(Advanced) Computer Architechture

Prof. Dr. Hasan Hüseyin BALIK

(7th Week)

Outline

3. Instruction sets
   - Instruction Sets: Characteristics and Functions
   - Instruction Sets: Addressing Modes and Formats
   - Assembly Language and Related Topics

3.3 Assembly Language and Related Topics

3.3 Outline

- Assembly Language Concepts
- Motivation for Assembly Language Programming
- Assembly Language Elements
- Examples
- Types of Assemblers
- Assemblers
- Loading and Linking

Assembler

A program that translates assembly language into machine code.

Assembly Language

A symbolic representation of the machine language of a specific processor, augmented by additional types of statements that facilitate program writing and that provide instructions to the assembler.

Compiler

A program that converts another program from some source language (or programming language) to machine language (object code). Some compilers output assembly language which is then converted to machine language by a separate assembler. A compiler is distinguished from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions. A compiler may support such features as automatic allocation of variables, arbitrary arithmetic expressions, control structures such as FOR and WHILE loops, variable scope, input/output operations, higher-order functions and portability of source code.

Executable Code

The machine code generated by a source code language processor such as an assembler or compiler.
This is software in a form that can be run in the computer.

Instruction Set

The collection of all possible instructions for a particular computer; that is, the collection of machine language instructions that a particular processor understands.

Linker

A utility program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code.

Loader

A program routine that copies an executable program into memory for execution.

Machine Language, or Machine Code

The binary representation of a computer program which is actually read and interpreted by the computer. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings which may be either all the same size (e.g., one 32-bit word for many modern RISC microprocessors) or of different sizes.

Object Code

The machine language representation of programming source code. Object code is created by a compiler or assembler and is then turned into executable code by the linker.

Key Terms For This Week

Programming the Statement n = *i* + *j*+ *k*

Address	Contents			
	Opcode		Operand	
101	0010	0010	1100	1001
102	0001	0010	1100	1010
103	0001	0010	1100	1011
104	0011	0010	1100	1100
201	0000	0000	0000	0010
202	0000	0000	0000	0011
203	0000	0000	0000	0100
204	0000	0000	0000	0000

(a) Binary program

Address	Contents
101	22C9
102	12CA
103	12CB
104	32CC
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	0002
202	DAT	0003
203	DAT	0004
204	DAT	0000

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Motivation for Assembly Language Programming

- Assembly language is a programming language that is one step away from machine language
- Typically each assembly language instruction is translated into one machine instruction by the assembler
- Assembly language is hardware dependent, with a different assembly language for each type of processor
- Assembly language instructions can make reference to specific registers in the processor, include all of the opcodes of the processor, and reflect the bit length of the various registers of the processor and operands of the machine language
  - Therefore, an assembly language programmer must understand the computer's architecture

Assembly Language Programming (1 of 2)

Disadvantages

- The disadvantages of using an assembly language rather than an HLL include:
  - Development time
  - Reliability and security
  - Debugging and verifying
  - Maintainability
  - Portability
  - System code can use intrinsic functions instead of assembly
  - Application code can use intrinsic functions or vector classes instead of assembly
  - Compilers have been improved a lot in recent years

Assembly Language Programming (2 of 2)

Advantages

- Advantages to the occasional use of assembly language include:
  - Debugging and verifying
  - Making compilers
  - Embedded systems
  - Hardware drivers and system code
  - Accessing instructions that are not accessible from high-level language
  - Self-modifying code
  - Optimizing code for size
  - Optimizing code for speed
  - Function libraries
  - Making function libraries compatible with multiple compilers and operating systems

Assembly Language vs. Machine Language

- The terms *assembly language* and *machine language* are sometimes, erroneously, used synonymously
- Machine language:
  - Consists of instructions directly executable by the processor
  - Each machine language instruction is a binary string containing an opcode, operand references, and perhaps other bits related to execution, such as flags
  - For convenience, instead of writing an instruction as a bit string, it can be written symbolically, with names for opcodes and registers
- Assembly language:
  - Makes much greater use of symbolic names, including assigning names to specific main memory locations and specific instruction locations
  - Also includes statements that are not directly executable but serve as instructions to the assembler that produces machine code from an assembly language program

