Assembly Language in 8086 Memory Architecture

Assembly Language Programs — 1

Assembly language program
List of 8086 instructions + data
CPU executes — one by one — instructions in order of listing
Example:

MOV AX, BX
ADD AX, CX
SUB AX, [BX+1234]
PUSH AX
MOV AH, 41
INT 21

Running this program requires CPU access to memory:
Code segment access ([CS:IP]) to fetch instructions
Data segment access ([DS:BX+1234]) to load operand
Stack segment access ([SS:SP]) to execute push

Assembly Language Programs — 2

Assembly language converted to machine language

89D8 MOV AX,BX
01C8 ADD AX,CX
2B873412 SUB AX,[BX+1234]
50 PUSH AX
B441 MOV AH,41
CD21 INT 21

Machine language (*.com) version of program
Machine code = string of hexadecimal codes

89D801C82B87341250B441CD21
Assembly Language Programs — 3

OS loads program to 20-bit physical address
OS loads registers

SS register ← Stack base address / 10h
Copy stack to memory from address SS:SP

CS register ← Code base address / 10h
IP register ← Code base address % 10h
Copy code to memory from address CS:IP

89D801C82B87341250B441CD21

ES = DS register ← Data base address / 10h
Copy data to memory from address DS:EA

Assembly Language Programs — 4

Assign default values to general registers
CX ← size of program (bytes)
AX = BX = DX = SI = DI = BP ← 0

Load initial values for program registers
IP ← offset to first instruction in code segment
SP ← offset to first location in stack segment

Run program
Fetch machine instructions from code segment (CS:IP)
Access data and stack

OS services (interventions)
Loading program into memory
Assigning initial values for registers in 8086
Adjusting pointers stored in program’s data segment

Assembly Language Programs — 5

DOS program files
Stored on disk or other long term storage
Contains

HEADER (only in *.exe files):
Information for operating system

LOAD MODULE (in *.exe, *.com, *.sys files):
Data , Code, Stack part of program

Loading and Running
OS copies load module into memory
OS initializes registers
OS adjusts stored pointers (if file has header)
OS points CS:IP at program start instruction
Assembly Language Programs — 6

**Data locations**
Transfer and ALU instructions refer (default) to data segment (DS)
- Execute `SUB AX, [BX+1234]`
- Calculate Effective Address (EA) `BX+1234`
- Load operand from data segment address `DS:BX+1234`
- Perform `AX ← AX - [DS:BX+1234]`
- Program must know where data is stored

**Code locations**
Instructions stored in memory in order of program list
- OS sets `IP` to first instruction in `CS`
- CPU updates `IP` after loading an instruction
  - `IP ← address of next instruction`
- Program only needs to know `CS:IP` for branch instructions

Assembly Language Programs — 7

**Branch instructions**
A simple `JUMP` (skip over `SUB` and `PUSH`) looks like:
- `MOV AX, BX`
- `ADD AX, CX`
- `JMP L1`
- `SUB AX, [BX+1234]`; fall-through instruction
- `PUSH AX`
- `L1: MOV AH, 41`; target instruction
- `INT 21`

- `JMP` assembly coded with **pointer** `L1` to line `MOV AH, 41h`
  - Pointer is address of target instruction
- `JMP` machine coded with **displacement** to line `MOV AH, 41`
  - Displacement is target address – fall-through address

Assembly Language Programs — 8

**Branch instructions**
Program listing with instruction lengths:
- `MOV AX, BX`
- `ADD AX, CX`
- `JMP L1`; `IP ← IP + 5`
- `SUB AX, [BX+1234]`; fall-through
- `PUSH AX`
- `L1: MOV AH, 41`; target
- `INT 21`

- `JMP` instruction **ASSEMBLY CODED** with `IP` of target
- `JMP` instruction **MACHINE CODED** with `displacement`
  - `displacement = 4 + 1 = 5`

The Assembly Language Programming Process

<table>
<thead>
<tr>
<th>Stage</th>
<th>User Action</th>
<th>Detailed Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Top-down design</strong></td>
<td>Plan</td>
<td>1. High level design&lt;br&gt;2. Organization into modules and functions</td>
</tr>
<tr>
<td><strong>Coding modules</strong></td>
<td>Program</td>
<td>1. Define local data structures&lt;br&gt;2. Write assembly language code for modules</td>
</tr>
<tr>
<td><strong>Assembly</strong></td>
<td>Run Assembler</td>
<td>1. Convert assembly code to machine code for modules&lt;br&gt;2. Resolve internal references</td>
</tr>
<tr>
<td><strong>Linking</strong></td>
<td>Run Linker</td>
<td>1. Resolve external address references for jump and call instructions&lt;br&gt;2. Build program header&lt;br&gt;3. Store executable program file on disk</td>
</tr>
<tr>
<td><strong>Loading</strong></td>
<td>Run Loader</td>
<td>1. Place program in memory&lt;br&gt;2. Set instruction pointer to program</td>
</tr>
</tbody>
</table>
Top-Down Design

**Define requirements for program**

**High Level Design**
- Divide problem into modules
  - Function — block of code to perform specific task
  - Module — source file containing one or more functions
  - Program — one or more modules

**Low Level Design**
- Code modules and their interfaces

**Graphical or other design technique**
- Flowchart
- Pseudo-code
- Use cases
- UML
- etc

---

Coding Program Modules

**Assign variables to memory locations**
- **VARIABLE NAME = SYMBOLIC LABEL** for memory address
- List variables and addresses in scratchpad
- Used at assembly/compile time
- **NOT** part of program listing

**Code ALU operations as assembly instructions**
- Reference to variable uses reference by pointer

**Example:** \( z = y + x \)

<table>
<thead>
<tr>
<th>Assignments</th>
<th>HEX Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \sim DS:0000 )</td>
<td></td>
</tr>
<tr>
<td>( y \sim DS:0002 )</td>
<td>; scratchpad</td>
</tr>
<tr>
<td>( z \sim DS:0004 )</td>
<td></td>
</tr>
</tbody>
</table>

**Coding**
- \( MOV \ AX, [0000] \) ; \( AX \leftarrow x \)
- \( ADD \ AX, [0002] \) ; \( AX \leftarrow AX + y \)
- \( MOV \ [0004], AX \) ; \( z \leftarrow AX \)

