under the mat) to find the key to the house (or, perhaps, to find the address of the scheduled meeting, to propose a stricter analogy). Indirect addressing, therefore, normally

PROGRAMMING THE Z80

uses an opcode followed by a 16-bit address. This address is used to retrieve a word from the memory. Usually, it will be a 16-bit word (in our case, two bytes) within the memory since it is an address. This is illustrated by Figure 5.4. The two bytes at the specified address A₁ contain "A₂". A₂ is then interpreted as the actual address of the data that one wishes to access.

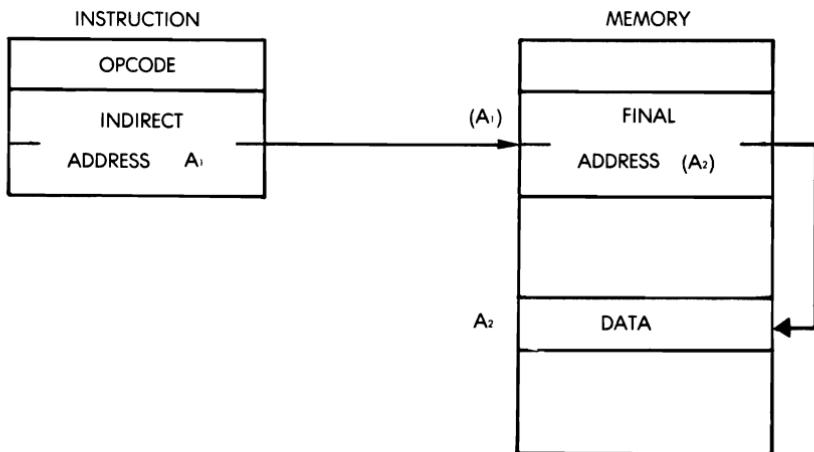


Fig. 5.4: Indirect Addressing

Indirect addressing is particularly useful any time that pointers are used. Various areas of the program can then refer to these pointers to access a word or a block of data conveniently and elegantly. The final address may also be obtained by pointing within the instruction to a 16-bit register in which it is contained. This is called "register indirect."

Combinations of Modes

The above addressing modes may be combined. In particular, it should be possible in a completely general addressing scheme to use many levels of indirection. The address A₂ could be interpreted as an indirect address again, and so on.

Indexed addressing can also be combined with indirect access. This allows the efficient access to word n of a block of data, provided one knows where the pointer to the starting address is (see figure 5.2).

We have now become familiar with all usual addressing modes that can be provided in a system. Most microprocessor systems, because of the limitation on the complexity of an MPU, which must be realized within a single chip, do not provide all possible modes but only a small subset of these. The Z80 provides a good subset of possibilities. Let us examine them now.

Z80 ADDRESSING MODES

Implied Addressing (Z80)

Implied addressing is essentially used by single-byte instructions which operate on internal registers. Whenever implicit instructions **operate exclusively on internal registers, they require only one machine cycle to execute.**

Examples of instructions using implied (or “register”) addressing are: LD r,r'; ADD A,r; ADC A,s; SUB s; SBC A,s; AND s; OR s; XOR s; CPs; INC r.

Zilog further distinguishes between “register addressing” and “implied addressing.” Implied addressing is then limited, in that definition, to instructions that do not have a specific field to point to an internal register. This introduces one more addressing mode. This is one reason why the number of addressing modes is insufficient to characterize the capabilities of a microprocessor.

Immediate Addressing (Z80)

Since the Z80 has both single-length registers (eight bits), and double-length register pairs (16 bits), it provides two types of immediate addressing, both with 8-bit and 16-bit literals. Instructions are then either two or three bytes long. The second (and sometimes the third) byte contains the opcode, followed by the constant, or literal, to be loaded in a register or used for an operation. Exceptions are LD IX and LD IY, which require 16-bit opcodes.

Examples of instructions using the immediate addressing mode are:

LD r,n (two bytes)

LD dd,nn (three bytes)

and

ADD A,n (two bytes)

When the literal is two bytes long, the mode is called “immediate extended,” in the case of the Z80.

PROGRAMMING THE Z80

Absolute or “Extended” Addressing (Z80)

By definition, absolute addressing requires three bytes. The first byte is the opcode and the next two bytes are the 16-bit address specifying the memory location (the “absolute address”).

By contrast with “short addressing” (eight-bit address), this mode is also called “extended addressing.”

Examples of instructions using extended addressing are:

LD HL, (nn) and JP nn

where nn represents the 16-bit memory address, and (nn) represents the contents of the specified location.

Modified Zero-Page Addressing (Z80)

Zero-page addressing is not available in the Z80, except through the RST instruction. The special addressing mode used by this instruction is called “modified zero-page addressing.”

The RST instruction contains a 3-bit field in bit position b_5, b_4, b_3 , used to point to one of 8 locations in page 0 memory. The effective address is $b_5b_4b_3000$ and is loaded into PC. Since it requires only a single byte, this instruction executes rapidly, and is easily generated in hardware. It was generally used to respond to multiple interrupts (up to 8.) Its disadvantage is either to limit the execution sequence to 8 locations, or to require a jump eliminating the speed advantage. This is because each of the 8 branch addresses are 8-bytes apart.

Relative Addressing (Z80)

By definition, relative addressing requires two bytes. The first one is the “jump relative” opcode, whereas the second one specifies the displacement and its sign.

In order to differentiate this mode from the absolute jump instruction, it is labeled “JR”.

From a timing standpoint, this instruction should be examined with caution. Whenever a test fails, i.e., whenever there is no branch, this in-

struction requires only seven “T cycles.” This is because the next instruction to be executed is already pointed to by the program counter.

However, when the test succeeds, i.e., whenever the jump takes place, this instruction requires 12 “T-states”; a new effective address must be computed and loaded into the program counter.

When computing the duration of the execution of a program segment, caution must be exercised. Whenever one is not sure whether or not the jump will succeed, one must take into consideration the fact that sometimes the jump *will require 12 T-states, (condition met), sometimes 7 (condition not met)*.

When designing a loop, execution will, therefore, be faster using a JR (Jump Relative) testing a condition usually *not met*, such as a non-zero condition for the counter.

When JR's are used outside of loops, and the condition under test is unknown, an average timing value is often used for the duration of JR.

This timing problem does not apply to the unconditional jump JR e. It does not test any condition, and always lasts 12 T-states.

Indexed Addressing (Z80)

This addressing mode did not exist in the 8080, and was added to the Z80 (as well as the two index registers). As a result, it became necessary to add an extra byte to the opcode, making it a 16-bit opcode in the Z80 instruction set (LDIR is another example of a 16-bit opcode). The structure of an indexed instruction is shown on Figure 5.5.

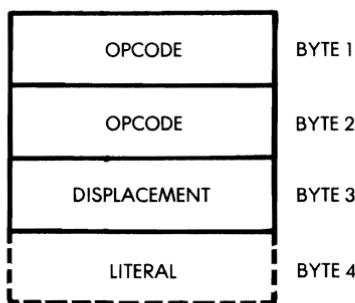


Fig. 5.5: Indexed Addressing Has 2-byte Opcode

Instructions allowing indexed addressing are:

LD, ADD, INC, RLC, BIT, SET, CP, and others.

This mode will be used extensively in the programs operating on blocks of data, tables or lists.

Indirect Addressing (Z80)

The Z80 provides a limited indirect addressing capability called "Register Indirect Addressing." In this mode, each of the 16-bit register pairs BC, DE, HL may be used as a memory address.

Whenever they point to 16-bit data, they point to the lower part. The higher part resides at the next (higher) sequential address.

Combinations of Modes

Combinations of modes are essentially non-existent, except that instructions referring to two operands may use a different type of addressing for each.

Thus, a *load* or an arithmetic instruction may access one operand in the immediate mode, and the other one through an indexed access.

Also, the bit addressing mechanism may access the eight-bit byte through one of the three addressing modes, as explained in the following paragraph. The specific addressing modes available for each instruction are indicated in the tables of the preceding chapter.

Bit Addressing

Bit addressing is generally not considered an addressing mode if addressing is defined as accessing a *byte*. However, whether defined as a mode or a group of instructions, it is a valuable facility. Since it is defined as an "addressing mode" in Zilog nomenclature, it will be so described here. It is specific to the Z80 and was not provided on the 8080.

Bit addressing refers to the access mechanism to specified bits. The Z80 is equipped with special instructions for setting, resetting and testing specified bits in a memory location or a register. The specified byte may be accessed through one of three addressing modes: register, register-indirect, and indexed. Three bits are used within the opcode to select one of eight bits.

USING THE Z80 ADDRESSING MODES

Long and Short Addressing

We have already used relative jump instructions in various programs that we have developed. They are self-explanatory. One interesting question is: What can we do if the permissible range for branching is not sufficient for our needs? On many microprocessors, the solution is to use a so called *long jump*. This is simply a jump to a location which contains an absolute or "long" jump specification:

| | |
|--------------------|---|
| JR NC, \$ + 3 | BRANCH TO CURRENT ADDRESS + 3 IF C CLEAR |
| JP FAR | OTHERWISE JUMP TO FAR |
| (NEXT INSTRUCTION) | |

The two-line program above will result in branching to location FAR whenever the carry is set. In the case of the Z80, JP may be used instead of JR to test all conditions and removes this problem.

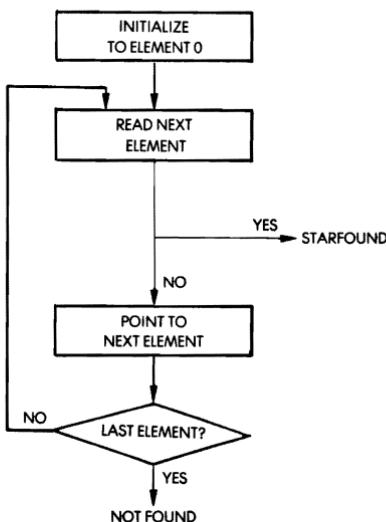
Use of Indexing for Sequential Block Accesses

Indexing is primarily used to address successive locations within a table. The restriction is that the maximum length must be less than 256 so that the displacement can reside in an eight-bit index register.

We have learned to check for a character. Now we will search a table of 100 elements for the presence of a '*'. The starting address for this table is called BASE. The table has only 100 elements. The program appears below: (see flowchart on Figure 5.6):

| | |
|--------|-------------|
| SEARCH | LD IX, BASE |
| | LD A, '*' |
| | LD B, COUNT |
| TEST | CP (IX) |
| | JR Z, FOUND |
| | INC IX |
| | DEC B |
| | JR NZ, TEST |
| NOTFND | ... |

An improved program will be presented below in the section on Block Transfer, using DJ NZ.

**Fig. 5.6: Character Search Flowchart****A Block Transfer Routine for Fewer Than 256 Elements**

We will call “COUNT” the number of elements in the block to be moved. The number is assumed to be less than 256. FROM is the base address of the block. TO is the base of the memory area where it should be moved. The algorithm is quite simple: we will move a word at a time, keeping track of which word we are moving by storing its position in the counter C. The program appears below:

| | | | |
|--------|-----|-----------|----------|
| BLKMOV | LD | IX, FROM | |
| | LD | IY, TO | |
| | LD | BC, COUNT | |
| NEXT | LD | A, (IX) | GET WORD |
| | LD | (IY), A | |
| | INC | IX | |
| | INC | IY | |
| | DEC | C | |
| | JR | NZ, NEXT | |

Let us examine it:

| | | | |
|--------|----|----------|--|
| BLKMOV | LD | IX, FROM | |
| | LD | IY, TO | |
| | LD | C, COUNT | |

These three instructions initialize registers IX, IY, and C respectively, as

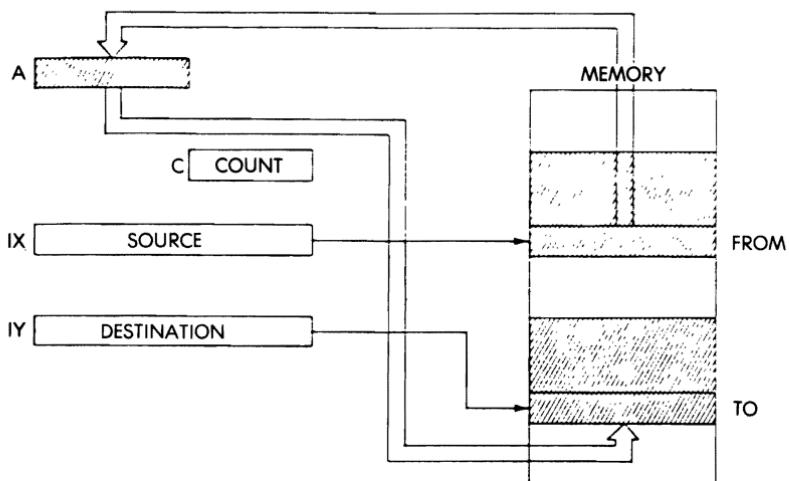


Fig. 5.7: Block Transfer: Initializing the Register

illustrated in Figure 5.7. Index register IX is used as the source pointer, and will be incremented regularly. Index register IY is used as the destination pointer, and would be incremented regularly. Register C is loaded with the maximum number of elements to be transferred (limited to 256 since this is an eight-bit register) and will be decremented regularly. Whenever C decrements to zero, all elements have been transferred. The next two instructions:

```
NEXT      LD    A, (IX)
          LD    (IY), A
```

load the contents of the memory location pointed to by IX into the accumulator, then transfer it into the memory location pointed to by register IY. In other words, these two instructions transfer an element of the source block into the destination block. The two index registers are then incremented:

```
INC  IX
INC  IY
```

And the counter register is decremented:

```
DEC  C
```

Finally, as long as the counter is not 0, the program loops back to the label NEXT:

```
JR   NZ, NEXT
```

PROGRAMMING THE Z80

This is an example of the possible utilization of index registers. However, let us compare it to the same program written for another microprocessor, the MOS Technology 6502, which is also equipped with an indexing capability, but uses different conventions (i.e., has different limitations on a general-purpose indexing facility). The program appears below:

```
        LDX    #NUMBER
NEXT     LDA    FROM, X
        STA    TO, X
        DEX
        BNE    NEXT
```

Without going into the details of the above program, the reader will immediately notice how much shorter it is than the previous one. This is because the index register X is used as a variable displacement, whereas FROM and TO are used as the fixed source and destination addresses.

This example should point out that although in theory indexing is a powerful facility, it does not necessarily lead to efficient coding, due to the addressing limitations imposed on it in the case of various microprocessors. Truly general-purpose indexing requires the possibility of a 16-bit displacement or address field as well as a 16-bit index register.

However, it should be noted that this specific problem is solved, in the Z80 by the presence of specialized instructions. A general-purpose block transfer will now be described which can be implemented in just four instructions. However, to be fair to the Z80, let us suggest additional exercises for the reader:

Exercise 5.1: Write the block transfer program for the Z80 in the style of the above program for the 6502, i.e., assuming that the index register contains a displacement. Assume that the source and the destination block are located in page 0, i.e., at addresses 0 to 256. Naturally, it will be assumed that the number of elements within each block is small enough that they do not overlap.

Exercise 5.2: Assume now that the source and the destination blocks are located anywhere in the memory, except that they are both within the same page. Rewrite the above program in that case. (Is there a difference, i.e., does page zero play any role for the Z80?)

Generalized Block Transfer Routine (More Than 256 Elements)

The register allocation and the memory map are shown in Figure 5.8.

The program is shown below:

| | |
|--------------|---------------------|
| LD BC, COUNT | NUMBER OF BYTES |
| LD DE, TO | DESTINATION ADDRESS |
| LD HL, FROM | START ADDRESS |
| LDIR | TRANSFER ALL BYTES |

Memory used: 11 bytes

Timing: 21 cycles/byte transferred

The first instruction is:

LD BC, COUNT

It loads the number of elements to be transferred (a 16-bit value) into the register pair BC. The next two instructions initialize the register pair DE and the register pair HL respectively:

LD DE, TO
LD HL, FROM

Finally the fourth instruction:

LDIR

performs the complete transfer.

LDIR is an *automated block-transfer* instruction. Its power should be obvious from this example. LDIR results in the following sequence: The contents of the memory location pointed to by H and L are transferred into the memory location pointed to by DE: $(DE) = (HL)$. Next, DE is incremented: $DE = DE + 1$. Then, HL is incremented: $HL = HL + 1$. Next, BC is decremented: $BC = BC - 1$. If BC becomes 0, the instruction is terminated. Otherwise, the instruction is repeated.

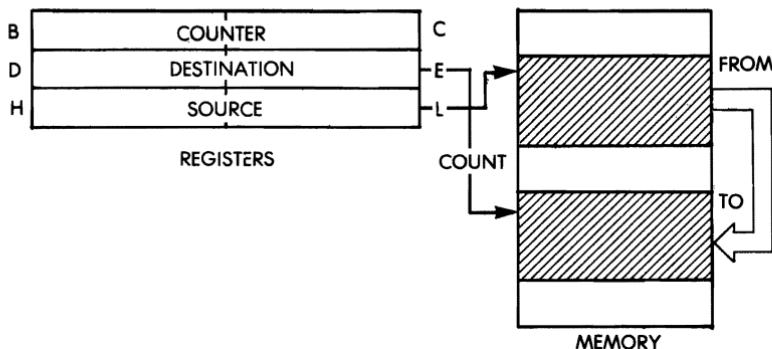


Fig. 5.8: A Block Transfer-Memory Map

PROGRAMMING THE Z80

The value and power of the LDIR instruction should be apparent at this point without further comments. Similarly, our search for the character "star" can be improved by the use of an automated instruction, CPIR, special to the Z80. The corresponding program appears below:

| | |
|--------|---------------|
| | LD A, " |
| | LD BC, COUNT |
| | LD HL, STRING |
| STAR | CPIR |
| | JR Z, STAR |
| NOSTAR | --- |

The first instruction loads the accumulator with the code for the character star. Next, the register pair BC is initialized to the count of the number of words to be searched within the block:

LD BC, COUNT

The register pair H and L is set to the starting address of the block to be searched (STRING). The automated instruction is then executed:

LD HL, STRING
CPIR

The CPIR instruction is an automated compare instruction. The contents of the memory location specified by the address contained in H and L is compared to the contents of the accumulator. If the comparison succeeds, then Z of the flags register will be set to 1. Then, the register pair H and L is incremented and the register pair BC is decremented. The instruction is repeated until either the pair BC goes to 0 or else the comparison succeeds. After the instruction CPIR is executed, it is therefore necessary to test the Z flag to determine whether the comparison has succeeded (the CPIR might have looped through 64K words without success in the extreme case). This is the purpose of the last instruction of the program:

JR Z, STAR

Exercise 5.3: Rewrite the above program so that a search proceeds backwards. (Hint: Use the CPDR instruction) Continue the block transfer until '' is found.*

Let us now develop a program combining the features of the two previous ones. We will implement the block transfer from location FROM

to location TO, which shall stop automatically whenever an escape character, "star", is found. The program appears below:

| | |
|--------------|---|
| LD BC, COUNT | |
| LD HL, FROM | |
| LD DE, TO | |
| LD A, '*' | DELIMITER (ESCAPE CHAR) |
| TEST CP (HL) | COMPARE WITH MEMORY CHARACTER |
| JR Z, END | END IF SUCCESS |
| LDI | TRANSFER CHARACTER AND UPDATE POINTERS AND COUNT |
| JP PE, TEST | KEEP TESTING UNLESS DONE P/V INDICATES WHETHER BC = 0 |

The first three instructions of the program perform the usual initialization, setting up the counter registers and the source and destination pointers:

```
LD BC, COUNT
LD HL, FROM
LD DE, TO
```

The star character is deposited, "as usual" into the accumulator, so that it can be compared to the character read from a memory location.

LD A, '*'

This is exactly what is done by the next instruction:

TEST CP (HL)

The success or failure of the comparison is determined by testing the Z bit. The Z bit will have been set if the comparison has succeeded. This is performed by the next instruction:

JR Z, END

The next instruction is an *automated transfer* instruction:

LDI

This instruction transfers the character, and updates the pointers and the *count* in a single instruction. LDI transfers the contents pointed to by H and L into the memory location pointed to by D and E: (DE) = (HL). It increments DE and HL:

DE = DE + 1
HL = HL + 1

PROGRAMMING THE Z80

Finally, it decrements BC: BC becomes $BC - 1$. The particularity of this instruction is that the P/V flag is cleared if BC decrements to “0” and set otherwise. This will be explicitly tested by the last instruction in the program to determine whether exit should occur:

JP PE, TEST

Adding Two Blocks

A program will be developed here to add element by element two blocks starting respectively at addresses BLK1, and BLK2, and having equal numbers of elements, COUNT. The program is shown below:

| | | |
|--------|-----|-------------|
| BLKADD | LD | IX, BLK1 |
| | LD | IY, BLK2 |
| | LD | B, COUNT |
| | XOR | A |
| LOOP | LD | A, (IX + 0) |
| | ADC | A, (IY + 0) |
| | LD | (IX), A |
| | DEC | IX |
| | DEC | IY |
| | DEC | B |
| | JR | NZ, LOOP |

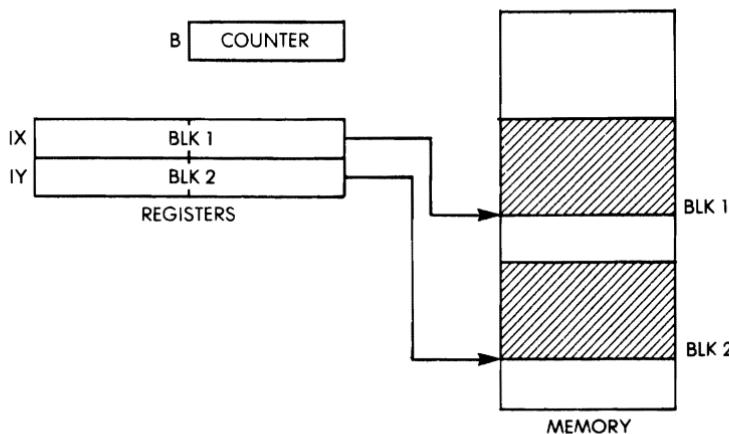


Fig. 5.9: Adding Two Blocks: $BLK1 = BLK1 + BLK2$

The memory layout is shown in Figure 5.9. The program is straightforward. The number of elements to be added is loaded into the counter register B, and the two index registers IX and IY are initialized to their values BLK1 and BLK2:

```
BLK ADD    LD  IX, BLK1
            LD  IY, BLK2
            LD  B, COUNT
```

The carry bit is then cleared in anticipation of the first addition:

```
XOR  A
```

The first element is loaded into the accumulator:

```
LOOP      LD  A, (IX + 0)
```

The corresponding element of BLK2 is then added to it:

```
ADC  A, (IY + 0)
```

and finally saved into the element of BLK1:

```
LD  (IX), A
```

The two pointer registers X and Y are decremented:

```
DEC  IX
DEC  IY
```

as well as the counter register:

```
DEC  B
```

As long as the counter register is not 0, the addition loop is executed:

```
JR  NZ, LOOP
```

Exercise 5.4: Can you use the above program to perform a 32-bit addition?

Exercise 5.5: Can you use the above program to perform a 64-bit addition?

Exercise 5.6: Modify the above program so that the result is stored in a separate block starting at address BLK3.

Exercise 5.7: Modify the above program to perform a subtraction rather than an addition.

PROGRAMMING THE Z80

Exercise 5.8: Modify the original program above so that BLK1 and BLK2 are at the top of each block rather than the bottom (see Fig.5.10).

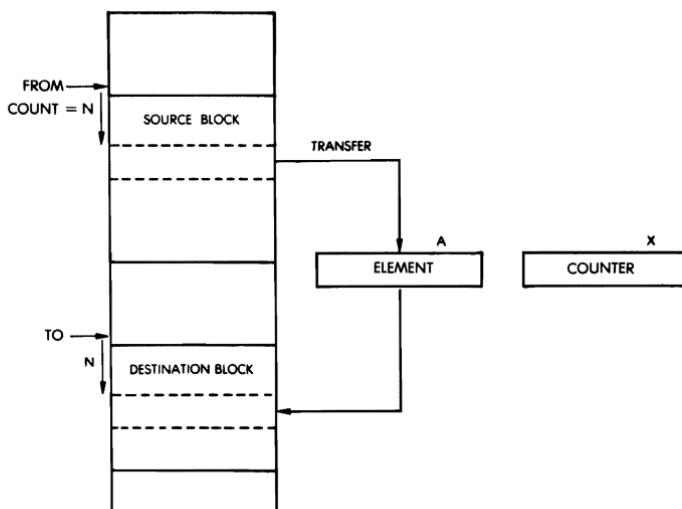


Fig. 5.10: Memory Organization for Block Transfer

SUMMARY

A complete description of addressing modes has been presented. It has been shown that the Z80 offers many possible mechanisms, and the specific addressing modes available on the Z80 have been analyzed. Finally, several application programs have been presented to demonstrate the value of the various addressing mechanisms. Programming the Z80 efficiently requires an understanding of these mechanisms. They will be used throughout the programs in the remainder of this book.

EXERCISES

5.9: Write a program to add the first 10 bytes of a table stored at location "BASE". The result will have 16 bits. (This is a checksum computation).

5.10: Can you solve the same problem without using the indexing mode?

5.11: Reverse the order of the 10 bytes of this table. Store the result at address “REVER”.

5.12: Search the same table for its largest element. Store it at memory address “LARGE”.

5.13: Add together the corresponding elements of three tables, whose bases are BASE1, BASE2, BASE3. The length of these tables is stored at address “LENGTH”.

6

INPUT/OUTPUT TECHNIQUES

INTRODUCTION

We have learned so far how to exchange information between the memory and the various registers of the processor. We have learned to manage the registers and to use a variety of instructions to manipulate the data. We must now learn to communicate with the external world. This is called input/output.

Input refers to the capture of data from outside peripherals (keyboard, disk, or physical sensor). *Output* refers to the transfer of data from the microprocessor or the memory to external devices such as a printer, a CRT, a disk, or actual sensors and relays.

We will proceed in two steps. First, we will learn to perform the input/output operations required by common devices. Secondly, we will learn to manage several input/output devices simultaneously, i.e., to *schedule* them. This second part will cover, in particular, polling vs. interrupts.

INPUT/OUTPUT

In this section we will learn to sense or to generate simple signals, such as pulses. Then we will study techniques for enforcing or measuring correct timing. We will then be ready for more complex types of input/output, such as high-speed serial and parallel transfers.

The Z80 Input/Output Instructions

The Z80 is equipped with a special set of input and output instructions. Most eight-bit microprocessors are not equipped with a special set of input and output instructions, and use the general instruction set

on input/output devices. The Z80, like the 8080, is equipped with basic input and output instructions. However, the Z80 is also equipped with additional I/O instructions. These will be described in more detail here in order to facilitate understanding of the programs that will be presented throughout this section.

The basic input and output instructions are respectively: IN A, (n) and OUT (n),A. These two instructions are inherited from the 8080. They will respectively read or write one byte between the selected port and the accumulator. The actual addressing process is such that the I,O device address "n" is gated on lines A0 through A7 of the address bus, while the contents of the accumulator appear on address lines A8 through A15. When only 256 devices are addressed, it may be necessary to zero the contents of the accumulator explicitly if any of the address lines A8 through A15 may be decoded by an I/O device. In the simple examples that follow, we will assume that fewer than 256 devices are present and that they are not connected to addresses A8 through A15, so that it will not be necessary to zero the contents of the accumulator explicitly, for example prior to using the IN instruction.

A special input instruction: IN r, (C), allows using the contents of register C as the I/O device address. When using this instruction, the contents of register B automatically provide the top part of the address (A8 through A15). The specified register r is loaded from the specified address. "r" may be any of the usual seven general-purpose registers.

Generate a Signal

In the simplest case, an output device will be turned off (or on) from the computer. In order to change the state of the output device, the programmer will merely change a level from a logical "0" to a logical "1", or from "1" to "0". Let us assume that an external relay is connected to bit "0" of a register called "OUT1". In order to turn it on, we will simply write a "1" into the appropriate bit position of the register. We assume here that OUT1 represents the address of this output register within our system. A program which will turn the relay on is:

| | |
|------------------------|---------------------|
| TURNON LD A, 00000001B | LOAD PATTERN INTO A |
| OUT (OUT1), A | OUTPUT IT TO DEVICE |

where OUT is the output instruction.

We have assumed that the state of the other seven bits of the register OUT1 is irrelevant. However, this is often not the case. These bits might be connected to other relays. Let us, therefore, improve this simple program. We want to turn the relay on, without changing the state

PROGRAMMING THE Z80

of any other bit within this register. We will assume that it is possible to read and write the contents of this register. Our improved program now becomes:

| | | | |
|--------|-----|-----------|---------------------------|
| TURNON | IN | A, (OUT1) | READ CONTENTS OF OUT1 |
| | OR | 00000001B | FORCE BIT "0" TO "1" IN A |
| | OUT | (OUT1), A | |

The program first reads the contents of location OUT1, then performs an inclusive OR on its contents. This only changes bit position 0 to "1", and leaves the rest of the register intact. (For more details on the OR operation, refer to Chapter 4.) This is illustrated by Figure 6.1.

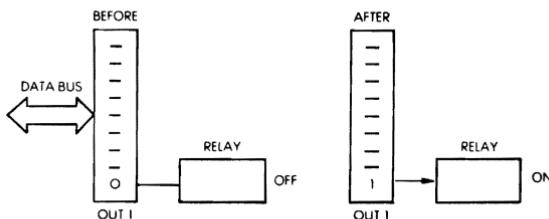


Fig. 6.1: Turning on a Relay

Pulses

Generating a *pulse* is accomplished exactly as in the case of the *level* above. An output bit is first turned on, then later turned off. This results in a pulse. This is illustrated in Figure 6.2. This time, however, an additional problem must be solved: one must generate the pulse for the correct length of time. Let us, therefore, study the generation of a computed delay.

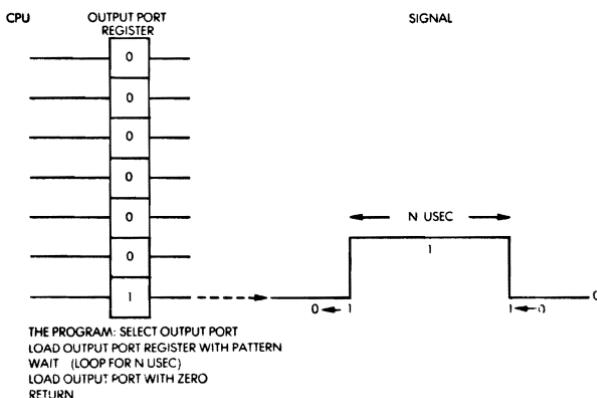


Fig. 6.2: A Programmed Pulse

Delay Generation and Measurement

A delay may be generated by software or by hardware methods. We will here study the way to perform it by program, and later show how it can also be accomplished with a hardware counter, called a programmable interval timer (PIT).

Programmed delays are achieved by counting. A counter register is loaded with a value, then is decremented. The program loops on itself and keeps decrementing until the counter reaches the value "0". The total length of time used by this process will implement the required delay. As an example, let us generate a delay of 82 clock cycles:

| | | | |
|-------|-----|----------|--------------|
| DELAY | LD | A, 5 | A IS COUNTER |
| NEXT | DEC | A | DECREMENT |
| | JR | NZ, NEXT | NEXT TEST |

This program loads A with the value 5. The next instruction decrements A and the following instruction will cause a branch to NEXT to occur as long as A does not decrement to "0". When A finally decrements to zero, the program will exit from this loop and execute whatever instruction follows. The logic of the program is simple and appears in the flowchart of Figure 6.3.

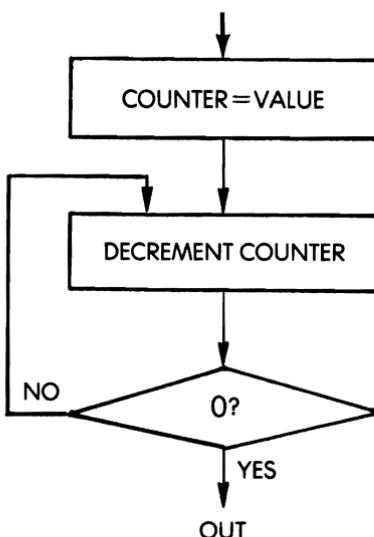
Let us now compute the effective delay which will be implemented by the program. In Chapter 4 of the book, we will look up the number of cycles required by each of these instructions:

LD in the immediate mode requires seven clock cycles. DEC will use four cycles. Finally, JR will use 12 cycles except during the last iteration, where it will use 7 cycles. When looking up the number of cycles for JR in the table, verify that two possibilities exist: if the branch does not occur, JR will only require seven cycles. If the branch does succeed, which will usually be the case during the loop, then 12 cycles are required.

The timing is, therefore, seven cycles for the first instruction, plus 11 cycles for the next two, multiplied by the number of times the loop will be executed, minus an extra five-cycle delay for the last unsuccessful JR:

$$\text{Delay} = 7 + 16 \times 5 - 5 = 82 \text{ cycles.}$$

Assuming a .5 microsecond cycle, this programming delay will be 41 microseconds.

**Fig. 6.3: Basic Delay Flowchart**

The delay loop which has been described is used by most input/output programs. It should be well understood. Try to do the following exercises:

Exercise 6.1: What are the maximum and the minimum delays which can be implemented with these three instructions?

Exercise 6.2: Modify the program to obtain a delay of about 100 microseconds.

If one wishes to implement a longer delay, a simple solution is to add extra instructions in the program, before DEC. The simplest way to do so is to add NOP instruction. (The NOP does nothing for four cycles.)

Longer Delays

Generating longer delays by software can be achieved through using a wider counter. A register pair can be used to hold a 16-bit count. To

simplify, let us assume that the lower count is "0". The lower byte will be loaded with "0", the maximum count, then go through a decrementation loop. Since the first decrementation results in $00 \rightarrow FF$ and does not affect the Z flag whenever it is decremented to "0", the upper byte of the counter will be decremented by 1. Whenever the upper byte is decremented to the value "0", the program terminates. If more precision is required in the delay generation, the lower count can have a non-null value. In this case, we would write the program just as explained and add at the end the three-line delay generation program, which has been described above.

A 24-bit delay program appears below:

| | | | |
|-------|------|------------|-----------------------|
| DEL24 | LD | B, COUNTH | COUNTER HIGH (8 BITS) |
| DEL16 | LD | DE, -1 | |
| LOOPA | LD | HL, COUNTL | COUNTER LOW |
| LOOPB | ADD | HL, DE | DECREMENT IT |
| | JR | C, LOOPB | GO ON UNTIL NULL |
| | DJNZ | LOOPA | DECREMENT B AND JUMP |

Note that DE is loaded with "-1", and used to decrement the 16-bit counter HL.

Naturally, still longer delays could be generated by using more than three words. This is analogous to the way an odometer works on a car. When the right-most wheel goes from "9" to "0", the next wheel to the left is incremented by 1. This is the general principle when counting with multiple discrete units.

However, the main disadvantage of this method is that when one is counting delays, the microprocessor will be doing nothing else for hundreds of milliseconds or even seconds. If the computer has nothing else to do, this is perfectly acceptable. However, in general the microcomputer should be available for other tasks, so that longer delays are normally not implemented by software. In fact, even short delays may be objectionable in a system if it is to provide some guaranteed response time in given situations. Hardware delays must then be used. In addition, if interrupts are used, timing accuracy may be lost if the counting loop can be interrupted.

Exercise 6.3: Write a program to implement a 100 ms delay (typical of a Teletype).

Hardware Delays

Hardware delays are implemented by using a *programmable interval timer* or "timer" in short. A register of the timer is loaded with a value.

The difference is that the timer will automatically decrement the counter periodically. The period can usually be adjusted or selected by the programmer. Whenever the timer has decremented to "0", it will normally send an interrupt to the microprocessor. It may also set a status bit which can be sensed periodically by the computer. The use of interrupts will be explained later in this chapter.

Other timer operating modes may include starting from "0" and counting the duration of the signal, or, counting the number of pulses received. When functioning as an interval timer, the timer is said to operate in a *one-shot* mode. When counting pulses, it is said to operate in a *pulse counting* mode. Some timer devices may even include multiple registers and a number of optional facilities which the programmer can select.

Sensing Pulses

The problem with sensing pulses is the reverse of that of generating pulses, and includes one more difficulty: whereas an output pulse is generated under program control, input pulses occur *asynchronously* with the program. In order to detect a pulse, two methods may be used: *polling* and *interrupts*. Interrupts will be discussed later in this chapter.

Let us now consider the polling technique. Using this technique, the program reads the value of a given input register continuously, testing a bit position, perhaps bit 0. It will be assumed that bit 0 is originally "0". Whenever a pulse is received, this bit will take the value "1". The program continuously monitors bit 0 until it takes the value "1". When a "1" is found, the pulse has been detected. The program appears below:

| | | | |
|------|-----|------------|---------------------|
| POLL | IN | A, (INPUT) | READ INPUT REGISTER |
| ON | BIT | 0, A | TEST FOR 0 |
| | JR | Z, POLL | KEEP POLLING IF 0 |

Conversely, let us assume that the input line is normally "1" and that we wish to detect a "0". This is the usual case for detecting a START bit, when monitoring a line connected to a Teletype. The program appears below:

| | | | |
|-------|-----|------------|---------------------|
| POLL | IN | A, (INPUT) | READ INPUT REGISTER |
| | BIT | 0, A | SET Z FLAG |
| | JR | NZ, POLL | TEST IS REVERSED |
| START | ... | | |

Monitoring the Duration

Monitoring the duration of the pulse may be accomplished in the same way as computing the duration of an output pulse. Either a hardware or a software technique may be used. When monitoring a pulse by software, a counter is regularly incremented by 1, then the presence of the pulse is verified. If the pulse is still present, the program loops upon itself. Whenever the pulse disappears, the count contained in the counter register is used to compute the effective duration of the pulse. The program appears below:

| | | | |
|--------|-----|------------|-------------------|
| DURTN | LD | B, 0 | CLEAR COUNTER |
| AGAIN | IN | A, (INPUT) | READ INPUT |
| | BIT | 0, A | MONITOR BIT 0 |
| | JR | Z, AGAIN | WAIT FOR A "1" |
| LONGER | INC | B | INCREMENT COUNTER |
| | IN | A, (INPUT) | CHECK BIT 0 |
| | BIT | 0, A | |
| | JR | NZ, LONGER | WAIT FOR A "0" |

Naturally, we assume that the maximum duration of the pulse will not cause register B to overflow. If this were the case, the program would have to be changed to take that into account (or else it would be a programming error!).

Since we now know how to sense and generate pulses, let us capture or transfer larger amounts of data. Two cases will be distinguished: serial data and parallel data. Then we will apply this knowledge to actual input/output devices.

PARALLEL WORD TRANSFER

It is assumed here that eight bits of transfer data are available in parallel at address "INPUT" (see Fig. 6.4). The microprocessor must read the data word at this location whenever a status word indicates that it is valid. The status information will be assumed to be contained in bit 7 of address "STATUS". We will here write a program which will read and automatically save each word of data as it comes in. To simplify, we will assume that the number of words to be read is known in advance and is contained in location "COUNT". If this information were not available, we would test for a so-called *break character*, such as a *rubout*, or perhaps the character "*". We have learned to do this already.

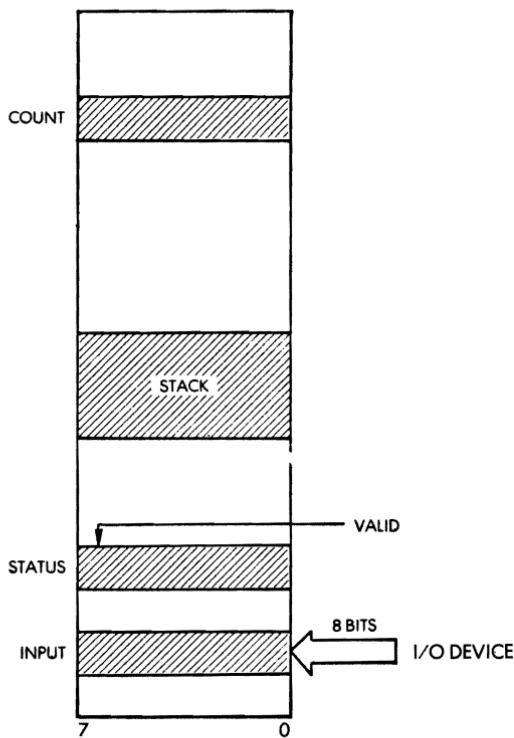


Fig. 6.4: Parallel Word Transfer - The Memory

The flowchart appears in Figure 6.5. It is quite straightforward. We test the status information until it becomes “1”, indicating that a word is ready. When the word is ready, we read it and save it at an appropriate memory location. We then decrement the counter and test whether it has decremented to “0”. If so, we are finished; if not, we read the next word. A simple program which implements this algorithm appears below:

| | | | |
|-------|------|-------------|-------------------------------|
| PARAL | LD | A, (COUNT) | READ COUNT INTO A |
| | LD | B, A | B IS COUNTER |
| WATCH | IN | A, (STATUS) | LOOK FOR ‘DATA READY’ TRUE |
| | BIT | 7, A | BIT 7 IS “1” IF DATA READY |
| | JR | Z, WATCH | DATA VALID? |
| | IN | A, (INPUT) | READ DATA |
| | PUSH | AF | SAVE DATA INTO STACK |

DEC B DECREMENT COUNT
 JR NZ, WATCH DO IT UNTIL ZERO

It is assumed that the "data ready" flag is automatically cleared when STATUS is read.

The first two instructions initialize the counter register B:

PARAL LD A, (COUNT)
 LD B, A

Note that there is no easy way to load B only from memory. One must either load A, then transfer its contents to B, or load B and C simultaneously.

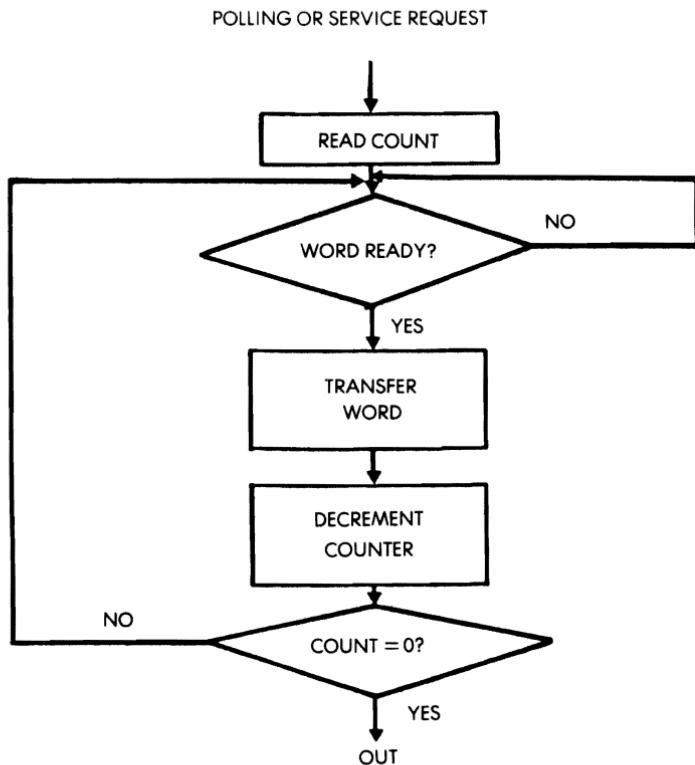


Fig. 6.5: Parallel Word Transfer: Flowchart

PROGRAMMING THE Z80

The next three instructions of the program read the status information and cause a loop to occur as long as bit seven of the status register is "0". (It is the sign bit, i.e., bit N.)

| | | |
|-----|-------------|-----------------------------|
| IN | A, (STATUS) | |
| BIT | 7, A | "IN" DOES NOT SET THE FLAGS |
| JR | Z, WATCH | |

When JP fails, data is valid and we can read it:

| | | |
|----|------------|--|
| IN | A, (INPUT) | |
|----|------------|--|

The word has now been read from address INPUT where it was, and must be saved. Assuming that a sufficient stack area is available, we can use:

| | | |
|------|----|--|
| PUSH | AF | |
|------|----|--|

which saves A (and F) in the stack. If the stack is full, or the number of words to be transferred is large, we could not push them on the stack and we would have to transfer them to a designated memory area, using, for example, an indexed instruction. However, this would require an extra instruction to increment or decrement the index register. PUSH is faster (only 11 clock cycles).

The word of data has now been read and saved. We will simply decrement the word counter and test whether we are finished:

| | | |
|-----|-----------|--|
| DEC | B | |
| JR | NZ, WATCH | |

This nine-instruction program can be called a *benchmark*. A benchmark program is a carefully optimized program designed to test the capabilities of a given processor in a specific situation. Parallel transfers are one such typical situation. This program has been designed for maximum speed and efficiency. Let us now compute the maximum transfer speed of this program. We will assume that COUNT is contained in memory. The duration of every instruction is determined by inspecting the tables in Chapter Four and is found to be the following:

| | | | |
|-------|-----|-------------|------|
| PARAL | LD | A, (COUNT) | 13 |
| | LD | B, A | 4 |
| WATCH | IN | A, (STATUS) | 11 |
| | BIT | 7, A | 8 |
| | JR | Z, WATCH | 7/12 |

| | | |
|------|------------|------|
| IN | A, (INPUT) | 11 |
| PUSH | AF | 11 |
| DEC | B | 4 |
| JR | NZ, WATCH | 7/12 |

The minimum execution time is obtained by assuming that data is available every time that we sample STATUS. In other words, the first JP will be assumed to fail every time. Timing is then:

$$13 + 4 + (11 + 8 + 7 + 11 + 4 + 12) * \text{COUNT}$$

Neglecting the first 17 cycles necessary to initialize the counter register, the time used to transfer one word is 64 clock cycles or 32 microseconds with a 2 MHz clock.