Assembly-Language Statement Structure

label:	**mnemonic**	**operand(s)**	**;comment**
optional	opcode name or directive name or macro name	zero or more	optional

Statements (1 of 3)

Label

- If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded
- The programmer may subsequently use the label as an address or as data in another instruction's address field
- The assembler replaces the label with the assigned value when creating an object program
- Labels are most frequently used in branch instructions
- Reasons for using a label:
  - Makes a program location easier to find and remember
  - Can easily be moved to correct a program
  - Programmer does not have to calculate relative or absolute memory addresses, but just uses labels as needed

Statements (2 of 3)

Mnemonic

- The mnemonic is the name of the operation or function of the assembly language statement
- In the case of a machine instruction, a mnemonic is the symbolic name associated with a particular opcode

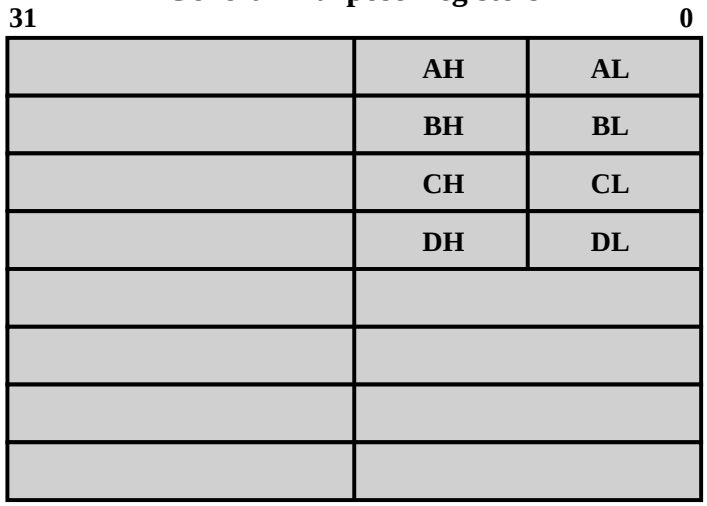
Statements (3 of 3)

Operands

- An assembly language statement includes zero or more operands
- Each operand identifies an immediate value, a register value, or a memory location
- Typically the assembly language provides conventions for distinguishing among the three types of operand references, as well as conventions for indicating addressing mode

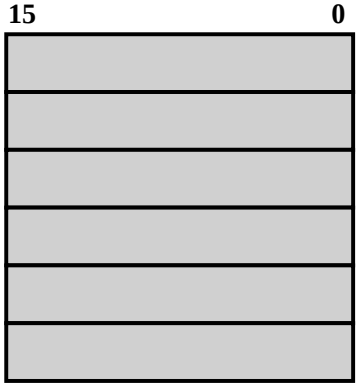
Intel x86 Program Execution Registers

Generall-Purpose Registers



16-bit	32-bit
AX	EAX (000)
BX	EBX (011)
CX	ECX (001)
DX	EDX (010)
	ESI (110)
	EDI (111)
	EBP (101)
	ESP (100)

Segment Registers



CS
DS
SS
ES
FS
GS

Statements (1 of 2)

Comment

- All assembly languages allow the placement of comments in the program
- A comment can either occur at the right-hand end of an assembly statement or can occupy and entire test line
- The comment begins with a special character that signals to the assembler that the rest of the line is a comment and is to be ignored by the assembler
- Typically, assembly languages for the x86 architecture use a semicolon (;) for the special character

Statements (2 of 2)

Pseudo-instructions

- Pseudo-instructions are statements which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them
- Pseudo-instructions are not directly translated into machine language instructions
- Instead, directives are instructions to the assembler to perform specified actions during the assembly process
- Examples include:
  - Define constants
  - Designate areas of memory for data storage
  - Initialize areas of memory
  - Place tables or other fixed data in memory
  - Allow references to other programs

