Chapter 6

ASSEMBLY LANGUAGE

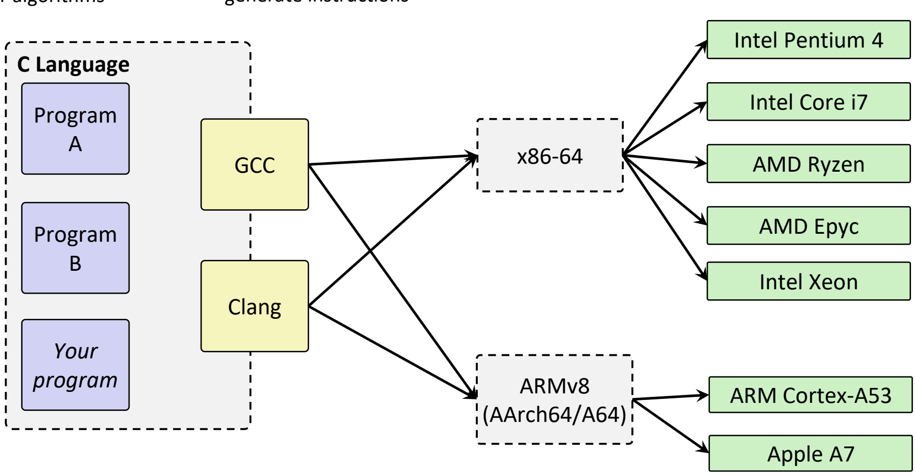
HW Interface Affects Performance

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

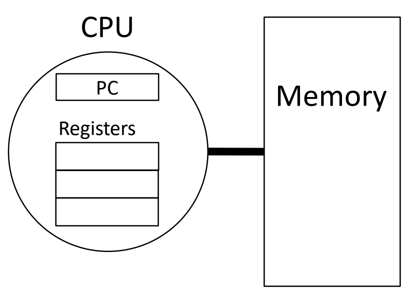
Architecture
Instruction set

Hardware
Different implementations



Instruction Set Architectures

- The ISA defines:
  - The system's state (*e.g.* registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state



General ISA Design Decisions

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded?
- Registers
  - How many registers are there?
  - How wide are they?
- Memory
  - How do you specify a memory location?

Mainstream ISAs

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86-64 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

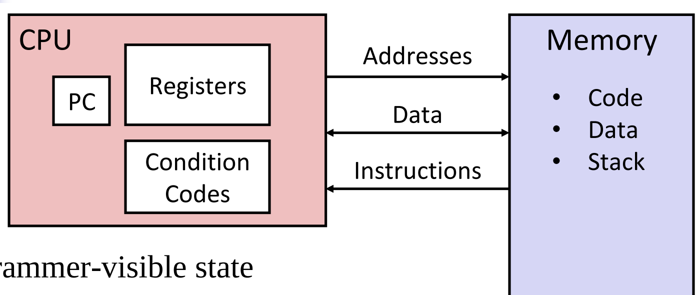
Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

Digital home & networking equipment
(Blu-ray, PlayStation 2)
MIPS Instruction Set

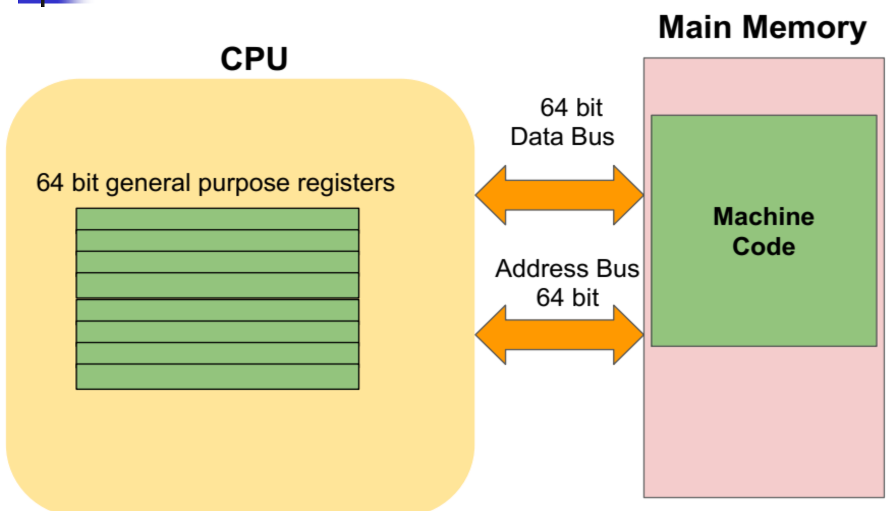
Assembly Programmer's View



- Programmer-visible state
  - PC: the Program Counter (`rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

64 bit x86 systems (x86-64)



x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `xmm1`, `ymm2`)
  - Come from *extensions to x86* (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, …
  - “Intel”: used by Intel documentation, Intel tools, …
  - Must know which you’re reading

x86-64 Integer Registers – 64 bits wide

rax	eax	r8	r8d
rbx	ebx	r9	r9d
rcx	ecx	r10	r10d
rdx	edx	r11	r11d
rsi	esi	r12	r12d
rdi	edi	r13	r13d
rsp	esp	r14	r14d
rbp	ebp	r15	r15d

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

Some History: IA32 Registers – 32 bits wide

general purpose

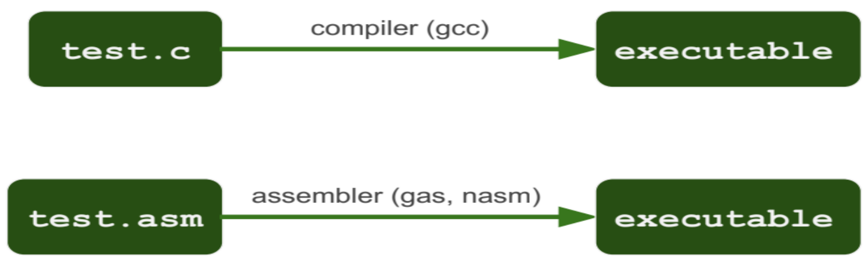
eax	ax	ah	al	*accumulate*
ecx	cx	ch	cl	*counter*
edx	dx	dh	dl	*data*
ebx	bx	bh	bl	*base*
esi	si			*source index*
edi	di			*destination index*
esp	sp			***stack pointer***
ebp	bp			***base pointer***

16-bit virtual registers (backwards compatibility)

Name Origin (mostly obsolete)

What is an Assembler?