The maximum data transfer rate is, therefore:

$$\frac{1}{32(10^{-6})} = 31 \text{ K bytes per second}$$

Exercise 6.4: Assume that the number of words to be transferred is greater than 256. Modify the program accordingly and determine the impact on the maximum data transfer rate.

Exercise 6.5: Modify this program in order to try to improve its speed:

- 1—using JR instead of JP
- 2—using DJNZ
- 3—using INI or IND

Was the above program truly optimal?

We have now learned to perform high-speed parallel transfers. Let us consider a more complex case.

BIT SERIAL TRANSFER

A serial input is one in which the bits of information (0's or 1's) come in successively on a line. These bits may come in at regular intervals. This is normally called *synchronous* transmission. Or, they may come as bursts of data at random intervals. This is called *asynchronous* transmission. We will develop a program which can work in both cases. The principle of the capture of sequential data is simple: we will watch an input line, which will be assumed to be line 0. When a bit of data is detected on this line, we will read the bit in, and shift it into a holding register. Whenever eight bits have been assembled, we will preserve the byte of data into the memory and assemble the next one. In order to simplify, we will assume that the number of bytes to be received is

PROGRAMMING THE Z80

known in advance. Otherwise, we might, for example, have to watch for a special break character, and stop the bit-serial transfer at this point. We have learned to do that. The flowchart for this program appears in Figure 6.6. The program appears below:

| | | | |
|--------|-----|------------|--------------------------------|
| SERIAL | LD | C, 0 | CLEAR INPUT WORD |
| | LD | A, (COUNT) | LOAD B WITH BYTE COUNT |
| | LD | B, A | |
| LOOP | IN | A, (INPUT) | READ PORT |
| | BIT | 7, A | BIT 7 IS STATUS, BIT 0 IS DATA |
| | JR | Z, LOOP | WAIT FOR A "1" |
| | SRL | A | SHIFT DATA BIT INTO CARRY |
| | RL | C | SAVE INPUT B INTO C |
| | JR | NC, LOOP | CONTINUE UNTIL 8 BITS IN |

POLLING OR SERVICE REQUEST

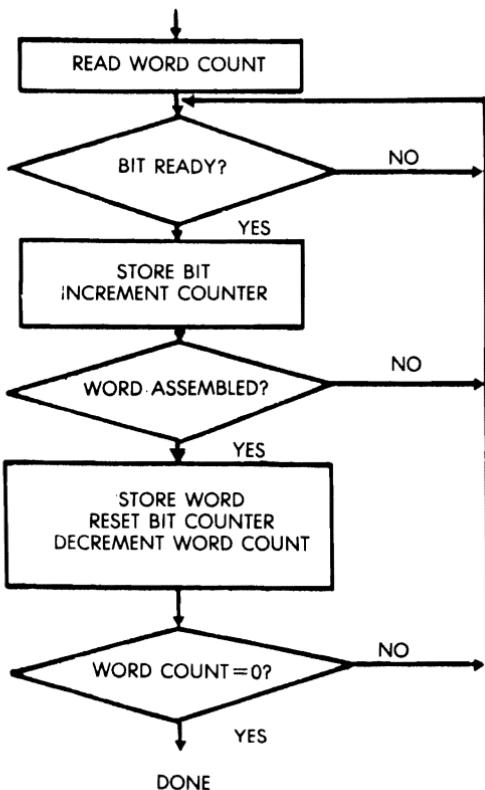


Fig. 6.6: Bit Serial Transfer—Flowchart

| | |
|-------------|------------------------|
| PUSH BC | SAVE WORD IN STACK |
| LD C, 01H | RESET MARKER BIT |
| DEC B | DECREMENT BYTE COUNTER |
| JR NZ, LOOP | ASSEMBLE NEXT WORD |

This program has been designed for efficiency and will use new techniques which we will explain (see Fig. 6.7).

The conventions are the following: memory location COUNT is assumed to contain a count of the number of words to be transferred. Register C will be used to assemble eight consecutive bits coming in. Address INPUT refers to an input register. It is assumed that bit position 7 of this register is a status flag, or a clock bit. When it is "0", data is not valid. When it is "1", the data is valid. The data itself will be assumed to appear in bit position 0 of this same address. In many instances, the status information will appear on a different register than the data register. It should be a simple task, then, to modify this program accordingly. In addition, we will assume that the first bit of data to be received by this program is guaranteed to be a "1". It indicates that the real data follows. If this were not the case, we will later see an obvious modification to take care of it. The program corresponds exactly to the flowchart of Fig. 6.6. The first few lines of the program implement a waiting loop which tests whether a bit is ready. To determine whether a bit is ready, we read the input register, then test the zero bit (Z). As long as this bit is "0", the instruction JR will succeed, and we will branch back to the loop. Whenever the status (or clock) bit becomes true ("1"), then JR will fail and the next instruction will be executed.

This initial sequence of instructions corresponds to arrow 1 in Fig. 6.7.

At this point, the accumulator contains a "1" in bit position 7 and the actual data bit in bit position 0. The first data bit to arrive is going to be a "1". However, the following bits may be either "0" or "1". We now wish to preserve the data bit which has been collected in position 0. The instruction:

SRL A

shifts the contents of the accumulator right by one position. This causes the right-most bit of A, which is our data bit, to fall into the carry bit. We will now preserve this data bit into register C (this process is illustrated by arrows 2 and 3 in Fig. 6.7):

RL C

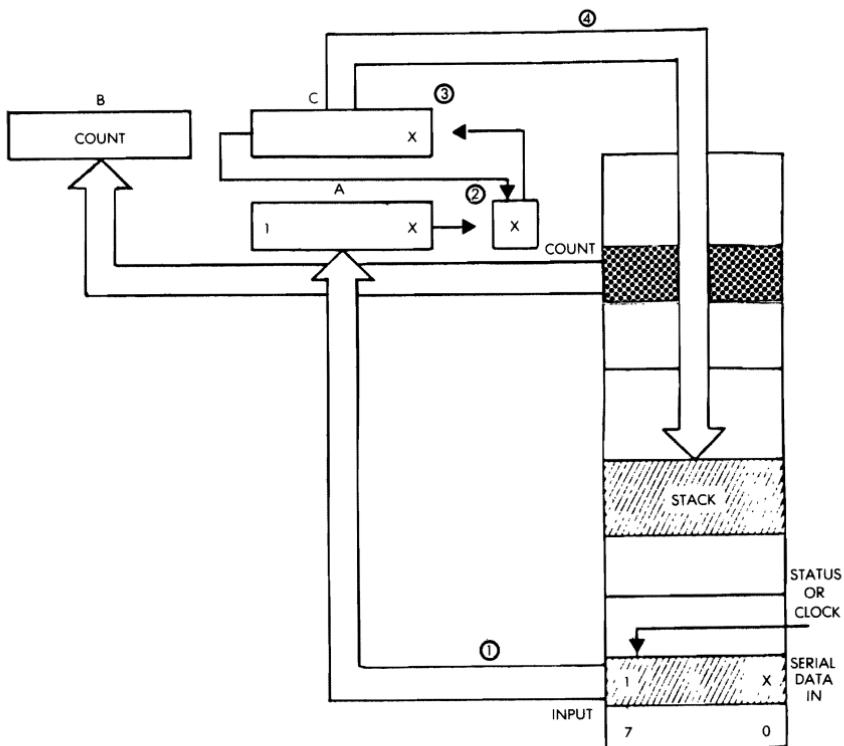


Fig. 6.7: Serial-to-Parallel: The Registers

The effect of this instruction is to read the carry bit into the right-most bit position of C. At the same time, the left-most bit of C falls into the carry bit. (If you have any doubts about the rotation operation, refer to Chapter 4!)

It is important to remember that a rotation with carry operation will both save the carry bit, here into the right-most bit position, and also recondition the carry bit with the value of bit 7 (or bit 0).

Here, a "0" will fall into the carry. The next instruction:

JR NC, LOOP

tests the carry and branches back to address LOOP as long as the carry

is "0". This is our automatic bit counter. It can readily be seen that, as a result of the first RL, C will contain "00000001". Eight shifts later, the "1" will finally fall into the carry bit and stop the branching. This is an ingenious way to implement an automatic loop counter without having to waste an instruction to decrement the contents of an index register. This technique is used in order to shorten the program and improve its performance.

When JR NC finally fails, 8 bits will have been assembled into C. This value should be preserved in the memory. This is accomplished by the next instruction (arrow 4 on Fig. 6.7):

PUSH BC

We are here saving the contents of B and C into the stack. Saving into the stack is possible only if there is enough room in the stack. Provided that this condition is met, it is usually the fastest way to preserve a word in the memory, even though we save an unnecessary register (B). The stack pointer is updated automatically. If we were not pushing a word in the stack, we would have to use one more instruction to update a memory pointer. We could equivalently perform an indexed addressing operation, but that would also involve decrementing or incrementing the index, using extra time.

After the first word of data has been saved, there is no longer any guarantee that the first data bit to come in will be a "1". It can be anything. We must, therefore, reset the contents to "00000001" so that we can keep using it as a bit counter. This is performed by the next instruction:

LD C, 01H

Finally, we will decrement the word counter, since a word has been assembled, and test whether we have reached the end of the transfer. This is accomplished by the next two instructions:

**DEC B
JR NZ, LOOP**

The above program has been designed for speed, so that one may capture a fast input stream of data bits. Once the program terminates, it is naturally advisable to immediately read away from the stack the words that have been saved there and transfer them elsewhere into the memory. We have already learned to perform such a block transfer in Chapter 2.

PROGRAMMING THE Z80

Exercise 6.6: Compute the maximum speed at which this program will be able to read serial bits. Look up the number of cycles required by every instruction in the table at the end of this book, then compute the time which will elapse during execution of this program. To compute the length of time which will be used by a loop, simply multiply the total duration of this loop, expressed in microseconds, by the number of times it will be executed. Also, when computing the maximum speed, assume that a data bit will be ready every time that the input location is sensed.

This program is more difficult to understand than the previous ones. Let us look at it again (refer to Fig. 6.6) in more detail, examining some trade-offs.

A bit of data comes into bit position 0 of “INPUT” from time to time. There might be, for example, three “1s” in succession. We must, therefore, *differentiate between the successive bits coming in*. This is the function of the “clock” signal.

The clock (or STATUS) signal tells us that the input bit is now valid. Before reading a bit, we will therefore first test the status bit. If the status is “0”, we must wait. If it is “1”, then the data bit is good.

We assume here that the status signal is connected to bit 7 of register INPUT.

Exercise 6.7: Can you explain why bit 7 is used for status, and bit 0 for data? Does it matter?

Once we have captured a data bit, we want to preserve it in a safe location, then shift it left, so that we can get the next bit.

Unfortunately, the accumulator is used to read and test both data and status in this program. If we were to accumulate data in the accumulator, bit position 7 would be erased by the status bit.

Exercise 6.8: Can you suggest a way to test status without erasing the contents of the accumulator (a special instruction)? If this can be done, could we use the accumulator to accumulate the successive bits coming in? Can you improve speed by using an “automated jump”?

Exercise 6.9: Rewrite the program, using the accumulator to store the bits coming in. Compare it to the previous one in terms of speed and number of instructions.

Let us address two more possible variations.

We have assumed that, in our particular example, the very first bit to come in would be a special signal, guaranteed to be “1”. However, in

general, it may be anything.

Exercise 6.10: Modify the program above, assuming that the very first bit to come in is valid data (not to be discarded), and can be "0" or "1". Hint: our "bit counter" should still work correctly, if you initialize it with the correct value.

Finally, we have been saving the assembled word in the stack, to gain time. We could naturally save it in a specified memory area.

Exercise 6.11: Modify the program above, and save the assembled word in the memory area starting at BASE.

Exercise 6.12: Modify the program above so that the transfer will stop when the character "S" is detected in the input stream.

The Hardware Alternative

As usual for most standard input/output algorithms, it is possible to implement this procedure by hardware. The chip is called a UART. It will automatically accumulate the bits. However, when one wishes to reduce the component count, this program, or a variation of it, will be used instead.

Exercise 6.13: Modify the program, assuming that data is available in bit position 0 of location INPUT, while the status information is available in bit position 0 of address INPUT + 1.

BASIC I/O SUMMARY

We have now learned to perform elementary input/output operations as well as to manage a stream of parallel data or serial bits. We are now ready to communicate with real input/output devices.

COMMUNICATING WITH INPUT/OUTPUT DEVICES

In order to exchange data with input/output devices, we will first have to ascertain whether data is available, if we want to read it; or whether the device is ready to accept data, if we want to send it. Two procedures may be used: handshaking and interrupts. Let us study handshaking first.

Handshaking

Handshaking is generally used to communicate between any two

PROGRAMMING THE Z80

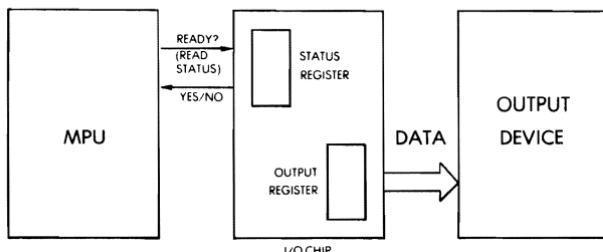


Fig. 6.8: Handshaking (Output)

asynchronous devices, i.e., between any two devices which are not synchronized. For example, if we want to send a word to a parallel printer, we must first make sure that the input buffer of this printer is available. We will, therefore, ask the printer: Are you ready? The printer will say “yes” or “no.” If it is not ready we will wait. If it is ready, we will send the data (see Fig. 6.8).

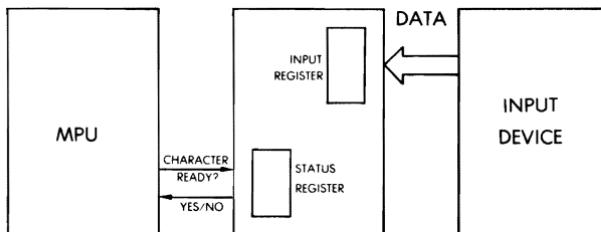


Fig. 6.8a: Handshaking (Input)

Conversely, before reading data from an input device, we will verify whether the data is valid. We will ask: “Is data valid?” And the device will tell us “yes” or “no.” The “yes or no” may be indicated by status bits, or by other means (see Fig. 6.8a).

As an analogy, whenever you wish to exchange information with someone who is independent and might be doing something else at the time, you should ascertain that he is ready to communicate with you. The usual rule of courtesy is to shake his hand. Data exchange may then follow. This is the procedure normally used in communicating with in-

put/output devices.

Let us now illustrate this procedure with a simple example.

Sending a Character To The Printer

The character will be assumed to be contained in memory location CHAR. The program to print it appears below:

| | | | |
|------|-----|-------------|----------------|
| WAIT | IN | A, (STATUS) | |
| | BIT | 7, A | TEST IF READY |
| | JR | Z, WAIT | OTHERWISE WAIT |
| | LD | A, (CHAR) | GET CHARACTER |
| | OUT | (PRNTD), A | PRINT IT |
| | JR | WAIT | GO FOR NEXT |

The print program is straightforward and uses the handshaking procedure which has been described above. The data paths are shown in Figure 6.9.

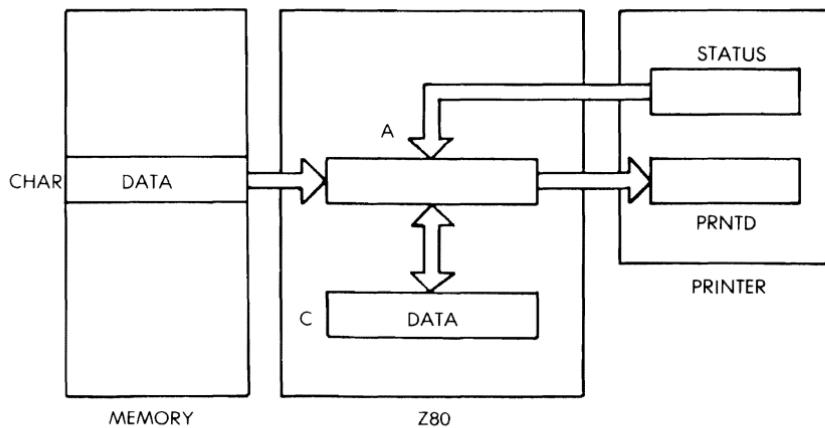


Fig. 6.9: Printer—Data Paths

The character (called DATA) is located at memory location CHAR. First, the status of the printer is checked. Whenever bit 7 of the status

register becomes 1, it indicates that the printer ready for input, i.e., its input buffer is available. At this point, the character is loaded into the accumulator, then output to the printer, via the accumulator. As long as the status bit remains 0, the program will remain in a loop, called WAIT in the program.

Exercise 6.14: How many instructions would be saved in the above program by loading data directly into register C as well as outputting the contents of register C directly?

Exercise 6.15: When using an actual printer, it is usually necessary to send a start order before using the device. Modify this program to generate such an order, assuming that the start command is obtained by writing a 1 in bit position 0 of the STATUS register, which is assumed to be bidirectional.

Exercise 6.16: If the BIT instruction were not available, could you use another instruction instead, in line 2 of the program? If so, explain the advantage of using the BIT instruction, if any.

Exercise 6.17: Modify the program above to print a string of n characters, where n will be assumed to be less than 255.

Exercise 6.18: Modify the above program to print a string of characters until a “carriage-return” code is encountered.

Let us now complicate the output procedure by requiring a code conversion and by outputting to several devices at a time:

Output To a Seven-Segment LED

A traditional seven-segment light-emitting diode (LED) may display the digits “0” through “9”, or even “0” through “F” hexadecimal by lighting combinations of its 7 segments. A seven-segment LED is shown in Figure 6.10. The characters that may be generated with this LED appear in Figure 6.11.

The segments of an LED are labeled “a” through “g” in Figure 6.10.

For example, “0” will be displayed by lighting the segments abcdef. Let us assume, now, that bit “0” of an output port is connected to segment “a”, that “1” is connected to segment “b”, and so on. Bit 7 is not used. The binary code required to light up fedcba (to display “0”) is, therefore, “0111111”. In hexadecimal this is “3F”. Do the following exercise.

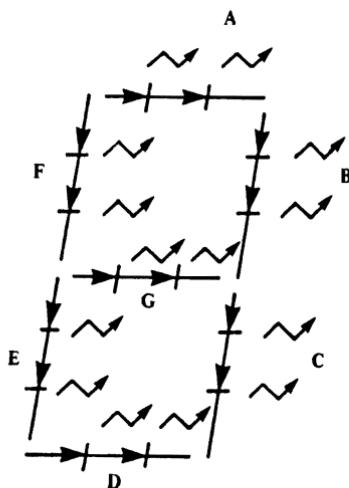


Fig. 6.10: Seven-Segment LED

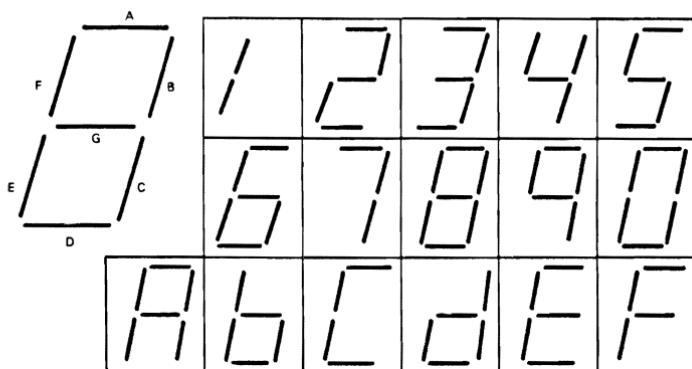


Fig. 6.11: Hexadecimal Characters Generated with a Seven-Segment LED

PROGRAMMING THE Z80

Exercise 6.19: Compute the seven-segment equivalent for the hexadecimal digits “0” through “F”. Fill out the table below:

| Hex | LED code |
|-----|----------|-----|----------|-----|----------|-----|----------|
| 0 | 3F | 4 | | 8 | | C | |
| 1 | | 5 | | 9 | | D | |
| 2 | | 6 | | A | | E | |
| 3 | | 7 | | B | | F | |

Let us now display hexadecimal values on *several* LED's.

Driving Multiple LED's

An LED has no memory. It will display the data only as long as its segment lines are active. In order to keep the cost of an LED display low, the microprocessor will display information on *each of the LED's* in turn. The rotation between the LED's must be fast enough so that there is no apparent blinking. This implies that the time spent from one LED to the next is less than 100 milliseconds. Let us design a program which will accomplish this. Register C will be used to point to the LED on which we want to display a digit. The accumulator is assumed to contain the hexadecimal value to be displayed on the LED. Our first concern is to convert the hexadecimal value into its seven-segment representation. In the preceding section, we have built the equivalence table. Since we are accessing a table, we will use the indexed addressing mode, where the displacement index will be provided by the hexadecimal value. This means that the seven-segment code for hexadecimal digit “3” is obtained by looking up the third element of the table after the base. The address of the base will be called SEGBAS. The program appears below:

| | | | |
|-------|-----|------------|--------------------------------|
| LEDS | LD | E, A | A CONTAINS HEX DIGIT |
| | LD | D, 0 | USE “DE” AS DISPLACEMENT |
| | LD | HL, SEGBAS | USE “HL” AS INDEX |
| | ADD | HL, DE | TABLE ADDRESS |
| | LD | A, (HL) | READ CODE FROM TABLE |
| | LD | B, 50H | DELAY VALUE = ANY LARGE NBR |
| DELAY | OUT | (C), A | OUTPUT FOR SET DURATION |
| | DEC | B | DELAY COUNTER |

| | | |
|-----|--------------|---------------------------|
| JR | NZ, DELAY | KEEP LOOPING |
| LD | A, C | C IS PORT NUMBER |
| DEC | C | |
| CP | MINLED | DONE FOR LAST LED? |
| JR | NZ, OUT | |
| LD | BC, (MAXLED) | IF SO, RESET C TO TOP LED |
| OUT | RET | |

The program assumes that register C contains the address of the LED to be illuminated next, and that the accumulator A contains the digit to be displayed.

The program first looks up the seven-segment code corresponding to the hexadecimal value contained in the accumulator. Registers D and E are used as a displacement field, and registers H and L are used as a 16-bit index register. The hexadecimal digit is added to the base address of the table:

| | | | |
|------|-----|------------|----------------|
| LEDS | LD | E, A | 7-SEGMENT CODE |
| | LD | D, 0 | |
| | LD | HL, SEGBAS | |
| | ADD | HL, DE | |

A delay loop is then implemented, so that the code obtained from the table is displayed for an appropriate duration. Here the constant “50” hexadecimal has been arbitrarily chosen:

| | | |
|----|---------|----------------------|
| LD | A, (HL) | READ CODE FROM TABLE |
| LD | B, 50H | DELAY VALUE |

The delay is accomplished using a classic delay loop. The first instruction:

DELAY OUT (C), A

outputs the contents of the accumulator at the I/O port pointed to by register C (the LED number). The next two instructions implement the delay loop:

| | | |
|-----|-----------|--|
| DEC | B | |
| JR | NZ, DELAY | |

Once the delay has been implemented, we must simply decrement the LED pointer, and make sure that we loop around to the highest LED address if the smallest LED address has been reached:

LD A,C

```
DEC    C
CP     MINLED
JR     NZ, OUT
LD     BC, (MAXLED)
OUT    RET
```

It is assumed here that the above program has been written as a subroutine, and the last instruction is then RET: "return from subroutine"

Exercise 6.20: *It is usually necessary to turn off the segment drivers for the LED prior to displaying the digit. Modify the above program by adding the necessary instructions (output "00" as the character code prior to outputting the character).*

Exercise 6.21: *What would happen to the display if the DELAY label were moved up by one line position? Would this change the timing? Would this change the appearance of the display?*

Exercise 6.22: *You will notice that the first four instructions of the program are, in fact, performing a 16-bit indexed memory access. However, it seems clumsy, without using the indexing mechanism. Assume that the SEGBAS address is known in advance. Call SEGBSH the high-order part of this address, and SEGBSL the low part of this address. Store SEGBSH in the high-order part of the IX register. Now write the above program, using the Z80 index-addressing mechanism, and using SEGBSL as the displacement field of the instruction. What are the advantages and disadvantages of this approach?*

Exercise 6.23: *Assuming that the above program is a subroutine, you will notice that it uses registers B, D, E, H and L internally, and modifies their contents. If the subroutine may freely use the memory area designated by address T1, T2, T3, T4, T5, could you add instructions at the beginning and at the end of this program which will guarantee that, when the subroutine returns, the contents of registers B, D, E, H and L will be the same as when the subroutine was entered?*

Exercise 6.24: *Same exercise as above, but assume that the memory area T1, etc., is not available to the subroutine. (Hint: remember that there is a built-in mechanism in every computer for preserving information in a chronological order.)*

We have now solved common input/output problems. Let us consider the case of a common peripheral: the Teletype.

Teletype Input-Output

The Teletype is a serial device. It both sends and receives words of information in a serial format. Each character is encoded in an 8-bit ASCII format (the ASCII table appears at the end of this book). In addition, every character is preceded by a "start" bit, and terminated by two "stop" bits. In the so-called 20-milliamper current loop interface, which is most frequently used, the state of the line is normally a "1". This is used to indicate to the processor that the line has not been cut. A start is a "1"-to-"0" transition. It indicates to the receiving device that data bits follow. The standard Teletype is a 10-characters-per-second device. We have just established that each character requires 11 bits. This means that the Teletype will transmit 110 bits per second. It is said to be a 110-baud device. We will design a program to serialize bits out to the Teletype at the correct speed.

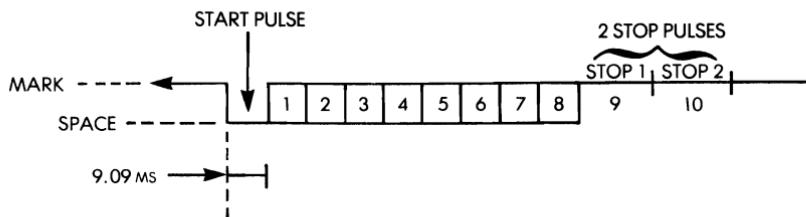


Fig. 6.12: Format of a Teletype Word

One-hundred-and-ten bits per second implies that bits are separated by 9.09 milliseconds. This will have to be the duration of the delay loop to be implemented between successive bits. The format of a Teletype word appears in Figure 6.12. The flowchart for bit input appears in Figure 6.13. The program follows:

| | | | |
|-------|------|-------------|-------------------|
| TTYIN | IN | A, (STATUS) | |
| | BIT | 7, A | DATA READY? |
| | JR | Z, TTYIN | OTHERWISE WAIT |
| | CALL | DELAY1 | CENTER OF PULSE |
| | IN | A, (TTYBIT) | START BIT |
| | OUT | (TTYBIT), A | ECHO IT |
| | CALL | DELAY9 | NEXT PULSE (9 MS) |
| | LD | B, 08H | BIT COUNT |
| NEXT | IN | A, (TTYBIT) | READ DATA BIT |
| | OUT | (TTYBIT), A | ECHO IT |
| | SRL | A | SAVE IT IN CARRY |

PROGRAMMING THE Z80

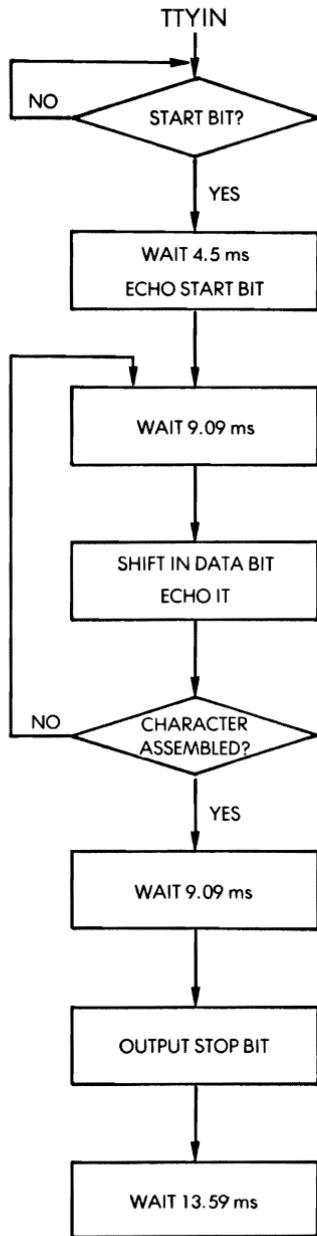


Fig. 6.13: TTY Input with Echo

| | | |
|------|-------------|---------------------|
| RR | C | PRESERVE IT INTO C |
| CALL | DELAY9 | NEXT PULSE (9 MS) |
| DEC | B | DECREMENT BIT COUNT |
| JR | NZ, NEXT | |
| IN | A, (TTYBIT) | READ STOP BIT |
| OUT | (TTYBIT), A | ECHO IT |
| CALL | DELAY9 | SKIP SECOND STOP |
| RET | | |

Fig. 6.14: Teletype Program

Let us examine the program in detail. First, the status of the Teletype must be tested to determine if a character is available:

| | | |
|-------|-----|-------------|
| TTYIN | IN | A, (STATUS) |
| | BIT | 7, A |
| | JR | Z, TTYIN |

The “BIT” instruction is a useful Z80 facility which allows testing any bit in any data register. It does not modify the contents of the register under test. The Z flag is set if the specified bit is 0, and reset otherwise.

This program will, therefore, loop until the status finally becomes “1”. It is a classic polling loop.

Note that, since the STATUS does not need to be preserved, we could also use

AND 1000000B

instead of

BIT 7, A

However, using the AND instruction destroys the contents of A (acceptable here).

When optimizing a program, remember that each new instruction may introduce side-effects.

Next, a 4.5 ms delay is implemented in order to sense the start bit in the middle of the pulse.

CALL DELAY1

where DELAY1 is the delay subroutine implementing the required delay. The first bit to come is the start bit. It should be echoed to the Teletype, but otherwise ignored. This is done by the next instructions:

| | | |
|-------|-----|-------------|
| TTYIN | IN | A, (TTYBIT) |
| | OUT | (TTYBIT), A |

PROGRAMMING THE Z80

We must then wait for the first data bit. The necessary delay is equal to 9.09 milliseconds and is implemented by a subroutine:

```
CALL    DELAY9
```

Register B is used as a counter and is loaded with the value 8 in order to capture the 8 data bits:

```
LD     B, 08H
```

Next, each data bit will be read in turn into the accumulator, then echoed. It is assumed to arrive in bit position 0 of the accumulator. The data bit will then be preserved into register C, where it will be shifted in. The transfer from A to C is performed through the carry bit:

```
NEXT   IN    A, (TTYBIT)
       OUT   (TTYBIT), A
       SRL   A
       RR    C
```

This sequence is illustrated in Figure 6.15.

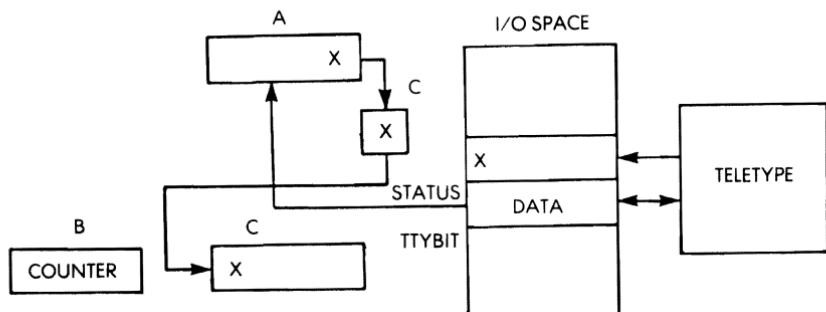


Fig. 6.15: Teletype Input

Next, the usual 9 millisecond delay is implemented, the bit-counter is decremented, and the loop is entered again as long as the eight bits have not been captured:

```
CALL  DELAY9
DEC   B
JR    NZ, NEXT
```

Finally, the STOP bit is captured, and echoed. It is usually sufficient to send a single STOP bit, however both could be sent back using two more instructions:

```
IN    A, (TTYBIT)
OUT  (TTYBIT), A
CALL DELAY9
RET
```

The program should be examined with attention. The logic is quite simple. The new fact is that whenever a bit is read from the Teletype (at address TTYBIT), it is echoed back to the Teletype. This is a standard feature of the Teletype. Whenever a user presses a key, the information is transmitted to the processor and then back to the printing mechanism of the Teletype. This verifies that the transmission lines are working and that the processor is operating when a character is, indeed, printing correctly on the paper.

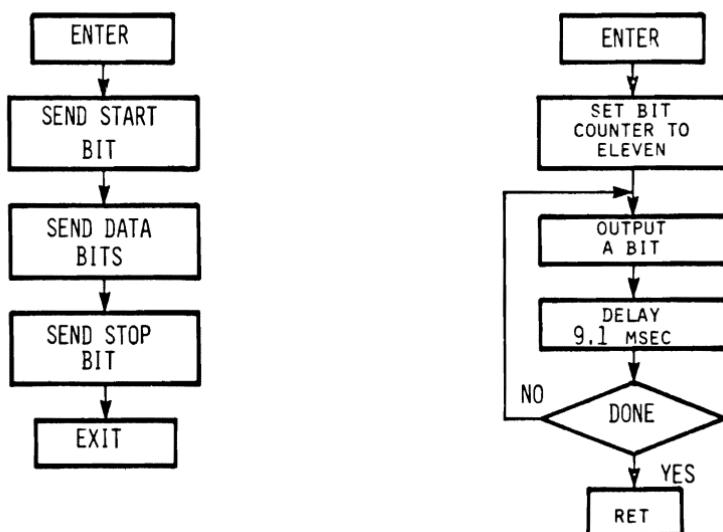


Fig. 6.16: Teletype Output

Exercise 6.25: Write the delay routine which results in the 9.09 millisecond delay. (*DELAY subroutine*)

Exercise 6.26: Using the example of the program developed above, write a *PRINTC* program which will print on the Teletype the contents of memory location *CHAR* (see Fig. 6.15).

The answer appears below:

| | | | |
|--------|-----|-----------|-------------------------|
| PRINTC | LD | B, 11 | COUNTER = 11 BITS |
| | LD | A, (CHAR) | GET CHARACTER |
| | OR | A | CLEAR CARRY = START BIT |
| | RLA | | CARRY INTO A |

PROGRAMMING THE Z80

| | | |
|------|-----------------|----------------------|
| NEXT | OUT (TTYBIT), A | OUTPUT |
| | CALL DELAY | |
| | RRA | NEXT BIT |
| | SCF | CARRY = 1 (STOP BIT) |
| | DEC B | BIT COUNT |
| | JR NZ, NEXT | |
| | RET | |

Register B is used as a bit counter for the transmission. The contents of bit 0 of A will be sent to the Teletype line ("TTYBIT"). Note how the carry is used to provide a ninth bit (the START bit). Also, note that the carry is cleared by:

OR A

At the end of the program, the carry is set to one by:

SCF

in order to generate a stop bit.

Exercise 6.27: Modify the program so that it waits for a START bit instead of a STATUS bit.

Printing a String of Characters

We will assume that the PRINTC routine (see Exercise 6.26) takes care of printing a character on our printer, or display, or any output device. We will here print the contents of memory locations (START) to (START + N).

The program is straightforward (see Figure 6.17):

| | | |
|---------|--------------|------------------|
| PSTRING | LD B, NBR | LENGTH OF STRING |
| | LD HL, START | BASE ADDRESS |
| NEXT | LD A, (HL) | GET CHARACTER |
| | CALL PRINTC | PRINT IT |
| | INC HL | NEXT ELEMENT |
| | DEC B | |
| | JR NZ, NEXT | DO IT AGAIN |
| | RET | |

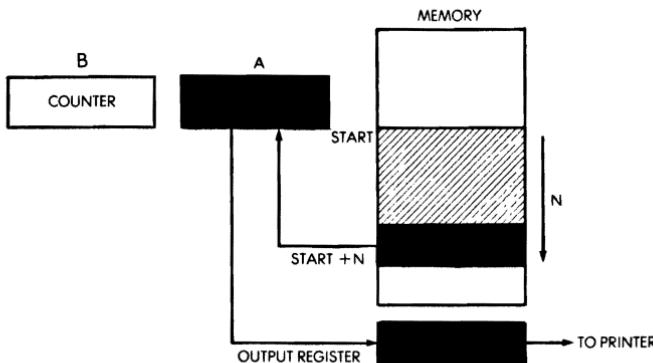


Fig. 6.17: Printing a Memory Block

PERIPHERAL SUMMARY

We have now described the basic programming techniques used to communicate with typical input/output devices. In addition to the data transfer, it will be necessary to condition one or more control registers within each I/O device in order to condition the transfer speeds, the interrupt mechanism, and the various other options correctly. The manual for each device should be consulted. (For more details on the specific algorithms for exchanging information with all the usual peripherals, the reader is referred to our book, C207, *Microprocessor Interfacing Techniques*.)

We have now learned to manage single devices. However, in a real system, all peripherals are connected to the buses, and may request service simultaneously. How are we going to schedule the processor's time?

INPUT/OUTPUT SCHEDULING

Since input/output requests may occur simultaneously, a scheduling mechanism must be implemented in every system to determine in which order service will be granted. Three basic input/output techniques are used, which can be combined with each other. They are: polling, interrupt, DMA. Polling and interrupts will be described here. DMA is purely a hardware technique, and as such will not be described here. (It is covered in the reference books C201 and C207.)

Polling

Conceptually, polling is the simplest method for managing multiple peripherals. With this strategy, the processor interrogates the devices connected to the buses in turn. If a device requests service, the service is granted. If it does not request service, the next peripheral is examined. Polling is used not just for the devices, but for *any device service routine*.

As an example, if the system is equipped with a Teletype, a tape recorder, and a CRT display, the polling routine would interrogate the Teletype: "Do you have a character to transmit?" It would interrogate the Teletype *output routine*, asking: "Do you have a character to send?" Then, assuming that the answers are negative so far, it would interrogate the tape-recorder routines, and finally the CRT display. If only one device is connected to a system, polling will be used as well to determine whether it needs service. As an example, the flowcharts for reading a paper-tape reader and for printing on a printer appear in Figures 6.20 and 6.21.

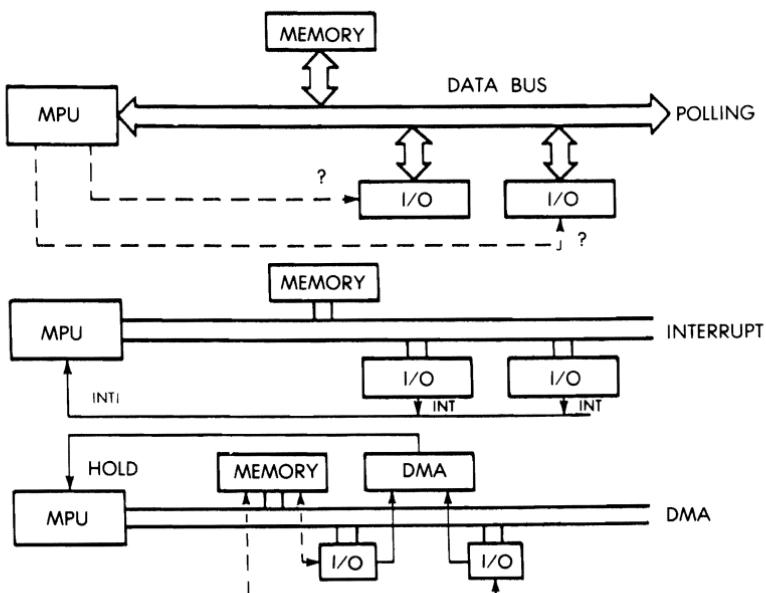


Fig. 6.18: Three Methods of I/O Control

Example: a polling loop for devices 1, 2, 3, 4 (see Fig. 6.19):

| | | | |
|-------|------|---------------|------------------------|
| POLL4 | IN | A, (STATUS 1) | GET STATUS OF DEVICE 1 |
| | BIT | 7, A | SERVICE REQUEST? |
| | CALL | NZ, ONE | BIT 7 = 1? |
| | IN | A, (STATUS2) | DEVICE 2 |
| | BIT | 7, A | |
| | CALL | NZ, TWO | |
| | IN | A, (STATUS3) | DEVICE 3 |
| | BIT | 7, A | |
| | CALL | NZ, THREE | |
| | IN | A, (STATUS4) | DEVICE 4 |
| | BIT | 7, A | |
| | CALL | NZ, FOUR | |
| | JR | POLL4 | NO REQUEST, TRY AGAIN |

Bit 7 of the status register for each device is “1” when it wants service. When a request is sensed, this program branches to the device handler, at address ONE for device 1, TWO for device 2, etc.

A fine point is worth noting here. For each instruction, it is important to verify carefully the way in which it affects the condition codes. It should be noted that the IN A instruction does not change the flags. If an IN r instruction has been used instead of an IN A instruction, bit 7 of the input would automatically be reflected as the SIGN bit in the flags register. The special instruction “BIT 7,A” would become unnecessary. However, because the IN A instruction does not change the flags, this extra test must be included in the program.

In some hardware implementations, input/output devices may be treated as memory devices for purposes of addressing. This is called memory-mapped input/output. In this case, the IN instruction would be replaced by an LD instruction and the rest of the program would be as above, since LD does not affect the flags.

The advantages of polling are obvious: it is simple, does not require any hardware assistance, and keeps all input/output synchronous with the program operation. Its disadvantage is just as obvious: most of the processor’s time is wasted looking at devices that do not need service. In addition, by wasting so much time, the processor might give service to a device too late.

Another mechanism is, therefore, desirable in order to guarantee that the processor’s time can be used to perform useful computations rather than polling devices needlessly all the time. However, let us stress that polling is used extensively whenever a microprocessor has nothing bet-

PROGRAMMING THE Z80

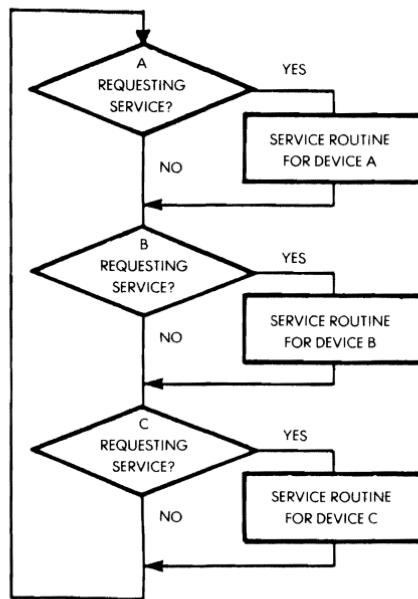


Fig. 6.19: Polling Loop Flowchart

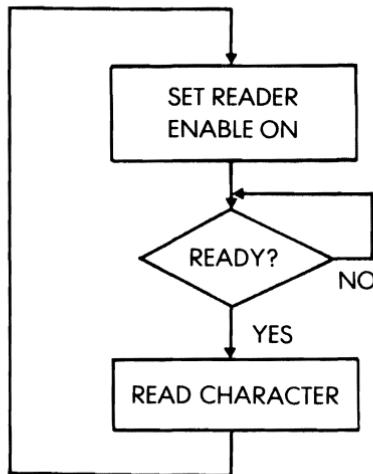


Fig. 6.20: Reading from a Paper-Tape Reader

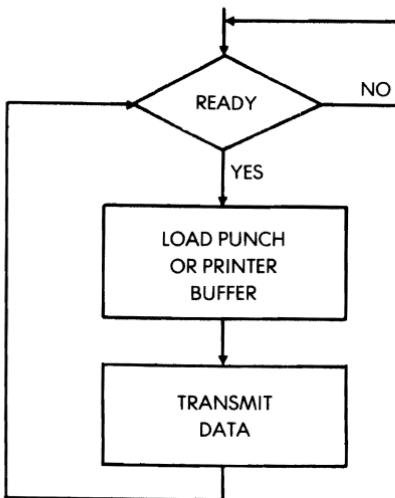


Fig. 6.21: Printing on a Punch or Printer

ter to do, as it keeps the overall organization simple. Let us examine the essential alternative to polling: interrupts.

Interrupts

The concept of interrupts is illustrated in Figure 6.18. A special hardware line, the interrupt line, is connected to a specialized pin of the microprocessor. Multiple input/output devices may be connected to this interrupt line. When any one of them needs service, it sends a level or a pulse on this line. An interrupt signal is the service request from an input/output device to the processor. Let us examine the response of the processor to this interrupt.

In any case, the processor completes the instruction that it was currently executing; otherwise, this would create chaos inside the microprocessor. Next, the microprocessor should branch to an interrupt-handling routine which will process the interrupt. Branching to such a subroutine implies that the contents of the program counter must be saved on the stack. *An interrupt must, therefore, cause the automatic preservation of the program counter on the stack.* In addition, the flag register F should be also preserved automatically, as its contents will be altered by any subsequent instruction. Finally, if the interrupt-handling

routine should modify any internal registers, these internal registers should also be preserved on the stack (see Figures 6.22 and 6.23).