Assembly-Language Directives

(a) Letters for RESx and Dx Directives

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

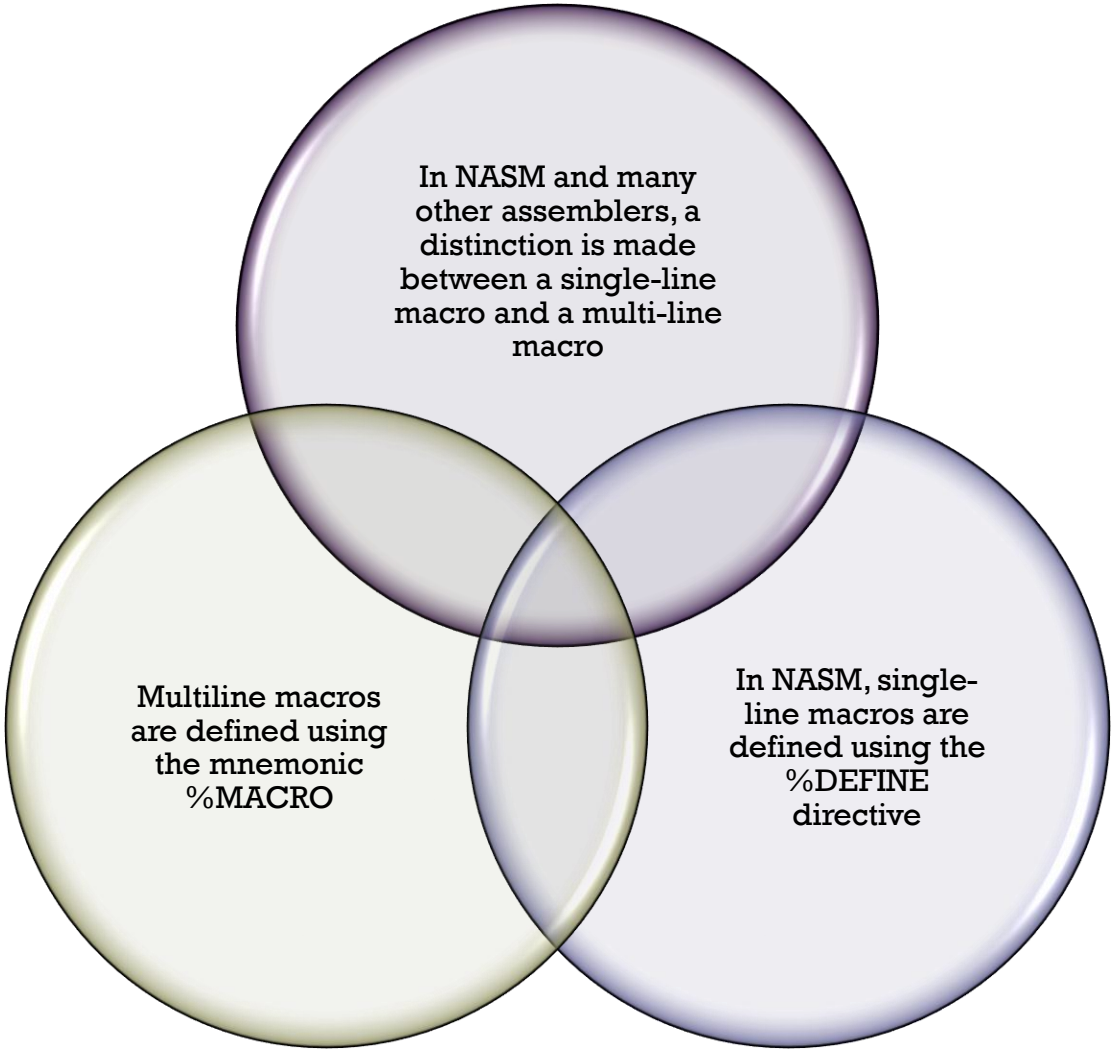
(b) Directives

Name	Description	Example
DB, DW, DD, DQ, DT	Initialize locations	`L6 DD 1A92H` `;doubleword at L6 initialized to 1A92H`
RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	`BUFFER RESB 64` `;reserve 64 bytes starting at BUFFER`
INCBIN	Include binary file in output	`INCBIN "file.dat" ; include this file`
EQU	Define a symbol to a given constant value	`MSGLEN EQU 25` `;the constant MSGLEN equals decimal 25`
TIMES	Repeat instruction multiple times	`ZEROBUF TIMES 64 DB 0` `;initialize 64-byte buffer to all zeros`