- Major Assemblers
  - Microsoft Assembler (MASM)
  - GNU Assembler (GAS)
  - Flat Assembler (FASM)
  - Turbo Assembler (TASM)
  - **Netwide Assembler (NASM)**
- An assembler is a program that translates an assembly language program into binary code



Our platform

- **Hardware:** 80x86 processor (**32**, 64 bit)
- **OS:** Linux
- **Assembler:** Netwide Assembler (NASM)
- **C Compiler:** GNU C Compiler (GCC)
- **Linker:** GNU Linker (LD)
- We will use the NASM assembler, as it is:
  - Free. You can download it from various web sources.
  - Well-documented and you will get lots of information on net.
  - Could be used on both Linux and Windows.

Introduction to NASM assembler

- NASM Command Line Options
  - -h for usage instructions
  - -o output file name
  - -f output file format
    - Must be coff always
  - -l generate listing file, i.e. file with code generated
  - -e preprocess only
  - -g enable debugging information
- Example
  nasm -g -f coff foo.asm -o foo.o

Base elements of NASM Assemble

- Character Set
  - Letters **a**..**z** **A**..**Z**
  - Digits **0**..**9**
  - Special characters **? _ @ $.** ~
- NASM (unlike most assemblers) is case-sensitive with respect to labels and variables
- It is not case-sensitive with respect to keywords, mnemonics, register names, directives, etc.

Literals

- Literals are values that are known or calculated at assembly time. Examples:
  - 'This is a string constant‘
  - "So is this“
  - ‘Backquoted strings can use escape chars\n‘
  - 123
  - 1.2
  - 0FAAh
  - $1A01
  - 0x1A01

Integers

- Numeric digits (including **A**..**F**) with no decimal point
- may include radix specifier at end:
  - **b** or **y** binary
  - **d** decimal
  - **h** hexadecimal
  - **q** octal
- Examples
  - **200** decimal (default)
  - **200d** decimal
  - **200h** hex
  - **200q** octal
  - **10110111b** binary

NASM Syntax

- In order to refer to the contents of a memory location, use square brackets.
- In order to refer to the address of a variable, leave them out, e.g.,
  - **mov** *eax*, bar ;Refers to the address of bar
  - **mov** *eax*, [bar] ;Refers to the contents of bar

  No need for the OFFSET directive.
- NASM does not support the hybrid syntaxes such as:
  - **mov** *eax*,table[*ebx*] ;ERROR
  - **mov** *eax*,[table+*ebx*] ;O.K
  - **mov** *eax*,[*es*:*edi*] ;O.K
- NASM does NOT remember variable types:
  - data dw 0 ;Data type defi ned as double word.
  - mov [data], 2 ;Doesn't work.
  - **mov** word [data], 2 ;O.K

- NASM does NOT remember variable types. Therefore, un-typed operations are not supported, e.g.
  LODS, MOVS, STOS, SCAS, CMPS, INS, and OUTS.
- You must use instead:
  LODSB, MOVSW, and SCASD, etc.
- NASM does not support ASSUME.
  It will not keep track of what values you choose to put in your segment registers.
- NASM does not support memory models.
- The programmer is responsible for coding CALL FAR instructions where necessary when calling external functions.
  call (**seg** procedure):proc ;call segment:offset
- *seg* returns the segment base of procedure *proc*.

- NASM does not support memory models.
  - The programmer has to keep track of which functions are supposed to be called with a *far call* and which with a *near call*, and is responsible for putting the correct form of RET instruction (RETN or RETF).
- NASM uses the names *st0*, *st1*, etc. to refer to floating point registers.
- NASM's declaration syntax for un-initialized storage is different.
  - stack **DB** 64 **DUP** (?) ;ERROR
  - stack **resb** 64 ;Reserve 64 bytes
- Macros and directives work differently than they do in MASM

Statemenmts

- Syntax:
  [label[:]] [mnemonic] [operands] [;comment]
  - **[ ]** indicates optionality
  - Note that **all** parts are optional → blank lines are legal
  - **[label]** can also be **[name]**
    - *Variable names are used in data definitions*
    - *Labels are used to identify locations in code*
  - Statements are free form; they need not be formed into columns
  - Statement must be on a single line, max 128 chars

- Example:
  - L100: add eax, edx ; add subtotal to total
- Labels often appear on a separate line for code clarity:
  - L100:
    add eax, edx ; add subtotal to total

Labels and Names

- Names identify labels, variables, symbols, and keywords
- May contain:
  - letters: **a**..**z** **A**..**Z**
  - digits: **0**..**9**
  - special chars: **? _ @ $.** ~
- NASM is case-sensitive (unlike most x86 assemblers)
- First character must be a letter, _ or . (which has a special meaning in NASM as a "local label" indicating it can be redefined)
- Names cannot match a reserved word (and there are many reserved words!)