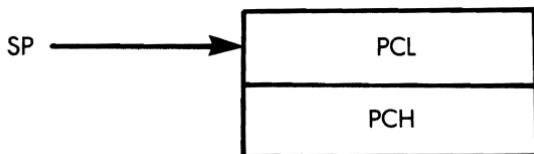


Fig. 6.22: Z80 Stack After Interruption

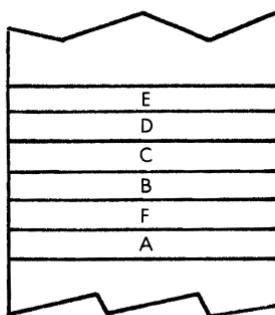


Fig. 6.23: Saving Some Working Registers

After all these registers have been preserved, one can branch to the appropriate interrupt-handling address. At the end of this routine, all the registers should be restored, and a special interrupt return should be executed so that the main program will resume execution. Let us examine in more detail the interrupt lines of the Z80.

Z80 Interrupts

An interrupt is a signal sent to the microprocessor, which may request service at any time and is asynchronous to the program. Whenever a program branches to a subroutine, such branching is *synchronous* to program execution, i.e., scheduled by the program. An interrupt, however, may occur at any time, and will generally suspend the execution of the current program (without the program knowing it). Because it may happen at any time relative to program execution, it is called *asynchronous*.

Three interruption mechanisms are provided on the Z80: the bus request (BUSRQ), the non-maskable interrupt (NMI) and the usual interrupt (INT).

Let us examine these three types.

The Bus Request

The bus request is the highest priority interrupt mechanism on the Z80. The interrupt sequence for the Z80 is shown in Figure 6.24. As a general rule, no interrupt will be sensed by the Z80 until the current machine cycle is completed. The NMI and INT interrupts will not be taken into account until the current instruction is finished. However, the BUSRQ will be handled at the end of the current machine cycle, without necessarily waiting for the end of the instruction. It is used for

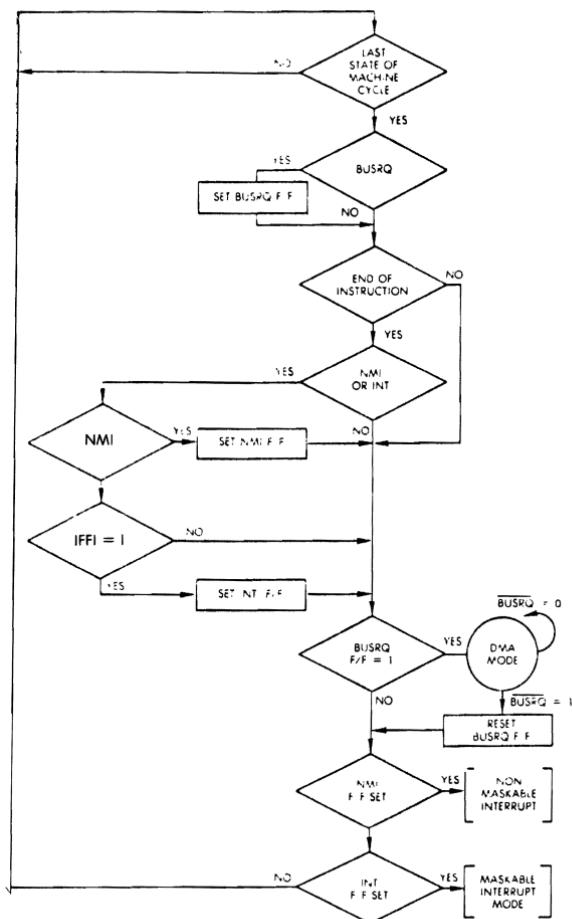


Fig. 6.24: Interrupt Sequence

a direct memory access (DMA), and will cause the Z80 to go into DMA mode (see ref. C201 for an explanation of the DMA mechanism). If the end of an instruction has been reached, and if any NMI or INT were pending, they would be memorized internally in the Z80 by setting specialized flip-flops: the NMI flip-flop, and the INT flip-flop. In DMA mode, the Z80 suspends operation and releases its data-bus and address-bus in the high-impedance state. This mode is normally used by a DMA controller to perform transfers between a high-speed input-output device and the memory, using the microprocessor data-bus and address-bus. The end of a DMA operation is indicated to the Z80 by BUSRQ changing levels. At this point, the Z80 will resume normal operation. In particular, it will first check whether its internal NMI or INT flip-flops had been set and, if so, execute the corresponding interrupts.

The DMA should normally not be of concern to the programmer, unless timing is important. If a DMA controller is present in the system, the programmer must understand that the DMA may delay the response to an NMI or an INT.

The Non-Maskable Interrupt

This type of interrupt cannot be inhibited by the programmer. It is therefore said to be *non-maskable*, hence its name. It will always be accepted by the Z80 upon completion of the current instruction, assuming no bus request was received. (If an NMI is received during a BUSRQ, it will set the internal NMI flip-flop, and will be processed at the end of the instruction following the end of the BUSRQ.)

The NMI will cause an automatic push of the program counter into the stack and branch to address 0066H: the two bytes representing the address 0066H will be installed in the program counter. They represent the start address of the handling routine for the NMI (see figure 6.25).

This interrupt mechanism has been designed for speed, as it is used in case of "emergencies". Therefore, it does not offer the flexibility of the maskable interrupt mode, described below.

Note also that an interrupt routine must have been loaded at address 0066H prior to using the NMI.

NMI will first cause:

$$\left. \begin{array}{l} SP \leftarrow SP - 1 \\ (SP) \leftarrow PCH \\ SP \leftarrow SP - 1 \\ (SP) \leftarrow PCL \end{array} \right\} \text{push PC}$$

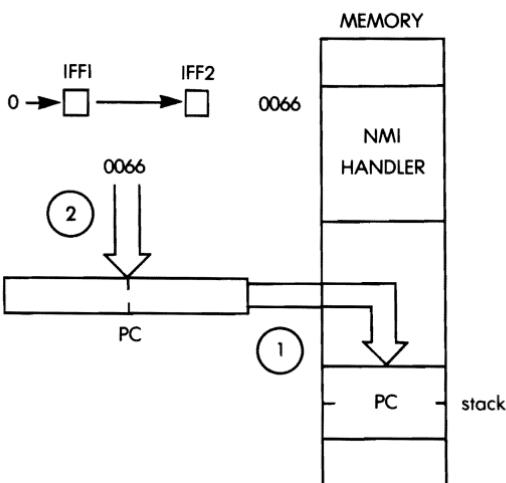


Fig. 6.25: NMI Forces Automatic Vectoring

Then, NMI causes an automatic restart at location 0066H. The complete sequence of events is the following:

| | | | |
|---------------|---|-------|-----------------------------|
| PC | → | STACK | (preserve program counter) |
| IFF1 | → | IFF2 | (preserve IFF) |
| 0 | → | IFF1 | (reset IFF) |
| JUMP TO 0066H | | | (execute interrupt handler) |

Also, the status of interrupt-mask-bit flip-flop (IFF1) at the time that NMI was received is preserved automatically into IFF2. Then, IFF1 is reset in order to prevent any further interrupts. This feature is important to prevent the loss of lower-priority INT's and simplifies the external hardware: the status of a pending INT is preserved internally in the Z80.

The NMI interrupt is normally used for high priority events such as a real-time clock or a power failure.

The return from an NMI is accomplished by a special instruction, RETN: "return from non-maskable interrupt." The contents of IFF1 are restored from IFF2, and the contents of the program counter PC are restored from their location in the stack. Since IFF1 had been reset during execution of the NMI, no external INT's could be accepted during the NMI (unless the programmer uses an EI instruction within the NMI routine): there has been no loss of information.

Upon termination of the interrupt handler, the sequence is:

| | | | |
|-------|---|------|---------------------------|
| IFF2 | → | IFF1 | (restore IFF) |
| STACK | → | PC | (restore program counter) |

Note that, once IFF1 is restored, maskable interrupt enable status is restored.

Interrupt

The ordinary, maskable, interrupt INT may operate in one of three modes. They are specific to the Z80, as the 8080 is equipped with only a single interrupt mode. The ordinary interrupt INT may also be masked selectively by the programmer. Setting the interrupt flip-flops IFF1 and IFF2 to a "1" will authorize interruptions. Setting them to a "0" (masking them) will prevent detection of INT. The EI instruction is used to set them, and the DI instruction is used to reset them. IFF1 and IFF2 are set or reset simultaneously. During execution of the EI and DI instructions, INT's are disabled in order to prevent any loss of information.

Let us now examine the three interrupt modes:

Interrupt Mode 0

This mode is identical to the 8080 interrupt mode. The Z80 will operate in interrupt mode 0 either when initially started (when the RESET signal has been applied) or else when an IM0 instruction has been executed. Once mode 0 has been set, an interrupt will be recognized if the interrupt enable flip-flop IFF1 is set to 1, provided no bus-request or non-maskable interrupt occurs at the same time. The interrupt will be detected only at the end of an instruction. Essentially, the Z80 will respond to the interrupt by generating an IORQ (and an M1 signal), and then do nothing, except wait.

It is the responsibility of an *external device* to recognize the IORQ and M1 (this is called an *interrupt acknowledge* or INTA) and to place an instruction on the data-bus. The Z80 expects an instruction to be placed on its data bus by the external device within the next cycle. Typically, an RST or a CALL instruction is placed on the bus. Both of these instructions automatically preserve the program-counter in the stack, and cause branching to a specific address. The advantage of the RST instruction is that it resides within a single byte, i.e., it executes rapidly. Its disadvantage is to branch to only one of eight possible locations in page zero (addresses 0 through 255). The advantage of the CALL instruction is that it is a general-purpose branch instruction which specifies a full 16-bit address. However, it requires three bytes and therefore executes less rapidly.

Note that once the interrupt processing starts, all further interrupts are disabled. IFF1 and IFF2 are automatically set to "0". It is then the responsibility of the programmer to insert an EI instruction (Enable In-

terrupts) at the appropriate location within his program if he wishes to enable interrupts, and, in any case, before returning from the interrupt.

The detailed sequence corresponding to the mode 0 interrupt is shown in Figure 6.26.

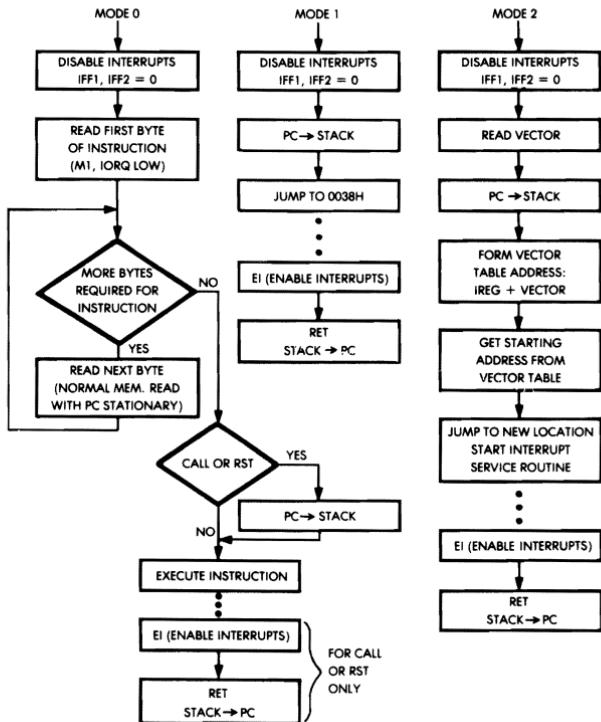


Fig. 6.26: Interrupt Modes

The return from the interrupt is accomplished by an RETI instruction. Let us remind the programmer at this point that he/she is usually responsible for explicitly clearing the interrupt which has been serviced on the I/O device, and always for restoring the interrupt disable flag inside the Z80. However, the peripheral controller may use the INTA signal to clear the INT request, thus freeing the programmer of this chore.

In addition, should the interrupt-handling routine modify the contents of any of the internal registers, the programmer is specifically responsible for preserving these registers in the stack prior to executing the interrupt-handling routine. Otherwise, the contents of these registers will be destroyed, and when the interrupted program resumes exe-

PROGRAMMING THE Z80

cution, it will fail. For example, assuming that registers A, B, C, D, E, H and L will be used within the interrupt handler, they will have to be saved (see Figure 6.27).

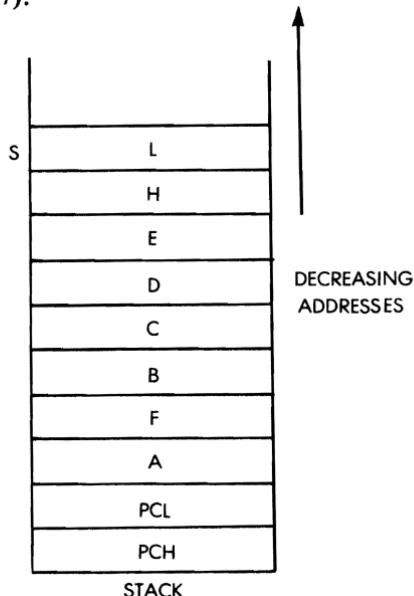


Fig. 6.27: Saving the Registers

The corresponding program is:

```
SAVREG    PUSH  AF  
          PUSH  BC  
          PUSH  DE  
          PUSH  HL
```

Upon completion of the interrupt-handling routine, these registers must be restored. The interrupt handler will terminate with the following sequence of instructions:

```
POP  HL  
POP  DE  
POP  BC  
POP  AF  
EI
```

(unless EI was used earlier in the routine)

Additionally, if registers IX and IY are used by the routine they must also be preserved, then restored.

Interrupt Mode 1

This interrupt mode is set by executing the IM1 instruction. It is an automated interrupt handler which causes an automatic branch to location 0038H. It is therefore essentially analogous to the NMI interrupt mechanism except that it may be masked. The Z80 automatically preserves the contents of PC into the stack (see Figure 6.28).

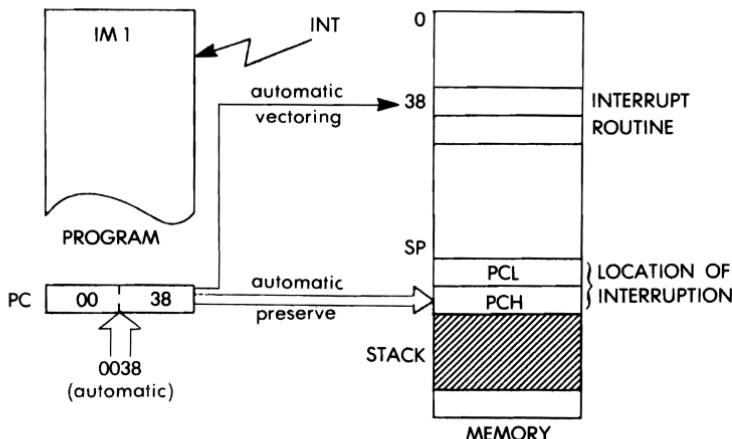


Fig. 6.28: Mode 1 Interrupt

This automated interrupt response, which “vectors” all interrupts to memory location 38H, stems from the early 8080’s requirement to minimize the amount of external hardware necessary for using interrupts. Its possible disadvantage is to cause a branch to a *single* memory location. In case several devices are connected to the INT line, the program starting at location 38H will be responsible for determining which device requested service. This problem will be addressed below.

One precaution must be taken with respect to the timing of this interrupt: when performing programmed input/output transfers, the Z80 will ignore any data that may be present in the data bus during the cycle which follows the interrupt (the interrupt acknowledge cycle).

Interrupt Mode 2 (Vectored Interrupts)

This mode is set by executing an IM2 instruction. It is a powerful mode which allows automatic vectoring of interrupts. The interrupt vector is an address supplied by the peripheral device which generated the interrupt, and used as a memory pointer to the start address of the interrupt-handling routine. The addressing mechanism provided by the Z80 in mode 2 is indirect, rather than direct. Each peripheral supplies a seven-bit branching address which is appended to the 8-bit address contained in the special I register in the Z80. The right-most bit of the final 16-bit address bit 0 is set to "0". This resulting address points to an entry in a table anywhere in the memory. This table may contain up to 128 double-word entries. Each of these double words is the address of the interrupt handler for the corresponding device. This is illustrated in Figures 6.29 and 6.30.

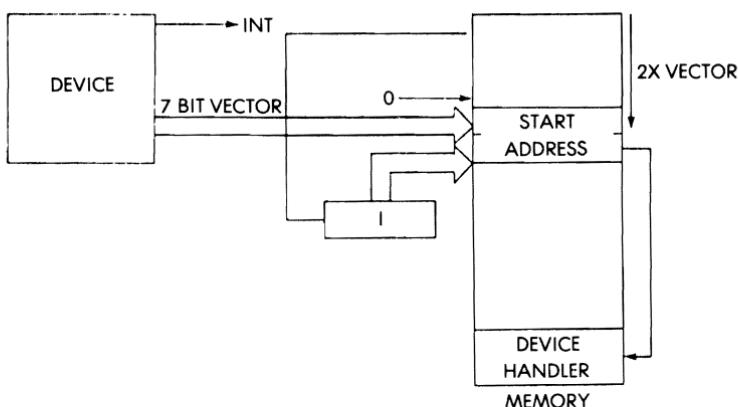


Fig. 6.29: Mode 2 Interrupt

The interrupt table may have up to 128 double-word entries.

In this mode, the Z80 also automatically pushes the contents of the program counter into the stack. This is obviously necessary, since PC will be reloaded with the contents of the interrupt table entry corresponding to the vector provided by the device.

Interrupt Overhead

For a graphic comparison of the polling process vs. the interrupt process, refer to Figure 6.18, where the polling process is illustrated on the top, and the interrupt process underneath. It can be seen that in the polling technique the program wastes a lot of time waiting.

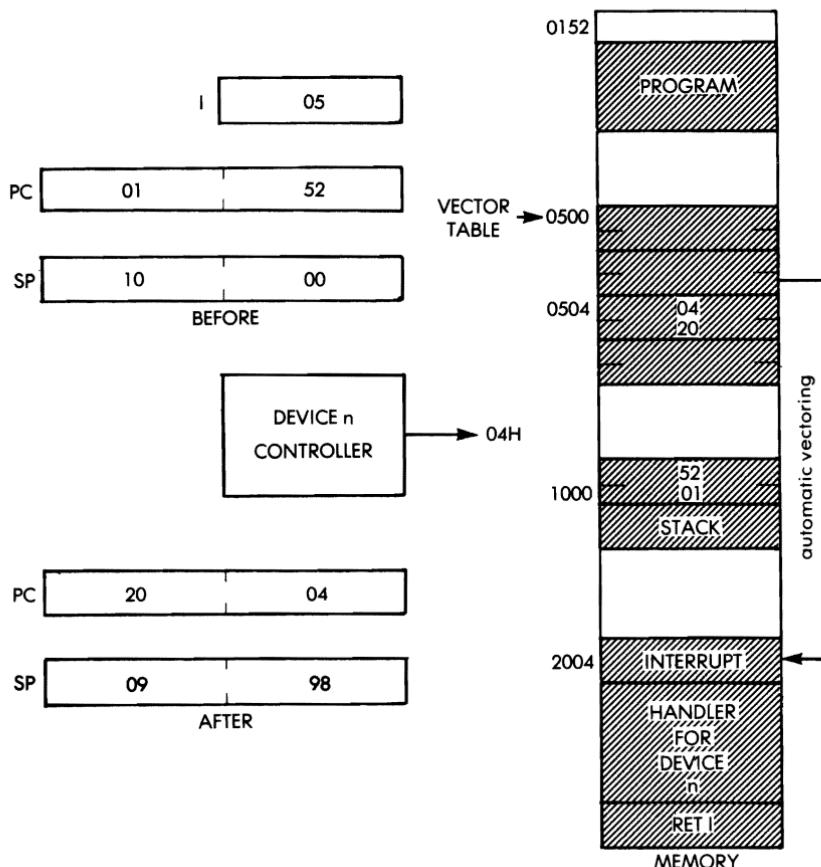


Fig. 6.30: Mode 2.- A Practical Example

Using interrupts, the program is interrupted, the interrupt is serviced, then the program resumes. However, the obvious disadvantage of an interrupt is to introduce several additional instructions at the beginning and at the end, resulting in a delay before the first instruction of the device handler can be executed. This is additional overhead.

Exercise 6.28: Using the tables indicating the number of cycles per instruction, in Chapter 4, compute how much time will be lost to save and then restore registers A, B, D, H.

Having clarified the operation of the interrupt lines, let us now consider two important remaining problems:

- 1—How do we resolve the problem of multiple devices triggering an

interrupt at the same time?

2—How do we resolve the problem of an interrupt occurring while another interrupt is being serviced?

Multiple Devices Connected to a Single Interrupt Line

Whenever an interrupt occurs, the processor branches to a specified address. Before it can do any effective processing, the interrupt handling routine must determine which device triggered the interrupt. Two methods are available to identify the device, as usual: a software method and a hardware method.

In the software method, polling is used: the microprocessor interrogates each of the devices in turn and asks them, "Did you trigger the interrupt?" If the answer is negative, it interrogates the next one. This process is illustrated in Figure 6.31. A sample program is:

```
POLINT IN A, (STATUS1) READ STATUS
        BIT 7, A DID DEVICE REQUEST INT?
        JP   NZ, ONE HANDLE IT IF SO
        IN   A, (STATUS2)
        BIT 7, A
        JP   NZ, TWO
etc.    ---
```

The hardware method provides the address of the interrupting device simultaneously with the interrupt request.

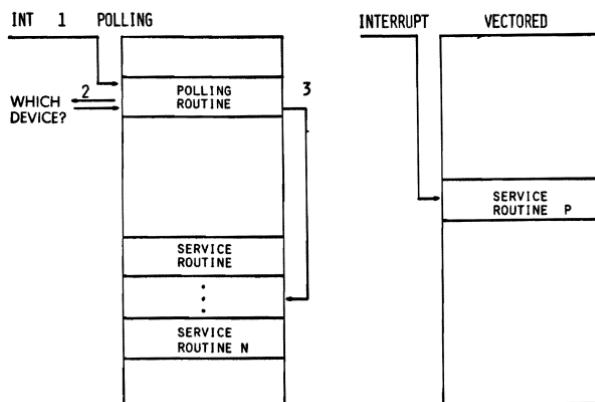


Fig. 6.31: Polled vs. Vectored Interrupt

To be more precise, when operating in mode 0, the peripheral device controller will supply a one-byte RST or a three-byte CALL on the data bus in response to the interrupt acknowledge, thus automating the interrupt vectoring, and minimizing the overhead.

Note that a subroutine call instruction is required as the Z80 does not save the PC when operating in mode 0.

In most cases, the speed of reaction to an interrupt is not crucial, and a polling approach is used. If response time is a primary consideration, a hardware approach must be used.

Simultaneous Interrupts

The next problem which may occur is that a new interrupt can be triggered during the execution of an interrupt-handling routine. Let us examine what happens and how the stack is used to solve the problem. We have indicated in Chapter 2 that this was another essential role of the stack, and the time has come now to demonstrate its use. We will refer to Figure 6.33 to illustrate multiple interrupts. Time elapses from left to right in the illustration. The contents of the stack are shown at the bottom of the illustration. Looking at the left, at time T0, program P is in execution. Moving to the right, at time T1, interrupt I1 occurs. We will assume that the interrupt mask was enabled, authorizing I1. Program P will be suspended. This is shown at the bottom of the illustration. The stack will contain the program counter and the status register of program P, at least, plus any optional registers that might be saved by the interrupt handler or I1 itself.

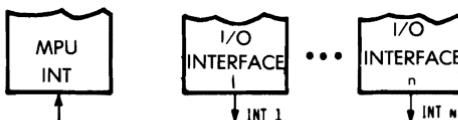


Fig. 6.32: Several Devices May Use the Same Interrupt Line

At time T1, interrupt I1 starts executing until time T2. At time T2, interrupt I2 occurs. We will assume that interrupt I2 has a higher priority than interrupt I1. If it had a lower priority, it would be ignored until I1 had been completed. At time T2, the registers for I1 are stacked, and this appears at the bottom of the illustration. Again, the contents of the program counter and AF are pushed into the stack. In addition, the routine for I2 might decide to save an additional few registers. I2 will now execute to completion at time T3.

PROGRAMMING THE Z80

When 12 terminates (with an RETI), the contents of the stack are automatically popped back into the Z80, and this is illustrated at the bottom of Figure 6.33. Thus, automatically I1 resumes execution. Unfortunately, at time T4, an interrupt 13 of higher priority occurs again. We can see at the bottom of the illustration that again the registers for I1 are pushed into the stack. Interrupt 13 executes from T4 to T5 and

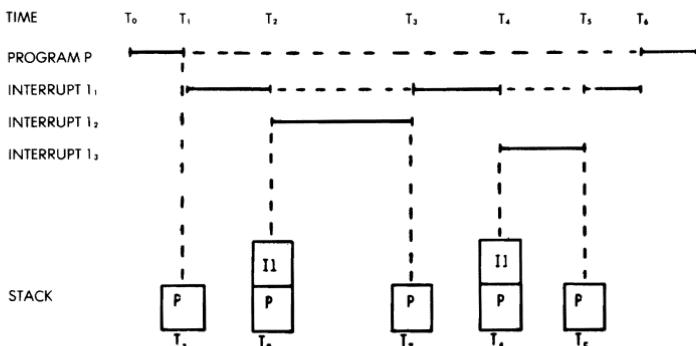


Fig. 6.33: Stack Contents During Multiple Interrupts

terminates at T₅. At that time, the contents of the stack are popped into Z80, and interrupt I₁ resumes execution. This time it runs to completion and terminates at T₆. At T₆, the remaining registers that have been saved in the stack are popped into Z80, and program P may resume execution. The reader will verify that the stack is empty at this point. In fact, the number of dashed lines indicating program suspension indicates at the same time how many levels there are in the stack.

Exercise 6.29: Assume that the area available to the stack is limited to 300 locations in a specific program. Assume that all the registers must always be saved and that the programmer allows interrupts to be nested, i.e., to interrupt each other. Which is the maximum number of simultaneous interrupts that can be handled? Will any other factor contribute to still reduce further the maximum number of simultaneous interrupts?

It must be stressed, however, that, in practice, microprocessor systems are normally connected to a small number of devices using interrupts. It is, therefore, unlikely that a high number of simultaneous interrupts will occur in such a system.

We have now solved all the problems usually associated with interrupts. Their use is, in fact, simple and they should be employed to advantage even by the novice programmer.

SUMMARY

In this chapter we have presented the range of techniques used to communicate with the outside world. From elementary input/output routines to more complex programs for communication with actual peripherals, we have learned to develop all the usual programs and have even examined the efficiency of benchmark programs in the case of a parallel transfer and a parallel-to-serial conversion. Finally, we have learned to schedule the operation of multiple peripherals by using polling and interrupts. Naturally, many other exotic input/output devices might be connected to a system. With the array of techniques which have been presented so far, and with an understanding of the peripherals involved, it should be possible to solve most common problems.

In the next chapter, we will examine the actual characteristics of the input/output interface chips usually connected to a Z80. Then, we will consider the basic data structures that the programmer may use.

Exercise 6.30: *Compute the overhead when operating in mode 0, assuming that all registers are saved, and that an RST is received in response to the interrupt acknowledge. The overhead is defined as the total delay incurred, exclusive of the instructions required to implement the interrupt processing proper.*

Exercise 6.31: *A 7-segment LED display can also display digits other than the hex alphabet. Compute the codes for: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.*

Exercise 6.32: *The flowchart for interrupt management appears in Figure 6.34. Answer the following questions:*

- a—*What is done by hardware, what is done by software?*
- b—*What is the use of the mask?*
- c—*How many registers should be preserved?*
- d—*How is the interrupting device identified?*
- e—*What does the RETI instruction do? How does it differ from a subroutine return?*
- f—*Suggest a way to handle a stack overflow situation.*
- g—*What is the overhead ("lost time") introduced by the interrupt mechanism?*

PROGRAMMING THE Z80

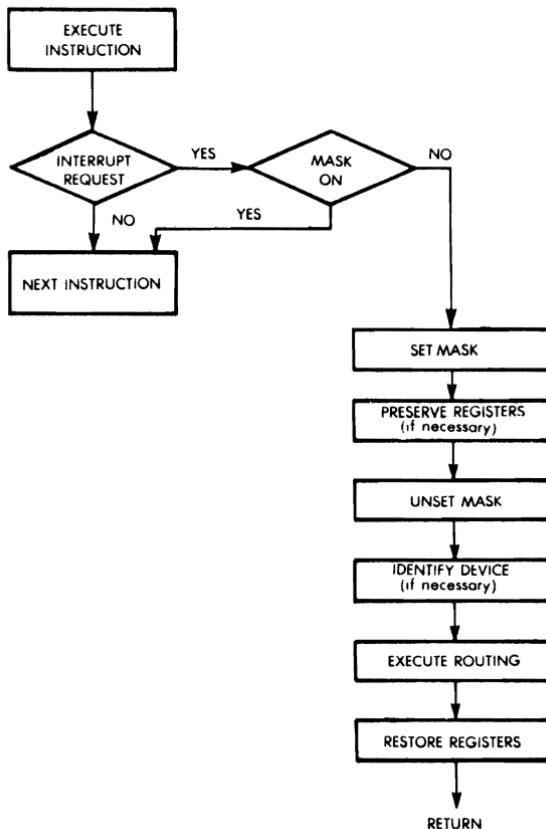


Fig. 6.34: Interrupt Logic

INPUT/OUTPUT DEVICES

INTRODUCTION

We have learned how to program the Z80 microprocessor in most usual situations. However, we should make a special mention of the input/output chips normally connected to the microprocessor. Because of the progress in LSI integration, new chips have been introduced which did not exist before. As a result, programming a system requires, naturally, first to program a microprocessor itself, and then *to program the input/output chips*. In fact, it is often more difficult to remember how to program the various control options of an input/output chip than to program the microprocessor itself! This is not because the programming in itself is more difficult, but because each of these devices has its own idiosyncrasies. We are going to examine here first the most general input/output device, the programmable input/output chip (in short a “PIO”), then some Zilog I/O devices.

The “Standard PIO”

There is no “standard PIO”. However, each PIO device is essentially analogous in function to all similar PIO’s produced by other manufacturers for the same purpose. The purpose of a PIO is to provide a multiport connection for input/output devices. (A “port” is simply a set of 8 input/output lines.) Each PIO provides at least two sets of 8-bit lines for I/O devices. Each I/O device needs a *data buffer* in order to stabilize the contents of the data bus on output at least. Our PIO will, therefore, be equipped at a minimum with a buffer for each port.

In addition, we have established that the microcomputer will use a *handshaking* procedure, or else *interrupts* to communicate with the

PROGRAMMING THE Z80

I/O device. The PIO will also use a similar procedure to communicate with the peripheral. Each PIO must, therefore, be equipped with at least *two control lines per port* to implement the handshaking function.

The microprocessor will also need to be able to read the status of each port. Each port must be equipped with one or more *status bits*. Finally, a number of options will exist within each PIO to configure its resources. The programmer must be able to access a special register within the PIO to specify the programming options. This is the *control-register*. In some cases the status information is part of the control register.

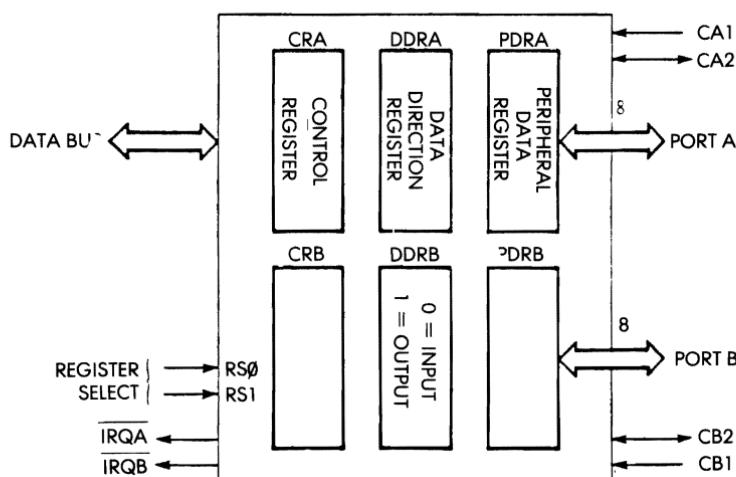


Fig. 7.1: Typical PIO

One essential faculty of the PIO is the fact that each line may be configured as either an input or an output line. The diagram of a PIO appears in illustration 7.1. The programmer may specify whether any line will be input or output. In order to program the direction of the lines, a *data-direction register* is provided for each port. On many PIO's, "0" in a bit position of the data-direction register specifies an input. A "1" specifies an output. Zilog uses the reverse convention.

It may be surprising to see that a "0" is used for input and a "1" for output when really "0" should correspond to output and "1" to input. This is quite deliberate: whenever power is applied to the system, it is of great importance that all the I/O lines be configured as *input*. Otherwise, if the microcomputer is connected to some

dangerous peripheral, it might activate it by accident. When a reset is applied, all registers are normally zeroed and that will result in configuring all input lines of the PIO as inputs. The connection to the microprocessor appears on the left of the illustration. The PIO naturally connects to the 8-bit data bus, the microprocessor address bus, and the microprocessor control-bus. The programmer will simply specify the address of any register that it wishes to access within the PIO.

The Internal Control Register

The Control Register of the PIO provides a number of options for generating or sensing interrupts, or for implementing automatic hand-shake functions. The complete description of the facilities provided is not necessary here. Simply, the user of any practical system which uses a PIO will have to refer to the data-sheet showing the effect of setting the various bits of the control register. Whenever the system is initialized, the programmer will have to load the control register of the PIO with the correct contents for the expected application.

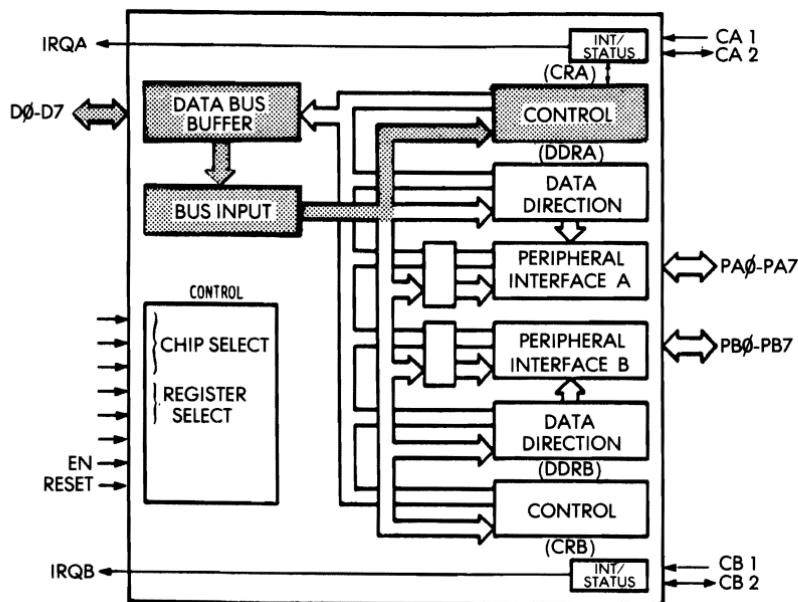


Fig. 7.2: Using a PIO-Load Control Register

PROGRAMMING THE Z80

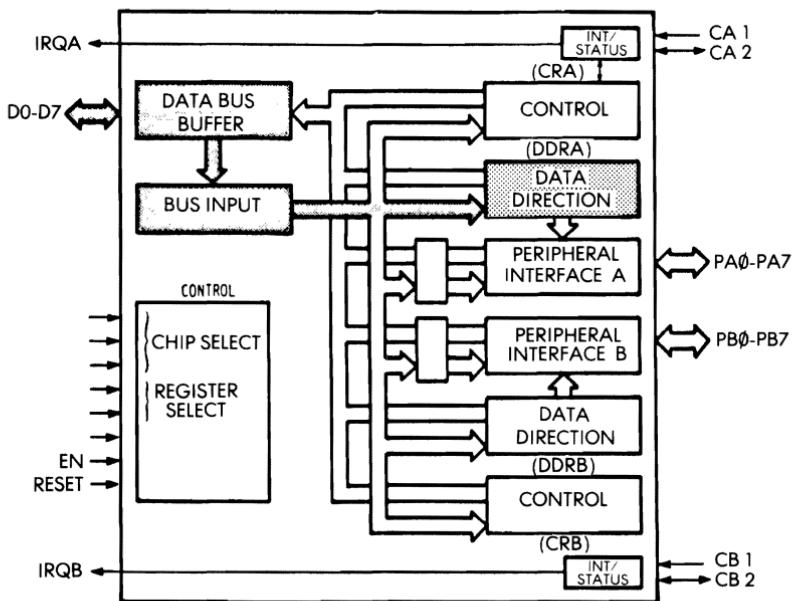


Fig. 7.3: Using a PIO-Load Data Direction

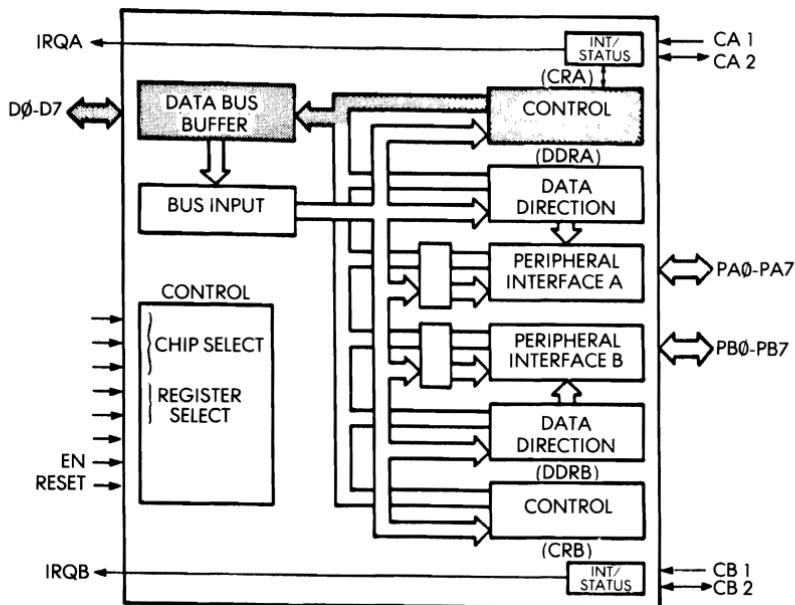


Fig. 7.4: Using a PIO-Read Status

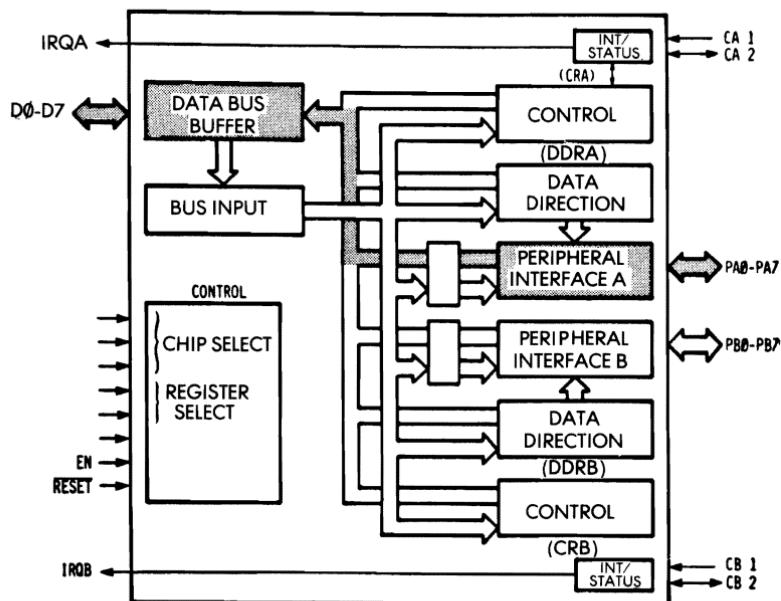


Fig. 7.5: Using a PIO Read INPUT

Programming a PIO

A typical sequence, when using a PIO channel, is the following (assuming an input):

Load the control register

This is accomplished by a programmed transfer between a Z80 register (usually the accumulator) and the PIO control register. This sets the options and operating mode of the PIO (see Figure 7.2). It is normally done only once at the beginning of a program.

Load the direction register

This specifies the direction in which the I/O lines will be used. (See Figure 7.3.)

Read the status

The status register indicates whether a valid byte is available on input. (See Figure 7.4).

Read the port

The byte is read into the Z80. (See Figure 7.5).

PROGRAMMING THE Z80

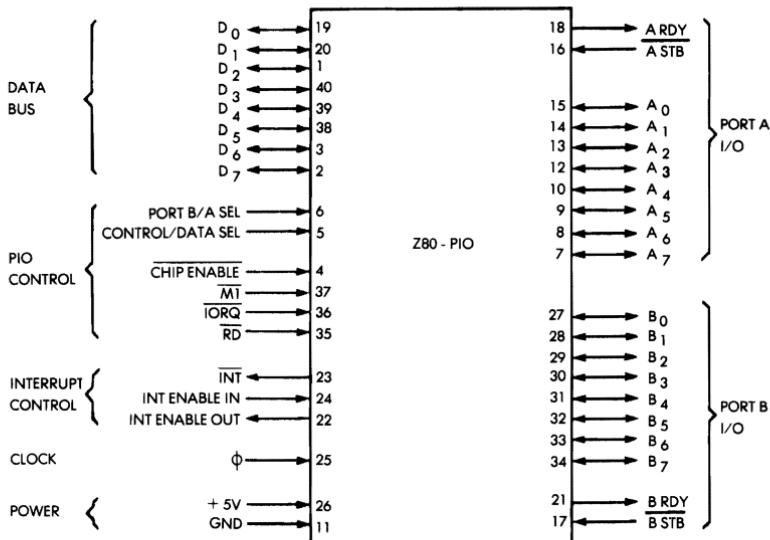


Fig. 7.6: Z80 PIO pinout

The Zilog Z80 PIO

The Z80 PIO is a two-port PIO whose architecture is essentially compatible with the standard model we have described. The actual pinout is shown in Figure 7.6, and a block diagram is shown in Figure 7.7.

Each PIO port has six registers: an 8-bit input register, an 8-bit output register, a 2-bit mode-control register, an 8-bit mask register, an 8-bit input/output select (direction register), and a 2-bit mask-control register. The last three registers are used only when the port is programmed to operate in the bit mode.

Each port may operate in one of four modes, as selected by the contents of the mode-control registers (2 bits). They are: byte output, byte input, byte bidirectional bus, and bit mode.

The two bits of the mask control register are loaded by the programmer, and specify the high or low state of a peripheral device which is to be monitored, and conditions for which an interrupt can be generated. generated.

The 8-bit input/output select register allows any pin to be either an input or an output when operating in the bit mode.

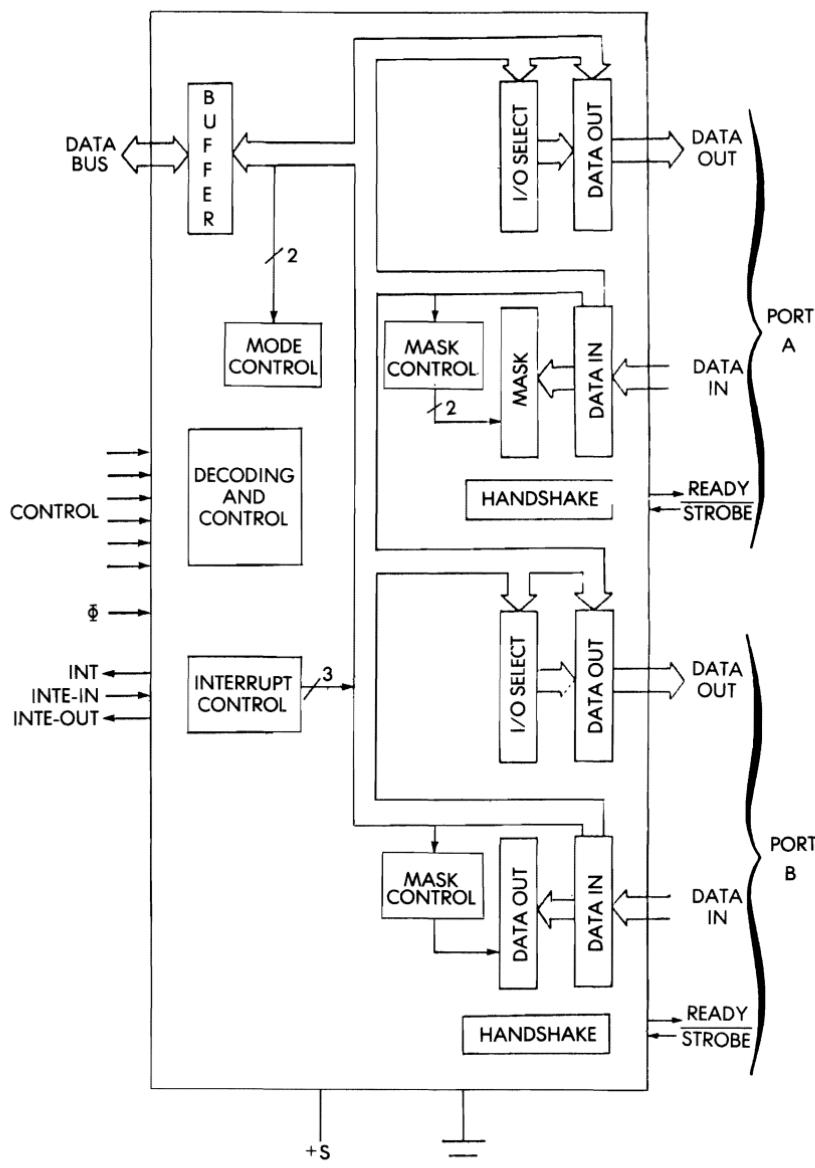


Fig. 7.7: Z80 PIO Block Diagram

Programming the Zilog PIO

A typical sequence for using a PIO, say in bit mode, would be the following:

Load the mode control register to specify the bit mode.

