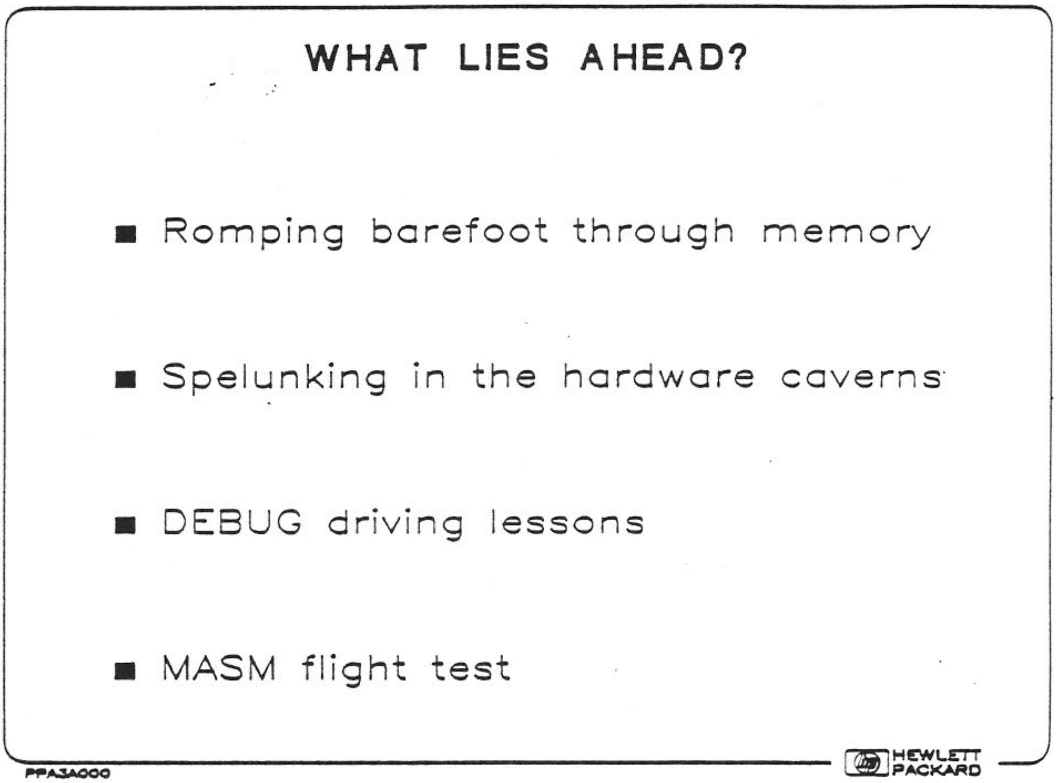


WHAT LIES AHEAD?

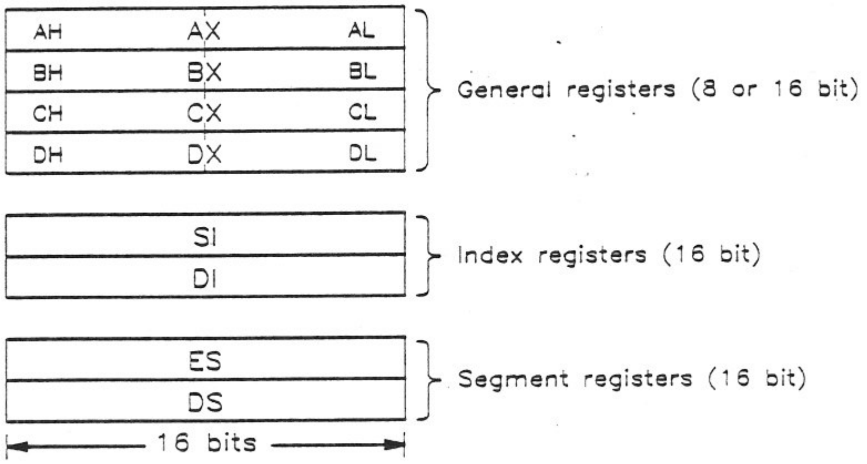
- Romping barefoot through memory
- Spelunking in the hardware caverns
- DEBUG driving lessons
- MASM flight test

PPA3A000

HEWLETT PACKARD

Notes: PPA3A000

8086 CPU
FREQUENTLY-USED REGISTERS



AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL

General registers (8 or 16 bit)

SI

DI

Index registers (16 bit)

ES

DS

Segment registers (16 bit)

16 bits

16 bits can represent 2^{16} = 65536 (64K) memory locations. How can a 16-bit CPU access more than 64K of memory?

HEWLETT PACKARD

PPA3A010

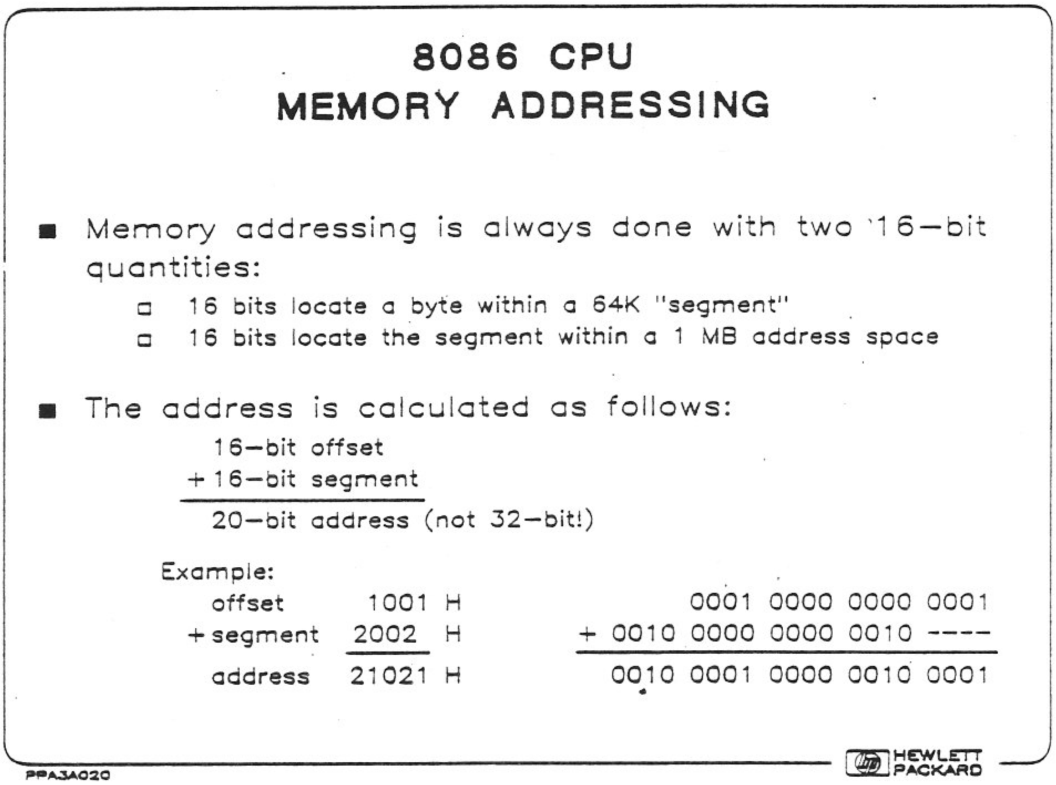
Notes:

PPA3010

The 8086 is an upgrade of the 8080--an 8-bit CPU with 16-bit addressing. Some of the strange organization of the 8086 can be explained as necessary to maintain upward compatibility from the 8080.

The GENERAL REGISTERS are where most data manipulation takes place. These may be used as 16-bit registers (AX) or 8-bit registers (AH and AL).

INDEX REGISTERS and SEGMENT REGISTERS are used for memory addressing.



8086 CPU
MEMORY ADDRESSING

- Memory addressing is always done with two 16-bit quantities:
  - □ 16 bits locate a byte within a 64K "segment"
  - □ 16 bits locate the segment within a 1 MB address space
- The address is calculated as follows:

```
    16-bit offset
  + 16-bit segment
  ----------------
    20-bit address (not 32-bit!)
```

Example:

```
    offset     1001  H           0001 0000 0000 0001
  + segment    2002  H    + 0010 0000 0000 0010 ----
              ------      --------------------------
    address   21021  H      0010 0001 0000 0010 0001
```

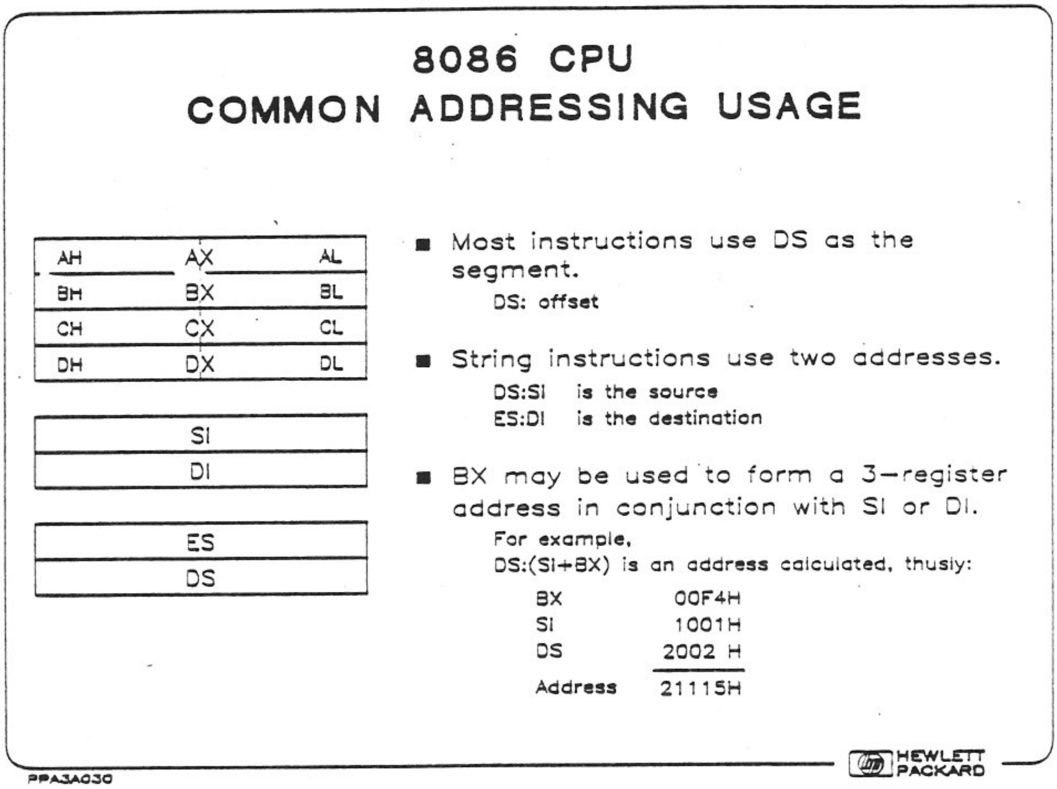
PPA3A020

HEWLETT PACKARD

Notes:

PPA3A020

Because the segment and offset overlap, a segment may begin at any multiple of 16 bytes (a "paragraph").



8086 CPU
COMMON ADDRESSING USAGE

AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL

SI
DI

ES
DS

- Most instructions use DS as the segment.
  DS: offset

- String instructions use two addresses.
  DS:SI is the source
  ES:DI is the destination

- BX may be used to form a 3–register address in conjunction with SI or DI.
  For example,
  DS:(SI+BX) is an address calculated, thusly:

BX	00F4H
SI	1001H
DS	2002 H
Address	21115H

PPA3A030

HEWLETT PACKARD

Notes:

PPA3A030

Addresses are usually written as a segment register, a colon, and whatever follows the colon defines the offset. Offsets may be:

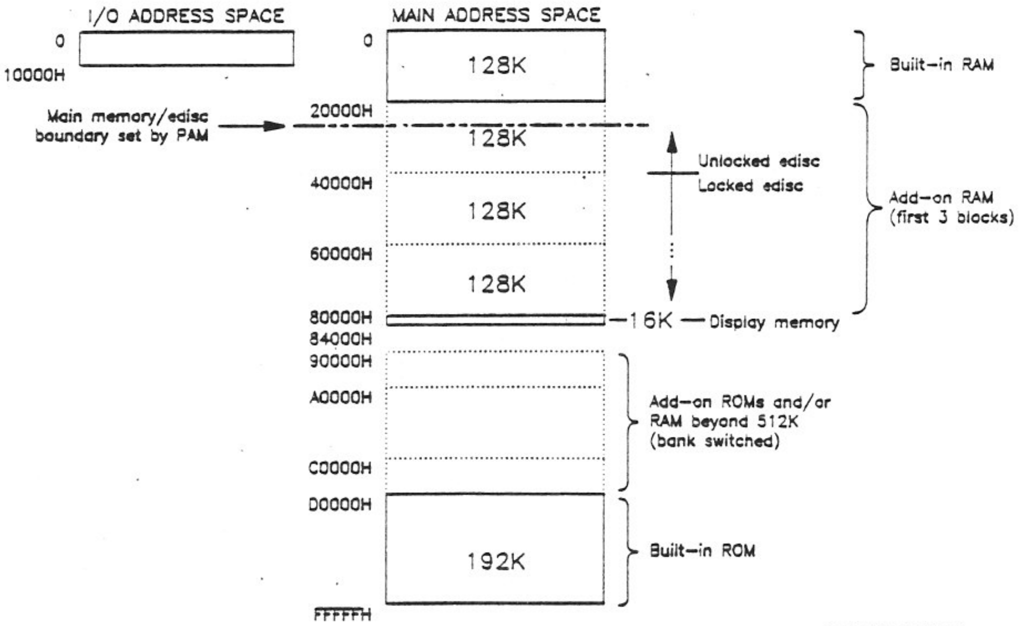
- A fixed value (for example, the storage location of a variable used by the program).

- The value currently contained in SI, DI, or BX.

- The sum of SI and BX, or DI and BX (useful for finding entries in a table).

String instructions not only work with two memory locations, but also handle autoincrement or autodecrement and multiple repetitions. More on that later.

PORTABLE PLUS MEMORY MAP (OVERVIEW)



I/O ADDRESS SPACE

0

10000H

MAIN ADDRESS SPACE

0

128K

Built-in RAM

Main memory/edisc boundary set by PAM

20000H

128K

Unlocked edisc

Locked edisc

40000H

128K

Add-on RAM (first 3 blocks)

60000H

128K

80000H

84000H

16K — Display memory

90000H

A0000H

Add-on ROMs and/or RAM beyond 512K (bank switched)

C0000H

D0000H

192K

Built-in ROM

FFFFFH

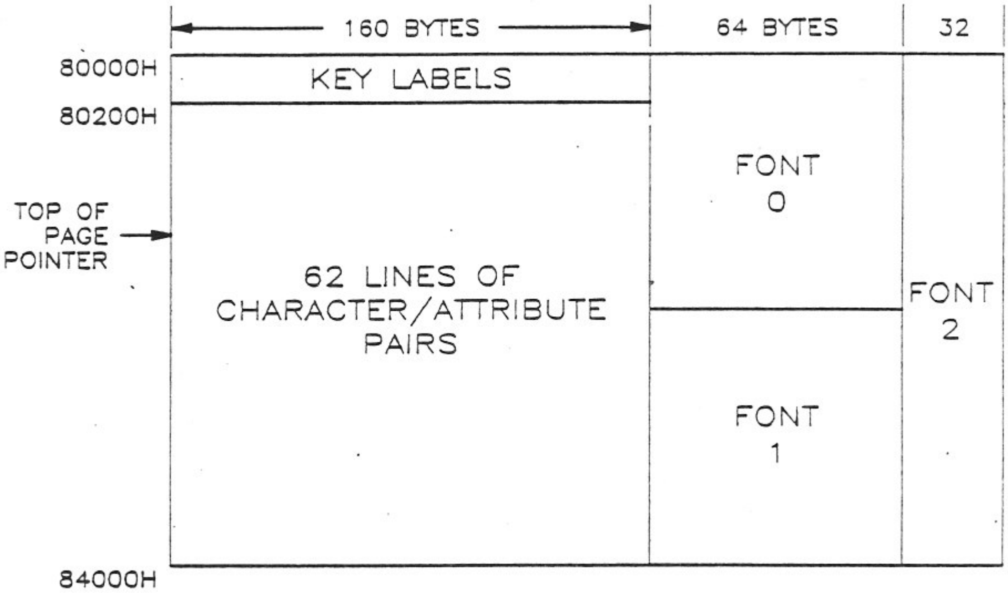
PPA3A040

HEWLETT PACKARD

Notes:

PPA3A040

PORTABLE PLUS DISPLAY MEMORY (ALPHA)



160 BYTES

64 BYTES

32

80000H

80200H

KEY LABELS

TOP OF PAGE POINTER

62 LINES OF CHARACTER/ATTRIBUTE PAIRS

FONT 0

FONT 1

FONT 2

84000H

PPA3A050

HEWLETT PACKARD

Notes:

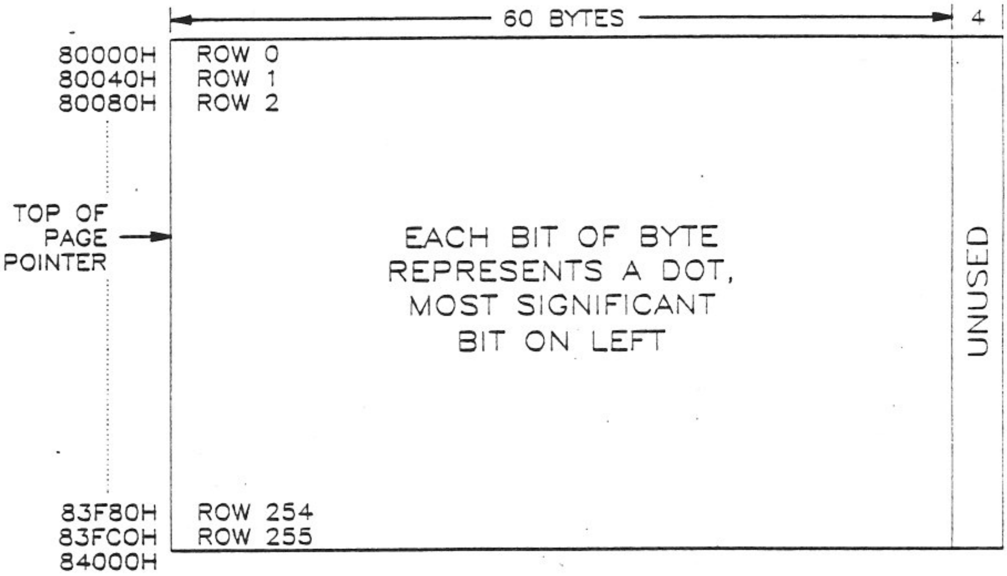
PPA3A050

Single-line scroll to page top pointer, display automatically skips key labels.