Type of statements

- 1. Directives
  - limit EQU 100 ; defines a symbol limit
  - % define limit 100 ; like C #define
- 2. Data Definitions
  - msg db 'Welcome to Assembler!‘
  - db 0Dh, 0Ah
  - count dd 0
  - mydat dd 1,2,3,4,5
  - resd 100 ; reserves 400 bytes
- 3. Instructions
  - mov eax, ebx
  - add ecx, 10

Directives

- A directive is an instruction to the assembler, not the CPU
- A directive is not an executable instruction
- A directive can be used to
  - define a constant
  - define memory for data
  - include source code & other file
  - They are similar to C’s #include and #define

- equ directive : EQU defi nes a symbol to a constant
  - format: symbol equ value
  - Defines a symbol
  - Cannot be redefined later
  - Examples : message db 'hello, world'
    msglen equ $-message
- % directive
  - format: %define symbol value
  - Similar to #define in C
  - Example : **%define N 100**
    mov eax , N

- Including files
  - %include "some_file"
- If you know the C preprocessor, these are the same ideas as
  - #define SIZE 100 or #include "stdio.h

Data formats

- Defines storage for uninitialized or uninitialized data
- Double and single quotes are treated the same

Unit	Letter(X)	Size in bytes
byte	B	1
word	W	2
double word	D	4
quad word	Q	8
ten bytes	T	10

There are two kinds of data directives

- **RESx** directive; **x** is one of **b, w, d, q, t** REServe memory (uninitialized data)
- **Dx** directive; **x** is one of **b, w, d, q, t** Define memory (initialized data)
- Example :
  - L1 db 0 ;defines a byte and initializes to 0
  - L2 dw FF0Fh ;define a word and initialize to FF0Fh
  - L3 db "A" ;byte holding ASCII value of A
  - L4 resd 100 ;reserves space for 100 double words
  - L5 times 100 db 0 ;defines 100 bytes init. to 0
  - L6 db "s","t","r","i","n","g",0 ;defines "string“
  - L7 db ’string’,0 ;same as above
  - L8 resb 10 ; reserves 10 bytes

The DX data directives

- One declares a zone of initialized memory using three elements:
  - Label: the name used in the program to refer to that zone of memory
    - A pointer to the zone of memory, i.e., an address
  - DX, where X is the appropriate letter for the size of the data being declared
  - Initial value, with encoding information
    - default: decimal
    - b: binary
    - h: hexadecimal
    - o: octal
    - quoted: ASCII
- Example : L8 db 0, 1, 2, 3

- Examples
  - mov al , [L2] ;move a byte at L2 to al
  - mov eax, L2 ;move the address of L2 to eax
  - mov [L1], ah ;move ah to the byte pointed to by L1
  - mov eax, dword 5
  - add [L2], eax ;double word at L2 containing [L2]+eax
  - mov [L2], 1 ;does not work, why?
  - mov dword [L2], 1 ;works, why

DX with the times qualifier

- Say you want to declare 100 bytes all initialized to 0
- NASM provides a nice shortcut to do this, the "times" qualifier
  - L11 times 100 db 0
  - Equivalent to L11 db 0,0,0,....,0 (100 times)

NASM directives

- BITS 32 generate code for 32 bit processor mode
- CPU 386 | 686 | ... restrict assembly to the specified processor
- SECTION <section_name>

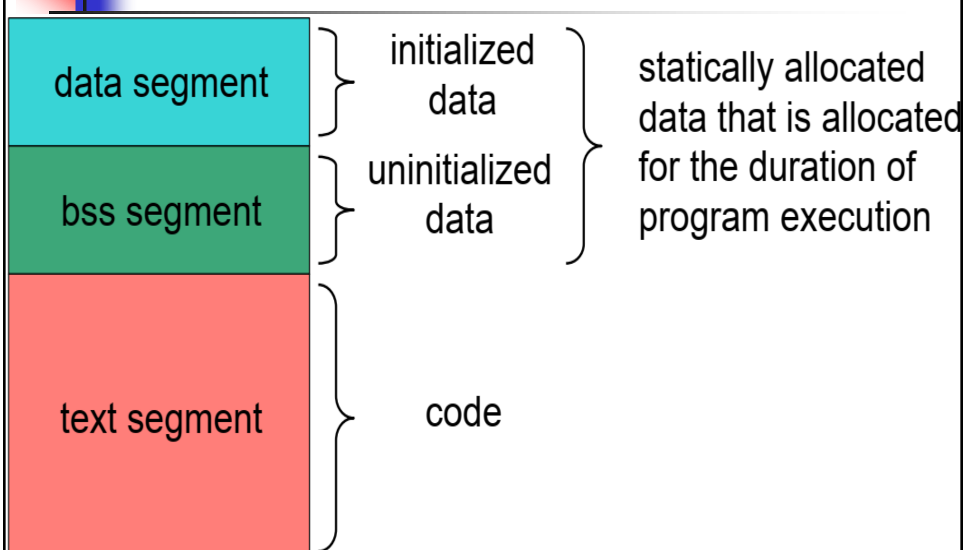
specifies the section the assembly code will be assembled into. For COFF can be one of:

  - .text code (program) section
  - .data initialized data section
  - .bss uninitialized data section
- EXTERN <symbol> declare <symbol> as declared elsewhere, allowing it to be used in the module;
- GLOBAL <symbol> declare <symbol> as global so that it can be used in other modules that import it via EXTERN

Examples using $

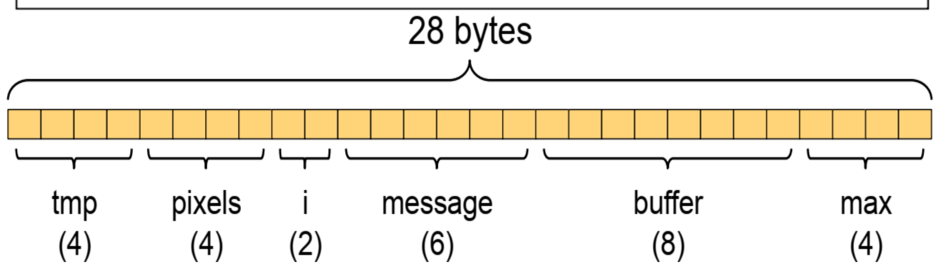
- message db ’hello, world’
- msglen equ $-message
- **Note**
  - The msglen is evaluated **once** using the value of $ at the point of definition
  - $ evaluates to the assembly position at the beginning of the line containing the expression