Macro Definitions (1 of 2)

- A macro definition is similar to a subroutine in several ways
  - A subroutine is a section of a program that is written once, and can be used multiple times by calling the subroutine from any point in the program
  - When a program is compiled or assembled, the subroutine is loaded only once
  - A call to the subroutine transfers control to the subroutine and a return instruction in the subroutine returns control to the point of the call
- Similarly, a macro definition is a section of code that the programmer writes once, and then can use many times
  - The main difference is that when the assembler encounters a macro call, it replaces the macro call with the macro itself
  - This process is call *macro expansion*
- Macros are handled by the assembler at assembly time
- Macros provide the same advantage as subroutines in terms of modular programming, but without the runtime overhead of a subroutine call and return
  - The tradeoff is that the macro approach uses more space in the object code

Macro Definitions (2 of 2)



Directives

- A directive is a command embedded in the assembly source code that is recognized and acted upon by the assembler
- NASM includes the following directives:
- **BITS**
  - Specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, 32-bit mode, or 64-bit mode
- **DEFAULT**
  - Can change some assembler defaults, such as whether to use relative or absolute addressing
- **SECTION or SEGMENT**
  - Changes that section of the output file the source code will be assembled into
- **EXTERN**
  - Used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one
- **GLOBAL**
  - Is the other end of EXTERN: if one module declares a symbol as EXTERN and refers to it, then in order to prevent linker errors, some other module must actually define the symbol and declare it as GLOBAL
- **COMMON**
  - Used to declare common variables
- **CPU**
  - Restricts assembly to those instructions that are available on the specified CPU
- **FLOAT**
  - Allows the programmer to change some of the default settings to options other than those used in IEEE 754
- **[WARNING]**
  - Used to enable or disable classes of warnings

System Calls

- The assembler makes use of the x86 INT instruction to make system calls
- There are six registers that store the arguments of the system call used
  - EBX
  - ECX
  - EDX
  - ESI
  - EDI
  - EDP
- These registers take the consecutive arguments, starting with the EBX register
- If there are more than six arguments, then the memory location of the first argument is stored in the EBX register

Assembly Programs for Greatest Common Divisor

```
gcd:    mov     ebx,eax
        mov     eax,edx
        test    ebx,ebx
        jne     L1
        test    edx,edx
        jne     L1
        mov     eax,1
        ret
 L1:    test    eax,eax
        jne     L2
        mov     eax,ebx
        ret
 L2:    test    ebx,ebx
        je      L5
 L3;    cmp     ebx,eax
        je      L5
        jae     L4
        sub     eax,ebx
        jmp     L3
 L4:    sub     ebx,eax
        jmp     L3
 L5:    ret
```

(a) Compiled program

```
gcd:    neg     eax
        je      L3
 L1:    neg     eax
        xchg    eax,edx
 L2:    sub     eax,edx
        jg      L2
        jne     L1
 L3:    add     eax,edx
        jne     L4
        inc     eax
 L4:    ret
```

(b) Written directly in assembly language

C Program for Generating Prime Numbers

```
unsigned guess;             /* current guess for prime */
unsigned factor ;           /* possible factor of guess */
unsigned limit ;            /* find primes up to this value */

printf (”Find primes up to : ”);
scanf(”%u”, &limit);
printf (”2\n”);             /* treat first two primes as */
printf (”3\n”);             /* special case */
guess = 5;                  /* initial guess */
while ( guess <= limit ) {     /* look for a factor of guess */
   factor = 3;
   while ( factor * factor < guess && guess % factor != 0 )
   factor += 2;
   if ( guess % factor != 0 )
        printf (”%d\n”, guess);
   guess += 2;              /* only look at odd numbers */
}
```