--------

Assembly by Hand — Coding Chart

**Enter instructions**
- Write complete assembly program
- List op-code (mnemonic) and operands
- Write line labels when necessary

**Calculate Addresses**
- Start with arbitrary start address

**Calculate HEX codes (machine instruction) for each line**
- Branch instructions require knowing addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>HEX Code</th>
<th>Label</th>
<th>Op-Code</th>
<th>Operand</th>
<th>Comments</th>
</tr>
</thead>
</table>

---

Example of Assembly Coding

Implement pseudocode example:

```plaintext
a = 3, b = -1, c = 0
while (a < 10){
    b = a * 2
    c = b + 7
    a++
}
end
```
**Assembly by Hand — Coding**

```
address | HEX code | label | op-code and operands | comment
---------|---------|-------|----------------------|---------
0000     | C70600000300 | MOV WORD [0000],3 | ; define a
0006     | C7060200FFFF | MOV WORD [0002],FFFF | ; define b
000C     | C70604000000 | MOV WORD [0004],0 | ; define c
8336E0000000 | loop: | CMP WORD [0000],A | ; CMP a == 10
7D       | JGE end | MOV AX,[0000] | ; AX ← a
A10000   | MOV AX,[0000] | AX ← a
D1E0     | SHL AX,1   | AX ← 2*AX
A30200   | MOV [0000],AX | b ← AX
050700   | ADD AX,7   | AX ← AX+7
A30400   | MOV [0004],AX | c ← AX
FF060000 | INC WORD [0000] | a ← a+1
EB       | JMP loop  | back to loop
C3       | end: | RET | ; return to DOS
```

```

**Enter op code, line labels, comments**
```

---

**Hand Assembly — Convert to Machine Code**

```
Enter hexadecimal code for each line
```

```
address | HEX code | label | op-code and operands | comment
---------|---------|-------|----------------------|---------
C70600000300 | MOV WORD [0000],3 | ; define a
C7060200FFFF | MOV WORD [0002],FFFF | ; define b
C70604000000 | MOV WORD [0004],0 | ; define c
8336E0000000 | loop: | CMP WORD [0000],A | ; CMP a == 10
7D       | JGE end | MOV AX,[0000] | ; AX ← a
A10000   | MOV AX,[0000] | AX ← a
D1E0     | SHL AX,1   | AX ← 2*AX
A30200   | MOV [0000],AX | b ← AX
050700   | ADD AX,7   | AX ← AX+7
A30400   | MOV [0004],AX | c ← AX
FF060000 | INC WORD [0000] | a ← a+1
EB       | JMP loop  | back to loop
C3       | end: | RET | ; return to DOS
```

**Address of Instruction in Memory**

```
Arbitrary start address
Default IP = 0000
Relative to code segment base (CS)
```

```
Count instruction length (bytes in each instruction)
Next address = previous address + previous instruction length
```

```
address | HEX code | label | op-code and operands | comment
---------|---------|-------|----------------------|---------
0000     | C70600000300 | MOV WORD [0000],3 | ; define a
0006     | C7060200FFFF | MOV WORD [0002],FFFF | ; define b
000C     | C70604000000 | MOV WORD [0004],0 | ; define c
0012     | MOV WORD [0004],0 | ; define c
000F     | 0007 | 0008 | 0009 | 000A | 000B
```

---

**Machine Language Instruction**

Translate op-code and operands to HEX Code

HEX Code format for each addressing mode
Translation based on Intel op-code tables

```
MOV WORD PTR [0000],0003
```

---

Branch codes incomplete:
Target addresses unresolved (not calculated)
Example of Assembly Coding — Unresolved

<table>
<thead>
<tr>
<th>address</th>
<th>HEX code</th>
<th>label</th>
<th>op-code and operands</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>C706000000300</td>
<td>MOV</td>
<td>WORD [0000],3</td>
<td>define a</td>
</tr>
<tr>
<td>0006</td>
<td>C706020000000</td>
<td>MOV</td>
<td>WORD [0002],FFFF</td>
<td>define b</td>
</tr>
<tr>
<td>000C</td>
<td>C706040000000</td>
<td>MOV</td>
<td>WORD [0004],0</td>
<td>define c</td>
</tr>
<tr>
<td>0012</td>
<td>83E000000A</td>
<td>loop:</td>
<td>CMP WORD [0000],A</td>
<td>CMP a == 10</td>
</tr>
<tr>
<td>0017</td>
<td>7D</td>
<td>JGE</td>
<td>end</td>
<td>if a&gt;=10 then 002D</td>
</tr>
<tr>
<td>0019</td>
<td>A10000</td>
<td>MOV</td>
<td>AX,[0000]</td>
<td>AX ← a</td>
</tr>
<tr>
<td>001C</td>
<td>D1E0</td>
<td>SHL</td>
<td>AX,1</td>
<td>AX ← 2*AX</td>
</tr>
<tr>
<td>001E</td>
<td>A30200</td>
<td>MOV</td>
<td>[0002],AX</td>
<td>b ← AX</td>
</tr>
<tr>
<td>0021</td>
<td>050700</td>
<td>ADD</td>
<td>AX,7</td>
<td>AX ← AX+7</td>
</tr>
<tr>
<td>0024</td>
<td>A30400</td>
<td>MOV</td>
<td>[0004],AX</td>
<td>c ← AX</td>
</tr>
<tr>
<td>0027</td>
<td>FF060000</td>
<td>INC</td>
<td>WORD [0000]</td>
<td>a ← a+1</td>
</tr>
</tbody>
</table>