NASM Program Structure



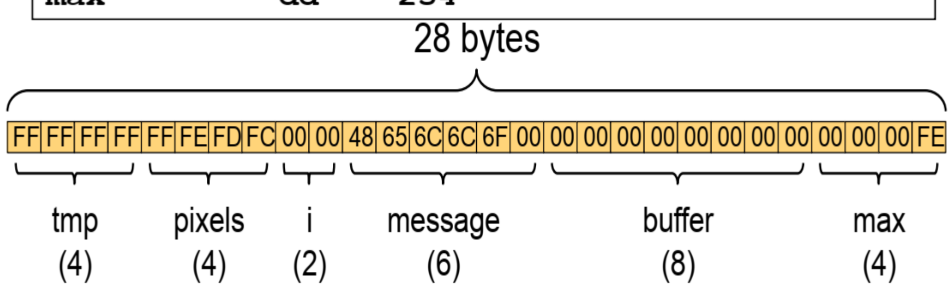
Data segment example

```
tmp         dd      -1
pixels      db      0FFh, 0FEh, 0FDh, 0FCh
i           dw      0
message     db      "H", "e", "llo", 0
buffer      times   8     db   0
max         dd      254
```



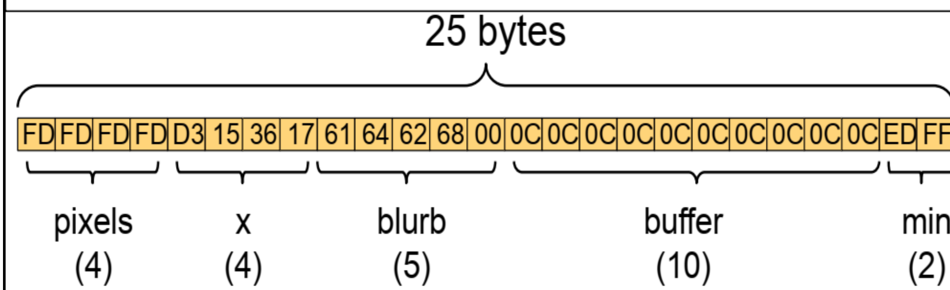
Data segment example

```
tmp         dd      -1
pixels      db      0FFh, 0FEh, 0FDh, 0FCh
i           dw      0
message     db      "H", "e", "llo", 0
buffer      times   8     db   0
max         dd      254
```



Example

```
pixels      times 4     db     0FDh
x           dd     00010111001101100001010111010011b
blurb       db     "ad", "b", "h", 0
buffer      times  10   db   14o
min         dw     -19
```



Uninitialized Data

- The RESX directive is very similar to the DX directive, but *always specifies* the number of memory elements
- L20 resw 100
  - 100 uninitialized 2-byte words
  - L20 is a pointer to the first word
- L21 resb 1
  - 1 uninitialized byte named L21

Moving immediate values

- Consider the instruction: mov [L], 1
  - The assembler will give us an error: "operation size not specified"!
  - This is because the assembler has no idea whether we mean for "1" to be 01h, 0001h, 00000001h, etc.
    - Labels have no type (they're NOT variables)
- Therefore the assembler must provide us with a way to specify the size of immediate operands
  - mov dword [L], 1
  - 4-byte double-word
- 5 size specifiers: byte, word, dword, qword, tword

Size Specifier Examples

- mov [L1], 1 ; Error Vì không biết size dữ liệu
- mov byte [L1], 1 ; 1 byte
- mov word [L1], 1 ; 2 bytes
- mov dword [L1], 1 ; 4 bytes
- mov [L1], eax ; 4 bytes
- mov [L1], ax ; 2 bytes
- mov [L1], al ; 1 byte
- mov eax, [L1] ; 4 bytes
- mov ax, [L1] ; 2 bytes
- mov ax, 12 ; 2 bytes

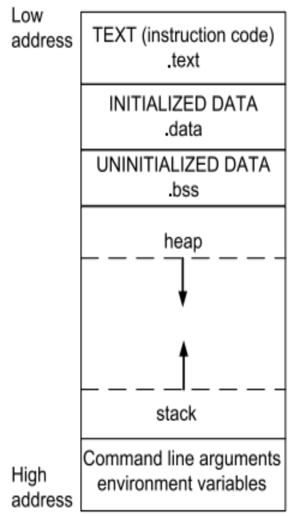
Program structure

```
SECTION .data                          ;data section
    msg: db "Hello World",10           ;the string to print 10=newline
    len: equ $-msg                     ;len is value, not an addr.
SECTION .text                          ;code section
    global main                        ;for linker
main:                                  ;standard gcc entry point
    mov edx, len                       ;arg3, len of str. to print
    mov ecx, msg                       ;arg2, pointer to string
    mov ebx, 1                         ;arg1, write to screen
    mov eax, 4                 ;write sysout command to int 80 hex
    int 0x80                   ;interrupt 80 hex, call kernel
    mov ebx, 0                 ;exit code, 0=normal
    mov eax, 1                 ;exit command to kernel
    int 0x80                   ;interrupt 80 hex, call kernel
```

- To produce hello.o object file:
  nasm -f elf hello.asm
- To produce hello ELF executable:
  ld -s -o hello hello.o

Program layout

- Consit of 3 parts:
  - Text
  - Data
  - Bss



BSS came from "Block Started by Symbol", an assembler for IBM 704 in the 1950s.

