Assembly Language in x86 Memory Architecture

Short Overview

Assembly Language Programs — 1

Assembly language program

List of x86 instructions + data

CPU executes — one by one — instructions in order of listing

Example:

```
MOV EAX, EBX
ADD EAX, ECX
SUB EAX, [EBX+1234h]
PUSH EAX
MOV EAX, 1
INT 80h
```

Running this program requires CPU access to memory:

Code segment access (**[CS:EIP]**) to fetch instructions

Data segment access (**[DS:EBX+1234]**) to load operand

Stack segment access (**[SS:ESP]**) to execute push

Assembly Language Programs — 2

Assembly language converted to machine language

```
89D8            MOV     EAX,EBX
01C8            ADD     EAX,ECX
2B8334120000    SUB     EAX,[EBX+00001234h]
50              PUSH    EAX
B801000000      MOV     EAX,00000001
CD80            INT     80
```

Machine language version of program

Machine code = string of hexadecimal codes

89D801C82B8334120000 50 B801000000 CD80

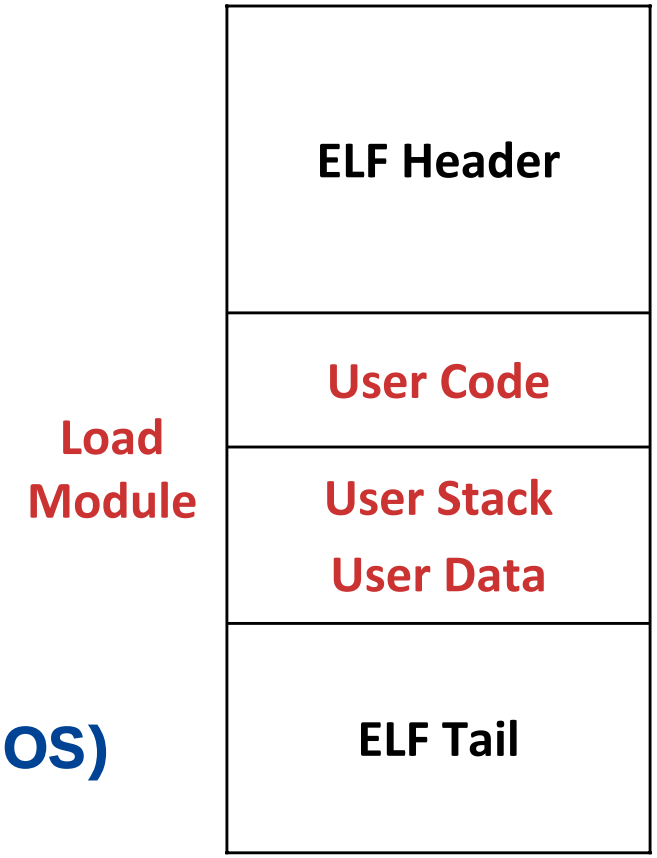
Assembly Language Programs — 3

Linux program files

Stored on disk or other long term storage

Contains

- Header
  - Information for operating system
- Load module
  - Data, Stack, Code part of program



Loading and running by operating system (OS)

Copy load module into memory

Initialize registers

- **`CS, SS, DS, ES`** point to same segment start address
- In Linux, **`ESP`** points to system stack