Load the input/output select register of port A to specify that lines 0-5 are inputs and lines 6 and 7 are outputs.

Then a word would be read by reading the contents of the input buffer.

Additionally, the mask register could be used to specify the status conditions.

For a detailed description of the operation of the PIO, the reader is referred to the companion volume in this series, the *Z80 Applications Book*.

The Z80 SIO

The SIO (Serial Input/Output) is a dual-channel peripheral chip designed to facilitate asynchronous communications in serial form. It includes a UART, i.e., a universal asynchronous receiver-transmitter. Its essential function is serial-to-parallel and parallel-to-serial conversion. However, this chip is equipped with sophisticated capabilities, like automatic handling of complex byte-oriented protocols, such as IBM bisync as well as HDLC and SDLC, two bit-oriented protocols.

Additionally, it can operate in synchronous mode like a USRT, and generate and check CRC codes. It offers a choice of polling, interrupt, and block-transfer modes. The complete description of this device is beyond the scope of this introductory book and appears in the *Z80 Applications Book*.

Other I/O Chips

Because the Z80 is commonly used as a replacement for the 8080, it has been designed so that it can be associated with almost any of the usual 8080 input/output chips, as well as the specific I/O chips manufactured by Zilog. All the 8080 input/output chips may be considered for use in a Z80 system.

SUMMARY

In order to make effective use of input/output components it is necessary to understand in detail the function of every bit, or group of bits, within the various control registers. These complex new chips automate a number of procedures that had to be carried out by software or special logic before. In particular, a good deal of the handshaking procedures are automated within components such as an SIO. Also, interrupt handling and detection may be internal. With the information that has been presented in the preceding chapter, the reader should be able to understand what the functions of the basic signals and registers are. Naturally, still newer components are going to be introduced which will offer a hardware implementation of still more complex algorithms.

8

APPLICATION EXAMPLES

INTRODUCTION

This chapter is designed to test your new programming skills by presenting a collection of utility programs. These programs or “routines” are frequently encountered in applications, and are generally called “utility routines.” They will require a synthesis of the knowledge and techniques presented so far.

We are going to fetch characters from an I/O device and process them in various ways. But first, let us clear an area of the memory (this may not be necessary—each of these programs is only presented as a programming example).

CLEARING A SECTION OF MEMORY

We want to clear (zero) the contents of the memory from address **BASE** to address **BASE ± LENGTH**, where **LENGTH** is less than 256.

The program is:

| | | | |
|-------|-----|-----------|--------------------|
| ZEROM | LD | B, LENGTH | LOAD B WITH LENGTH |
| | LD | A,0 | CLEAR A |
| | LD | HL, BASE | POINT TO BASE |
| CLEAR | LD | (HL), A | CLEAR A LOCATION |
| | INC | HL | POINT TO NEXT |
| | DEC | B | DECREMENT COUNTER |
| | JR | NZ, CLEAR | END OF SECTION? |
| | RET | | |

In the above program, the length of the section of memory is assumed to be equal to LENGTH. The register pair HL is used as a pointer to the current word which will be cleared. Register B is used, as

usual, as a counter.

The accumulator A is loaded only once with the value 0 (all zeros), then copied into the successive memory locations.

In a memory test program, for example, this utility routine could be used to zero the contents of a block. Then the memory test program would usually verify that its contents remained 0.

The above was a straightforward implementation of a clearing routine. Let us improve on it.

The improved program appears below.

```

ZERO M LD B, LENGTH
      LD HL, BASE
LOOP   LD (HL), 0
      INC HL
      DJNZ LOOP
      RET

```

The two improvements were obtained by eliminating the LD A, 0 instruction and loading a “zero” directly into the location pointed to by H and L, and also by using the special Z80 instruction DJNZ.

This improvement example should demonstrate that *every time a program is written, even though it may be correct, it can usually be improved by examining it carefully*. Familiarity with the complete instruction set is essential for bringing about such improvements. These improvements are not just cosmetic. They improve the execution time of the program, require fewer instructions and therefore less memory space, and also generally improve the readability of the program and, therefore, its chances of being correct.

Exercise 8.1: Write a memory test program which zeroes a 256-word block, then verifies that each location is 0. Then, it will write all 1's and verify the contents of the block. Then it will write 01010101 and verify the contents. Finally, it will write 10101010, and verify the contents.

Exercise 8.2: Modify the above program so that it will fill the memory section with alternating 0's and 1's (all 0's, then all 1's).

Let us now poll our I/O devices to find which one needs service.

POLLING I/O DEVICES

We will assume that those I/O devices are connected to our system. Their status registers are located at addresses STATUS1, STATUS2, STATUS3. The program is:

```

TEST      IN     A, (STATUS1) READ IO STATUS1
          BIT    7, A      TEST "READY" BIT (BIT 7)
          JP     NZ, FOUND1 JUMP TO HANDLER 1
          IN     A, (STATUS2) SAME FOR DEVICE 2
          BIT    7, A
          JP     NZ, FOUND2
          IN     A, (STATUS3) SAME FOR DEVICE 3
          BIT    7, A
          JP     NZ, FOUND3
          (failure exit)

```

As a result of the BIT instruction, the Z bit of the status flags will be set to 1 if STATUS is zero. The JP NZ instruction (jump if non-equal to zero) will then result in a branch to the appropriate FOUND routine.

GETTING CHARACTERS IN

Assume we have just found that a character is ready at the keyboard. Let us accumulate characters in a memory area called BUFFER until we encounter a special character called SPC, whose code has been previously defined.

The subroutine GETCHAR will fetch one character from the keyboard (see Chapter 6 for more details) and leave it in the accumulator. We assume that 256 characters maximum will be fetched before an SPC character is found.

| | | | |
|--------|------|------------|------------------------|
| STRING | LD | HL, BUFFER | POINT TO BUFFER |
| NEXT | CALL | GETCHAR | GET A CHARACTER |
| | CP | SPC | CHECK FOR SPECIAL CHAR |
| | JR | Z, OUT | FOUND IT? |
| | LD | (HL), A | STORE CHAR IN BUFFER |
| | INC | HL | NEXT BUFFER LOCATION |
| | JR | NEXT | GET NEXT CHAR |
| OUT | RET | | |

Exercise 8.3: Let us improve this basic routine:

- a—Echo the character back to the device (for a Teletype, for example).
- b—Check that the input string is no longer than 256 characters.

We now have a string of characters in a memory buffer. Let us proc-

ess them in various ways.

TESTING A CHARACTER

Let us determine if the character at memory location LOC is equal to 0, 1, or 2:

| | | | |
|-----|----|----------|-----------------|
| ZOT | LD | A, (LOC) | GET CHARACTER |
| | CP | 00 | IS IT A ZERO? |
| | JP | Z, ZERO | JUMP TO ROUTINE |
| | CP | 01 | A ONE? |
| | JP | Z, ONE | |
| | CP | 02 | A TWO? |
| | JP | Z, TWO | |
| | JP | NOTFND | FAILURE |

We simply read the character, then use the CP instruction to check its value.

Let us run a different test now.

BRACKET TESTING

Let us determine if the ASCII character at memory location LOC is a digit between 0 and 9:

| | | | |
|-------|-----|----------|---------------------|
| BRACK | LD | A, (LOC) | GET CHARACTER |
| | AND | 7FH | MASK OUT PARITY BIT |
| | CP | 30H | ASCII 0 |
| | JR | C, OUT | CHAR TOO LOW? |
| | CP | 39H | ASCII 9 |
| | JR | NC, OUT | CHAR TOO HIGH? |
| | CP | A | FORCE ZERO FLAG |
| OUT | RET | EXIT | |

ASCII "0" is represented in hexadecimal by "30" or by "B0", depending upon whether the parity bit is used or not. Similarly, ASCII "9" is represented in hexadecimal by "39" or by "B9".

The purpose of the second instruction of the program is to delete bit 7, the parity bit, in case it was used, so that the program is applicable to both cases. The value of the character is then compared to the ASCII values for "0" and "9". When using a comparison instruction, the Z flag is set if the comparison succeeds. The carry bit is set in the case of borrow, and reset otherwise. In other words, when using the CP instruction, the carry bit will be set if the value of the literal that appears

PROGRAMMING THE Z80

in the instruction is greater than the value contained in the accumulator. It will be reset ("0") if less than or equal.

The last instruction, CP A, forces a "1" into the Z flag. The Z flag is used to indicate to the calling routine that the character in CHAR was indeed in the interval (0, 9). Other conventions can be used, such as loading a digit in the accumulator in order to indicate the result of the test.

Exercise 8.4: Is the following program equivalent to the one above?:

```
LD    A, (CHAR)
SUB  30H
JP    M, OUT
SUB  10
JP    P, OUT
ADD  10
```

Exercise 8.5: Determine if an ASCII character contained in the accumulator is a letter of the alphabet.

When using an ASCII table, you will notice that parity is often used. For example, the ASCII for "0" is "0110000", a 7-bit code. However, if we use odd parity, for example, we guarantee that the total number of ones in a word is odd; then the code becomes: "10110000". An extra "1" is added to the left. This is "B0" in hexadecimal. Let us therefore develop a program to generate parity.

PARITY GENERATION

This program will generate an even parity with bit position 7:

| | | | |
|--------|-----|-----------|------------------|
| PARITY | LD | A, (CHAR) | GET CHARACTER |
| | AND | 7FH | CLEAR PARITY BIT |
| | JP | PE, OUT | CHECK IF PARITY |
| | | | ALREADY EVEN |
| | OR | 80H | SET PARITY BIT |
| OUT | LD | (LOC), A | STORE RESULT |

The program uses the internal parity detection circuit available in the Z80.

The third instruction: JP PE, OUT checks whether parity of the word in the accumulator is already even. This instruction will succeed if the parity is even, "PE", and will exit.

If the parity is not even, i.e., if the jump instruction failed, then the parity is odd, and a "1" must be written in bit position 7. This is the

purpose of the fourth instruction:

OR 80H

Finally, the resulting value is saved in memory location LOC.

Exercise 8.6: *The above problem was too simple to solve, using the internal parity detection circuitry. As an exercise, you are requested to solve the same problem without using this circuitry. Shift the contents of the accumulator, and count the number of 1's in order to determine which bit should be written into the parity position.*

Exercise 8.7: *Using the above program as an example, verify the parity of a word. You must compute the correct parity, then compare it to the one expected.*

CODE CONVERSION: ASCII TO BCD

Converting ASCII to BCD is very simple. We will observe that the hexadecimal representation of ASCII characters 0 to 9 is 30 to 39 or B0 to B9, depending on parity. The BCD representation is simply obtained by dropping the “3” or the “B”, i.e., masking off the left nibble (4 bits):

| | | | |
|---------|------|--------------|---------------------------|
| ASCBBCD | CALL | BRACK | CHECK THAT CHAR IS 0 TO 9 |
| | JP | NZ, ILLEGAL | EXIT IF ILLEGAL CHAR |
| | AND | OFH | MASK HIGH NIBBLE |
| | LD | (BCDCHAR), A | STORE RESULT |

Exercise 8.8: *Write a program to convert BCD to ASCII.*

Exercise 8.9: *Write a program to convert BCD to binary (more difficult).*

Hint: $N_3 N_2 N_1 N_0$ in BCD is $((N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ in binary.

To multiply by 10, use a left shift ($= \times 2$), another left shift ($= \times 4$), an ADC ($= \times 5$), another left shift ($= \times 10$).

In full BCD notation, the first word may contain the count of BCD digits, the next nibble contain the sign, and every successive nibble contain a BCD digit (we assume no decimal point). The last nibble of the block may be unused.

CONVERT HEX TO ASCII

“A” contains one hexadecimal digit. We simply need to add a “3” (or a

PROGRAMMING THE Z80

“B”) into the left nibble:

| | | |
|-----|--------|-----------------------------|
| AND | 0FH | ZERO LEFT NIBBLE (optional) |
| ADD | A, 30H | ASCII |
| CP | 3AH | CORRECTION NECESSARY? |
| JP | M, OUT | |
| ADD | A, 7 | CORRECTION FOR A TO F |

Exercise 8.10: Convert HEX to ASCII, assuming a packed format (two hex digits in A).

FINDING THE LARGEST ELEMENT OF A TABLE

The beginning address of the table is contained at memory address BASE. The first entry of the table is the number of bytes it contains. This program will search for the largest element of the table. Its value will be left in A, and its position will be stored in memory location INDEX.

This program uses registers A, F, B, H and L, and will use indirect addressing, so that it can search a table anywhere in the memory (see Figure 8.1).

| | | | |
|----------|-----|--------------|------------------------|
| MAX | LD | HL, BASE | TABLE ADDRESS |
| | LD | B, (HL) | NBR OF BYTES IN TABLE |
| | LD | A, 0 | CLEAR MAXIMUM VALUE |
| | INC | HL | INITIALIZE INDEX |
| | LD | (INDEX), HL | NEXT ENTRY |
| LOOP | CP | (HL) | COMPARE ENTRY |
| | JR | NC, NOSWITCH | JUMP IF LESS THAN MAX |
| | LD | A, (HL) | LOAD NEW MAX VALUE |
| | LD | (INDEX), HL | LOAD NEW MAX VALUE |
| NOSWITCH | INC | HL | POINT TO NEXT ENTRY |
| | DEC | B | DECREMENT COUNTER |
| | JR | NZ, LOOP | KEEP GOING IF NOT ZERO |
| | RET | | |

This program tests the nth entry first. If it is greater than 0, the entry goes in A, and its location is remembered into INDEX. The (n-1)st entry is then tested, etc.

This program works for positive integers.

Exercise 8.11: Modify the program so that it works also for negative numbers in two's complement.

Exercise 8.12: Will this program also work for ASCII characters?

Exercise 8.13: Write a program which will sort n numbers in ascending

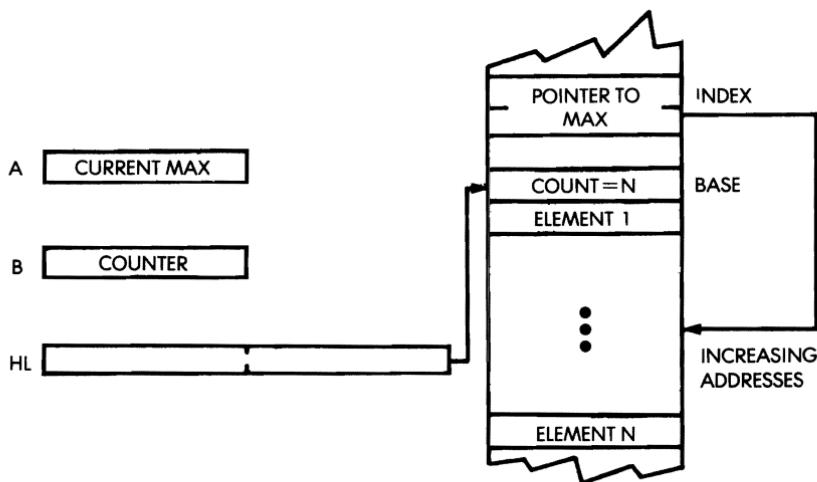


Fig. 8.1: Largest Element in a Table

order.

Exercise 8.14: Write a program which will sort n names (3 characters each) in alphabetical order.

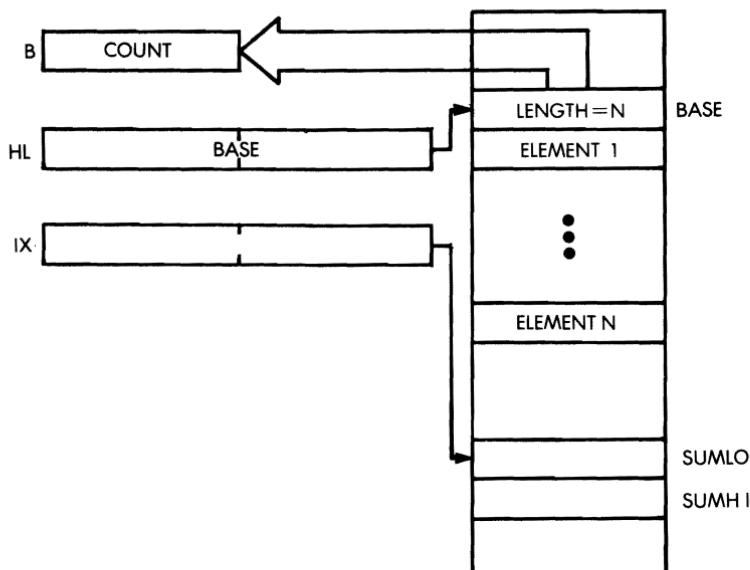
SUM OF N ELEMENTS

This program will compute the 16-bit sum of N positive entries of a table. The starting address of the table is contained at memory address BASE. The first entry of the table contains the number of elements N . The 16-bit sum will be left in memory locations SUMLO and SUMHI. If the sum should require more than 16 bits, only the lower 16 will be kept. (The high order bits are said to be truncated.)

This program will modify registers A, F, B, H, L, IX. It assumes 256 elements maximum (see Figure 8.2).

| | | | |
|-------|-----|-----------|--------------------------|
| SUMN | LD | HL, BASE | POINT TO TABLE BASE |
| | LD | B, (HL) | READ LENGTH INTO COUNTER |
| SUMIG | INC | HL | POINT TO FIRST ENTRY |
| | LD | IX, SUMLO | POINT TO RESULT, LOW |

| | | |
|---------|-----------------|------------------------|
| | LD (IX + 0), 0 | CLEAR RESULT LOW |
| | LD (IX + 1), 0 | AND HIGH |
| ADLOOP | LD A, (HL) | GET TABLE ENTRY |
| | ADD A, (IX + 0) | COMPUTE PARTIAL SUM |
| | LD (IX + 0), A | STORE IT AWAY |
| | JR NC, NOCARRY | CHECK FOR CARRY |
| | INC (IX + 1) | ADD CARRY TO HIGH BYTE |
| NOCARRY | INC HL | POINT TO NEXT ENTRY |
| | DEC B | DECREMENT BYTE COUNT |
| | JR NZ, ADLOOP | KEEP ADDING TILL END |
| | RET | |

**Fig. 8.2: Sum of N Elements**

This program is straightforward and should be self-explanatory.

Exercise 8.15: Modify this program to:

- a—compute a 24-bit sum
- b—compute a 32-bit sum
- c—detect any overflow.

A CHECKSUM COMPUTATION

A checksum is a digit or set of digits computed from a block of successive characters. The checksum is computed at the time the data is

stored and put at the end. In order to verify the integrity of the data, the data is read, then the checksum is recomputed and compared against the stored value. A discrepancy indicates an error or a failure.

Several algorithms are used. Here, we will exclusive-OR all bytes in a table of N elements, and leave the result in the accumulator. As usual, the base of the table is stored at address BASE. The first entry of the table is its number of elements N. The program modifies A, F, B, H, L. N must be less than 256

| | | |
|--------|-----------------|----------------------------------|
| CHKSUM | LD HL, BASE | LOAD ADDRESS OF TABLE INTO HL |
| | LD B, (HL) | GET N = LENGTH |
| | XOR A | CLEAR CHECKSUM |
| | INC HL | POINT TO FIRST ELEMENT |
| CHLOOP | XOR (HL) | COMPUTE CHECKSUM |
| | INC HL | POINT TO NEXT ELEMENT |
| | DEC B | DECREMENT COUNTER |
| | JR NZ, CHLOOP | DO IT AGAIN IF NOT END |
| | LD (CHECKSUM),A | PRESERVE CHECKSUM |
| | RET | |

COUNT THE ZEROES

This program will count the number of zeroes in our usual table, and leave it in location TOTAL. It modifies A, B, C, H, L, F.

| | | |
|-------|---------------|-----------------------------|
| ZEROS | LD HL, BASE | POINT TO TABLE |
| | LD B, (HL) | READ LENGTH INTO COUNTER |
| | LD C, 0 | ZERO TOTAL |
| | INC HL | POINT TO FIRST ENTRY |
| ZLOOP | LD A, (HL) | GET ELEMENT |
| | OR 0 | SET ZERO FLAG |
| | JR NZ, NOTZ | IS IT A ZERO? |
| | INC C | IF SO, INCREMENT ZERO COUNT |
| NOTZ | INC HL | POINT TO NEXT ENTRY |
| | DEC B | DECREMENT LENGTH COUNTER |
| | JR NZ, ZLOOP | |
| | LD A,C | |
| | LD (TOTAL), A | SAVE IT |

*Exercise 8.16: Modify this program to count
 a—the number of stars (the character “*”)
 b—the number of letters of the alphabet
 c—the number of digits between “0” and “9”*

BLOCK TRANSFER

Let us pick up every third entry in the source block at address FROM and store it into a block at address TO:

| | | | |
|------|-----|----------|--------------------|
| FER3 | LD | HL, FROM | |
| | LD | DE, TO | SET UP POINTERS |
| | LD | BC, SIZE | |
| LOOP | LDI | | AUTOMATED TRANSFER |
| | INC | HL | |
| | INC | HL | SKIP 2 ENTRIES |
| | JP | PE, LOOP | |

BCD BLOCK TRANSFER

We will push up BCD digits in the memory, i.e., shift 4-bit nibbles (see Figure 8.3). The program appears below:

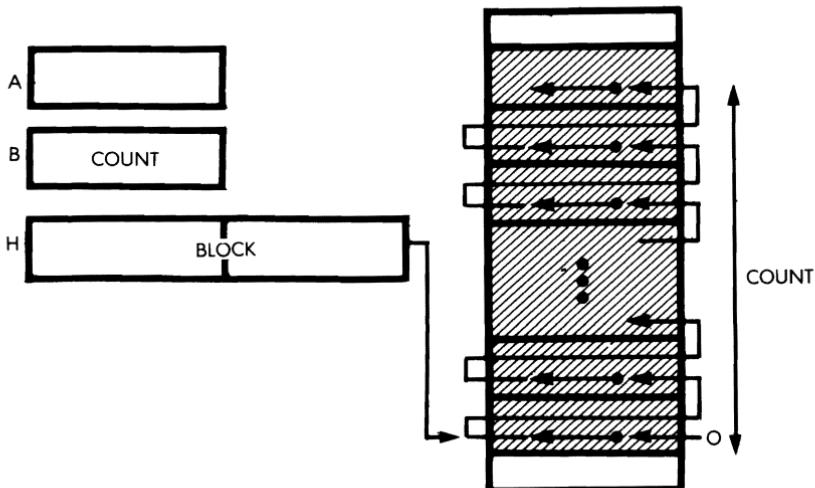


Fig. 8.3: BCD Block Transfer - The Memory

| | | | |
|------|------|-----------|---------------------------|
| DMOV | LD | B, COUNT | |
| | LD | HL, BLOCK | |
| | XOR | A | A = 0 |
| LOOP | RLD | | |
| | DEC | HL | POINT TO NEXT BYTE |
| | DJNZ | LOOP | DEC COUNT LOOP UNTIL ZERO |

The program uses the RLD instruction, which we have not used yet. RLD rotates a BCD digit left between A and (HL). (HL) or M designate the contents of the memory location pointed to by H and L.

M LOW goes into M HIGH

M HIGH goes into A LOW

A LOW goes into M LOW

Here, "low" and "high" refer to a 4-bit nibble.

In order to use the powerful DJNZ instruction, register B is used as the digit counter. HL is set to point to the beginning of the block.

A is used to store the left digit displaced by each rotation between two successive accesses to the block.

By convention, "0" will be entered at the bottom of the block.

COMPARE TWO SIGNED 16-BIT NUMBERS

IX points to the first number N1.

IY points to N2 (see Figure 8.4).

The program sets the carry bit if $N1 < N2$, and the Z bit if $N1 = N2$.

| | | | |
|-------|-----|-------------|---------------------|
| COMP | LD | B, (IX + 1) | GET SIGN OF N1 |
| | LD | A, B | |
| | AND | 80H | TEST SIGN, CLEAR CY |
| | JR | NZ, NEGM1 | N1 IS NEG |
| | BIT | 7, (IY + 1) | |
| | RET | NZ | N2 IS NEG |
| | LD | A, B | |
| | CP | (IY + 1) | SIGNS ARE BOTH POS |
| | RET | NZ | |
| | LD | A, (IX) | |
| | CP | (IY) | |
| | RET | | |
| NEGM1 | XOR | (IY + 1) | |
| | RLA | | SIGN BIT INTO CY |
| | RET | C | SIGNS DIFFERENT |
| | LD | A, B | |
| | CP | (IY + 1) | BOTH SIGNS NEG |
| | RET | NZ | |
| | LD | A, (IX) | |
| | CP | (IY) | |
| | RET | | |

The program first tests the signs of N1 and N2. If N1 is negative, a

PROGRAMMING THE Z80

jump occurs to NEGM1. Otherwise, the top of the program is executed.

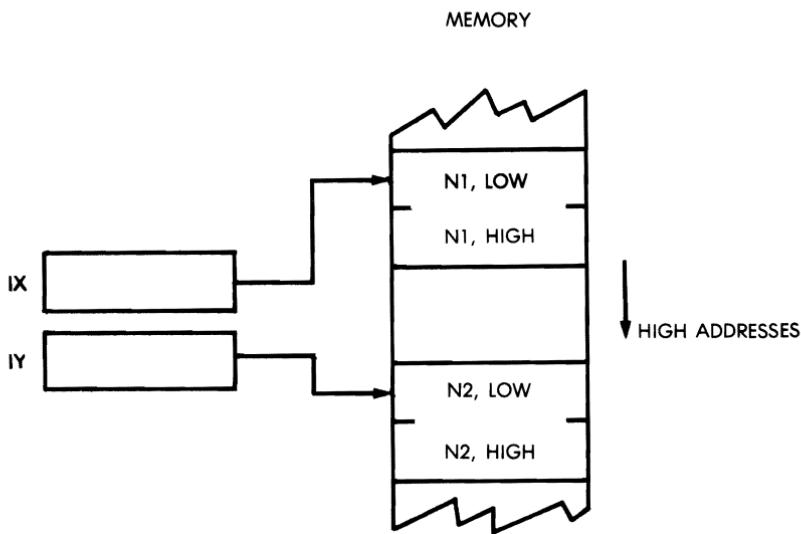


Fig. 8.4: Comparing Two Signed Numbers

Note that the BIT instruction is used in the 5th line to test directly the sign bit of N2 in the memory:

BIT 7, (IY + 1)

The same could have been done for N1, except that we will need the value of N1 shortly. It is therefore simpler to read N1 from memory and preserve it into B:

COMP LD B, (IX + 1)

It is necessary to preserve N1 into B because the AND may destroy the contents of A:

LD A, B
AND 80H

Note also that a conditional return is used (line 6):

RET NZ

This is a powerful feature of the Z80 which simplifies programming.

Note that the comparison instruction executes directly on the contents of memory, in indexed mode:

CP (IY + 1)

When comparing the two numbers, the most significant byte is compared first, the least significant one second.

Note the extensive use of the indexing mechanism in this program, which results in efficient code.

BUBBLE-SORT

Bubble-sort is a sorting technique used to arrange the elements of a table in ascending or descending order. The bubble-sort technique derives its name from the fact that the smallest element “bubbles up” to the top of the table. Every time it “collides” with a “heavier” element, it jumps over it.

A practical example of a bubble-sort is shown on Figure 8.5. The list to be sorted contains: (10, 5, 0, 2, 100), and must be sorted in descending order (“0” on top). The algorithm is simple, and the flowchart is shown on Figure 8.7.

The top two (or else bottom two) elements are compared. If the lower one is less (“lighter”) than the top one, they are exchanged. Otherwise not. For practical purposes, the exchange, if it occurs, will be remembered in a flag called “EXCHANGED”. The process is then repeated on the next pair of elements, etc., until all elements have been compared two by two.

This first pass is illustrated by steps 1, 2, 3, 4, 5, 6 on Figure 8.5, going from the bottom up. (Equivalently we could go from the top down.)

If no elements have been exchanged, the sort is complete. If an exchange has occurred, we start all over again.

Looking at Figure 8.6, it can be seen that four passes are necessary in this example.

The process is simple, and is widely used.

One additional complication resides in the actual mechanism of the exchange.

When exchanging A and B, one may not write

**A = B
B = A**

as this would result in the loss of the previous value of A (try it on an example).

PROGRAMMING THE Z80

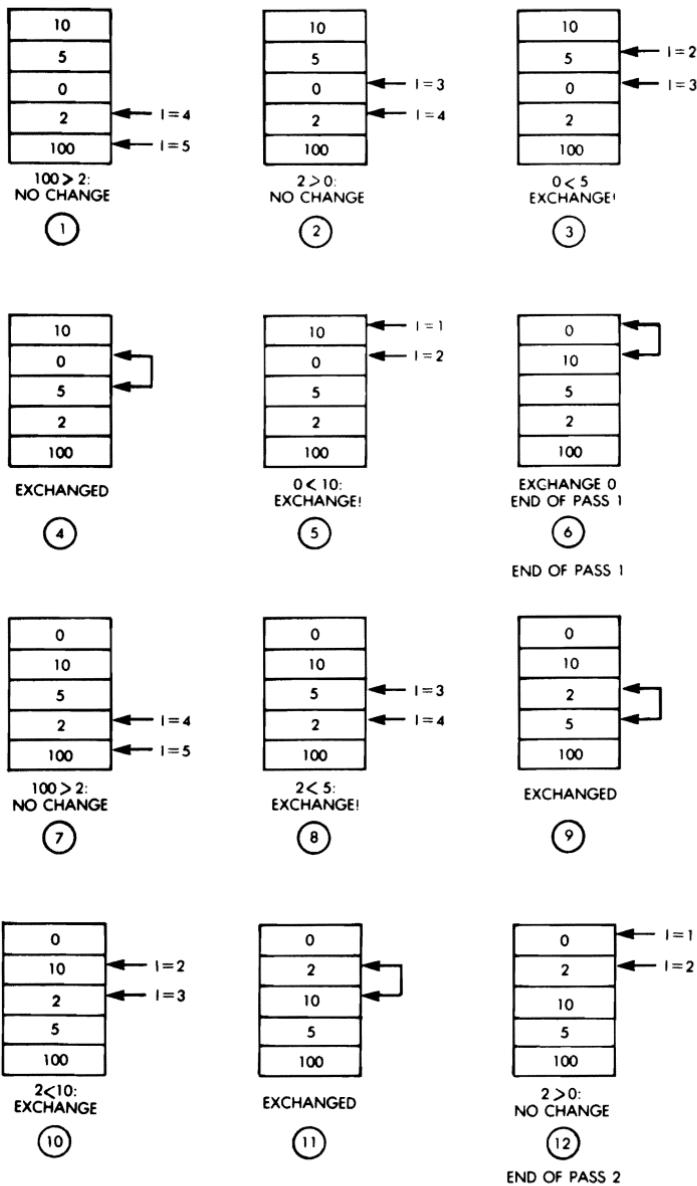


Fig. 8.5: Bubble-Sort Example: Phases 1 to 12

| |
|-----|
| 0 |
| 2 |
| 10 |
| 5 |
| 100 |

100 > 5
NO CHANGE

(13)

| |
|-----|
| 0 |
| 2 |
| 10 |
| 5 |
| 100 |

5 < 10:
EXCHANGE!

(14)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

EXCHANGED

(15)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

5 > 2:
NO CHANGE

(16)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

2 > 0:
NO CHANGE

(17)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

100 > 10:
NO CHANGE

(18)

END OF PASS 3

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

10 > 5:
NO CHANGE

(19)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

5 > 2:
NO CHANGE

(20)

| |
|-----|
| 0 |
| 2 |
| 5 |
| 10 |
| 100 |

2 > 0:
NO CHANGE

(21)

END

Fig. 8.6: Bubble-Sort Example: Phases 13 to 21

The correct solution is to use a temporary variable or location to preserve the value of A:

| | |
|------|--------|
| TEMP | = A |
| A | = B |
| B | = TEMP |

It works (try it on an example). This is called a circular permutation.

This is the way all programs implement the exchange. This technique is illustrated on the flowchart of Figure 8.7.

PROGRAMMING THE Z80

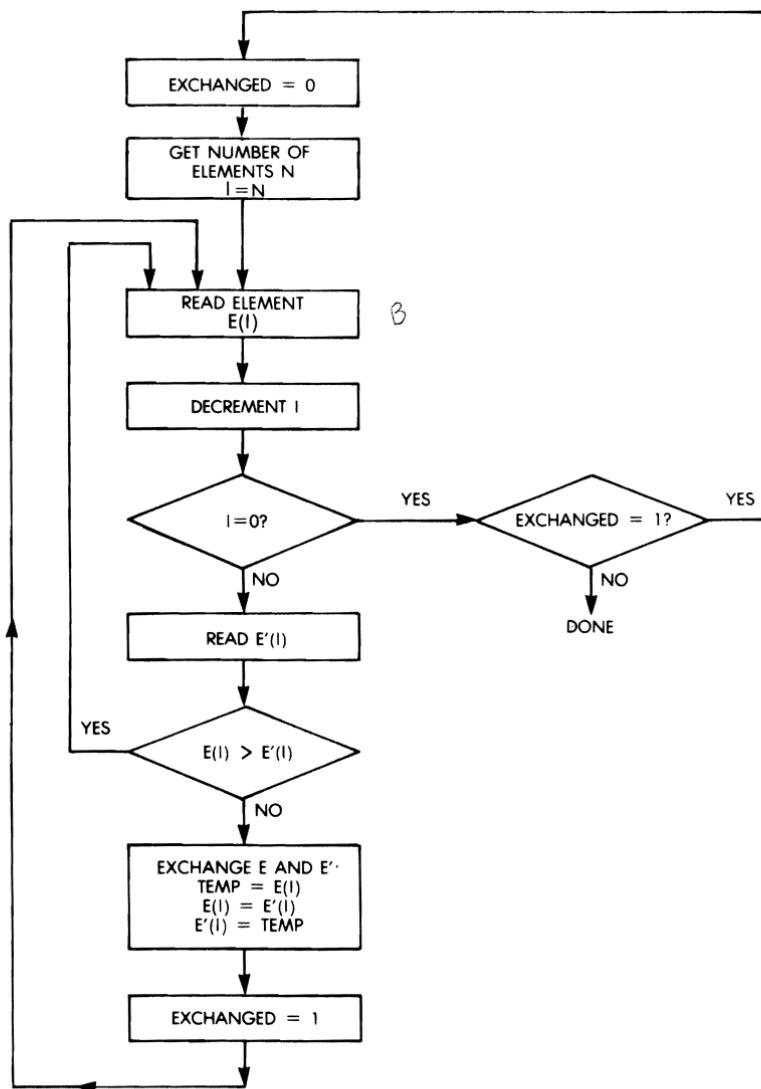
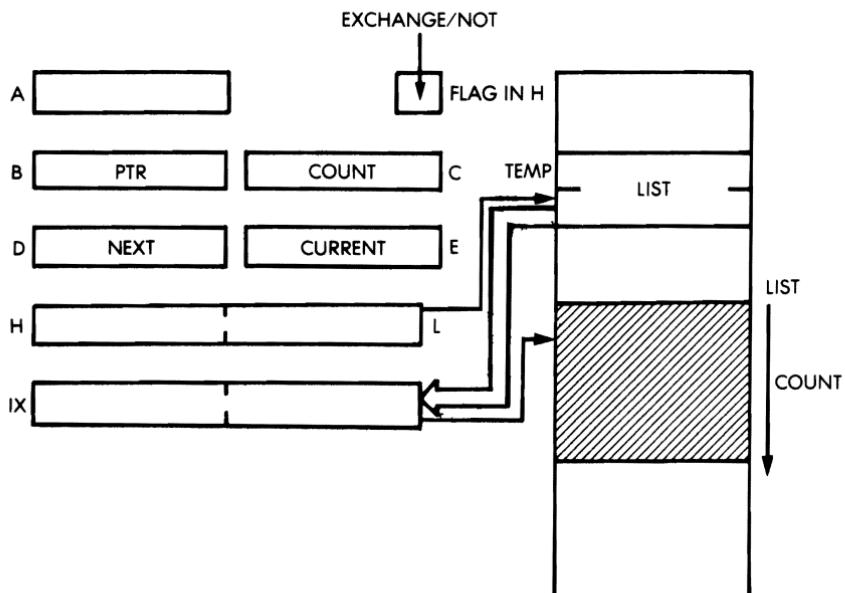


Fig. 8.7: Bubble-Sort Flowchart

**Fig. 8.8: Bubble-Sort**

The register and memory assignments are shown on Figure 8.8, and the program is:

| | | | |
|---------|-----|--------------|-------------------------------------|
| BUBBLE | LD | (TEMP), HL | TEMP = (HL) |
| AGAIN | LD | IX, (TEMP) | IX = (HL) |
| | RES | FLAG, H | EXCHANGED FLAG = 0 |
| | LD | B, C | |
| | DEC | B | |
| NEXT | LD | A, (IX) | |
| | LD | D, A | D = CURRENT ENTRY |
| | LD | E, (IX + 1) | E = NEXT ENTRY |
| | CP | E | COMPARE |
| | JR | NC, NOSWITCH | GO TO NOSWITCH IF CURRENT > NEXT |
| XCHANGE | LD | (IX), E | STORE NEXT INTO CURRENT |
| | LD | (IX + 1), D | STORE CURRENT INTO NEXT |
| | SET | FLAG, H | EXCHANGED FLAG = 1 |

```
NOSWITCH INC IX  
        DJNZ NEXT  
  
        BIT FLAG, H  
        JR NZ, AGAIN  
        RET
```

```
NEXT ENTRY  
DEC B, CONTINUE UNTIL  
ZERO  
EXCHANGED=1?  
RESTART IF FLAG=1
```

SUMMARY

Common utility routines have been presented in this chapter which use combinations of the techniques we have described in the previous chapters. They should allow you to start designing your own programs now. Many of these routines have used a special data structure, the table. Other possibilities exist for structuring data, and will now be reviewed.

9

DATA STRUCTURES

PART I — THEORY

INTRODUCTION

The design of a good program involves two tasks: *algorithm design* and *data structures design*. In most simple programs, no significant data structures are involved, so the main objective in learning programming is designing algorithms and coding them efficiently in a given machine language. This is what we have accomplished here. However, designing more complex programs also requires an understanding of data structures. Two data structures have already been used throughout the book: the table and the stack. The purpose of this chapter is to present other, more general, data structures that you may want to use. This chapter is completely independent of the microprocessor, or even the computer, selected. It is theoretical and involves the logical organization of data in the system. Specialized books exist on the topic of data structures, just as specialized books exist on the subject of efficient multiplication, division or other usual algorithms. This chapter, therefore, will be limited to essentials only. It does not claim to be complete. The most common data structures will now be reviewed.

POINTERS

A pointer is a number which is used to designate the location of the actual data. Every pointer is an address. However, every address is not necessarily called a pointer. An address is a pointer only if it points at

some type of data or at structured information. We have already encountered a typical pointer: the stack pointer, which points to the top of the stack (or usually just over the top of the stack). We will see that the stack is a common data structure, called an LIFO structure.

As another example, when using indirect addressing, the indirect address is always a pointer to the data that one wishes to retrieve.

Exercise 9.1: Examine Fig. 9.1. At address 15 in the memory, there is a pointer to Table T. Table T starts at address 500. What are the actual contents of the pointer to T?

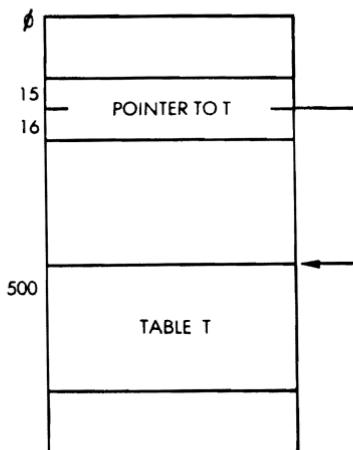


Fig. 9.1: An Indirection Pointer

LISTS

Almost all data structures are organized as lists of various kinds.

Sequential Lists

A sequential list, or table, or block, is probably the simplest data structure, and is one that we have already used. Tables are normally ordered in function of a specific criterion, such as alphabetical ordering or numerical ordering. It is then easy to retrieve an element in a table, using, for example, indexed addressing, as we have done. A block normally refers to a group of data which has definite limits but whose contents are not ordered. It may contain a string of characters; it may

be a sector on a disk; or it may be some logical area (called segment) of the memory. In such cases, it may not be easy to access a random element of the block.

In order to facilitate the retrieval of blocks of information, directories are used.

Directories

A directory is a list of tables or blocks. For example, the file system will normally use a directory structure. As a simple example, the master directory of the system may include a list of the users' names. This is illustrated in Figure 9.2. The entry for user "John" points to John's file directory. The file directory is a table which contains the names of all of John's files and their location. In this case, we have just designed a two-level directory. A flexible directory system will allow the inclusion of additional intermediate directories, as may be found convenient by the user.

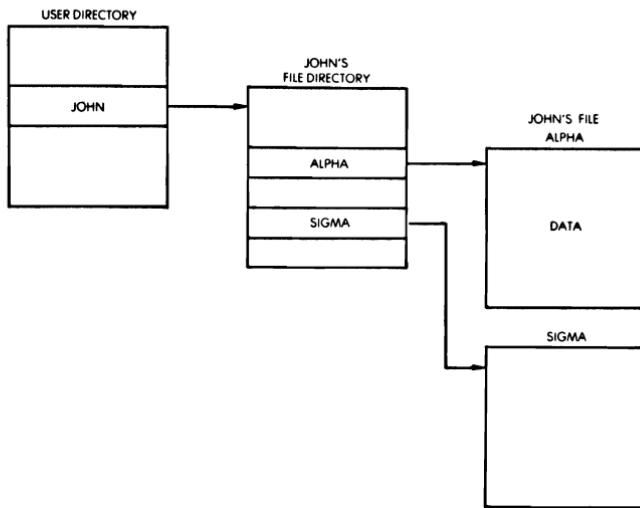


Fig. 9.2: A Directory Structure

Linked List

In a system there are often blocks of information which represent data, events, or other structures which cannot be moved around eas-

PROGRAMMING THE Z80

ily. If they could, we would probably assemble them in a table in order to sort or structure them. The problem now is that we wish to leave them where they are and still establish an ordering among them such as first, second, third, fourth. A linked list will be used to solve this problem. The concept of a linked list is illustrated by Figure 9.3. On the illustration, we see that a list pointer, called FIRSTBLOCK, points to the beginning of the first block. A dedicated location within Block 1 such as, perhaps, the first or the last word in it, contains a pointer to Block 2, called PTR1. The process is then repeated for Block 2 and Block 3. Since Block 3 is the last entry in the list, PTR3, by convention, either contains a special "nil" value, or points to itself, so that the end of the list can be detected. This structure is economical, as it requires only a few pointers (one per block) and frees the user from having to physically move the blocks in the memory.

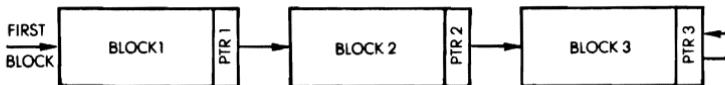


Fig. 9.3: A Linked List

Let us examine, for example, how a new block will be inserted. This is illustrated by Figure 9.4. Let us assume that the new block is at address NEWBLOCK, and is to be inserted between Block 1 and Block 2. Pointer PTR1 is simply changed to the value NEWBLOCK, so that it now points to Block X. PTRX will contain the former value of PTR1, i.e., it will point to Block 2. The other pointers in the structure are left unchanged. We can see that the insertion of a new block has simply required updating two pointers in the structure. This is clearly efficient.

Exercise 9.2: Draw a diagram showing how Block 2 would be removed from this structure.

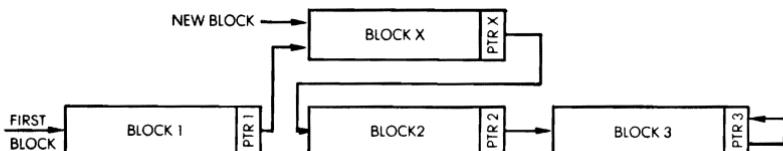


Fig. 9.4: Inserting a New Block

Several types of lists have been developed to facilitate specific types of access, insertions, and deletions to and from the list. Let us examine some of the most frequently used types of linked lists.

Queue

A queue is formally called a FIFO, or first-in-first-out list. A queue is illustrated in Figure 9.5. To clarify the diagram, we can assume, for example, that the block on the left is a service routine for an output device, such as a printer. The blocks appearing on the right are the request blocks from various programs or routines, to print characters. The order in which they will be serviced is the order established by the waiting queue. It can be seen that the first event which will obtain service is Block 1, the next one is Block 2, and the following one is Block 3. In a queue, the convention is that any new event arriving in the queue will be inserted at the end. Here it will be inserted after PTR3. This guarantees that the first block to be inserted in the queue will be the first one to be serviced. It is quite common in a computer system to have queues for a number of events whenever they must wait for a scarce resource, such as the processor or some input/output device.

