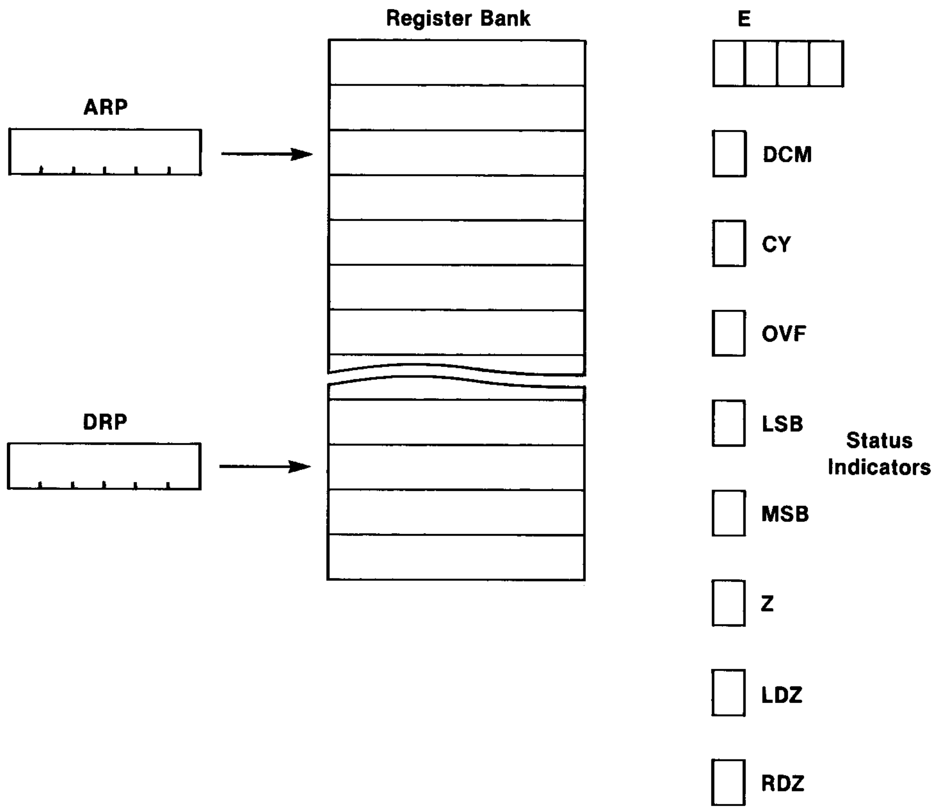
SECTION 3

CPU STRUCTURE AND OPERATION

This section explains the structure, addressing modes and operation of the central processing unit (CPU) in the HP-83/85.

The HP-83/85 CPU consists of a 64_{10}-byte register bank, a pair of address pointers called the address register pointer (ARP) and the data register pointer (DRP), an arithmetic and logic unit (ALU) and a shifter, and a set of status indicators.



CENTRAL PROCESSING UNIT

ARP AND DRP

The address register pointer (ARP) and the data register pointer (DRP) are independent six-bit CPU locations. Both the ARP and the DRP can be used to address any of the bytes in the CPU register bank.

The CPU register addressed by the ARP is called the address register, or AR. The register addressed by the DRP is called the data register, or DR.

CPU REGISTER BANK

The heart of the CPU is the register bank of 64 8-bit bytes of random-access memory. These bytes form registers which are grouped into two-byte (16-bit) sections and eight-byte (64-bit) sections. The diagram on the following page shows the organization of the CPU registers, which are numbered from 0 to 77_8, and specified by RØ - R77.

Some of the registers in the CPU register bank are dedicated by hardware to specific tasks.

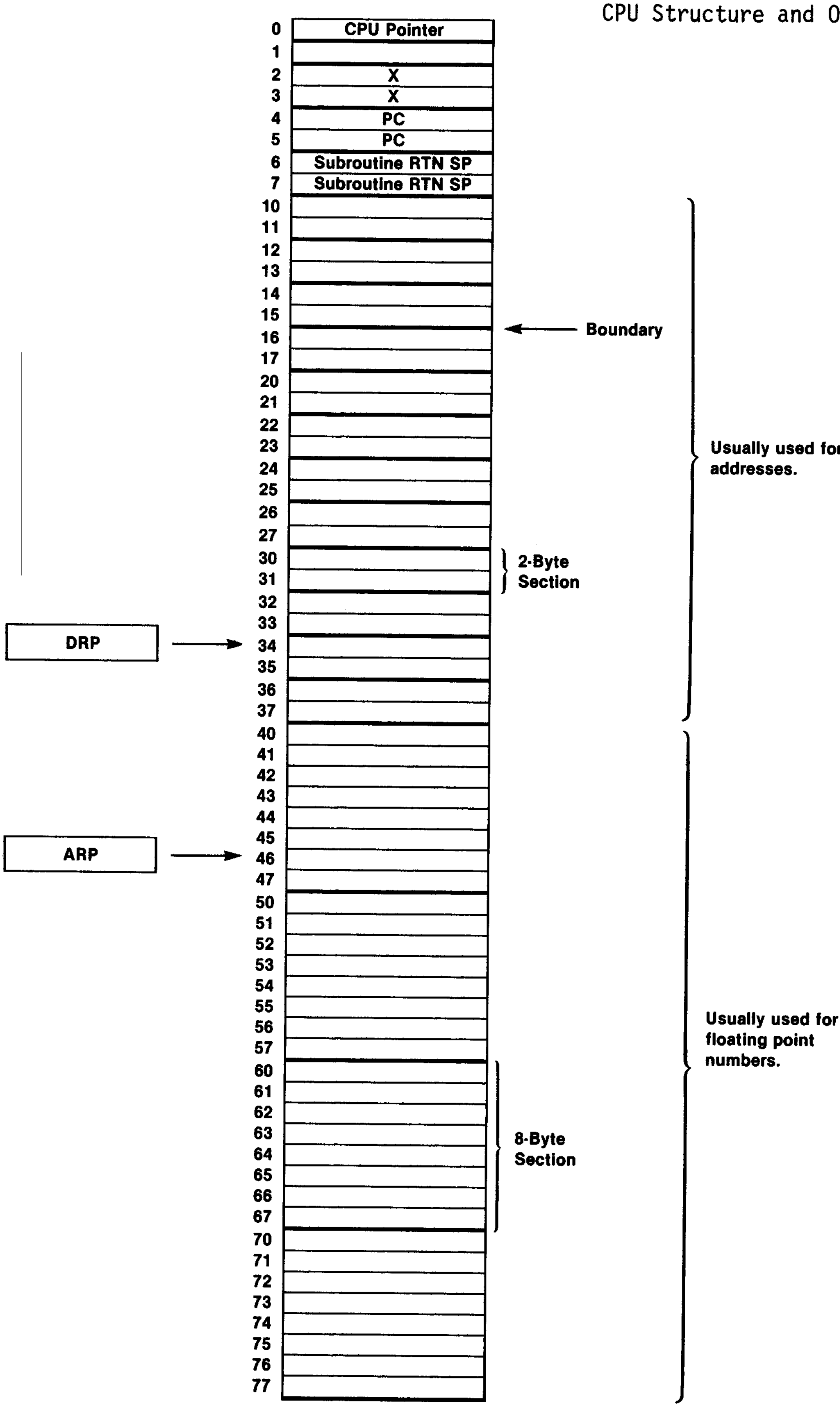
HARDWARE-DEDICATED REGISTERS

The first 40_8 registers of the CPU (RØ - R37) are divided into two-byte (16-bit) sections. Of these, many of the bytes are reserved by hardware for use as special-purpose registers. These hardware-dedicated registers are:

Register Bank Pointer. Register 0 is a pointer to the remainder of the CPU register bank. Register 1 is inaccessible except through register 0.

Index Scratch. Registers 2 and 3 are scratch registers used for indexed addressing (X). Their contents are destroyed by execution of instructions using indexed addressing.

Program Counter. Registers 4 and 5 contain the program counter (PC).



CPU REGISTER BANK

Return Stack Pointer. Registers 6 and 7 contain the pointer for the subroutine return stack. (The space allocated for this stack in the computer's system memory comprises addresses 101300 through 101777, although sometimes these addresses may be used for other purposes.)

In addition to the special-purpose registers described above, certain other CPU registers are commonly used for specific purposes by internal HP-83/85 routines. (For example, registers R40 and R50 are used by internal mathematics routines for addition, subtraction, etc.)

REGISTER BOUNDARIES

The CPU registers are separated by boundaries, shown as heavy lines in the illustration of the register bank above. In the first 32 bytes, there is a boundary every two bytes. In the next 32 bytes, there is a boundary every eight bytes.

This partitions the first 32 bytes into 16-bit sections (used primarily for address manipulation) and the next 32 bytes into 64-bit sections (used primarily for floating point quantities). The register array is, therefore, capable of holding up to four floating-point numbers and twelve 16-bit addresses.

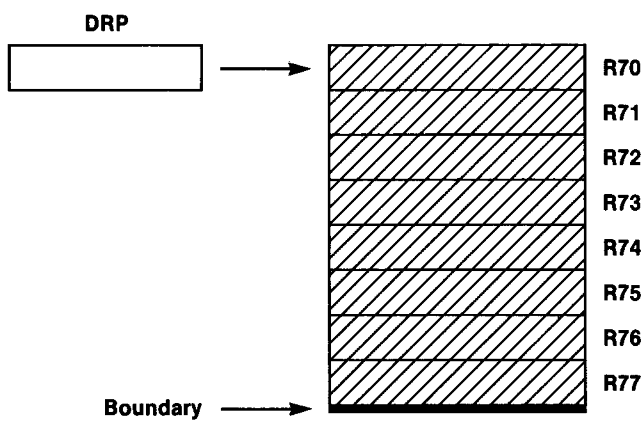
MULTI-BYTE OPERATIONS

The HP-83/85 CPU structure permits "multi-byte operations," involving a string of bytes rather than just a single byte. A string can consist of from one to eight consecutive CPU registers. The exact number is determined by the DRP and the next boundary.

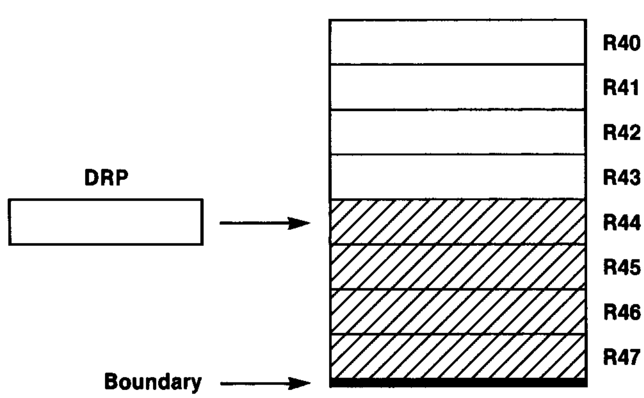
The locations involved in a multi-byte operation are those beginning with the location pointed to by the DRP and ending with the next boundary. The next boundary is the one in the direction of increasing addresses (except in the case of a shift right instruction.)

The following examples should help explain this concept:

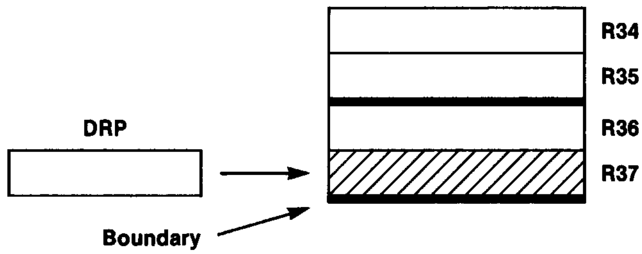
--A multi-byte increment with DRP set to 70 (that is, executing ICM R70) results in an increment of the 64-bit quantity stored between locations R70 and R77. Higher addresses always refer to more significant bytes.



--A multi-byte test with DRP set to 44 (that is, executing TSM R44) results in the status flags being set according to the data found in registers R44, R45, R46 and R47. Location R47 is the most significant byte.

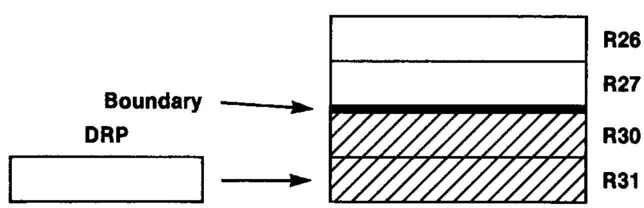


--A multi-byte complement with DRP set to 37 (that is, executing TCM R37) complements only R37.



The only exception to the rule that the next boundary is in the direction of increasing addresses is the shift right instruction. If a multi-byte instruction is a shift right, then the next boundary is the one in the direction of decreasing addresses.

Thus:
--A multi-byte shift right with DRP set to 31 (that is, executing LRM R31) shifts the combined contents of R31 and R30 right. R31 is the most significant byte.



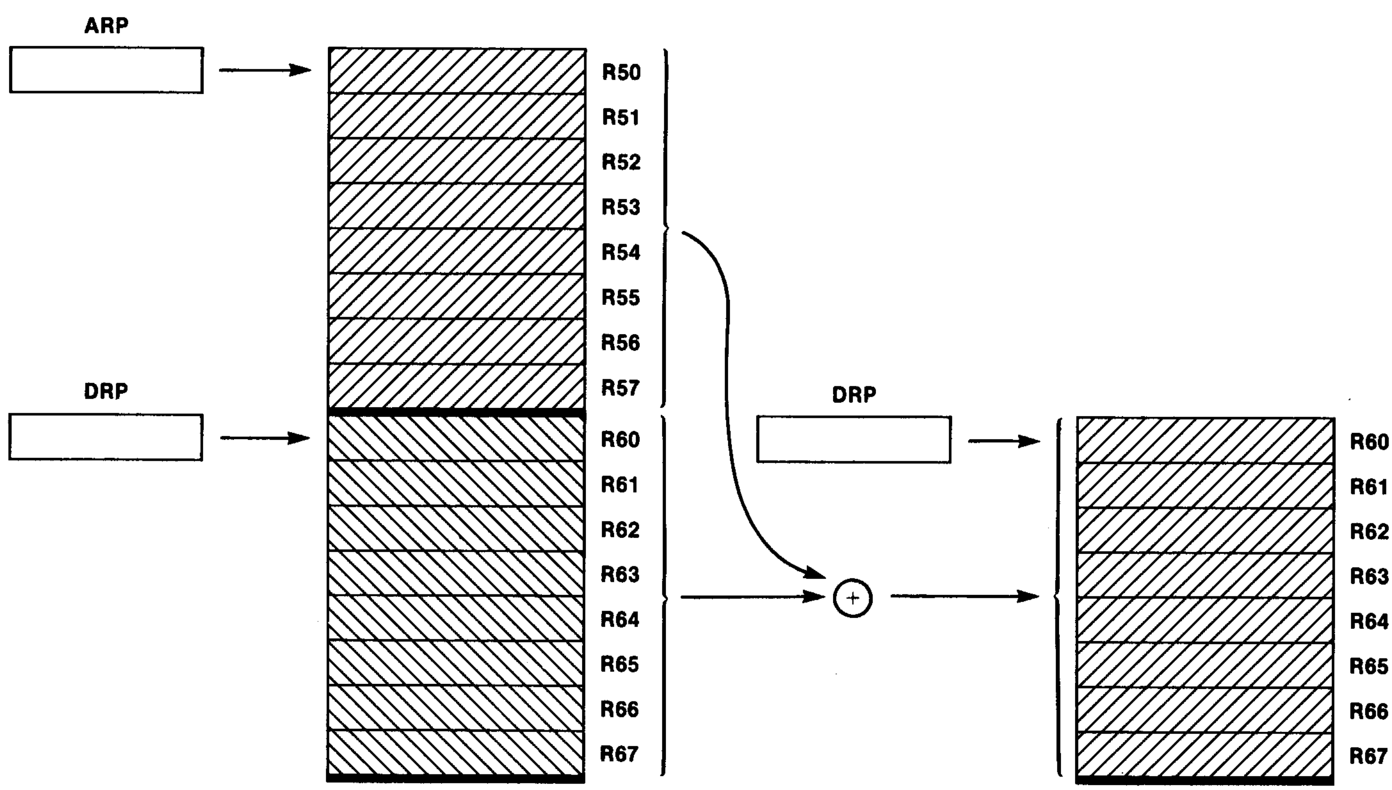
SINGLE-BYTE OPERATIONS

Besides executing multi-byte instructions, the HP-83/85 CPU also executes instructions using single bytes. In a single-byte operation, the DRP refers to only a single byte.