Example of Assembly Coding — Resolving Code

<table>
<thead>
<tr>
<th>address</th>
<th>HEX code</th>
<th>label</th>
<th>op-code and operands</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>C706000000300</td>
<td>MOV</td>
<td>WORD [0000],3</td>
<td>define a</td>
</tr>
<tr>
<td>0006</td>
<td>C706020000000</td>
<td>MOV</td>
<td>WORD [0002],FFFF</td>
<td>define b</td>
</tr>
<tr>
<td>000C</td>
<td>C706040000000</td>
<td>MOV</td>
<td>WORD [0004],0</td>
<td>define c</td>
</tr>
<tr>
<td>0012</td>
<td>83E000000A</td>
<td>loop:</td>
<td>CMP WORD [0000],A</td>
<td>CMP a == 10</td>
</tr>
<tr>
<td>0017</td>
<td>7D</td>
<td>JGE</td>
<td>002D</td>
<td>if a&gt;=10 then 002D</td>
</tr>
<tr>
<td>0019</td>
<td>A10000</td>
<td>MOV</td>
<td>AX,[0000]</td>
<td>AX ← a</td>
</tr>
<tr>
<td>001C</td>
<td>D1E0</td>
<td>SHL</td>
<td>AX,1</td>
<td>AX ← 2*AX</td>
</tr>
<tr>
<td>001E</td>
<td>A30200</td>
<td>MOV</td>
<td>[0002],AX</td>
<td>b ← AX</td>
</tr>
<tr>
<td>0021</td>
<td>050700</td>
<td>ADD</td>
<td>AX,7</td>
<td>AX ← AX+7</td>
</tr>
<tr>
<td>0024</td>
<td>A30400</td>
<td>MOV</td>
<td>[0004],AX</td>
<td>c ← AX</td>
</tr>
<tr>
<td>0027</td>
<td>FF060000</td>
<td>INC</td>
<td>WORD [0000]</td>
<td>a ← a+1</td>
</tr>
</tbody>
</table>

Resolving branch references

Line label = symbolic reference to instruction address
Replace line labels in code with IP address of target line

loop → 0012
end → 002D

Example — Object Code with Resolved References

<table>
<thead>
<tr>
<th>address</th>
<th>HEX code</th>
<th>label</th>
<th>op-code and operands</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>C706000000300</td>
<td>MOV</td>
<td>WORD [0000],3</td>
<td>define a</td>
</tr>
<tr>
<td>0006</td>
<td>C706020000000</td>
<td>MOV</td>
<td>WORD [0002],FFFF</td>
<td>define b</td>
</tr>
<tr>
<td>000C</td>
<td>C706040000000</td>
<td>MOV</td>
<td>WORD [0004],0</td>
<td>define c</td>
</tr>
<tr>
<td>0012</td>
<td>83E000000A</td>
<td>loop:</td>
<td>CMP WORD [0000],A</td>
<td>CMP a == 10</td>
</tr>
<tr>
<td>0017</td>
<td>7D</td>
<td>JGE</td>
<td>002D</td>
<td>if a&gt;=10 then 002D</td>
</tr>
<tr>
<td>0019</td>
<td>A10000</td>
<td>MOV</td>
<td>AX,[0000]</td>
<td>AX ← a</td>
</tr>
<tr>
<td>001C</td>
<td>D1E0</td>
<td>SHL</td>
<td>AX,1</td>
<td>AX ← 2*AX</td>
</tr>
<tr>
<td>001E</td>
<td>A30200</td>
<td>MOV</td>
<td>[0002],AX</td>
<td>b ← AX</td>
</tr>
<tr>
<td>0021</td>
<td>050700</td>
<td>ADD</td>
<td>AX,7</td>
<td>AX ← AX+7</td>
</tr>
<tr>
<td>0024</td>
<td>A30400</td>
<td>MOV</td>
<td>[0004],AX</td>
<td>c ← AX</td>
</tr>
<tr>
<td>0027</td>
<td>FF060000</td>
<td>INC</td>
<td>WORD [0000]</td>
<td>a ← a+1</td>
</tr>
</tbody>
</table>

Branch machine coded with displacement

displacement = (target address) - (fall-through address)

0017 JGE 002D: 002D - 0019 = 0014
0017 JGE 002D: 002D - 0019 = 0014
002B JMP 0012: 0012 - 002D = FFE5

Linking

Program file

File contents = Header + Load Module (data, code, stack)
No header for *.com program (no linking)
Code section = string of executable machine instructions

Example — Object Code with Resolved References

<table>
<thead>
<tr>
<th>address</th>
<th>HEX code</th>
<th>label</th>
<th>op-code and operands</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>C706000000300</td>
<td>MOV</td>
<td>WORD [0000],3</td>
<td>define a</td>
</tr>
<tr>
<td>0006</td>
<td>C706020000000</td>
<td>MOV</td>
<td>WORD [0002],FFFF</td>
<td>define b</td>
</tr>
<tr>
<td>000C</td>
<td>C706040000000</td>
<td>MOV</td>
<td>WORD [0004],0</td>
<td>define c</td>
</tr>
<tr>
<td>0012</td>
<td>83E000000A</td>
<td>loop:</td>
<td>CMP WORD [0000],A</td>
<td>CMP a == 10</td>
</tr>
<tr>
<td>0017</td>
<td>7D</td>
<td>JGE</td>
<td>002D</td>
<td>if a&gt;=10 then 002D</td>
</tr>
<tr>
<td>0019</td>
<td>A10000</td>
<td>MOV</td>
<td>AX,[0000]</td>
<td>AX ← a</td>
</tr>
<tr>
<td>001C</td>
<td>D1E0</td>
<td>SHL</td>
<td>AX,1</td>
<td>AX ← 2*AX</td>
</tr>
<tr>
<td>001E</td>
<td>A30200</td>
<td>MOV</td>
<td>[0002],AX</td>
<td>b ← AX</td>
</tr>
<tr>
<td>0021</td>
<td>050700</td>
<td>ADD</td>
<td>AX,7</td>
<td>AX ← AX+7</td>
</tr>
<tr>
<td>0024</td>
<td>A30400</td>
<td>MOV</td>
<td>[0004],AX</td>
<td>c ← AX</td>
</tr>
<tr>
<td>0027</td>
<td>FF060000</td>
<td>INC</td>
<td>WORD [0000]</td>
<td>a ← a+1</td>
</tr>
<tr>
<td>002B</td>
<td>EBE5</td>
<td>JMP</td>
<td>0012</td>
<td>back to 0012</td>
</tr>
<tr>
<td>002D</td>
<td>C3</td>
<td>end:</td>
<td>RET</td>
<td>return to DOS</td>
</tr>
</tbody>
</table>

Linking
Simple *.com Program

0100 BE1101 MOV SI,0111
0103 B80000 MOV AX,0
0106 0304 ADD AX,[SI]
0108 83C602 ADD SI,2
010B 833C00 CMP WORD [SI],0
010E 75F6 JNZ 0106
0110 C3 RET
0111 D1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,0
0110 C3 01 00 02 00 03 00 04-00 05 00 06 00 07 00 08
0120 00 09 00 0A 00 0B 00 0C-00 0D 00 0E 00 0F 00 00