Characters created on the fly from font table lookup plus enhancements from attribute byte.

Refresh rate 60/second.

PORTABLE PLUS
DISPLAY MEMORY (GRAPHICS)



60 BYTES

4

80000H ROW 0

80040H ROW 1

80080H ROW 2

TOP OF PAGE POINTER

EACH BIT OF BYTE REPRESENTS A DOT, MOST SIGNIFICANT BIT ON LEFT

UNUSED

83F80H ROW 254

83FC0H ROW 255

84000H

PPA3A060

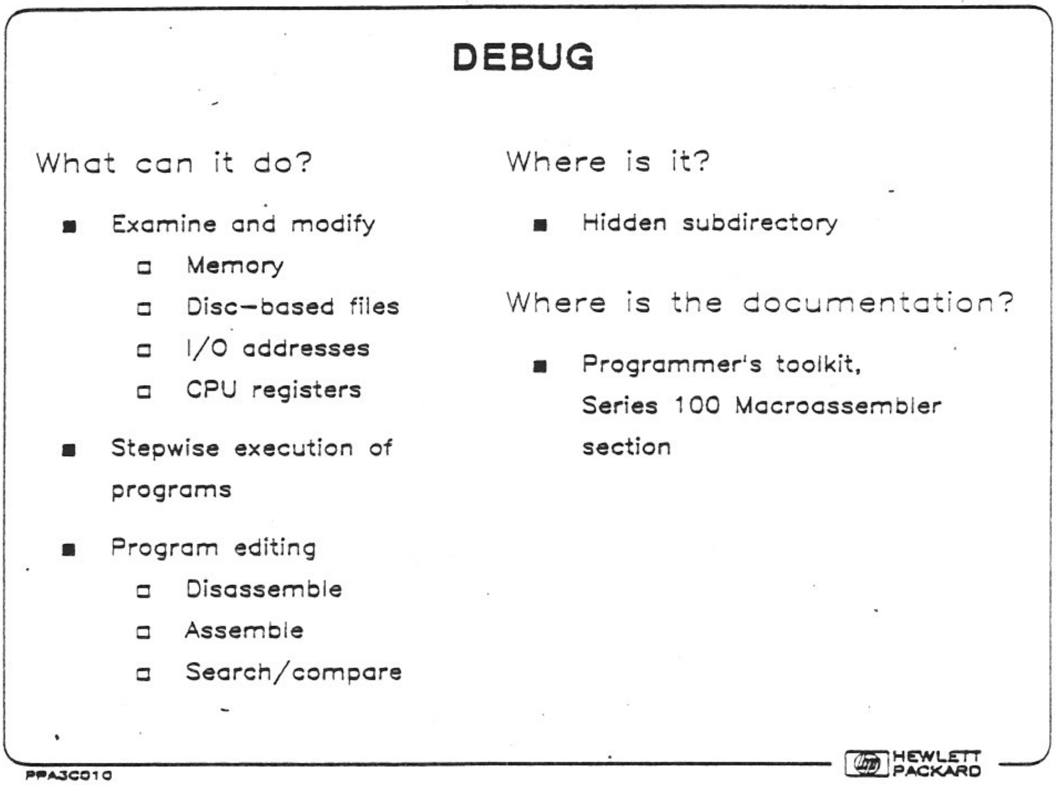
Notes:

PPA3A060

One line (pixel) scroll possible with pointer change.

Four bytes per line unused (480 dots of 512 used).

Alpha and graphics mutually exclusive (same space), but you can do alpha in graphics via interrupt 10h.



DEBUG

What can it do?

- Examine and modify
  - □ Memory
  - □ Disc–based files
  - □ I/O addresses
  - □ CPU registers
- Stepwise execution of programs
- Program editing
  - □ Disassemble
  - □ Assemble
  - □ Search/compare

Where is it?

- Hidden subdirectory

Where is the documentation?

- Programmer's toolkit, Series 100 Macroassembler section

PPA3C010

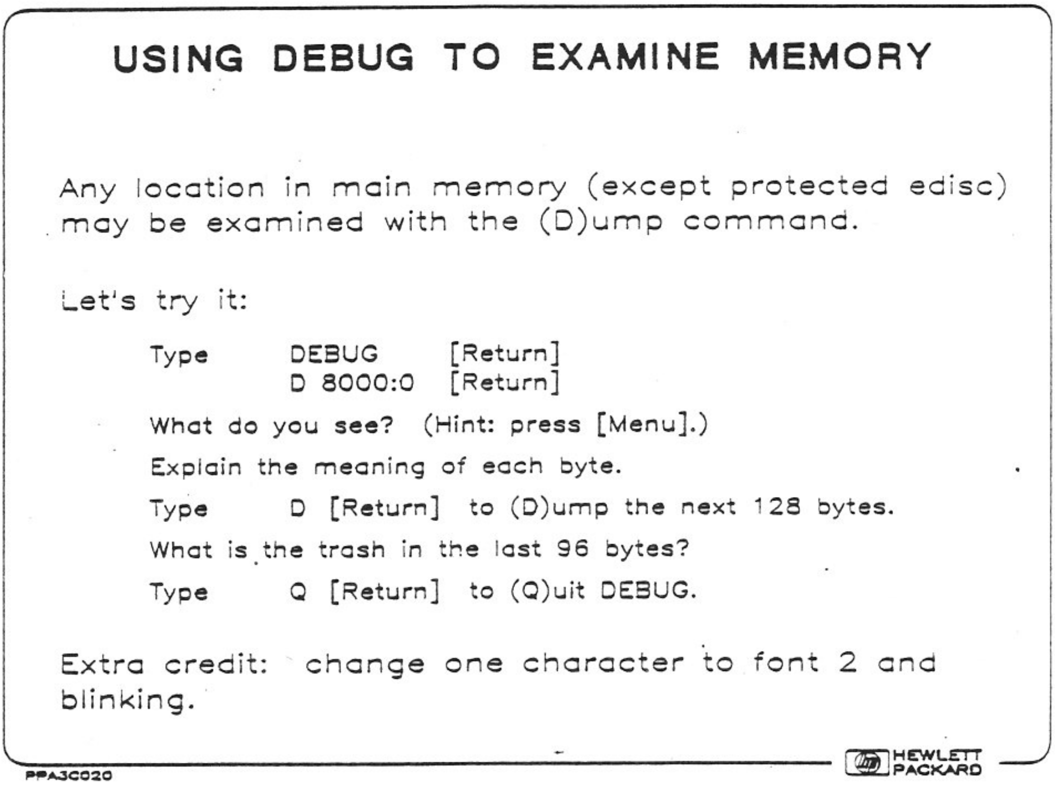
Notes:

PPA3C010

The debug utility can examine or modify any byte(s) in RAM or on disc, including boot sectors, file allocation tables, and directories.

When used to step through a program, DEBUG's ability to examine and modify CPU registers or I/O addresses simplifies program debugging.

Debug is built into every Portable PLUS but not documented. It is in the B:\BIN\ETC hidden directory.



USING DEBUG TO EXAMINE MEMORY

Any location in main memory (except protected edisc) may be examined with the (D)ump command.

Let's try it:

```
Type      DEBUG      [Return]
          D 8000:0   [Return]
```

What do you see? (Hint: press [Menu].)

Explain the meaning of each byte.

```
Type      D  [Return]  to (D)ump the next 128 bytes.
```

What is the trash in the last 96 bytes?

```
Type      Q  [Return]  to (Q)uit DEBUG.
```

Extra credit: change one character to font 2 and blinking.

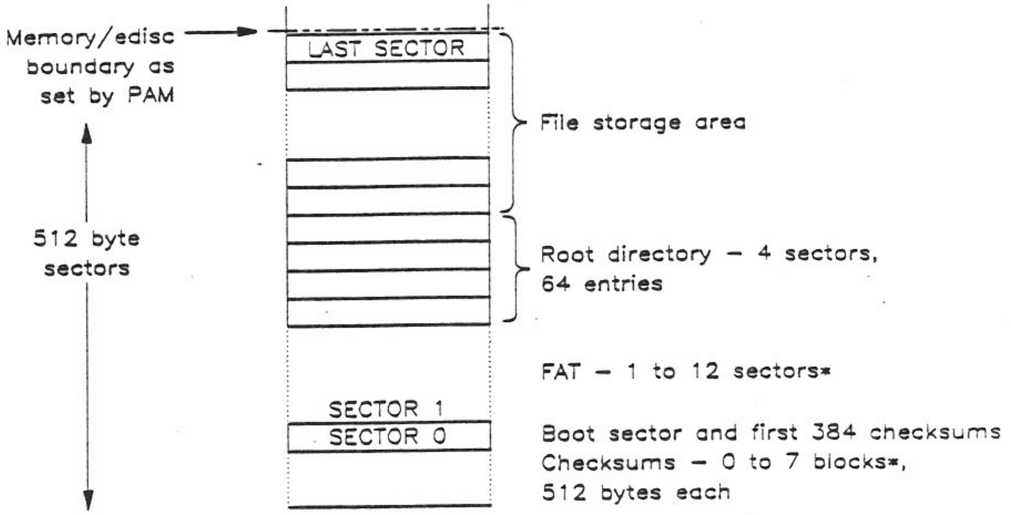
PPA3C020

HEWLETT PACKARD

Notes:

PPA3C020

EDISC MEMORY MAP


Memory/edisc boundary as set by PAM
LAST SECTOR
File storage area
512 byte sectors
Root directory – 4 sectors, 64 entries
FAT – 1 to 12 sectors*
SECTOR 1
SECTOR 0
Boot sector and first 384 checksums
Checksums – 0 to 7 blocks*, 512 bytes each

* Size depends on amount of memory installed.

PPA3C030

HEWLETT PACKARD

Notes:

PPA3C030

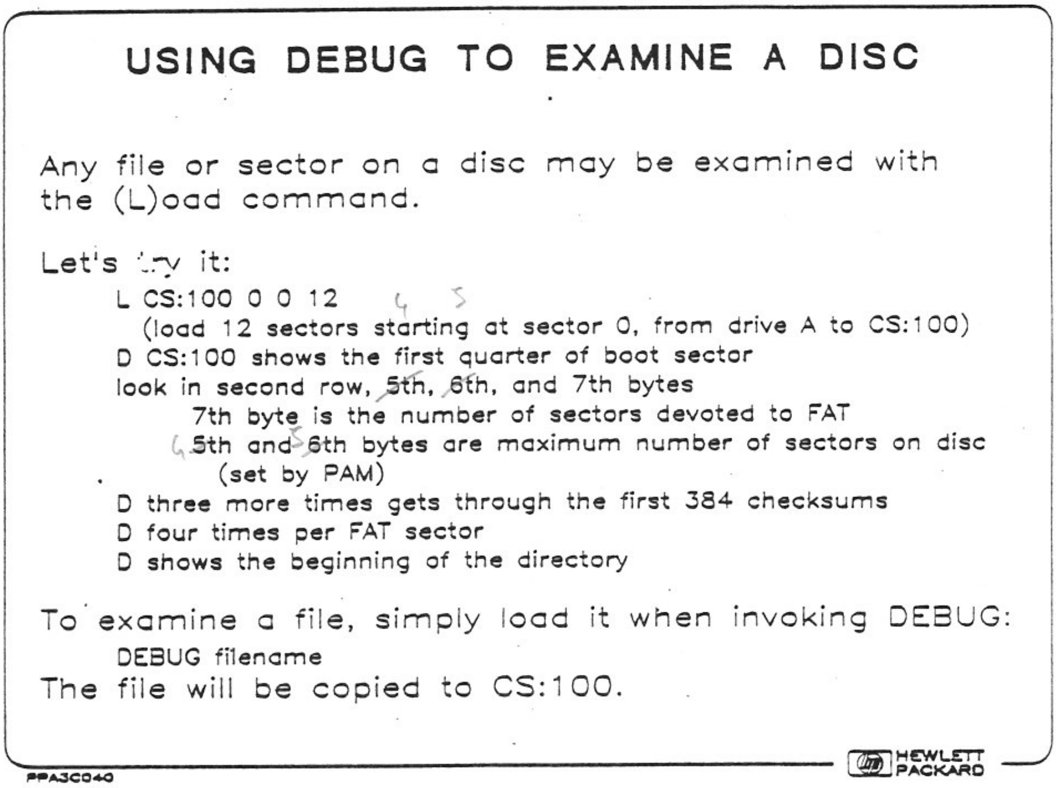
Edisc is 512-byte sectors.

If memory size is no larger than 512K, the low numbered sectors are at high addresses and sectors are sequential in reverse order. (Within the sector, bytes read from low to high address, but at the end of the sector, the next sector is normally found at a lower address.)

If memory size is above 512K, the first 256 sectors are in the first block-swapable 128K, then other block-swapable memory is used, then the highest-numbered sectors are in the low 512K of memory.

The operating system supports up to 2MB of edisc, plus 512K of main memory.

To minimize wasted space, the areas reserved for checksums (one byte per sector) and the File Allocation Table are of variable size depending on the amount of memory installed.



USING DEBUG TO EXAMINE A DISC

Any file or sector on a disc may be examined with the (L)oad command.

Let's try it:

L CS:100 0 0 12 4 5
(load 12 sectors starting at sector 0, from drive A to CS:100)
D CS:100 shows the first quarter of boot sector
look in second row, ~~5~~th, ~~6~~th, and 7th bytes
7th byte is the number of sectors devoted to FAT
4 ~~5~~th and 5 ~~6~~th bytes are maximum number of sectors on disc (set by PAM)
D three more times gets through the first 384 checksums
D four times per FAT sector
D shows the beginning of the directory

To examine a file, simply load it when invoking DEBUG:

DEBUG filename

The file will be copied to CS:100.