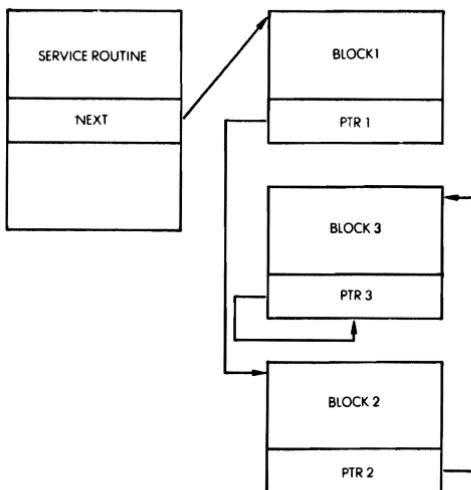


Fig. 9.5: A Queue

Stack

The stack structure has already been studied in detail throughout the book. It is a last-in-first-out structure (LIFO). The last element deposited on top is the first one to be removed. A stack may either be implemented as a sorted block, or it may be implemented as a list. Because most stacks in microprocessors are used for high-speed events, such as subroutines and interrupts, a continuous block is usually allocated to the stack instead of using a linked list.

Linked List vs. Block

Similarly, the queue could be implemented as a block of reserved locations. The advantage of using a continuous block is fast retrieval and the elimination of the pointers. The disadvantage is that it is usually necessary to dedicate a fairly large block to accommodate the worst-case size of the structure. Also, it makes it difficult or impractical to insert or remove elements from within the block. Since memory is traditionally a scarce resource, blocks have usually been reserved for fixed-size structures or structures requiring the maximum speed of retrieval, such as the stack.

Circular List

“Round robin” is a common name for a circular list. A circular list is a linked list in which the last entry points back to the first one. This is illustrated in Figure 9.6. In the case of a circular list, a *current-block* pointer is often kept. In the case of events, or programs, waiting for service, the *current-event* pointer will be moved by one position to the left or to the right every time. A round robin usually corresponds to a structure in which all blocks are assumed to have the same priority. However, a circular list may also be used as a subcase of other structures simply to facilitate the retrieval of the first block after the last one, when performing a search.

As an example of a circular list, a polling program usually goes in a round robin fashion, interrogating all peripherals and then coming back to the first one.

Trees

Whenever a logical relationship exists among all elements of a structure (this is usually called a syntax), a tree structure may be used. A simple example of a tree structure is a descendant, or genealogical, tree.

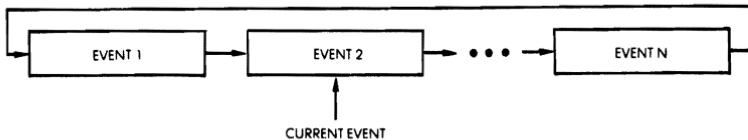


Fig. 9.6: Round Robin is Circular List

This is illustrated in Figure 9.7. It can be seen that Smith has two children: a son, Robert, and a daughter, Jane. Jane, in turn, has three children: Liz, Tom and Phil. Tom, in turn, has two more children: Max and Chris. However, Robert, on the left of the illustration, has no descendants.

This is a structured tree. We have, in fact, already encountered an example of a simple tree in Figure 9.2. The directory structure is a two-level tree. Trees are used to advantage whenever elements may be classified according to a fixed structure. This facilitates insertion and retrieval. In addition, they may establish groups of information in a structured way which may be required for later processing, such as in a compiler or interpreter design.

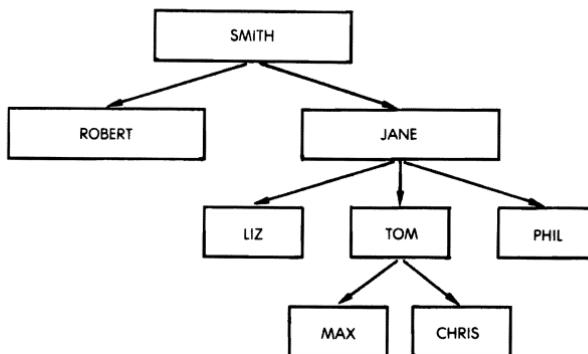


Fig. 9.7: Genealogical Tree

Doubly-Linked Lists

Additional links may be established between elements of a list. The

simplest example is the doubly-linked list. This is illustrated in Figure 9.8. We can see that we have the usual sequence of links from left to right, plus another sequence of links from right to left. The goal is to allow easy retrieval of the element just before the one which is being processed, as well as just after it. This costs an extra pointer per block.

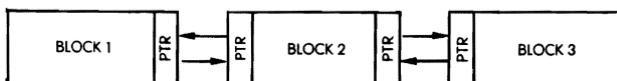


Fig. 9.8: Doubly-Linked List

SEARCHING AND SORTING

Searching and sorting elements of a list depends directly on the type of structure which has been used for the list. Many searching algorithms have been developed for the most frequently used data structures. We have already used indexed addressing. This is possible whenever the elements of a table are ordered in function of a known criterion. Such elements may then be retrieved by their numbers.

Sequential searching refers to the linear scanning of an entire block. This is clearly inefficient but may have to be used when no better technique is available, for lack of ordering of the elements.

Binary, or logarithmic, searching attempts to find an element in a sorted list by dividing the search interval in half at every step. Assuming that we are searching an alphabetical list, one might start, for example, in the middle of a table and determine if the name we are looking for is before or after this point. If it is after this point, we will eliminate the first half of the table and look at the middle element of the second half. We compare this entry again to the one we are looking for, and we restrict our search to one of the two halves, and so on. The maximum length of a search is then guaranteed to be $\log_2 n$, where n is the number of elements in the table.

Many other search techniques exist.

SECTION SUMMARY

This section was intended as only a brief presentation of usual data structures which may be used by a programmer. Although most com-

mon data structures have been organized in types and given a name, the overall organization of data in a complex system may use any combination of them, or require the programmer to invent more appropriate structures. The array of possibilities is only limited by the imagination of the programmer. Similarly, a number of well-known sorting and searching techniques have been developed for coping with the usual data structures. A comprehensive description is beyond the scope of this book. The contents of this section were intended to stress the importance of designing appropriate section structures for the data to be manipulated and to provide the basic tools to that effect.

Actual programming examples will now be presented in detail.

PART II = DESIGN EXAMPLES

INTRODUCTION

Actual design examples will be presented here for typical data structures: table, sorted list, linked list. Practical searching and insertion and deletion algorithms will be programmed for these structures.

The reader interested in these advanced programming techniques is encouraged to analyze in detail the programs presented in this section.

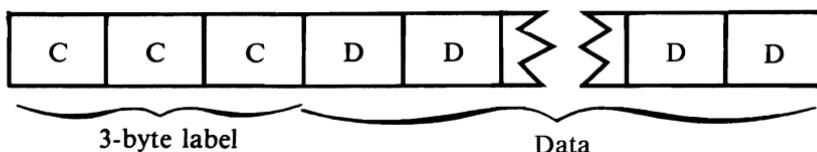
However, the beginning programmer may skip this section initially, and come back to it when he feels ready for it.

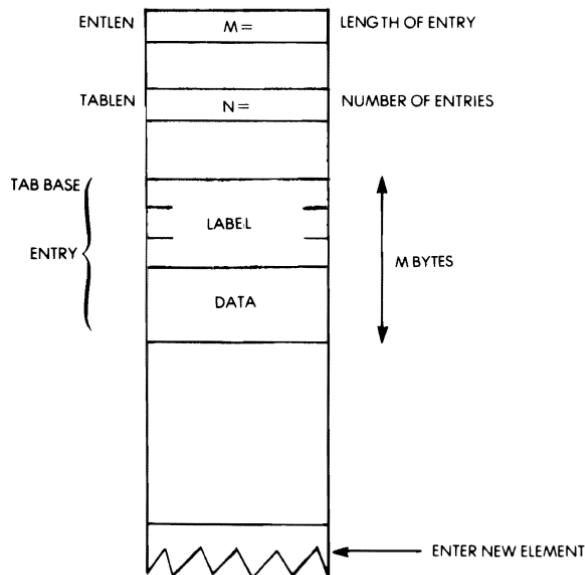
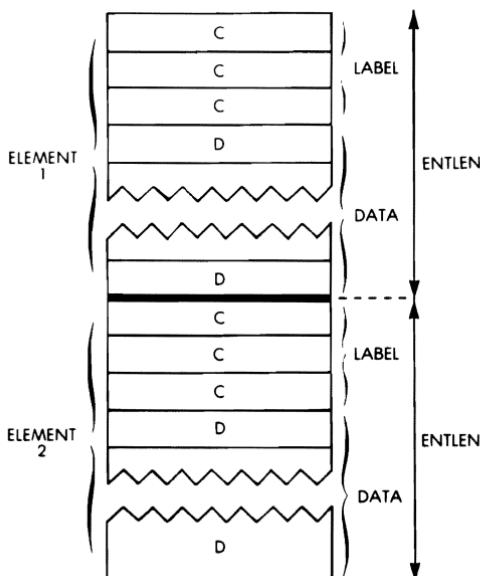
A good understanding of the concepts presented in the first part of this chapter is necessary to follow the design examples. Also, the programs will use all of the addressing modes of the Z80, and integrate many of the concepts and techniques presented in the previous chapters.

Three structures will now be introduced: a simple list, an alphabetical list and a linked-list plus directory. For each structure, three programs will be developed: search, enter and delete.

DATA REPRESENTATION FOR THE LIST

Both the simple list and the alphabetic list will use a common representation for each list element:



**Fig. 9.9: The Table Structure****Fig 9.10: Typical List Entries in the Memory**

PROGRAMMING THE Z80

Each element, or “entry”, includes a 3-byte label, and an n-byte block of data, with n between 1 and 253. Thus, at most, each entry uses one page (256 bytes). Within each list, all elements have the same length (see Figure 9.10). The programs operating on these two simple lists use some common variable conventions:

ENTLEN is the length of an element. For example, if each element has 10 bytes of data, $\text{ENTLEN} = 3 + 10 = 13$

TABASE is the base of the list or table in the memory

POINTR is a running pointer to the current element

OBJECT is the current entry to be located, inserted or deleted

TABLEN is the number of entries.

All labels are assumed to be distinct. Changing this convention would require a minor change in the programs.

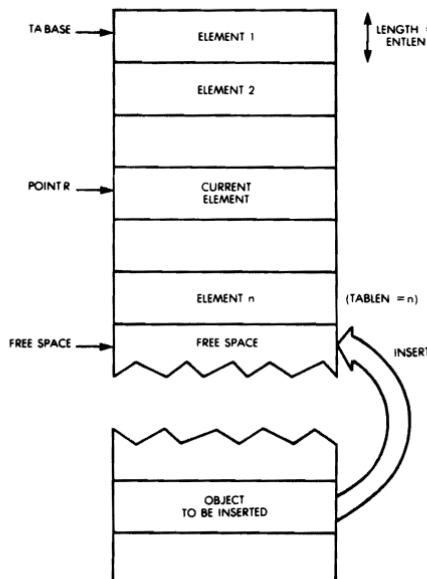


Fig. 9.11: The Simple List

A SIMPLE LIST

The simple list is organized as a table of n elements. The elements are not sorted (see Figure 9.11). When searching, one must scan through the list until an entry is found or the end of the table is reached. When inserting, new entries are appended to the existing ones. When an entry is deleted, the entries in higher memory locations, if any, will be shifted up to keep the table continuous.

Searching

A serial search technique is used. Each entry's label field is compared in turn to the OBJECT's label, letter by letter.

The running pointer **POINTR** is initialized to the value of **TABASE**.

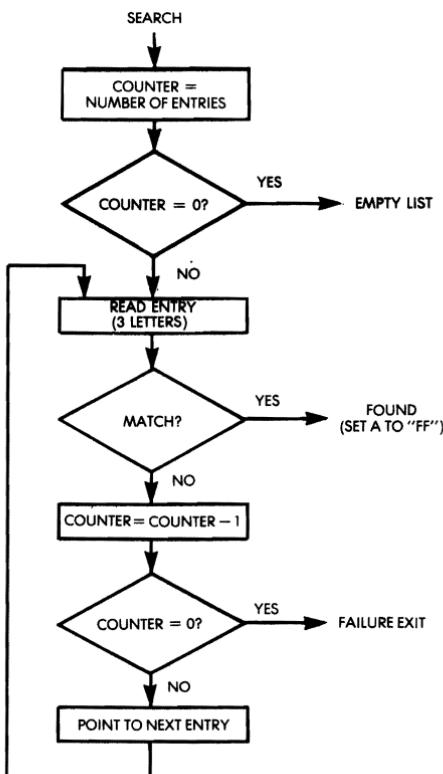


Fig. 9.12: Table Search Flowchart

The search proceeds in the obvious way, and the corresponding flowchart is shown on Figure 9.12. The program appears on Figure 9.16 at the end of this section (program “SEARCH”). A sample run of the program is shown in Figure 9.17.

Inserting

When inserting a new element, the first available memory block of (ENTLEN) bytes at the end of the list is used (see Figure 9.11).

The program first checks that the new entry is not already in the list (all labels are assumed to be distinct in this example). If not, it increments the list length TABLEN, and moves the OBJECT to the end of the list. The corresponding flowchart is shown in Figure 9.13.

The program is shown in Figure 9.16. It is called “NEW” and resides at memory locations 0135 to 015E.

The index register IY points to the source. HL and DE are destination pointers.

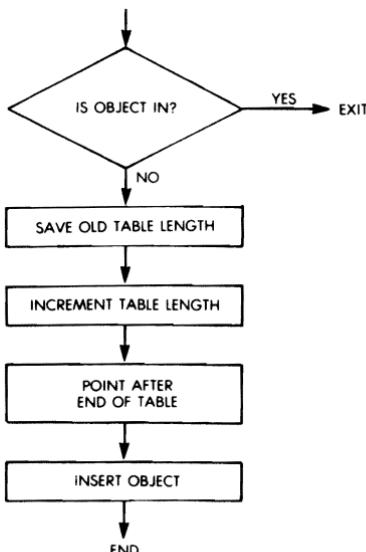


Fig. 9.13: Table Insertion Flowchart

Deleting

In order to delete an element from the list, the elements following it in the list at higher addresses are merely moved up by one element position. The length of the list is decremented. This is illustrated on Figure 9.14.

The corresponding program is straightforward and appears on Figure 9.16. It is called “DELETE”, and resides at memory addresses 015F to 0187. The flowchart is shown in Figure 9.15.

Memory location TEMPTR is used as a temporary pointer pointing to the element to be moved up.

During the transfer, POINTR always points to the “hole” in the list, i.e., the destination of the next block transfer.

The Z flag is used to indicate a successful deletion upon exit.

Note how the LDIR instruction is used for efficient automated block transfer (refer to address 0178 in Figure 9.16).

| | | | |
|---------|------|--------------|---------------|
| | LD | A, B | BLOCK COUNTER |
| NEWBLOC | LD | BC, (ENTLEN) | BLOCK LENGTH |
| | LDIR | | |
| | DEC | A | |
| | JP | NZ, NEWBLOC | |

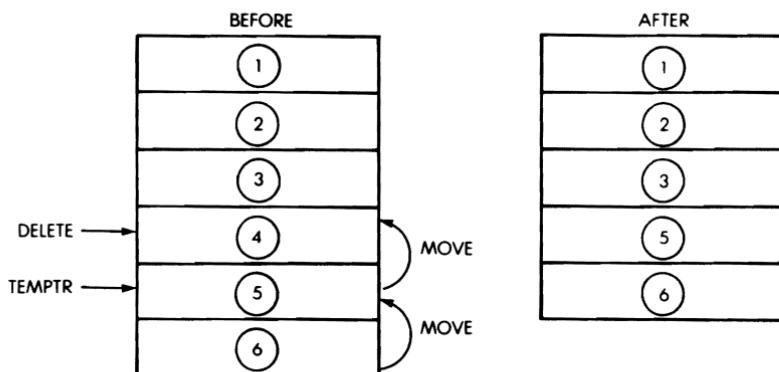


Fig. 9.14: Deleting an Entry (Simple List)

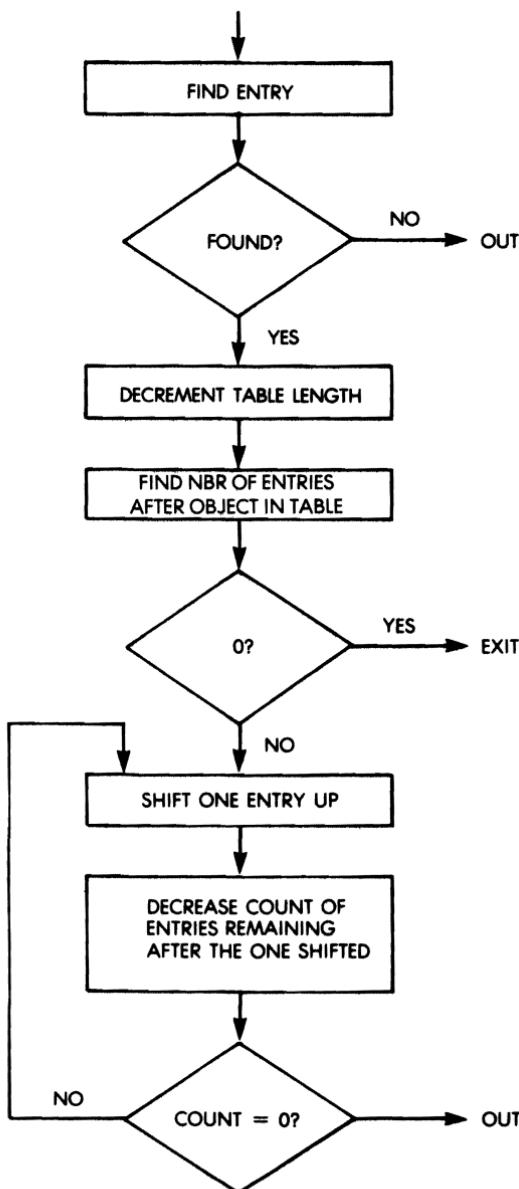


Fig. 9.15: Table Deletion Flowchart

DATA STRUCTURES

```

0000      ORG 0100H
          (0187) ENTLEN DL ENDER
          (0189) TABLEN DL ENDER+2
          (018A) TABASE DL ENDER+3
          (018C) TEMP  DL ENDER+5
;
0100 1600  SEARCH LD D,0           ;CLEAR D
0102 3A8901  LD A,(TABLEN)    ;CHECK FOR A ZERO TABLE LENGTH
0105 A7   AND A             ;SET FLAGS
0106 C8   RET Z             ;
0107 47   LD B,A           ;STORE TABLE LENGTH
0108 DD2ABA01 LD IX,(TABASE) ;PUT BASE ADDR. IN IX
010C DD7E00  LOOP LD A,(IX+0)  ;CHECK FIRST LETTER OF ENTRY
010F FDBE00
0112 C22701 JP NZ,NEXTONE
0115 DD7E01 LD A,(IX+1)  ;CHECK 2ND LETTER
0118 FDBE01 CP (IY+1)
011B C22701 JP NZ,NEXTONE
011E DD7E02 LD A,(IX+2)  ;CHECK 3RD LETTER
0121 FDBE02 CP (IY+2)
0124 CA3201 JP Z,FOUND
0127 05   NEXTONE DEC B           ;EXIT IF ALL LETTERS MATCH
0128 C8   RET Z             ;DECREMENT TABLE LENGTH COUNTER
0129 ED5B8701 LD DE,(ENTLEN) ;EXIT IF AT END OF TABLE
012D DD19  ADD IX,DE
012F C30C01 JP LOOP
0132 16FF  FOUND LD D,0FFH  ;TRY AGAIN
0134 C9   RET D,OFFFH ;SET D TO SHOW IX CONTAINS ADDR.
;
;
;
0135 CD0001 NEW  CALL SEARCH    ;SEE IF OBJECT IS THERE
0138 14   INC D             ;IF D WAS FF, EXIT
0139 CA5E01 JP Z,DUTE
013C 3A8901 LD A,(TABLEN)
013F 5F   LD E,A           ;LOAD E WITH TABLE LENGTH
0140 3C   INC A             ;INCREMENT TABLE LENGTH
0141 328901 LD (TABLEN),A
0144 1600 LD D,0
0146 2ABA01 LD HL,(TABASE)
0149 ED4B8701 LD BC,(ENTLEN) ;SET B TO LENGTH OF AN ENTRY
014D 41   LD B,C
014E 19   LOOPE ADD HL,DE
014F 10FD  DJNZ LOOPE
0151 ED4B8701 LD BC,(ENTLEN) ;ADD HL TO (ENTLEN:TABLEN)
0155 FDE5  PUSH IY
0157 D1   POP DE           ;MOVE IY TO DE
0158 EB   EX DE,HL
0159 EDB0  LDIR
015B 01FFFF LD BC,0FFFFH ;MOVE MEMORY FROM OBJECT TO END
015E C9   DUTE RET
;
;
;
015F CD0001 DELETE CALL SEARCH    ;FIND ENTRY TO BE DELETED
0162 14   INC D             ;SEE IF IT WAS FOUND
0163 C28601 JP NZ,DUT
0166 3A8901 LD A,(TABLEN)
0169 3D   DEC A             ;DECREMENT TABLE LENGTH
016A 328901 LD (TABLEN),A
016D 05   DEC B             ;B NOW=# OF ENTRIES LEFT IN TABLE
016E CA8301 JP Z,EXIT
0171 DDE5  PUSH IX
0173 D1   POP DE           ;MOVE IX TO DE
0174 2AB701 LD HL,(ENTLEN) ;SET HL ONE ENTRY AHEAD OF DE
0177 19   ADD HL,DE
0178 78   LD A,B
0179 ED4B8701 NEWBLOC LD BC,(ENTLEN) ;SET BLOCK COUNTER
017D EDB0  LDIR             ;SET BLOCK LENGTH COUNTER
017F 3D   DEC A             ;SHIFT 1 ENTRY OF TABLE
0180 C27901 JP NZ,NEWBLOC ;SHIFT ANOTHER BLOCK
0183 01FFFF EXIT  LD BC,0FFFFH ;SHOW THAT IT WAS DONE
0186 C9   OUT  RET
;
;
0187 (0000) ENDER END

```

Fig. 9.16: Simple List—The Programs

PROGRAMMING THE Z80

SYMBOL TABLE

| | | | | | | | | | |
|--------|------|-------|------|--------|------|--------|------|--------|------|
| DELETE | 015F | ENDER | 0187 | ENTLEN | 0187 | EXIT | 0183 | FOUND | 0132 |
| LOOP | 010C | LOOPE | 014E | NEW | 0135 | NEWBLO | 0179 | NEXTON | 0127 |
| OUT | 0186 | OUTE | 015E | SEARCH | 0100 | TABASE | 018A | TABLEN | 0189 |
| TEMP | | | | | | | | | |

Fig. 9.16: Simple List—The Programs (cont.)

| Display Memory | | | | | | | | | | Listing of Objects with their locations in memory | | | |
|--|--|--|--|--|--|--|--|--|--|---|--|--|--|
| -DM300 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0300 53 4F 4E 31 31 31 31 31-31 31 31 31 31 31 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0310 44 41 44 32 32 32 32 32-32 32 32 32 32 32 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0320 40 4F 4D 33 33 33 33 33-33 33 33 33 33 33 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0330 55 4E 43 34 34 34 34-34 34 34 34 34 34 34 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0340 41 4E 54 35 35 35 35 35-35 35 35 35 35 35 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0350 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0360 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| 0370 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | SON1111111111... DAD2222222222... HOM3333333333... UNC4444444444... ANT5555555555... | | | |
| -SY | | | | | | | | | | | | | |
| Y=0000 300 Set IY to 0300H (pointer to OBJECT) | | | | | | | | | | | | | |
| -G193/196 | | | | | | | | | | | | | |
| P=0196 0196' Run 'INSERT' | | | | | | | | | | | | | |
| -DM400 | | | | | | | | | | | | | |
| 0400 53 4F 4E 31 31 31 31 31-31 31 31 31 31 31 00 00 00 | | | | | | | | | | | | | |
| 0410 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0420 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0430 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0440 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0460 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| -SY | | | | | | | | | | | | | |
| Y=0300 310 Set IY to 0310H (next OBJECT) | | | | | | | | | | | | | |
| -G193/196 | | | | | | | | | | | | | |
| P=0196 0196' Run 'INSERT' | | | | | | | | | | | | | |
| -DM400 | | | | | | | | | | | | | |
| 0400 53 4F 4E 31 31 31 31 31-31 31 31 31 31 31 44 41 44 | | | | | | | | | | | | | |
| 0410 32 32 32 32 32 32 32 32-32 32 32 32 32 32 00 00 00 | | | | | | | | | | | | | |
| 0420 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0430 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0440 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0460 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| *** (More insertions) *** | | | | | | | | | | | | | |
| -DM400 | | | | | | | | | | | | | |
| 0400 53 4F 4E 31 31 31 31 31-31 31 31 31 31 31 44 41 44 | | | | | | | | | | | | | |
| 0410 32 32 32 32 32 32 32 32-32 32 32 32 32 32 55 4E 43 34 34 34 | | | | | | | | | | | | | |
| 0420 34 34 34 34 34 34 34 34-4D-4F 4D 33 33 33 33 33 33 33 | | | | | | | | | | | | | |
| 0430 33 33 33 33 41 4E 54 35-35 35 35 35 35 35 35 35 35 35 | | | | | | | | | | | | | |
| 0440 35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0450 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0460 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| 0470 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00 | | | | | | | | | | | | | |
| Table configuration after several inserts | | | | | | | | | | | | | |
| SON111111111111DAD | | | | | | | | | | | | | |
| 2222222222UNC444 | | | | | | | | | | | | | |
| 4444444HOM333333 | | | | | | | | | | | | | |
| 3333ANT5555555555 | | | | | | | | | | | | | |
| 5..... | | | | | | | | | | | | | |

Fig. 9.17: Simple List—A Sample Run

-SY
Y=0340 320
-G190/193

P=0193 0193' Run 'SEARCH'

- Reg D shows that Object was found

-DR Register contents
Z N A=4D BC=02FF DE=FF0D HL=034D S=0100 P=0193 0193' CALL 0135
A'=00 B'=0000 D'=0000 H'=0000 X=0422 Y=0320 I=00
(0135')

—Address of Object

-G196/199

P=0199 0199' Run 'DELETE'

Table configuration after deletion

| ID400 | after deletion | | | | | | | | | | | | | | |
|------------------|----------------|----|----|----|----|----|----|-------|-------|----|----|----|----|----|----|
| 0400 | 53 | 4F | 4E | 31 | 31 | 31 | 31 | 31-31 | 31 | 31 | 31 | 31 | 44 | 41 | 44 |
| 0410 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32-32 | 32 | 55 | 4E | 43 | 34 | 34 | 34 |
| 0420 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 41-4E | 54 | 35 | 35 | 35 | 35 | 35 |
| 0430 | 35 | 35 | 35 | 35 | 41 | 4E | 54 | 35-35 | 35 | 35 | 35 | 35 | 35 | 35 | 35 |
| 0440 | 35 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0450 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0460 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0470 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

-SY
Y=0240 340
-G184/188

Delete last entry in table

Note: no apparent
change in table
configuration

-DM189S1
0189 03 ← Memory location 'TABLEN' — shows true length of table
-G190/193

P=0193 0193' Run 'SEARCH' for deleted Object

D shows that Object was not found

-DR
Z N A=55 BC=0OFF DE=000D HL=0441 S=0100 P=0193 0193' CALL 0135
A'=00 B'=0000 D'=0000 H'=0000 X=041A Y=0340 I=00 (0135')

Fig. 9.17: Simple List— A Sample Run (cont.)

ALPHABETIC LIST

The alphabetic list, or “table,” unlike the previous one, keeps all its elements sorted in alphabetic order. This allows the use of faster search techniques than the linear one. A binary search is used here.

Searching

The search algorithm is a classic binary search. Let us recall that the technique is essentially analogous to the one used to find a name in a telephone book. One usually starts somewhere in the middle of the book, and then, depending on the entries found there, goes either backwards or forward to find the desired entry. This method is fast and reasonably simple to implement.

The binary search flowchart is shown in Fig. 9.18, and the program is shown in Fig. 9.23.

This list keeps the entries in alphabetical order and retrieves them by using a binary or “logarithmic” search. An example is shown in Figure 9.19. The search is somewhat complicated by the need to keep track of several conditions. The major problem to be avoided is searching for an object that is not there. In such a case, the entries with immediately higher and lower alphabetic values could be alternately tested forever. To avoid this, a flag is maintained in the program to preserve the value of the carry flag after an unsuccessful comparison. When the INCMNT value, which shows by how much the pointer will next be incremented reaches a value of “1”, another flag called “CLOSENOW”, which we will abbreviate to “CLOSE”, is set to the value of the COMPRES flag. Thus, since all further increments will be “1”, if the pointer goes past the point where the object should be, COMPRES will no longer equal CLOSE and the search will terminate. This feature also enables the NEW routine to determine where the logical and physical pointers are located, relative to where the object will go.

Thus, if the OBJECT searched for is not in the table, and the running pointer is incremented by one, the CLOSE flag will be set. On the next pass of the routine, the result of the comparison will be opposite to the previous one. The two flags will no longer match, and the program will exit indicating “not found”.

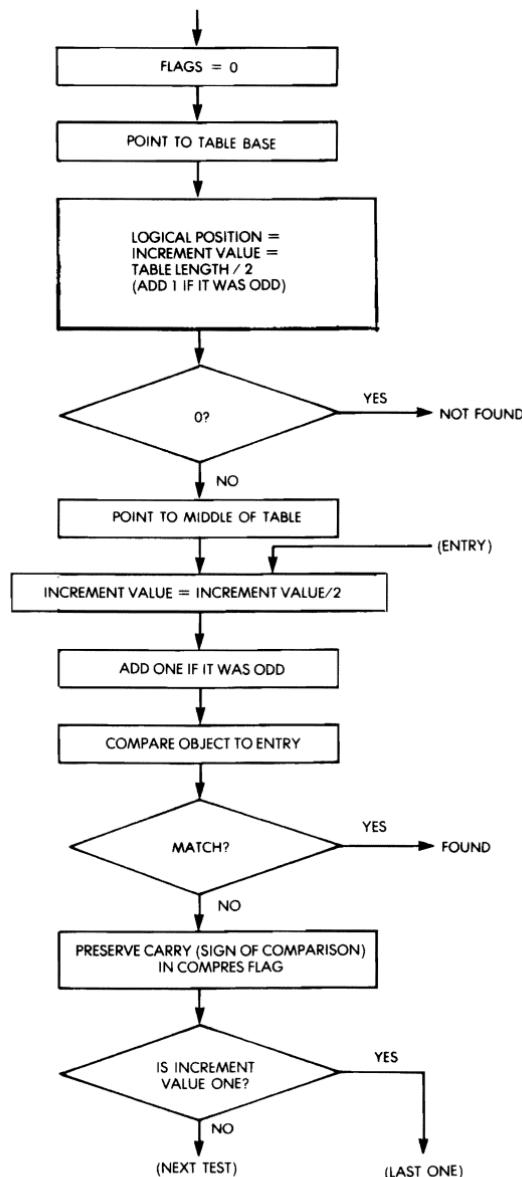


Fig. 9.18: Binary Search Flowchart

PROGRAMMING THE Z80

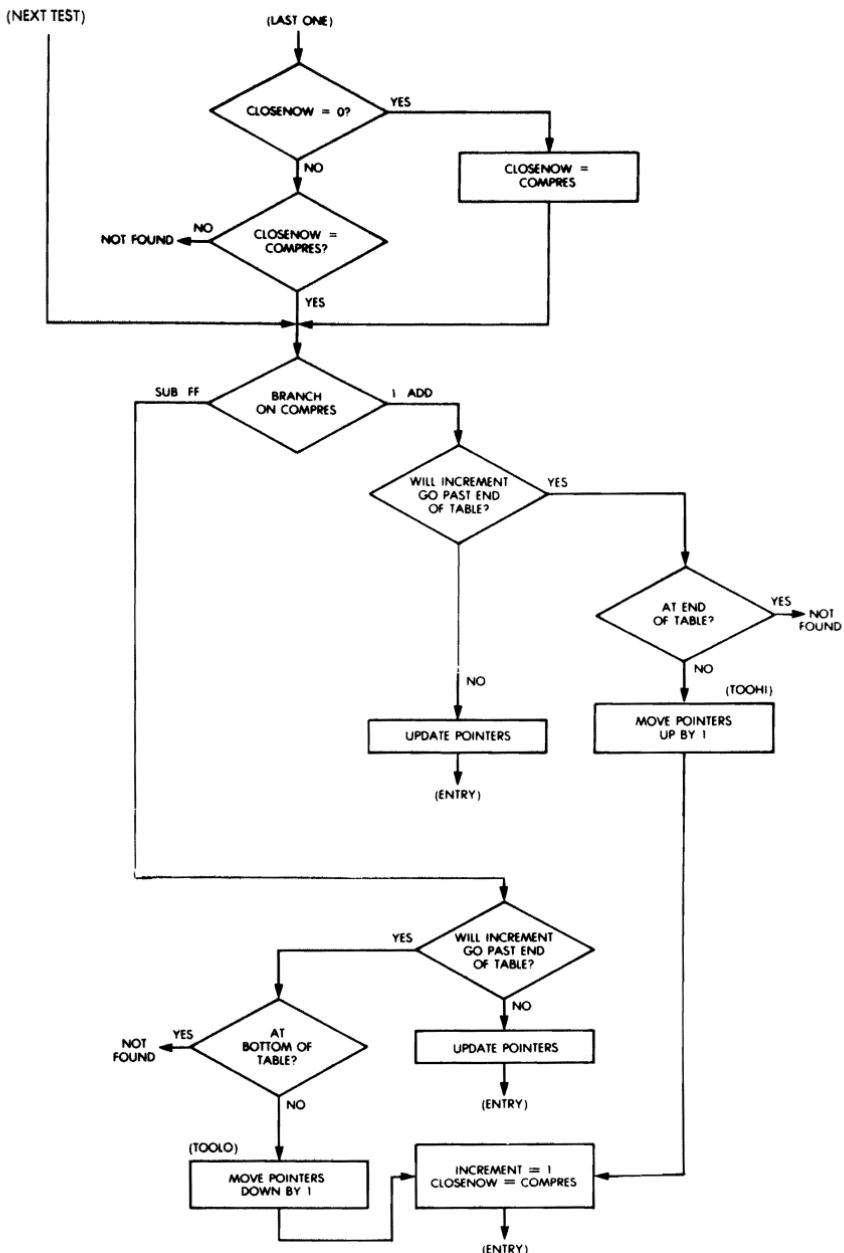


Fig. 9.18: Binary Search Flowchart (cont.)

The other major problem that must be dealt with is the possibility of running off one end of the table when adding or subtracting the increment value. This is solved by performing a test "add" or "subtract" using the logical pointer and length value which record the actual number of entries, not the physical positions in memory used by the physical pointers.

In summary, two flags are used by the program to memorize infor-

| | | |
|--------|-----|------|
| (0121) | LD | A, C |
| | SRL | A |
| | ADC | 0 |
| | LD | C, A |

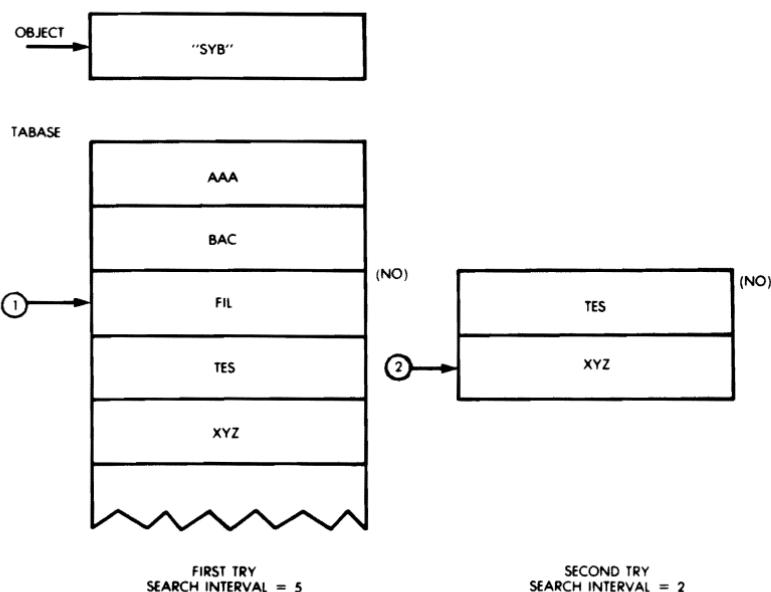


Fig. 9.19: A Binary Search

mation: COMPRES and CLOSE. The COMPRES flag is used to preserve the fact that the carry was either "0" or "1" after the most recent comparison. This determines if the element under test was larger or smaller than the one with which it was compared. The C indicates the relation. Whenever the carry C was "1", and the element was smaller than the object COMPRES is set to "1". Whenever the carry C was "0", indicating that the element was greater than the object, COMPRES will be set to "FF".

The second flag used by the program is CLOSE. This flag is set equal to COMPRESS when the search increment INCMNT becomes equal to "1". It will detect the fact that the element has not been found if COMPRES is not equal to CLOSE the next time around.

Other variables used by the program are:

LOGPOS which indicates the logical position in the table
(element number)

INCMNT which represents the value by which the running
pointer will be incremented or decremented if
the next comparison fails

TABLEN represents as usual the total length of the list.

LOGPOS and INCMNT will be compared to TABLEN in order to assure that the limits of the list are not exceeded.

The program called "SEARCH" is shown on Figure 9.23. It resides at memory locations 0100 to 01CF, and deserves to be studied with care, as it is much more complex than in the case of a linear search.

An additional complication is due to the fact that the search interval may at times be either even or odd. When it is odd, a correction must be introduced. (It cannot, for instance, point to the middle element of a four-element list.) When it is odd, a "trick" is used to point to the middle element: the division by 2 is accomplished by a right shift. The bit "falling off" into the carry after the SRL instruction will be "1" if the interval was odd. It is merely added to the pointer.

The OBJECT is then matched against the entry in the middle of the new search interval. If the comparison succeeds, the program exits. Otherwise ("NOGOOD"), the carry is set to "0" if the OBJECT is less than the entry. Whenever the INCMNT becomes "1", the CLOSE flag (which had been initialized to "0") is then checked to see if it was set. If it was not, it gets set. If it was set, a check is run to determine whether we passed the location where the OBJECT should have been but is not.

Also note that when the carry was “1”, the running pointer will point to the entry below the OBJECT.

Element Insertion

In order to insert a new element, a binary search is conducted. If the element is found in the table, it does not need to be inserted. (We assume here that all elements are distinct). If the element was not found in the table, it must be inserted immediately before or immediately after the last element to which it was compared. The value of the COMPRES flag after the search indicates whether it should be inserted immediately before or immediately afterwards. All the elements following the new location where it is going to be placed are moved down by one block position, and the new element is inserted.

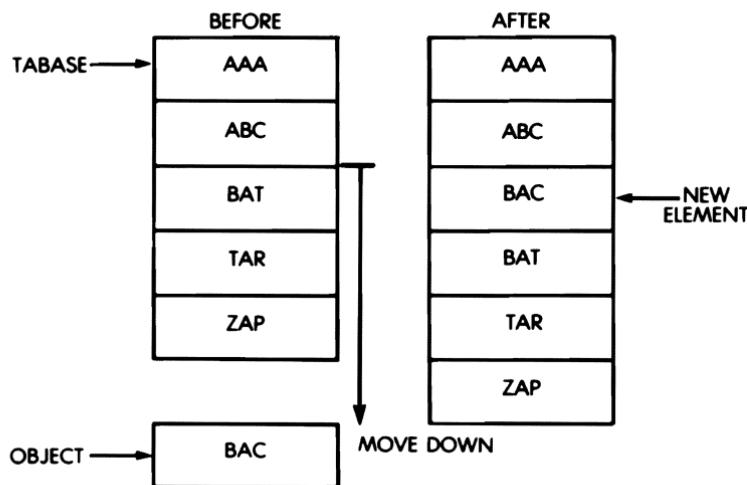


Fig. 9.20: Insert: “BAC”

The insertion process is illustrated in Figure 9.20, and the corresponding program appears in Figure 9.23.

The program is called NEW, and starts at memory location 01D0. Note that the automated Z80 instructions LDDR and LDIR are used for efficient block transfers.

Element Deletion

Similarly, a binary search is conducted to find the object. If the search fails, it does not need to be deleted. If the search succeeds, the element is deleted, and all the following elements are moved up by one block position. A corresponding example is shown in Figure 9.21, and the program appears in Figure 9.23. The flowchart is shown in Fig. 9.22.

The program is called “DELETE” and resides at address 0221. A sample run of the above programs is shown in Fig. 9.24.