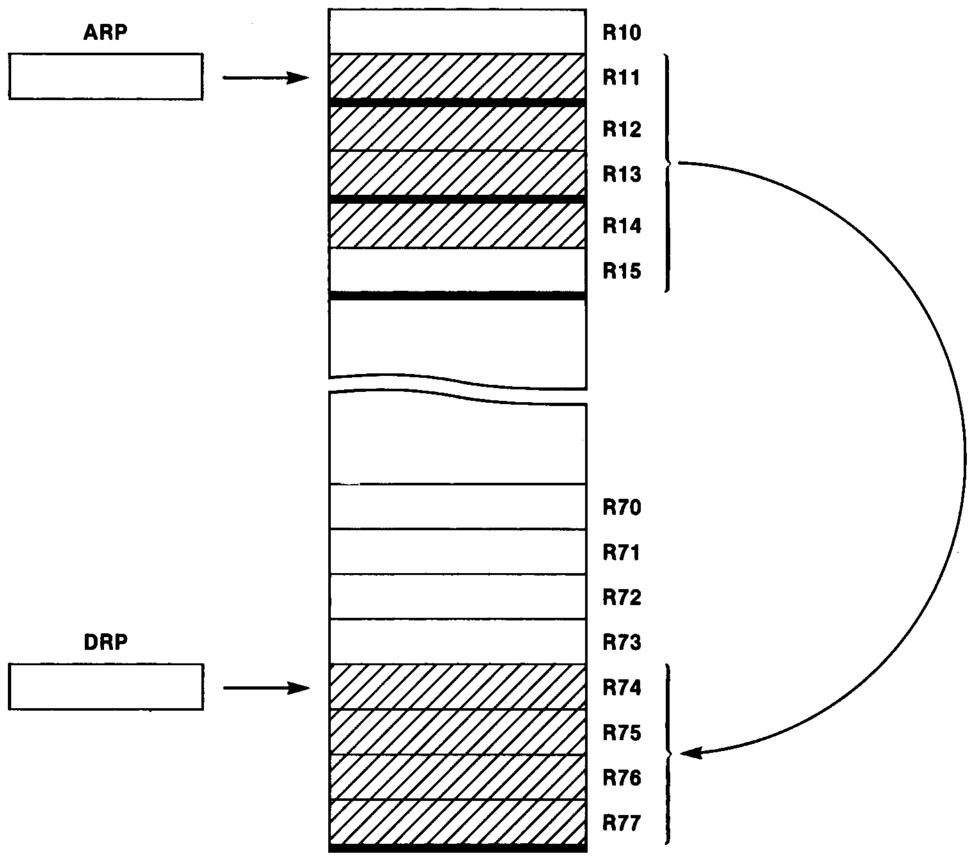
TWO-OPERAND OPERATIONS

Two-operand multi- and single-byte instructions may also be executed. In the case of a multi-byte two-operand instruction, DRP points to the first operand and ARP points to the second. DRP is still used to determine the number of bytes involved for the first operand. The other operand consists of the same number of bytes, beginning with the location to which the ARP points. For example:

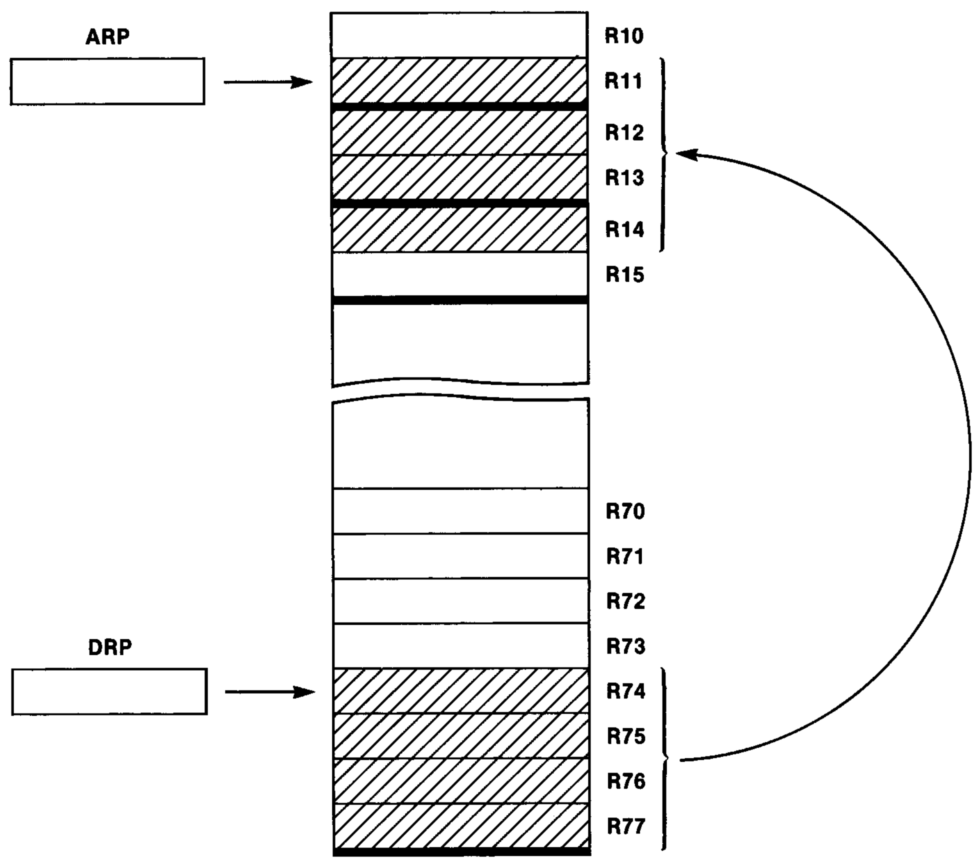
--A multi-byte add with DRP set to 60 and ARP set to 50 (that is, executing ADM R60, R50) results in the 64-bit quantity starting with R50 being added to the 64-bit quantity starting with R60. The sum is stored in R60 through R67.



--A multi-byte load with DRP set to 74 and ARP set to 11 (that is, executing LDM R74, R11) transfers the contents of four bytes beginning with R11 to locations R74, R75, R76 and R77.



--A multi-byte store with DRP set to 74 and ARP set to 11 transfers the contents of R74 through R77 to the four consecutive locations beginning with R11.



Remember: The number of bytes in a multi-byte operation is always determined by the setting of DRP (not ARP) and the next boundary.

There are also two-operand operations where the DRP points to one operand and the second is located in the computer's memory. Once again, the number of bytes to be operated upon is determined by the DRP. The corresponding number of bytes are accessed from memory beginning with the calculated effective address.

NUMBER REPRESENTATION

Numbers in the HP-83/85 are manipulated in a variety of formats. The user has the option of specifying quantities as octal, BCD or decimal. In addition, the internal quantities used in the HP-83/85 occur in various formats, depending on their use.

ADDRESSES

An address, whether in the CPU register bank or in system memory, is always an octal value that occupies two bytes, or 16 bits. The lower-numbered byte contains the less significant byte of the address, and the higher-numbered byte contains the more significant byte of the address. Only the first byte of the two-byte address is referenced by other instructions.

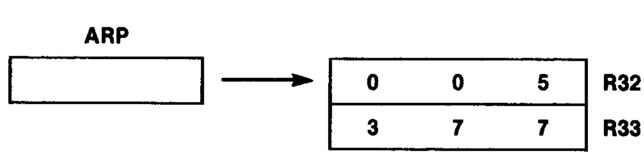
For example, address 177405, translated into a binary quantity, appears like this:

1	7	7	4	0	5	Octal Representation
1	111	111	100	000	101	Binary Representation

When this binary quantity is split into two eight-bit registers, it appears as:

11	111	111	00	000	101	Binary Quantity
3	7	7	0	0	5	Register Contents

Only the first byte of the two-byte address is referenced by other instructions, so an address pointing to ROM location 177405 from the CPU might look like this:



NUMERIC QUANTITIES

Numeric quantities in the HP-83/85 may be of three types: Real, short, and integer. The following illustration shows how numeric quantities are represented internally in the computer. For the illustration, the numbers are shown in CPU registers R40 - R47.

Real

40	E1	E2
41	E0	MS
42	M10	M11
43	M8	M9
44	M6	M7
45	M4	M5
46	M2	M3
47	M0	M1

Integer

45	D1	D0
46	D3	D2
47	S	D4

Short

44	E0	E1
45	M3	M4
46	M1	M2
47	0 0 SM SE	M0

FORMATS OF NUMERIC QUANTITIES

In real or floating-point format, the mantissa is a 12-digit quantity expressed as a magnitude. Each digit consists of four bits. The least significant digit, represented by M11, is stored in R42. The most significant digit, represented by MØ, is stored in R47. The number is normalized; thus, there is an implied decimal point between MØ and M1 in R47. The sign of the mantissa is stored in the least significant digit of R41. A zero is stored as the sign of the mantissa if the number is positive; otherwise, a nine is stored. The exponent is a three-digit number stored in R40 and in the most significant digit position of R41. Exponents are expressed in ten's complement form.

Integer variables are stored in three bytes, with five digits and a sign. Short variables are stored as a mantissa sign (SM) an exponent sign (SE), five mantissa digits, and a two-digit exponent.

STATUS INDICATORS

The HP-83/85 CPU contains eight flags and a four-bit register for program status. The flags signal the present condition of the data, while the four-bit register serves as an "extended" register for counting and data manipulation.

Status can affect or be affected by CPU instructions. In the HP-83/85 CPU, the instruction set has data movement instructions of both the arithmetic and non-arithmetic types. These instructions include:

--Arithmetic: Add, subtract, compare, increment, decrement, complement.

--Non-arithmetic: Load, store, logical and, or, exclusive or, shift, clear, test.

The following status indicators are present in the HP-85 CPU:

E: Extend Register. A four-bit register which can be cleared, incremented, or decremented independent of DCM. Shifts can be made into and out of the extend register only when DCM is set.

DCM: Decimal Mode Flag. When set, binary-coded decimal (BCD) operations will be performed. When cleared, binary operations will be performed. The operations affected by DCM are all the arithmetic data movement instructions and the shift instructions. The DCM flag can be modified only by two CPU instructions, BCD and BIN. The BCD instruction sets DCM, while the BIN instruction clears DCM.

CY: Carry Flag. This one-bit register can be shifted into and out of when DCM is cleared (i.e., BIN mode). It is loaded with the carry from the most significant bit (MSB) according to the table shown here:

CPU Instruction	Carry Flag
Add	CY set according to carry of add.
Subtract	CY set if result is positive, cleared if result is negative.
Compare	Same setting as for subtract.
Increment	CY set as for add.
Decrement	CY set as for subtract.
Shift	CY loaded with bit shifted out, if in binary mode. (Right shift loads CY from LSB.)
Complement	CY cleared by nine's complement, set by ten's complement, if contents of data register (DR) were zero.

All other data movement instructions clear CY.

OVF: Overflow Flag. The overflow flag is set whenever the result of a binary arithmetic operation exceeds the maximum positive or negative number that can be contained in the destination register. This can occur as the result of a compare, binary add, binary subtract, binary complement, or binary left shift instruction. Thus, an arithmetic data movement instruction or a left shift with DCM cleared affects OVF; all other data movement instructions clear OVF. The remaining instructions do not affect OVF.

LSB: Least Significant Bit Flag. LSB is set the same as the least significant bit (LSB) of the result of each data movement instruction.

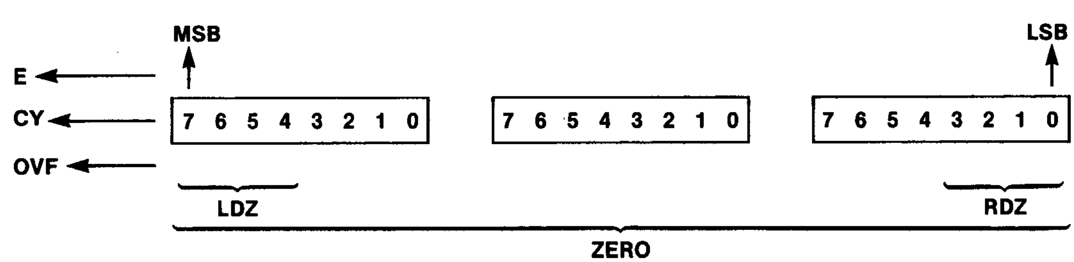
MSB: Most Significant Bit Flag. MSB is set the same as the most significant bit (MSB) of the result of each data movement instruction.

Z: Zero Flag. Z is set if a data movement instruction produces a result of all zeros. If the result is not all zeros, Z is cleared. Other instructions do not affect Z.

LDZ: Left Digit Zero Flag. LDZ is affected only by data movement instructions. LDZ is set if the most significant nibble (four bits) of the result is 0000. If the most significant four bits are not 0000, LDZ is cleared.

RDZ: Right Digit Zero Flag. RDZ is affected only by data movement instructions. RDZ is set if the least significant nibble (four bits) of the result is 0000, regardless of the setting of DCM. If the most significant four bits are not 0000, RDZ is cleared.

Status information is based on the entire single or multi-byte quantity that is processed. The figure below illustrates status on a three-byte quantity.



MULTI-BYTE STATUS

All multi-byte operations except right shift start execution with the least significant byte. All status flags except LSB, RDZ, and DCM are updated after each byte of an operation, and therefore will be correct whenever the memory boundary is reached. The LSB and RDZ flags are set only for the first byte.

For a shift right instruction, where the shift is from the most significant byte to the least significant, the MSB and LDZ flags are set only for the most significant byte; the rest are updated after each byte.

For a complete list of all CPU instructions and their relationships to status indicators, refer to section 4 and appendix C.

SECTION 4

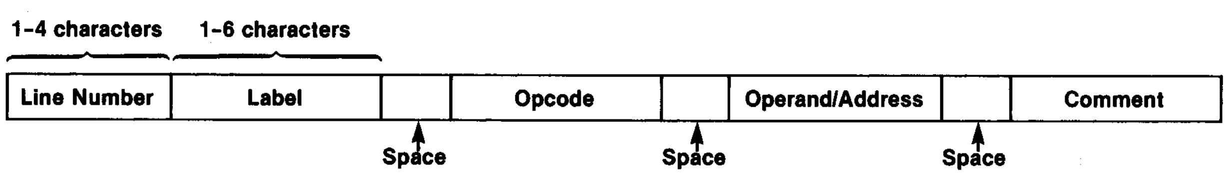
ASSEMBLER INSTRUCTIONS

The HP-83/85 Assembler instructions can manipulate data in the HP-83 or HP-85 central processing unit, and through the CPU, in HP-83/85 RAM as well.

Assembler instructions are of two types: Instructions and pseudo-instructions. Instructions operate directly on the CPU and during assembly are translated directly into machine language object instructions. They are specified by means of opcodes. Pseudo-instructions are entered in the same way as CPU instructions, but they are actually messages to the Assembler ROM. They are specified by means of pseudo-opcodes.

ENTERING INSTRUCTIONS AND PSEUDO-INSTRUCTIONS

Source code is typed into the CRT by entering the line number, followed by a label (if any), followed by the opcode, followed by the address or operand, if required, followed by a comment (if any). When [END LINE] is then pressed, the line is parsed and the elements are assigned to their respective fields on the CRT.



SOURCE CODE INSTRUCTION FORMAT

In assembler mode, the HP-83/85 is sensitive to spacing among the elements of a line of source code. For example:

A statement entered to the CRT as:

```
6Ø LBL LDMD R70,R40
7Ø Label jsb=numval
8Ø  PUBD R52,+R12
9Ø  PUBD 52,+12
1ØØ  CLB R40 !THIS IS A COMMENT
```

After parsing appears as:

```
60 LBL    LDMD R70,R40
70 Label  JSB =NUMVAL
80        PUBD R52,+R12
90        PUBD R52,+R12
100        CLB R40
!THIS IS A COMMENT
    ↑       ↑         ↑
Label   Opcode    Operand or
Field   Field     Address Field
```

LINE NUMBERING

Each line of binary program source code must begin with a line number. These line numbers may be entered individually, or automatic line numbering may be specified with the [AUTO] key.