PPA3C040

HEWLETT PACKARD

Notes:

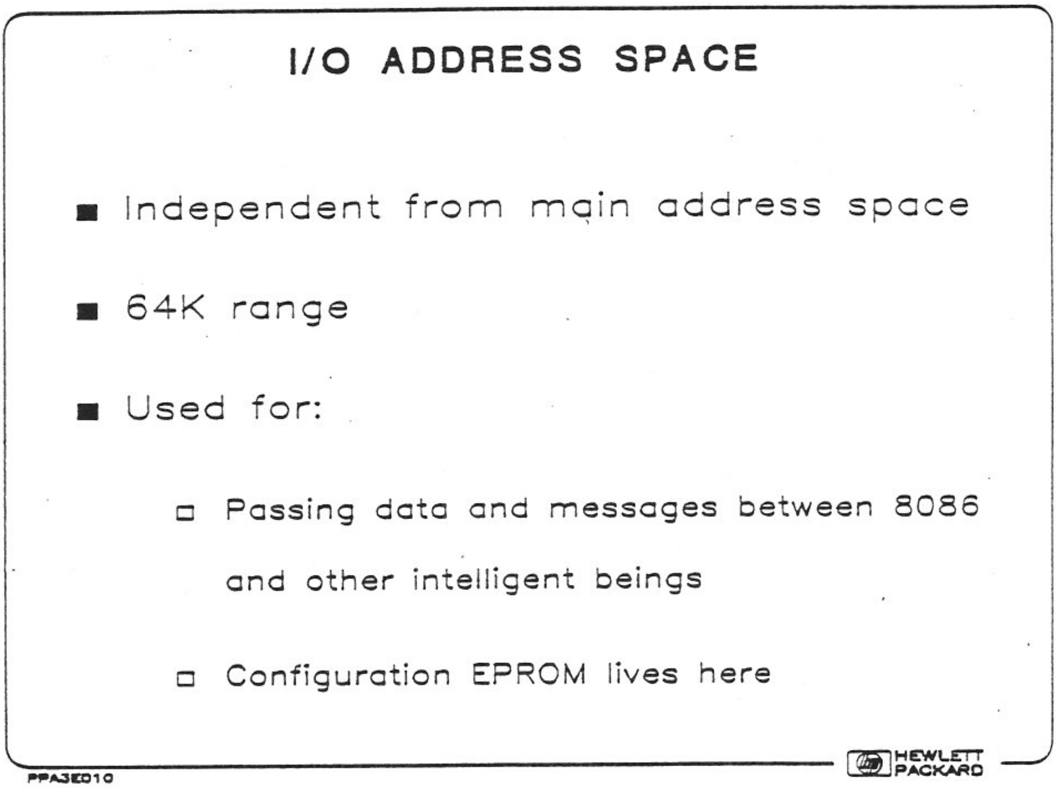
PPA3C040

Debug can load files or sectors into memory for examination, modification, and writing back to disc. (Don't write changes back to the disc unless you know what you're doing--changing the first few sectors can render the whole disc inaccessible!)

The boot sector includes the name of the machine which formatted the disc (45711) and lots of other information about the disc. Most of a boot sector is undefined on a floppy disc, but the Portable PLUS uses the last 3/4 to store checksums. (FF is the checksum for a sector which has never been used.)

The File Allocation Table is a collection of 12-bit pointers which link the records of each file.

Each directory entry is 32 bytes and includes such information as time and date last modified, file type, file size, and sector number of the first sector.



I/O ADDRESS SPACE

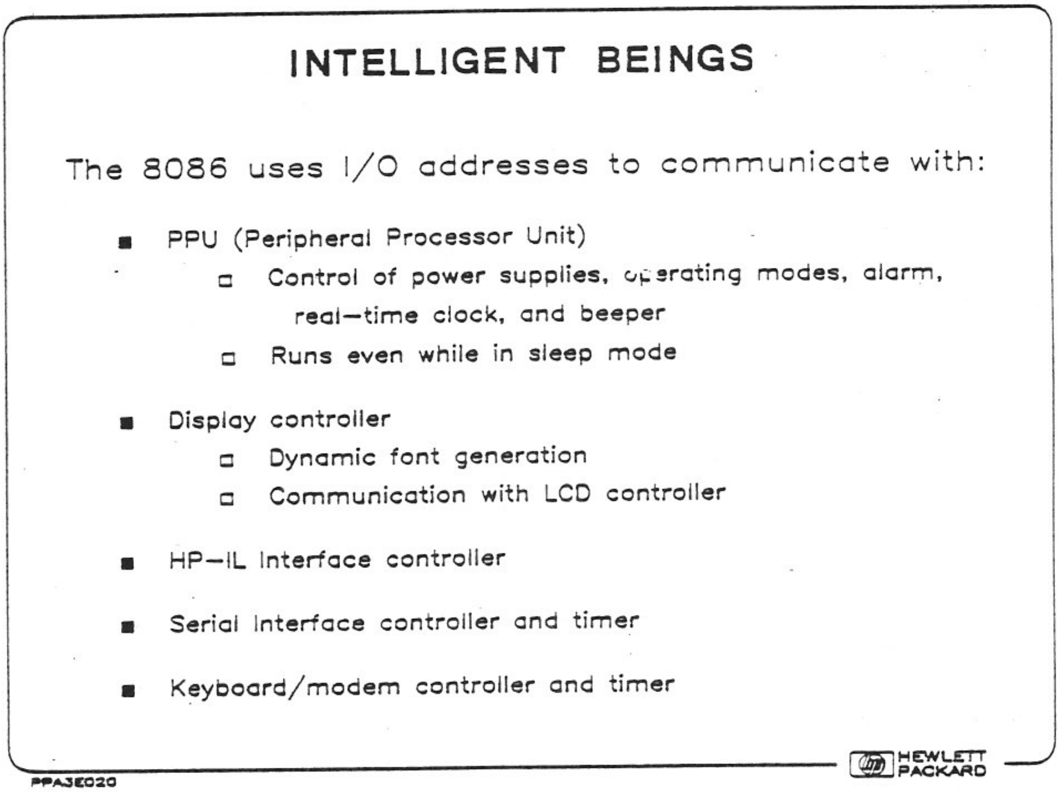
- Independent from main address space
- 64K range
- Used for:
  - □ Passing data and messages between 8086 and other intelligent beings
  - □ Configuration EPROM lives here

PPA3E010

HEWLETT PACKARD

Notes:

PPA3E010



INTELLIGENT BEINGS

The 8086 uses I/O addresses to communicate with:

- PPU (Peripheral Processor Unit)
  - □ Control of power supplies, operating modes, alarm, real-time clock, and beeper
  - □ Runs even while in sleep mode
- Display controller
  - □ Dynamic font generation
  - □ Communication with LCD controller
- HP-IL Interface controller
- Serial Interface controller and timer
- Keyboard/modem controller and timer

HEWLETT PACKARD

PPA3E020

Notes:

PPA3E020

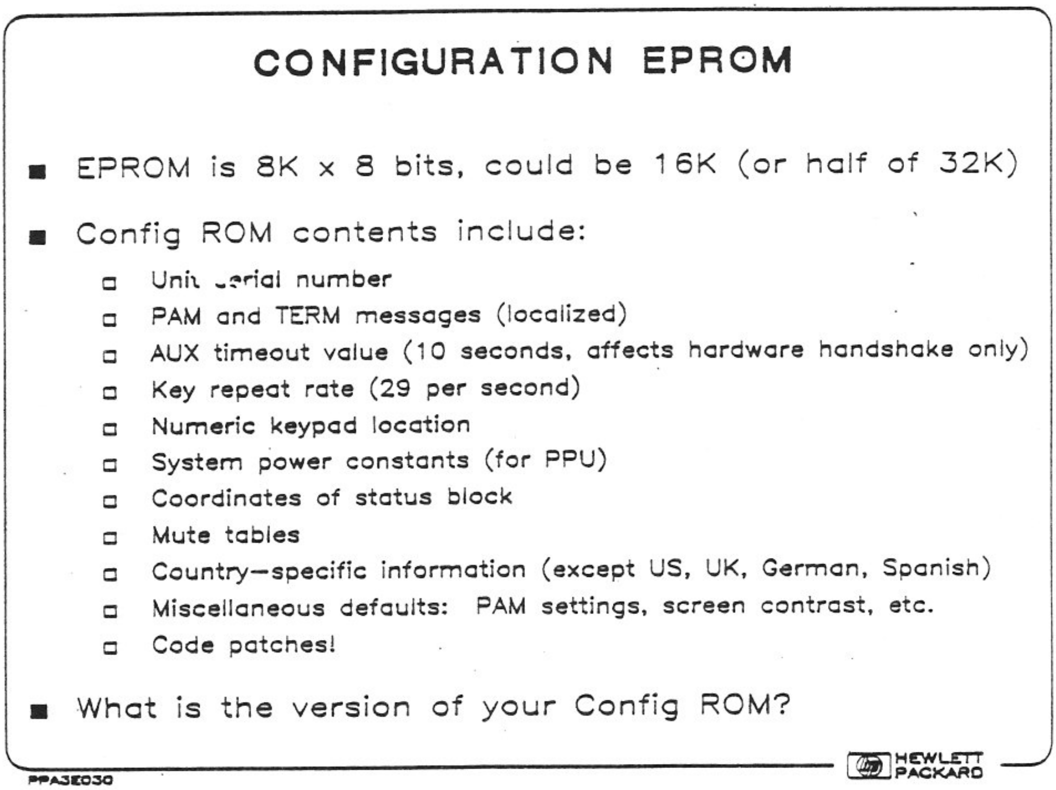
The PPU brings the system up after drawer changes or hard resets, puts the unit to sleep, runs while asleep, and provides the real-time clock.

The display controller takes the LCD RAM and uses it, along with current settings for alpha or graphics mode, window location, cursor location and top of page location, to generate a dot pattern to be sent to the LCD controller (and the optional video output).

The HP-IL controller is a standard part which is documented in other manuals.

Two "kitchen sink" chips control the serial interface and keyboard and modem respectively. (One chip can handle both the keyboard and modem because both are slow.)

Note the two timers and one real-time clock, above and beyond the two oscillators which run the 8086 and the LCD controller. One timer provides the MS-DOS "heartbeat" while the other keeps track of timeouts and such.



CONFIGURATION EPROM

- EPROM is 8K x 8 bits, could be 16K (or half of 32K)
- Config ROM contents include:
  - □ Unit serial number
  - □ PAM and TERM messages (localized)
  - □ AUX timeout value (10 seconds, affects hardware handshake only)
  - □ Key repeat rate (29 per second)
  - □ Numeric keypad location
  - □ System power constants (for PPU)
  - □ Coordinates of status block
  - □ Mute tables
  - □ Country–specific information (except US, UK, German, Spanish)
  - □ Miscellaneous defaults: PAM settings, screen contrast, etc.
  - □ Code patches!
- What is the version of your Config ROM?

PPA3E030

HEWLETT PACKARD

Notes:

PPA3E030

The AUX driver, unlike the 110, will not time out after receiving an XOFF.

The key repeat rate is over twice as fast as the 110, but no repeat happens unless the buffer is empty. Therefore, repeat rate will often be limited by the application.

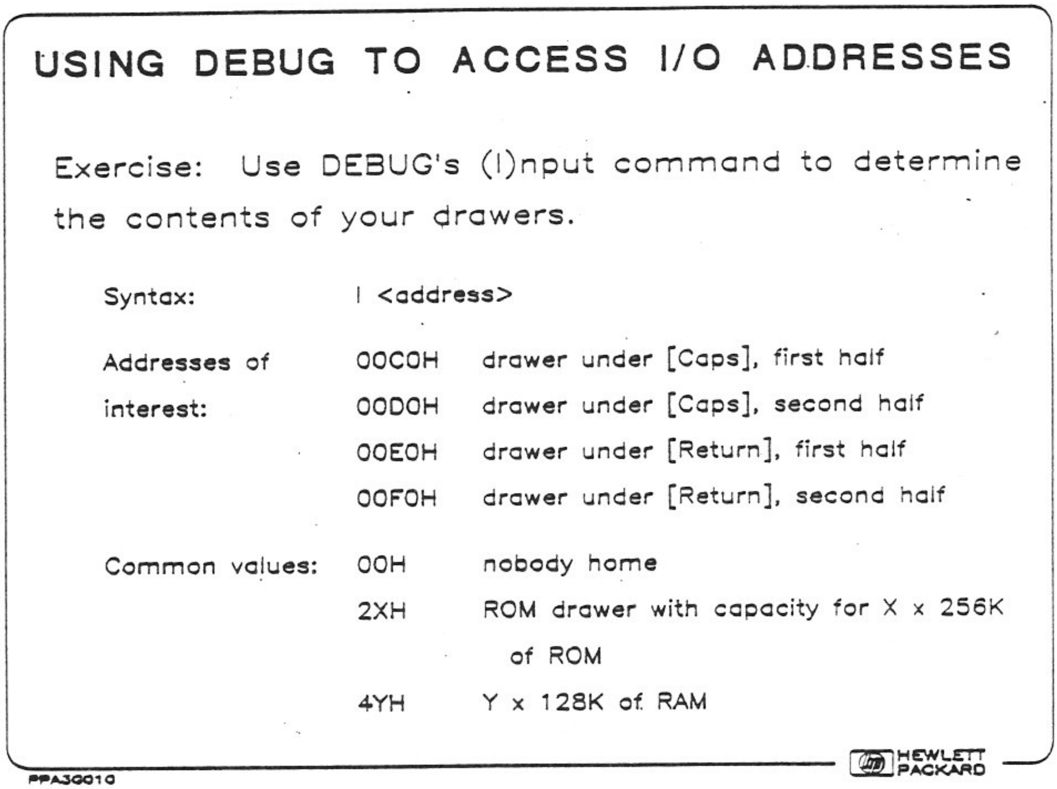
The power constants tell the PPU how much power is used by the serial interface, etc. for use in its fuel gauge calculation.

The status block elements (toggle key indicators, time, cursor position) can be located anywhere on the screen.

Mute tables determine whether the cursor is advanced after hitting the umlaut key, etc.

Country-specific information includes local currently symbols, radix, uppercasing table, etc. The primary languages are included in the system ROMs.

Because the EPROM contains code patches, to know what version of software a customer has, you must check the EPROM version. (Reboot and watch the upper right corner of the screen.)



USING DEBUG TO ACCESS I/O ADDRESSES

Exercise: Use DEBUG's (I)nput command to determine the contents of your drawers.

Syntax:	I <address>	
Addresses of interest:	00C0H	drawer under [Caps], first half
	00D0H	drawer under [Caps], second half
	00E0H	drawer under [Return], first half
	00F0H	drawer under [Return], second half
Common values:	00H	nobody home
	2XH	ROM drawer with capacity for X x 256K of ROM
	4YH	Y x 128K of RAM

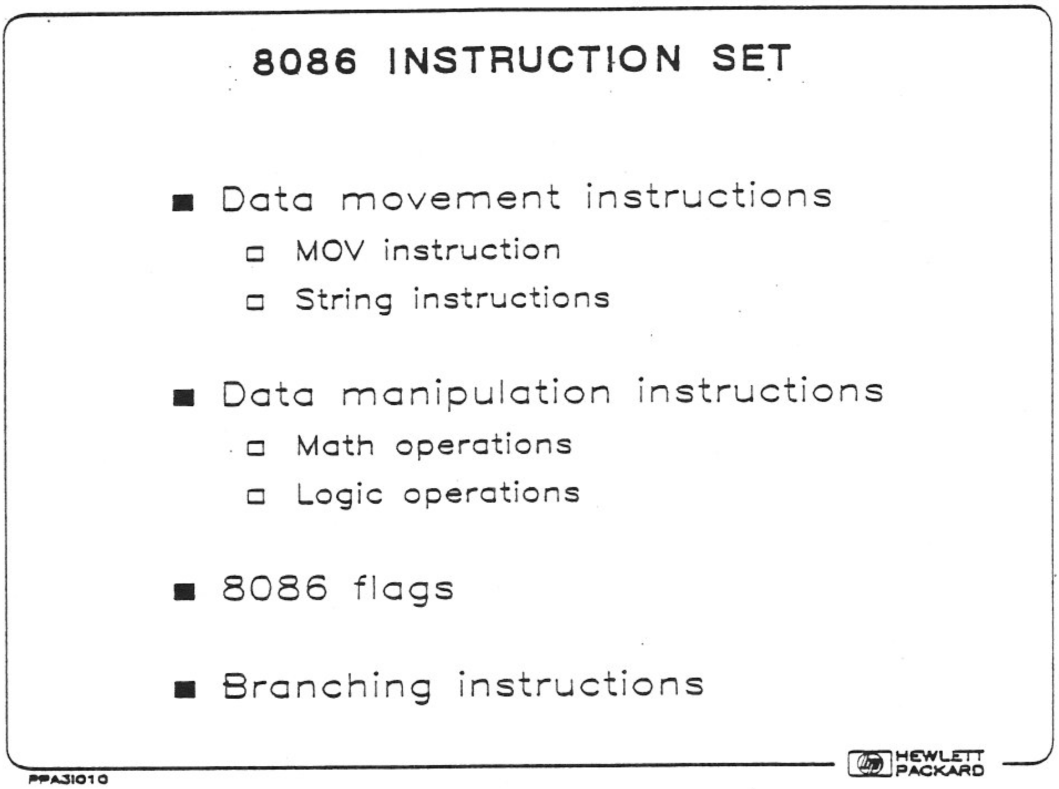
PPA3G010

HEWLETT PACKARD

Notes:

PPA3G010

For each drawer (and virtual drawer), there is a status byte which indicates the drawer contents. This status byte is obtained by (I)nputting from the appropriate I/O address.