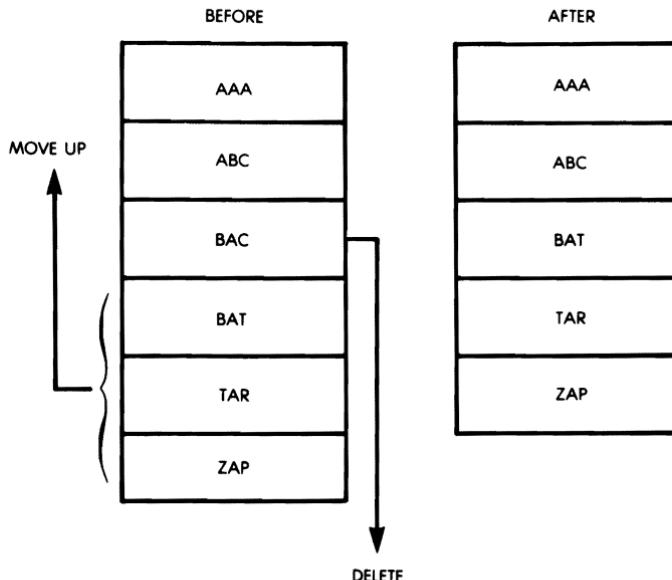


Fig. 9.21: Delete “BAC”

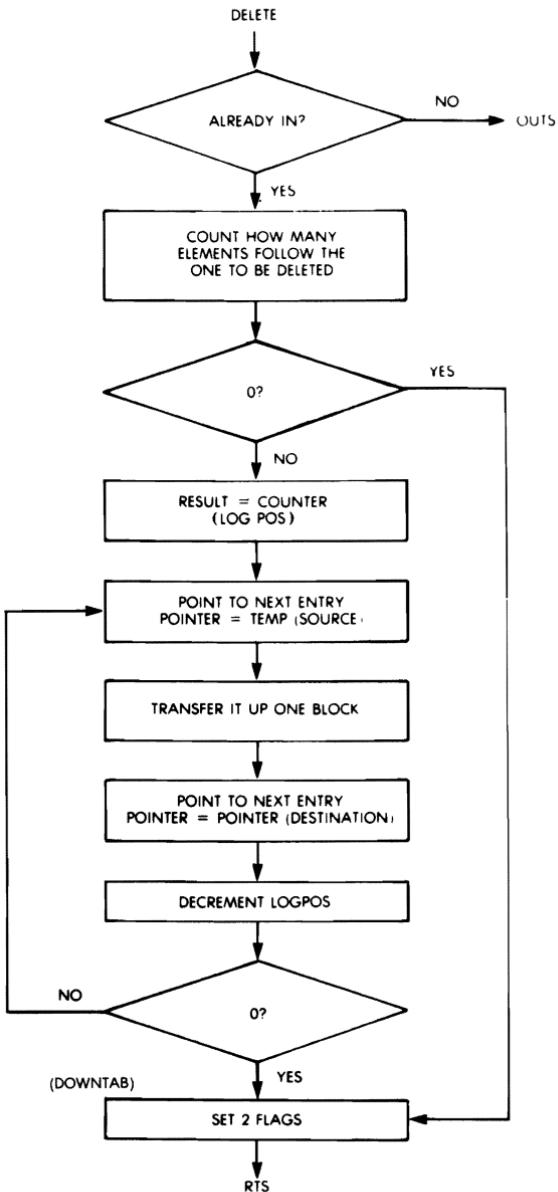


Fig. 9.22: Deletion Flowchart (Alphabetic List)

PROGRAMMING THE Z80

```

0000      ORG 0100H
        (024A) CLOSENOW DL ENDED
        (024B) COMPRES DL ENDED+1
        (024C) TABLEN DL ENDED+2
        (024D) TABASE DL ENDED+3
        (024F) ENTLEN DL ENDED+5
;
0100 3E00    SEARCH LD A,0
0102 324A02  LD (CLOSENOW)+A ;ZERO FLAG LOCATIONS
0105 324B02  LD (COMPRES),A
0108 57      LD D,A
0109 244D02  LD HL,(TABASE) ;INITIALIZE HL
010C 3A4C02  LD A,(TABLEN)
010F CB3F    SRL A
0111 CE00    ADC 0
0113 4F      LD C,A
0114 47      LD B,A
0115 CAB0A1  JP Z,NOTFOUND ;CHECK IF LENGTH IS ZERO
0118 5F      LD E,A
0119 1D      DEC E
011A CDBD01  CALL MULT
011D 19      ADD HL,DE ;SET HL TO MIDDLE OF TABLE
011E E5      ENTRY PUSH HL ;LOAD HL INTO IX
011F DDE1    POP IX
0121 79      LD A,C
0122 CB3F    SRL A
0124 CE00    ADC 0
0126 4F      LD C,A
0127 DD7E00  LD A,(IX+0) ;COMPARE FIRST LETTER
0128 FDRE00  CP (IY+0)
012D C24201  JP NZ,NOGOOD
0130 DD7E01  LD A,(IX+1) ;COMPARE 2ND LETTER
0133 FDRE01  CP (IY+1)
0136 C24201  JP NZ,NOGOOD
0139 DD7E02  LD A,(IX+2) ;COMPARE 3RD LETTER
013C FDRE02  CP (IY+2)
013F CABC01  JP Z,FOUND
0142 3E01    NOGOODB LD A,1 ;SET COMPARE RESULT FLAG TO
0144 DA4901  JP C,TESTS ;..RESULT OF COMPARE (1,FF)
0147 3EFF    LD A,OFFH
0149 324B02  TESTS LD (COMPRES),A
014C 79      LD A,C
014D 3D      DEC A
014E C26901  JP NZ,NEXTTEST
0151 3A4A02  LD A,(CLOSENOW) ;YES, IS CLOSE FLAG SET?
0154 A7      AND A
0155 CA6301  JP Z,NOTCLOSE
0158 57      LD D,A
0159 3A4B02  LD A,(COMPRES) ;YES, SEE IF HAVE PASSED WHERE
015C 92      SUB D ;..ENTRY SHOULD BE BUT ISN'T
015D CA6901  JP Z,NEXTTEST
0160 C3BA01  JP NOTFOUND
0163 3A4B02  NOTCLOSE LD A,(COMPRES) ;SET CLOSE FLAG TO DIRECTION OF
0166 324A02  LD (CLOSENOW),A ;..SEARCH TO PREVENT REPETITION
0169 DDE5    NEXTTEST PUSH IX ;PREPARE HL AND DE FOR ADD OR
016B E1      POP HL ;..SUB OF INCREMENT VALUE
016C 59      LD E,C
016D CDBD01  CALL MULT
0170 3A4B02  LD A,(COMPRES) ;TEST IF WANT TO ADD OR SUB
0173 3C      INC A
0174 C29601  JP NZ,ADDIT
0177 78      LD A,B
0178 91      SUB C ;TEST TO SEE IF SUB WILL RUN
0179 CA8501  JP Z,TOOLOW ;..OFF BOTTOM OF TABLE
017C D48501  JP C,TOOLOW
017F 47      LD B,A ;SET NEW LOGICAL POSITION VALUE
0180 ED52    SBC HL,DE ;CHANGE ADDRESS ITSELF
0182 C31E01  JP ENTRY
0185 78      TOOLOW LD A,B ;SEE IF POSITION IS 1
0186 3D      DEC A
0187 CAB0A1  JP Z,NOTFOUND ;IF SO, EXIT
018A ED5B4F02 LD DE,(ENTLEN) ;JUST SUB 1 ENTRY POSITION
018E 37      SCF
018F 3F      CCF
0190 ED52    SBC HL,DE ;CHANGE LOGICAL POSITION
0192 05      DEC B
0193 C3AF01  JP REALCLOS
;
```

Fig. 9.23: Binary Search Program

```

0196 3A4C02 ADDIT LD A,(TABLEN) ;TEST TO SEE IF CURRENT POSITION
0199 90 SUB R ;...PLUS INCREMENT WILL GO PAST
019A 91 SUB C ;...END OF THE TABLE
019B DAA501 JP C,TOOHIGH
019E 19 ADD HL,DE ;IS OK, CHANGE ACTUAL ADDRESS
019F 78 LD A,B ;CHANGE LOGICAL POS. VALUE
01A0 81 ADD C
01A1 47 LD B,A
01A2 C31E01 JP ENTRY
01A5 81 TOOHIGH ADD C ;SEE IF POSITION IS AT TOP OF
01A6 CABAO1 JP Z,NOTFOUND ;...TABLE (SAME AS TABLEN=R)
01A9 ED5B4F02 LD DE,(ENTLEN) ;ADD 1 ENTRY POSITION
01AB 19 ADD HL,DE
01AE 04 INC R ;INCREMENT LOGICAL POSITION
01AF 0E01 REALCLOS LD C,I ;SET INCREMENT TO 1
01B1 3A4B02 LD A,(COMPRES) ;SET CLOSE FLAG TO COMPARE
01B4 324A02 LD (CLOSENOW),A ;...RESULT
01B7 C31E01 JP ENTRY
01BA 16FF NOTFOUND LD D,OFFH
01BC C9 FOUND RET D,OFFH
;
01BD E5 MULT PUSH HL ;MULTIPLIES E BY (ENTLEN)
01BE C5 PUSH BC ;...VALUE IN DE ON EXIT
01BF 1600 LD R,0
01C1 210000 LD HL,0000
01C4 ED4B4F02 LD BC,(ENTLEN)
01CB 41 ADD B,C
01C9 J9 ADDM ADD HL,DE
01CA 10FD DJNZ ADDM
01CC C1 POP BC
01CD ER EX DE,HL
01CE E1 POP HL
01CF C9 RET I
;
;
;
;
01D0 CD0001 NEW CALL SEARCH ;SEE IF OBJECT IS ALREADY THERE
01D3 14 INC H
01D4 C22002 JP NZ,OUT
01D7 3A4C02 LD A,(TABLEN) ;CHECK FOR 0 TABLE
01DA A7 AND A
01DB CA0C02 JP Z,INSERT
01DE 3A4B02 LD A,(COMPRES)
01E1 3C INC A
01E2 CAE001 JP Z,HISTIDE
01E5 ED5B4F02 LD DE,(ENTLEN) ;COMPRES=1, SET HL ABOVE WHERE
01E9 19 ADD HL,DE ;OBJECT SHOULD GO
01EA C3EE01 JP SETUP
01ED 05 HISTDE DEC R ;COMPRES=0, SET R FOR SURTRACT
01EE 3A4C02 LD A,(TABLEN) ;SEE HOW MANY ENTRIES ARE LEFT
01F1 90 SUB R
01F2 CA0C02 JP Z,INSERT ;SET HL TO LAST POSITION IN LAST
01F5 SF LD E,A ;ENTRY
01F6 C0BD01 CALL MULTI
01F9 19 ADD HL,DE
01FA 2B DEC HL
01FB EB EX DE,HL ;SET DE 1 ENTRY ABOVE HL
01FC 2A4F02 LD HL,(ENTLEN)
01FF 19 ADD HL,DE
0200 ER EX DE,HL
0201 ED4B4F02 MOVEM LD BC,(ENTLEN) ;SHIFT UP ONE ENTRY OF MEMORY
0205 ED88 LDDR
0207 3D INC A
0208 C20102 JP NZ,MOVEM ;REPEAT IF NECESSARY
020B 23 INC HL ;HL IS FRONT OF NOW EMPTY SPACE
020C FDE5 INSERT PUSH IY ;LOAD OBJECT INTO EMPTY SPACE
020E D1 POP DE
020F ER EX DE,HL
0210 ED4B4F02 LD BC,(ENTLEN)
0214 ED80 LDIR
0216 3A4C02 LD A,(TABLEN) ;INCREMENT TABLE LENGTH
0219 3C INC A
021A 324C02 LD (TABLEN)+A
021D 01FFFF LD BC,OFFFH ;SHOW THAT IT WAS DONE
0220 C9 OUT RET
;
;
;
;
```

Fig. 9.23: Binary Search Program (cont.)

PROGRAMMING THE Z80

```

0221 C00001    DELETE   CALL    SEARCH      ;GET ADDRESS OF OBJECT
0224 14          INC      D        ;SEE IF OBJECT IS THERE
0225 CA4902    JP       Z,DUTE
0228 ED5B4F02    LD       DE,(ENTLEN)
022C EB          EX       DE,HL
022D 19          ADD      HL,DE      ;DE IS LOC. OF OBJECT, HL IS
022E 3A4C02    LD       A,(TABLEN) ;..ONE ENTRY ABOVE
0231 90          SUB      B
0232 CA3F02    JP       Z,DOWNTAB ;SEE HOW MANY ENTRIES ARE LEFT
0235 ED4B4F02    LD       BC,(ENTLEN)
0239 EDB0    LDIR      ;SHIFT DOWN 1 ENTRY LENGTH
023B 3D          DEC      A
023C C23502    JP       NZ,SHIFTIN
023F 3A4C02    LD       A,(TABLEN) ;DECREMENT TABLE LENGTH
0242 3D          DEC      A
0243 324C02    LD       (TABLEN),A
0246 01FFFF    LD       BC,OFFFFF ;SHOW THAT ACTION WAS TAKEN
0249 C9          DUTE      RET
024A (0000)    ENDED      END

SYMBOL TABLE
ADDEM 01C9    ADDIT 0196    CLOSEN 024A    COMPRE 024B    DELETE 0221
DOWNTA 023F    ENDED 024A    ENTLEN 024F    ENTRY 011E    FOUND 01BC
HISIDE 01ED    INSERT 020C    MOVEM 0201    MULT 01BD    NEW 01D0
NEXTES 0169    NOGOOD 0142    NOTCLO 0163    NOTFOU 01BA    OUT 0220
DUTE 0249     REALCL 01AF    SEARCH 0100    SETUP 01EE    SHIFTI 0235
TABASE 024D    TABLEN 024C    TESTS 0149    TOOHIGH 01AS    TOOLOW 0185

```

Fig. 9.23: Binary Search Program (cont.)

LINKED LIST

The linked list is assumed to contain, as usual, the three alphanumeric characters for the label, followed by one to 250 bytes of data, followed by a two-byte pointer which contains the starting address of the next entry, and lastly followed by a one-byte marker. Whenever this one-byte marker is set to “1”, it will prevent the insert-routine from substituting a new entry in the place of the existing one.

Further, a directory contains a pointer to the first entry for each letter of the alphabet, in order to facilitate retrieval. It is assumed in the program that the labels are ASCII alphabetic characters. All pointers at the end of the list are set to a NIL value which has been chosen here to be equal to the table base, as this value should never occur within the linked list.

The insertion and the deletion programs perform the obvious pointer manipulations. They use the flag INDEXED to indicate if a pointer pointing to an object came from a previous entry in the list or from the directory table. The corresponding programs are shown in Figure 9.29.

The data structure is shown in Figure 9.25.

DATA STRUCTURES

| -DIM400 | Initial table |
|---------|---|
| 0400 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0410 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0420 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0430 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0440 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0450 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0460 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0470 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |

| Listing of Objects and their locations in memory | | | | | | | | | | | | | | |
|--|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|
| -IM300 | | | | | | | | | | | | | | |
| 0300 | 53 | 4F | 4E | 31 | 31 | 31 | 31 | 31 | 31-31 | 31 | 31 | 31 | 31 | 00 |
| 0310 | 44 | 41 | 44 | 32 | 32 | 32 | 32 | 32 | 32-32 | 32 | 32 | 32 | 32 | 00 |
| 0320 | 4D | 4F | 4D | 33 | 33 | 33 | 33 | 33 | 33-33 | 33 | 33 | 33 | 33 | 00 |
| 0330 | 55 | 4E | 43 | 34 | 34 | 34 | 34 | 34 | 34-34 | 34 | 34 | 34 | 34 | 00 |
| 0340 | 41 | 4E | 54 | 35 | 35 | 35 | 35 | 35 | 35-35 | 35 | 35 | 35 | 35 | 00 |
| 0350 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 |
| 0360 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 |
| 0370 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 |

-SY
Y=0000 320
-G263/266
F=0266 0266' }

| Table after insertion | | | | | | | | | | | | | |
|-----------------------|----|----|----|----|----|----|----|----|-------|----|----|----|----|
| M0M3333333333. | | | | | | | | | | | | | |
| 0400 | 4D | 4F | 4D | 33 | 33 | 33 | 33 | 33 | 33-33 | 33 | 33 | 33 | 33 |
| 0410 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0420 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0430 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0440 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0450 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0460 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |
| 0470 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 |

-SY
Y=0320 310
-G263/266 }
P=0266 0266' } Run 'INSERT' on another Object

• • • (additional inserts) • • •

Fig. 9.24: Alphabetic List—A Sample Run

PROGRAMMING THE Z80

-DM400

| | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|
| 0400 | 41 | 4E | 54 | 35 | 35 | 35 | 35 | 35 | 35-35 | 35 | 35 | 35 | 44 | 41 | 44 |
| 0410 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32-32 | 32 | 4D | 4F | 4D | 33 | 33 |
| 0420 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33-34 | 4F | 31 | 31 | 31 | 31 | 31 |
| 0430 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31-34 | 43 | 34 | 34 | 34 | 34 | 34 |
| 0440 | 34 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0450 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0460 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0470 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |

-SY
Y=0340 300
-G260/263
P=0263 0263' } Run 'SEARCH' for "SON" (at address 0300)

-DR

| | | | | | | | | | | |
|-------|---------|---------|---------|---------|---------|--------|---------|-------|------|------|
| Z | N | A=4E | BC=0401 | DE=000D | HL=0427 | S=0100 | P=0263 | 0263' | CALL | 01D0 |
| A'=00 | B'=0000 | D'=0000 | H'=0000 | X=0427 | Y=0300 | I=00 | (01D0') | | | |

Found

Address of Object in table
(verify in Table above that it is "SON")

-G266/269
P=0269 0269' Run 'DELETE' on "SON"

-DM400

| | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|
| 0400 | 41 | 4E | 54 | 35 | 35 | 35 | 35 | 35 | 35-35 | 35 | 35 | 35 | 44 | 41 | 44 |
| 0410 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32-32 | 32 | 4D | 4F | 4D | 33 | 33 |
| 0420 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33-34 | 4F | 31 | 31 | 31 | 31 | 31 |
| 0430 | 34 | 34 | 34 | 34 | 34 | 35 | 4E | 43 | 34-34 | 34 | 34 | 34 | 34 | 34 | 34 |
| 0440 | 34 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0450 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0460 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0470 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |

Table configuration after deletion. Note:
that UNC was shifted up. The last UNC entry must be disregarded

-ANT555555555555DAD
-222222222222DM333
-333333UNC44444444
-4444UNC444444444444

-G260/263

Try run of "SEARCH" again (on "SON")
P=0263 0263'

-DR

| | | | | | | | | | | |
|-------|---------|---------|---------|---------|---------|--------|---------|-------|------|------|
| S | N | A=FE | BC=0401 | DE=FF0D | HL=0427 | S=0100 | P=0263 | 0263' | CALL | 01D0 |
| A'=00 | B'=0000 | D'=0000 | H'=0000 | X=0427 | Y=0300 | I=00 | (01D0') | | | |

Not found

-G263/266

Re-insert Object ("SON")
P=0266 0266'

-DM400

| | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|
| 0400 | 41 | 4E | 54 | 35 | 35 | 35 | 35 | 35 | 35-35 | 35 | 35 | 35 | 44 | 41 | 44 |
| 0410 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32-32 | 32 | 4D | 4F | 4D | 33 | 33 |
| 0420 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33-34 | 4F | 31 | 31 | 31 | 31 | 31 |
| 0430 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31-34 | 43 | 34 | 34 | 34 | 34 | 34 |
| 0440 | 34 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0450 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0460 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0470 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00-00 | 00 | 00 | 00 | 00 | 00 | 00 |

Shows that action was executed

-DR

| | | | | | | | | |
|-------|---------|---------|---------|--------|--------|-------|---------|------|
| A=05 | BC=FFFF | DE=043A | HL=030D | S=0100 | P=0266 | 0266' | CALL | 0221 |
| A'=00 | B'=0000 | D'=0000 | H'=0000 | X=0427 | Y=0300 | I=00 | (0221') | |

Fig. 9.24: Alphabetic List—A Sample Run (cont.)

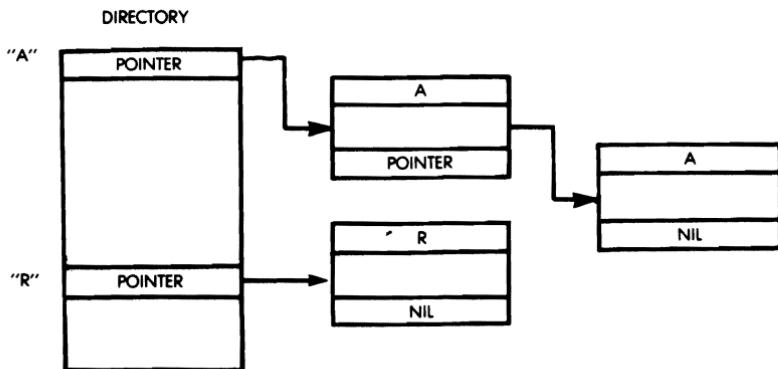
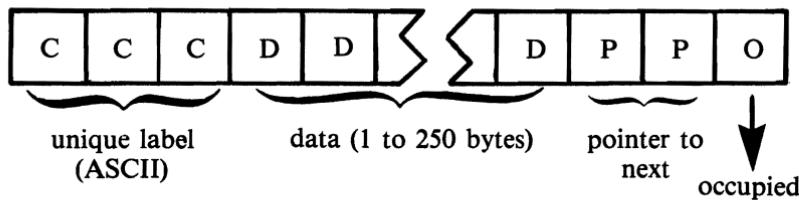


Fig. 9.25: Linked List Structure

An application for this data structure would be a computerized address book, where each person is represented by a unique three-letter code (perhaps the usual initials) and the data field contains a simplified address, plus the telephone number (up to 250 characters). Let us examine the structure in more detail. The entry format is:



As usual the conventions are:

ENTLEN: total element length (in bytes)

TABASE: address of base of list

The address of the OBJECT is always assumed to reside in the IY register prior to entering the program. Here, REFBASE points to the base address of the directory, or "reference table."

Each two-byte address within this directory points to the first occurrence of the letter to which it corresponds in the list. Thus, each group

of entries with an identical first letter in their labels actually forms a separate list within the whole structure. This feature facilitates searching and is analogous to an address book. Note that no data are moved during an insert or delete. Only pointers are changed, as in every well-behaved linked list structure.

If no entry starting with a specific letter is found, or if there is no entry alphabetically following an existing one, their pointers will point to the beginning of the table (= "NIL"). At the bottom of the table, by convention a value is stored such that the absolute value of the difference between it and "Z" is greater than the difference between "A" and "Z". This represents an End Of Table (EOT) marker. The EOT value is assumed here to occupy the same amount of memory as a normal entry but could be just one byte if desired. The letters are assumed to be alphabetic letters in ASCII code. Changing this would require changing the constant in the PRETAB routine.

The end-of-table marker is set to the value of the beginning of the table ("NIL").

By convention, the "NIL pointers", found at the end of a string, or within a directory location which does not point to a string, are set to the value of the table base to provide a unique identification. Another convention could be used. In particular, a different marker for EOT results in some space savings, as no NIL entries need be kept for non-existing entries.

Insertion and deletion are performed in the usual way (see Part I of this chapter) by merely modifying the required pointers. The INDEXED flag is used to indicate if the pointer to the object is in the reference table or another string element.

Searching

The SEARCH program resides at memory locations 0100 to 0155 and uses subroutine PRETAB at address 01D2.

The search principle is straightforward:

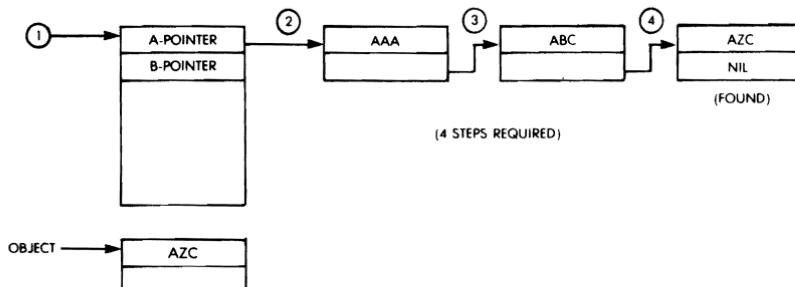
1—Get the directory entry corresponding to the letter of the alphabet in the first position of the OBJECT's label.

2—Get the pointer. Access the element. If NIL, the entry does not exist.

3—If not NIL, match the element against the OBJECT. If a match is found, the search has succeeded. If not, get the pointer to the next entry down the list.

4—Go back to 2.

An example is shown in Figure 9.26.

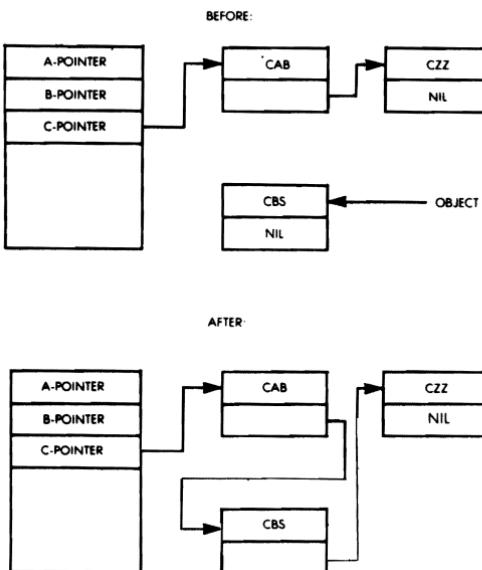
**Fig. 9.26: Linked List—A Search**

Inserting

The insertion is essentially a search followed by an insertion once a “NIL” has been found.

A block of storage for the new entry is allocated past the EOT marker by looking for an occupancy marker set at “available”.

The program is called “NEW” in Figure 9.29 and resides at addresses 0156 to 1A3. An example is shown in Figure 9.27.

**Fig. 9.27: Linked List: Example of Insertion**

Deleting

The element is deleted by setting its occupancy marker to “available” and adjusting the pointer to it from the directory or else the previous element.

The program is called “DELETE”, and resides at addresses 01A4 to 01D1.

An example of a deletion is shown in Figure 9.28.

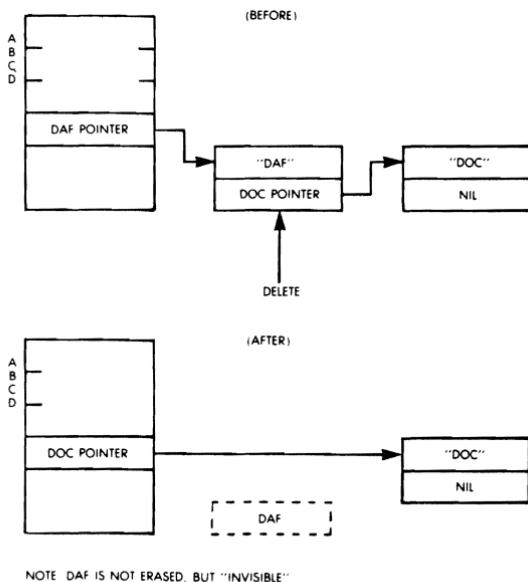


Fig. 9.28: Example of Deletion (Linked List)

```

0000' (01E7) INDEXEDI ING 0100H
(01EB) TARASE INI ENDER11
(01EA) REFRASE INI ENDER13
(01EC) ENTLEN DL ENDER15
;
0100 3E00 SEARCH LD A,0
0102 47 LD R,A
0103 3C INC A
0104 32E701 LD (INDEXEDI)+A
0107 C01201 CALL PRETR
010A 1A LD A,(DE)
010B 6F LD L,A
010C 13 INC DE
010D 1A LD A,(DE)
010E 67 LD H,A
010F E5 PUSH HL
0110 DDE1 POP TX
0112 DB2E00 COMPARE LD A,(TX10)
0115 FE7C CP ZCH
0117 D25501 JP NC,NOTFOUND
011A D02E00 LD A,(TX10)
011B FDBE00 CP (TY+0)
0120 DA3E01 JP C,NOGOOD
0123 C25501 JP NZ,NOTFOUND
0124 DB2E01 LD A,(TX11)
0129 FDBE01 CP (TY+1)
012C DA3E01 JP C,NOGOOD
012F C25501 JP NZ,NOTFOUND
0132 DB2E02 LD A,(TX12)
0135 FDBE02 CP (TY+2)
0138 CA5301 JP Z,FOUND
013B D25501 JP NC,NOTFOUND
013E DDE5 NOGOOD PUSH TX
0140 D1 POP DE
0141 2AEC01 LD HL,(ENTLEN)
0144 19 ADD HL,DE
0145 4E LD C,(HL)
0146 23 INC HL
0147 46 LD B,(HL)
0148 C5 PUSH BC
0149 DDE1 POP TX
014B 3E00 LD A,0
014P 32E701 LD (INDEXEDI)+A
0150 C01201 JP COMPARE
0153 06FF FOUND LD B,OFFH
0155 C9 NOTFOUND RET
;
;
;
;
0156 CD0001 NEW CALL SEARCH
0159 04 TNC R
015A CAA301 JP Z,OUT
015D 1S PUSH DE
015E 2AE801 LD HL,(TARASE)
0161 FB NEXTONE EX DE,HL
0162 2AEC01 LD HL,(ENTLEN)
0165 23 INC HL
0166 23 INC HL
0167 23 INC HL
0168 19 ADD HL,DE
0169 2F LD A,(HL)
016A 3B DEC A
016B CA6101 JP Z,NEXTONE
016E 13 INC DE
016F D5 PUSH DE
0170 DDE5 PUSH TY
0172 E1 POP HL
0173 ED4BEC01 LD BC,(ENTLEN)
0177 ED80 LHIR
0179 DDE5 PUSH TX
017B E1 POP HL
017C EB EX DE,HL
017D 73 LD (HL)+E
017E 23 INC HL
017F 22 LD (HL)+I
0180 23 INC HL
0181 3E01 LD (HL)+I
;
```

; INITIALIZE FLAGS

; GET ADDR OF INDEX POINTER

; MOVE POINTER CONTENTS TO HL

; LOOK AT FIRST LETTER OF ENTRY

; SEE IF IS END MARKER

; COMPARE FIRST LETTERS

; COMPARE 2ND LETTERS

; COMPARE 3RD LETTERS

; JUMP TO POINTER OF ENTRY

; PUT POINTER VALUE IN BC

; LOAD TX WITH POINTER

; RESET FLAG

; SEE WHERE OBJECT SHOULD GO

; STORE ADDR. OF PREVIOUS ENTRY

; FIND SPACE IN TABLE FOR NEW

; MOVE TO END OF NEXT ENTRY

; ADD 3 FOR REAL LENGTH OF ENTRY

; IF SOMETHING IS THERE, TRY AGAIN

; SAVE POSITION OF EMPTY SPACE

; MOVE TY TO HL

; MOVE OBJECT INTO TABLE

; PUT ADDR OF ENTRY AFTER OBJECT

; ..AT POINTER POSITION

; SET OCCUPANCY MARKER

Fig. 9.29: Linked List—The Programs

PROGRAMMING THE Z80

```

0183 E1          POP   HL      ;GET ADDR OF WHERE THIS SPACE IS
0184 3AE701       LD    A,(INDEXED) ;SEE WHAT PREVIOUS POINTERS MUST
0187 3D          DEC   A        ;..BE SET
0188 CA9801       JP    Z,SETINX
0188 E3          EX    (SP),HL  ;GET ADDR OF ENTRY PREVIOUS TO
018C ED5BEC01     LD    DE,(ENTLEN) ;..OBJECT & MOVE TO POINTER AREA
0190 19          ADD   HL,DE
0191 D1          POP   DE
0192 73          LD    (HL),E   ;RETRIEVE ADDR OF OBJECT
0193 23          INC   HL
0194 72          LD    (HL),D   ;PUT IT AT POINTER POSITION
0195 C3A001       JP    FINISH
0198 C1          SETINX POP   BC      ;CLEAR OUT STACK
0199 CDD201       CALL  PRETAB ;GET INDEX ADDRESS
019C EB          EX    DE,HL  ;LOAD HL INTO IT
019D 73          LD    (HL),E
019E 23          INC   HL
019F 72          LD    (HL),D
01A0 01FFFF       FINISH LD    BC:0FFFFH ;SHOW THAT IT WAS DONE
01A3 C9          OUT   RET
;
;
;

01A4 CD0001       DELETE CALL  SEARCH ;GET ADDRESS OF OBJECT
01A7 04          INC   B      ;SEE IF JT IS THERE
01A8 C2D101       JP    NZ,DUTE
01AB DDE5          PUSH  IX
01AD E1          POP   HL      ;SET HL TO POINTER AREA OF OBJECT
01AE ED4BEC01     LD    BC,(ENTLEN)
01B2 09          ADD   HL,BC
01B3 4E          LD    C,(HL) ;RETRIEVE POINTER
01B4 23          INC   HL
01B5 46          LD    B,(HL)
01B6 23          INC   HL
01B7 3600       LD    (HL),0  ;REMOVE OCCUPANCY MARKER
01B9 3AE701       LD    A,(INDEXED) ;SEE IF INDEX NEEDS CHANGING
01B8 3D          DEC   A
01B0 C2C701       JP    NZ,CHANGEM
01C0 CDD201       CALL  PRETAB ;YES,PUT ADDR INTO HL
01C3 EB          EX    DE,HL
01C4 C3CB01       JP    MOVIN
01C7 2AEC01       CHANGEM LD    HL,(ENTLEN) ;SET HL TO PONTER OF PREVIOUS
01CA 19          ADD   HL,DE
01CB 71          MOVIN LD    (HL),C ;PUT ADDR OF NEXT INTO WHATEVER
01CC 23          INC   HL ;..(EITHER INDEX OR ENTRY)
01CD 70          LD    (HL),B
01CE 01FFFF       DUTE   LD    BC:0FFFFH
01D1 C9          DUTE   RET
;
;
;

01D2 E5          PRETAB PUSH  HL      ;GET FIRST LETTER OF OBJECT
01D3 FD7E00       LD    A,(IY+0) ;REMOVE ASCII LEADER
01D6 3D          DEC   A
01D7 D640       SUB   40H
01D9 CB27       SLA   A      ;MULTIPLY BY 2
01DB 2AAE01       LDII  HL,(REFBASE)
01DE 85          ADD   L
01DF 6F          LD    L,A
01E0 D2E401       JP    NC,FIXUP
01E3 24          INC   H
01E4 EB          FIXUP EX    DE,HL
01E5 E1          POP   HL
01E6 C9          RET
;
01E7 (0000)      ENDER END
;

SYMBOL TABLE
CHANGE 01C7  COMPAR 0112  DELETE 01A4  ENDER 01E7  ENTLEN 01EC
FINISH 01A0  FIXUP 01E4  FOUND 0153  INDEXE 01E7  MOVIN 01CB
NEW    0156  NEXTON 0161  NOGOOD 013E  NOTFOU 0155  OUT   01A3
DUTE   01D1  PRETAB 01D2  REFBS  01EA  SEARCH 0100  SETINX 0198
DATABASE 01E8

```

Fig. 9.29: Linked List—The Programs (cont.)

The Objects in memory**Listing of Objects and their locations in memory**

| -IM300 | SON11111111111... |
|--|-------------------|
| 0300 53 4F 4E 31 31 31 31 31 31-31 31 31 31 31 31 00 00 00 | DAD2222222222... |
| 0310 44 41 44 32 32 32 32 32 32-32 32 32 32 32 00 00 00 | MOM3333333333... |
| 0320 4D 4F 4D 33 33 33 33 33 33-33 33 33 33 33 00 00 00 | UNC4444444444... |
| 0330 55 4E 43 34 34 34 34 34 34-34 34 34 34 34 00 00 00 | ANT5555555555... |
| 0340 41 4E 54 35 35 35 35 35 35-35 35 35 35 35 00 00 00 | AAA6666666666... |
| 0350 41 41 41 36 36 36 36 36 36-36 36 36 36 36 00 00 00 | AZZ7777777777... |
| 0360 41 5A 5A 37 37 37 37 37 37-37 37 37 37 37 00 00 00 | STD8888888888... |
| 0370 53 49 44 38 38 38 38 38 38-38 38 38 38 38 00 00 00 | |

EOT character in initial table

| -IM400 |
|--|
| 0400 7B 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0410 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0420 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0430 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0440 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0450 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0460 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0470 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |

Initial Directory

| -IM500 |
|--|
| 0500 00 04 00 04 00 04 00 04 00-04 00 04 00 04 00 04 00 04 |
| 0510 00 04 00 04 00 04 00 04 00-04 00 04 00 04 00 04 00 04 |
| 0520 00 04 00 04 00 04 00 04 00-04 00 04 00 04 00 04 00 04 |
| 0530 00 04 00 04 00 04 00 04 00-04 00 04 00 04 00 04 00 04 |
| 0540 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0550 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0560 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0570 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |

Occupancy markers**Pointers****Table configuration after several insertions.**

| -IM400 |
|--|
| 0400 7B 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0410 41 4E 54 35 35 35 35 35 35-35 35 35 35 35 35 70 04 01 |
| 0420 44 41 44 32 32 32 32 32 32-32 32 32 32 32 32 00 04 01 |
| 0430 41 41 41 36 36 36 36 36 36-36 36 36 36 36 36 10 04 01 |
| 0440 53 4F 4E 31 31 31 31 31 31-31 31 31 31 31 31 00 04 01 |
| 0450 4D 4F 4D 33 33 33 33 33 33-33 33 33 33 33 33 00 04 01 |
| 0460 53 49 44 38 38 38 38 38 38-38 38 38 38 38 38 40 04 01 |
| 0470 41 5A 5A 37 37 37 37 37 37-37 37 37 37 37 37 00 04 01 |

-SY
Y=0360 310
-0226/229 } Delete an entry
P=0229 0229 }

| -IM400 |
|--|
| 0400 7B 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 |
| 0410 41 4E 54 35 35 35 35 35 35-35 35 35 35 35 35 70 04 01 |
| 0420 44 41 44 32 32 32 32 32 32-32 32 32 32 32 32 00 04 01 |
| 0430 41 41 41 36 36 36 36 36 36-36 36 36 36 36 36 10 04 01 |
| 0440 53 4F 4E 31 31 31 31 31 31-31 31 31 31 31 31 00 04 01 |
| 0450 4D 4F 4D 33 33 33 33 33 33-33 33 33 33 33 33 00 04 01 |
| 0460 53 49 44 38 38 38 38 38 38-38 38 38 38 38 38 40 04 01 |
| 0470 41 5A 5A 37 37 37 37 37 37-37 37 37 37 37 37 00 04 01 |

Only change**Fig. 9.30: Linked List—A Sample Run**

PROGRAMMING THE Z80

Fig. 9.30: Linked List— A Sample Run (cont.)

SUMMARY

The beginning programmer need not concern himself yet with the details of data structures implementation and management. However, efficient programming of non-trivial algorithms requires a good understanding of data structures. The actual examples presented in this chapter should help the reader achieve such an understanding and solve all the common problems encountered with reasonable data structures.

10

PROGRAM DEVELOPMENT

INTRODUCTION

All the programs we have studied and developed so far have been developed by hand without the aid of any software or hardware resource. The only improvement over straight binary coding has been the use of mnemonic symbols, those of the assembly language. For effective software development, it is necessary to understand the range of hardware and software development aids. It is the purpose of this chapter to present and evaluate these aids.

BASIC PROGRAMMING CHOICES

Three basic alternatives exist: writing a program in binary or hexadecimal, writing it in assembly-level language, or writing it in a high-level language. Let us review these alternatives.

Hexadecimal Coding

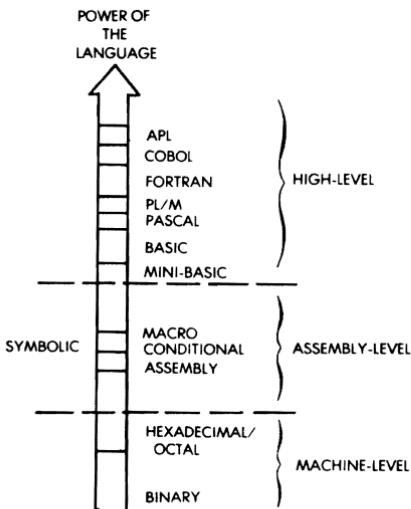
The program will normally be written using assembly language mnemonics. However, most low-cost, one-board computer systems do not provide an assembler. The assembler is the program which will automatically translate the mnemonics used for the program into the required binary codes. When no assembler is available, this translation from mnemonics into binary must be performed by hand. Binary is *unpleasant* to use and error-prone, so that hexadecimal is normally used. It has been shown in Chapter 1 that one hexadecimal digit will represent four binary bits. Two hexadecimal digits will, therefore, be used to represent the contents of every byte. As an example, the table showing the hexadecimal equivalent of the Z80 instructions appears in the Appendix.

In short, whenever the resources of the user are limited and no assembler is available, he will have to translate the program by hand into hexadecimal. This can reasonably be done for a small number of instructions, such as, perhaps, 10 to 100. For larger programs, this process is tedious and error-prone, so that it tends not to be used. However, nearly all single-board microcomputers require the entry of programs in hexadecimal mode. They are not equipped with an assembler and a full alphanumeric keyboard, in order to limit their cost.

In summary, hexadecimal coding is not a desirable way to enter a program in a computer. It is simply an economical one. The cost of an assembler and the required alphanumeric keyboard is traded-off against increased labor required to enter the program in the memory. However, this does not change the way the program itself is written. *The program is still written in assembly-level language* so that it can be examined by the human programmer and be meaningful.

Assembly Language Programming

Assembly-level programming covers both programs that may be entered in hexadecimal and those that may be entered in symbolic assembly-level form in the system. Let us now examine the entry of a program directly in its assembly language representation. An assembler program must be available. The assembler will read each of the mnemonic instructions of the program and translate it into the required bit pattern using 1 to 5 bytes, as specified by the encoding of the instructions. In addition, a good assembler will offer a number of additional facilities for writing the program. These will be reviewed in the section on the assembler below. In particular, *directives* are available which will modify the value of symbols. Symbolic addressing may be used and a branch to a symbolic location may be specified. During the debugging phase, when a user may remove or add instructions, it will not be necessary to rewrite the entire program if an extra instruction is inserted between a branch and the point to which it branches, as long as symbolic labels are used. The assembler will take care of automatically adjusting all the labels during the translation process. In addition, an assembler allows the user to debug his program in symbolic form. A disassembler may be used to examine the contents of a memory location and reconstruct the assembly-level instruction that it represents. The various software resources normally available on a system will be reviewed below. Let us now examine the third alternative.

**Fig. 10.1: Programming Levels**

High-Level Language

A program may be written in a high-level language such as BASIC, APL, PASCAL, or others. Techniques for programming in these various languages are covered by specific books and will not be reviewed here. We will, therefore, only briefly review this mode of programming. A high-level language offers powerful instructions which make programming much easier and faster. These instructions must then be translated by a complex program into the final binary representation that a microcomputer can execute. Typically, each high-level instruction will be translated into a large number of individual binary instructions. The program which performs this automatic translation is called a *compiler* or an *interpreter*. A compiler will translate all the instructions of a program in sequence into object code. In a separate phase, the resulting code will then be executed. By contrast, an interpreter will interpret a single instruction, then execute it, then "translate" the next one, then execute it. An interpreter offers the advantage of interactive response, but results in low efficiency compared to a compiler. These topics will not be studied further here. Let us revert to the programming of an actual microprocessor in the assembly-level language.

SOFTWARE SUPPORT

We will review here the main software facilities which are (or should be) available in the complete system for convenient software development. Some of the definitions have already been introduced. They will be summarized here and the rest of the important programs will be defined before we proceed.

The *assembler* is the program which translates the mnemonic representation of instructions into their binary equivalent. It normally translates one symbolic instruction into one binary instruction (which may occupy 1, 2 or 3 bytes). The resulting binary code is called *object code*. It is directly executable by the microcomputer. As a side effect, the assembler will also produce a complete symbolic listing of the program, as well as the equivalence tables to be used by the programmer and the symbol occurrence list in the program. Examples will be presented later in this chapter.

In addition, the assembler will list syntax errors such as instructions misspelled or illegal, branching errors, duplicate labels or missing labels.

It will not delete *logical errors* (this is *your problem*).

A *compiler* is the program which translates high-level language instructions into their binary form.

An *interpreter* is a program similar to a compiler, which also translates high-level instructions into their binary form but does not keep the intermediate representation and executes them immediately. In fact, it often does not even generate any intermediate code, but rather executes the high-level instructions directly.

A *monitor* is the basic program which is indispensable for using the hardware resources of this system. It continuously monitors the input devices for input and manages the rest of the devices. As an example, a minimal monitor for a single-board microcomputer, equipped with a keyboard and with LED's, must continuously scan the keyboard for a user input and display the specified contents on the light-emitting diodes. In addition, it must be capable of understanding a number of limited commands from the keyboard, such as START, STOP, CONTINUE, LOAD MEMORY, EXAMINE MEMORY. On a large system, the monitor is often qualified as the *executive* program, when complex file management or task scheduling is also provided. The overall set of facilities is called an *operating system*. If files are residing on a disk, the operating system is qualified as the *disk operating system*, or DOS.

An *editor* is the program designed to facilitate the entry and the modification of text or programs. It allows the user to enter characters conveniently, append them, insert them, add lines, remove lines, search for characters or strings. It is an important resource for convenient and effective text entry.

A *debugger* is a facility necessary for debugging programs. When a program does not work correctly, there may typically be no indication whatsoever of the cause. The programmer, therefore, wishes to insert breakpoints in his program in order to suspend the execution of the program at specified addresses, and to be able to examine the contents of registers or memory at this point. This is the primary function of a debugger. The debugger allows for the possibility of suspending a program, resuming execution, examining, displaying and modifying the contents of registers or memory. A good debugger will be equipped with a number of additional facilities, such as the ability to examine data in symbolic form, hex, binary, or other usual representations, as well as to enter data in this format.

A *loader*, or *linking loader*, will place various blocks of object code at specified positions in the memory and adjust their respective symbolic pointers so that they can reference each other. It is used to relocate programs or blocks in various memory areas. A *simulator* or an *emulator* program is used to simulate the operation of a device, usually the microprocessor, in its absence, when developing a program on a simulated processor prior to placing it on the actual board. Using this approach, it becomes possible to suspend the program, modify it, and keep it in RAM memory. The disadvantages of a simulator are that:

1—It usually simulates only the processor itself, not input/output devices

2—The execution speed is slow, and one operates in simulated time. It is therefore not possible to test real-time devices, and synchronization problems may still occur even though the logic of the program may be found correct.

An *emulator* is essentially a simulator in real time. It uses one processor to simulate another one, and simulates it in complete detail.

Utility routines are essentially all the routines which are necessary in most applications and that the user wishes the manufacturer had provided!

They may include multiplication, division and other arithmetic operations, block move routines, character tests, input/output device handlers (or "drivers"), and more.

THE PROGRAM DEVELOPMENT SEQUENCE

We will now examine a typical sequence for developing an assembly-level program. We will assume that all the usual software facilities are available in order to demonstrate their value. If they should not be available in a particular system, it will still be possible to develop programs, but the convenience will be decreased and, therefore, the amount of time necessary to debug the program is likely to be increased.

The normal approach is to first design an algorithm and define the data structures for the problem to be solved. Next, a comprehensive set of flowcharts is developed which represents the program flow. Finally, the flowcharts are translated into the assembly-level language for the microprocessor; this is the coding phase.

Next, the program has to be entered on the computer. We will examine in the next section the hardware options to be used in this phase.

The program is entered in RAM memory of the system under the control of the editor. Once a section of the program, such as one or more subroutines, has been entered, it will be tested.

First, the assembler will be used. If the assembler did not already reside in the system, it would be loaded from an external memory, such as a disk. Then, the program will be assembled, i.e., translated into a binary code. This results in the object program, ready to be executed.

One does not normally expect a program to work correctly the first time. To verify its correct operation, a number of breakpoints will normally be set at crucial locations where it is easy to test whether the intermediate results are correct. The debugger will be used for this purpose. Breakpoints will be specified at selected locations. A "Go" command will then be issued so that program execution is started. The program will automatically stop at each of the specified breakpoints. The programmer can then verify, by examining the contents of the registers, or memory, that the data so far is correct. If it is correct, we proceed until the next breakpoint. Whenever we find incorrect data, an error in the program has been detected. At this point, the programmer normally refers to his program listing and verifies whether his coding has been correct. If no error can be found in the programming, the error might be a logical one and one might refer to the flowchart. We will assume here that the flowcharts have been checked by hand and are assumed to be reasonably correct. The error is likely to come from the coding. It will, therefore, be necessary to modify a section of the program. If the symbolic representation of the program is still in the memory, we will

simply re-enter the editor and modify the required lines, then go through the preceding sequence again. In some systems, the memory available may not be large enough, so that it is necessary to flush out the symbolic representation of the program onto a disk or cassette prior to executing the object code. Naturally, in such a case, one would have to reload the symbolic representation of the program from its support medium prior to entering the editor again.

The above procedure will be repeated as long as necessary until the results of the program are correct. Let us stress that prevention is much more effective than cure. A correct design will typically result in a program which runs correctly very soon after the usual typing mistakes or obvious coding errors have been removed. However, sloppy design may result in programs which will take an extremely long time to be debugged. The debugging time is generally considered to be much longer than the actual design time. In short, it is always worth investing more time in the design in order to shorten the debugging phase.