These line numbers are useful for entering and editing a binary program, but do not correspond to the addresses of the machine language object code that is generated during assembly.

LABELS

No spaces or one space may be typed between the line number and the label field. A label is optional, and may be from one to six characters. A label cannot have a digit as the first character, nor a space as any character; one or more spaces denote the end of the label.

When a label has been entered and parsed, it appears in a label field on the CRT or printer. This field begins in the second character space to the right of the line number.

OPCODES AND PSEUDO-OPCODES

The opcodes and pseudo-opcodes for assembly language instructions may be entered after typing at least two spaces after the line number or at least a single space after a label. Entries in the opcode field are restricted to valid instructions and pseudo-instructions. Blanks are not allowed within the opcode field.

When an opcode or pseudo-opcode has been entered and parsed, it begins in the field nine spaces to the right of the line number.

Opcodes (but not pseudo-opcodes) may be either single-byte (specified by a "B") or multi-byte (specified by an "M").

OPERANDS OR ADDRESSES

Depending upon the format of the instruction, the operand or address field may specify one or more of the following:

--Data Register. A CPU register which may signify single-byte or multi-byte operation.

--Operand. May be a CPU register or a memory location. Depending on the addressing mode, memory can be addressed immediately, indirectly, or by an index.

--Register Pointer. Constant used to load ARP or DRP.

--Label. A label to specify an address or constant.

--Nothing. Some instructions do not require an entry in this field.

An AR or DR in the CPU is specified by an "R" before the register number (e.g., R32), or by an "X" before the register number when indexed addressing is used. The "R" may be omitted when CPU register numbers are typed, since the assembler inserts a missing "R" automatically. The "X" must be typed to indicate register numbers for indexed operations.

COMMENTS

A comment or remark must begin with an exclamation point. A comment must be typed beginning in the first or second space after the line number, or beginning one or more spaces after the other elements of the line of source code.

After being parsed, a comment which has been entered immediately following the other elements of the line begins in column 33; thus, on the HP-83/85 CRT it appears on the following line. A peripheral printer with a column width greater than 32 can permit a comment to appear on the same line as the source code statement.

NUMERIC VALUES

Numeric values can be entered in octal, BCD or decimal notation. A BCD value is entered by immediately following the value with a "C," while a decimal value is followed by a "D;" otherwise the assembler assumes octal values.

Example: LDM R45,=31, 19C, 25D Loads the same bit pattern into registers R45, R46 and R47.

Registers can be specified by octal values only.

SYNTAX AND SYMBOLS USED

The following shows the syntax guidelines once again and also includes a list of the symbols used in the descriptions of assembler instructions.

LDB	Instructions shown in capital letters, but not underlined, must be entered exactly as shown (in either upper-case or lower-case letters).
___	Items shown underlined (e.g., DR) are expressions or names that must be specified in the instruction, statement, or command.
[ ]	Items shown between brackets are optional. (e.g., CMB[D] indicates there is a CMB instruction and also a CMBD instruction available.) If several items are stacked between brackets, any one or none of the items may be specified.
...	Three dots (ellipsis) following a set of brackets indicate that the items between the brackets may be repeated.
←	Is transferred to.
( )	Contents of.
‾‾‾‾	Complement (e.g., $\overline{x}$ is complement of x). This is one's complement if DCM=0 and nine's complement if DCM=1.

B/M Single-byte or multi-byte instruction.

AR Address register location--location of first byte addressed by ARP. Can be a register (e.g., R32), R* or R#.

DR Data register location--location of first byte addressed by DRP. Can be a register (e.g., R32), R* or R#.

A Address mode for load/store. Can be blank (for immediate), D (for direct), or I (for indirect).

ARP Address Register Pointer. A 6-bit register used to point to one of 64 CPU registers. The byte to which ARP points is often used as the first of two consecutive bytes forming a memory address.

DRP Data Register Pointer. A 6-bit register used to point to one of 64 CPU registers. The location to which DRP points is often used as the destination for data loaded into the CPU.

R(x) CPU register addressed by (x).

M(x) Memory location addressed by (x). (x) must be a 16-bit address.

PC Program Counter. CPU registers R4 and R5. Used to address the instruction being executed.

SP Subroutine Stack Pointer. CPU registers 6 and 7. Used to point to the next available location on the subroutine return address stack.

EA Effective Address. The location from which data is read for load-type instructions or the location where data is placed for store-type instructions.

ADR Address. The two-byte quantity directly following an instruction that uses the literal direct, literal indirect, index direct or index indirect addressing mode. This quantity is always an address.

The following pages show the HP-83/85 Assembler ROM instructions that are used to manipulate the CPU and external memory. These instructions are illustrated in an abbreviated form in this section; for a complete list of all forms of each instruction, refer to appendix C.

Also contained in this section are the Assembler ROM pseudo-instructions.

LOAD/STORE INSTRUCTIONS

The instructions for loading and storing data have access to all eight addressing modes, and they can be single-byte or multi-byte.

LD

CPU Instruction

Load

Format: LDBA DR, operand — Single byte
LDMA DR, operand — Multi-byte

Operation: DR←(EA)

Description: Data register is loaded with the contents of the effective address determined by the operand and the addressing mode.

ST

CPU Instruction

Store

Format: STBA DR, operand — Single byte
STMA DR, operand — Multi-byte

Operation: (DR)→EA

Description: Contents of data register are stored in effective address determined by the operand and the addressing mode.

ADDRESSING MODES

The HP-83/85 CPU allows for several addressing modes. These include literal, register, indexed and stack modes of memory access.

Not all addressing modes are available to all instructions. The load (LD) and store (ST) instructions have access to all addressing modes except stack addressing, and they are used here for illustration. For a list of the addressing modes available to any particular instruction, consult the description of that instruction in this section or in appendix C.

In addressing, all addresses are referred to as two-byte quantities. Because all addresses are two consecutive bytes, only the first byte of the sequence is referenced. For instance, the AR is actually a single byte within the CPU register bank that is pointed to by the ARP. When the AR is described as being an address, remember that R (ARP) contains the low byte of the address and R (ARP + 1) contains the upper byte of the address.

The multi-byte feature of the CPU allows data to be manipulated in quantities of from one to eight bytes. Therefore, in the following descriptions, only the address of the first byte of data is specified. As explained earlier, the number of bytes is determined by the distance of the DR from the next consecutive boundary.

In the following descriptions, the effective address (EA) points to the first byte of data to be loaded for load instructions.

For store instructions, EA points to the location where the first byte of data is stored.

REGISTER MODE

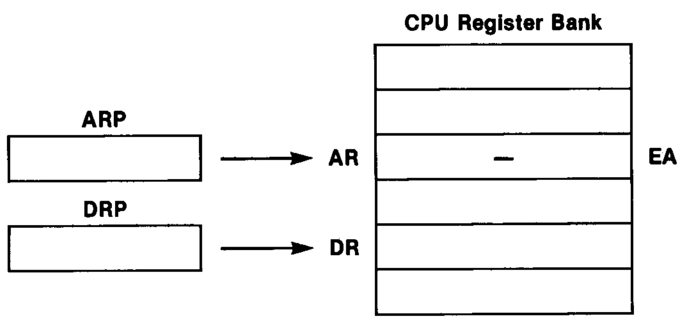
The first category of addressing is the register addressing mode. This mode allows the CPU registers (64_{10} bytes) to be used as addresses as well as for data. There are three levels of register addressing modes.

REGISTER IMMEDIATE

Format: Opcode B/M DR, AR

Effective Address: AR

Description: The operand is another CPU register (single or multi-byte) beginning at AR. Thus, the AR is the source for load instructions or the destination for store instructions.



REGISTER IMMEDIATE ADDRESSING

Examples: LDB R36, R32 Loads contents of R32 into CPU register R36.

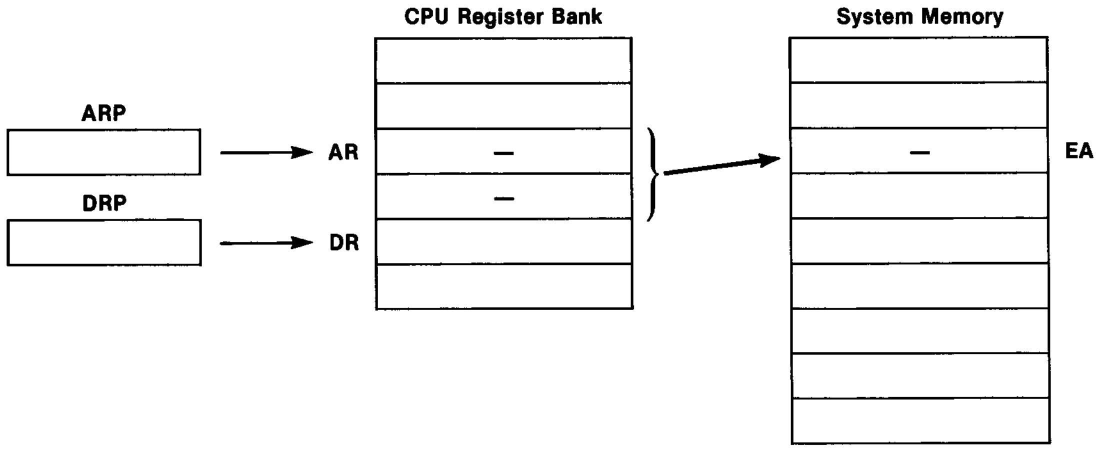
STM R40, R50 Stores contents of registers R40 through R47 into registers R50 through R57.

REGISTER DIRECT

Format: Opcode B/M D DR, AR

Effective Address: M(AR)

Description: The effective address is a location in system memory that is addressed by the AR. This mode is useful when using a CPU register as a pointer to system memory.



REGISTER DIRECT ADDRESSING

Examples: LDBD R36, R32 Loads CPU register R36 with the contents of the system memory location addressed by R32-R33.

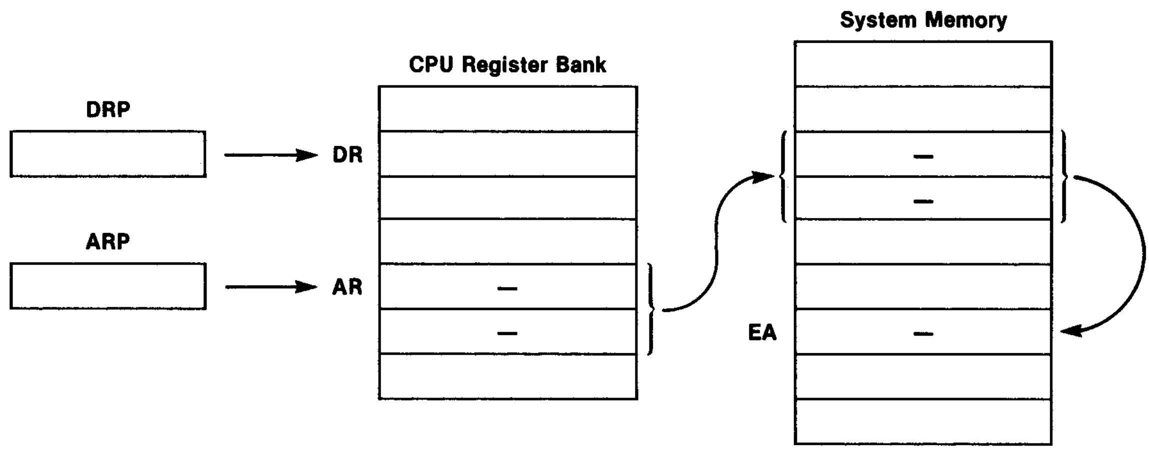
STMD R40, R50 Stores contents of R40-R47 into system memory beginning with location addressed by R50-R51.

REGISTER INDIRECT

Format: Opcode B/M I DR, AR

Effective Address: M(M(AR))

Description: The address register points to a system memory location, which in turn points to another memory location that is the effective address.



REGISTER INDIRECT ADDRESSING

Example: LDBI R36, R32 If R32 and R33 contain the address 105371, loads CPU register R36 with the contents of the memory location that is addressed by the contents of system memory locations 105371 and 105372.

LITERAL MODE

The second of the categories of address modes is the literal mode. In literal mode, the operand is a literal quantity stored in memory immediately following the opcode. A literal string can be:

--BCD constant, e.g., 99C, ..., 79C ($\leq 10_8$ bytes)

--Octal constant, e.g., 12, ..., 277 ($\leq 10_8$ bytes)

--Decimal constant, e.g., 201D, ..., 9D ($\leq 10_8$ bytes)

--Label (The literal quantity is a one- or two-byte value or address assigned to the label.)

The programmer is responsible for ensuring that the number of bytes of the literal string matches the DRP setting. The assembler does not check for mismatch.

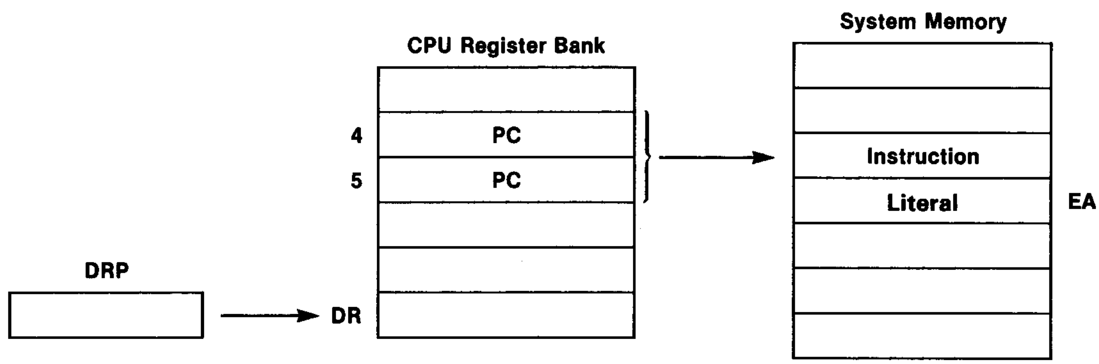
There are three types of literal addressing modes.

LITERAL IMMEDIATE

Format: Opcode B/M DR, = literal

Effective Address: (PC+1)

Description: The operand is a literal string that, during assembly, is stored in memory immediately after the instruction opcode. This mode is useful for loading constants into the CPU register bank.



LITERAL IMMEDIATE ADDRESSING

Examples: LDB R36, = 3D Loads 3_{10} into CPU register R36.

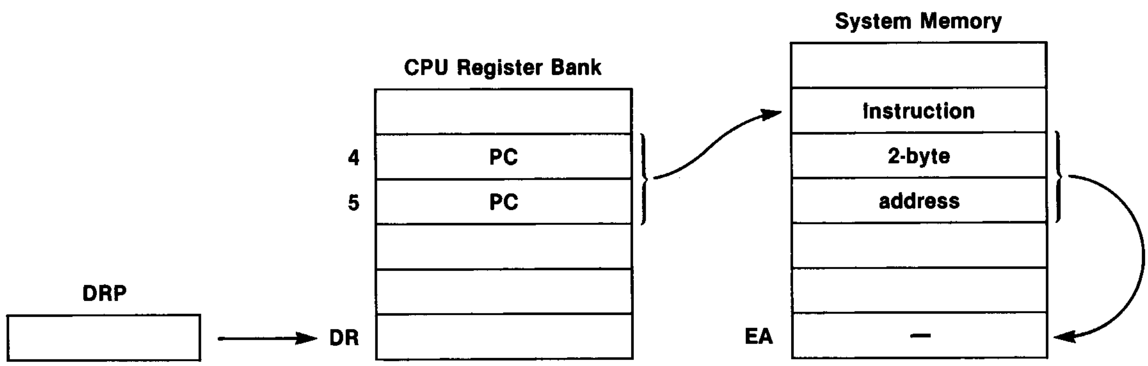
LDM R40, = 0,0,0,0,0,0,0,120 Loads 120_8 (i.e., a floating-point 5) into registers R40-R47.

LITERAL DIRECT

Format: Opcode B/M D DR, = label

Effective Address: M(PC+1)

Description: The operand is a memory location that, after assembly, is addressed by a two-byte literal quantity stored immediately after the instruction opcode. The label defines the two-byte literal quantity to be used by the Assembler ROM.