NASM Program Structure

- ; include directives thư viện,...
  segment .data chỉ thị
  ; DX directives không khởi động giá trị ban đầu
-
  segment .bss
  ; RESX directives
-
  segment .text
  global asm_main
  asm_main:
  ; instructions

More on the text segment

- Before and after running the instructions of your program there is a need for some "setup" and "cleanup"
- We'll understand this later, but for now, let's just accept the fact that your text segment will always looks like this:

```
enter 0,0
pusha
;
; Your program here
;
popa
mov eax, 0
leave
ret
```

Assembly program structure

```
%include "asm_io.inc"
        segment .data                   ;initialized data

        segment .bss                    ;uninitialized data

        segment .text
        global asm_main
asm_main:
        enter 0,0                       ;setup
        pusha                           ;save all registers
        ;put your code here
        popa                            ;restore all registers
        mov eax, 0                      ;return value
        leave
        ret
```

OR NASM Skeleton File

```
; include directives
segment .data
        ; DX directives
segment .bss
        ; RESX directives
segment .text
        global asm_main
asm_main:
        enter 0,0
        pusha
        ; Your program here
        popa
        mov eax, 0
        leave
        ret
```

Example

```
segment .data
    integer1 dd 15 ; first int
    integer2 dd 6 ; second int
segment .bss
    result resd 1 ; result
segment .text
global asm_main
asm_main:
    enter 0,0
    pusha
```

```
mov eax, [integer1] ; eax = int1
add eax, [integer2] ; eax = int1 + int2
mov [result], eax ; result = int1 + int2
popa
mov eax, 0
leave
ret
```

I/O?

- This is all well and good, but it's not very interesting if we can't "see" anything
- We would like to:
  - Be able to provide input to the program
  - Be able to get output from the program
- Also, debugging will be difficult, so it would be nice if we could tell the program to print out all register values, or to print out the content of some zones of memory
- Doing all this requires quite a bit of assembly code and requires techniques that we will not see for a while
- The author of our textbook provides a nice I/O package that we can just use, without understanding how it works for now

asm_io.asm and asm_io.inc

- The “PC Assembly Language” book comes with many add-ons and examples
  - Downloadable from the course’s Web site
- A very useful one is the I/O package, which comes as two files:
  - asm_io.asm (assembly code)
  - asm_io.inc (macro code)
- Simple to use:
  - Assemble asm_io.asm into asm_io.o
  - Put “%include asm_io.inc” at the top of your assembly code
  - Link everything together into an executable

I/O

- C: I/O done through the standard C library
- NASM: I/O through the standard C library
  %include "asm_io.inc“
- Contains routines for I/O
  - **print_int** prints **EAX**
  - **print_char** prints ASCII value of **AL**
  - **print_string** prints the string stored at the address stored in **EAX**; must be 0 terminated
  - **print_nl** prints newline
  - **read_int** reads an integer into **EAX**
  - **read_char** reads a character into **AL**

Examples

```
%include "asm_io.inc"

segment .data
      integer1          dd    15      ; first int
      integer2          dd     6      ; second int
segment .bss
      result            resd  1       ; result
segment .text
            global asm_main
      asm_main:
            enter       0,0
            pusha
            mov         eax, [integer1]          ; eax = int1
            add         eax, [integer2]          ; eax = int1 + int2
            mov         [result], eax            ; result = int1 + int2
            call        print_int                ; print result
            popa
            mov         eax, 0
```

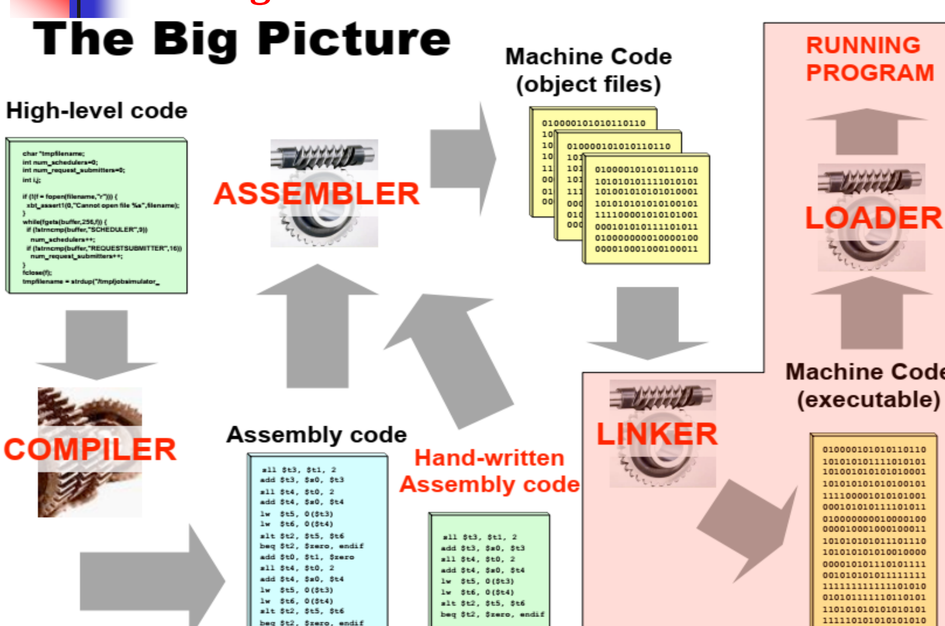
Modified example

```
%include "asm_io.inc"

segment .data
        msg1      db              "Enter a number: ", 0
        msg2      db              "The sum of ", 0
        msg3      db              " and ", 0
        msg4      db              " is: ", 0
segment .bss
        integer1   resd    1      ; first integer
        integer2   resd    1      ; second integer
        result      resd    1     ; result
segment .text
        global asm_main
asm_main:
        enter     0,0
        pusha
        mov       eax, msg1           ; note that this is a pointer!
        call      print_string
        call      read_int            ; read the first integer
        mov       [integer1], eax   ; store it in memory
        mov       eax, msg1           ; note that this is a pointer!
        call      print_string
        call      read_int            ; read the second integer
        mov       [integer2], eax   ; store it in memory
```

```
mov     eax, [integer1]   ; eax = first integer
add     eax, [integer2]   ; eax += second integer
mov     [result], eax     ; store the result
mov     eax, msg2         ; note that this is a pointer
call    print_string
mov     eax, [integer1]   ; note that this is a value
call    print_int
mov     eax, msg3         ; note that this is a pointer
call    print_string
mov     eax, [integer2]   ; note that this is a value
call    print_int
mov     eax, msg4         ; note that this is a pointer
call    print_string
mov     eax, [result]     ; note that this is a value
call    print_int
call    print_nl
popa
mov     eax, 0
leave
ret
```

The Big Picture



How do we run the program?