Point **`EIP`** at program start instruction

OS load operations

Copy data to memory at address **DS:EA**

Copy code to memory at address **CS:EIP**

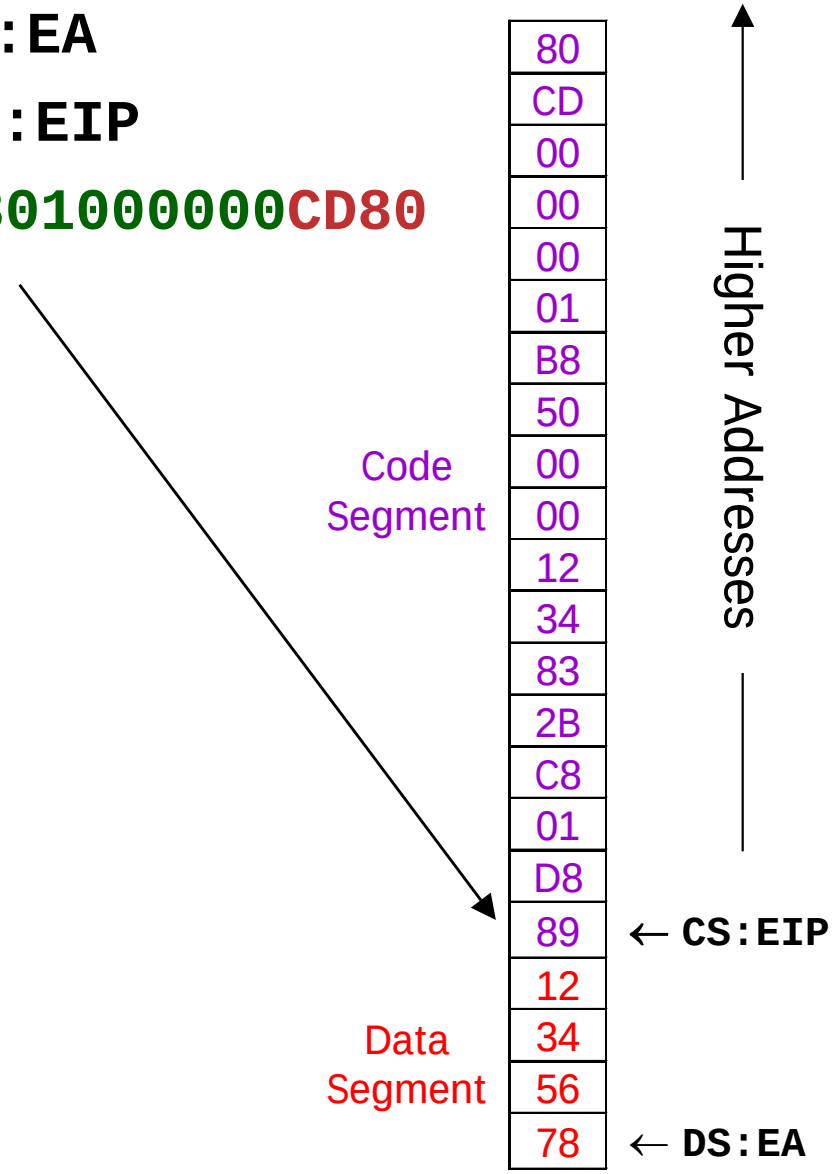
89D801C82B83341200050B801000000CD80

Initialize registers

DS register ← data selector

SS:ESP ← stack pointer

CS:EIP ← code pointer



Assembly Language Programs — 6

Data locations for running program

Transfer and ALU instructions refer (default) to data segment (**DS**)

Execute **SUB EAX, [EBX+123h4]**

Calculate Effective Address (EA) **EBX+1234h**

Load operand from data segment address **DS:EBX+1234h**

Perform **EAX ← EAX - [DS:EBX+1234h]**

Program must know where data is stored

Code locations for running program

Instructions stored in memory in order of program list

OS sets **EIP** to first instruction in **CS**

CPU updates **EIP** after loading an instruction

EIP ← address of next instruction

Program only needs to know **CS:EIP** for branch instructions

Branch instructions

A simple **JUMP** (skip over **SUB** and **PUSH**) looks like:



```
        MOV EAX, EBX
        ADD EAX, ECX
        JMP L1
        SUB EAX, [EBX+1234h]    ; fall-through
        PUSH EAX
L1:     MOV EAX, 1              ; target
        INT 80h
```

JMP assembly coded with **pointer** **L1** to line **MOV EAX, 1**

Pointer is address of target instruction

JMP machine coded with **displacement** to line **MOV EAX, 1**

Displacement = target address – fall-through address

Branch instructions

Program listing with instruction lengths:



```
bytes   instruction
  2       MOV EAX, EBX
  2       ADD EAX, ECX
  5       JMP L1                    ; EIP ← EIP + 7
  6       SUB EAX, [EBX+1234h]      ; fall-through
  1       PUSH EAX
  5  L1:  MOV EAX, 1                ; target
  2       INT 80h
```

JMP instruction **ASSEMBLY CODED** with **EIP** of target

JMP instruction **MACHINE CODED** with **displacement**

displacement = **6 + 1 = 7**

The Assembly Language Programming Process

Stage	User Action	Detailed Actions
Top-down design	Plan	1. High level design 2. Organization into modules and functions
Coding modules	Program	1. Define local data structures 2. Write assembly language code for modules
Assembly	Run Assembler	1. Convert assembly code to machine code 2. Resolve internal references
Linking	Run Linker	1. Resolve external address references for jump and call instructions 2. Build program header 3. Store executable program file on disk
Loading	Run Loader	1. Place program in memory 2. Set instruction pointer to program

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 = reference by pointer

Example: `z = y + x`

Assignments

```
x ~ DS:0000
y ~ DS:0004          ; scratchpad
z ~ DS:0008
```

Coding

```
MOV EAX,[0000]      ; EAX ← x
ADD EAX,[0004]      ; EAX ← EAX + y
MOV [0008],EAX      ; z ← EAX
```

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

Address	HEX Code	Label	Op-Code	Operand	Comments

Example of Assembly Coding

Implement pseudocode example:

```
a = 3, b = -1, c = 0
while (a < 10){
      b = a * 2
      c = b + 7
      a++
       }
end
```

Assembly by Hand — Coding

```
a = 3, b = -1, c = 0
while (a < 10){
        b = a * 2
        c = b + 7
        a++
         }
end
```