AX = 0 → 1 → 3 → 6 → A → F → 15 → 1C →
24 → 2D → 37 → 42 → 4E → 5B → 69 → 78

String Processing Using Labels

0100 JMP L1
0102 S1: db "Line of Text",0,0,0,0
0112 S2: db "New Line",0,0,0,0,0,0,0
0122 L1: MOV DI,0080 ; destination
0125 LEA SI,[S1] ; source_1
0129 CALL L2 ; call copy
012C LEA SI,[S2] ; source_2
0130 CALL L2 ; call copy
0133 RET ; end of main routine
0134 L2: LODSB ; AL ← [SI], SI++
0135 STOSB ; [DI] ← AL, DI++
0136 CMP AL,00
0138 JNZ L2
013A RET

String Processing without Labels

0100 JMP 0122
0102 db "Line of Text",0,0,0,0
0112 db "New Line",0,0,0,0,0,0,0
0122 MOV DI,0080 ; destination
0125 LEA SI,[0102] ; source_1
0129 CALL 0134 ; call copy
012C LEA SI,[0112] ; source_2
0130 CALL 0134 ; call copy
0133 RET ; end of main routine
0134 LODSB ; AL ← [SI], SI++
0135 STOSB ; [DI] ← AL, DI++
0136 CMP AL,00
0138 JNZ 0134
013A RET

Arithmetic: Stored Data, Variables, and Pointers

0100 LEA SI,[0106] 0123 LODSW
0104 JMP 010A 0124 IMUL WORD [BX]
0106 DW 0000 0126 MOV [BX+02],AX
0108 DW 0000 0129 MOV [BX+04],DX
0110A MOV [SI+2],DS 012C CMP WORD [SI],0
010D LEA AX,[0132] 012F JNZ 0123
0111 CWD 0131 RET
0112 MOV BX,10 0132 DW 1122,0
0115 DIV BX 0136 DW 1,2,3,4,5,6,7,8
0117 ADD [SI+2],AX 0138 DW 9,a,b,c,d,e,f,0
011A MOV [SI],DX 011C LDS SI,[SI]
011E LEA BX,[SI] 011F ADD SI,4
0120 ADD SI,4

String Processing Using Labels

0100 JMP L1
0102 S1: db "Line of Text",0,0,0,0
0112 S2: db "New Line",0,0,0,0,0,0,0
0122 L1: MOV DI,0080 ; destination
0125 LEA SI,[S1] ; source_1
0129 CALL L2 ; call copy
012C LEA SI,[S2] ; source_2
0130 CALL L2 ; call copy
0133 RET ; end of main routine
0134 L2: LODSB ; AL ← [SI], SI++
0135 STOSB ; [DI] ← AL, DI++
0136 CMP AL,00
0138 JNZ L2
013A RET

String Processing without Labels

0100 JMP 0122
0102 db "Line of Text",0,0,0,0
0112 db "New Line",0,0,0,0,0,0,0
0122 MOV DI,0080 ; destination
0125 LEA SI,[0102] ; source_1
0129 CALL 0134 ; call copy
012C LEA SI,[0112] ; source_2
0130 CALL 0134 ; call copy
0133 RET ; end of main routine
0134 LODSB ; AL ← [SI], SI++
0135 STOSB ; [DI] ← AL, DI++
0136 CMP AL,00
0138 JNZ 0134
013A RET

Arithmetic: Stored Data, Variables, and Pointers

0100 LEA SI,[0106] 0123 LODSW
0104 JMP 010A 0124 IMUL WORD [BX]
0106 DW 0000 0126 MOV [BX+02],AX
0108 DW 0000 0129 MOV [BX+04],DX
0110A MOV [SI+2],DS 012C CMP WORD [SI],0
010D LEA AX,[0132] 0131 RET
0111 CWD 0132 DW 1122,0
0112 MOV BX,10 0136 DW 1,2,3,4,5,6,7,8
0115 DIV BX 0138 DW 9,a,b,c,d,e,f,0
0117 ADD [SI+2],AX 011F ADD SI,4
011A MOV [SI],DX 011C LDS SI,[SI]
011E LEA BX,[SI] 011F ADD SI,4
Assembly Programming — Matters of Style

Different methods are possible for:
- DOS *.exe and *.com programs
- Modular programming
- Segmentation
- Allocating memory for data
- Creating loops and control flow
- Defining function calls

Standard programming styles
- Assembler programming models
  Coding in native assembly language development environment
- C language model
  Assembly coding according to C language requirements
  For combining assembly and C code into one program

Program File Types

DOS/Windows
- .com file = Load Module
  Very simple programs
  Limited to one segment \( CS = DS = SS = ES \)
  No Program Header
  First instruction at \( IP = 0100 \)
- .exe file = DOS Header + Load Module
  Uses multiple segments and allows far branches
  OS provides value for \( DS \)
  Header contains initial values for \( CS:IP, SS:SP \)
  Header contains other OS parameters

Unix/Linux (on PC)
- ELF file = ELF Header + Load Module
  Use one only segment \( CS = DS = SS = ES \)
  Uses advanced 32-bit processor features

Modular Programming (Main + Functions)

Build program from separate source files (modules)
- Each source module edited in a separate file

Compile or Assemble source files into separate object files
- Object code is machine code with symbolic external references

Link object files together to create executable file
- Easier to design, read and understand programs
- Write most modules in high level language
- Write critical sections in assembly language
- Write, debug, and change modules independently

Linking

Store executable program file on disk
- Required for DOS *.exe files and Linux executable files
- Place segments into standard order
- Create program header

For programs compiled/ assembled from separate source files
- Combines separate OBJECT CODE modules
- Resolves EXTERNAL REFERENCES between modules

Function Stack
Function Code
Function Data
Main Stack
Main Code
Main Data
Program Header
Unified Stack
Function Stack
Function Code
Function Data
Main Stack
Main Code
Main Data
Program Header

Higher Addresses

- main.C -> compile -> main.OBJ
- f1.ASM -> assemble -> f1.OBJ
- f2.C -> compile -> f2.OBJ
- f_std.LIB -> load -> load
- link -> prog.EXE
- function.obj
- main.obj
- localized *.exe file
- unified *.exe file
- Unified Stack
- Function Stack
- Function Code
- Function Data
- Main Stack
- Main Code
- Main Data
- Program Header
Program Modules and Resolving References