- Now that we have written our program, say in file vd1.asm using a text editor, we need to assemble it
- When we assemble a program we obtain an object file (vd1.o file)
- We use NASM to produce the .o file:
  % nasm **-f elf** vd1.asm **-o** vd1.o
- So now we have vd1.o file, that is a machine code translation of our assembly code
- We also need driver.o file for the C driver:
  % gcc -m32 -c driver.c -o driver.o
  - We generate a 32-bit object (our machines are likely 64-bit)
- We also create asm_io.o by assembling asm_io.asm
- Now we have three .o files.
- We link them together to create an executable:
  % gcc driver.o vd1.o asm_io.o -o first_vd1

First program

- ;
  ; file: first.asm
  ; First assembly program. This program asks for two integers as
  ; input and prints out their sum.
  ;
  ; To create executable:
  ;
  ; Using Linux and gcc:
  ; nasm -f elf first.asm
  ; gcc -o first first.o driver.c asm_io.o

```
%include "asm_io.inc" ;
; initialized data is put in the .data segment
segment .data
;
; These labels refer to strings used for output
prompt1 db "Enter a number: ", 0 ; don't forget null
prompt2 db "Enter another number: ", 0
outmsg1 db "You entered ", 0
outmsg2 db " and ", 0
outmsg3 db ", the sum of these is ", 0
; uninitialized data is put in the .bss segment
;
segment .bss
```

```
;
; These labels refer to double words used to store the inputs;
;
input1 resd 1
input2 resd 1
; code is put in the .text segment
segment .text
      global asm_main
asm_main:
      enter 0,0 ; setup routine
      pusha
      mov eax, prompt1 ; print out prompt
      call print_string
      call read_int ; read integer
      mov [input1], eax ; store into input1
      mov eax, prompt2 ; print out prompt
```

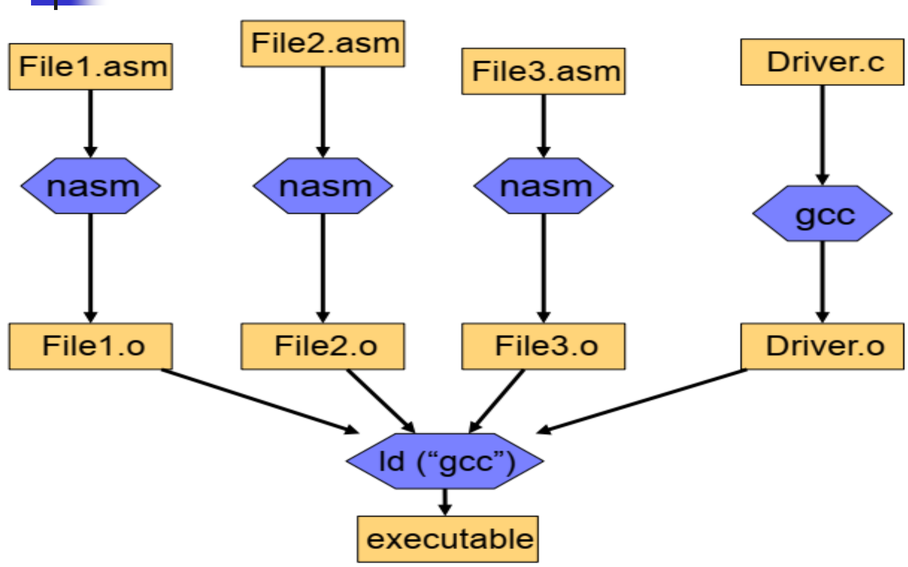
```
call print_string
call read_int ; read integer
mov [input2], eax ; store into input2
mov eax, [input1] ; eax = dword at input1
add eax, [input2] ; eax += dword at input2
mov ebx, eax ; ebx = eax
dump_regs 1 ; dump out register values
dump_mem 2, outmsg1, 1 ; dump out memory
; next print out result message as series of steps
mov eax, outmsg1
call print_string ; print out first message
mov eax, [input1]
call print_int ; print out input1
mov eax, outmsg2
call print_string ; print out second message
mov eax, [input2]
```

```
        print_int ; print out input2
        mov eax, outmsg3
        call print_string ; print out third message
        mov eax, ebx
        call print_int ; print out sum (ebx)
        call print_nl ; print new-line
        popa
        mov eax, 0 ; return value
leave
ret
```

C driver

- ```
  #include "cdecl.h"
  int PRE_CDECL asm_main( void ) POST_CDECL;
  int main() {
      int ret_status;
      ret_status = asm_main();
      return ret_status;
      }
  ```
- All segments and registers are initialized by the C system
- I/O is done through the C standard library
- Initialized data in .data
- Uninitialized data in .bss
- Code in .text
- Stack later