Enter op code, line labels, comments

relative address	HEX code	label	op-code and operands	comment
			`MOV DWORD [0000],3`	`; define a`
			`MOV DWORD [0004],FFFFFFFFh`	`; define b`
			`MOV DWORD [0008],0`	`; define c`
		`loop:`	`CMP DWORD [0000],Ah`	`; CMP a == 10`
			`JGE end`	`; if a>= 10 then end`
			`MOV EAX,[0000]`	`; EAX ← a`
			`SHL EAX,1`	`; EAX ← 2*EAX`
			`MOV [0004],EAX`	`; b ← EAX`
			`ADD EAX,7`	`; AX ← EAX+7`
			`MOV [0008],EAX`	`; c ← EAX`
			`INC DWORD [0000]`	`; a ← a+1`
			`JMP loop`	`; back to loop`
		`end:`	`MOV EAX, 1`	
			`INT 80h`	

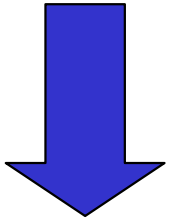
Machine Language Instruction

Translate op-code and operands to HEX Code

HEX Code format for each addressing mode

Translation based on Intel op-code tables

```
MOV DWORD PTR [0000], 3
```



```
C705000000000300000
```

Hand Assembly — Convert to Machine Code

Enter hexadecimal code for each line

relative address	HEX code	label	op-code and operands
	C705000000003000000		MOV DWORD [00000000],00000003
	C705040000000FFFFFFFF		MOV DWORD [00000004],FFFFFFFF
	C705080000000000000		MOV DWORD [00000008],00000000
	813D000000000A000000	loop:	CMP DWORD [00000000],0000000A
	7D__		JGE end
	A100000000		MOV EAX,[00000000]
	D1E0		SHL EAX,1
	A304000000		MOV [00000004],EAX
	0507000000		ADD EAX,00000007
	A308000000		MOV [00000008],EAX
	FF0500000000		INC DWORD [00000000]
	E9__		JMP loop
	B801000000	end:	MOV EAX,00000001
	CD80		INT 80

Branch codes incomplete:

Target addresses unresolved (not calculated)

Address of Instruction in Memory

Arbitrary start address

Default **`EIP = 0`**

Relative to code segment base (**CS**)

Count instruction length (bytes in each instruction)

Next address = previous address + previous instruction length

relative address	HEX code	label	op-code and operands
00	C705000000003000000		MOV DWORD [00000000],00000003
0A	C70504000000FFFFFFFF		MOV DWORD [00000004],FFFFFFFF
14	C705080000000000000		MOV DWORD [00000008],00000000
1E	813D000000000A000000	loop:	CMP DWORD [00000000],0000000A



C7	05	04	00	00	00	FF	FF	FF	FF
0A	0B	0C	0D	0E	0F	10	11	12	13

Example of Assembly Coding — Unresolved

relative address	HEX code	label	op-code and operands
00	C705000000003000000		MOV DWORD [00000000],00000003
0A	C705040000000FFFFFFFF		MOV DWORD [00000004],FFFFFFF
14	C705080000000000000		MOV DWORD [00000008],00000000
1E	813D000000000A000000	loop:	CMP DWORD [00000000],0000000A
28	7D__	2	JGE 4B ; short = 1 byte
2A	A100000000	5	MOV EAX,[00000000]
2F	D1E0	2	SHL EAX,1
31	A304000000	5	MOV [00000004],EAX
36	0507000000	5	ADD EAX,00000007
3B	A308000000	5	MOV [00000008],EAX
40	FF0500000000	6	INC DWORD [00000000]
46	E9__	5	JMP 1E ; reverse = 4 bytes
4B	B801000000	end:	MOV EAX,00000001
50	CD80		INT 80

Resolving branch references

Line label = symbolic reference to instruction address

Replace line labels in code with **EIP** address of target line

loop → 001E

end → 004B

Example of Assembly Coding — Resolving Code

relative address	HEX code	label	op-code and operands
00	C705000000003000000		MOV DWORD [00000000],00000003
0A	C705040000000FFFFFFFF		MOV DWORD [00000004],FFFFFFFF
14	C705080000000000000		MOV DWORD [00000008],00000000
1E	813D000000000A000000	loop:	CMP DWORD [00000000],0000000A
28	7D__		JGE 4B ; short = 1 byte
2A	A100000000		MOV EAX,[00000000]
2F	D1E0		SHL EAX,1
31	A304000000		MOV [00000004],EAX
36	0507000000		ADD EAX,00000007
3B	A308000000		MOV [00000008],EAX
40	FF0500000000		INC DWORD [00000000]
46	E9__		JMP 1E ; reverse = 4 bytes
4B	B801000000	end:	MOV EAX,00000001
50	CD80		INT 80