LITERAL DIRECT ADDRESSING

Examples: LDBD R34, = ROMFL Loads the contents of the memory location addressed by the label ROMFL into CPU register R34.

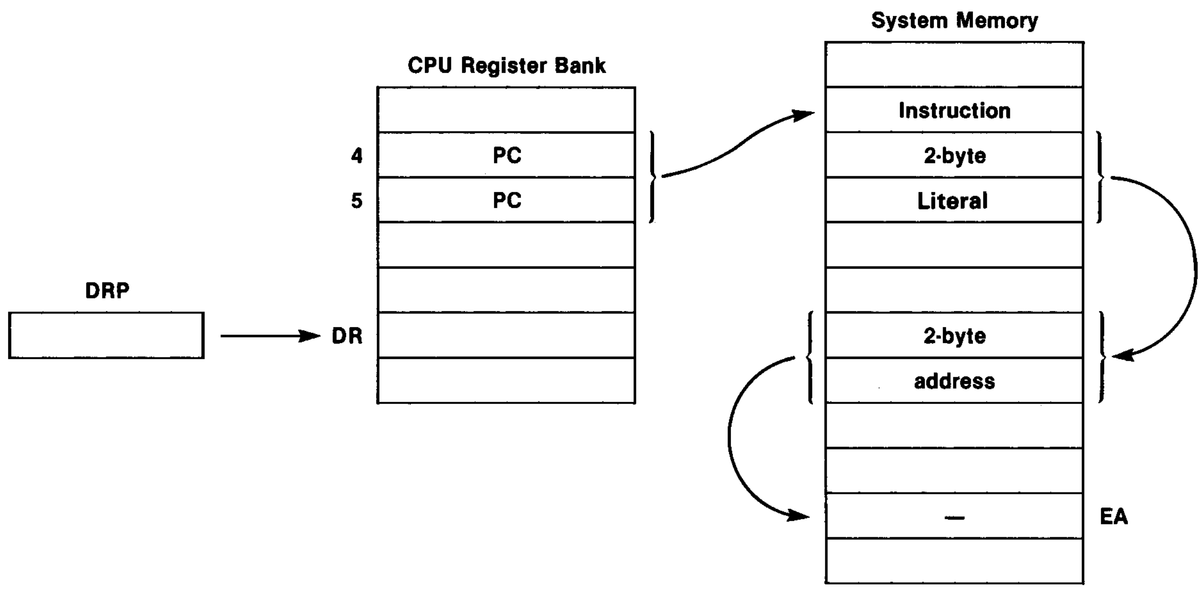
STMD R74, = CHIDLE Stores contents of CPU registers R74 through R77 into four memory locations beginning with the location addressed by the label CHIDLE.

LITERAL INDIRECT

Format: Opcode B/M I DR, = label

Effective Address: M(M(PC+1))

Description: The operand is a memory location that, after assembly, is addressed by a two-byte memory location that itself is addressed by a two-byte literal quantity stored immediately after the instruction opcode. The label defines the two-byte literal quantity used by the Assembler ROM.



LITERAL INDIRECT ADDRESSING

Example: STBI R30, = ADDR Stores the contents of CPU register R30 into the memory location addressed by another memory location which is itself addressed by the two-byte literal quantity specified by the label ADDR.

INDEX MODE

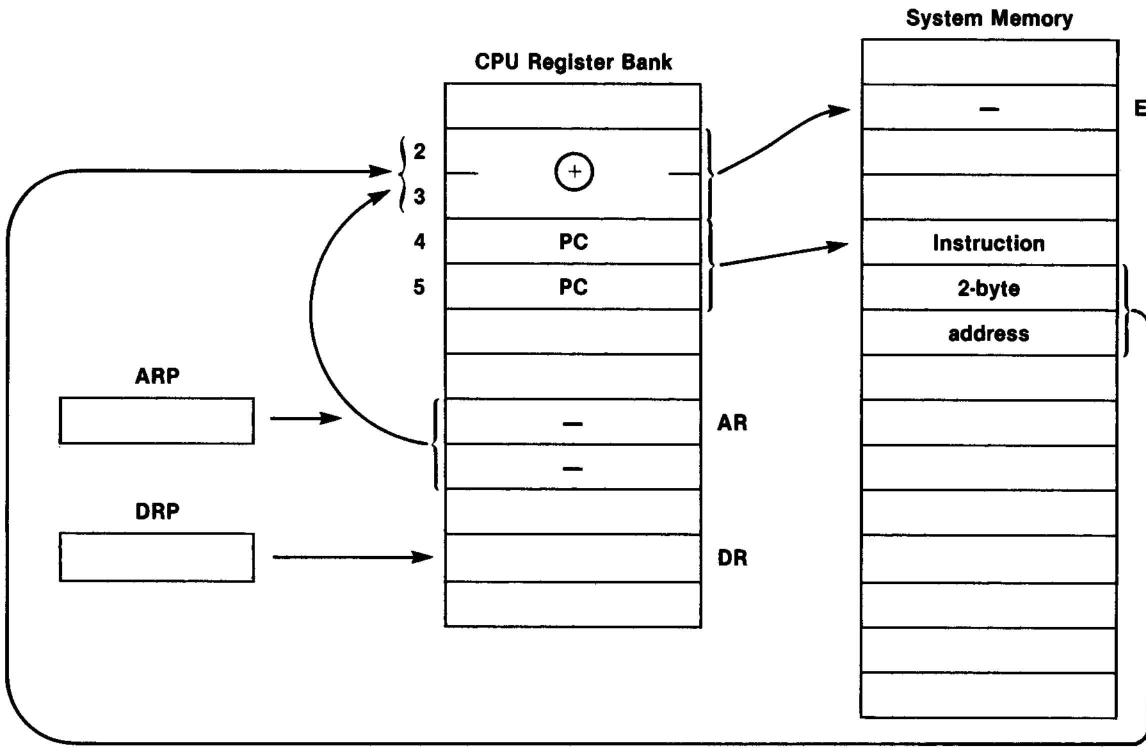
The index mode is the third addressing category. Indexing is useful for accessing data when the data is stored in a table. In indexed addressing, a fixed base address is added to an offset to create the desired address. The CPU performs this addition using CPU registers 2 and 3. After an index instruction, registers 2 and 3 contain the effective address (i.e., the sum of the base and the offset). Neither the original base nor the offset is altered in memory. There are two modes for indexed addressing.

INDEX DIRECT

Format: Opcode B/M D DR, XAR, label

Effective Address: M(AR+(PC+1))

Description: The effective address is found by adding (in binary) the two-byte contents of the AR to the two-byte address that immediately follows the instruction opcode in memory.



INDEXED DIRECT ADDRESSING

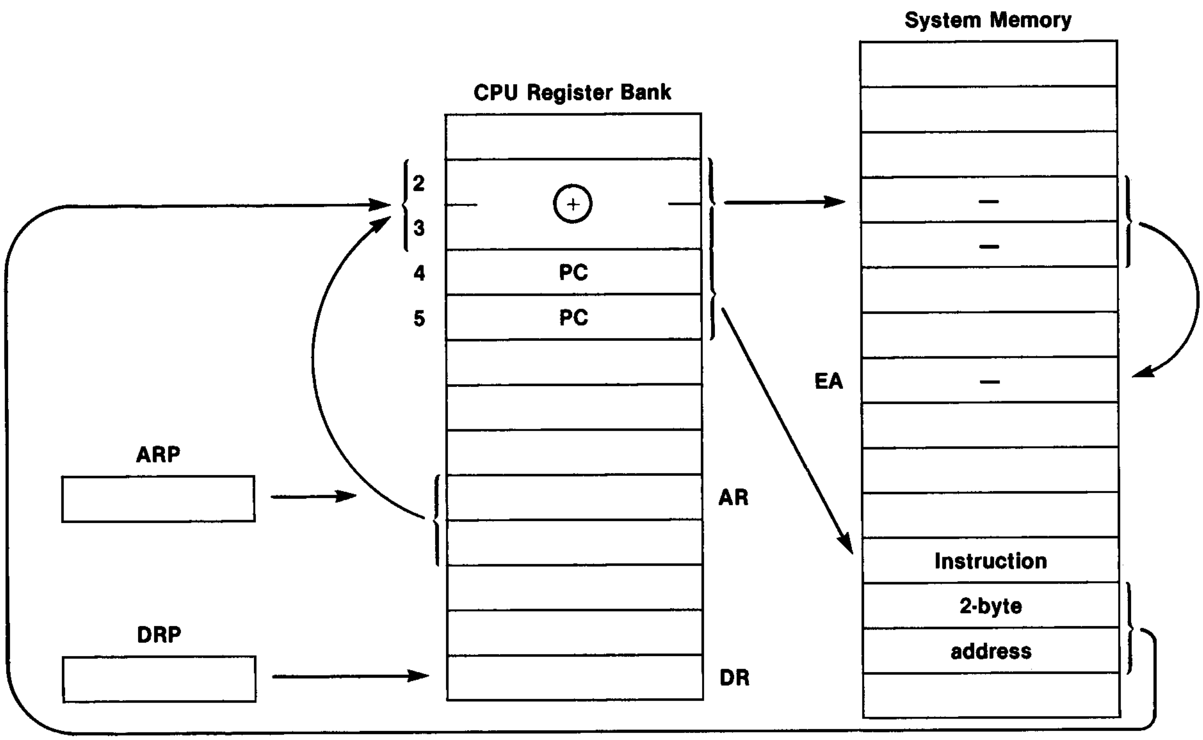
Example: LDBD R36, X30, TABLE Loads into CPU register R36 the contents of the memory location addressed by registers R2 and R3. R2 and R3 contain the sum of the contents of R30 and the contents of the address TABLE.

INDEX INDIRECT

Format: Opcode B/M I DR, XAR, label

Effective Address: M(M(AR+(PC+1)))

Description: The effective address is found in a memory location. This memory location is found by adding (in binary) the two-byte contents of the AR to the two-byte address that immediately follows the instruction opcode in memory. This mode is useful when addresses are stored in table form.



INDEXED INDIRECT ADDRESSING

Example: STMI R36, X30, OFFST Stores the contents of CPU register R36 and R37 in memory, beginning with the location addressed by another memory location which is itself addressed by CPU registers 2 and 3. Registers 2 and 3 contain th sum of the address in R30 plus the offset specified by the label OFFST.

STACK INSTRUCTIONS

There is a large set of instructions that are available to push data onto and pop data from stacks in the main memory of the HP-83/85. These stacks can be addressed by the instructions using direct or indirect addressing.

PU
CPU Instruction

Push

Format: PUB D/I DR +/- AR Push single byte
PUM D/I DR +/- AR Push multi-byte

Description: Pushes single byte or multi-byte onto stack. D/I indicates direct or indirect addressing. +/- indicates stack pointer is incremented (increasing stack) or decremented (decreasing stack) in memory.

Examples:
```
PUBD R32, +R12
PUBI R32, -R46
```

PO
CPU Instruction

Pop

Format: POBD/I DR +/- AR Pop single byte
POMD/I DR +/- AR Pop multi-byte

Description: Pops single byte or multi-byte off stack. D/I indicates direct or indirect addressing. +/- indicates stack pointer is incremented (increasing stack) or decremented (decreasing stack) in memory.

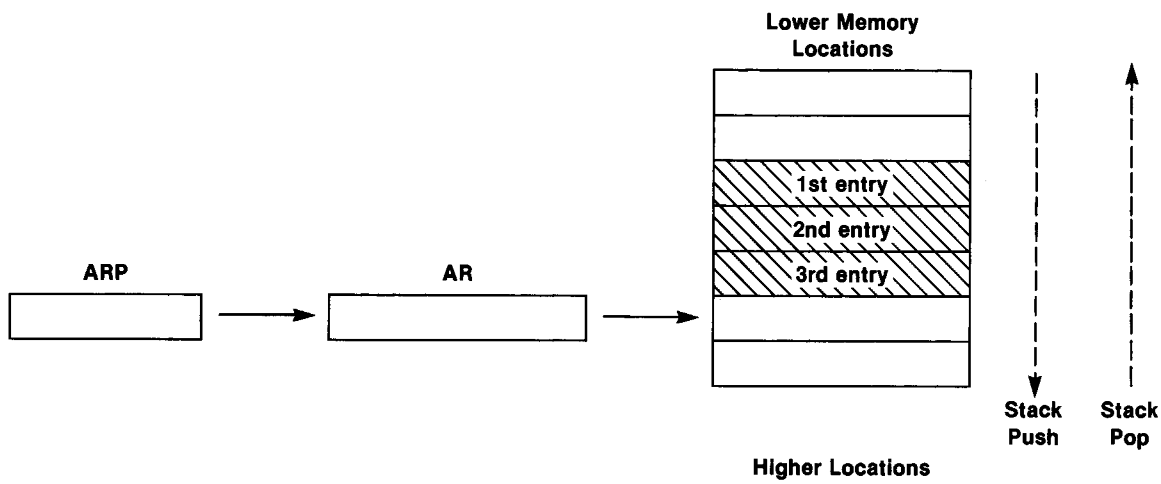
STACK ADDRESSING

CPU registers R6 and R7 are permanently dedicated, and always contain the address of the subroutine return stack. CPU registers R12 and R13 contain, by convention, the address of the operational stack used during runtime by many of the internal HP-85 routines. The user can, of course, address a stack from nearly any CPU register pair.

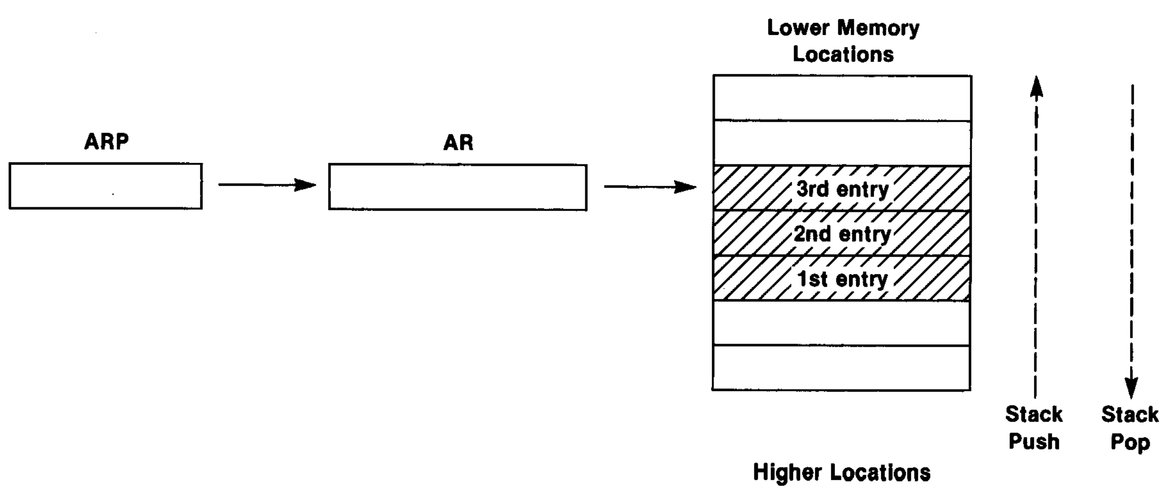
Stacks may be increasing or decreasing. An increasing stack is one which is filled in the direction of higher memory locations and from which data is removed in the direction of lower memory locations. In a decreasing stack, data is pushed in the direction of lower memory locations, and taken off in the direction of higher memory locations. To avoid confusion, it is best to address a particular stack using only instructions for an increasing stack or only instructions for a decreasing stack, but not both.

For stack addressing, the stack pointer is contained in the AR. Multiple stacks are handled by having multiple stack pointers within the CPU register space. A stack is activated by setting ARP equal to the location of that stack's pointer.

For an increasing stack, the AR must point to the next available location on the stack. For a decreasing stack, the AR points to the occupied location on top of that stack.



INCREASING STACK



DECREASING STACK

STACK DIRECT

In this addressing mode, the stack is presumed to contain data. Stores to the stack (pushes) fill the stack. Loads from the stack (pops) empty the stack.

For a push onto an increasing stack, the AR points to the location where data is to be stored. Following the store, the AR is incremented by the number of bytes stored. For a pop operation from an increasing stack, the AR is first decremented by the number of bytes to be popped off. The AR then points to the location of the data to be removed from the stack.