The Big Picture

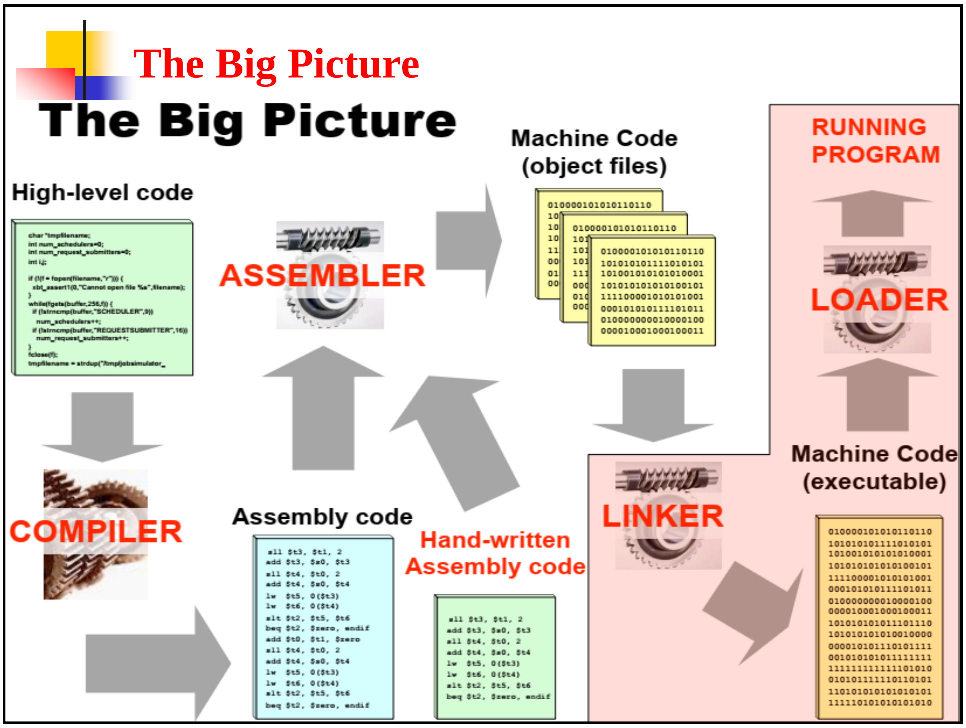

  ```

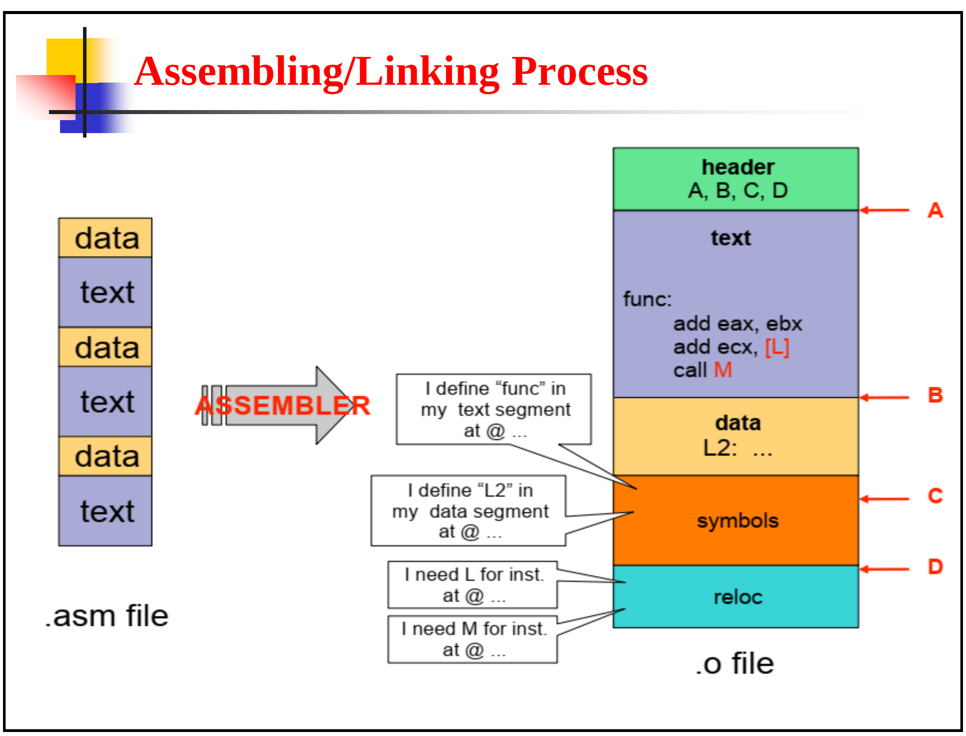
Compiling

- **nasm -f elf first.asm**
  produces **first.o**
- ELF: executable and linkable format
- **gcc -c driver.c**
  produces **driver.o**
  option **-c** means compile only
- We need to compile **asm_io.asm**:
  nasm -f elf -d ELF_TYPE asm_io.asm
  produces **asm_io.o**
- On 64-bit machines, add the option **-m32** to generate 32-bit code, e.g. **gcc -m32 -c driver.c**

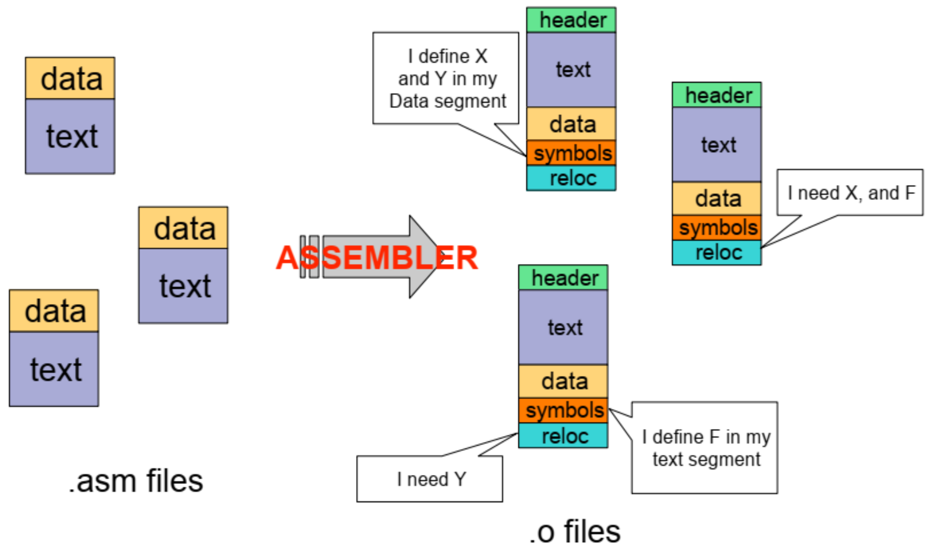
Linking

- Linker: combines machine code & data in object files and libraries together to create an executable
- **gcc -o first driver.o first.o asm_io.o**
- On 64-bit machines,
  gcc -m32 -o first driver.o first.o asm_io.o
- **-o outputfile** specifies the output file
- **gcc driver.o first.o asm_io.o**
  produces **a.out** by default

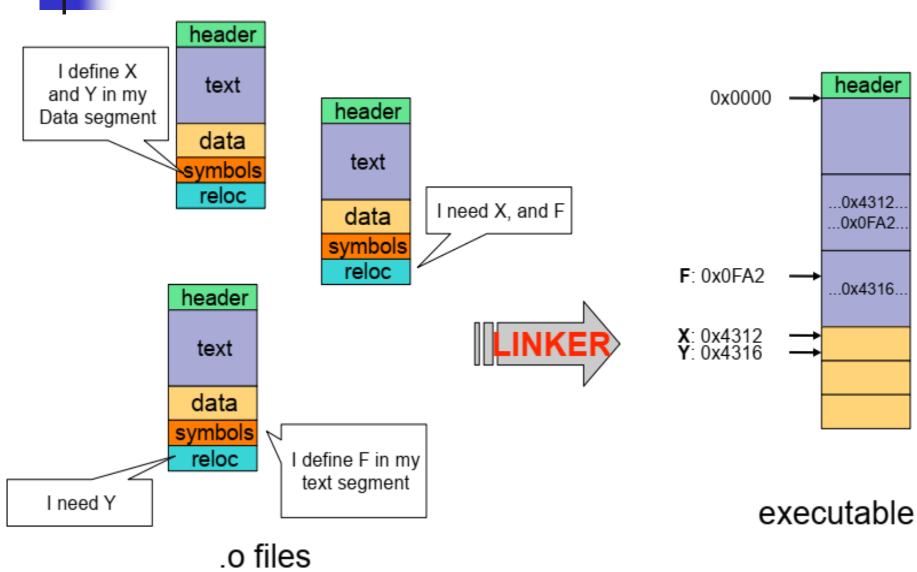

The Big Picture
The Big Picture
High-level code
COMPILER
Assembly code
ASSEMBLER
Hand-written
Assembly code
Machine Code
(object files)
LINKER
Machine Code
(executable)
LOADER
RUNNING
PROGRAM


Assembling/Linking Process
data
text
data
text
data
text
.asm file
ASSEMBLER
I define "func" in
my text segment
at @ ...
I define "L2" in
my data segment
at @ ...
I need L for inst.
at @ ...
I need M for inst.
at @ ...
header
A, B, C, D
text
func:
add eax, ebx
add ecx, [L]
call M
data
L2: ...
symbols
reloc
A
B
C
D
.o file

Assembling/Linking Process



Assembling/Linking Process



dum_regs and dump_mem

- The macro dump_regs prints out the bytes stored in all the registers (in hex), as well as the bits in the FLAGS register (only if they are set to 1)

  dump_regs 13
  - '13' above is an arbitrary integer, that can be used to distinguish outputs from multiple calls to dump_regs
- The macro dump_memory prints out the bytes stored in memory (in hex). It takes three arguments:
  - An arbitrary integer for output identification purposes
  - The address at which memory should be displayed
  - The number minus one of 16-byte segments that should be displayed
  - for instance
    dump_mem 29, integer1, 3
  - prints out "29", and then (3+1)*16 bytes

- To demonstrate the usage of these two macros, let's just write a program that highlights the fact that the Intel x86 processors use Little Endian encoding
- We will do something ugly using 4 bytes
- Store a 4-byte hex quantity that corresponds to the ASCII codes: "live"
  - "l" = 6Ch
  - "i" = 69h
  - "v" = 76h
  - "e" = 65h
- Print that 4-byte quantity as a string

Example