Branch machine coded with **displacement**

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

28 JGE 4B: 4B – 2A = 21 (short)

44 JMP 1E: 1E – 4B = FFFFFFD3 → D3FFFFFF

Example — Object Code with Resolved References

relative address	HEX code	label	op-code and operands
00	C705000000003000000		MOV DWORD [00000000],00000003
0A	C705040000000FFFFFFFF		MOV DWORD [00000004],FFFFFFFF
14	C705080000000000000		MOV DWORD [00000008],00000000
1E	813D000000000A000000	loop:	CMP DWORD [00000000],0000000A
28	7D21		JGE 4B ; short = 1 byte
2A	A100000000		MOV EAX,[00000000]
2F	D1E0		SHL EAX,1
31	A304000000		MOV [00000004],EAX
36	0507000000		ADD EAX,00000007
3B	A308000000		MOV [00000008],EAX
40	FF0500000000		INC DWORD [00000000]
46	E9D3FFFFFF		JMP 1E ; reverse = 4 bytes
4B	B801000000	end:	MOV EAX,00000001
50	CD80		INT 80

Code part of Load Module

C705000000003000000C705040000000FFFFFFFFC705080000000
0000000813D000000000A0000007D21A100000000D1E0A3040000
00050700000A308000000FF0500000000E9D3FFFFFFB80100000
0CD80

Code Addressing in Linux

All addresses relative to same segment start address

CS, DS, ES, SS point to same segment start address

All Linux programs start at **EIP = 080480C0**

Data section starts after code at **EA** set by Linux

Stack pointer **ESP** set by Linux to system stack

Code in memory

Address	Machine Instruction	Assembly Instruction
080480C0	C70514910408030000000	MOV DWORD [08049114],00000003
080480CA	C70518910408FFFFFFFF	MOV DWORD [08049118],FFFFFFFF
080480D4	C7051C9104080000000000	MOV DWORD [0804911C],00000000
080480DE	813D149104080A000000	CMP DWORD [08049114],0000000A
080480E8	7D21	JGE 0804810B
080480EA	A100000000	MOV EAX,[08049114]
080480EF	D1E0	SHL EAX,1
080480F1	A304000000	MOV [08049118],EAX
080480F6	0507000000	ADD EAX,00000007
080480FB	A308000000	MOV [0804911C],EAX
08048100	FF0500000000	INC DWORD [08049114]
08048106	E9D3FFFFFF	JMP 080480DE
0804810B	B801000000	MOV EAX,00000001
08048110	CD80	INT 80

Data section starts at
EA = 08049114

Addressing Example — 1

ex9.asm

```
section .data
    var: dd 1,2,3,4,5,6,7,8
section .text
global _start
_start:
    MOV EAX, var
    MOV EBX, st
    MOV EAX,1
    INT 80h
section .stack    ; local user stack
    st: dd 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88
        stacktop:
```

Addressing Example — 2

```
section .data
      var: dd 1,2,3,4,5,6,7,8
section .text
global _start
_start:
      MOV EAX, var          ; var = 080490f4 = start of data section
      MOV EBX, st           ; st =  080480d1 = start of user stack section
      MOV EAX,1
      INT 80h
section .stack
      st: dd 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88
        stacktop:
```

Object dump: **objdump –d ex9**

```
Disassembly of section .text:
080480c0 <_start>:
 80480c0:       b8 f4 90 04 08          mov    $0x80490f4,%eax
 80480c5:       bb d1 80 04 08          mov    $0x80480d1,%ebx
 80480ca:       b8 01 00 00 00          mov    $0x1,%eax
 80480cf:       cd 80                   int    $0x80
```

Addressing Example — 3

```
var = 080490f4 = start of data section
st =  080480d1 = start of local user stack section
```

Object dump: `objdump –s ex9`

OS Data

```
Contents of section .note.gnu.build-id:
 8048094 04000000 14000000 03000000 474e5500  ............GNU.
 80480a4 7fb4986d 9ba171ba 5c0a1b92 3dd42c81  ...m..q.\...=.,.
 80480b4 34d411b3                             4...
```

Load Module

```
Contents of section .text:
 80480c0 b8f49004 08bbd180 0408b801 000000cd  ................
 80480d0 80                                   .
```

```
Contents of section .stack:
 80480d1 11000000 22000000 33000000 44000000  ....".3...D...
 80480e1 55000000 66000000 77000000 88000000  U...f...w.......
```

```
Contents of section .data:
 80490f4 01000000 02000000 03000000 04000000  ................
 8049104 05000000 06000000 07000000 08000000  ................
```

Addressing Example — 4

```
section .data
      var: dd 1,2,3,4,5,6,7,8
section .text
global _start
_start:
      MOV EAX, var ; var = 080490f4 = start of data section
      MOV EBX, st          ; st =  080480d1 = start of local stack section
      MOV EAX,1
      INT 80h
section .stack
      st: dd 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88
      stacktop:
```

Program loaded in debugger: `debug ex9`

Code	080480C0 B8F4900408 MOV EAX,080490F4 080480C5 BBD1800408 MOV EBX,080480D1 080480CA B801000000 MOV EAX,00000001 080480CF CD80 INT 80
Stack	080480D1 11 00 00 00 22 00 00 00-33 00 00 00 44 00 00 00 080480E1 55 00 00 00 66 00 00 00-77 00 00 00 88 00 00 00
Data	080480F1 00 00 00 01 00 00 00 02-00 00 00 03 00 00 00 04 08048101 00 00 00 05 00 00 00 06-00 00 00 07 00 00 00 08