Assembly Program for Generating Prime Numbers

```
%include "asm_io.inc"
segment .data
Message db "Find primes up to: ", 0

segment .bss
Limit resd 1                            ; find primes up to this limit
Guess resd 1                            ; the current guess for prime

segment .text
       global _asm_main
_asm_main:
       enter 0,0                        ; setup routine
       pusha

       mov eax, Message
       call print_string
       call read_int                    ; scanf("%u", & limit );
       mov [Limit], eax
       mov eax, 2                       ; printf("2\n");
       call print_int
       call print_nl
       mov eax, 3                       ; printf("3\n");
       call print_int
       call print_nl

       mov dword [Guess], 5             ; Guess = 5;
while_limit:                            ; while ( Guess <= Limit )
       mov eax,[Guess]
       cmp eax, [Limit]
       jnbe end_while_limit             ; use jnbe since numbers are unsigned

       mov ebx, 3                       ; ebx is factor = 3;
while_factor:
       mov eax,ebx
       mul eax                          ; edx:eax = eax*eax
       jo end_while_factor              ; if answer won't fit in eax alone
       cmp eax, [Guess]
       jnb end_while_factor             ; if !(factor*factor < guess)
       mov eax,[Guess]
       mov edx,0
       div ebx                          ; edx = edx:eax % ebx
       cmp edx, 0
       je end_while_factor              ; if !(guess % factor != 0)

       add ebx,2 ; factor += 2;
       jmp while_factor
end while factor:
       je end_if                        ; if !(guess % factor != 0)
       mov eax,[Guess]                  ; printf("%u\n")
       call print int
       call print_nl
end_if:
       add dword [Guess], 2             ; guess += 2
       jmp while_limit
end while limit:

       popa
       mov eax, 0                       ; return back to C
       leave
       ret
```

x86 String Instructions

Operation Name	Description
MOVSB	Moves the string byte addressed by the ESI register to the location addressed by the EDI register.
CMPSB	Subtracts the destination string byte from the source string element and updates the status flags in the EFLAGS register according to the results.
SCASB	Subtracts the destination string byte from the contents of the AL register and updates the status flags according to the results.
LODSB	Loads the source string byte identified by the ESI register into the EAX register.
STOSSB	Stores the source string byte from the AL register into the memory location identified with the EDI register.
REP	Repeat while the ECX register is not zero.
REPE/REPZ	Repeat while the ECX register is not zero and the ZF flag is set.
REPNE/REPNZ	Repeat while the ECX register is not zero and the ZF flag is clear.