However, using this approach, it is possible to test the overall organization of the program, but not to test it in real time with input/output devices. If input/output devices are to be tested, the direct solution consists of transferring the program onto EPROM's and installing it on the board and then watching whether it works.

There is a better solution. It is the use of an *in-circuit emulator*. An in-circuit emulator uses the Z80 microprocessor (or any other one) to emulate a Z80 in (almost) real time. It emulates the Z80 physically. The emulator is equipped with a cable terminated by a 40-pin connector, exactly identical to the pin-out of a Z80. This connector can then be inserted on the real application board that one is developing. The signals generated by the emulator will be exactly those of the Z80, only perhaps a little slower. The essential advantage is that the program under test will still reside in the RAM memory of the development system. It will generate the real signals which will communicate with the real input/output devices that one wishes to use. As a result, it becomes possible to keep developing the program using all the resources of the development system (editor, debugger, symbolic facilities, file system) while testing input/output in real time.

In addition, a good emulator will provide special facilities, such as a *trace*. A trace is a recording of the last instructions or status of various data busses in the system prior to a breakpoint. In short, a trace provides the film of the events that occurred prior to the breakpoint or the malfunction. It may even trigger a scope at a specified address or upon the occurrence of a specified combination of bits. Such a facility is of

PROGRAMMING THE Z80

great value, since when an error is found it is usually too late. The instruction, or the data, which caused the error has occurred prior to the detection. The availability of a trace allows the user to find which segment of the program caused the error to occur. If the trace is not long enough, we will simply set an earlier breakpoint.

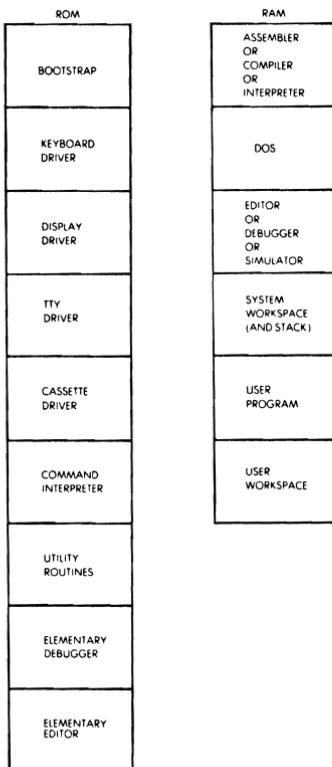


Fig. 10.2: A Typical Memory Map

This completes our description of the usual sequence of events involved in developing a program. Let us now review the hardware alternatives available for developing programs.

HARDWARE ALTERNATIVES

Single-Board Microcomputer

The single-board microcomputer offers the lowest cost approach to program development. It is normally equipped with a hexadecimal keyboard, plus some function keys, plus 6 LED's which can display address and data. Since it is equipped with a small amount of memory, an assembler is not usually available. At best, it has a small monitor and virtually no editing or debugging facilities, except for a very few commands. All programs must, therefore, be entered in hexadecimal form. They will also be displayed in hexadecimal form on the LED's. A single-board microcomputer has, in theory, the same hardware power as any other computer. Simply because of its restricted memory size and keyboard, it does not support all the usual facilities of a larger system and makes program development much longer. Because it is tedious to develop programs in hexadecimal format, a single board microcomputer is best suited for education and training where programs of limited length have to be developed and their short length is not an obstacle to programming. Single-boards are probably the cheapest way to learn programming by doing. However, they cannot be used for complex program development unless additional memory boards are attached and the usual software aids are made available.

The Development System

A development system is a microcomputer system equipped with a significant amount of RAM memory (32K, 48K) as well as the required input/output devices, such as a CRT display, a printer, disks, and, usually, a PROM programmer, as well as, perhaps, an in-circuit emulator. A development system is specifically designed to facilitate program development in an industrial environment. It normally offers all, or most, of the software facilities that we have mentioned in the preceding section. In principle, it is the ideal software development tool.

The limitation of a microcomputer development system is that it may not be capable of supporting a compiler or an interpreter. This is because a compiler typically requires a very large amount of memory, often more than is available on the system. However, for developing programs in assembly-level language, it offers all the required facilities. But because development systems sell in relatively small numbers compared to hobby computers, their cost is significantly higher.

Hobby-Type Microcomputers

The hobby-type microcomputer hardware is naturally exactly analogous to that of a development system. The main difference lies in the fact that it is normally not equipped with the sophisticated software development aids which are available on an industrial development system. As an example, many hobby-type microcomputers offer only elementary assemblers, minimal editors, minimal file systems, no facilities to attach a PROM programmer, no in-circuit emulator, no powerful debugger. They represent, therefore, an intermediate step between the single-board microcomputer and the full microprocessor development system. For a user who wishes to develop programs of modest complexity, they are probably the best compromise, since they offer the advantage of low cost and a reasonable array of software development tools, even though they are quite limited as to their convenience.

Time-Sharing System

It is possible to rent terminals from several companies which will connect to time-sharing networks. These terminals share the time of the larger computer and benefit from all the advantages of large installations. *Cross assemblers* are available for all microcomputers on virtually all commercial time-sharing systems. A cross assembler is simply an assembler for, say, a Z80 which resides, for example, in an IBM370. Formally, a cross assembler is an assembler for microprocessor X, which resides on processor Y. The nature of the computer being used is irrelevant. The user still writes a program in Z80 assembly-level language, and the cross assembler translates it into the appropriate binary pattern. The difference, however, is that the program cannot be executed at this point. It can be executed by a simulated processor, if one is available, provided it does not use any input/output resources. This solution is used, therefore, only in industrial environments.

In-House Computer

Whenever a large in-house computer is available, cross assemblers may also be available to facilitate program development. If such a computer offers time-shared service, this option is essentially analogous to the one above. If it offers only batch service, this is probably one of the most inconvenient methods of program development, since submitting programs in batch mode at the assembly level for a microprocessor results in a very long development time.

Front Panel or No Front Panel?

The front panel is a hardware accessory often used to facilitate program debugging. It has traditionally been a tool for conveniently displaying the binary contents of a register or of memory. However, all the functions of the control panel may be accomplished from a terminal, and the dominance of CRT displays now offers a service almost equivalent to the control panel by displaying the binary value of bits. The additional advantage of using the CRT display is that one can switch at will from binary representation to hexadecimal, to symbolic, to decimal (if the appropriate conversion routines are available, naturally). The disadvantage of the CRT is that one must hit several keys to obtain the appropriate display rather than turn a knob. However, since the cost of providing a control panel is quite substantial, most recent microcomputers have abandoned this debugging tool. The value of the control panel is often considered more on the basis of emotional arguments influenced by one's own past experience than by the use of reason. It is not indispensable.

Summary of Hardware Resources

Three broad cases may be distinguished. If you have only a minimal budget and if you wish to learn how to program, buy a single-board microcomputer. Using it, you will be able to develop all the simple programs in this book and many more. Eventually, however, when you want to develop programs of more than a few hundred instructions, you will feel the limitations of this approach.

If you are an industrial user, you will need a full development system. Any solution short of the full development system will cause a significantly longer development time. The trade-off is clear: hardware resources vs. programming time. Naturally, if the programs to be developed are quite simple, a less expensive approach may be used. However, if complex programs are to be developed, it is difficult to justify any hardware savings when buying a development system, since the programming costs will be by far the dominant cost of the project.

For a personal computerist, a hobby-type microcomputer will typically offer sufficient, although minimal, facilities. Good development software is still to come for many of the hobby computers. The user will have to evaluate his system in view of the comments presented in this chapter.

Let us now analyze in more detail the most indispensable resource: the assembler.

THE ASSEMBLER

We have used assembly-level language throughout this book without presenting the formal syntax or definition of assembly-level language. The time has come to present this definition. An assembler is designed to allow the convenient symbolic representation of the user program, and yet to make it simple for the assembler program to convert these mnemonics into their binary representation.

Assembler Fields

When typing in a program for the assembler, we have seen that fields are used. They are:

The label field, optional, which may contain a symbolic address for the instruction that follows.

The instruction field, which includes the opcode and any operands. (A separate operand field may be distinguished.)

The comment field, far to the right, which is optional and is intended to clarify the program.

These fields are shown on the programming form in Figure 10.3.

Once the program has been fed to the assembler, the assembler will produce a *listing* of it. When generating a listing, the assembler will provide three additional fields, usually on the left of the page. An example appears on Figure 10.4. On the far left is the line number. Each line which has been typed by the programmer is assigned a symbolic line number.

The next field to the right is the actual address field, which shows in hexadecimal the value of the program counter which will point to that instruction.

Moving still further to the right, we find the hexadecimal representation of the instruction.

This shows one of the possible uses of an assembler. Even if we are designing programs for a single-board microcomputer which accepts only hexadecimal, we should still write the program in assembly-level language, providing we have access to a system equipped with an assembler. We can then run the programs on the system, using the assembler. The assembler will automatically generate the correct hexadecimal codes on our system. This shows, in a simple example, the value of additional software resources.

| | | | | COMMENTS | |
|--------------------|---|---|---|--------------------|---------|
| | | | | SYMBOLIC OPCODE | OPERAND |
| | | | | LABEL | |
| HEX INSTRUCTION | 1 | 2 | 3 | 4 | |
| ADDRESS | | | | | |

Fig. 10.3: Microprocessor Programming Form

Tables

When the assembler translates the symbolic program into its binary representation, it performs two essential tasks:

- 1—It translates the mnemonic instructions into their binary encoding.
- 2—It translates the symbols used for constants and addresses into their binary representation.

In order to facilitate program debugging, the assembler shows at the end of the listing the equivalence between the symbol used and its hexadecimal value. This is called the symbol table.

Some symbol tables will not only list the symbol and its value, but also the line numbers where the symbol occurs, thereby providing an additional facility.

Error Messages

During the assembly process, the assembler will detect syntax errors and include them as part of the final listing. Typical diagnostics include: undefined symbols, label already defined, illegal opcode, illegal address, illegal addressing mode. Many more detailed diagnostics are naturally desirable and are usually provided. They vary with each assembler.

The Assembly Language

Opcodes have already been defined. We will here define the symbols, constants and operators which may be used as part of the assembler syntax.

Symbols

Symbols are used to represent numerical values, either data or addresses. Symbols may include up to six characters, and must start with an alphabetical character. The characters are restricted to letters of the alphabet and numbers. Also, the user may not choose names identical to the opcodes utilized by the Z80, the names of registers such as A,B,C,D,E,H,L, BC, DE, HL, AF, BC, DE, IX, IY, SP, as well as the various short names used as pseudo-operators by the assembler. The names of these assembler “directives” are listed below in the corresponding sections. Also, the abbreviations used to designate the flags should not be used as symbols: C,Z,N,PE,NC,P,PO,NZ,M.

Assigning a Value to a Symbol

Labels are special symbols whose values do not need to be defined by the programmer. The value will automatically be defined by the assembler program whenever it finds that label. The label value thus automatically corresponds to the address of the instruction generated at the line where it appears. Special pseudo-instructions are available to force a new starting value for labels, or to assign them a specific value.

```
CROMEMCO CBOS Z80 ASSEMBLER version 02.15          PAGE 0001
0000'      0001      ORG    0100H
             (0200)    0002  MPRAD  DL     0200H
             (0202)    0003  MPDAD  DL     0202H
             (0204)    0004  RESAD  DL     0204H
             0005  ;
0100  ED4B0002  0006  MP488  LD     BC,(MPRAD)   ;LOAD MULTIPLIER INTO C
0104  0608      0007  LD     B,B      ;B IS BIT COUNTER
0106  ED5B0202  0008  LD     DE,(MPDAD)  ;LOAD MULTIFLICAND INTO E
010A  1600      0009  LD     D,O      ;CLEAR D
010C  210000    0010  LD     HL,0      ;SET RESULT TO 0
010F  CB39      0011  MULT   SRL   C       ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001      0012  JR     NC,NOADD ;TEST CARRY
0113  19        0013  ADD   HL,DE    ;ADD MPD TO RESULT
0114  CB23      0014  NOADD  SLA   E       ;SHIFT MPD LEFT
0116  CB12      0015  RL    D       ;SAVE BIT IN D
0118  05        0016  DEC   B       ;DECREMENT SHIFT COUNTER
0119  C20F01    0017  JP     NZ,MULT  ;DO IT AGAIN IF COUNTER <> 0
011C  220402    0018  LD     (RESAD),HL ;STORE RESULT
011F  (0000)    0019  END

Errors      0
```

Fig. 10.4: Assembler Output—An Example

PROGRAMMING THE Z80

However, other symbols used for constants or memory addresses must be defined by the programmer prior to their use.

A special assembler *directive* may be used to assign a value to any symbol. A *directive* is essentially an instruction to the assembler which will not be translated into an executable statement. For example, the constant LOG will be defined as:

```
LOG EQU 3002H
```

This assigns the value 3002 hexadecimal to the variable LOG. The assembler directives will be examined in detail in a later section.

Constants or Literals

Constants may traditionally be expressed either in decimal, in hexadecimal, in octal, or in binary, or as alphanumeric strings. In order to differentiate between the base used to represent the number, a symbol must be used. To load “0” into the accumulator, we will simply write:

```
LD A, 0
```

Optionally a “D” may be used at the end of the constant.

A hexadecimal number will be terminated by the symbol “H”. To load the value “FF” into the accumulator, we will write:

```
LD A, 0FFH
```

An octal symbol is terminated by the symbol “0” or “Q”. A binary symbol is terminated by “B”.

For example, in order to load the value “11111111” into the accumulator, we will write:

```
LD A, 11111111B
```

Literal ASCII characters may also be used in the literal field. The ASCII symbol must be enclosed in single quotes.

For example, in order to load the symbol “S” into the accumulator, we will write:

```
LD A, 'S'
```

Exercise 10.1: Will the following two instructions load the same value in the accumulator: LD A, ‘5’, and LD A, 5H?

Note that in the Zilog convention, parentheses denote an address. For example:

LD A, (10)

specifies that the accumulator is loaded from the contents of memory location 10 (decimal).

Operators

In order to further facilitate the writing of symbolic programs, assemblers allow the use of operators. At a minimum, they should allow plus and minus so that one can specify, for example:

LD A, (ADDRESS)

LD A, (ADDRESS + 1)

It is important to understand that the expression ADDRESS + 1 will be computed by the assembler in order to determine the actual memory address which must be inserted as the binary equivalent. It will be computed *at assembly time*, not at program-execution time.

In addition, more operators may be available, such as multiply and divide, a convenience when accessing tables in memory. More specialized operators may be also available, such as greater than and less than, which truncate a two-byte value respectively into its high and low byte.

Naturally, an expression must *evaluate* to a positive value. Negative numbers may normally not be used and should be expressed in a hexadeciml format.

Finally, a special symbol is traditionally used to represent the current value of the address of the line: “\$”. This symbol should be interpreted as “current location” (value of PC).

Exercise 10.2: What is the difference between the following instructions?

LD A, 10101010B

LD A, (10101010B)

Exercise 10.3: What is the effect of the following instruction?

JR NC, \$ - 2

Expressions

The Z80 assembler specifications allow a wide range of expressions

with arithmetic and logical operations. The assembler will evaluate the expressions in a left-to-right manner, using the priorities specified by the table in Figure 10.5. Parentheses may be used to enforce a specific order of evaluation. However, the outermost parentheses will denote that the contents are to be treated as an address.

Assembler Directives

Directives are special orders given by the programmer to the assembler, which result either in storing values into symbols or into the memory, or in controlling the execution or printing modes of the assembler. The set of commands which specifically controls the printing modes of the assembler is also called "commands" and is described in a separate section.

To provide a specific example, let us review here the 11 assembler directives available on the Zilog development system:

ORG nn

This directive will set the assembler address counter to the value nn. In other words, the first executable instruction encountered after this directive will reside at the value nn. It can be used to locate different segments of a program at different memory locations.

EQU nn

This directive is used to assign a value to a label.

DEFL nn

This directive also assigns a value nn to a label, but may be repeated within the program with different values for the same label, whereas EQU may be used only once.

DEFB n

This directive assigns eight-bit contents to a byte residing at the current reference counter.

DEFB 'S'

assigns the ASCII value of "S" to the byte.

DEFW nn

This assigns the value nn to the two-byte word residing at the current reference counter and the following location.

| OPERATOR | FUNCTION | PRIORITY |
|------------|-----------------------|----------|
| + | UNARY PLUS | 1 |
| - | UNARY MINUS | 1 |
| .NOT. or \ | LOGICAL NOT | 1 |
| .RES. | RESULT | 1 |
| ** | EXPONENTIATION | 2 |
| * | MULTIPLICATION | 3 |
| / | DIVISION | 3 |
| .MOD. | MODULO | 3 |
| .SHR. | LOGICAL SHIFT RIGHT | 3 |
| .SHL. | LOGICAL SHIFT LEFT | 3 |
| + | ADDITION | 4 |
| - | SUBTRACTION | 4 |
| .AND. or & | LOGICAL AND | 5 |
| .OR. or 1 | LOGICAL OR | 6 |
| .XOR. | LOGICAL XOR | 6 |
| .EQ. or = | EQUALS | 7 |
| .GT. or > | GREATER THAN | 7 |
| .LT. or < | LESS THAN | 7 |
| .UGT. | UNSIGNED GREATER THAN | 7 |
| .ULT. | UNSIGNED LESS THAN | 7 |

Fig. 10.5: Operator Precedence**DEFS nn**

reserves a block of memory size nn bytes, starting at the current value of the reference counter.

DEFM 'S'

stores into memory the string 'S' starting at the current reference counter. It must be less than 63 in length.

MACRO P0 P1...Pn

is used to define a label as a macro, and to define its formal parameter list. Macros are defined in another section below.

END

indicates the end of the program. Any other statements following it will be ignored.

ENDM

is used to mark the end of a macro definition.

Assembler Commands

Commands are used to modify the format of the listing to control the printing modes of the assembler. All commands start with a star in column one. Seven commands are provided by the Z80 assembler. Typical examples are:

EJECT

which causes the listing to move to the top of the next page; and

LIST OFF

which causes the printing to be suspended, effective with this command. The others are: “*HEADING S”, “*LIST ON”, “*MACLIST ON”, “*MACLIST OFF”, “*INCLUDE FILENAME”.

Macros

A macro is simply a name assigned to a group of instructions. It is a convenience to the programmer. If a group of instructions is used several times in a program, we could define a macro to represent them, instead of always having to write this group of instructions.

As an example, we could write:

```
SAVREG MACRO
    PUSH AF
    PUSH BC
    PUSH DE
    PUSH HL
ENDM
```

then simply write the name “SAVREG” instead of the above instructions. Any time that we write SAVREG, the five corresponding lines will get substituted instead of the name. An assembler equipped with a macro facility is called a macro-assembler. When the macro assembler encounters a SAVREG, it performs a mere physical substitution of equivalent lines.

Macro or Subroutine?

At this point, a macro may seem to operate in a way analogous to a subroutine. This is not the case. When the assembler is used to produce the object code, any time that a macro name is encountered, it will be replaced by the actual instructions that it stands for. At execution time, the group of instructions will appear as many times as the name of the macro did.

By contrast, a subroutine is defined only once, and then it can be used repeatedly; the program will jump to the subroutine address. A macro is called an *assembly-time* facility. A subroutine is an *execution-time* facility. Their operation is quite different.

Macro Parameters

Each macro may be equipped with a number of parameters. As an example, let us consider the following macro:

```
SWAP MACRO #M, #N, #T
    LD    A, #M      ; M INTO A
    LD    #T, A      ; A INTO T (=M)
    LD    A, #N      ; N INTO A
    LD    #M, A      ; A INTO M (=N)
    LD    A, #T      ; T INTO A
    LD    #N, A      ; A INTO N (=T)
END      M
```

This macro will result in swapping (exchanging) the contents of memory locations M and N. A swap between two registers, or two memory locations, is an operation which is not provided by the Z80. A macro may be used to implement it. "T" in this instance is simply the name for a temporary storage location required by the program. As an example, let us swap the contents of memory locations ALPHA and BETA. The instruction which does this appears below:

```
SWAP (ALPHA), (BETA), (TEMP)
```

In this instruction, TEMP is the name of some temporary storage location, which we know to be available and which can be used by the macro. The resulting expansion of the macro appears below:

```
LD  A, (ALPHA)
LD  (TEMP), A
LD  A, (BETA)
LD  (ALPHA), A
LD  A, (TEMP)
LD  (BETA), A
```

The value of a macro should now be apparent: it is convenient for the programmer to use pseudo-instructions, which have been defined with macros. In this way, the apparent instruction set of the Z80 can be expanded at will. Unfortunately, one must bear in mind that each macro

directive will expand into whatever number of instructions were used. A macro will, therefore, run more slowly than any single instruction. Because of its convenience for the development of any long program, a macro facility is highly desirable for such applications.

Additional Macro Facilities

Many other directives and syntactic facilities may be added to a simple macro facility; macros may be *nested*, i.e., a macro call may appear within a macro definition. Using this facility, a macro may modify itself with a nested definition! A first call will produce one expansion, whereas subsequent calls will produce a modified expansion of the same macro. This is allowed by the Z80 assembler, but nested definitions are not allowed.

CONDITIONAL ASSEMBLY

Conditional assembly is another facility provided in the Z80 assembly. With a conditional assembly facility, the programmer can devise programs for a variety of cases, and then conditionally assemble the segments of codes required by a specific application. As an example, an industrial user might design programs to take care of any number of traffic lights at an intersection, for a variety of control algorithms. He will then receive the specifications from the local traffic engineer, who specifies how many traffic lights there should be and which algorithms should be used. The programmer will then simply set parameters in his program and assemble conditionally. The conditional assembly will result in a "customized" program which will retain only those routines which are necessary for the solution to the problem.

Conditional assembly is, therefore, of specific value to industrial program generation in an environment where many options exist and where the programmer wishes to assemble portions of programs quickly and automatically in response to external parameters.

Only two conditional pseudo-OPs are provided in the standard micro-assembler version supplied by Zilog. They are respectively:

COND NN and ENDC

where NN represents an expression. The pseudo-OP "COND NN" will result in the evaluation of the expression NN. As long as the expression evaluates to a true value (non-zero), the statement following the COND will be assembled. However, if the expression should be false, i.e., eval-

uate to a zero value, the assembly of all subsequent statements will be disabled up to the ENDC instruction.

ENDC is used to terminate a COND, so that the assembly of subsequent statements is re-enabled. The COND pseudo-OP's cannot be nested.

In theory, more powerful conditional assembly facilities could exist, with "IF" and "ELSE" specification. They may become available in future versions of the assembler.

SUMMARY

This chapter has presented the techniques and the hardware and software tools required to develop a program, along with the various trade-offs and alternatives.

These range at the hardware level from the single-board microcomputer to the full development system; at the software level, from binary coding to high-level programming.

You will have to select them on the basis of your goals and resources.

CHAPTER 11

CONCLUSION

We have now covered all important aspects of programming, from definitions and basic concepts to the internal manipulation of the Z80 registers, to the management of input/output devices, as well as the characteristics of software development aids. What is the next step? Two views can be offered, the first one relating to the development of technology, the second one relating to the development of your own knowledge and skill. Let us address these two points.

TECHNOLOGICAL DEVELOPMENT

The progress of integration in MOS technology makes it possible to implement more and more complex chips. The cost of implementing the processor function itself is constantly decreasing. The result is that many of the input/output chips or the peripheral-controller chips used in a system now incorporate a simple processor. This means that most LSI chips in the system are becoming *programmable*. An interesting conceptual dilemma is now developing. In order to simplify the software design task, as well as to reduce the component count, the new I/O chips now incorporate sophisticated programmable capabilities: many programmed algorithms are now integrated within the chip. However, as a result, the development of programs is complicated by the fact that all these input/output chips are radically different and need to be studied in detail by the programmer! *Programming the system is no longer programming the microprocessor alone, but also programming all the other chips attached to it.* The learning time for every chip can be significant.

Naturally, this is only an apparent dilemma. If these chips were not available, the complexity of the interface to be realized, as well as of the corresponding programs, would be still greater. The new complexity that is introduced is the need to program more than just a processor,

and to learn the various features of the different chips in a system. However, it is hoped that the techniques and concepts presented in this book will make this a reasonably easy task.

THE NEXT STEP

You have now learned the basic techniques required to program simple applications on paper. That was the goal of this book. The next step is actual practice for which there is no substitute. It is impossible to learn programming completely on paper; experience is required. You should now be in a position to start writing your own programs. It is hoped that this journey will be a pleasant one.

APPENDIX A

HEXADECIMAL CONVERSION TABLE

| HEX | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 00 | 000 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|-------|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 0 |
| 1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 256 | 4096 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 512 | 8192 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 768 | 12288 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 1024 | 16384 |
| 5 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 1280 | 20480 |
| 6 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 1536 | 24576 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 1792 | 28672 |
| 8 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | 2048 | 32768 |
| 9 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 2304 | 36864 |
| A | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 2560 | 40960 |
| B | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 2816 | 45056 |
| C | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 3072 | 49152 |
| D | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 | 3328 | 53248 |
| E | 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 3584 | 57344 |
| F | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 | 3840 | 61440 |

| 5 | | 4 | | 3 | | 2 | | 1 | | 0 | |
|-----|------------|-----|---------|-----|--------|-----|-------|-----|-----|-----|-----|
| HEX | DEC | HEX | DEC | HEX | DEC | HEX | DEC | HEX | DEC | HEX | DEC |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1,048,576 | 1 | 65,536 | 1 | 4,096 | 1 | 256 | 1 | 16 | 1 | 1 |
| 2 | 2,097,152 | 2 | 131,072 | 2 | 8,192 | 2 | 512 | 2 | 32 | 2 | 2 |
| 3 | 3,145,728 | 3 | 196,608 | 3 | 12,288 | 3 | 768 | 3 | 48 | 3 | 3 |
| 4 | 4,194,304 | 4 | 262,144 | 4 | 16,384 | 4 | 1,024 | 4 | 64 | 4 | 4 |
| 5 | 5,242,880 | 5 | 327,680 | 5 | 20,480 | 5 | 1,280 | 5 | 80 | 5 | 5 |
| 6 | 6,291,456 | 6 | 393,216 | 6 | 24,576 | 6 | 1,536 | 6 | 96 | 6 | 6 |
| 7 | 7,340,032 | 7 | 458,752 | 7 | 28,672 | 7 | 1,792 | 7 | 112 | 7 | 7 |
| 8 | 8,388,608 | 8 | 524,288 | 8 | 32,768 | 8 | 2,048 | 8 | 128 | 8 | 8 |
| 9 | 9,437,184 | 9 | 589,824 | 9 | 36,864 | 9 | 2,304 | 9 | 144 | 9 | 9 |
| A | 10,485,760 | A | 655,360 | A | 40,960 | A | 2,560 | A | 160 | A | 10 |
| B | 11,534,336 | B | 720,896 | B | 45,056 | B | 2,816 | B | 176 | B | 11 |
| C | 12,582,912 | C | 786,432 | C | 49,152 | C | 3,072 | C | 192 | C | 12 |
| D | 13,631,488 | D | 851,968 | D | 53,248 | D | 3,328 | D | 208 | D | 13 |
| E | 14,680,064 | E | 917,504 | E | 57,344 | E | 3,584 | E | 224 | E | 14 |
| F | 15,728,640 | F | 983,040 | F | 61,440 | F | 3,840 | F | 240 | F | 15 |

APPENDIX B

ASCII CONVERSION TABLE

| HEX | MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|------|-----|-----|-------|-----|-----|-----|-----|-----|
| LSD | BITS | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 0000 | NUL | DLE | SPACE | 0 | @ | P | ~ | p |
| 1 | 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | 0100 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 5 | 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | 0111 | BEL | ETB | , | 7 | G | W | g | w |
| 8 | 1000 | BS | CAN | (| 8 | H | X | h | x |
| 9 | 1001 | HT | EM |) | 9 | I | Y | i | y |
| A | 1010 | LF | SUB | * | : | J | Z | j | z |
| B | 1011 | VT | ESC | + | ; | K | [| k | { |
| C | 1100 | FF | FS | . | < | L | \ | l | -- |
| D | 1101 | CR | GS | - | = | M |] | m | } |
| E | 1110 | SO | RS | . | > | N | ^ | n | ~ |
| F | 1111 | SI | US | / | ? | O | < | o | DEL |

THE ASCII SYMBOLS

NUL — Null
SOH — Start of Heading
STX — Start of Text
ETX — End of Text
EOT — End of Transmission
ENQ — Enquiry
ACK — Acknowledge
BEL — Bell
BS — Backspace
HT — Horizontal Tabulation
LF — Line Feed
VT — Vertical Tabulation
FF — Form Feed
CR — Carriage Return
SO — Shift Out
SI — Shift In

DLE — Data Link Escape
DC — Device Control
NAK — Negative Acknowledge
SYN — Synchronous Idle
ETB — End of Transmission Block
CAN — Cancel
EM — End of Medium
SUB — Substitute
ESC — Escape
FS — File Separator
GS — Group Separator
RS — Record Separator
US — Unit Separator
SP — Space (Blank)
DEL — Delete

APPENDIX C

RELATIVE BRANCH TABLES

FORWARD RELATIVE BRANCH TABLE

| LSD MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 5 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 6 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |

BACKWARD RELATIVE BRANCH TABLE

| LSD MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 8 | 128 | 127 | 126 | 125 | 124 | 123 | 122 | 121 | 120 | 119 | 118 | 117 | 116 | 115 | 114 | 113 |
| 9 | 112 | 111 | 110 | 109 | 108 | 107 | 106 | 105 | 104 | 103 | 102 | 101 | 100 | 99 | 98 | 97 |
| A | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 88 | 87 | 86 | 85 | 84 | 83 | 82 | 81 |
| B | 80 | 79 | 78 | 77 | 76 | 75 | 74 | 73 | 72 | 71 | 70 | 69 | 68 | 67 | 66 | 65 |
| C | 64 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 |
| D | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 |
| E | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 |
| F | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

APPENDIX D

DECIMAL TO BCD CONVERSION

| DECIMAL | BCD | DEC | BCD | DEC | BCD |
|---------|------|-----|----------|-----|-----------|
| 0 | 0000 | 10 | 00010000 | 90 | 10010000 |
| 1 | 0001 | 11 | 00010001 | 91 | 10010001 |
| 2 | 0010 | 12 | 00010010 | 92 | 10010010 |
| 3 | 0011 | 13 | 00010011 | 93 | 10010011 |
| 4 | 0100 | 14 | 00010100 | 94 | 10010100 |
| 5 | 0101 | 15 | 00010101 | 95 | 10010101 |
| 6 | 0110 | 16 | 00010110 | 96 | 10010110 |
| 7 | 0111 | 17 | 00010111 | 97 | 10010111* |
| 8 | 1000 | 18 | 00011000 | 98 | 10011000 |
| 9 | 1001 | 19 | 00011001 | 99 | 10011001 |

APPENDIX E

Z80 INSTRUCTION CODES

(The literal d is shown as 05 in the object code.)

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| 8E | ADC A,(HL) |
| DD8E05 | ADC A,(IX+d) |
| FD8E05 | ADC A,(IY+d) |
| 8F | ADC A,A |
| 88 | ADC A,B |
| 89 | ADC A,C |
| 8A | ADC A,D |
| 8B | ADC A,E |
| 8C | ADC A,H |
| 8D | ADC A,L |
| CE20 | ADC A,n |
| ED4A | ADC HL,BC |
| ED5A | ADC HL,DE |
| ED6A | ADC HL,HL |
| ED7A | ADC HL,SP |
| 86 | ADD A,(HL) |
| DD8605 | ADD A,(IX+d) |
| FD8605 | ADD A,(IY+d) |
| 87 | ADD A,A |
| 80 | ADD A,B |
| 81 | ADD A,C |
| 82 | ADD A,D |
| 83 | ADD A,E |
| 84 | ADD A,H |
| 85 | ADD A,L |
| C620 | ADD A,n |
| 09 | ADD HL,BC |
| 19 | ADD HL,DE |
| 29 | ADD HL,HL |
| 39 | ADD HL,SP |
| DD09 | ADD IX,BC |
| DD19 | ADD IX,DE |
| DD29 | ADD IX,IX |
| DD39 | ADD IX,SP |
| FD09 | ADD IY,BC |
| FD19 | ADD IY,DE |
| FD29 | ADD IY,IY |
| FD39 | ADD IY,SP |
| A6 | AND (HL) |
| DDA605 | AND (IX+d) |
| FDA605 | AND (IY+d) |
| A7 | AND A |
| A0 | AND B |
| A1 | AND C |
| A2 | AND D |
| A3 | AND E |
| A4 | AND H |
| A5 | AND L |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| E620 | AND n |
| CB46 | BIT 0,(HL) |
| DDCB0546 | BIT 0,(IX+d) |
| FDCB0546 | BIT 0,(IY+d) |
| CB47 | BIT 0,A |
| CB40 | BIT 0,B |
| CB41 | BIT 0,C |
| CB42 | BIT 0,D |
| CB43 | BIT 0,E |
| CB44 | BIT 0,H |
| CB45 | BIT 0,L |
| CB4E | BIT 1,(HL) |
| DDCB054E | BIT 1,(IX+d) |
| FDCB054E | BIT 1,(IY+d) |
| CB4F | BIT 1,A |
| CB48 | BIT 1,B |
| CB49 | BIT 1,C |
| CB4A | BIT 1,D |
| CB4B | BIT 1,E |
| CB4C | BIT 1,H |
| CB4D | BIT 1,L |
| CB56 | BIT 2,(HL) |
| DDCB0556 | BIT 2,(IX+d) |
| FDCB0556 | BIT 2,(IY+d) |
| CB57 | BIT 2,A |
| CB50 | BIT 2,B |
| CB51 | BIT 2,C |
| CB52 | BIT 2,D |
| CB53 | BIT 2,E |
| CB54 | BIT 2,H |
| CB55 | BIT 2,L |
| CB5E | BIT 3,(HL) |
| DDCB055E | BIT 3,(IX+d) |
| FDCB055E | BIT 3,(IY+d) |
| CB5F | BIT 3,A |
| CB58 | BIT 3,B |
| CB59 | BIT 3,C |
| CB5A | BIT 3,D |
| CB5B | BIT 3,E |
| CB5C | BIT 3,H |
| CB5D | BIT 3,L |
| CB66 | BIT 4,(HL) |
| DDCB0566 | BIT 4,(IX+d) |
| FDCB0566 | BIT 4,(IY+d) |
| CB67 | BIT 4,A |
| CB60 | BIT 4,B |
| CB61 | BIT 4,C |
| CB62 | BIT 4,D |

APPENDIX

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| CB63 | BIT 4.E |
| CB64 | BIT 4.H |
| CB65 | BIT 4.L |
| CB6E | BIT 5,(HL) |
| DDCB056E | BIT 5,(IX+d) |
| FDCB056E | BIT 5,(IY+d) |
| CB6F | BIT 5,A |
| CB68 | BIT 5,B |
| CB69 | BIT 5,C |
| C86A | BIT 5,D |
| CB6B | BIT 5,E |
| CB6C | BIT 5,H |
| CB6D | BIT 5,L |
| C876 | BIT 6,(HL) |
| DDCB0576 | BIT 6,(IX+d) |
| FDCB0576 | BIT 6,(IY+d) |
| CB77 | BIT 6,A |
| CB70 | BIT 6,B |
| CB71 | BIT 6,C |
| CB72 | BIT 6,D |
| CB73 | BIT 6,E |
| CB74 | BIT 6,H |
| CB75 | BIT 6,L |
| CB7E | BIT 7,(HL) |
| DDCB057E | BIT 7,(IX+d) |
| FDCB057E | BIT 7,(IY+d) |
| CB7F | BIT 7,A |
| CB78 | BIT 7,B |
| CB79 | BIT 7,C |
| CB7A | BIT 7,D |
| CB7B | BIT 7,E |
| CB7C | BIT 7,H |
| CB7D | BIT 7,L |
| DC8405 | CALL C,nn |
| FC8405 | CALL M,nn |
| D48405 | CALL NC,nn |
| C48405 | CALL NZ,nn |
| F48405 | CALL P,nn |
| EC8405 | CALL PE,nn |
| E48405 | CALL PO,nn |
| CC8405 | CALL Z,nn |
| CD8405 | CALL nn |
| 3F | CCF |
| BE | CP (HL) |
| DBBE05 | CP (IX+d) |
| FDBE05 | CP (IY+d) |
| BF | CP A |
| B8 | CP B |
| B9 | CP C |
| BA | CP D |
| BB | CP E |
| BC | CP H |
| BD | CP L |
| FE20 | CP n |
| EDA9 | CPD |
| EDB9 | CPDR |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| EDB1 | CPIR |
| EDA1 | CPI |
| 2F | CPL |
| 27 | DAA |
| 35 | DEC (HL) |
| DD3505 | DEC (IX+d) |
| FD3505 | DEC (IY+d) |
| 3D | DEC A |
| 05 | DEC B |
| 08 | DEC BC |
| 0D | DEC C |
| 15 | DEC D |
| 1B | DEC DE |
| 1D | DEC E |
| 25 | DEC H |
| 2B | DEC HL |
| DD28 | DEC IX |
| FD28 | DEC IY |
| 2D | DEC L |
| 3B | DEC SP |
| F3 | DI |
| 102E | DJNZ e |
| FB | EI |
| E3 | EX (SP),HL |
| DDE3 | EX (SP),IX |
| FDE3 | EX (SP),IY |
| 08 | EX AF,AF' |
| EB | EX DE,HL |
| D9 | EXX |
| 76 | HALT |
| ED46 | IM 0 |
| ED56 | IM 1 |
| ED5E | IM 2 |
| ED78 | IN A,(C) |
| ED40 | IN B,(C) |
| ED48 | IN C,(C) |
| ED50 | IN D,(C) |
| ED58 | IN E,(C) |
| ED60 | IN H,(C) |
| ED68 | IN L,(C) |
| 34 | INC (HL) |
| DD3405 | INC (IX+d) |
| FD3405 | INC (IY+d) |
| 3C | INC A |
| 04 | INC B |
| 03 | INC BC |
| 0C | INC C |
| 14 | INC D |
| 13 | INC DE |
| 1C | INC E |
| 24 | INC H |
| 23 | INC HL |
| DD23 | INC IX |
| FD23 | INC IY |
| 2C | INC L |
| 33 | INC SP |
| DB20 | IN A,(n) |

PROGRAMMING THE Z80

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| EDAA | IND |
| EDBA | INDR |
| EDA2 | INI |
| EDB2 | INIR |
| C38405 | JP nn |
| E9 | JP (HL) |
| DDE9 | JP (IX) |
| FDE9 | JP (IY) |
| DA8405 | JP C,nn |
| FA8405 | JP M,nn |
| D28405 | JP NC,nn |
| C28405 | JP NZ,nn |
| F28405 | JP P,nn |
| EA8405 | JP PE,nn |
| E28405 | JP PO,nn |
| CA8405 | JP Z,nn |
| 382E | JR C,e |
| 302E | JR NC,e |
| 202E | JR NZ,e |
| 282E | JR Z,e |
| 182E | JR e^l,l |
| 02 | LD (BC),A |
| 12 | LD (DE),A |
| 77 | LD (HL),A |
| 70 | LD (HL),B |
| 71 | LD (HL),C |
| 72 | LD (HL),D |
| 73 | LD (HL),E |
| 74 | LD (HL),H |
| 75 | LD (HL),L |
| 3620 | LD (HL),n |
| DD7705 | LD (IX+d),A |
| DD7005 | LD (IX+d),B |
| DD7105 | LD (IX+d),C |
| DD7205 | LD (IX+d),D |
| DD7305 | LD (IX+d),E |
| DD7405 | LD (IX+d),H |
| DD7505 | LD (IX+d),L |
| DD360520 | LD (IX+d),n |
| FD7705 | LD (IY+d),A |
| FD7005 | LD (IY+d),B |
| FD7105 | LD (IY+d),C |
| FD7205 | LD (IY+d),D |
| FD7305 | LD (IY+d),E |
| FD7405 | LD (IY+d),H |
| FD7505 | LD (IY+d),L |
| FD360520 | LD (IY+d),n |
| 328405 | LD (nn),A |
| ED438405 | LD (nn),BC |
| ED538405 | LD (nn),DE |
| 228405 | LD (nn),HL |
| DD228405 | LD (nn),IX |
| FD228405 | LD (nn),IY |
| ED738405 | LD (nn),SP |
| 0A | LD A,(BC) |
| 1A | LD A,(DE) |
| 7E | LD A,(HL) |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| DD7E05 | LD A,(IX+d) |
| FD7E05 | LD A,(IY+d) |
| 3A8405 | LD A,(nn) |
| 7F | LD A,A |
| 78 | LD A,B |
| 79 | LD A,C |
| 7A | LD A,D |
| 7B | LD A,E |
| 7C | LD A,H |
| ED57 | LD A,I |
| 7D | LD A,L |
| 3E20 | LD A,n |
| ED5F | LD A,R |
| 46 | LD B,(HL) |
| DD4605 | LD B,(IX+d) |
| FD4605 | LD B,(IY+d) |
| 47 | LD B,A |
| 40 | LD B,B |
| 41 | LD B,C |
| 42 | LD B,D |
| 43 | LD B,E |
| 44 | LD B,H |
| 45 | LD B,L |
| 0620 | LD B,n |
| ED4B8405 | LD BC,(nn) |
| 018405 | LD BC,nn |
| 4E | LD C,(HL) |
| DD4E05 | LD C,(IX+d) |
| FD4E05 | LD C,(IY+d) |
| 4F | LD C,A |
| 48 | LD C,B |
| 49 | LD C,C |
| 4A | LD C,D |
| 4B | LD C,E |
| 4C | LD C,H |
| 4D | LD C,L |
| 0E20 | LD C,n |
| 56 | LD D,(HL) |
| DD5605 | LD D,(IX+d) |
| FD5605 | LD D,(IY+d) |
| 57 | LD D,A |
| 50 | LD D,B |
| 51 | LD D,C |
| 52 | LD D,D |
| 53 | LD D,E |
| 54 | LD D,H |
| 55 | LD D,L |
| 1620 | LD D,n |
| ED5B8405 | LD DE,(nn) |
| 118405 | LD DE,nn |
| 5E | LD E,(HL) |
| DD5E05 | LD E,(IX+d) |
| FD5E05 | LD E,(IY+d) |
| 5F | LD E,A |
| 58 | LD E,B |
| 59 | LD E,C |
| 5A | LD E,D |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| 5B | LD E,E |
| 5C | LD E,H |
| 5D | LD E,L |
| 1E20 | LD E,n |
| 66 | LD H,(HL) |
| DD6605 | LD H,(IX+d) |
| FD6605 | LD H,(IY+d) |
| 67 | LD H,A |
| 60 | LD H,B |
| 61 | LD H,C |
| 62 | LD H,D |
| 63 | LD H,E |
| 64 | LD H,H |
| 65 | LD H,L |
| 2620 | LD H,n |
| 2A8405 | LD HL,(nn) |
| 218405 | LD HL,nn |
| ED47 | LD I,A |
| DD2A8405 | LD IX,(nn) |
| DD218405 | LD IX,nn |
| FD2A8405 | LD IY,(nn) |
| FD218405 | LD IY,nn |
| 6E | LD L,(HL) |
| DD6E05 | LD L,(IX+d) |
| FD6E05 | LD L,(IY+d) |
| 6F | LD L,A |
| 68 | LD L,B |
| 69 | LD L,C |
| 6A | LD L,D |
| 6B | LD L,E |
| 6C | LD L,H |
| 6D | LD L,L |
| 2E20 | LD L,n |
| ED4F | LD R,A |
| ED7B8405 | LD SP,(nn) |
| F9 | LD SP,HL |
| DDF9 | LD SP,IX |
| FDF9 | LD SP,IY |
| 318405 | LD SP,nn |
| EDA8 | LDD |
| EDB8 | LDDR |
| EDA0 | LDI |
| EDB0 | LDIR |
| ED44 | NEG |
| 00 | NOP |
| B6 | OR (HL) |
| DDB605 | OR (IX+d) |
| FDB605 | OR (IY+d) |
| B7 | OR A |
| B0 | OR B |
| B1 | OR C |
| B2 | OR D |
| B3 | OR E |
| B4 | OR H |
| B5 | OR L |
| F620 | OR n |
| ED8B | OTDR |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| EDB3 | OTIR |
| ED79 | OUT (C),A |
| ED41 | OUT (C),B |
| ED49 | OUT (C),C |
| ED51 | OUT (C),D |
| ED59 | OUT (C),E |
| ED61 | OUT (C),H |
| ED69 | OUT (C),L |
| D320 | OUT (n),A |
| EDAB | OUTD |
| EDA3 | OUTI |
| F1 | POP AF |
| C1 | POP BC |
| D1 | POP DE |
| E1 | POP HL |
| DDE1 | POP IX |
| FDE1 | POP IY |
| F5 | PUSH AF |
| C5 | PUSH BC |
| D5 | PUSH DE |
| E5 | PUSH HL |
| DDE5 | PUSH IX |
| FDE5 | PUSH IY |
| CB86 | RES 0,(HL) |
| DDCB0586 | RES 0,(IX+d) |
| FDCB0586 | RES 0,(IY+d) |
| CB87 | RES 0,A |
| CB80 | RES 0,B |
| CB81 | RES 0,C |
| CB82 | RES 0,D |
| CB83 | RES 0,E |
| CB84 | RES 0,H |
| CB85 | RES 0,L |
| CB8E | RES 1,(HL) |
| DDCB058E | RES 1,(IX+d) |
| FDCB058E | RES 1,(IY+d) |
| CB8F | RES 1,A |
| CB88 | RES 1,B |
| CB89 | RES 1,C |
| CB8A | RES 1,D |
| CB8B | RES 1,E |
| CB8C | RES 1,H |
| CB8D | RES 1,L |
| CB96 | RES 2,(HL) |
| DDCB0596 | RES 2,(IX+d) |
| FDCB0596 | RES 2,(IY+d) |
| CB97 | RES 2,A |
| CB90 | RES 2,B |
| CB91 | RES 2,C |
| CB92 | RES 2,D |
| CB93 | RES 2,E |
| CB94 | RES 2,H |
| CB95 | RES 2,L |
| CB9E | RES 3,(HL) |
| DDCB059E | RES 3,(IX+d) |
| FDCB059E | RES 3,(IY+d) |