For a pop from a decreasing stack, the AR points to the location of the data to be removed. Following the removal, the AR is incremented by the number of bytes moved. For a push operation onto a decreasing stack, the AR is first decremented by the number of bytes to be stored on the stack. Then the data is pushed onto the stack.

STACK INDIRECT

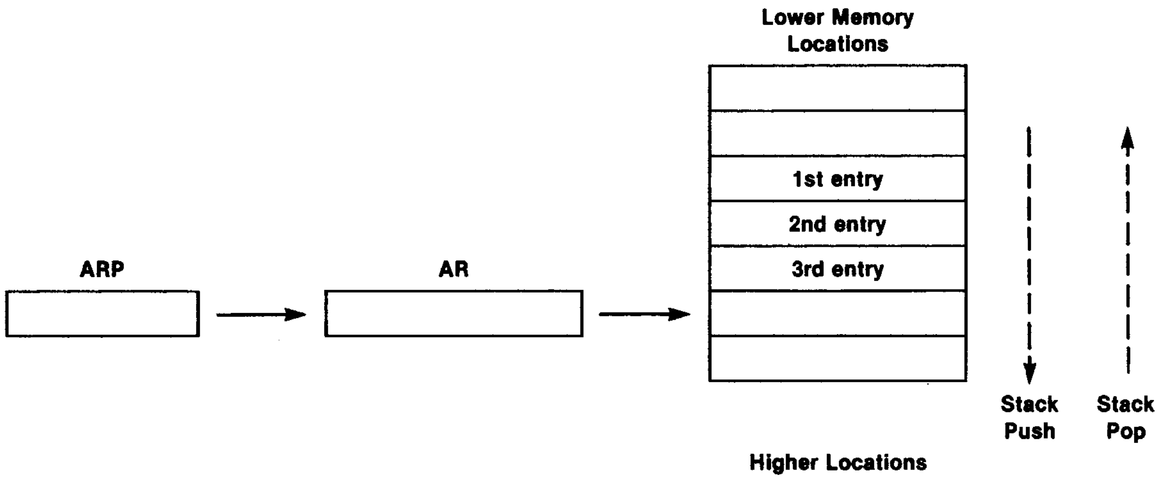
In this addressing mode, the stack is presumed to contain an ordered list of addresses. These addresses point to the location from which data is read by pops or to the location into which data is stored by pushes.

For a push onto an increasing stack, the AR points to the effective address. After storing data in M(EA), the AR is incremented by two. For a pop instruction from an increasing stack, the AR is first decremented by two in order to point to the effective address. M(EA) is then loaded into the CPU register designated by the DRP.

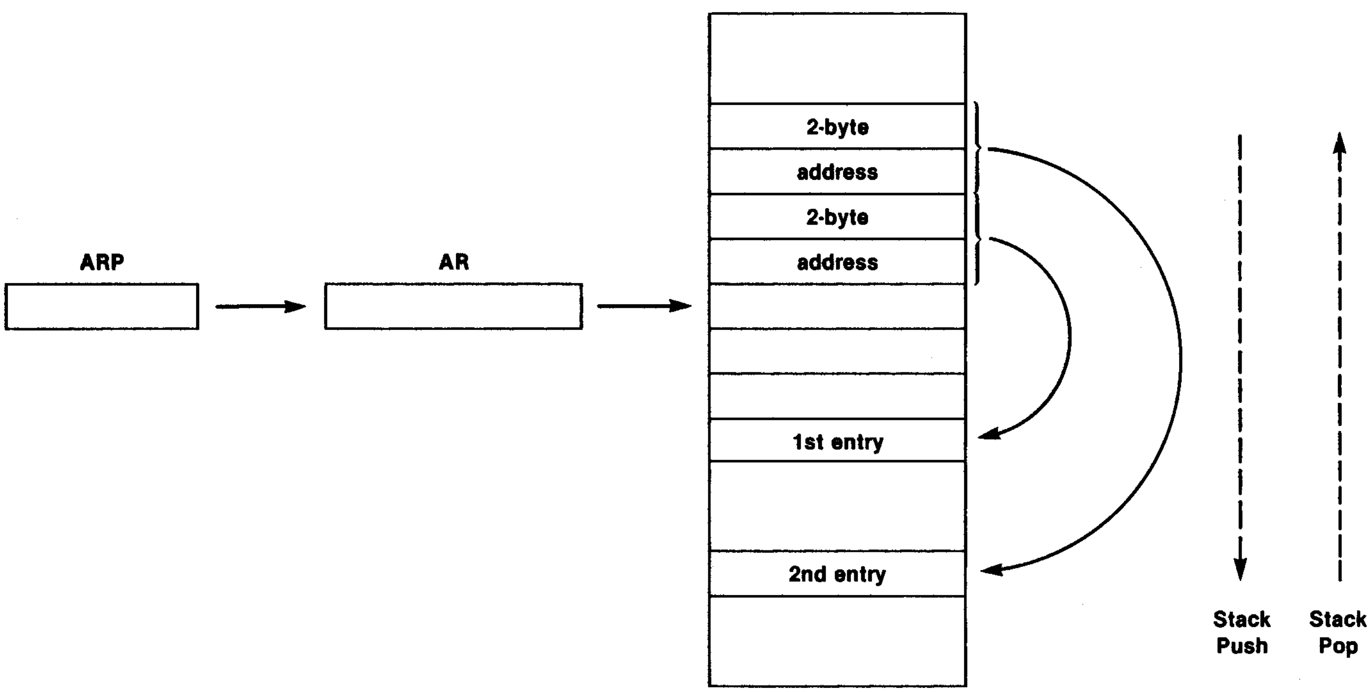
INSTRUCTIONS FOR AN INCREASING STACK

An increasing stack is one which is pushed in the direction of higher addresses (+) and popped in the direction of lower addresses (-).

D (Direct Mode)



I (Indirect Mode)



Each entry can be one or more bytes

INCREASING STACK

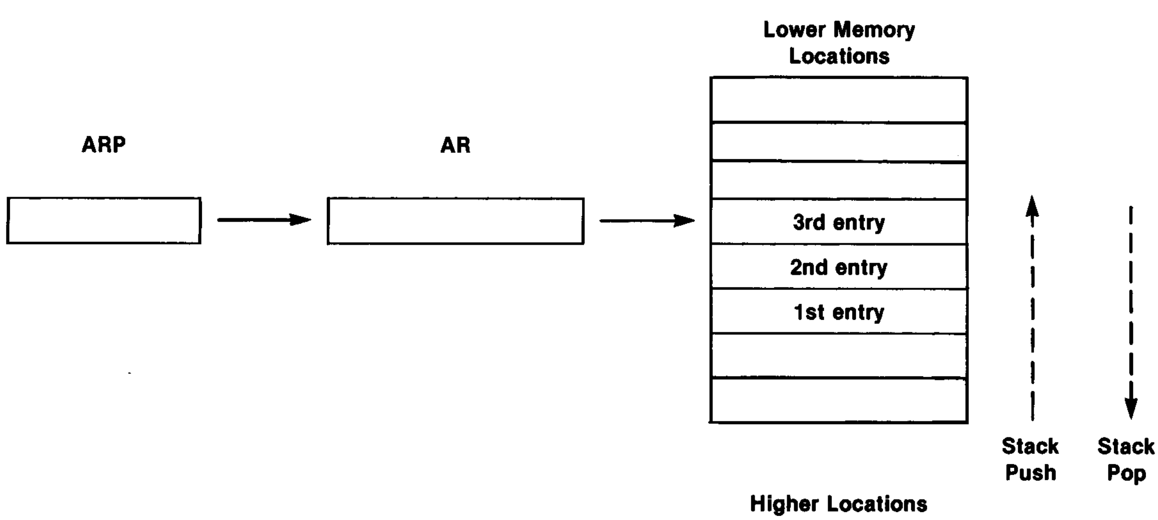
The instructions available for use with an increasing stack are:

PUBD DR, +AR	Push byte direct with increment
PUMD DR, +AR	Push multi-byte direct with increment
PUBI DR, +AR	Push byte indirect with increment
PUMI DR, +AR	Push multi-byte indirect with increment
POBD DR, -AR	Pop byte direct with decrement
POMD DR, -AR	Pop multi-byte direct with decrement
POBI DR, -AR	Pop byte indirect with decrement
POMI DR, -AR	Pop multi-byte indirect with decrement

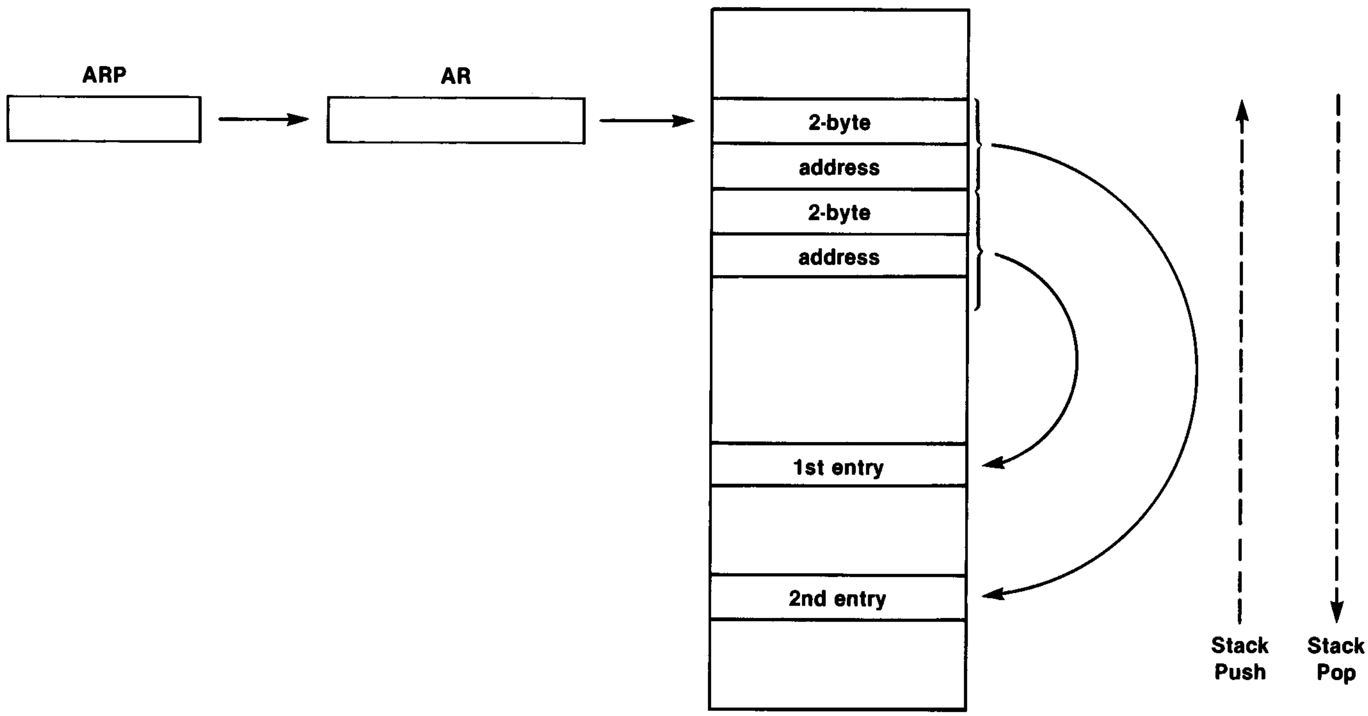
INSTRUCTIONS FOR A DECREASING STACK

A decreasing stack is one which is pushed in the direction of lower addresses (-) and popped in the direction of higher addresses (+).

D (Direct Mode)



I (Indirect Mode)



Each entry can be one or more bytes

DECREASING STACK

The instructions available for use with a decreasing stack are:

PUBD DR, -AR	Push byte direct with decrement
PUMD DR, -AR	Push multi-byte direct with decrement
PUBI DR, -AR	Push byte indirect with decrement
PUMI DR, -AR	Push multi-byte indirect with decrement
POBD DR, +AR	Pop byte direct with increment
POMD DR, +AR	Pop multi-byte direct with increment
POBI DR, +AR	Pop byte indirect with increment
POMI DR, +AR	Pop multi-byte indirect with increment

ARITHMETIC AND LOGICAL INSTRUCTIONS

The arithmetic and logical instructions consist of add, subtract, compare, logical AND and logical OR instructions.

AD

CPU Instruction

Add

Format: ADB [D] DR, operand — Add byte
ADM [D] DR, operand — Add multi-byte

Operation: DR ← DR + operand

Description: Add single or multi-byte. The contents of the effective address determined by the addressing mode are added to the DR. If DCM=1, BCD addition is performed; otherwise, binary addition is performed. The result is stored in the data register.

Examples:

```
ADB R40, R50
ADMD R30,=LABEL
```

ANM

CPU Instruction

Logical AND

Format: ANM [D] DR, operand

Operation: DR ← DR · operand

Description: The DR is loaded with the logical AND of itself and the contents of the effective address determined by the addressing mode used. This instruction is multi-byte only.

Examples:

```
ANM R40, R50
ANMD R32,=LABEL
```

CM

CPU Instruction

Compare

Format: CMB [D] DR, operand — Compare byte
CMM [D] DR, operand — Compare multi-byte

Operation: DR + ten's complement of operand if BCD mode set
DR + two's complement of operand if binary mode set

Description: Compares operand with data register(s). The contents of the effective address determined by the operand and the addressing mode are subtracted from DR. BCD subtraction is performed if DCM=1; otherwise a binary subtraction is performed. The result is used to affect CPU status indicators and is not stored; DR is not affected.

Examples:
```
CMB R24,=377
CMM R22, R32
```

OR

CPU Instruction

Logical OR (Inclusive)

Format: ORB DR, AR — Inclusive OR (single byte)
ORM DR, AR — Inclusive OR (multi-byte)

Operation: DR ← DR v AR

Description: Contents of DR are replaced with inclusive OR of DR and AR. CY and OVF are cleared.

Examples:
```
ORB R21, R41
ORM R40, R70
```

SB

CPU Instruction

Subtract

Format: SBB [D] DR, operand — Subtract byte
SBM [D] DR, operand — Subtract multi-byte

Operation: DR ← DR + ten's complement of operand if BCD mode
DR ← DR + two's complement of operand if binary mode

Description: The contents of the effective address determined by the addressing mode and the operand are subtracted from the contents of the DR. BCD subtraction is performed if DCM=1; otherwise binary subtraction is performed. The result is stored in DR. CY is set if the result is positive, cleared if the result is negative.

Example: SBM R26,=177, 0

XR

CPU Instruction

Logical OR (Exclusive)

Format: XRB DR, AR — Exclusive OR (single byte)
XRM DR, AR — Exclusive OR (multi-byte)

Operation: DR ← DR ⊕ AR

Description: Contents of DR are replaced with the exclusive OR of DR and AR. CY and OVF are cleared.

Example: XRM R40, R50

SHIFT INSTRUCTIONS

All shift instructions can be BCD or binary. The shift instructions consist of logical left, logical right, extended left and extended right instructions; all are available in single byte or multi-byte modes.

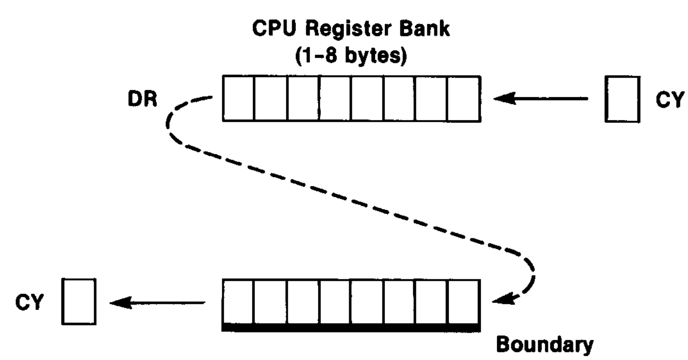
EL

CPU Instruction

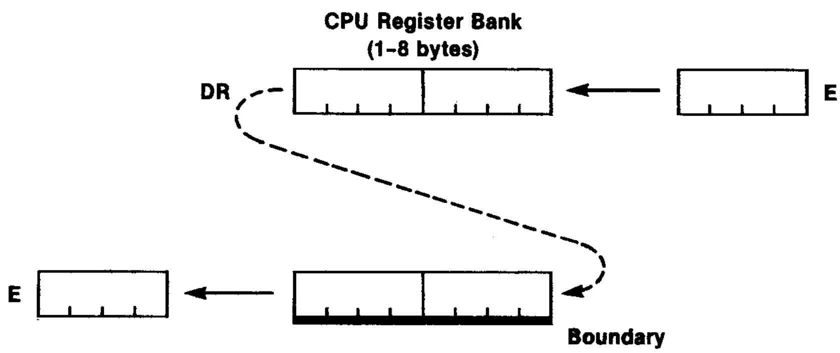
Extended Left Shift

Format: ELB DR   Extended left shift byte
ELM DR   Extended left shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted left one bit position. Carry flag CY is loaded from MSB. LSB is loaded from CY. OVF is set if the shift causes a sign change.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted left one digit position (i.e., four bits) through the E register. CY is cleared.