8086 INSTRUCTION SET

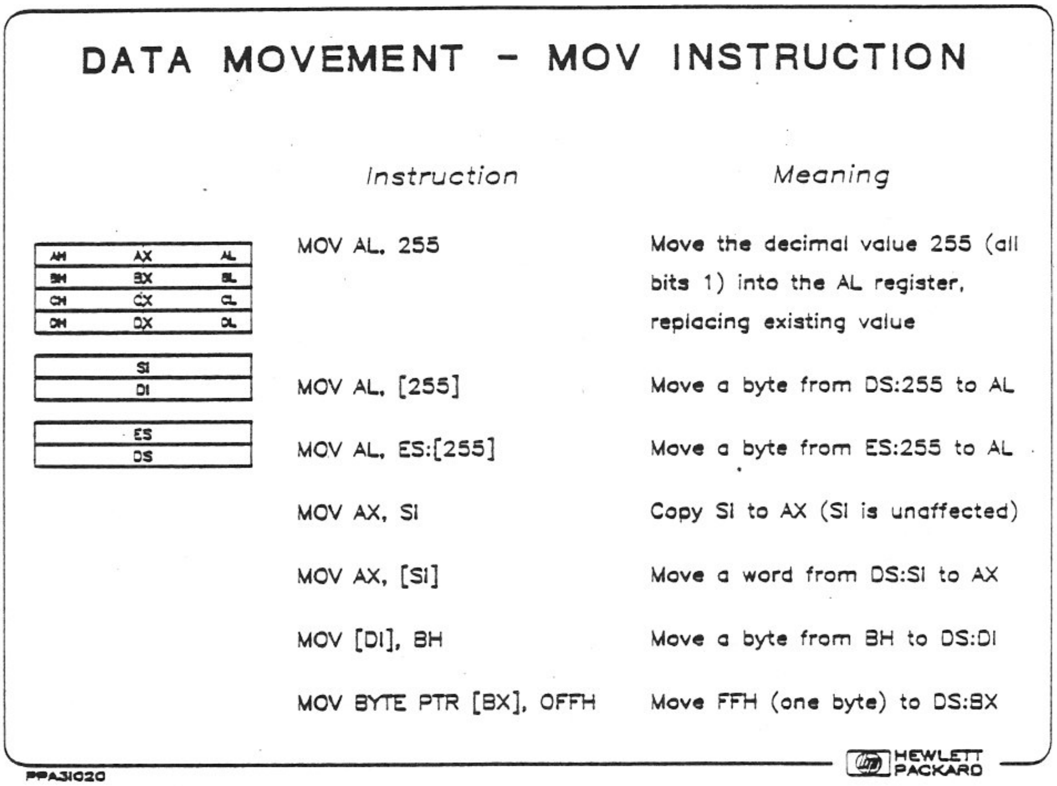
- Data movement instructions
  - MOV instruction
  - String instructions
- Data manipulation instructions
  - Math operations
  - Logic operations
- 8086 flags
- Branching instructions

HEWLETT PACKARD

PPA3I010

Notes:

PPA3I010



DATA MOVEMENT - MOV INSTRUCTION

AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL
	SI	
	DI	
	ES	
	DS	

Instruction	*Meaning*
MOV AL, 255	Move the decimal value 255 (all bits 1) into the AL register, replacing existing value
MOV AL, [255]	Move a byte from DS:255 to AL
MOV AL, ES:[255]	Move a byte from ES:255 to AL
MOV AX, SI	Copy SI to AX (SI is unaffected)
MOV AX, [SI]	Move a word from DS:SI to AX
MOV [DI], BH	Move a byte from BH to DS:DI
MOV BYTE PTR [BX], OFFH	Move FFH (one byte) to DS:BX

HEWLETT PACKARD

PPA3I020

Notes:

PPA3I020

Whenever an instruction has two operands, the first is the destination. The source is unaffected.

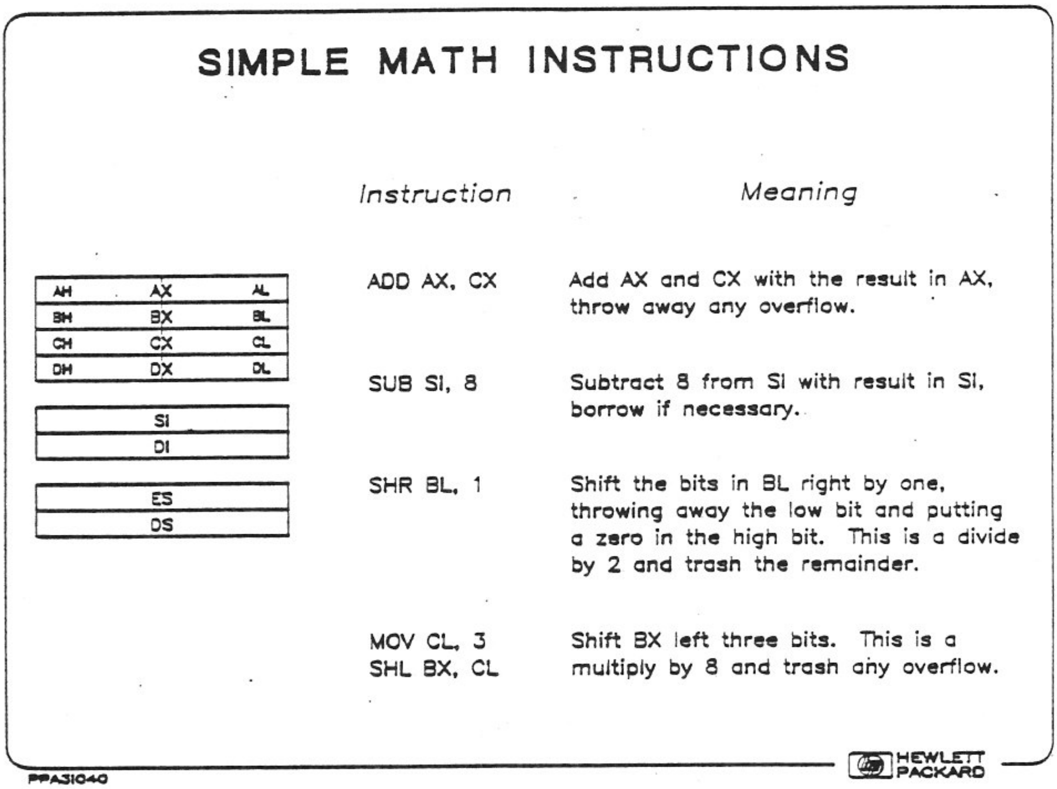
Numbers are decimal by default. You can change the assembler's default or follow numbers with B or H to indicate binary or hex.

Numbers without brackets are used as-is; anything in brackets is an offset into memory where the actual data is moved from/to.

For register-to-register moves, size must match.

Hex numbers must start with a digit 0 through 9. For hex numbers which start with A through F, precede with a zero.

For moves where no register is the source or destination, the assembler doesn't know the size of the data. Tell it with "BYTE PTR" or "WORD PTR".



SIMPLE MATH INSTRUCTIONS

AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL

SI
DI

ES
DS

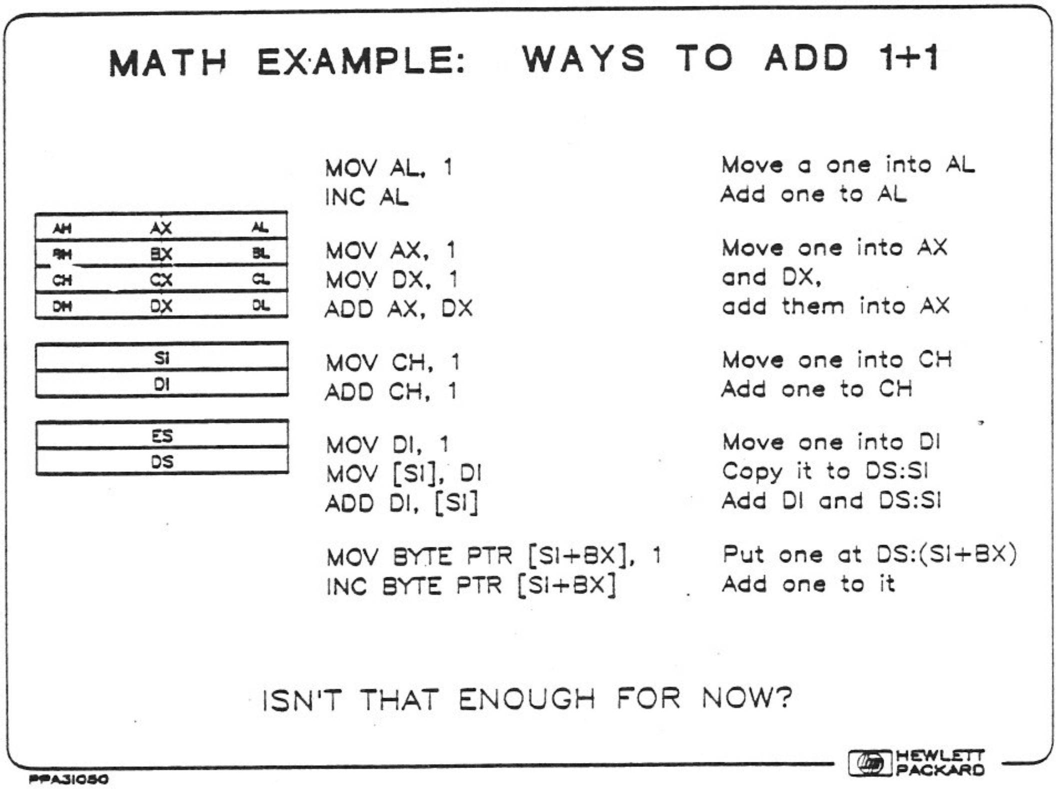
Instruction	*Meaning*
ADD AX, CX	Add AX and CX with the result in AX, throw away any overflow.
SUB SI, 8	Subtract 8 from SI with result in SI, borrow if necessary.
SHR BL, 1	Shift the bits in BL right by one, throwing away the low bit and putting a zero in the high bit. This is a divide by 2 and trash the remainder.
MOV CL, 3 SHL BX, CL	Shift BX left three bits. This is a multiply by 8 and trash any overflow.

PPA31040

HEWLETT PACKARD

Notes:

PPA31040



MATH EXAMPLE: WAYS TO ADD 1+1

AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL

SI
DI

ES
DS

```
MOV AL, 1                       Move a one into AL
INC AL                          Add one to AL

MOV AX, 1                       Move one into AX
MOV DX, 1                       and DX,
ADD AX, DX                      add them into AX

MOV CH, 1                       Move one into CH
ADD CH, 1                       Add one to CH

MOV DI, 1                       Move one into DI
MOV [SI], DI                    Copy it to DS:SI
ADD DI, [SI]                    Add DI and DS:SI

MOV BYTE PTR [SI+BX], 1         Put one at DS:(SI+BX)
INC BYTE PTR [SI+BX]            Add one to it
```

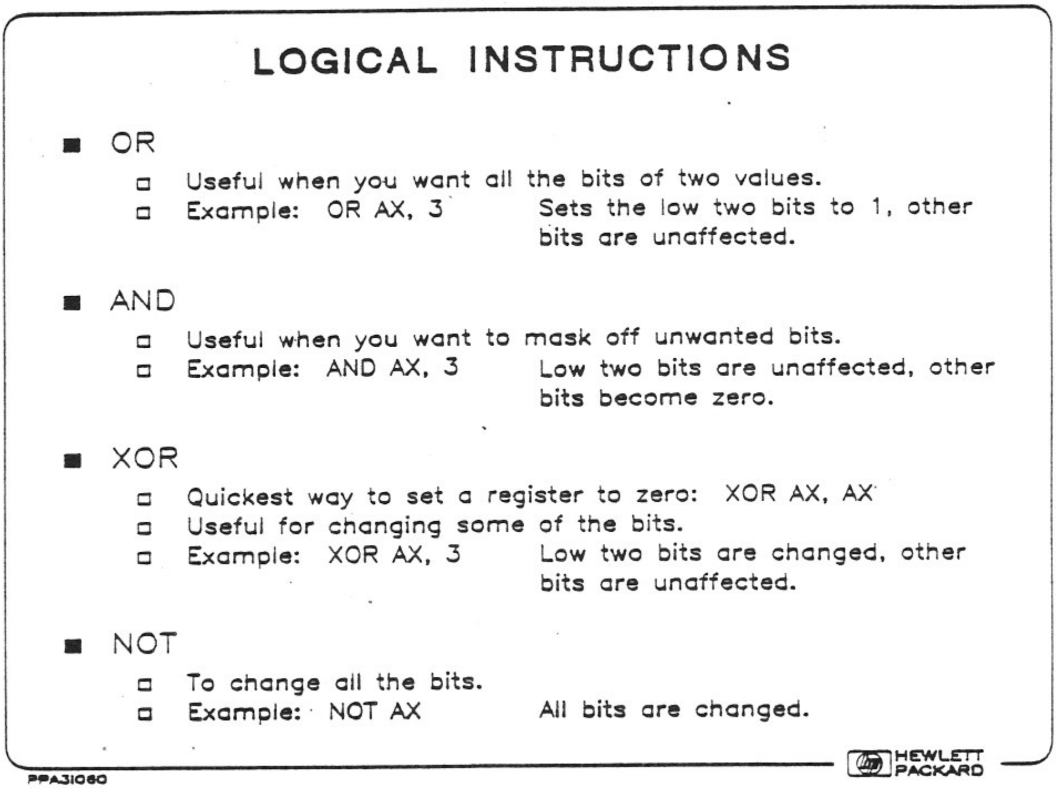
ISN'T THAT ENOUGH FOR NOW?

HEWLETT PACKARD

PPA3I050

Notes:

PPA3I050



LOGICAL INSTRUCTIONS

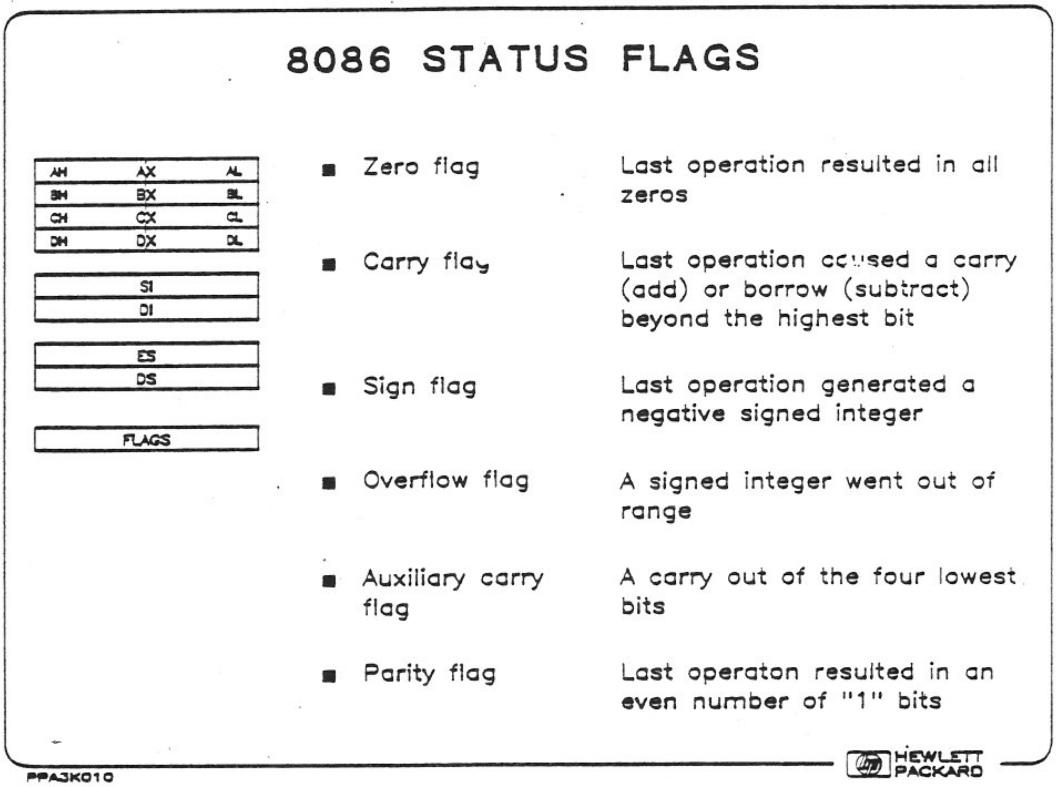
- OR
  - □ Useful when you want all the bits of two values.
  - □ Example: OR AX, 3 — Sets the low two bits to 1, other bits are unaffected.
- AND
  - □ Useful when you want to mask off unwanted bits.
  - □ Example: AND AX, 3 — Low two bits are unaffected, other bits become zero.
- XOR
  - □ Quickest way to set a register to zero: XOR AX, AX
  - □ Useful for changing some of the bits.
  - □ Example: XOR AX, 3 — Low two bits are changed, other bits are unaffected.
- NOT
  - □ To change all the bits.
  - □ Example: NOT AX — All bits are changed.