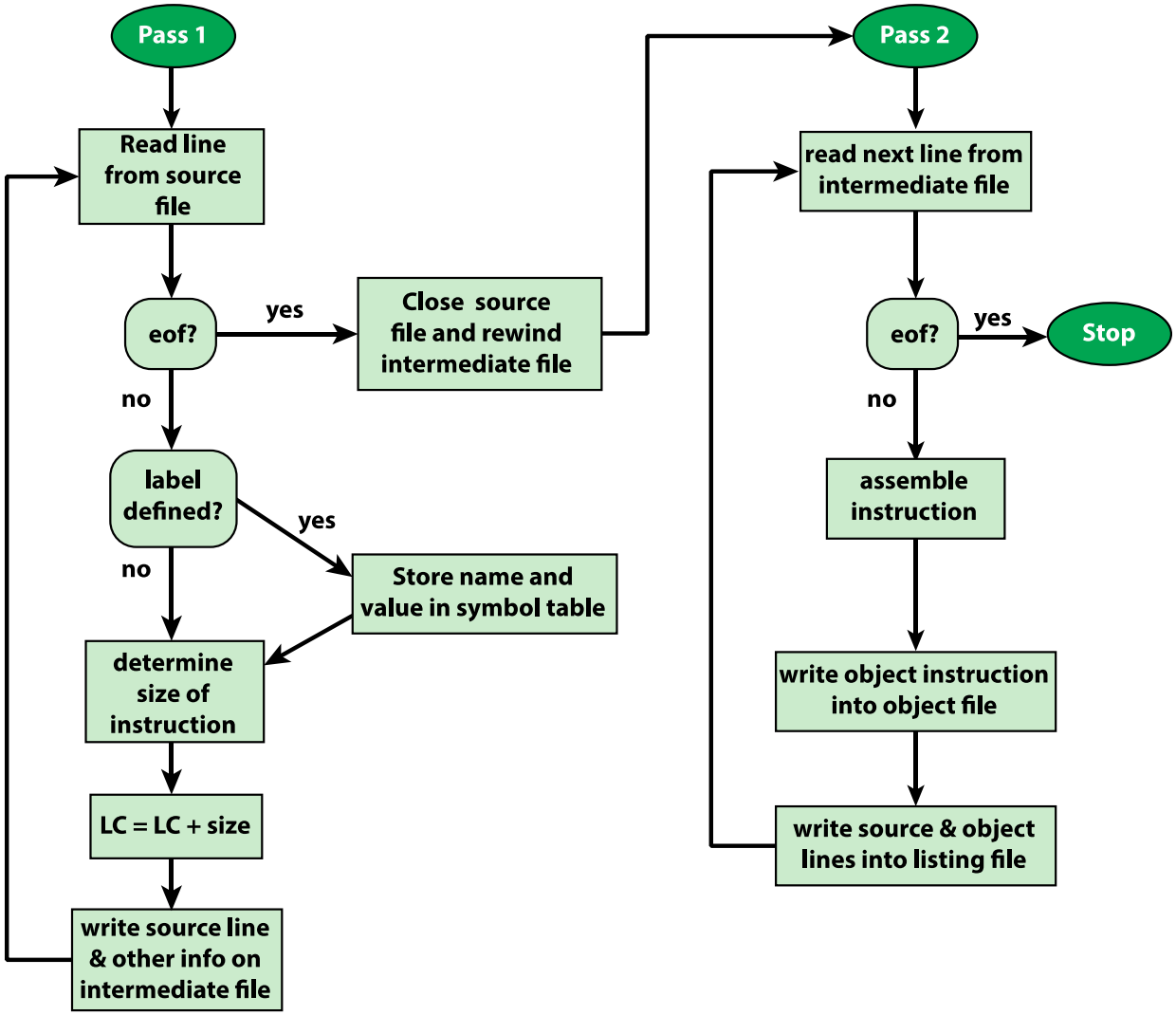
Assembly Program for Moving a String

```
section .text
    global  main                ;must be declared for using gcc
main:                           ;tell linker entry point
    mov    ecx, len
    mov    esi, s1
    mov    edi, s2
    cld
    rep    movsb
    mov    edx,20               ;message length
    mov    ecx,s2               ;message to write
    mov    ebx,1                ;file descriptor (stdout)
    mov    eax,4                ;system call number (sys_write)
    int    0x80                 ;call kernel
    mov    eax,1                ;system call number (sys_exit)
    int    0x80                 ;call kernel
section .data
s1 db 'Hello, world!',0         ;string 1
len  equ    $-s1
section .bss
s2   resb 20                    ;destination
```