Main module file — main.asm

```assembly
main:
    MOV AX, BX
    ADD AX, [0012] ; pass parameter in AX
    CALL function ; external reference
    ; resolved by linker
end:
    RET
```

Function module file — function.asm

```assembly
function:
    MOV CX, 20
    up:
    MUL CX       ; AX ← AX * CX
    CX ← CX - 1
    ; if CX > 0 IP ← up
    ; internal reference
    ; resolved by assembler
    LOOP up
    RET
```

Logical Offset — Paragraphs

Logical address to physical address

\[ \text{SEG:OFF} \rightarrow \text{PA} = \text{SEG} \times 10h + \text{OFF} \]

Physical address to logical address

\[ \text{PA} \rightarrow \text{SEG:OFF} = (\text{PA} / 10h):(\text{PA} \% 10h) \]

Paragraph

\[ 10h \text{ bytes} = \text{smallest 8086 segment} \]

Logical Offset

\[ N \text{ byte offset} \rightarrow (N / 10h):(N \% 10h) \]

Logical Offset = (N / 10h):(N % 10h)

DOS Program in Main Memory

- User Program
- Program Segment Prefix (PSP)
- DOS System Kernel
- User memory
  - Begins at DS:0000 — DOS always sets DS to point at PSP
  - DS paragraphs = DS \times 10h \text{ bytes from start of RAM}
- User program (load module)
  - Begins at start_segment:0000 = DS:0100 (PSP = 100h bytes)
  - DS:0100 \rightarrow DS \times 10h + 0100 = (DS + 10) \times 10h + 0000
  - start_segment = DS + 10 \Rightarrow DS = start_segment - 10

The Program Segment Prefix (PSP)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000h</td>
<td>2 bytes</td>
<td>int 20h (old style program terminate command)</td>
</tr>
<tr>
<td>0005h</td>
<td>5 bytes</td>
<td>Function call to DOS function handler</td>
</tr>
<tr>
<td>000Ah</td>
<td>4 bytes</td>
<td>Previous termination handler interrupt vector (int 22h)</td>
</tr>
<tr>
<td>000Eh</td>
<td>4 bytes</td>
<td>Previous contents of ctrl-C interrupt vector (int 23h)</td>
</tr>
<tr>
<td>0012h</td>
<td>4 bytes</td>
<td>Previous critical error handler interrupt vector (int 24h)</td>
</tr>
<tr>
<td>002Ch</td>
<td>2 bytes</td>
<td>Segment address of the program’s environment block</td>
</tr>
<tr>
<td>0080h</td>
<td>1 byte</td>
<td>Argument length</td>
</tr>
<tr>
<td>0081h</td>
<td>127 bytes</td>
<td>Number of characters — including spaces — following program name</td>
</tr>
</tbody>
</table>

Command tail

- All characters (including spaces) after file name
- Example: `format a: /q`
- command tail = " a: /q"
- command tail contains 6 characters
- command tail terminated by character 0x0d

Disk Transfer Area (DTA)

- DOS functions use PSP:0080 – PSP:00FF as write buffer
- Must save Command Tail before overwriting
Loading Program Arguments

```assembly
org 0x100
section .data
    argv: times 127 db 0 ; stores command tail

section .text
    mov si, 0x81 ; SI <-- pointer to command tail
    mov di, argv ; DI <-- pointer to command tail buffer
L1: lodsb ; al <-- character from command tail
    cmp al, 0x20 ; ignore initial spaces (0x20 characters)
    jne L2sjmp L1
L2: lodsb ; get file name character
    L2s: cmp al, 0x0d ; end on 0x0d character
    je L3stosb ; copy character to file name buffer
    jmp L2
L3: ; user program
    jmp L2
end: MOV AH,4Ch ; exit
    INT 21h ; call DOS
```

DOS *.com Program

```assembly
ORG 0x100
; define initiated data
section .data
    var1 dw 0x1234 ; var1 labels address of data

section .bss
    ; define uninitiated data

section .text
    ; code instructions
    mov ax, var1
    mov ah,4ch
    int 21h
```

Template for *.com Program

```assembly
ORG 0x100
section .data
    ; define initiated data
section .bss
    ; define uninitiated data
section .text
    ; code instructions
    mov ax, 0C00h
    int 21h
```

*.com Example — 1

```assembly
ex1.asm
ORG 0x100
section .data
    var1 dw 0x1234 ; var1 labels address of data

section .text
    mov ax, ds
    add ax, 10h
    mov cx, ax
    mov bx, ax
    mov ax, [var1] ; NASM keeps track of variable
    mov ah, 4ch
    int 21h
    ret
```

C:\nasm\programs\examples>nasm ex1.asm
**EQU Example — 1**

```assembly
org 0x100
section .data
  message1 db 'hello, world'
  message2 db '123456789'
  here equ $
  ; $ = &('9')+1
  ; here = numerical constant <- value &('9')+1
  msglen2 equ here-message2
  ; message2 = address of '1' = &('1')
  ; msglen2 = &('9')+1 - &('1')
  ; = length of string message2
  message3 db 'goodbye, world'
section .text
  mov ax, message1
  mov bx, here
  mov cx, msglen2
  mov ax,4C00H
  INT 21H
```

**EQU Example — 2**

```assembly
org 0x100
section .data
  message1 db 'Hello, world'
  message2 db '123456789'
  here equ $
  ; $ = &('9')+1
  ; here = numerical constant <- value &('9')+1
  msglen2 equ here-message2
  ; message2 = address of '1' = &('1')
  ; msglen2 = &('9')+1 - &('1')
  ; = length of string message2
  message3 db 'goodbye, world'
section .text
  mov ax, message1
  mov bx, here
  mov cx, msglen2
  mov ax,4C00H
  INT 21H
```

**Labels and Variables**

NASM associates a string with an address

- **LABEL** = pointer (start address) to line of code
- **VARIABLE** = pointer (start address) to data declaration

**Label arithmetic**

```assembly
.mov ax,[var1 + 2]
```

**AX** ← word after value of variable **var1**

Label arithmetic performed at assembly time

Assembler keeps track of addresses
Performs arithmetic
Uses numerical results in instructions

Example:

```assembly
.var1 labels a variable defined at DS:0006
.mov ax,[var1 + 2] coded as MOV AX,[0008]
```
**Simplest Segmentation Model**