PPA3I060

HEWLETT PACKARD

Notes:

PPA3I060



8086 STATUS FLAGS

AH
AX
AL
BH
BX
BL
CH
CX
CL
DH
DX
DL
SI
DI
ES
DS
FLAGS

■ Zero flag	Last operation resulted in all zeros
■ Carry flag	Last operation caused a carry (add) or borrow (subtract) beyond the highest bit
■ Sign flag	Last operation generated a negative signed integer
■ Overflow flag	A signed integer went out of range
■ Auxiliary carry flag	A carry out of the four lowest bits
■ Parity flag	Last operaton resulted in an even number of "1" bits

HEWLETT PACKARD

PPA3K010

Notes:

PPA3K010

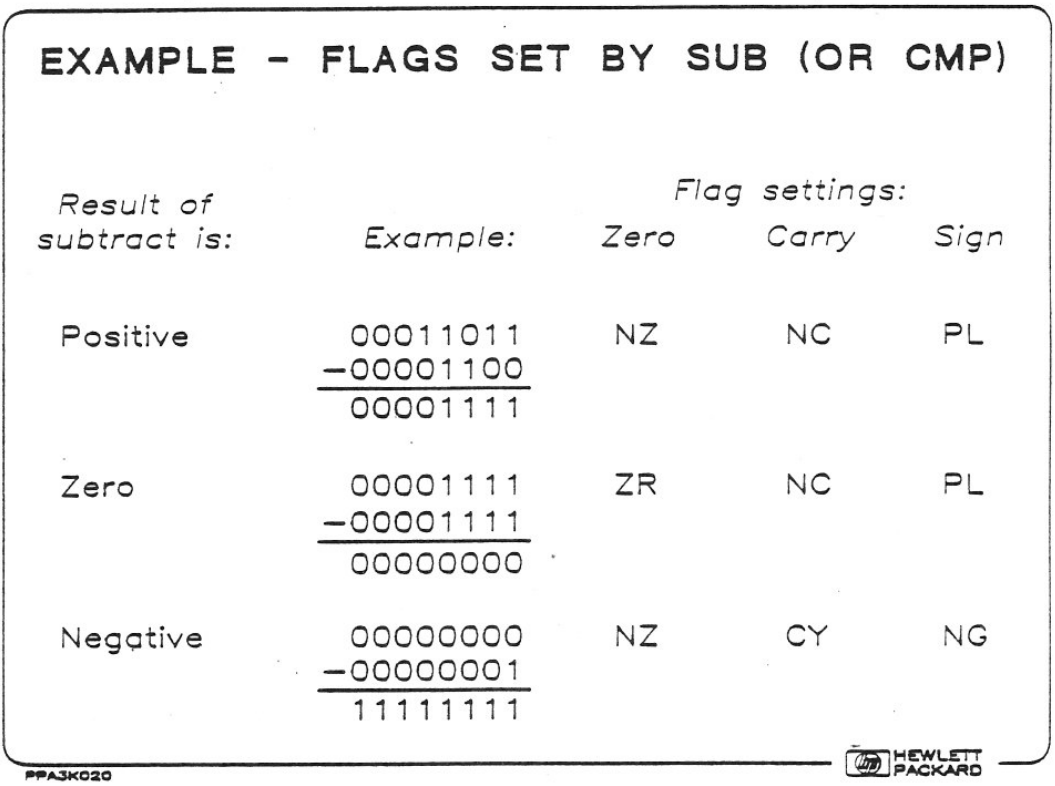
The flags register is strange in that only nine of the bits are used. Six of these bits give you summary information about the result of the last operation. The location of the bit is irrelevant, but the information it signifies may be useful.

The first two flags listed are the ones you are likely to use.

The zero and carry flags are used together to determine whether the result of an operation was positive, negative, or zero.

The carry flag is used to check for out-of-range positive integers (which may be an error, or just because you need to adjust the next-higher part of a multi-byte number).

Carry is also used to hold the spare bit moved out by some shift and rotate commands.



EXAMPLE - FLAGS SET BY SUB (OR CMP)

Result of subtract is:	Example:	Flag settings: Zero	Carry	Sign
Positive	00011011 −00001100 00001111	NZ	NC	PL
Zero	00001111 −00001111 00000000	ZR	NC	PL
Negative	00000000 −00000001 11111111	NZ	CY	NG

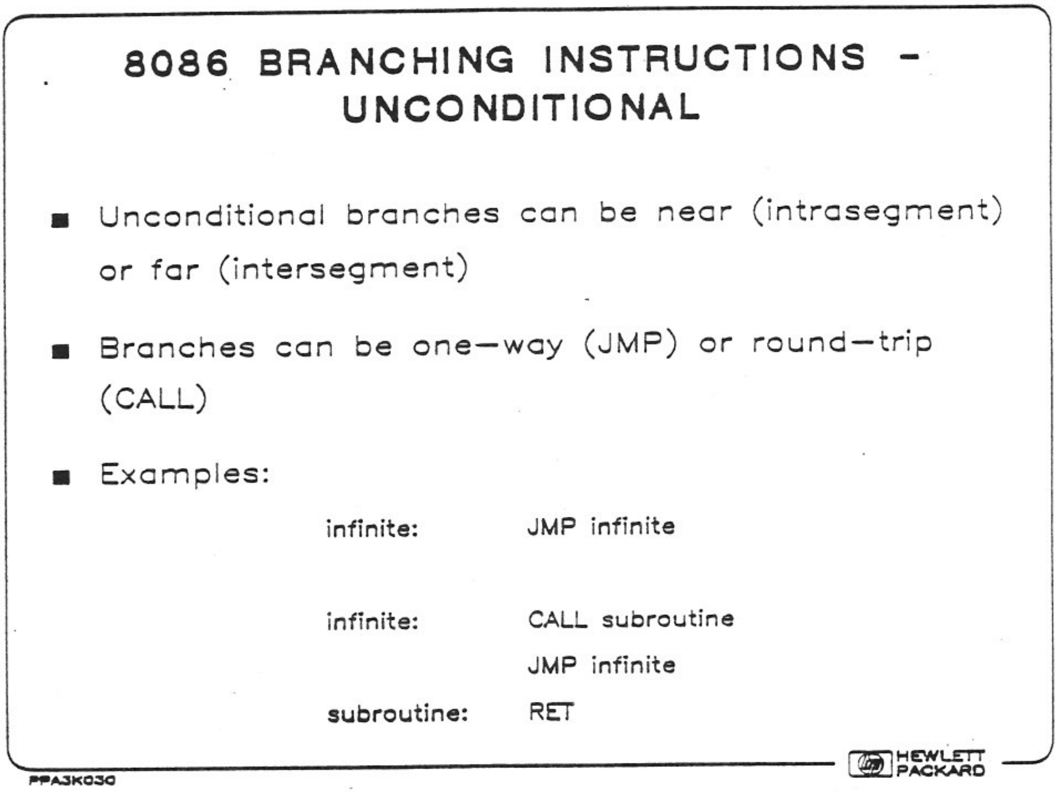
PPA3K020

Notes:

PPA3K020

When subtracting positive integers, the sign and carry flags contain redundant information. (This is not true of the ADD instruction or signed subtraction.)

Note that decrementing zero results in all ones--this can cause disasters unless you check for negative results. For example, you calculate an address which lands you 64K from where you should be!



8086 BRANCHING INSTRUCTIONS - UNCONDITIONAL

- Unconditional branches can be near (intrasegment) or far (intersegment)
- Branches can be one-way (JMP) or round-trip (CALL)
- Examples:

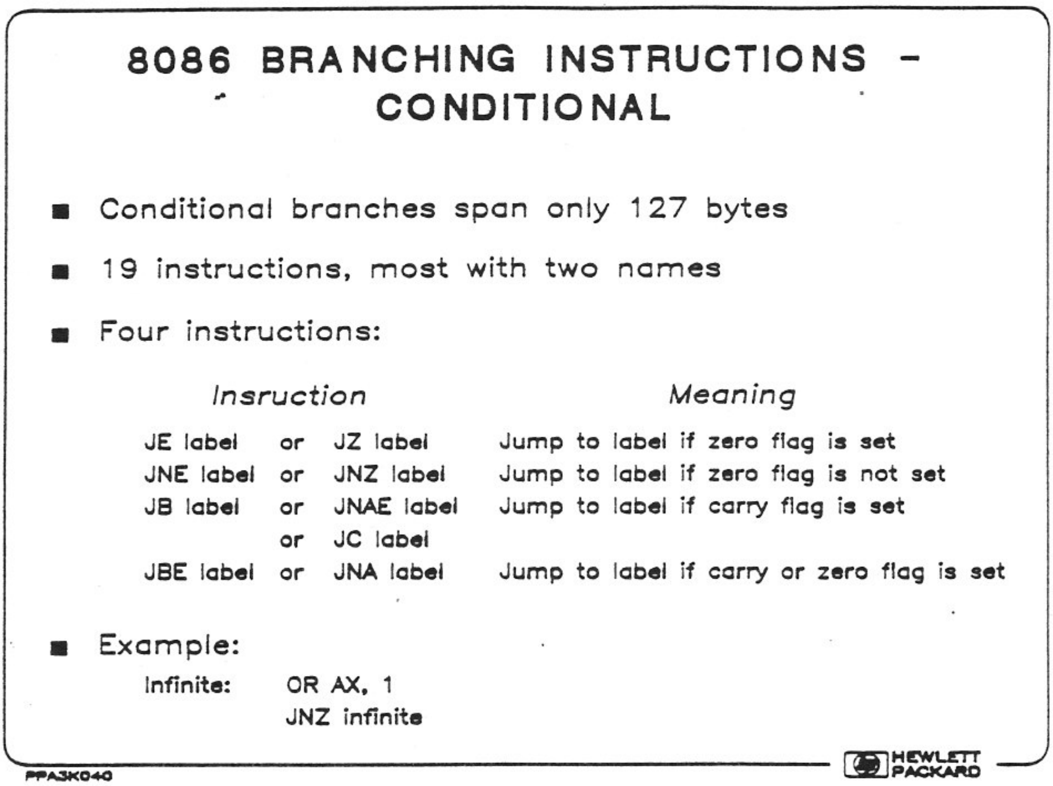
```
infinite:      JMP infinite


infinite:      CALL subroutine
               JMP infinite
subroutine:    RET
```

HEWLETT PACKARD

PPA3K030

Notes:

PPA3K030



8086 BRANCHING INSTRUCTIONS - CONDITIONAL

- Conditional branches span only 127 bytes
- 19 instructions, most with two names
- Four instructions:

Insruction			*Meaning*
JE label	or	JZ label	Jump to label if zero flag is set
JNE label	or	JNZ label	Jump to label if zero flag is not set
JB label	or	JNAE label	Jump to label if carry flag is set
	or	JC label	
JBE label	or	JNA label	Jump to label if carry or zero flag is set

- Example:

```
Infinite:   OR AX, 1
            JNZ infinite
```

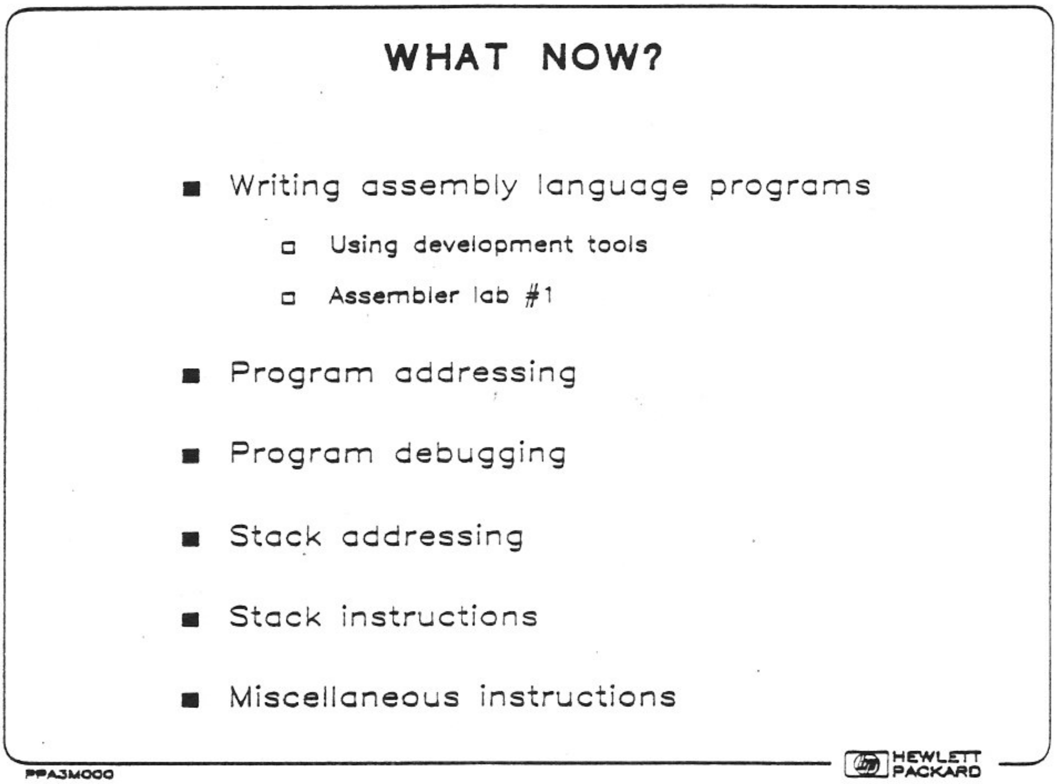
PPA3K040

Notes:

PPA3K040

Conditional branches allow you to test one or more flags and make a jump according to flag conditions.

Not all instructions set flags! For example, MOV does not.



WHAT NOW?

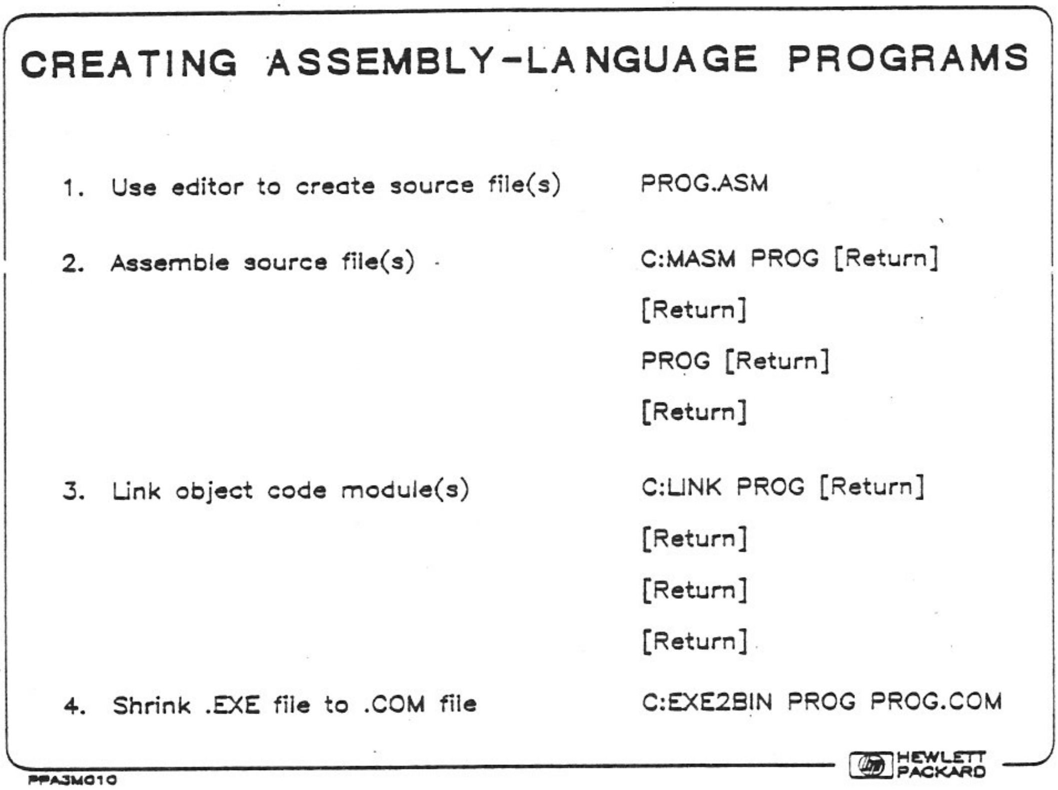
- Writing assembly language programs
  - □ Using development tools
  - □ Assembler lab #1
- Program addressing
- Program debugging
- Stack addressing
- Stack instructions
- Miscellaneous instructions