ER

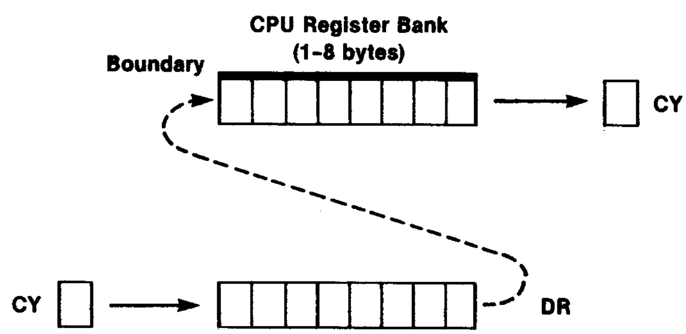
CPU Instruction

Extended Right Shift

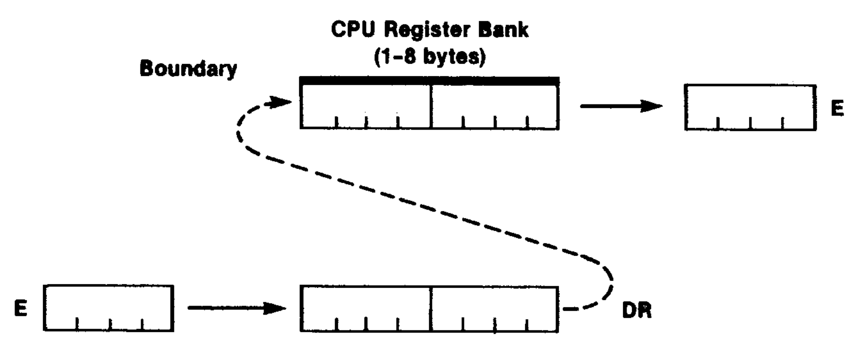
Format: ERB DR Extended right shift byte

ERM DR Extended right shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted right one bit position. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. Carry flag CY is loaded from LSB. MSB is loaded from CY.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted right one digit position (i.e., four bits) through the four-bit E register. CY is cleared.



Notice that a multi-byte right shift instruction, unlike other multi-byte instructions, proceeds from the DR to the preceding (i.e., lower-numbered) boundary.

Example: ERM R47 Shifts all eight bytes of R40 - R47 right.

LR

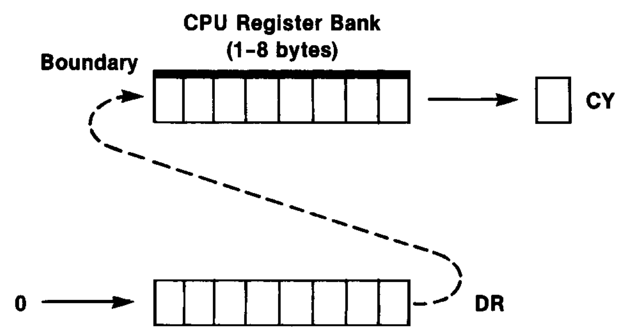
CPU Instruction

Logical Right Shift

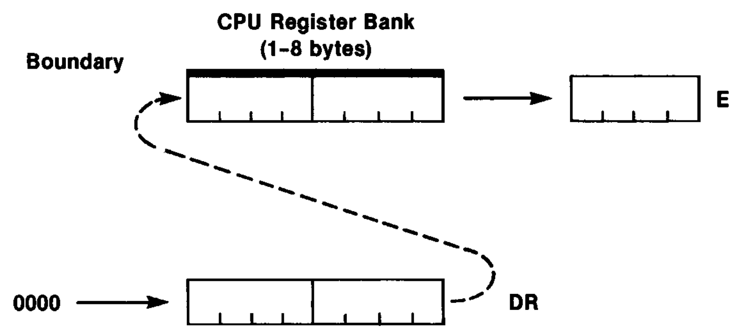
Format: LRB DR Logical right shift byte

LRM DR Logical right shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted right one bit position, and the MSB is cleared. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. Carry flag CY is loaded from LSB.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted right one digit position (i.e., four bits), and the most significant digit is cleared. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. The least significant digit is shifted into the four-bit E register.



Notice that a multi-byte right shift instruction, unlike other multi-byte instructions, proceeds from the DR to the preceding (i.e., lower-numbered) boundary.

Example: LRM R54 Shifts contents of R54, R53, R52, R51, and R50 right.

LL

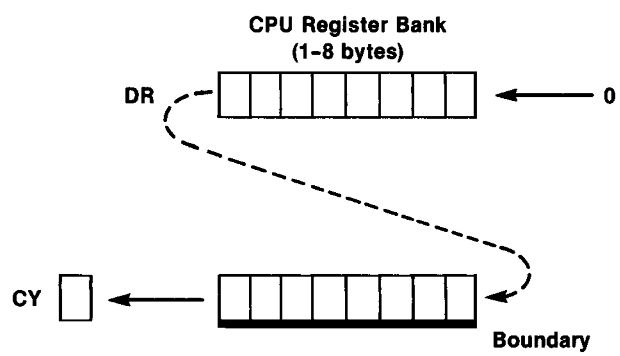
CPU Instruction

Logical Left Shift

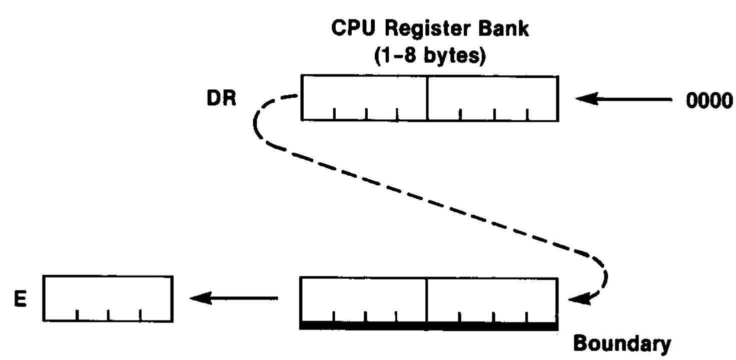
Format: LLB DR Logical left shift byte

LLM DR Logical left shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR are shifted left one bit position, and the LSB is cleared. The bit shifted out of MSB is saved in CY. OVF is set if the shift causes a sign change.



BCD Mode. In BCD mode, the contents of DR are shifted left one digit position (i.e., four bits), and the least significant digit is cleared. The digit shifted out of the most significant digit position is saved in the E register. CY is cleared.



Example: LLM R45 Shifts contents of R45, R46, and R47 left one bit position through CY (in binary mode) or left one digit position through E (in BCD mode).

REGISTER INCREMENT AND DECREMENT INSTRUCTIONS

The increment and decrement instructions for the CPU registers can be BCD or binary.

DC

CPU Instruction

Decrement

Format: DCB DR Decrement byte

DCM DR Decrement multi-byte

Operation: DR ← DR + two's complement of 1 (binary mode)

DR ← DR + ten's complement of 1 (BCD mode)

Description: Binary Mode. In binary mode, DR is decremented by 1 (binary). OVF is set if this operation causes a sign to change to a positive value. CY is set by decrementing a non-zero number.

BCD Mode. In BCD mode, DR is decremented by 1 (decimal). OVF is cleared. CY is set by decrementing a non-zero number.

Example: DCB R12

IC

CPU Instruction

Increment

Format: ICB DR — Increment byte

ICM DR — Increment multi-byte

Operation: DR ← DR + 1

Description: Binary Mode. In binary mode, DR is incremented in binary by 1. OVF is set if this operation causes a sign change to a negative value.

BCD Mode. In BCD mode, DR is incremented in decimal by 1. OVF is cleared.

Example: ICM R40

COMPLEMENT INSTRUCTIONS

The complement instructions can be BCD or binary.

NC

CPU Instruction

Nine's (Or One's) Complement

Format: NCB DR — Nine's (or one's) complement byte
NCM DR — Nine's (or one's) complement multi-byte

Operation: DR ← $\overline{DR}$

Description: Binary Mode. In binary mode, the one's complement of the contents of DR replace the contents of DR. CY and OVF are cleared.

BCD Mode. In BCD mode, the nine's complement of the contents of DR replace the contents of DR. CY and OVF are cleared.

Example: NCB R30

TC

CPU Instruction

Ten's (Or Two's) Complement

Format: TCB DR — Ten's (or two's) complement byte

TCM DR — Ten's (or two's) complement multi-byte

Operation: $DR \leftarrow \overline{DR} + 1$

Description: Binary Mode. In binary mode, the two's complement of the contents of DR replaces the contents of DR. CY is set if the contents of DR were zero. OVF is set if contents of DR were 100...000.

BCD Mode. In BCD mode, the contents of DR are replaced with their ten's complement. CY is set if the contents of DR were zero. OVF is cleared.

Example: TCM R50

TEST INSTRUCTION

The test instruction can check the status of single-byte or multi-byte CPU registers.

TS

CPU Instruction

Test

Format: TSB DR Test byte
TSM DR Test multi-byte

Description: The contents of DR are tested and condition flags are set accordingly. CY and OVF are cleared.

Example: TSM R36

REGISTER CLEAR INSTRUCTION

The clear instruction permits the clearing of any byte or of any multi-byte portion of the CPU register bank.

CL

CPU Instruction

Clear

Format: CLB DR Clear byte
CLM DR Clear multi-byte

Operation: DR ← 0

Description: DR is cleared. CY and OVF are cleared.

Example: CLB R47

SUBROUTINE JUMP INSTRUCTION

The subroutine jump instruction is available in the literal direct or the indexed addressing mode.

JSB

CPU Instruction

Jump to Subroutine

Format: JSB = label — Jump subroutine literal direct

JSB XR, label — Jump subroutine indexed

Operation: Literal Direct. M(SP) ← PC+3, SP ← SP+2, PC ← M(PC+1)

Indexed. M(SP) ← PC+3, SP ← SP+2, PC ← AR + M(PC+1)

Description: The PC is saved in the memory location addressed by the R6 stack pointer. Program control is then transferred to the location defined by the label. In indexed addressing, control is transferred to the location defined by the two-byte contents of the address register plus the label.

After a subroutine jump, the next RTN instruction executed causes a return to the instruction after the JSB.

Examples:

```
JSB = LOC1
JSB X32, LOC2
```

Note: Since an indexed subroutine jump (i.e., JSB XR, label) can cause a jump to an unlabeled destination, the programmer must ensure that the ARP and DRP are set to ensure proper operation at the destination. See Handling of ARP and DRP During Assembly later in this section.

CONDITIONAL JUMP INSTRUCTION

The conditional jump instruction can alter execution based on 16 different conditions in the CPU.

J

CPU Instruction

Conditional Jump

Format:

JMP label	Unconditional jump	
JNO label	Jump on no overflow	
JOD label	Jump on odd	
JEV label	Jump on even	
JPS label	Jump on positive	Takes overflow into consideration. (Exclusive OR of MSB and OVF.)
JNG label	Jump on negative	Takes overflow into consideration. (Exclusive OR of MSB and OVF.)
JZR label	Jump on zero	
JNZ label	Jump on non-zero	
JEZ label	Jump on E zero	
JEN label	Jump on E non-zero	
JCY label	Jump on carry	
JNC label	Jump on no carry	
JLZ label	Jump on left digit zero	
JLN label	Jump on left digit non-zero	
JRZ label	Jump on right digit zero	
JRN label	Jump on right digit non-zero	

Description: This group of instructions gives the capability of branching as a function of status conditions previously generated. The branching capability uses relative addressing. If the status condition interrogated is found to be true, then the relative branch to the address of the label will be taken. Otherwise, the next instructions after the jump will be executed.

Each jump instruction is assembled into two bytes: An opcode, and an offset in two's complement notation.

A jump can cover 400_8 destinations from 200_8 before the next instruction to 177_8 after the next instruction. The address to which the jump is made is the sum of the address of the jump instruction plus the offset plus two.

Example: JMP INITAL When assembled, this instruction would appear as shown below.

```
200
 .
 .
 .
375    n       ---             Offset = -3
376    n + 1 JMP               Offset = -2
377    n + 2 Offset            Offset = -1  (Current byte)
  0    n + 3 ---               Offset = 0   (Next byte)
  1    n + 4 ---               Offset = +1
  2    n + 5 ---               Offset = +2
 .
 .
 .
177
```

ARP AND DRP LOAD INSTRUCTIONS

Two instructions are available for loading the address register pointer or the data register pointer. These instructions are not normally needed because the assembler automatically generates necessary ARPs and DRPs where required.

ARP

CPU Instruction

Load ARP

Format: ARP AR

Operation: ARP

Description: Sets address register pointer to point to address register.

Example: ARP R25 Sets ARP to point to R25.

DRP

CPU Instruction

Load DRP

Format: DRP DR

Operation: DRP

Description: Sets data register pointer to point to data register.

Example: DRP R25 Sets DRP to R25.

NOTE

The instructions to load DRP indirectly with RØ and to load ARP indirectly with RØ are:

DRP 1
ARP 1

Thus, to avoid confusion, R1 is not allowed in either the DR or AR fields. This means that CPU register R1 is for all practical purposes inaccessible except by means of a multi-byte RØ operation or when RØ = 1 and the ARP or DRP is specified by R*. See Using R* later in this section.

OTHER INSTRUCTIONS

In addition to the instructions above, there are a few other instructions which the programmer can use to manipulate quantities in the CPU and memory.

BCD

CPU Instruction

Set Decimal Mode

Format: DCM

Operation: DCM ← 1

Description: Sets DCM to 1 so that arithmetic operations will be in binary-coded decimal.

BIN

CPU Instruction

Set Binary Mode

Format: BIN

Operation: DCM ← 0

Description: Sets DCM to zero so arithmetic operations performed will be in binary.

CLE

CPU Instruction

Clear E

Format: CLE

Operation: E ← 0

Description: All four bits of the E (extend) register are cleared to zero.

DCE

CPU Instruction

Decrement E

Format: DCE

Operation: $E \leftarrow E - 1$

Description: E (extend) register decremented by 1. This instruction is always a binary operation, regardless of the setting of the DCM status flag.

ICE

CPU Instruction

Increment E

Format: ICE

Operation: $E \leftarrow E + 1$

Description: E (extend) register incremented by 1. This instruction is always a binary operation, regardless of the setting of the DCM status flag.

PAD

CPU Instruction

Pop ARP, DRP and Status

Format: PAD

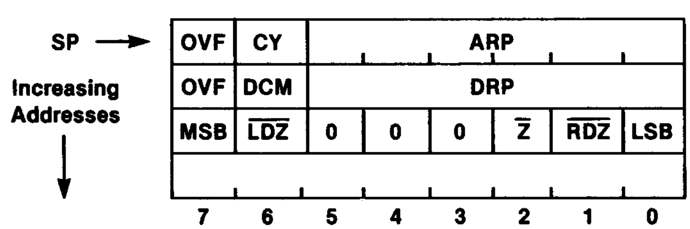
Operation: M(SP) → ARP, DRP and all status flags except E.

Description: Restore ARP, DRP and status (usually after a PAD instruction) by popping them off the stack.

Stack pointer is decremented by 3, and all status flags except E are altered by the contents of the three stack locations that are read.