Another Addressing Example — 1

ex10.asm

```
section .data
    var: dd 0x11223344
section .text
global _start
_start: mov eax, var
        mov ebx, [var]
        push var
        push dword [var]
        pop ecx
        pop edx
mov eax,1
mov ebx,0
int 80h
```

ex10 loaded into debug

```
080480C0 B8E4900408   MOV  EAX,080490E4
080480C5 8B1DE4900408 MOV  EBX,DWORD [080490E4]
080480CB 68E4900408   PUSH DWORD 080490E4
080480D0 FF35E4900408 PUSH DWORD [080490E4]
080480D6 59           POP  ECX
080480D7 5A           POP  EDX
080480D8 B801000000   MOV  EAX,00000001
080480DD BB00000000   MOV  EBX,00000000
080480E2 CD80         INT  80
080480E4 44           INC  ESP
080480E5 3322         XOR  ESP,DWORD [EDX]
080480E7 1100         ADC  DWORD [EAX],EAX
```

```
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BFF4A560 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF
```

Another Addressing Example — 2

ex10 loaded into debug

```
080480C0 B8E4900408   MOV  EAX,080490E4
080480C5 8B1DE4900408 MOV  EBX,DWORD [080490E4]
080480CB 68E4900408   PUSH DWORD 080490E4
080480D0 FF35E4900408 PUSH DWORD [080490E4]
080480D6 59           POP  ECX
080480D7 5A           POP  EDX
080480D8 B801000000   MOV  EAX,00000001
080480DD BB00000000   MOV  EBX,00000000
080480E2 CD80         INT  80
080480E4 44           INC  ESP
080480E5 3322         XOR  ESP,DWORD [EDX]
080480E7 1100         ADC  DWORD [EAX],EAX
```

objdump -s ex10

```
ex10:     file format elf32-i386

Contents of section .note.gnu.build-id:
 8048094 04000000 14000000 03000000 474e5500  ............GNU.
 80480a4 01b4c8e0 d5209c45 d7bf3ad1 8fa96b05  ..... .E..:...k.
 80480b4 7155c892                             qU..
Contents of section .text:
 80480c0 b8e49004 088b1de4 90040868 e4900408  ...........h....
 80480d0 ff35e490 0408595a b8010000 00bb0000  .5....YZ........
 80480e0 0000cd80                             ....
Contents of section .data:
 80490e4 44332211                             D3".
```

Running Executable in Linux

Linux

- Loads program to memory
- Allocates system resources for program
  - Memory
  - System stack
  - Environment variables
  - Other system resources
- Pushes program information onto system stack
- Initializes registers
  - **ESP** set to system stack
  - User must define label for local stack as part of data segment

Information in System Stack at Run Time

Address	Contents
ESP	Number of arguments (including executable)
ESP + 4	Pointer to argument 1 = path to running program
ESP + 8	Pointer to argument 2
ESP + 12	Pointer to argument 3
...	
ESP + 4*arg_number	Pointer to last argument
ESP + 4*arg_number + 1	Four 0 bytes
ESP + 4*arg_number + 2	Pointer to first environment variable
...	

Null-terminated argument strings

```
ASCIIZ string = string bytes followed by 0 byte
MOV EBX, [ESP + 4]    ; EBX ← pointer to arg_1
MOV AL, [EBX]         ; AL ← first byte of arg_1
MOV EDX, [ESP + 8]    ; EDX ← pointer to arg_2
SUB EDX, EBX          ; EDX ← length(ASCIIZ arg_1)
```

Getting Command Line Arguments — 1

Command line

```
~/asm$ prog arg1 arg2 <ENTER>
```

```
%macro sNL
        mov eax, 4  ; write
        mov ebx, 1  ; stdout
        mov ecx, NL ; newline
        mov edx, 1  ; 1 byte
        int 80h
%endmacro
section .data
        addr_buffer: dd 0
        NL: db 10
section .text
global _start
_start: mov ebp, esp
        mov esi, 1
L1:     mov eax, [esp + 4*esi]
        mov [addr_buffer], eax
        mov eax, 4
        mov ebx, 1
        mov ecx, [addr_buffer]
        inc esi
        mov edx, [esp + 4*esi]
        cmp edx, 0
        je L2
        sub edx, ecx
        dec edx
        int 80h
        sNL
        jmp L1
L2:     inc esi
        mov edx, [esp + 4*esi]
        sub edx, ecx
        dec edx
        int 80h
        sNL
        MOV EAX,1
        INT 80h
```