PPA3M000

HEWLETT PACKARD

Notes:

PPA3M000



CREATING ASSEMBLY-LANGUAGE PROGRAMS

1. Use editor to create source file(s)	PROG.ASM
2. Assemble source file(s)	C:MASM PROG [Return]
	[Return]
	PROG [Return]
	[Return]
3. Link object code module(s)	C:LINK PROG [Return]
	[Return]
	[Return]
	[Return]
4. Shrink .EXE file to .COM file	C:EXE2BIN PROG PROG.COM

HEWLETT PACKARD

PPA3M010

Notes:

PPA3M010

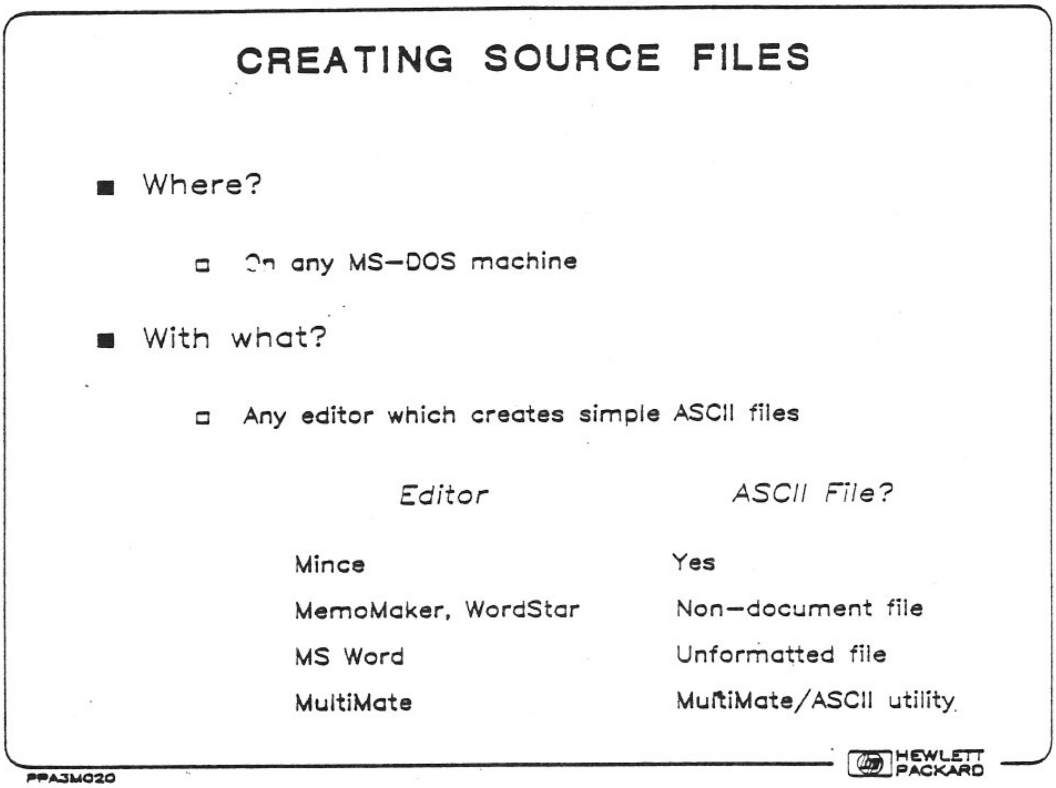
A source-code file is a simple ASCII file which contains:

- Directives - information for the assembler
- Instructions - 8086 instructions
- Comments - information for the programmer

The macroassembler takes the source file and attempts to convert it to machine instructions--an object file.

The linker takes one or more object files and puts them in executable form--an .EXE file. Ignore the "Missing stack segment" error.

If the assembly code is a stand-alone program, it will take less space as a binary program. EXE2BIN strips off some miscellaneous overhead and shrinks it to a .COM program. (Default result is a .BIN file which must be renamed .COM to run.)



CREATING SOURCE FILES

- Where?
  - □ On any MS-DOS machine
- With what?
  - □ Any editor which creates simple ASCII files

Editor	*ASCII File?*
Mince	Yes
MemoMaker, WordStar	Non-document file
MS Word	Unformatted file
MultiMate	MultiMate/ASCII utility

PPA3M020

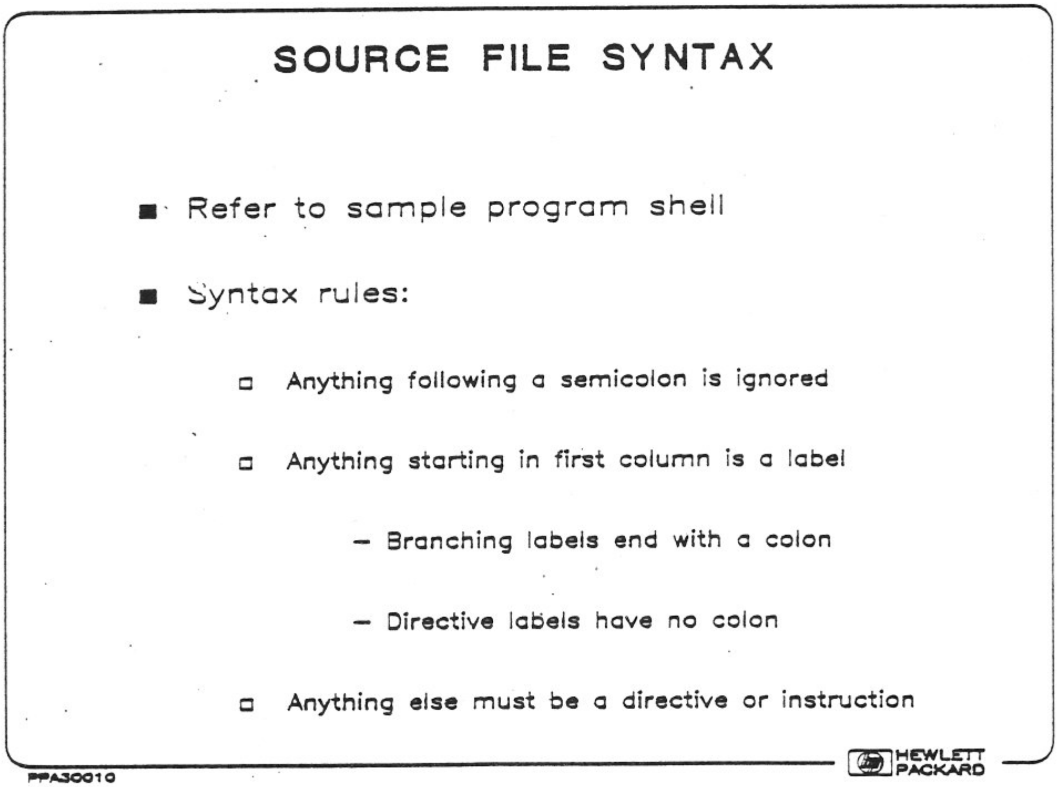
HEWLETT PACKARD

Notes:

PPA3M020

See next page.

```
        ;                  SAMPLE ASSEMBLY LANGUGAE PROGRAM SHELL
        ;                  ======================================
        ;
        ;
        PAGE 60,132               ;determines page height and width of
        ;                         .LST file
        ;
        TITLE whatever title you want on each page of .LST printout
        ;
CSEG    SEGMENT                   ;labels beginning of code segment
        ;
        ASSUME CS:CSEG, DS:CSEG   ;both CS and DS point to beginning
        ;                         of program
        ;
        ;     <-----------------  Define all constants here with the
        ;                         EQU statement (labels to be used instead
        ;                         of numbers for programming convenience).
        ;
        ;
        ORG 100H                  ;program entry point is CS:100h
        ;
START   PROC FAR                  ;this is the entry point
        ;
        ;
        ;     <-----------------  Program code goes here.
        ;
        ;
DONE:   MOV AX,4C00H              ;jumping to DONE sends you back to DOS
        INT 21H
        ;
START   ENDP                      ;defines the end of main program
        ;
        ;
        ;     <-----------------  Subroutines (accessed with the CALL
        ;                         statement) go here.  The RET statement
        ;                         sends you back to the line following
        ;                         the CALL.
        ;
        ;
CSEG    ENDS                      ;defines the end of code segment
        ;
        END START                 ;defines the end of program file
```



SOURCE FILE SYNTAX

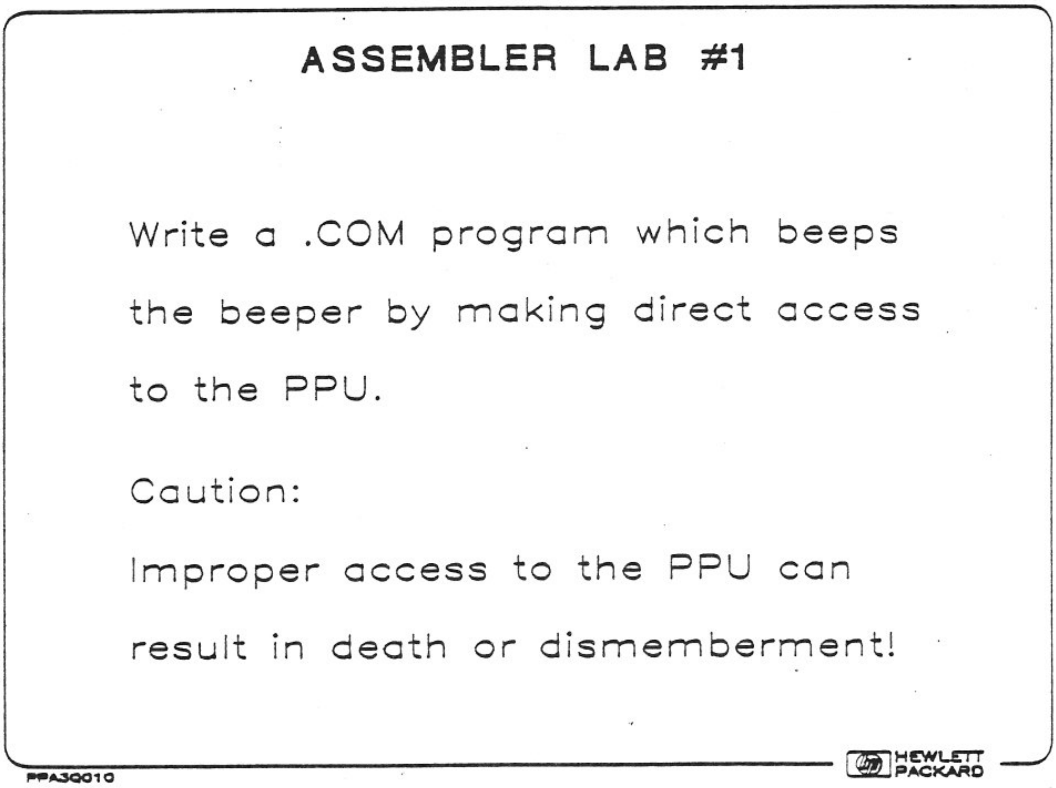
- Refer to sample program shell
- Syntax rules:
  - □ Anything following a semicolon is ignored
  - □ Anything starting in first column is a label
    - – Branching labels end with a colon
    - – Directive labels have no colon
  - □ Anything else must be a directive or instruction

PPA30010

HEWLETT PACKARD

Notes:

PPA30010



ASSEMBLER LAB #1

Write a .COM program which beeps the beeper by making direct access to the PPU.

Caution:

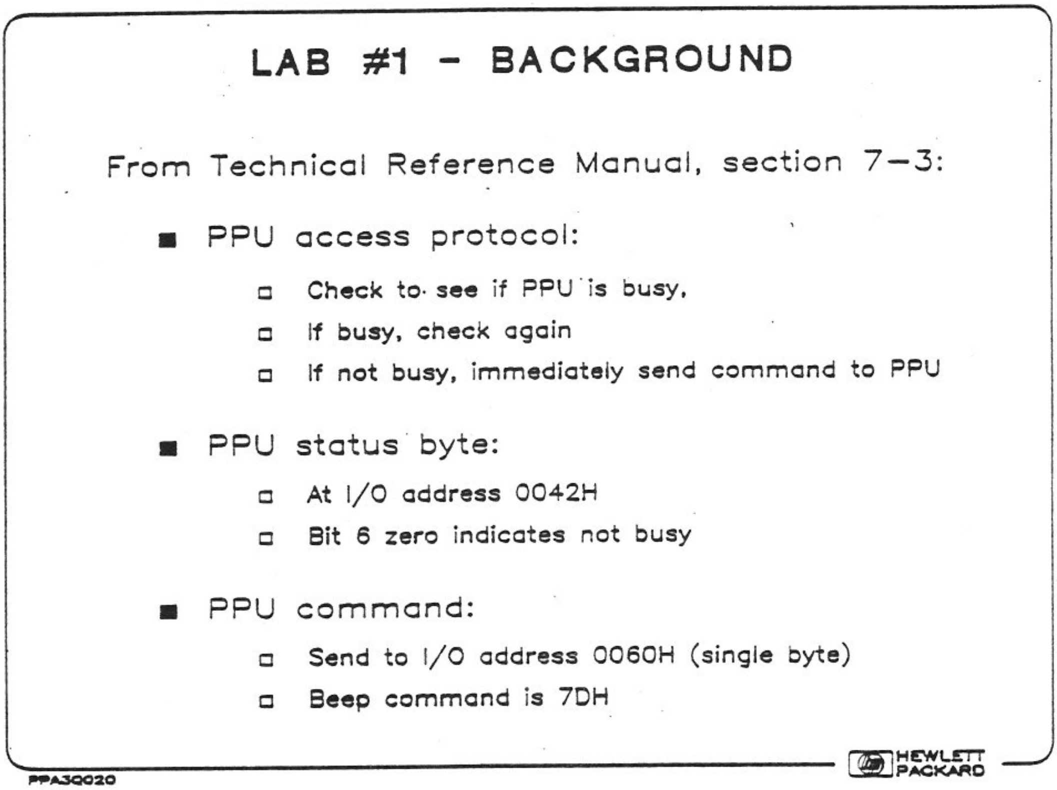
Improper access to the PPU can result in death or dismemberment!

HEWLETT PACKARD

PPA3Q010

Notes:

PPA3Q010



LAB #1 - BACKGROUND

From Technical Reference Manual, section 7-3:

- PPU access protocol:
  - □ Check to see if PPU is busy,
  - □ If busy, check again
  - □ If not busy, immediately send command to PPU
- PPU status byte:
  - □ At I/O address 0042H
  - □ Bit 6 zero indicates not busy
- PPU command:
  - □ Send to I/O address 0060H (single byte)
  - □ Beep command is 7DH

HEWLETT PACKARD

PPA3Q020

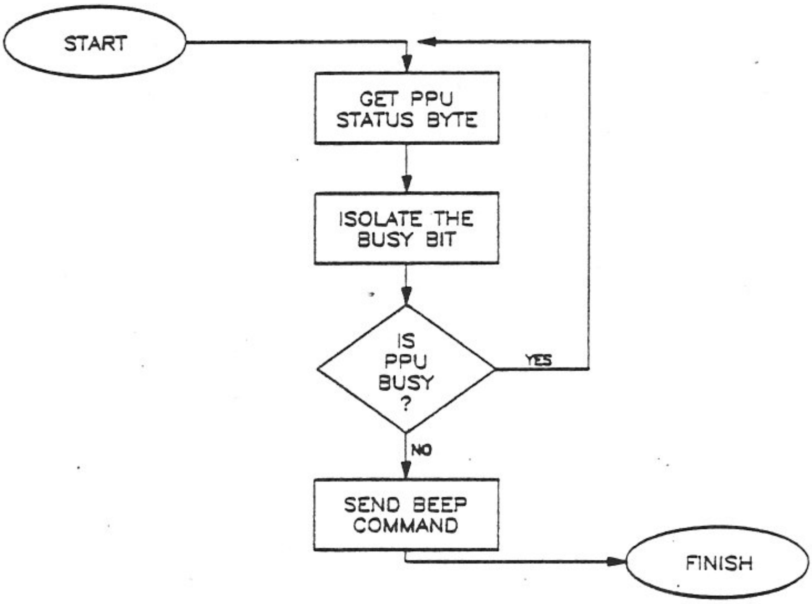
Notes:

PPA3Q020

The PPU is periodically busy doing various chores--clock update, prior beep, etc. It cannot accept commands while busy--you could send it out to lunch. When not busy, the PPU clears its busy status bit and will accept any command which comes in during a (short) time window.

Because the status check must be followed quickly by the command, you cannot safely beep the beeper from DEBUG.