Define non-overlapping segments for data, code, stack

**Example**

- **DS = 1000, CS = 2000, SS = 3000**
  - Data in addresses: DS:0000 → DS:FFFF → 10000 → 1FFFF
  - Code in addresses: CS:0000 → CS:FFFF → 20000 → 2FFFF
  - Stack in addresses: SS:0000 → SS:FFFF → 30000 → 3FFFF

**Very inefficient**

- Full segment = 64 KB
- Small program in 64 KB segment wastes space

**Example**

- **DS = 1000**
  - Code = 2000
  - Data segment = 32 bytes
    - Addresses: 10000 to 1001F
    - Wastes memory from 10020 to 1FFFF
Overlapping Segment Model — 2

Physical addresses → logical addresses

<table>
<thead>
<tr>
<th>Segment</th>
<th>Physical Load Address</th>
<th>Logical Address</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>10310h</td>
<td>SS:SP</td>
<td>0300h</td>
</tr>
<tr>
<td>code</td>
<td>10108h</td>
<td>CS:IP</td>
<td>0208h</td>
</tr>
<tr>
<td>data</td>
<td>10000h</td>
<td></td>
<td>0108h</td>
</tr>
<tr>
<td>PSP</td>
<td>100h</td>
<td>DS:0000</td>
<td>10h</td>
</tr>
</tbody>
</table>

PSP Logical Address: DS:0000

\[
DS = \text{int}(10100h / 10h) = 0FF0
\]

Logical Address of data = DS:0100

Overlapping Segment Example — on Disk

DOS program in *.exe file on disk

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
<th>Size</th>
<th>Segment Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stored Stack Values</td>
<td>b5</td>
<td>ISS = (b1 + b2 + b3 + b4)/10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SP = 0</td>
</tr>
<tr>
<td>Padding</td>
<td>Zeros</td>
<td>b4</td>
<td>10 - (b1 + b2 + b3) % 10</td>
</tr>
<tr>
<td>Function</td>
<td>MOV CX,20</td>
<td>b3</td>
<td>Fn_CS = (b1 + b2)/10</td>
</tr>
<tr>
<td>Code</td>
<td>MUL CX</td>
<td></td>
<td>Fn_IP = (b1 + b2) % 10</td>
</tr>
<tr>
<td>Main</td>
<td>MOV AX,DX</td>
<td>b2</td>
<td>ICS = b1 / 10</td>
</tr>
<tr>
<td>Code</td>
<td>ADD AX,[0012]</td>
<td></td>
<td>IP = b1 % 10</td>
</tr>
<tr>
<td>Data</td>
<td>Stored Data</td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Program Header</td>
<td>b0</td>
<td>Stores: ICS,IP,ISS,SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Pointer for Fn_CS</td>
</tr>
</tbody>
</table>

Overlapping Segment Example — RAM

DOS program in main memory (RAM)

Physical load address \(PA_{\text{start}}\) determined by DOS

\[
s_{\text{seg}} = \text{start\_segment} = PA_{\text{start}} / 10h
\]

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
<th>Size in bytes</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stack Values</td>
<td>b5</td>
<td>SS = s_{\text{seg}} + ISS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SP = 0</td>
</tr>
<tr>
<td>Padding</td>
<td>Zeros</td>
<td>b4</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>MOV CX,20</td>
<td>b3</td>
<td>Fn_CS ← s_{\text{seg}} + Fn_CS</td>
</tr>
<tr>
<td>Code</td>
<td>MUL CX</td>
<td></td>
<td>Fn_IP</td>
</tr>
<tr>
<td>Main</td>
<td>MOV AX,DX</td>
<td>b2</td>
<td>CS = s_{\text{seg}} + ICS</td>
</tr>
<tr>
<td>Code</td>
<td>ADD AX,[0012]</td>
<td></td>
<td>IP</td>
</tr>
<tr>
<td>Data</td>
<td>Stored Data</td>
<td>b1</td>
<td>s_{\text{seg}} = PA_{\text{start}} / 10h</td>
</tr>
<tr>
<td>PSP</td>
<td>DOS Data</td>
<td>100h</td>
<td>DS = s_{\text{seg}} - 10h</td>
</tr>
</tbody>
</table>
Numerical Example — Linked *.exe

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
<th>Size (bytes)</th>
<th>Segment Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stored Stack Values</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Padding</td>
<td>Zeros</td>
<td>000E</td>
<td>10 - (0108 + 0052 + 0208) % 10 = 10 - (0362 % 10) = E</td>
</tr>
<tr>
<td>Function</td>
<td>Code</td>
<td>0208</td>
<td>Fn_CS = (0108 + 0052)/10 = (015A)/10 = 0015</td>
</tr>
<tr>
<td>Main Code</td>
<td></td>
<td>0052</td>
<td>ICS = 0108 / 10 = 0010</td>
</tr>
<tr>
<td>Data</td>
<td>Stored Data</td>
<td>0108</td>
<td>s_seg = 1000 (load address 10000h)</td>
</tr>
<tr>
<td>Prefix</td>
<td>DOS Data</td>
<td>—</td>
<td>Prefix only in memory</td>
</tr>
<tr>
<td>Header</td>
<td>Program Header</td>
<td>200</td>
<td>At load time</td>
</tr>
</tbody>
</table>

Template for *.exe Program

```
section data
; declare initialized data here
section bss
; declare uninitialized data here
section text
; starting IP at next line
   MOV AX, data ; points DS at data segment
   MOV DS, AX ;
   ; put user code in this area
   MOV AX, 4C00H
   INT 21H
section stack stack ; declares stack segment named stack
   ; name is required for linker
   resb 2000h ; default 8 KB stack
   stacktop: ; points SP at top of stack

section Declarations

Defines and separates named sections
Usage depends on code type
Unix, Linux, and DOS *.com files
   Standard section names are required
      .data, .bss, .text
   Linked file begins with .text section (code)
   Data placed after end of code
DOS *.exe files
   Any section names are allowed
   Section names have no special meaning
      NASM changes .data → data
   Linked file contains named sections in order of their definition
   Sections from separate sources
      Combined if names are identical
      Not combined if names are different

Default Code

With Default Code

```

Without Default Code

```

**.exe Example — 1**

```assembly
section data
var1 db "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
var2 dw 0x1234
var3 dw 0x0000

section text
..start:
mov ax, data
mov DS, ax
mov di,var1
mov si,var2
call L1
mov ax,4c00h
int 21h

L1: lea bx,[di]
mov ax,[si]
mov [var3],ax
ret

section stack
resb 64
stacktop:
```

**.exe Example — 2**

```bash
C:\nas\programs\examples>nasm -f obj ex.asm
NASM creates file ex.obj

C:\nas\programs\examples>link ex
Microsoft (R) Segmented Executable Linker  Version 5.60.339 Dec 5 1994
Copyright (C) Microsoft Corp 1984-1993. All rights reserved.