```
%include "asm_io.inc"

segment .data
      bytes   dd            06C697665h  ; "live"
      end     db            0             ; null

segment .text
      global asm_main
asm_main:
      enter                 0,0
      pusha
      mov                   eax, bytes      ; note that this is an address
      call                  print_string    ; print the string at that address
      call                  print_nl        ; print a new line
      mov                   eax, [bytes]    ; load the 4-byte value into eax
      dump_mem              0, bytes, 1     ; display the memory
      dump_regs             0               ; display the registers
      pusha
      popa
      mov                   eax, 0
      leave
      ret
```

Output of the program

The program prints "evil" and not "live"

The address of "bytes" is 0804A020"

"bytes" starts here

```
evil
Memory Dump # 0 Address = 0804A020
0804A020 65 76 69 6C 00 00 00 00 25 69 00 25 73 00 52 65 "evil????%i?%s?Re"
0804A030 67 69 73 74 65 72 20 44 75 6D 70 20 23 20 25 64 "gister Dump # %d"
Register Dump # 0
EAX = 6C697665 EBX = B7747FF4 ECX = BFBCB2C4 EDX = BFBCB254
ESI = 00000000 EDI = 00000000 EBP = BFBCB208 ESP = BFBCB1E8
EIP = 080484A4 FLAGS = 0282        SF
```

and yes, it's "evil"

The "dump" starts at address 0804A020 (a multiple of 16)

bytes in eax are in the "live" order