LAB #1 - FLOWCHART



PPA3Q030

Notes:

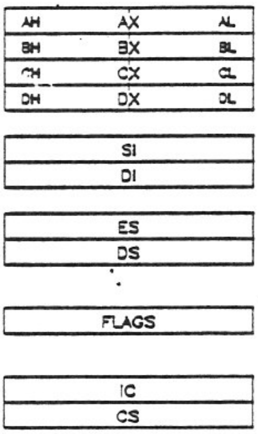
PPA3Q030

Handout

Notes:

PPA3Q040

ADDRESSING OF PROGRAM INSTRUCTIONS



- Two registers dedicated to program addressing:
  - □ Code segment
  - □ Instruction counter
- Next instruction is at CS:IC
- Near jumps alter IC
- Far jumps alter both CS and IC

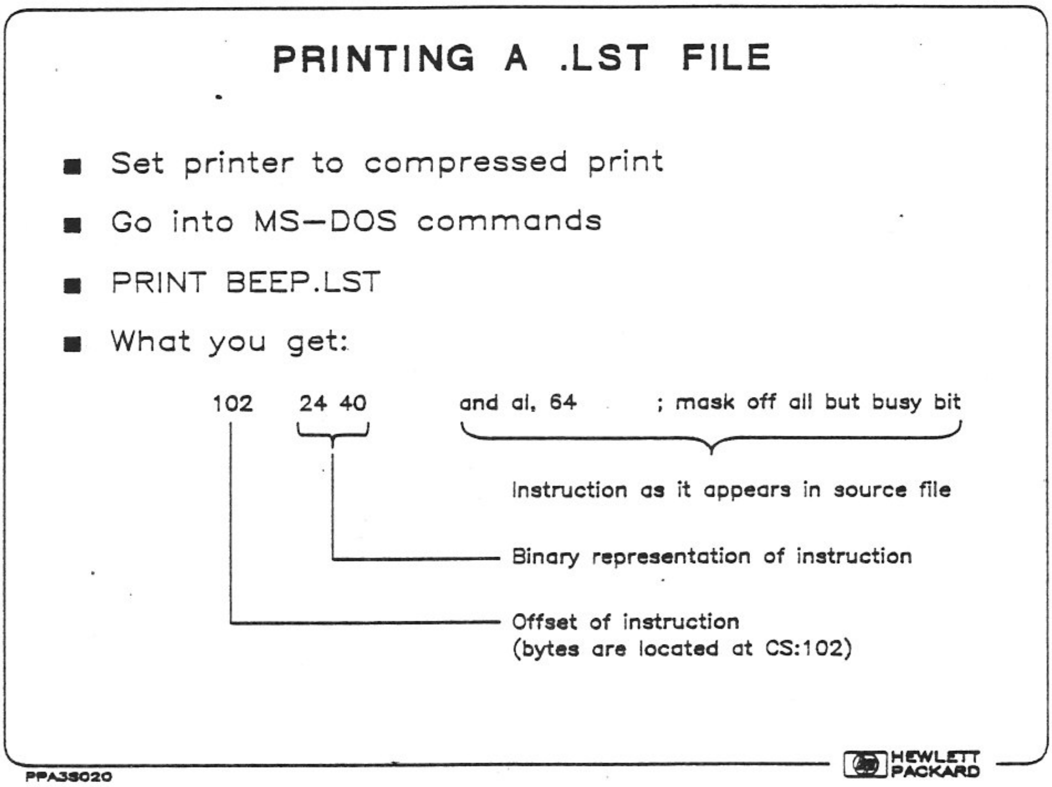
PPA3S010

Notes:

PPA3S010

The 8086 keeps track of its location in a program so it knows where to fetch the next instruction.

The instruction counter is also known as program counter. (Debug uses IC.)



PRINTING A .LST FILE

- Set printer to compressed print
- Go into MS–DOS commands
- PRINT BEEP.LST
- What you get:

```
102    24 40       and al, 64       ; mask off all but busy bit
```

Instruction as it appears in source file

Binary representation of instruction

Offset of instruction
(bytes are located at CS:102)

PPA3S020

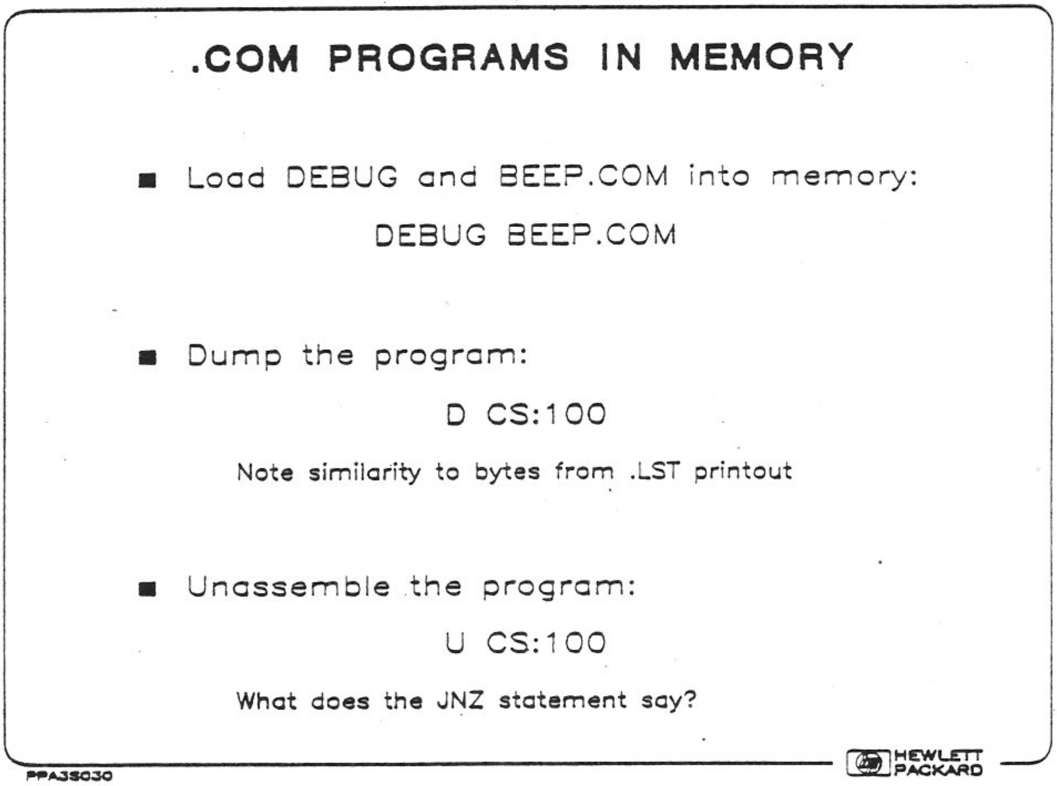
HEWLETT PACKARD

Notes:

PPA3S020

The .LST file which is optionally created by MASM is useful for debugging. Use MS-DOS PRINT to print it because lines are over 80 columns.

Instruction offsets and the instructions are shown in hex form. (Debug uses only hex, so this is appropriate.)



.COM PROGRAMS IN MEMORY

- Load DEBUG and BEEP.COM into memory:

  DEBUG BEEP.COM

- Dump the program:

  D CS:100

  Note similarity to bytes from .LST printout

- Unassemble the program:

  U CS:100

  What does the JNZ statement say?

PPA3S030

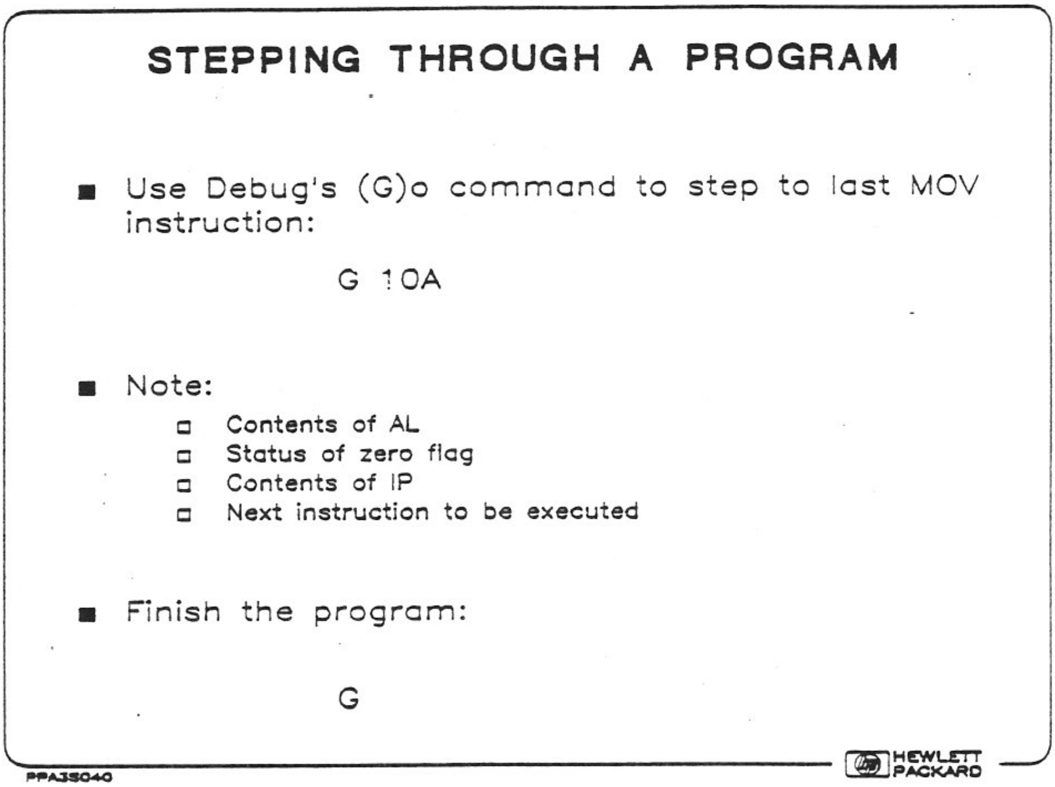
Notes:

PPA3S030

In a .COM program, the first 100h bytes are reserved for use by MS-DOS, so the program begins at CS:100.

A dump at CS:100 shows the instructions in less-comprehensible form.

Unassembling at CS:100 puts the instructions in more meaningful form, but instead of labels, you get offsets.



STEPPING THROUGH A PROGRAM

- Use Debug's (G)o command to step to last MOV instruction:

```
G 10A
```

- Note:
  - Contents of AL
  - Status of zero flag
  - Contents of IP
  - Next instruction to be executed

- Finish the program:

```
G
```

HEWLETT PACKARD

PPA3S040

Notes:

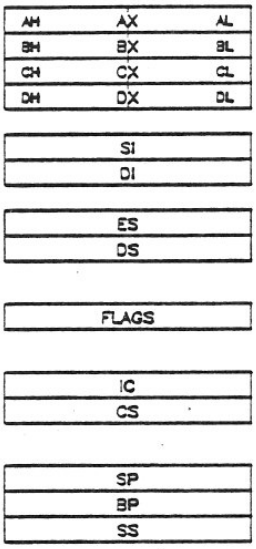
PPA3S040

Stepping through a program demonstrates the real power of DEBUG. This is done with (G)o or (T)race. Trace is for single-stepping; Go can go to anywhere.

The offset used with Go must be exact--using an invalid offset will not stop the program and may trash the byte specified.

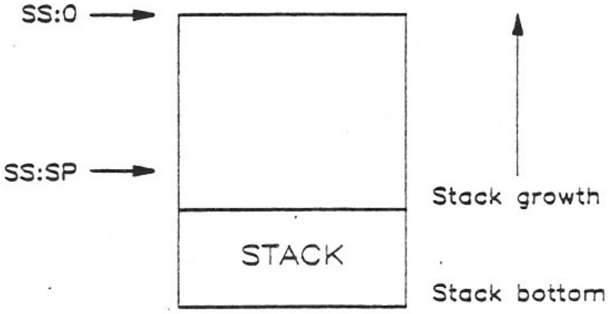
At the breakpoint, DEBUG shows the contents of all registers, the status of flags, and the next instruction which has not yet been executed.

8086 STACK ADDRESSING



Stack registers:

- Stack segment
- Stuck pointer
- Base pointer



PPA3U010

HEWLETT PACKARD

Notes:

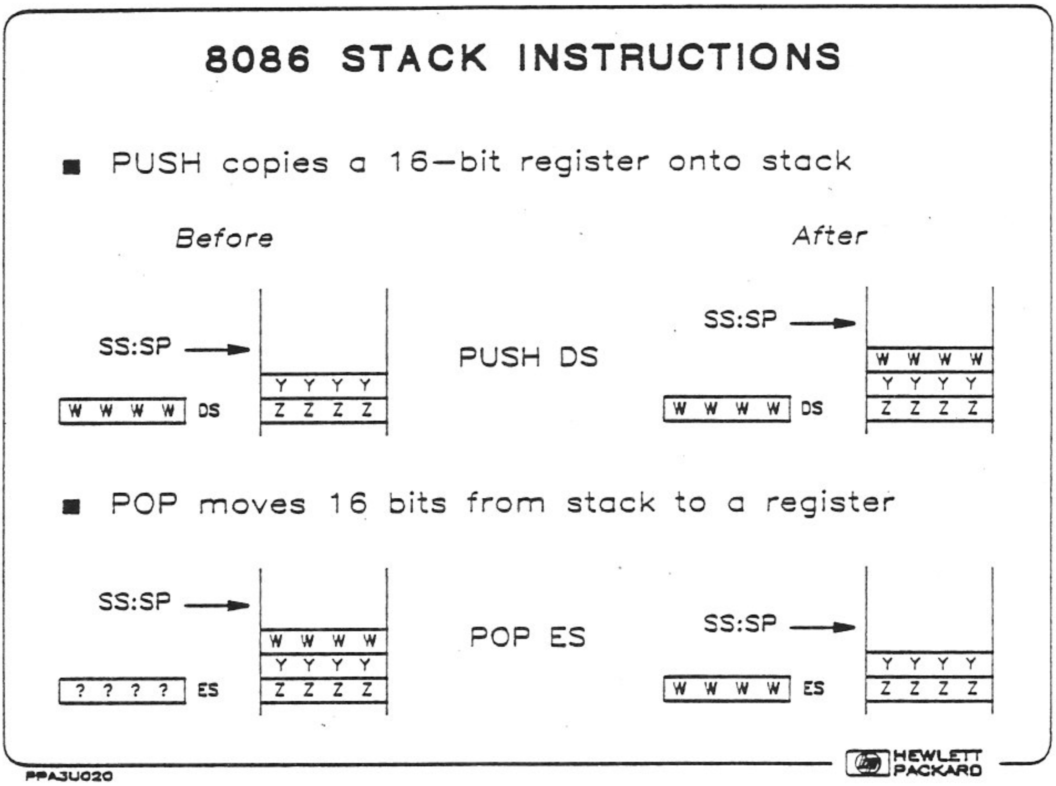
PPA3U010

The three stack registers are the last 8086 registers to be discussed. They define a storage area in memory to be used by the operating system and programs for temporary storage. Data is piled on (pushed) or lifted off (popped) the stack top.

The stack grows backwards through memory addresses. All data on the stack is in 16-bit chunks.

In a .COM program, SS=CS and SP is initialized as FFFE hex. Hence, a .COM program starts with 64K of memory, with the program in the low address end and the stack growing backwards through memory from 64K out. The stack can be relocated elsewhere if this isn't convenient.

The base pointer allows you to access the stack without removing anything from it. For example, MOV AX, [BP] gets a word from SS:BP.



8086 STACK INSTRUCTIONS

- PUSH copies a 16-bit register onto stack

- POP moves 16 bits from stack to a register

PPA3U020

HEWLETT PACKARD

Notes:

PPA3U020

Push moves a word to SS:SP, then subtracts two from SP.