PROGRAMMING THE Z80

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| CB9F | RES 3,A |
| CB98 | RES 3,B |
| CB99 | RES 3,C |
| CB9A | RES 3,D |
| CB9B | RES 3,E |
| CB9C | RES 3,H |
| CB9D | RES 3,L |
| CBA6 | RES 4,(HL) |
| DDCB05A6 | RES 4,(IX+d) |
| FDCB05A6 | RES 4,(IY+d) |
| CBA7 | RES 4,A |
| CBA0 | RES 4,B |
| CBA1 | RES 4,C |
| CBA2 | RES 4,D |
| CBA3 | RES 4,E |
| CBA4 | RES 4,H |
| CBA5 | RES 4,L |
| CBAE | RES 5,(HL) |
| DDCB05AE | RES 5,(IX+d) |
| FDCB05AE | RES 5,(IY+d) |
| CBAF | RES 5,A |
| CBA8 | RES 5,B |
| CBA9 | RES 5,C |
| CBAA | RES 5,D |
| CBAB | RES 5,E |
| CBAC | RES 5,H |
| CBAD | RES 5,L |
| CBB6 | RES 6,(HL) |
| DDCB05B6 | RES 6,(IX+d) |
| FDCB05B6 | RES 6,(IY+d) |
| CBB7 | RES 6,A |
| CBB0 | RES 6,B |
| CBB1 | RES 6,C |
| CBB2 | RES 6,D |
| CBB3 | RES 6,E |
| CBB4 | RES 6,H |
| CBB5 | RES 6,L |
| CBBE | RES 7,(HL) |
| DDCB05BE | RES 7,(IX+d) |
| FDCB05BE | RES 7,(IY+d) |
| CBBF | RES 7,A |
| CBB8 | RES 7,B |
| CBB9 | RES 7,C |
| CBBA | RES 7,D |
| CBBB | RES 7,E |
| CBBC | RES 7,H |
| CBBD | RES 7,L |
| C9 | RET |
| D8 | RET C |
| F8 | RET M |
| D0 | RET NC |
| C0 | RET NZ |
| F0 | RET P |
| E8 | RET PE |
| E0 | RET PO |
| C8 | RET Z |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| ED4D | RETI |
| ED45 | RETN |
| CB16 | RL (HL) |
| DDCB0516 | RL (IX+d) |
| FDCB0516 | RL (IY+d) |
| CB17 | RL A |
| CB10 | RL B |
| CB11 | RL C |
| CB12 | RL D |
| CB13 | RL E |
| CB14 | RL H |
| CB15 | RL L |
| 17 | RLA |
| CB06 | RLC (HL) |
| DDCB0506 | RLC (IX+d) |
| FDCB0506 | RLC (IY+d) |
| CB07 | RLC A |
| CB00 | RLC B |
| CB01 | RLC C |
| CB02 | RLC D |
| CB03 | RLC E |
| CB04 | RLC H |
| CB05 | RLC L |
| 07 | RLCA |
| ED6F | RLD |
| CB1E | RR (HL) |
| DDCB051E | RR (IX+d) |
| FDCB051E | RR (IY+d) |
| CB1F | RR A |
| CB18 | RR B |
| CB19 | RR C |
| CB1A | RR D |
| CB1B | RR E |
| CB1C | RR H |
| CB1D | RR L |
| 1F | RRA |
| CB0E | RR C (HL) |
| DDCB050E | RR C (IX+d) |
| FDCB050E | RR C (IY+d) |
| CB0F | RR C A |
| CB08 | RR C B |
| CB09 | RR C C |
| CB0A | RR C D |
| CB0B | RR C E |
| CB0C | RR C H |
| CB0D | RR C L |
| OF | RRCA |
| ED67 | RRD |
| C7 | RST 00H |
| CF | RST 08H |
| D7 | RST 10H |
| DF | RST 18H |
| E7 | RST 20H |
| EF | RST 28H |
| F7 | RST 30H |
| FF | RST 38H |
| DE20 | SBC A,n |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| 9E | SBC A,(HL) |
| DD9E05 | SBC A,(IX+d) |
| FD9E05 | SBC A,(IY+d) |
| 9F | SBC A,A |
| 98 | SBC A,B |
| 99 | SBC A,C |
| 9A | SBC A,D |
| 9B | SBC A,E |
| 9C | SBC A,H |
| 9D | SBC A,L |
| ED42 | SBC HL,BC |
| ED52 | SBC HL,DE |
| ED62 | SBC HL,HL |
| ED72 | SBC HL,SP |
| 37 | SCF |
| CBC6 | SET 0,(HL) |
| DDCB05C6 | SET 0,(IX+d) |
| FDCB05C6 | SET 0,(IY+d) |
| CBC7 | SET 0,A |
| CBC0 | SET 0,B |
| CBC1 | SET 0,C |
| CBC2 | SET 0,D |
| CBC3 | SET 0,E |
| CBC4 | SET 0,H |
| CBC5 | SET 0,L |
| C BCE | SET 1,(HL) |
| DDCB05CE | SET 1,(IX+d) |
| FDCB05CE | SET 1,(IY+d) |
| CBCF | SET 1,A |
| CBC8 | SET 1,B |
| CBC9 | SET 1,C |
| CBCA | SET 1,D |
| CBCB | SET 1,E |
| CBCC | SET 1,H |
| CBCD | SET 1,L |
| CBD6 | SET 2,(HL) |
| DDCB05D6 | SET 2,(IX+d) |
| FDCB05D6 | SET 2,(IY+d) |
| CBD7 | SET 2,A |
| CBD0 | SET 2,B |
| CBD1 | SET 2,C |
| CBD2 | SET 2,D |
| CBD3 | SET 2,E |
| CBD4 | SET 2,H |
| CBD5 | SET 2,L |
| CBD8 | SET 3,B |
| CBDE | SET 3,(HL) |
| DDCB05DE | SET 3,(IX+d) |
| FDCB05DE | SET 3,(IY+d) |
| CBDF | SET 3,A |
| CBD9 | SET 3,C |
| CBDA | SET 3,D |
| CBDB | SET 3,E |
| CBDC | SET 3,H |
| CBDD | SET 3,L |
| CBE6 | SET 4,(HL) |

| OBJ CODE | SOURCE STATEMENT |
|-------------|---------------------|
| DDCB05E6 | SET 4,(IX+d) |
| FDCB05E6 | SET 4,(IY+d) |
| CBE7 | SET 4,A |
| CBE0 | SET 4,B |
| CBE1 | SET 4,C |
| CBE2 | SET 4,D |
| CBE3 | SET 4,E |
| CBE4 | SET 4,H |
| CBE5 | SET 4,L |
| CBE6 | SET 5,(HL) |
| DDCB05EE | SET 5,(IX+d) |
| FDCB05EE | SET 5,(IY+d) |
| CBEF | SET 5,A |
| CBE8 | SET 5,B |
| CBE9 | SET 5,C |
| CBEA | SET 5,D |
| CBEB | SET 5,E |
| CBEC | SET 5,H |
| CBED | SET 5,L |
| CBF6 | SET 6,(HL) |
| DDCB05F6 | SET 6,(IX+d) |
| FDCB05F6 | SET 6,(IY+d) |
| CBF7 | SET 6,A |
| CBF0 | SET 6,B |
| CBF1 | SET 6,C |
| CBF2 | SET 6,D |
| CBF3 | SET 6,E |
| CBF4 | SET 6,H |
| CBF5 | SET 6,L |
| CBFE | SET 7,(HL) |
| DDCB05FE | SET 7,(IX+d) |
| FDCB05FE | SET 7,(IY+d) |
| CBFF | SET 7,A |
| CBF8 | SET 7,B |
| CBF9 | SET 7,C |
| CBFA | SET 7,D |
| CBFB | SET 7,E |
| CBFC | SET 7,H |
| CBFD | SET 7,L |
| CB26 | SLA (HL) |
| DDCB0526 | SLA (IX+d) |
| FDCB0526 | SLA (IY+d) |
| CB27 | SLA A |
| CB20 | SLA B |
| CB21 | SLA C |
| CB22 | SLA D |
| CB23 | SLA E |
| CB24 | SLA H |
| CB25 | SLA L |
| CB2E | SRA (HL) |
| DDCB052E | SRA (IX+d) |
| FDCB052E | SRA (IY+d) |
| CB2F | SRA A |
| CB28 | SRA B |
| CB29 | SRA C |
| CB2A | SRA D |

PROGRAMMING THE Z80

| OBJ | SOURCE | |
|----------|-----------|--------|
| CODE | STATEMENT | |
| CB2B | SRA | E |
| CB2C | SRA | H |
| CB2D | SRA | L |
| CB3E | SRL | (HL) |
| DDCB053E | SRL | (IX+d) |
| FDCB053E | SRL | (IY+d) |
| CB3F | SRL | A |
| CB38 | SRL | B |
| CB39 | SRL | C |
| CB3A | SRL | D |
| CB3B | SRL | E |
| CB3C | SRL | H |
| CB3D | SRL | L |
| 96 | SUB | (HL) |
| DD9605 | SUB | (IX+d) |
| FD9605 | SUB | (IY+d) |
| 97 | SUB | A |
| 90 | SUB | B |
| 91 | SUB | C |
| 92 | SUB | D |
| 93 | SUB | E |
| 94 | SUB | H |
| 95 | SUB | L |
| D620 | SUB | n |
| AE | XOR | (HL) |
| DDAE05 | XOR | (IX+d) |
| FDAE05 | XOR | (IY+d) |
| AF | XOR | A |
| A8 | XOR | B |
| A9 | XOR | C |
| AA | XOR | D |
| AB | XOR | E |
| AC | XOR | H |
| AD | XOR | L |
| EE20 | XOR | n |

(Courtesy of Zilog Inc.)

APPENDIX F

Z80 to 8080 EQUIVALENCE

| Z80 | 8080 | Z80 | 8080 | Z80 | 8080 |
|-------------|---------------|-------------|-------------------|-------------|-------------|
| ADC A, (HL) | ADC M | EX (SP), HL | XTHL | OR n | ORI [B2] |
| ADC A, n | ACI [B2] | HALT | HLT | OR r | ORA r |
| ADC A, r | ADC r | IN A, (n) | IN [B2] | OR (HL) | ORA M |
| ADD A, (HL) | ADD M | INC BC | INX B | OUT (n), A | OUT [B2] |
| ADD A, n | ADI [B2] | INC DE | INX D | POP AF | POP PSW |
| ADD A, r | ADD r | INC HL | INX H | POP BC | POP B |
| ADD HL, BC | DAD B | INC r | INR r | POP DE | POP D |
| ADD HL, DE | DAD D | INC SP | INX SP | POP HL | POP H |
| ADD HL, HL | DAD H | INC (HL) | INR M | PUSH AF | PUSH PSW |
| ADD HL, SP | DAD SP | JP C, nn | JC [B2] [B3] | PUSH BC | PUSH B |
| AND n | ANI [B2] | JP M, nn | JM [B2][B3] | PUSH DE | PUSH D |
| AND r | ANA r | JP NC, nn | JNC [B2] [B3] | PUSH HL | PUSH H |
| AND (HL) | ANA M | JP nn | JMP [B2] [B3] | RET | RET |
| CALL C, nn | CC [B2] [B3] | JP NZ, nn | JNZ [B2] [B3] | RET C | RC |
| CALL M, nn | CM [B2] [B3] | JP P, nn | JP [B2] [B3] | RET M | RM |
| CALL NC, nn | CNC [B2] [B3] | JP PE, nn | JPE [B2][B3] | RET NC | RNC |
| CALL nn | CALL | JP PO, nn | JPO [B2][B3] | RET NZ | RNZ |
| CALL NZ, nn | CNZ [B2] [B3] | JP Z, nn | JZ [B2] [B3] | RET P | RP |
| CALL P, nn | CP [B2] [B3] | JP (HL) | PCHL | RET PE | RPE |
| CALL PE, nn | CPE [B2] [B3] | LD A, (DE) | LDAX | RET PO | RPO |
| CALL PO, nn | CPO [B2] [B3] | LDA, (nn) | LDA [B2] [B3] | RET Z | RZ |
| CALL Z, nn | CZ [B2] [B3] | LD DE, nn | LXID, [B2] [B3] | RLA | RAL |
| CCF | CMC | LD SP, nn | LXI SP, [B2] [B3] | RLCA | RLC |
| CP r | CMP r | LD (BC), A | STAX B | RRA | RAR |
| CP (HL) | CMP M | LD (DE), A | STAX D | RRCA | RRC |
| CPL | CMA | LD (HL), r | MOV M, r | RST P | RST P |
| CP n | CPI [B2] | LD (nn), A | STA [B2] [B3] | SBC A, (HL) | SBB M |
| DAA | DAA | LD (nn), HL | SHLD [B2] [B3] | SBC A, n | SBI [B2] |
| DEC BC | DCX B | LD A, (BC) | LDAX B | SBC A, r | SBB r |
| DEC DE | DCX D | LD BC, nn | LXIB, [B2] [B3] | SCF | STC |
| DEC HL | DCX H | LD HL, (nn) | LHLD [B2] [B3] | SUB n | SUI [B2] |
| DEC r | DCR r | LD HL, nn | LXI H [B2] [B3] | SUB r | SUB r |
| DEC SP | DCX SP | LD r, (HC) | MOV 1, M | SUB (HL) | SUB M |
| DEC (HL) | DCR M | LD r, n | MVI r, [B2] | XOR n | XRI [B2] |
| DI | DI | LD r, r' | MOV r1, r2 | XOR r | XRA r |
| EI | EI | LD SP, HL | SPHL | XOR (HL) | XRA M |
| EX DE, HL | XCHG | NOP | NOP | | |

APPENDIX G

8080 to Z80 EQUIVALENCE

| 8080 | Z80 | 8080 | Z80 | 8080 | Z80 |
|---------------|-------------|------------------|-------------|----------------|-------------|
| ACI [B2] | ADC A, n | IN [B2] | IN A, (n) | POP H | POP HL |
| ADC M | ADC A, (HL) | INR M | INC (HL) | POP PSW | POP AF |
| ADC r | ADC A, r | INR r | INC r | PUSH B | PUSH BC |
| ADD M | ADD A, (HL) | INX B | INC BC | PUSH D | PUSH DE |
| ADD r | ADD A, r | INX D | INC DE | PUSH H | PUSH HL |
| ADI [B2] | ADD A, n | INX H | INC HL | PUSH PSW | PUSH AF |
| ANA M | AND (HL) | INX SP | INC SP | RAL | RLA |
| ANA r | AND r | JC [B2] [B3] | JP C, nn | RAR | RRA |
| ANI [B2] | AND n | JM [B2] [B3] | JP M, nn | RC | RET C |
| CALL | CALL nn | JMP [B2] [B3] | JP nn | RET | RET |
| CC [B2] [B3] | CALL C, nn | JNC [B2] [B3] | JP NC, nn | RLC | RLCA |
| CM [B2] [B3] | CALL M, nn | JNZ [B2] [B3] | JP NZ, nn | RM | RET M |
| CMA | CPL | JP [B2] [B3] | JP P, nn | RNC | RET NC |
| CMC | CCF | JPE [B2] [B3] | JP PE, nn | RNZ | RET NZ |
| CMP M | CP (HL) | JPO [B2] [B3] | JP PO, nn | RP | RET P |
| CMP r | CP r | JZ [B2] [B3] | JP Z, nn | RPE | RET PE |
| CNC [B2] [B3] | CALL NC, nn | LDA [B2] [B3] | LD A, (nn) | RPO | RET PO |
| CNZ [B2] [B3] | CALL NZ, nn | LDXA B | LD A, (BC) | RRC | RRCA |
| CP [B2] [B3] | CALL P, nn | LDXD A | LD A, (DE) | RST | RET P |
| CPE [B2] [B3] | CALL PE, nn | LH LD [B2] [B3] | LD HL, (nn) | RZ | RET Z |
| CPI [B2] | CP n | LXI B [B2] [B3] | LD BC, nn | SBB M | SBC A, (HL) |
| CPO [B2] [B3] | CALL PO, nn | L DID [B2] [B3] | LD DE, nn | SBB r | SBC A, r |
| CZ [B2] [B3] | CALL Z, nn | LXI H [B2] [B3] | LD HL, nn | SBI [B2] | SBC A, n |
| DAA | DAA | LXI SP [B2] [B3] | LD SP, nn | SHLD [B2] [B3] | LD (nn), HL |
| DAD B | ADD HL, BC | MOV M, r | LD (HL), r | SPHL | LD SP, HL |
| DAD D | ADD HL, DE | MOV r, M | LD r, (HL) | STA [B2] [B3] | LD (nn), A |
| DAD H | ADD HL, HL | MOV r1, r2 | LD r, r' | STAX B | LD (BC), A |
| DAD SP | ADD HL, SP | MVI M, n | LD (HL), n | STAX D | LD (DE), A |
| DCR M | DEC (HL) | MVI r [B2] | LD r, n | STC | SCF |
| DCR r | DEC r | NOP | NOP | SUB M | SUB (HL) |
| DCX B | DEC BC | ORA M. | OR (HL) | SUB r | SUB r |
| DCX D | DEC DE | ORA r | OR r | SUI [B2] | SUB n |
| DCX H | DEC HL | ORI [B2] | OR n | XCHG | EX DE, HL |
| DCX SP | DEC SP | OUT [B2] | OUT (n), A | XRA M | XOR (HL) |
| DI | DI | PCHL | JP (HL) | XRA r | XOR r |
| EI | EI | POP B | POP BC | XRI [B2] | XOR n |
| HALT | HLT | POP D | POP DE | XTHL | EX (SP), HL |

INDEX

A

| | |
|-------------------------------|--------------------|
| absolute addressing | 108, 439, 446 |
| ACT | 61 |
| accumulator | 439 |
| ADC | 101 |
| ADC, A, s | 190 |
| ADC HL, ss | 192 |
| ADD | 101 |
| ADD A, (HL) | 84, 194 |
| ADD A, (IX + d) | 196 |
| ADD A, (IY + d) | 198 |
| ADD A, n | 67, 200 |
| ADD A, r | 67, 75, 76, 201 |
| ADD HL, ss | 203 |
| ADD IX, rr | 205 |
| ADD IY, rr | 207 |
| addition | 58, 95, 100, 105 |
| address bus | 47 |
| address registers | 51 |
| addressing | 438, 442 |
| addressing modes | 438, 440, 444, 445 |
| addressing techniques | 438 |
| algorithm | 15, 16, 114, 539 |
| alphabetic list | 558, 565, 569, 570 |
| alphanumeric data | 39 |
| ALU | 46, 77, 85 |
| AND | 166, 167 |
| AND s | 209 |
| application examples | 520 |
| arithmetic-logical unit | 46, 61 |
| arithmetic programs | 94 |
| arithmetic shift | 119 |
| ASCII | 39, 524, 525 |
| ASCII conversion table | 40 |
| assembler | 96, 582, 590 |
| assembler directives | 596, 598 |
| assembler fields | 590 |
| assembly-language | 67, 580, 592 |
| assigning a value | 593 |
| asynchronous | 471, 496, 518 |
| automated Z80 instructions | 142, 453, 455 |

B

| | |
|--------------------------------|---|
| B | 62 |
| banks of registers | 62 |
| BASIC | 24 |
| basic architecture | 46 |
| basic concepts | 15 |
| basic programming choices | 579 |
| basic programming techniques | 94 |
| BCD | 35, 37, 525 |
| BCD addition | 107, 110 |
| BCD arithmetic | 107 |
| BCD block transfers | 530 |
| BCD flags | 112 |
| BCD representation | 35 |
| BCD subtraction | 110 |
| BCD table | 35 |
| benchmark | 470 |
| binary | 20, 21, 22, 41, 45 |
| binary code | 19 |
| binary digit | 18 |
| binary division | 133 |
| binary logic | 18 |
| binary representation | 41 |
| binary search | 546, 558, 559, 560, 561, 566, 567, 568 |
| BIT b, (HL) | 211 |
| BIT b, (IX + d) | 213 |
| BIT b, (IY + d) | 215 |
| BIT b, r | 217 |
| bit | 18, 20, 41 |
| bit addressing | 448 |
| bit manipulation | 172, 173 |
| bit serial transfer | 471, 472 |
| block | 540, 542, 544 |
| block transfer | 450, 451, 453, 458, 530 |
| block transfer instructions | 163, 450, 452 |
| bootstrap | 48 |
| bracket testing | 523 |
| branch instruction | 441 |
| branching point | 115 |
| break character | 467 |

PROGRAMMING THE Z80

| | | | |
|-------------------------|-------------------------|------------------------------|---------------|
| breakpoint | 584, 586 | CPI | 231 |
| bubble-sort | 533, 534, 535, 536, 537 | CPIR | 233 |
| buffer register | 59, 61 | CPL | 165, 235 |
| buffered | 49 | CPU | 46, 187 |
| buffers | 61 | critical race | 60 |
| bus request | 497 | CRT display | 44, 587 |
| BUSRQ | 92, 497 | crystal | 47 |
| byte | 18, 19, 41, 444 | CU | 46 |
| C | | | |
| C | 28, 30, 31, 62, 73 | D | 62, 74 |
| CALL | 145, 156, 446, 500 | DAA | 109, 236 |
| CALL cc, pq | 219 | data buffer | 511 |
| CALL pq | 222 | data bus | 47 |
| CCF | 224 | data counters | 51 |
| CALL SUB | 143, 144, 145 | data direction register | 512 |
| carry | 22, 23, 26, 28, 30, 174 | data processing | 155 |
| central-processing unit | 46 | data processing instructions | 164 |
| checksum computation | 528 | data ready | 469 |
| circular list | 544, 545 | data representation | 548 |
| classes of instructions | 154 | data structures | 539 |
| clearing memory | 520 | data transfers | 154, 158, 160 |
| clock | 47 | debugger | 583 |
| clock cycles | 69 | debugging | 18 |
| clock-synchronous logic | 86 | decimal | 20, 21, 22 |
| code conversion | 525 | DEC m | 238 |
| coding | 16 | DEC rr | 240 |
| combination chips | 48 | DEC IX | 242 |
| commands | 16 | DEC IY | 243 |
| comment field | 590 | decode | 71, 86 |
| compare | 531 | decoding | 56 |
| compiler | 545, 581, 582 | decoding logic | 49 |
| COND | 600 | decrement | 164, 442 |
| conclusion | 602 | DEFB | 596 |
| conditional assembly | 600 | DEFL | 596 |
| conditional instruction | 50 | DEFM | 597 |
| constants | 439, 445, 594 | DEFS | 597 |
| control box | 49 | DEFW | 596 |
| control bus | 47 | delay generation | 463 |
| control instructions | 157, 185 | delay loop | 464, 483 |
| control registers | 512, 513, 515 | deleting | 553, 565, 574 |
| control signals | 91 | design examples | 548 |
| control unit | 46 | destination register | 67 |
| count the zeroes | 529 | development systems | 587 |
| counter | 463, 465 | DFB | 596 |
| CP | 166 | DI | 244 |
| CP s | 225 | direct addressing | 439, 441 |
| CPD | 227 | direct binary | 19 |
| CPDR | 229 | direction register | 515 |

INDEX

| | | | |
|---|--------------------|-------------------------------|---|
| directives | 146, 571, 580, 594 | F | |
| directories | 541, 545 | F | 61 |
| disk operating system | 541, 582 | fetch | 55, 70, 84 |
| displacement | 63 | fetch-execute overlap | 78 |
| displacement field | 442 | FIFO | 543 |
| DJNZ e | 245 | file directory | 541 |
| DMA | 491, 498 | flags | 31, 50, 51, 179, 180 |
| documenting | 97 | flags register | 61 |
| DOS | 582 | flip-flops | 51 |
| doubly-linked lists | 545, 546 | floating point representation | 37, 38 |
| double-precision format | 34 | flowcharting | 16, 17, 114, 450, 464, 469, 494, 559 |
| drivers | 49 | front panel | 45, 589 |
| E | | | |
| E | 62 | G | |
| EBCDIC | 39 | general purpose registers | 51 |
| echo | 486 | getting characters in | 522 |
| editor | 583 | H | |
| EI | 247 | H | 62, 176 |
| 8-bit addition | 95 | half-carry flag (H) | 176 |
| 8-bit division | 134, 137 | HALT | 92, 185, 257 |
| element deletion | 564 | handshaking | 477, 478, 511 |
| element insertion | 550, 563 | hardware | 93 |
| emulator | 583 | hardware delays | 465 |
| END | 597 | hardware organization | 46 |
| ENDC | 600 | hardware resources | 587, 589 |
| ENDM | 597 | HEX | 525 |
| EPROM's | 585 | hexadecimal | 41, 42, 481 |
| EQU | 596 | hexadecimal coding | 43, 579 |
| error | 586 | high byte | 103 |
| error messages | 592 | high level language | 581 |
| EX AF, AF ^l | 162 | I | |
| exchange instructions | 162 | I | 63 |
| Exclusive ORing | 31 | IFF1 | 499 |
| EX DE, HL | 249 | IFF2 | 499 |
| executable statements | 16 | illegal code | 107 |
| execute | 71 | IM 0 | 258 |
| execution | 56, 69, 599 | IM 1 | 259 |
| execution cycle | 55 | IM 2 | 260 |
| exponent | 37, 38 | immediate addressing | 108, 159, 439, 445 |
| EX (SP), HL | 250 | immediate operation | 69 |
| EX (SP), IX | 252 | implicit addressing | 438, 445 |
| EX (SP), IY | 254 | implied addressing | 438 |
| extended addressing | 160, 441, 446 | improved multiplication | 126, 128, 129 |
| external representation of information | 41, 44 | IN r, (C) | 261 |
| EXX | 256 | IN A, (N) | 263 |
| | | in-circuit emulator | 585 |
| | | INC (HL) | 267 |

PROGRAMMING THE Z80

| | | | |
|------------------------------------|---|--------------------|-------------------|
| INC r | 264 | interrupt table | 504 |
| increment | 164, 442 | interrupt vector | 498 |
| incrementer | 57 | interrupts | 495 |
| INC rr | 265 | I/O control | 92 |
| INC (IX + d) | 268 | IORQ | 92, 500 |
| INC (IY + d) | 270 | IR | 55 |
| INC IX | 272 | IX | 53, 63 |
| INC IY | 273 | IY | 63 |
| IND | 274 | | |
| index register | 53, 63, 441, 442 | J | |
| indexed addressing | 160, 441, 447, 540 | JP cc, pq | 282 |
| indexing | 63 | JP nn | 89 |
| indirect addressing | 443, 444, 448, 540 | JP pq | 284 |
| indirect indexed addressing | 443 | JP (HL) | 285 |
| indirect memory access | 499 | JP (IX) | 286 |
| INDR | 276 | JP (IY) | 287 |
| information representation | 18 | JR cc, e | 288 |
| in-house computer | 588 | JR e | 290 |
| INI | 278 | JUMP | 90, 172, 179, 441 |
| INIR | 280 | jump instruction | 156, 182 |
| input/output | 157, 460, 518 | jump relative (JR) | 446, 447, |
| input/output devices | 511, 521 | | |
| input/output instructions | 183, 460 | K | |
| input register | 466 | 1K | 24 |
| inserting | 552, 573 | | |
| instruction | 96 | L | |
| instruction field | 590 | L | 62 |
| instruction formats | 66 | label field | 590 |
| instruction register | 55, 64 | largest element | 526, 527 |
| instruction set | 154 | LD A, (n, n) | 69, 86 |
| instruction types | 112 | LD D, C | 72 |
| INT | 91 | LDD | 164 |
| internal control registers | 51, 513 | LD DR | 164 |
| internal representation | | LDI | 164 |
| of information | 18 | LD IR | 142, 164 |
| interpreted | 69 | LD dd, (nn) | 291 |
| interpreter | 545, 581, 582 | LD dd, nn | 293 |
| interrupt | 466, 496, 497, 500, 505, 508, 509, 511 | LD r, n | 295 |
| interrupt acknowledge | 500 | LD r, r | 66 |
| interrupt flag | 187 | LD r, r' | 297 |
| interrupt handler | 502 | LD (BC), A | 299 |
| interrupt logic | 510 | LD (DE), A | 300 |
| interrupt-mask-bit | 499 | LD (HL), n | 301 |
| interrupt mode 0 | 500 | LD (HL), r | 303 |
| interrupt mode 1 | 503 | LD r, (IX + d) | 305 |
| interrupt mode 2 | 504 | LD r, (IY + d) | 307 |
| interrupt overhead | 504 | LD (IX + d), n | 309 |
| interrupt-page addressing register | 63 | LD (IY + d), n | 311 |

INDEX

| | | | |
|-----------------------|---|-------------------------|---|
| LD (IX + d), r | 313 | mantissa | 38 |
| LD (IY + d), r | 315 | MASK | 168, 522 |
| LD (nn), A | 317 | memory cycles | 55 |
| LD (nn), A | 319 | memory map | 453, 586 |
| LD (nn), dd | 321 | memory-mapped I/O | 157 |
| LD (nn), HL | 323 | memory-refresh register | 64 |
| LD (nn), IX | 325 | micro instructions | 86 |
| LD (nn), IY | 327 | mnemonic | 67, 579 |
| LD A, (BC) | 329 | M1 | 92 |
| LD A, (DE) | 330 | modes | 444 |
| LD A, I | 331 | monitor | 48, 582 |
| LD I, A | 332 | monitoring | 467 |
| LD A, R | 333 | MOS Technology 6502 | 452 |
| LD HL, (nn) | 334 | MPU | 52, 59 |
| LD IX, nn | 336 | MPU pinout | 91 |
| LD IX, (nn) | 338 | MREQ | 92 |
| LD IY, (nn) | 340 | multiple devices | 506 |
| LD IY, nn | 342 | multiple LED's | 482 |
| LD R, A | 344 | multiple precision | 98 |
| LD SP, HL | 345 | multiplexer | 52, 62 |
| LD SP, IX | 346 | multiplication | 113, 114, 115, 116, 124, 151, 152, 153 |
| LD SP, IY | 347 | MUX | 52, 62 |
| LDD | 348 | | |
| LDDR | 350 | | |
| LDI | 352 | N | |
| LDIR | 354 | N | 34 |
| LED | 41, 480 | NEG | 358 |
| LIFO structure | 540, 544 | negative | 24, 26, 32 |
| light emitting diodes | 41 | nested calls | 145 |
| linked list | 542, 544, 568, 571, 573, 574, 577, 578 | nibble | 18, 36 |
| linked loader | 583 | NMI | 91, 92, 498 |
| list | 540, 548, 549, 550, 555, 556, 557 | nonmaskable interrupt | 498 |
| listing | 590 | nonrestoring method | 133 |
| list pointer | 542 | NOP | 359 |
| literal | 69, 439, 455, 594 | NOPs | 92 |
| load | 96, 106 | normalize | 37 |
| loader | 583 | normalized mantissa | 37 |
| logarithmic searching | 546, 562 | | |
| logical | 166, 558 | O | |
| logical errors | 582 | octal | 41, 42 |
| logical operations | 141 | odometer | 465 |
| logical shift | 119 | one's complement | 25 |
| long addressing | 449 | one-shot | 466 |
| longer delay | 464 | opcode | 66, 86, 439, 444, 446 |
| | | operand | 100, 102, 438, 439 |
| M | | operating system | 582 |
| machine cycle | 69 | operator precedence | 587 |
| MACRO | 597, 598, 600 | OR | 166, 168 |
| | | OR s | 360 |

PROGRAMMING THE Z80

| | | | |
|--------------------------------|----------------------------------|------------------------------|-----------------------|
| ORG | 596 | pulse | 462, 467 |
| OTDR | 362 | pulse counting | 466 |
| OTIR | 364 | punch | 495 |
| OUT (C), r | 366 | PUSH qq | 379 |
| OUT (N), A | 368 | PUSH IX | 381 |
| OUTD | 369 | PUSH IY | 383 |
| OUTI | 371 | push | 53, 76, 154 |
| output register | 461 | | |
| overdraw | 133 | Q | |
| overflow | 28, 30, 31, 32 | queue | 543, 544 |
| overlap technique | 79 | | |
| P | | | |
| packed BCD | 36, 107 | R | 64 |
| packed BCD subtract | 110, 111 | RAM | 48, 75, 584, 587 |
| paper-tape readers | 494 | random element | 541 |
| parallel input/output | 48 | RLCA | 385 |
| parallel work transfer | 467, 468, 469 | RD | 92 |
| parity bit | 39, 40 | read operation | 96, 515 |
| parity generation | 524 | read-only memory | 48 |
| parity/overflow (P/V) | 175 | read-write memory | 48, 75 |
| PC | 52 | recursion | 148 |
| PIC | 446, 506 | reference table | 571 |
| PIO | 48, 511, 512, 513, 514, 515, 518 | register addressing | 438 |
| pointers | 51, 62, 444, 539, 544, 550, 551 | register indirect addressing | 444, 448 |
| polling | 466, 469, 492, 521, 544 | register-interrupt | 184 |
| polling loop | 493, 494 | register pairs | 51 |
| POP qq | 373 | registers | 31, 51, 149, 439, 474 |
| POP IX | 375 | relative addressing | 441, 446 |
| POP IY | 377 | relative jump | 156 |
| pop | 53, 76, 154 | relays | 461, 462 |
| port | 511, 515, 516 | request blocks | 543 |
| positional notation | 20 | RES b, s | 386 |
| positive | 24, 26, 32 | RESET | 92 |
| post-indexing | 442, 443 | restoring method | 133 |
| power failures | 48 | RET | 389 |
| pre-indexing | 442 | RET cc | 391 |
| printer | 44, 479, 495 | RETI | 181, 393, 501 |
| program | 16, 48 | RETN | 181, 395, 499 |
| program counter | 52 | RETURN | 144, 145 |
| program development | 579, 584 | RFSH | 93 |
| program loops | 63, 121 | RL s | 397 |
| programmable input/output chip | 511 | RLA | 399 |
| programmable interval | | RLC r | 103 |
| timer (PIT) | 463, 465 | RLC (HL) | 402 |
| programmer's model | 94 | RLC (IX + d) | 404 |
| programming | 15, 16, 515, 518, 602 | RLC (IY + d) | 406 |
| programming language | 16 | RLD | 408 |
| pseudo-instructions | 98 | ROM | 48 |
| | | rotation | 120, 155, 170, 171 |

INDEX

| | | | |
|---|------------------------------|---------------------------|----------------------------------|
| rotate | 50, 156 | SRA s | 430 |
| round robin | 544, 545 | SRL s | 432 |
| RR s | 410 | stack | 53, 146, 149, 496, 508, 539, 544 |
| RR A | 412 | stack pointer | 53, 540 |
| RRC s | 413 | standard architecture | 49 |
| RRCA | 415 | standard PIO | 511 |
| RRD | 416 | status | 31, 85, 476, 515 |
| RST | 183, 500 | status bits | 50, 512 |
| RST p | 418 | status register | 50 |
| rubout | 467 | storing operands | 102 |
| S | | | |
| S | 178 | string of characters | 490 |
| saving the registers | 502 | SUB A, s | 434 |
| SBC A, s | 420 | subroutine call | 143, 146 |
| SBC HL, ss | 422 | subroutine library | 150 |
| SCF | 424 | subroutine mechanism | 144 |
| scheduling | 491 | subroutine parameters | 149 |
| searching | 551, 558, 572 | subroutines | 142, 147, 443, 598 |
| segment drivers | 484 | subtraction | 104 |
| segments | 480, 541 | subtract (N) | 175 |
| sensing pulses | 466 | sum of N elements | 527, 528 |
| sequential lists | 540 | symbolic | 41, 44 |
| sequential searching | 546 | symbols | 592, 593 |
| service routing | 492 | synchronous | 471, 496 |
| SET b, s | 425 | syntactic ambiguity | 16 |
| seven-segment light-emitting diode (LED) | 480, 481 | syntax | 544 |
| shift | 50, 118, 120, 155, 156 | system architecture | 46 |
| short addressing | 441, 446, 449 | | |
| short instruction | 19 | | |
| sign | 178 | | |
| signal | 461 | | |
| signed binary | 24, 25 | | |
| signed numbers | 532 | | |
| simple list | 551 | | |
| simulator | 583 | | |
| simultaneous interrupts | 507 | | |
| single-board microcomputers | 587 | | |
| 16-bit accumulator | 103 | | |
| 16 by 8 division | 134, 135 | | |
| 16 by 16 multiplication | 130, 131 | | |
| skew operations | 169 | | |
| skip | 157 | | |
| SLA s | 428 | | |
| software aids | 582, 587 | | |
| SP | 53 | | |
| special digit instructions | 172 | | |
| speed | 476 | | |
| T | | | |
| tables | 526, 539, 540, 551, 554, 592 | technological development | 602 |
| teletype | 466, 485, 487, 488, 489 | temporary register | 61 |
| test | | test | 16, 156, 172 |
| testing a character | | testing a character | 523 |
| timer | | timer | 465 |
| time-sharing system | | time-sharing system | 588 |
| timing | | timing | 463 |
| trace | | trace | 585 |
| transfers | | transfers | 52 |
| trees | | trees | 544, 545 |
| truncating | | truncating | 34 |
| truth table | | truth table | 167 |
| two's complement | | two's complement | 25, 26, 27, 29 |
| two-level directory | | two-level directory | 541 |
| U | | | |
| UART | | UART | 477, 518 |
| underflow | | underflow | 32 |
| utility routines | | utility routines | 583 |

PROGRAMMING THE Z80

| | | | |
|-------------------------|------------|----------------------|----------|
| V | | X | |
| V | 28, 30, 31 | XOR | 166, 169 |
| \$ | 137 | XOR s | 436 |
| vectoring of interrupts | 504 | | |
| W | | Z | |
| W | 87 | Z | 87, 177 |
| WAIT | 92 | Z80 registers | 95 |
| working registers | 496 | zero | 177 |
| WR | 92 | zero page addressing | 441, 446 |
| | | Zilog Z80 PIO | 516, 517 |
| | | Zilog Z80 SIO | 518 |

The SYBEX Library

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller 340 pp., 120 illustr., Ref. B240

This second book in the "Programs for Scientists and Engineers" series provides a library of problem solving programs while developing proficiency in BASIC.

INSIDE BASIC GAMES

by Richard Mateosian

350 pp., 240 illustr., Ref. B245

Teaches interactive BASIC programming through games. Games are written in Microsoft BASIC and can run on the TRS-80, APPLE II and PET/CBM.

FIFTY BASIC EXERCISES

by J.P. Lamotier 240 pp., 195 illustr., Ref. B250

Teaches BASIC by actual practice using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.

EXECUTIVE PLANNING WITH BASIC

by X.T. Bui 192 pp., 19 illustr., Ref. B380

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

BASIC FOR BUSINESS

by Douglas Hergert

250 pp., 15 illustr., Ref. B390

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully explained accounting programs, and shows you how to write them.

BASIC EXERCISES FOR THE APPLE

by J.P. Lamotier 230 pp., 80 illustr., Ref. B500

For all Apple users, this learn-by-doing book is written in APPLESOFT II BASIC. Exercises have been chosen for their educational value and application to math, physics, games, business, accounting, and statistics.

YOUR FIRST COMPUTER

by Rodnay Zaks 260 pp., 150 illustr., Ref. C200A

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

DON'T

(or How to Care for Your Computer)

by Rodnay Zaks 220 pp., 100 illustr., Ref. C400

The correct way to handle and care for all elements of a computer system including what to do when something doesn't work.

INTRODUCTION TO WORD PROCESSING

by Hal Glatzer 200 pp., 70 illustr., Ref. W101

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

INTRODUCTION TO WORDSTAR

by Arthur Naiman 200 pp., 30 illustr., Ref. W105

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

by Rodnay Zaks 560 pp., 255 illustr., Ref. C201A

A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

MICROPROCESSOR INTERFACING TECHNIQUES

by Rodnay Zaks and Austin Lesca

460 pp., 400 illustr., Ref. C207

Complete hardware and software interconnect techniques including D to A conversion, peripherals, standard buses and troubleshooting.

PROGRAMMING THE 6502

by Rodnay Zaks 390 pp., 160 illustr., Ref. C202

Assembly language programming for the 6502, from basic concepts to advanced data structures.

6502 APPLICATIONS BOOK

by Rodnay Zaks 280 pp., 205 illustr., Ref. D302

Real life application techniques: the input/output book for the 6502.

ADVANCED 6502 PROGRAMMING

by Rodnay Zaks 300 pp., 140 Illustr., Ref. G402
Third in the 6502 series. Teaches more advanced
programming techniques, using games as a framework
for learning.

PROGRAMMING THE Z80

by Rodnay Zaks 620 pp., 200 Illustr., Ref. C280
A complete course in programming the Z80
microprocessor and a thorough introduction to
assembly language.

PROGRAMMING THE Z8000

by Richard Mateosian

300 pp., 125 Illustr., Ref. C281
How to program the Z8000 16-bit microprocessor.
Includes a description of the architecture
and function of the Z8000 and its family of support
chips.

THE CP/M HANDBOOK (with MP/M)

by Rodnay Zaks 330 pp., 100 Illustr., Ref. C300
An indispensable reference and guide to CP/M—
the most widely used operating system for small
computers.

INTRODUCTION TO PASCAL (Including UCSD PASCAL)

by Rodnay Zaks 420 pp., 130 Illustr., Ref. P310
A step-by-step introduction for anyone wanting
to learn the Pascal language. Describes UCSD
and Standard Pascals. No technical background
is assumed.

THE PASCAL HANDBOOK

by Jacques Tiberghien

490 pp., 350 Illustr., Ref. P320
A dictionary of the Pascal language, defining
every reserved word, operator, procedure and
function found in all major versions of Pascal.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan Miller 400 pp., 80 Illustr., Ref. P340

A comprehensive collection of frequently used
algorithms for scientific and technical applications,
programmed in Pascal. Includes such programs as
curve-fitting, integrals and statistical techniques.

APPLE PASCAL GAMES

by Douglas Hergert and Joseph T. Kalash

380 pp., 40 illustr., Ref. P360

A collection of the most popular computer games
in Pascal challenging the reader not only to play
but to investigate how games are implemented on
the computer.

INTRODUCTION TO UCSD PASCAL SYSTEMS

by Charles T. Grant and Jon Butah

300 pp., 110 illustr., Ref. P370

A simple, clear introduction to the UCSD Pascal
Operating System for beginners through experienced
programmers.

INTERNATIONAL MICROCOMPUTER DICTIONARY

140 pp., Ref. X2

All the definitions and acronyms of microcomputer
jargon defined in a handy pocket-size
edition. Includes translations of the most popular
terms into ten languages.

MICROPROGRAMMED APL IMPLEMENTATION

by Rodnay Zaks 350 pp., Ref. Z10

An expert-level text presenting the complete
conceptual analysis and design of an APL inter-
preter, and actual listings of the microcode.

FOR A COMPLETE CATALOG OF OUR PUBLICATIONS



U.S.A.
2344 Sixth Street
Berkeley,
California 94710
Tel: (415) 848-8233
Telex: 336311

SYBEX-EUROPE
4 Place Félix-Eboué
75583 Paris Cedex 12
France
Tel: 1/347-30-20
Telex: 211801

SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
West Germany
Tel: (0211) 287066
Telex: 08 588 163

PROGRAMMING THE Z80

has been designed as an educational text and a self-contained reference manual. This book presents a thorough introduction to machine language programming, from basic concepts to advanced data structures and techniques. Detailed illustrative examples and numerous programs show the reader how to write clear, well-organized programs in the language of the Z80.

This book is the result of the author's extensive experience in the fields of education and programming, and it has been designed for clarity and readability. All concepts are explained in simple yet precise terms, building progressively toward more complex techniques. The reader will gain not only an understanding of programming in the language of the Z80, but also a detailed understanding of the way a microprocessor actually executes instructions.

Among the subject areas covered in PROGRAMMING THE Z80 are:

- Z80 Hardware Organization
- Input/Output Techniques
- Complete Instruction Set
- Z80 Addressing Modes
- Data Structures—Theory and Design
- Applications Examples

With over 200 illustrations, a thorough index and 7 appendices, PROGRAMMING THE Z80 is an indispensable work for engineers, students, home computerists and anyone interested in learning machine language programming skills.

THE AUTHOR

Dr. Rodney Zaks has been involved with the industrial use of microprocessors since their initial development. He is the author of a number of best-selling books on all aspects of microprocessors, and has taught microprocessor courses to several thousand people internationally, ranging from the introductory level to bit-slice microprogramming techniques. He holds a Ph.D. in Computer Science from the University of California, Berkeley, and is a member of ACM and IEEE.