Getting Command Line Arguments — 2

```
_start:
      mov ebp, esp                ; use ebp as base
      mov esi, 1                  ; start with first argument
L1:   mov eax, [esp + 4*esi]      ; get pointer to argument # esi
      mov [addr_buffer], eax      ; save pointer
      mov eax, 4                  ; write system call
      mov ebx, 1                  ; stdout
      mov ecx, [addr_buffer]      ; pointer to argument
      inc esi                     ; esi++
      mov edx, [esp + 4*esi]      ; get pointer to next argument
      cmp edx, 0                  ; check for 0
                                  ; null pointer after last argument
      je L2                       ; break to L2 at end of argument list
      sub edx, ecx                ; length of null-terminated argument
      dec edx                     ; length of argument
      int 80h                     ; Linux system call
      sNL                         ; newline
      jmp L1                      ; continue
```

length of $\mathbf{arg}_N$ = pointer to $\mathbf{arg}_{N+1}$ – pointer to $\mathbf{arg}_N$ – 1

Getting Command Line Arguments — 3

```
L2:  inc esi                      ; if last argument esi++
     mov edx, [esp + 4*esi]       ; get pointer to first environment
                                  ; variable
     sub edx, ecx                 ; length of null-terminated argument
     dec edx                      ; length of argument
     int 80h                      ; Linux system call
     sNL                          ; newline
     MOV EAX, 1                   ; exit program
     INT 80h
```

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

`MOV EAX,[var1 + 4]`

AX ← value following value of variable **`var1`**

Label arithmetic performed at assembly time

Assembler

- Keeps track of addresses
- Performs arithmetic
- Uses numerical results in instructions
  - Example: **`var1`** labels variable defined at **`DS:0006`**
    `MOV AX,[var1 + 2]` coded as **`MOV AX,[0008]`**

EQU Example — 1

```
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
global _start
_start:
        mov eax, message1
        mov ebx, here
        mov ecx, msglen2
mov eax,1
mov ebx,0
int 80h
```

EQU Example — 2

```
~/asm$ debug ex11
-u
080480C0 B8DC900408  MOV     EAX,080490DC    ; pointer to message1
080480C5 BBF1900408  MOV     EBX,080490F1    ; value of here = &('9')+1
080480CA B909000000  MOV     ECX,00000009    ; length(message2) = f1 - e8 = 9
080480CF B801000000  MOV     EAX,00000001
080480D4 BB00000000  MOV     EBX,00000000
080480D9 CD80        INT     80
080480DB 006865      ADD     BYTE [EAX+65],CH
080480DE 6C          INSB
-d 080490DC
080480C0  B8 DC 90 04 08 BB F1 90-04 08 B9 09 00 00 00 B8 ................
080480D0  01 00 00 00 BB 00 00 00-00 CD 80 00 68 65 6C 6C ............hell
080480E0  6F 2C 20 77 6F 72 6C 64-31 32 33 34 35 36 37 38 o, world12345678
080480F0  39 67 6F 6F 64 62 79 65-2C 20 77 6F 72 6C 64 00 9goodbye, world.
```

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

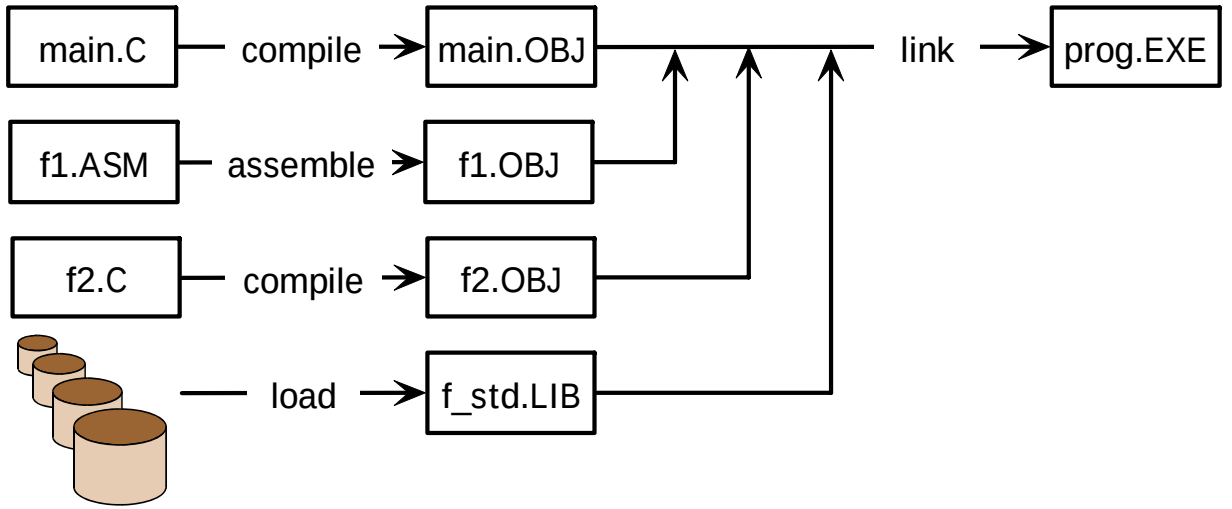
Link object files together to create executable file