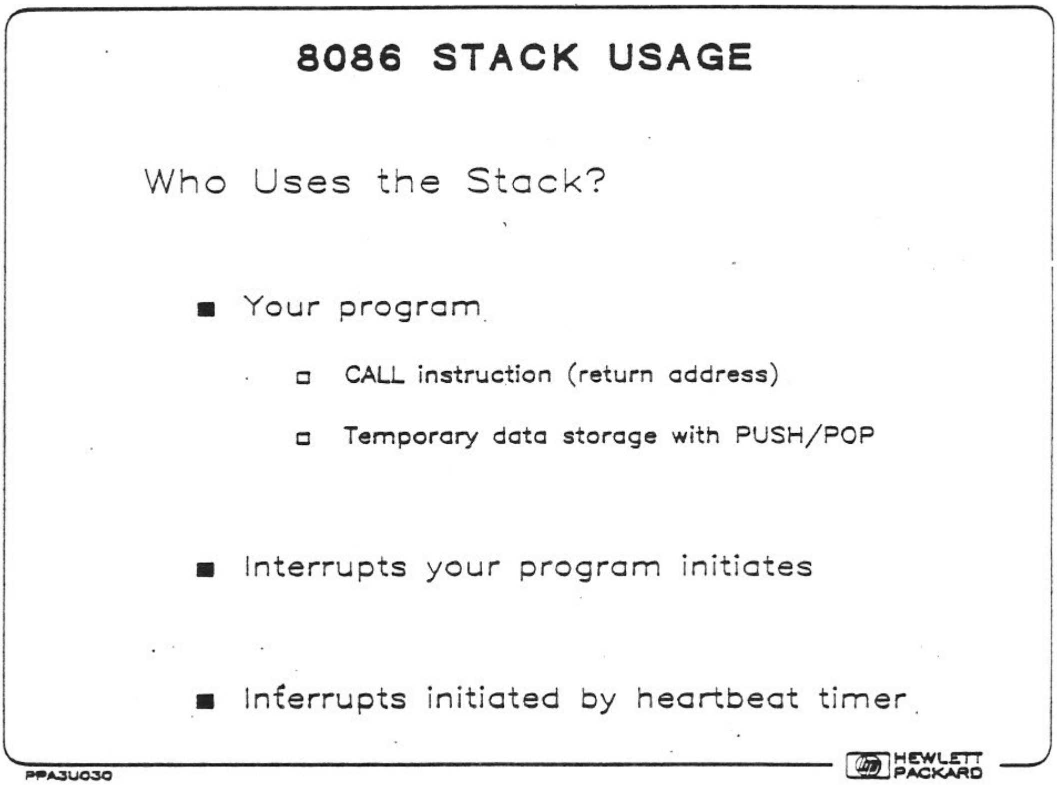
Pop adds two to SP, then moves a word from SS:SP.

There is also PUSHF and POPF for moving the flags onto and off of the stack.

The sequence

```
PUSH DS
POP ES
```

is common with segment registers because the MOV instruction cannot move directly from one to another.



8086 STACK USAGE

Who Uses the Stack?

- Your program
  - □ CALL instruction (return address)
  - □ Temporary data storage with PUSH/POP
- Interrupts your program initiates
- Interrupts initiated by heartbeat timer

PPA3U030

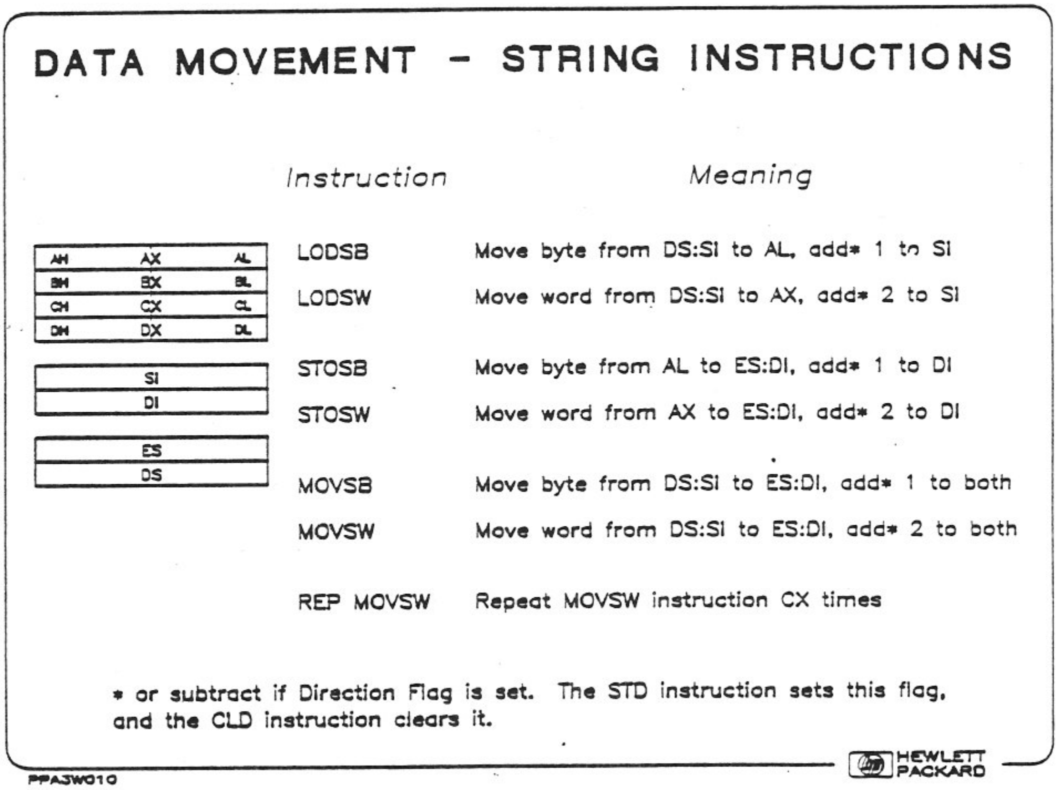
HEWLETT PACKARD

Notes: PPA3U030

Because subroutine calls keep their address on the stack, the subroutine must clean up the stack before executing a return.

MS-DOS interrupts (to be discussed later) use the stack.

The heartbeat generates a hardware interrupt 18 times per second and will use the stack. This will happen at any point in the program but is not a concern unless the stack is too small.



DATA MOVEMENT - STRING INSTRUCTIONS

AH	AX	AL
BH	BX	BL
CH	CX	CL
DH	DX	DL

SI
DI

ES
DS

Instruction	*Meaning*
LODSB	Move byte from DS:SI to AL, add* 1 to SI
LODSW	Move word from DS:SI to AX, add* 2 to SI
STOSB	Move byte from AL to ES:DI, add* 1 to DI
STOSW	Move word from AX to ES:DI, add* 2 to DI
MOVSB	Move byte from DS:SI to ES:DI, add* 1 to both
MOVSW	Move word from DS:SI to ES:DI, add* 2 to both
REP MOVSW	Repeat MOVSW instruction CX times

* or subtract if Direction Flag is set. The STD instruction sets this flag, and the CLD instruction clears it.

PPA3W010

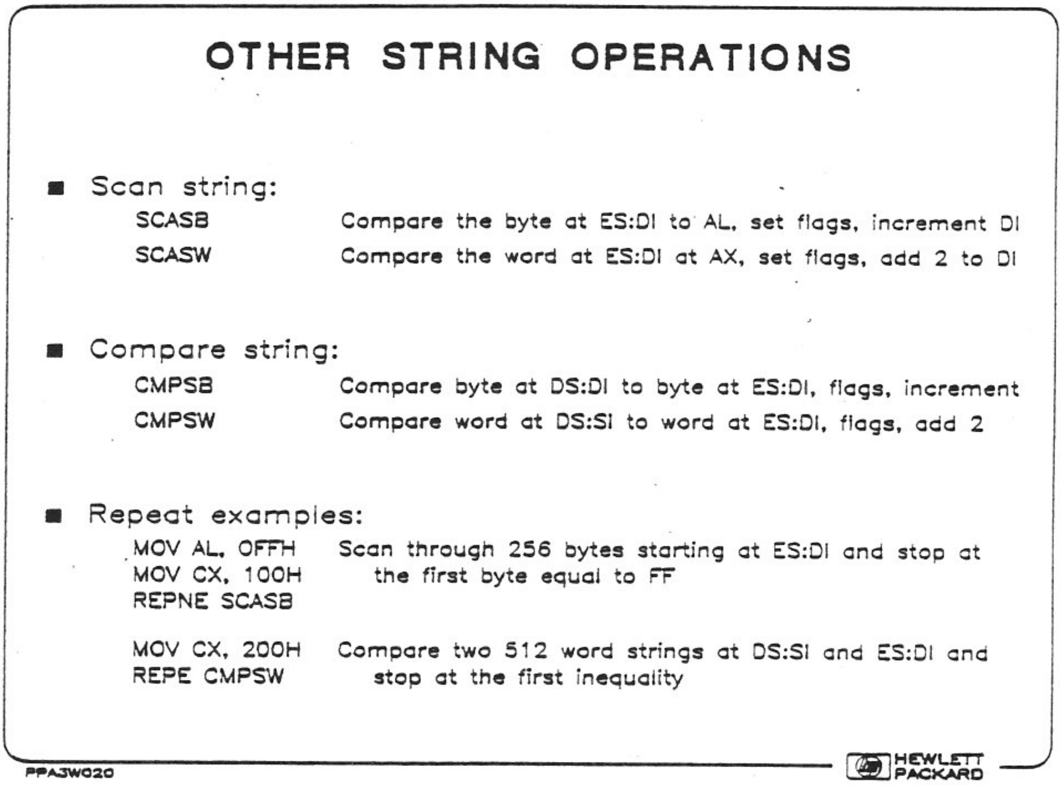
Notes:

PPA3W010

String movement instructions are useful for repeated access to the same memory area because of the autoincrement/autodecrement feature.

The MOVS instruction is the only memory-to-memory move instruction.

Any string instruction can use the REPeat prefix.



OTHER STRING OPERATIONS

- Scan string:

  SCASB   Compare the byte at ES:DI to AL, set flags, increment DI

  SCASW   Compare the word at ES:DI at AX, set flags, add 2 to DI

- Compare string:

  CMPSB   Compare byte at DS:DI to byte at ES:DI, flags, increment

  CMPSW   Compare word at DS:SI to word at ES:DI, flags, add 2

- Repeat examples:

```
MOV AL, OFFH     Scan through 256 bytes starting at ES:DI and stop at
MOV CX, 100H        the first byte equal to FF
REPNE SCASB

MOV CX, 200H     Compare two 512 word strings at DS:SI and ES:DI and
REPE CMPSW          stop at the first inequality
```

PPA3W020

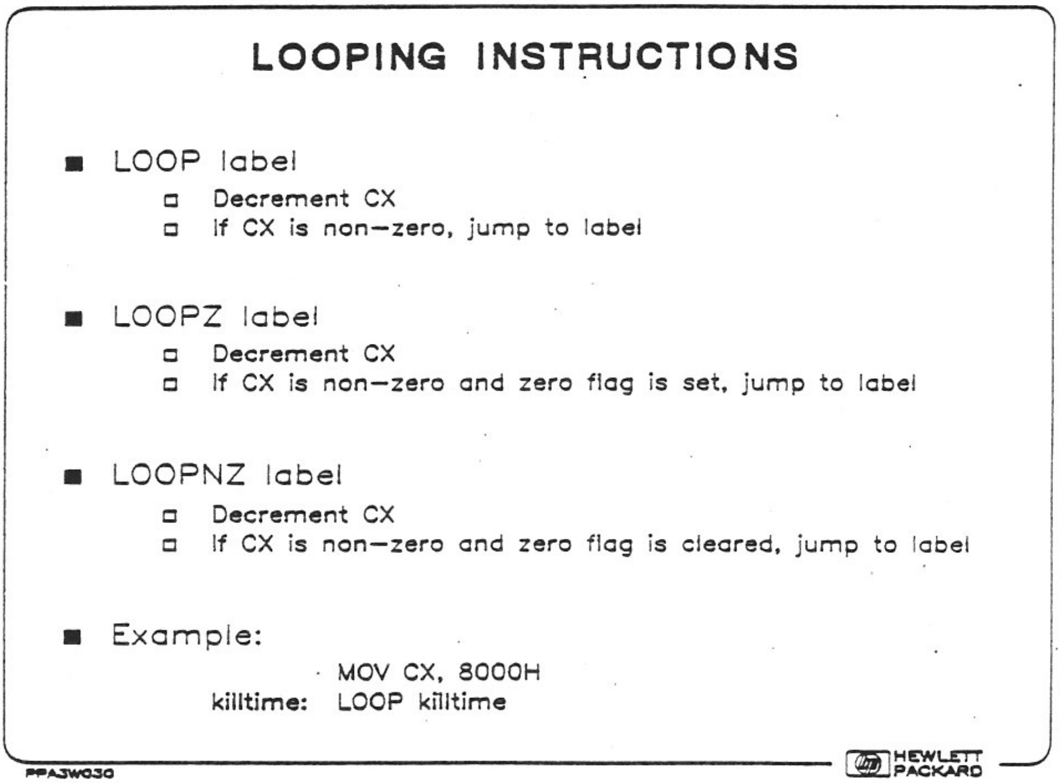
Notes:

PPA3W020

Scan string allows you to search for a particular byte in a block of memory.

Compare string allows you to test strings for equality.

REPNE = repeat while not equal (same as REPNZ)
REPE = repeat while equal (same as REPZ)



LOOPING INSTRUCTIONS

- LOOP label
  - □ Decrement CX
  - □ If CX is non—zero, jump to label
- LOOPZ label
  - □ Decrement CX
  - □ If CX is non—zero and zero flag is set, jump to label
- LOOPNZ label
  - □ Decrement CX
  - □ If CX is non—zero and zero flag is cleared, jump to label
- Example:

```
          MOV CX, 8000H
killtime: LOOP killtime
```

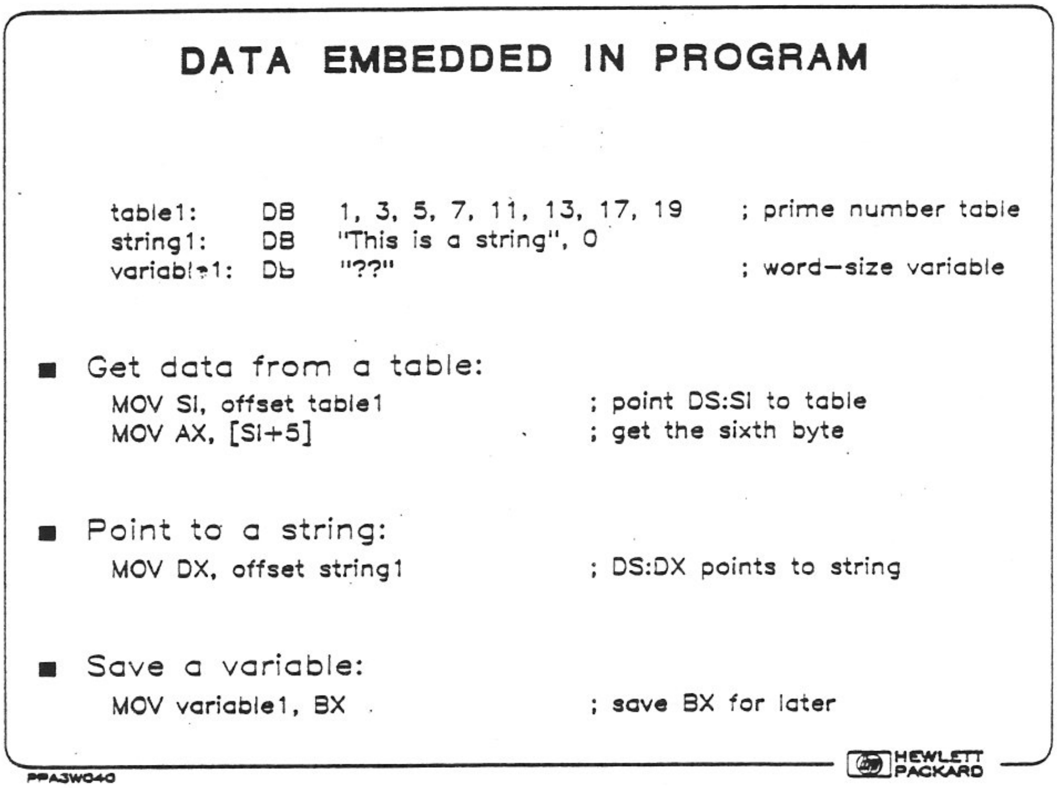
PPA3W030

HEWLETT PACKARD

Notes:

PPA3W030

The loop instructions are the most powerful of the conditional branching instructions. They combine a decrement of the counter register with a test for a zero count and can also test the zero flag at the same time.



DATA EMBEDDED IN PROGRAM

```
table1:     DB   1, 3, 5, 7, 11, 13, 17, 19      ; prime number table
string1:    DB   "This is a string", 0
variable1:  DB   "??"                            ; word-size variable
```

- Get data from a table:

```
MOV SI, offset table1                ; point DS:SI to table
MOV AX, [SI+5]                       ; get the sixth byte
```

- Point to a string:

```
MOV DX, offset string1               ; DS:DX points to string
```

- Save a variable:

```
MOV variable1, BX                    ; save BX for later
```

PPA3W040

HEWLETT PACKARD

Notes:

PPA3W040

Embedded data can be anywhere in a program; simply be sure to jump around it.

The define byte directive is one of several used to place literal bytes into a program. Numbers denote byte values, while quoted strings are assembled as ASCII character codes.

When "offset" is used, the assembler replaces it with the offset of the label given.