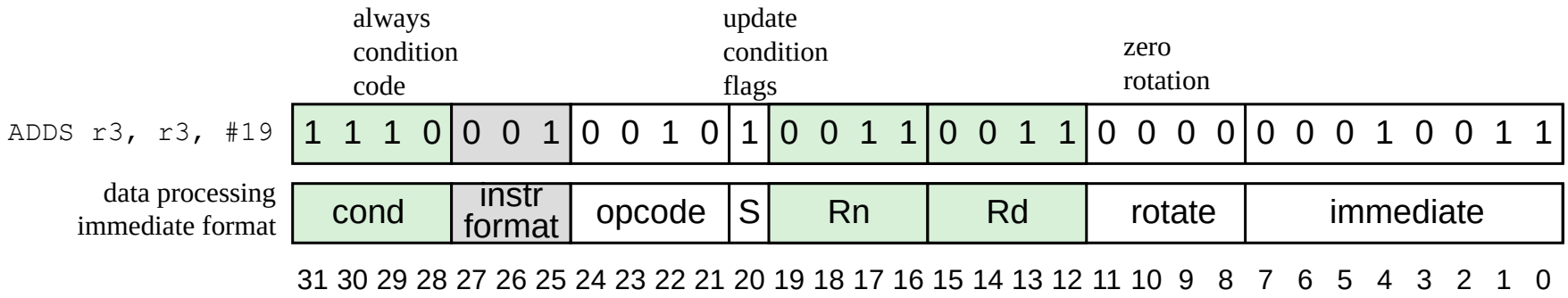
TYPES OF ASSEMBLERS

- An assembler is a software that translates assembly language into machine language
- Although all assemblers perform the same tasks, their implementations vary
- Some of the common terms that describe types of assemblers:
  - Cross-assembler
  - Resident assembler
  - Macroassembler
  - Microassembler
  - Meta-assembler
  - One-pass assembler
  - Two-pass assembler

Flowchart of Two-Pass Assembler



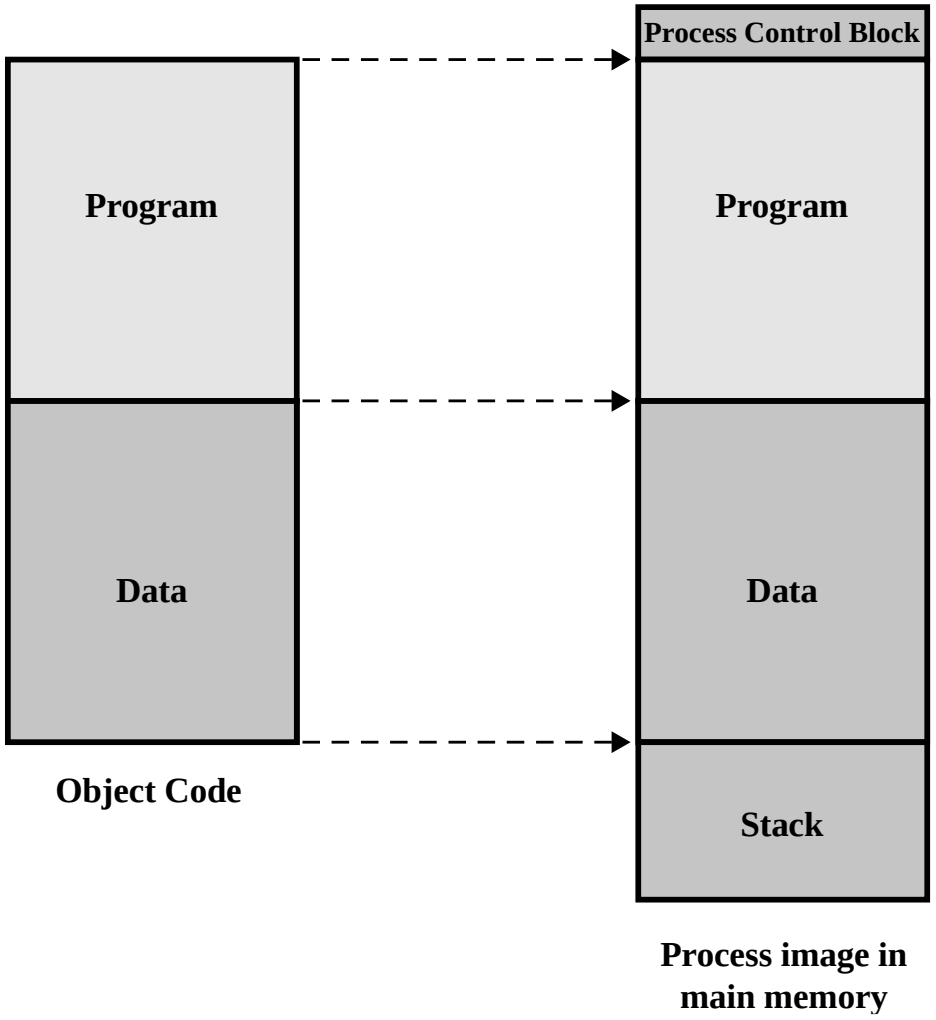
Translating an ARM Assembly Instruction into a Binary Machine Instruction