The first byte processed is read as LSB in bit 0, $\overline{RDZ}$ in bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6 and MSB in bit 7. The second byte is read as DRP in bits 0-5, DCM status in bit 6, and overflow flags in bit 7. The third byte is read as ARP in bits 0-5, carry flag in bit 6, and overflow flag in bit 7.

Following a PAD instruction, the stack has been read as shown here:



RTN

CPU Instruction

Return From Subroutine

Format: RTN

Operation: $SP \leftarrow SP - 2$, $PC \leftarrow M(SP)$

Description: Subroutine return stack pointer is decremented by two. Then the return address is read from the stack and written into the program counter.

SAD

CPU Instruction

Save ARP, DRP and Status

Format: SAD

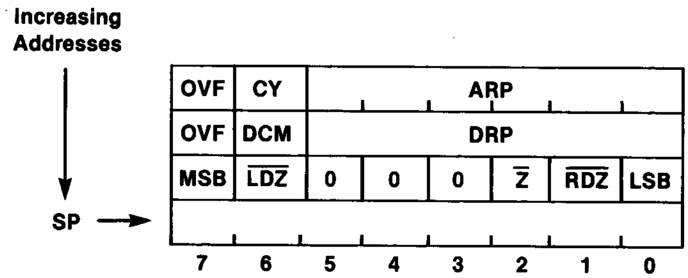
Operation: $M(SP) \leftarrow ARP$, and all status flags except E.

Description: Saves ARP, DRP and status (except E) in memory locations addressed by SP (stack pointer).

Three bytes are pushed onto the stack. The first byte contains ARP in bits 0-5, CY in bit 6, and the overflow flag in bit 7. The second byte contains DRP in bits 0-5, DCM status in bit 6, and the overflow flag in bit 7. The third byte contains LSB in bit 0, RDZ in bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6, and MSB in bit 7.

SP is then incremented by three. Status is not affected by this operation.

Following a SAD instruction, the stack contents are as shown here:



USE OF R*

When entering source code, the programmer may substitute R* for the AR or DR in any CPU instruction. R* causes the ARP or DRP to be loaded with the least significant six bits of CPU register RØ. The effect is that the DR and AR are specified by the contents of RØ.

Example:	LDB RØ, = 26	Loads RØ with 26.
	LDB R*, R30	Loads CPU register specified by RØ (i.e., R26 now) with contents of R30.
	STB R40, R*	Stores contents of R40 into register (R26 now) specified by RØ.

ASSEMBLY OF CPU INSTRUCTIONS

When the address field of an instruction consists of a DR and an AR, each source statement is usually assembled into three bytes of machine code. These bytes are assembled in order as:

1. DRP: DRP set to point to DR.
2. ARP: ARP set to point to AR.
3. Opcode: Perform operation.

Thus, a stack push instruction such as PUBD would be assembled and appear as shown here:

Byte No.	Machine Code	Source Code
000227	110 006 342	PUBD R10, -R6

When the address field of an instruction consists of a DR and a label, as in the case of literal direct and literal indirect addressing (e.g., LDMI R32, = ADDRS), each source statement is usually assembled into four bytes of machine code:

1. DRP: DRP set to point to DR.
2. Opcode: Perform operation.
3. Low-order byte of literal quantity.
4. High-order byte of literal quantity.

When the address field of an instruction consists of DR, AR, and a label, as in the case of indexed direct and indexed indirect addressing (e.g., LDBI R36, X32, TABLE), five bytes of machine code may be generated for each source statement:

1. DRP: DRP set to point to DR.
2. ARP: ARP set to point to AR.
3. Opcode: Perform operation.
4. Low-order byte of address.
5. High-order byte of address.

HANDLING OF ARP AND DRP DURING ASSEMBLY

An optimizing feature of the Assembler ROM is the deletion of "unnecessary" ARP and DRP instructions during assembly.

If an instruction is not labeled (i.e., there is not an entry in the label field) and the ARP (and/or DRP) is already set to the correct value, the previously-set ARP (and/or DRP) is not generated during assembly.

For example:

Byte No.	Machine Code	Source Code
000227	110 006 342	LABEL POBD R10, -R6
000232	342	POBD R10, -R6

In this example, both the ARP and the DRP are specified beginning with byte 227. Since they are now correctly set for the next instruction, they are automatically deleted when the second POBD R10, -R6 instruction is assembled. This results in the machine code shown in byte 232.

Not all previously-set ARPs and DRPs are deleted during assembly. Instances where a previously-set ARP and/or DRP will not be deleted include:

--Labeled instructions. Since a jump from anyplace in code may cause execution to resume at the label, the first ARP and DRP are not deleted after an instruction that contains an entry in the label field.

--Returns. After executing a JSB, then returning, the first ARP and DRP encountered are not deleted.

--PAD. Following a PAD instruction, the first ARP and DRP are not deleted.

USING R#

When entering CPU instructions, the user may substitute R# in almost any instruction requiring an AR or DR. R# causes the ARP or DRP to be deleted from the machine code, regardless of other conditions. For example:

Byte No.	Machine Code	Source Code
000265	240	LABEL LDB R#, R#

R# is normally used after labels, when the ARP and DRP are already set correctly. By using R#, it is not necessary to squander time or bytes resetting ARP and DRP.

PSEUDO-INSTRUCTIONS

Pseudo-instructions are instructions to the assembler. Each may be entered by typing a pseudo-opcode in the same field as the opcode for an instruction, followed by any additional required operand.

Pseudo-instructions perform three main functions when encountered during assembly:

--Assembly control
--Data definition
--Conditional Assembly

PSEUDO-INSTRUCTIONS FOR ASSEMBLY CONTROL

ABS

Pseudo-Instruction

Absolute Program

Format: ABS 16

ABS 32

ABS ROM base address

Description: Declares an absolute program (i.e., with addresses that cannot be relocated), for either a computer with 16K bytes of memory, a computer with 32K bytes, or for a ROM beginning with the specified base address. If ABS 16 or ABS 32 is declared, the instruction must precede a NAM instruction.

FIN

Pseudo-Instruction

Finish Program

Format: FIN

Description: Signifies the end of the source code. This pseudo-instruction is required for assembly.

GLO

Pseudo-Instruction

Declare Global File

Format: GLO

GLO file name

Description: If no file name, declares this source code to be a global file. Otherwise, declares the global file to be used in the assembling of the current source code. Comments are not allowed on the same line as the GLO instruction, and the instruction must precede ABS and NAM.

LNK

Pseudo-Instruction

Link Files At Assembly

Format: LNK file name

Description: Will load another file containing more source code and continue assembling. Allows assembly of larger programs than would otherwise be possible.

Example: LNK SOURC2 When this instruction is encountered during assembly, the assembler looks for the file SOURC2 on the current mass storage device, loads the file, and continues assembling using the source code from the file.

LST

Pseudo-Instruction

List

Format: LST

Description: Causes the code to be listed on the current PRINTER IS device at assembly time. If the column width of the printer is sufficient (>46 characters) the listing will contain both the object and source code; otherwise, only the object code will be listed.

An address that is undefined when its label is encountered will be printed in object code as 326, 336, or 377, depending upon whether it is a DEF, a relative jump, or a GTO statement.

NAM
Name Program

Pseudo-Instruction

Format: NAM unquoted string

Description: Sets up the PCB (Program Control Block) for a binary program. Should be preceded only by GLO, ABS, LST, UNL, DAD, EQU, or comments. Illegal when ABS ROM has been declared.

Example: NAM KEYHIT Names a binary program KEYHIT and sets up the 32_8-byte program control block for that program.

ORG
Origin

Pseudo-Instruction

Format: ORG address

Description: Specifies a base address which is added to all following defined addresses (DAD's). This pseudo-instruction is most useful in global files.

UNL
Unlist

Pseudo-Instruction

Format: UNL

Description: Turns off the list feature which was turned on by the LST pseudo-instruction. After an UNL, code is not listed during assembly.

PSEUDO-INSTRUCTIONS FOR DATA DEFINITION

ASC
ASCII

Pseudo-Instruction

Format: ASC numeric value, unquoted string
ASC quoted string

Description: Inserts into the object code the ASCII code for the number of characters specified of the unquoted string. Inserts the entire quoted string.

Example:
ASC 3, FTOC   Inserts the ASCII code for FTO.
ASC 4, FTOC   Inserts the ASCII code for FTOC.
ASC "LOCATION"   Inserts the ASCII code for LOCATION.

ASP
ASCII With Parity

Pseudo-Instruction

Format: ASP numeric value, unquoted string
ASP quoted string

Description: Same as ASC except that the parity bit (MSB) of the string's final character is set. (During operation, the HP-83/85 system determines the end of an ASCII string in some system tables by checking to see if the character's parity bit is set. When the bit is found set, the system assumes the next character begins a new string or entry in the table.)

BSZ

Pseudo-Instruction

Bytes To Zero

Format: BSZ numeric value

Description: Inserts into the object code the octal number of bytes of zeros specified by the numeric value.

Example: BSZ 30 Fills 30_8 bytes with zeros.

BYT

Pseudo-Instruction

Bytes To Values

Format: BYT numeric value [,numeric value...]

Description: Inserts literal values into the object code.

Examples: BYT 377 Inserts octal 377 (i.e., all ones) into object code.

BYT 20,55C Inserts octal 20 into this byte of object code and BCD 55 into next byte.

DAD

Pseudo-Instruction

Direct Address

Format: Label DAD address

Description: Assigns either an absolute address or a constant to a label. DAD and EQU are similar; DAD is usually used for addresses, while EQU is used for values other than addresses. ORG affects only DAD's.

Example: INTORL DAD 56343 Assigns absolute address 56343 to the label INTORL.

DEF

Pseudo-Instruction

Define Label Address

Format: DEF label

Description: Inserts the two-byte address associated with the label.

Example: DEF RUNTIM Inserts two-byte address of the label RUNTIM.

EQU

Pseudo-Instruction

Equals

Format: Label EQU numeric value

Description: Assigns either an absolute address or a constant to a label. DAD and EQU are similar; DAD is usually used for addresses, while EQU is used for values other than addresses. ORG affects only DAD's.

GTO
Go To

Pseudo-Instruction

Format: GTO label

Description: Generates four bytes of object code which load the program counter (CPU registers 4 and 5) with the address minus one (i.e., ADR-1) of the label. The label must be for an absolute address.

The CPU relative jump instructions (JRZ, JNO, etc.) can cause jumps of from 177_8 to -200_8 memory locations. The GTO pseudo-instruction is useful for jumping beyond the range of relative jumps.

WARNING

The GTO pseudo-instruction is primarily for use in ROMs. It should not be used in a binary program unless that program has been declared an absolute program.

Example: GTO INTORL

VAL
Value

Pseudo-Instruction

Format: VAL label

Description: Inserts the one-byte literal octal value associated with the label.

Example: PPROM# EQU 360

VAL PPROM# Inserts the one-byte literal octal value (360) of the label PPROM# into the object code.

PSEUDO-INSTRUCTIONS FOR CONDITIONAL ASSEMBLY

This set of pseudo-instructions permits the user to control assembly by means of conditional assembly flags. A conditional assembly flag has the same constraints as a label--it can be no more than six characters in length, and the first character cannot be a digit.

A conditional assembly flag is treated the same as a label by the HP-83/85 system. (For example, an assembly flag can be located by a label search.) For this reason, a conditional assembly flag name should be unique, and should not duplicate a label.

AIF

Pseudo-Instruction

Assemble If Flag True

Format: AIF assembly flag name

Description: Tests the specified conditional assembly flag and, if true, continues to assemble the following code. If the flag tests false, the source code after the flag is treated as if it were a series of comments until an EIF instruction is found.

Example: AIF CYCLE Tests assembly flag CYCLE.

CLR

Pseudo-Instruction

Clear Flag

Format: CLR flag name

Description: Clears the specified conditional assembly flag to the false state.

Example: CLR CYCLE Clears assembly flag CYCLE.

EIF

Pseudo-Instruction

End Of Conditional Assembly

Format: EIF

Description: Terminates any conditional assembly in process. Only one conditional assembly can be handled at a time. If a second one is encountered while the first is still active, the second will override the first.

SET

Pseudo-Instruction

Set Flag

Format: SET flag name

Description: Sets the specified conditional assembly flag to the true state.

Example: SET CYCLE Sets conditional assembly flag CYCLE.

Appendix C

ASSEMBLER INSTRUCTION SET

On the following pages is a list of all CPU instructions available on the Assembler ROM.

LEGEND

DR	Data register. Can be register number (e.g., R32), R* or R#.
AR	Address register. Can be register number (e.g., R32), R* or R#.
<u>Literal</u>	Literal value, up to 10_8 bytes in length. Can be BCD constant (e.g., 99C), octal constant (e.g., 12), or decimal constant (e.g., 20D). Can also be specified by a label, where the literal quantity is a one- or two-byte value or address assigned to the label.
<u>Label</u>	Address of literal quantity. Label name must begin with an alphabetic character, can use any combination of alphanumeric characters, and can be 1-6 characters in length.
Clock Cycle	1.6 μsec.
B	Number of bytes.
T	Add one clock cycle if true (i.e., the jump occurs).
R(x)	CPU register addressed by (x).
M(x)	Memory location addressed by (x). (x) must be a 16-bit address.
PC	Program Counter. CPU registers R4 and R5. Used to address the instruction being executed.

SP	Subroutine Stack Pointer. CPU registers R6 and R7. Used to point to the next available location on the subroutine return address stack.
EA	Effective Address. The location from which data is read for load-type instructions or the location where data is placed for store-type instructions.
ADR	Address. The two-byte quantity directly following an instruction that uses the literal direct, literal indirect, index direct or index indirect addressing mode. This quantity is always an address.
n	Literal value.
←	Is transferred to.
( )	Contents of.
$\overline{}$	Complement (e.g., $\overline{x}$ is complement of x). This is one's complement if DCM=0 and nine's complement if DCM=1.
•	Logical AND.
V	Inclusive OR.
⊕	Exclusive OR.
JIF	Jump if.
1	Status bit is set.
Ø	Status bit is cleared.
X	Status bit is affected.

-	Status bit is not affected.
Y	This option is available to this instruction.

The complete list of CPU instructions begins on the next page.