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



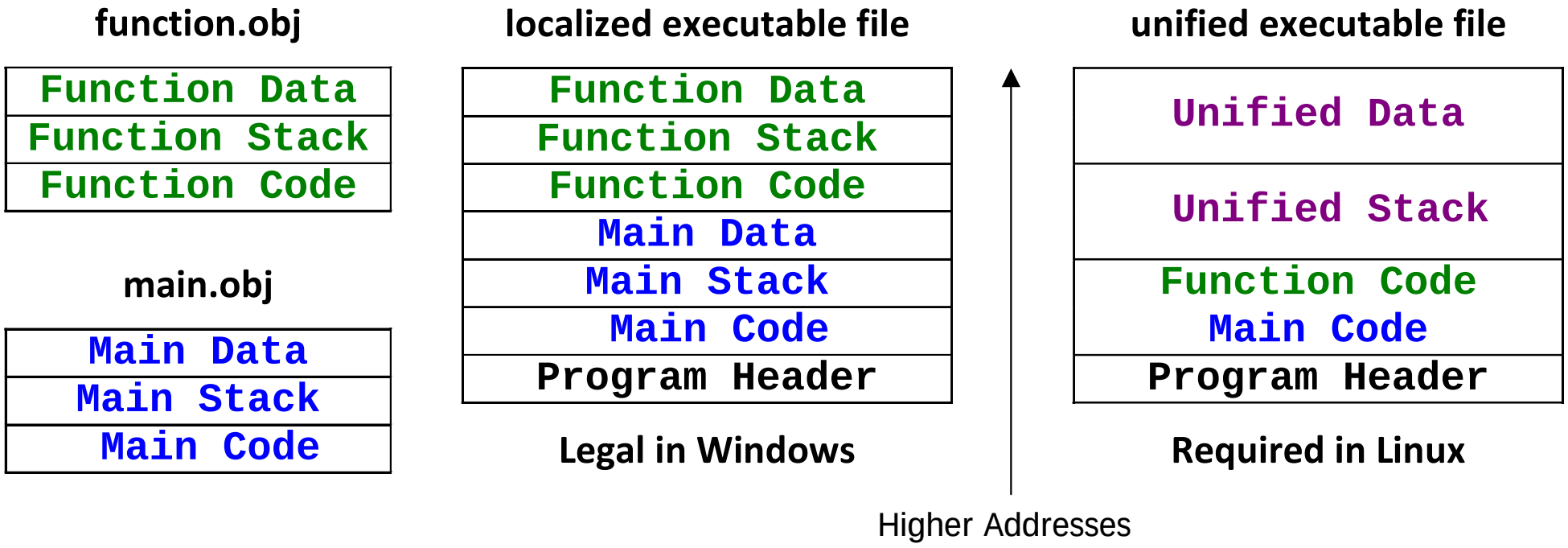
Linking

Store executable program file on disk

- 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



Program Modules and Resolving References

Main module file — main.asm

```
main:       MOV SI,DI
            MOV EAX,[param] ; pass parameter in EAX
            CALL function   ; external reference
                            ; resolved by linker
end:        MOV EAX, 1
            INT 80h
```

Function module file — function.asm

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

Named Sections — 1

Naming sections clarifies modular programming

```
section .data                   ; open .data section for writing
        var1: dd 0x11111111
section .text                   ; close .data section
                                ; open .text section for writing
global _start
_start:
part_1: mov eax, var1
        mov ebx, [var1]
```

Part 1

```
section .data                   ; close .text section
                                ; reopen .data section for writing
        var2: dd 0x22222222
section .text                   ; close .data section
                                ; reopen .text section for writing
part_2: mov eax, var2
        mov ebx, [var2]
mov eax,1
mov ebx,0
int 80h
```

Part 2

Named Sections — 2

```
section .data
        var1: dd 0x11111111
section .text
global _start
_start:
part_1: mov eax, var1
        mov ebx, [var1]
section .data
        var2: dd 0x22222222
section .text
part_2: mov eax, var2
        mov ebx, [var2]
mov eax,1
mov ebx,0
int 80h
```

Linux combines modules into unified executable file

```
ex12:     file format elf32-i386
Contents of section .text:
 80480c0 b8e49004 088b1de4 900408b8 e8900408
 80480d0 8b1de890 0408b801 000000bb 00000000
 80480e0 cd80
Contents of section .data:
 80490e4 11111111 22222222
```

```
Disassembly of section .text:
080480c0 <_start>:
 80480c0:       b8 e4 90 04 08          mov    $0x80490e4,%eax
 80480c5:       8b 1d e4 90 04 08       mov    0x80490e4,%ebx
080480cb <part_2>:
 80480cb:       b8 e8 90 04 08          mov    $0x80490e8,%eax
 80480d0:       8b 1d e8 90 04 08       mov    0x80490e8,%ebx
 80480d6:       b8 01 00 00 00          mov    $0x1,%eax
 80480db:       bb 00 00 00 00          mov    $0x0,%ebx
 80480e0:       cd 80                   int    $0x80
```