Run File [ex.exe]:
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:

LINK.EXE creates file ex.exe
```

**.exe Example — 3**

```assembly
C:\nas\programs\examples>debug ex.exe
-ax=0000 bx=0000 cx=0024 dx=6000 sp=0040 bp=0000 si=0000 di=0000
ds=0b76 es=0b76 ss=0b89 cs=0b86 ip=000c nv up ei pl nz na po nc
0b86:0000 b8860b mov ax,0b86
-d de=100
0b86:0100 41 42 43 44 45 46 47 48 34 12 00 00 08 06 08 08 00 00 4c cd 21 8d  .......L.!

-u cs:0c 1 20
0b86:0000 b8860b mov ax,0b86 ; data = DS + 10 = 0b76 + 10
0b86:000f 8ed8 mov di,0000 ; var1
0b86:0014 b88000 mov si,0008 ; var2
0b86:0017 b80500 call 001f
0b86:001a b80400 mov ax,4c00
0b86:001d cd21 int 21
0b86:001f bd1d lea bx,[di]
0b86:0021 b804 mov ax,[si]
0b86:0023 a30a00 mov [000a],ax ; var3
0b86:0026 c3 ret
0b86:0027 b80a18 call 1834
0b86:002a eb18 jmp 0044
```

**.exe Example — 4**