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	Status MSB	Status RDZ LDZ	Status Z	Status DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
ADB DR, AR	Add byte	Reg. imm.	302	5	DR←DR+AR	X	X	X	X	-	-	X	X	-	X	0	Y
ADB DR, = literal	Add byte	Lit. imm.	312	5	DR←DR+M(PC+1)	X	X	X	X	-	-	X	X	-	X	0	Y
ADBD DR, AR	Add byte	Reg. dir.	332	6	DR←DR+M(AR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADBD DR, = label	Add byte	Lit. dir.	322	5	DR←DR+M(ADR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADM DR, AR	Add multi-byte	Reg. imm.	303	4+B	DR←DR+AR	X	X	X	X	-	-	X	X	-	X	0	Y
ADM DR, = literal	Add multi-byte	Lit. imm.	313	4+B	DR←DR+M(PC+1)	X	X	X	X	-	-	X	X	-	X	0	Y
ADMD DR, AR	Add multi-byte	Reg. dir.	333	5+B	DR←DR+M(AR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADMD DR, = label	Add multi-byte	Lit. dir.	323	4+B	DR←DR+M(ADR)	X	X	X	X	-	-	X	X	-	X	0	Y
ANM DR, AR	Logical AND (multi-byte)	Reg. imm.	307	4+B	DR←DR·AR	X	X	X	X	-	-	0	0	-	0	0	
ANM DR, = literal	Logical AND (multi-byte)	Lit. imm.	317	4+B	DR←DR·M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
ANMD DR, AR	Logical AND (multi-byte)	Reg. Dir.	337	5+B	DR←DR·M(AR)	X	X	X	X	-	-	0	0	-	0	0	
ANMD DR, = literal	Logical AND (multi-byte)	Lit. dir.	327	5+B	DR←DR·M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
ARP AR	Load ARP		000-077 (≠001)	2	ARP←n	-	-	-	-	-	-	-	-	-	-	-	
ARP *	Load ARP with contents of RØ		001	3	ARP←RØ	-	-	-	-	-	-	-	-	-	-	-	
BCD	Set BCD mode		231	4	DCM←1	-	-	-	-	1	-	-	-	-	-	-	
BIN	Set binary mode		230	4	DCM←0	-	-	-	-	0	-	-	-	-	-	-	
CLB DR	Clear byte	Reg. imm.	222	5	DR←0	X	X	X	X	-	-	0	0	-	0	0	
CLM DR	Clear multi-byte	Reg. imm.	223	4+B	DR←0	X	X	X	X	-	-	0	0	-	0	0	
CLE	Clear E		235	2	E←0	-	-	-	-	-	0	-	-	0	-	-	
CMB DR, AR	Compare byte	Reg. imm.	300	5	DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	Status MSB	Status RDZ LDZ	Status Z	Status DCM	DCM=0 E	DCM=0 CY	DCM=0 OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
CMB DR, = literal	Compare byte	Lit. imm.	310	5	$DR+\overline{M(PC+1)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMBD DR, AR	Compare byte	Reg. dir.	330	6	$DR+\overline{M(AR)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMBD DR, = label	Compare byte	Lit. dir.	320	6	$DR+\overline{M(ADR)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMM DR, AR	Compare multi-byte	Reg. imm.	301	4+B	$DR+\overline{AR}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMM DR, = literal	Compare multi-byte	Lit. imm.	311	4+B	$DR+\overline{M(PC+1)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMMD DR, AR	Compare multi-byte	Reg. dir.	331	5+B	$DR+\overline{M(AR)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
CMMD DR, = label	Compare multi-byte	Lit. dir.	321	5+B	$DR+\overline{M(ADR)}+1$	X	X	X	X	-	-	X	X	-	X	0	Y
DCB DR	Decrement byte	Reg. imm.	212	5	DR←DR-1	X	X	X	X	-	-	X	X	-	X	0	Y
DCM DR	Decrement multi-byte	Reg. imm.	213	4+B	DR←DR-1	X	X	X	X	-	-	X	X	-	X	0	Y
DCE	Decrement E		233	2	E←E-1	-	-	-	-	-	X	-	-	X	-	-	
DRP DR	Load DRP		100-177 (≠101)	2	DRP←n	-	-	-	-	-	-	-	-	-	-	-	
DRP 1	Load DRP with contents of R0		101	3	DRP←R0	-	-	-	-	-	-	-	-	-	-	-	
ELB DR	Extended left byte	Reg. imm.	200	5	Circulate DR left once	X	X	X	X	-	-	X	X	X	0	0	Y
ELM DR	Extended left multi-byte	Reg. imm.	201	4+B	Circulate DR left once	X	X	X	X	-	-	X	X	X	0	0	Y
ERB DR	Extended right byte	Reg. imm.	202	5	Circulate DR right once	X	X	X	X	-	-	X	0	X	0	0	Y
ERM DR	Extended right multi-byte	Reg. imm.	203	4+B	Circulate DR right once	X	X	X	X	-	-	X	0	X	0	0	Y
ICB DR	Increment byte	Reg. imm.	210	5	DR←DR+1	X	X	X	X	-	-	X	X	-	X	0	Y
ICM DR	Increment multi-byte	Reg. imm.	211	4+B	DR←DR+1	X	X	X	X	-	-	X	X	-	X	0	Y

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status: LSB	Status: MSB	Status: RDZ LDZ	Status: Z	Status: DCM	Status: DCM=Ø E	Status: DCM=Ø CY	Status: DCM=Ø OVF	Status: DCM=1 E	Status: DCM=1 CY	Status: DCM=1 OVF	Binary/ BCD Option
ICE	Increment E		234	2	E←E+1	-	-	-	-	-	X	-	-	X	-	-	
JCY label	Jump on carry		373	4+T	JIF←CY=1	-	-	-	-	-	-	-	-	-	-	-	
JEN label	Jump on E non-zero		370	4+T	JIF E≠0000	-	-	-	-	-	-	-	-	-	-	-	
JEV label	Jump on even		363	4+T	JIF LSB=0	-	-	-	-	-	-	-	-	-	-	-	
JEZ label	Jump on E zero		371	4+T	JIF E=0000	-	-	-	-	-	-	-	-	-	-	-	
JLN label	Jump on left digit non-zero		375	4+T	JIF LDZ≠1	-	-	-	-	-	-	-	-	-	-	-	
JLZ label	Jump on left digit zero		374	4+T	JIF LDZ=1	-	-	-	-	-	-	-	-	-	-	-	
JMP label	Unconditional jump		360	4+T	Jump always	-	-	-	-	-	-	-	-	-	-	-	
JNC label	Jump on no carry		372	4+T	JIF CY=0	-	-	-	-	-	-	-	-	-	-	-	
JNG label	Jump on negative		364	4+T	JIF MSB≠OVF	-	-	-	-	-	-	-	-	-	-	-	
JNO label	Jump on no overflow		361	4+T	JIF OVF=0	-	-	-	-	-	-	-	-	-	-	-	
JNZ label	Jump on non-zero		366	4+T	JIF Z≠1	-	-	-	-	-	-	-	-	-	-	-	
JOD label	Jump on odd		362	4+T	JIF LSB=1	-	-	-	-	-	-	-	-	-	-	-	
JPS label	Jump on positive		365	4+T	JIF MSB=OVF	-	-	-	-	-	-	-	-	-	-	-	
JRN label	Jump on right digit non-zero		377	4+T	JIF RDZ≠1	-	-	-	-	-	-	-	-	-	-	-	
JRZ label	Jump on right digit zero		376	4+T	JIF RDZ=1	-	-	-	-	-	-	-	-	-	-	-	
JSB=label	Jump subroutine	Literal direct	316	9	Jump subroutine	-	-	-	-	-	-	-	-	-	-	-	
JSB XR, label	Jump subroutine	Indexed	306	11	Jump subroutine indexed	-	-	-	-	-	-	-	-	-	-	-	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=0 E	DCM=0 CY	DCM=0 OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
JZR label	Jump on zero		367	4+T	JIF Z=1	-	-	-	-	-	-	-	-	-	-	-	
LDB DR, AR	Load byte	Reg. imm.	240	5	DR←AR	X	X	X	X	-	-	0	0	-	0	0	
LDB DR, = literal	Load byte	Lit. imm.	250	5	DR←M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, AR	Load byte	Reg. dir.	244	6	DR←M(AR)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, = label	Load byte	Lit. dir.	260	6	DR←M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, XAR, label	Load byte	Index dir.	264	8	DR←M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, AR	Load byte	Reg-indir.	254	8	DR←M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, = label	Load byte	Lit. indir.	270	8	DR←M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, XAR, label	Load byte	Index indir.	274	10	DR←M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
LDM DR, AR	Load multi-byte	Reg. imm.	241	4+B	DR←AR	X	X	X	X	-	-	0	0	-	0	0	
LDM DR, = literal	Load multi-byte	Lit. imm.	251	4+B	DR←M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, AR	Load multi-byte	Reg. dir.	245	5+B	DR←M(AR)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, = label	Load multi-byte	Lit. dir.	261	5+B	DR←M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, XAR, label	Load multi-byte	Index dir.	265	7+B	DR←M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, AR	Load multi-byte	Reg. indir.	255	7+B	DR←M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, = label	Load multi-byte	Lit. indir.	271	7+B	DR←M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, XAR, label	Load multi-byte	Index indir.	275	9+B	DR←M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
LLB DR	Logical left byte	Reg. imm.	204	5	Logical left shift DR	X	X	X	X	-	-	X	X	X	0	0	Y
LLM DR	Logical left multi-byte	Reg. imm.	205	4+B	Logical left shift DR	X	X	X	X	-	-	X	X	X	0	0	Y
LRB DR	Logical right byte	Reg. imm.	206	5	Logical right shift DR	X	X	X	X	-	-	X	0	X	0	0	Y
LRM DR	Logical right multi-byte	Re. imm.	207	4+B	Logical right shift DR	X	X	X	X	-	-	X	0	X	0	0	Y
NCB DR	Nine's (or one's) complement byte	Reg. imm.	216	5	DR←$\overline{DR}$	X	X	X	X	-	-	X	X	-	X	0	Y
NCM DR	Nine's (or one's) complement multi-byte	Reg. imm.	217	4+B	DR←$\overline{DR}$	X	X	X	X	-	-	X	X	-	X	0	Y
ORB DR, AR	Or byte inclusive	Reg. imm.	224	5	DR←DR∨AR	X	X	X	X	-	-	0	0	-	0	0	
ORM DR, AR	Or multi-byte inclusive	Reg. imm.	225	4+B	DR←DR∨AR	X	X	X	X	-	-	0	0	-	0	0	
PAD	Pop ARP, DRP and status from stack		237	8	Status←M(SP)	X	X	X	X	X	-	X	X	-	X	X	
POBD DR,+AR	Pop byte with post-increment	Stk. dir.	340	6	DR←M(AR), AR←AR+1	X	X	X	X	-	-	0	0	-	0	0	
POBD DR,-AR	Pop byte with with pre-decrement	Stk. dir.	342	6	DR←M(AR), AR←AR-1	X	X	X	X	-	-	0	0	-	0	0	
POBI DR,+AR	Pop byte with post-increment	Stk. indir.	350	8	DR←M(M(AR)), AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
POBI DR,-AR	Pop byte with pre-decrement	Stk. indir.	352	8	DR←M(M(AR)), AR←AR-2	X	X	X	X	-	-	0	0	-	0	0	
POMD DR,+AR	Pop multi-byte with post-increment	Stk. dir.	341	5+B	DR←M(AR), AR←AR+M	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
POMD DR,-AR	Pop multi-byte with pre-decrement	Stk. dir.	343	5+B	DR←M(AR), AR←AR-M	X	X	X	X	-	-	0	0	-	0	0	
POMI DR,+AR	Pop multi-byte with post-increment	Stk. indir.	351	7+B	DR←M(M(AR)), AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
POMI DR,-AR	Pop multi-byte with pre-decrement	Stk. indir.	353	7+B	DR←M(M(AR)), AR←AR-2	X	X	X	X	-	-	0	0	-	0	0	
PUBD DR,+AR	Push byte with post-increment	Stk. dir.	344	6	M(AR)←DR, AR←AR+1	X	X	X	X	-	-	0	0	-	0	0	
PUBD DR,-AR	Push byte with pre-decrement	Stk. dir.	346	6	AR←AR-1, M(AR)←DR	X	X	X	X	-	-	0	0	-	0	0	
PUBI DR,+AR	Push byte with post-increment	Stk. indir.	354	8	M(M(AR))←DR, AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
PUBI DR,-AR	Push byte with pre-decrement	Stk. indir.	356	8	AR←AR-2, M(M(AR))←DR	X	X	X	X	-	-	0	0	-	0	0	
PUMD DR,+AR	Push multi-byte with post-increment	Stk. dir.	345	5+B	M(AR)←DR, AR←AR+M	X	X	X	X	-	-	0	0	-	0	0	
PUMD DR,-AR	Push multi-byte with pre-decrement	Stk. dir.	347	5+B	AR←AR-M, M(AR)←DR	X	X	X	X	-	-	0	0	-	0	0	
PUMI DR,+AR	Push multi-byte with post-increment	Stk. indir.	355	7+B	M(M(AR))←DR, AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
PUMI DR,-AR	Push multi-byte with pre-decrement	Stk. indir.	357	7+B	AR←AR-2, M(M(AR))←DR	X	X	X	X	-	-	0	0	-	0	0	
RTN	Subroutine return		236	5	SP←SP-2, PC←M(SP)	-	-	-	-	-	-	-	-	-	-	-	
SAD	Save ARP, DRP and status on stack		232	8	M(SP)←Status	-	-	-	-	-	-	-	-	-	-	-	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
SBB DR, AR	Subtract byte	Reg. imm.	304	5	DR←DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBB DR, = literal	Subtract byte	Lit. imm.	314	5	DR←DR+$\overline{M(PC+1)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBBD DR, AR	Subtract byte	Reg. dir.	334	6	DR←DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBBD DR, = label	Subtract byte	Lit. dir.	324	6	DR←DR+$\overline{M(ADR)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBM DR, AR	Subtract multi-byte	Reg. imm.	305	4+B	DR←DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBM DR, = literal	Subtract multi-byte	Lit. imm.	315	4+B	DR←DR+$\overline{M(PC+1)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBMD DR, AR	Subtract multi-byte	Reg. dir.	335	5+B	DR←DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBMD DR, = literal	Subtract multi-byte	Lit. dir.	325	5+B	DR←DR+$\overline{M(ADR)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
STB DR, AR	Store byte	Reg. imm.	242	5	DR→AR	X	X	X	X	-	-	0	0	-	0	0	
STB DR, = literal	Store byte	Lit. imm.	252	5	DR→M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, AR	Store byte	Reg. dir.	246	6	DR→M(AR)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, = label	Store byte	Lit. dir.	262	6	DR→M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, XAR, label	Store byte	Index dir.	266	8	DR→M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, AR	Store byte	Reg. indir.	256	8	DR→M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, = label	Store byte	Lit. indir.	272	8	DR→M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, XAR, label	Store byte	Index indir	276	10	DR→M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
STM DR, AR	Store multi-byte	Reg. imm.	243	4+B	DR→AR	X	X	X	X	-	-	0	0	-	0	0	
STM DR, = literal	Store multi-byte	Lit. imm.	253	4+B	DR→M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
STMD DR, AR	Store multi byte	Reg. dir.	247	5+B	DR→M(AR)	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status											Binary/ BCD Option
											DCM=0			DCM=1			
						LSB	MSB	RDZ LDZ	Z	DCM	E	CY	OVF	E	CY	OVF	
STMD DR, = label	Store multi-byte	Lit. dir.	263	5+B	DR→M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
STMD DR, XAR, label	Store multi-byte	Index dir.	267	7+B	DR→M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, AR	Store multi-byte	Reg. indir.	257	7+B	DR→M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, = label	Store multi-byte	Lit. indir.	273	7+B	DR→M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, XAR, label	Store multi-byte	Index indir	277	9+B	DR→M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
TCB DR	Ten's (or two's) complement byte	Reg. imm.	214	5	DR←$\overline{DR}$+1	X	X	X	X	-	-	0	0	-	0	0	Y
TCM DR	Ten's (or two's) complement multi-byte	Reg. imm.	215	4+B	DR←$\overline{DR}$+1	X	X	X	X	-	-	0	0	-	0	0	Y
TSB DR	Test byte	Reg. imm.	220	5	Test DR	X	X	X	X	-	-	X	X	-	X	0	Y
TSM DR	Test multi-byte	Reg. imm.	221	4+B	Test DR	X	X	X	X	-	-	X	X	-	X	0	Y
XRB DR, AR	Or byte exclusive	Reg. imm.	226	5	DR←DR ⊕ AR	X	X	X	X	-	-	0	0	-	0	0	
XRM DR, AR	Or multi-byte exclusive	Reg. imm.	227	4+B	DR←DR ⊕ AR	X	X	X	X	-	-	0	0	-	0	0	

Appendix D

ASSEMBLER INSTRUCTION CODING

The chart below shows how the CPU instructions appear when assembled into machine language object code by the computer.

7	6	5	4	3	2	1	0
0	DRP/ ARP	≠000001 Load with literal =000001 Load with RØ					
1	0	0	0	0	Logical/ Extended	Right/Left	M/B
1	0	0	0	1	0	Decrement/ Increment	M/B
1	0	0	0	1	1	Nine's Complement/ Ten's Complement	M/B
1	0	0	1	0	0	Clear/Test	M/B
1	0	0	1	0	1	XOR/OR	M/B
1	0	0	1	1	000 BIN 001 BCD 010 SAD 011 DCE 100 ICE 101 CLE 110 RTN 111 PAD		
1	0	1	000 REG IMM 001 REG DIR 010 LIT IMM 011 REG IND 100 LIT DIR 101 INX DIR 110 LIT IND 111 INX IND			Store/Load	M/B
1	1	0	00 REG IMM 01 LIT IMM 10 LIT DIR 11 REG DIR		00 CMP 01 ADD 10 SUB		M/B
					11 AND		1
1	1	0	00 INX 01 LIT		11 JSB		0
1	1	1	0	IND/ DIR	PUSH/ POP	-ADR/ +ADR	M/B
1	1	1	1	000 001 010 011 100 101 110 111			JNO/JMP JEV/JOD JPS/JNG JZR/JNZ JEZ/JEN JCY/JNC JLN/JLZ JRN/JRZ

X/Y = 1/0