Function Call

```
section .data
        var1: dd 0x11111111
section .text
global _start
_start:
pt_1: mov eax, var1
        mov ebx, [var1]
        call pt_2
mov eax,1
mov ebx,0
int 80h
section .data
        var2: dd 0x22222222
section .text
pt_2: mov eax, var2
        mov ebx, [var2]
        ret
```

```
ex13:     file format elf32-i386
Contents of section .text:
 80480c0 b8e89004 088b1de8 900408e8 0c000000
 80480d0 b8010000 00bb0000 0000cd80 b8ec9004
 80480e0 088b1dec 900408c3
Contents of section .data:
 80490e8 11111111 22222222



Disassembly of section .text:
080480c0 <_start>:
 80480c0: b8 e8 90 04 08    mov $0x80490e8,%eax
 80480c5: 8b 1d e8 90 04 08 mov 0x80490e8,%ebx
 80480cb: e8 0c 00 00 00    call 80480dc <pt_2>
 80480d0: b8 01 00 00 00    mov $0x1,%eax
 80480d5: bb 00 00 00 00    mov $0x0,%ebx
 80480da: cd 80             int $0x80
080480dc <pt_2>:
 80480dc: b8 ec 90 04 08    mov $0x80490ec,%eax
 80480e1: 8b 1d ec 90 04 08 mov 0x80490ec,%ebx
 80480e7: c3                ret
```

Separate Module Files — 1

ex14a.asm

```
extern func
section .data
        var1: dd 0x11111111
section .text
global _start
_start:
        mov eax, var1
        mov ebx, [var1]
        call func
mov eax,1
mov ebx,0
int 80h
```

ex14b.asm

```
global func
section .data
        var2: dd 0x22222222
section .text
func:   mov eax, var2
        mov ebx, [var2]
        ret
```

Assemble ex14a and ex14b separately ⟶ link object files into unified executable

```
~/asm$ nasm –f elf ex14a.asm
~/asm$ nasm –f elf ex14b.asm
~/asm$ gcc -nostartfiles -nostdlib ex14a.o ex14b.o -o ex14
```

Separate Module Files — 2

ex14a.asm

```
extern func
section .data
        var1: dd 0x11111111
section .text
global _start
_start:
        mov eax, var1
        mov ebx, [var1]
        call func
mov eax,1
mov ebx,0
int 80h
```

```
ex14:     file format elf32-i386
Disassembly of section .text:
080480c0 <_start>:
 80480c0: b8 ec 90 04 08      mov $0x80490ec,%eax
 80480c5: 8b 1d ec 90 04 08   mov 0x80490ec,%ebx
 80480cb: e8 10 00 00 00      call 80480e0 <func>
 80480d0: b8 01 00 00 00      mov $0x1,%eax
 80480d5: bb 00 00 00 00      mov $0x0,%ebx
 80480da: cd 80               int $0x80
 80480dc: 90 nop
 80480dd: 90 nop
 80480de: 90 nop
 80480df: 90 nop
080480e0 <func>:
 80480e0: b8 f0 90 04 08      mov $0x80490f0,%eax
 80480e5: 8b 1d f0 90 04 08   mov 0x80490f0,%ebx
 80480eb: c3                  ret
```

ex14b.asm

```
global func
section .data
        var2: dd 0x22222222
section .text
func:   mov eax, var2
        mov ebx, [var2]
        ret
```

```
ex14:     file format elf32-i386
Contents of section .text:
 80480c0 b8ec9004 088b1dec 900408e8 10000000
 80480d0 b8010000 00bb0000 0000cd80 90909090
 80480e0 b8f09004 088b1df0 900408c3
Contents of section .data:
 80490ec 11111111 22222222
```

Separate Module Files — 3

ex14a.asm

```
extern func
section .data
        var1: dd 0x11111111
section .text
global _start
_start:
        mov eax, var1
        mov ebx, [var1]
        call func
mov eax,1
mov ebx,0
int 80h
```

ex14b.asm

```
global func
section .data
        var2: dd 0x22222222
section .text
func:   mov eax, var2
        mov ebx, [var2]
        ret
```

Program Loaded into debug

```
080480C0 B8EC900408          MOV       EAX,080490EC
080480C5 8B1DEC900408        MOV       EBX,DWORD [080490EC]
080480CB E810000000          CALL      080480E0
080480D0 B801000000          MOV       EAX,00000001
080480D5 BB00000000          MOV       EBX,00000000
080480DA CD80                INT       80
080480DC 90                  NOP
080480DD 90                  NOP
080480DE 90                  NOP
080480DF 90                  NOP
080480E0 B8F0900408          MOV       EAX,080490F0
080480E5 8B1DF0900408        MOV       EBX,DWORD [080490F0]
080480EB C3                  RETN
080480EC 1111                ADC       DWORD [ECX],EDX
080480EE 1111                ADC       DWORD [ECX],EDX
080480F0 2222                AND       AH ,BYTE [EDX]
080480F2 2222                AND       AH ,BYTE [EDX]
```