```assembly
 Linking of Named Sections

section data
a1: dw 0x0011

section text
..start:
mov ax,data
mov bx,a1
mov ax,a2
ret

section data
a2: dw 0x0033

section text
Func: mov ax,a1
mov bx,a2
ret
```

```assembly
 Linking of Named Sections

section data_1
a1: dw 0x0011

section text_1
..start:
mov ax,data_1
...call far Func
mov ax,4c00h
int 21h

section data_2
a2: dw 0x0033

section text_2
Func: mov ax,a1
mov bx,a2
ret
```

```assembly
 Linking of Named Sections

section data
a1: dw 0x0011

section text
..start:
mov ax,data
...call far Func
mov ax,4c00h
int 21h

section data
a2: dw 0x0033

section text
Func: mov ax,a1
mov bx,a2
ret
```
Far Call — 1

section data
; global data section
; declare initialized variables here
stdout dw 0
str_1 db "hello world",0dh,0ah,0
section bss
; declare uninitialized variables here
section text
..start:
    MOV AX, data
    MOV DS, AX
    mov si, str_1
    call far print_scr
    MOV AX,4C00H
    INT 21H

Far Call — 2

C:\\nas\programs\examples>debug ex5.exe
- r
AX=0000 BX=0000 CX=0050 DX=0000 DS=0B5F ES=085F SS=0B72 CS=0B70
IP=0000
NV UP EI PL NZ NA PO NC
- 0B70:0000 B86F0B MOV AX,0B6F
0B70:0003 BED8 MOV DS,AX
0B70:0005 BE0200 MOV SI,0002
0B70:0008 9A0200710B CALL 0B71:0002 ↪ PA = 0B712 ➔ 0B70:0012
0B70:000D B8004C MOV AX,4C00
0B70:0010 CD21 INT 21
0B70:0012 B402 MOV AH,02
0B70:0014 AC LODSB
0B70:0015 3C00 CMP AL,00
0B70:0017 7501 JNZ 001A
0B70:0019 CB RETF
0B70:001A 88C2 MOV DL,AL
0B70:001C CD21 INT 21
0B70:001E EBF4 JMP 0014
0B70:0020 0000 ADD [BX+SI],AL
0B70:0022 0000 ADD [BX+SI],AL
0B70:0024 0000 ADD [BX+SI],AL

C:\\nas\programs\examples>nasm -f obj ex5.asm
C:\\nas\programs\examples>link ex5
Microsoft (R) Segmented Executable Linker  Version 5.40.339 Dec  5 1994
Copyright (C) Microsoft Corp 1984-1993.  All rights reserved.
Run File [ex5.exe]: ex5.exe
List File [nul.map]: Libraries [.lib]:
Definitions File [nul.def]:

Separate Modules — 1

ex6a.asm

extern print_scr

section data
stdout dw 0
str_1 db "hello world",0dh,0ah,0
section bss
section text
..start:
    MOV AX, data
    MOV DS, AX
    mov si, str_1
    call far print_scr
    MOV AX,4C00H
    INT 21H

Separate Modules — 2

C:\\nas\programs\examples>nasm -f obj ex6a.asm
C:\\nas\programs\examples>nasm -f obj ex6b.asm
C:\\nas\programs\examples>link ex6a ex6b
Microsoft (R) Segmented Executable Linker  Version 5.60.339 Dec  5 1994
Copyright (C) Microsoft Corp 1984-1993.  All rights reserved.
Run File [ex6a.exe]: ex6.exe
List File [nul.map]:
Libraries [.lib]:
Definitions File [nul.def]:
Separate Modules — 3

C:\nasm\programs\examples>debug ex6.exe

-A
AX=0000 BX=0000 CX=0030 DX=0000 SP=0020 BP=0000 SI=0000 DI=0000
DS=0B5F ES=0B5F SS=0B72 CS=0B70 IP=0000 NV UP EI PL NZ NA PO NC

-OB70:0000 B66FB MOV AX,0B6F ; DS ← data = start_seg = PSP + 10h
OB70:0003 8ED8 MOV DS,AX
OB70:0005 BE200 MOV SI,0002
OB70:0008 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:000D B8004C MOV AX,4C00
OB70:0010 CD21 INT 0x21
OB70:0012 B8004C MOV AX,4C00
OB70:0015 BE0200 MOV SI,0002
OB70:0018 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:001D B8004C MOV AX,4C00
OB70:0020 CD21 INT 0x21
OB70:0022 B8004C MOV AX,4C00
OB70:0025 BE0200 MOV SI,0002
OB70:0028 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:002D B8004C MOV AX,4C00
OB70:0030 CD21 INT 0x21
OB70:0032 B8004C MOV AX,4C00
OB70:0035 BE0200 MOV SI,0002
OB70:0038 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:003D B8004C MOV AX,4C00
OB70:0040 CD21 INT 0x21
OB70:0043 B8004C MOV AX,4C00
OB70:0046 BE0200 MOV SI,0002
OB70:0049 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:004E B8004C MOV AX,4C00
OB70:0051 CD21 INT 0x21
OB70:0054 B8004C MOV AX,4C00
OB70:0057 BE0200 MOV SI,0002
OB70:005A 9A020071B CALL OB71:0002 → PA = OB712 → OB70:0012
OB70:005F B8004C MOV AX,4C00

ex6.exe File Contents

C:\nasm\programs\examples>dump ex6.exe

Stack

ISS = 3h

01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...

Text

ICS = 1h

0210 B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0220 CD 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Data

0B5F:0100 00 00 68 65 6C 6C 6F 20 77 6F 72 6C 64 0D 0A 00 ..hello world...
0B5F:0110 B8 6F 0B 8E D8 BE 02 00 9A 02 00 02 00 BB 00 4C .o.........q...L

Header

0000 4D 5A 50 00 02 00 00 FF FF 03 00
0010 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...

ex6.exe Disassembly

00000010 B80000 mov ax, 0x0 ; start_seg = 0
00000013 8ED8 mov ds,ax
00000015 BE200 mov si,0x2
00000018 9A020071B call 0x2:0x2 ; FnCS = 0x2
0000001D B8004C mov ax,0x4c00
00000020 CD21 int 0x21
00000022 B402 mov ah,0x2
00000024 AC lodsb
00000025 3C00 cmp al,0x0
00000027 7501 jnz 0x1a
00000029 CB retf
0000002A 88C2 mov dl,al
0000002C CD21 int 0x21
0000002E EBF4 jmp short 0x14
00000030 0000 add [bx+si],al
00000032 0000 add [bx+si],al

Offset registers set at load time

SP ← value stored in header
IP ← value stored in header

Segment registers set at load time

ES = DS ← start_segment - 10 (point at PSP)
CS ← start_segment + ICS
SS ← start_segment + ISS

Relocation items: FAR BRANCH addresses adjusted

Target for branch instruction stored in data, code, or stack section

Example

CALL Fn_CS:Fn_IP
Stored value Fn_CS = paragraphs to Fn from start of load module
Fn_CS ← start_segment + Fn_CS
Adjusted Fn_CS = paragraphs to Fn from start of RAM
**.exe Program Relocation — 2**

FAR segment addresses in *.exe file — relative to start of load module
At load time, Relocation Items adjusted

FAR SEG ← start segment + FAR SEG

---

**Program Header Example — Relocation**

**Top of Header**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>000h - 01h</td>
<td>4D 5A h</td>
<td>Length of image mod 512 = 0050h</td>
</tr>
<tr>
<td>02h - 05h</td>
<td>02 00</td>
<td>Size of file in 512 byte pages (including remainder) = 0002h</td>
</tr>
<tr>
<td>04h - 07h</td>
<td>02 00</td>
<td>Maximum number of 16 byte paragraphs required by program = 0002</td>
</tr>
<tr>
<td>06h - 07h</td>
<td>02 00</td>
<td>Number of relocation table items following header = 0002</td>
</tr>
<tr>
<td>08h - 09h</td>
<td>1E 00</td>
<td>Size of header in 16 byte increments (paragraphs) = 001E</td>
</tr>
<tr>
<td>0Ah - 0Bh</td>
<td>00 00</td>
<td>Minimum number of 16 byte paragraphs required = 0000</td>
</tr>
<tr>
<td>0Ch - 0Dh</td>
<td>00 00</td>
<td>Maximum number of 16 byte paragraphs required = 0000</td>
</tr>
<tr>
<td>0Eh - 0Fh</td>
<td>03 00</td>
<td>ISS = 0003</td>
</tr>
</tbody>
</table>

**Relocation vectors**

<table>
<thead>
<tr>
<th>Byte displacement in load module</th>
<th>0001:0001</th>
<th>0010</th>
<th>0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load module in file ex6.exe</td>
<td>0010</td>
<td>B8 00</td>
<td>BE D8 BE 02 00 9A 02 00 71 0B 00 4C</td>
</tr>
<tr>
<td>Load module in memory</td>
<td>OB6F:0000</td>
<td>B8 00</td>
<td>BE D8 BE 02 00 9A 02 00 71 0B 00 4C</td>
</tr>
<tr>
<td>Relocation</td>
<td>OB6F + 0000 = OB6F</td>
<td>OB6F + 0002 = OB71</td>
<td></td>
</tr>
</tbody>
</table>

---

**Program Header**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h - 01h</td>
<td>4D 5A h — signature for a valid *.exe file.</td>
</tr>
<tr>
<td>02h - 03h</td>
<td>(Size of file including header) / 512</td>
</tr>
<tr>
<td>04h - 05h</td>
<td>Int [(Size of file including header) / (512)] + 1</td>
</tr>
<tr>
<td>06h - 07h</td>
<td>Number of relocation table items following header</td>
</tr>
<tr>
<td>08h - 09h</td>
<td>Size of header in 16 byte increments (paragraphs)</td>
</tr>
<tr>
<td>0Ah - 0Bh</td>
<td>Minimum number of 16 byte paragraphs required by program</td>
</tr>
<tr>
<td>0Ch - 0Dh</td>
<td>Maximum number of 16 byte paragraphs required</td>
</tr>
<tr>
<td>0Eh - 0Fh</td>
<td>ISS — paragraphs from start of load module to stack segment</td>
</tr>
<tr>
<td>10h - 11h</td>
<td>Starting SP</td>
</tr>
<tr>
<td>12h - 13h</td>
<td>Word Checksum - negative sum of words in file ignoring overflow</td>
</tr>
<tr>
<td>14h - 15h</td>
<td>Starting IP</td>
</tr>
<tr>
<td>16h - 17h</td>
<td>ICS — paragraphs from start of load module to code segment</td>
</tr>
<tr>
<td>18h - 19h</td>
<td>Displacement (bytes) of first relocation vector in file</td>
</tr>
<tr>
<td>1Ah - 1Bh</td>
<td>Overlay number (0 for main resident part of program)</td>
</tr>
</tbody>
</table>