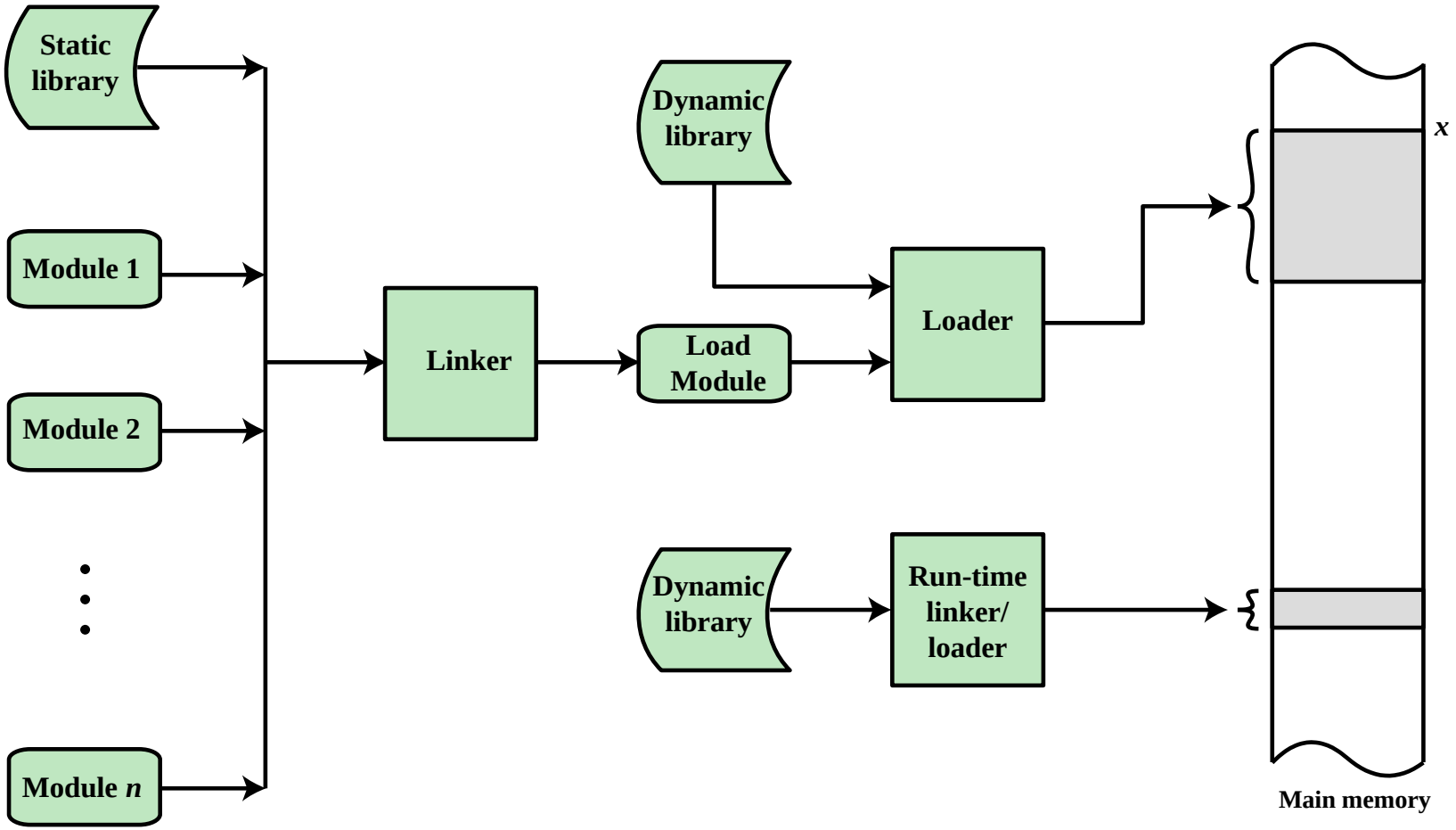
One-Pass Assembler

- It is possible to implement as assembler that makes only a single pass through the source code
- The main difficulty in trying to assemble a program in one pass involves forward references to labels
- Instruction operands may be symbols that have not yet been defined in the source program
  - Therefore, the assembler does not know with relative address to insert in the translated instruction
- When the assembler encounters an instruction operand that is a symbol that is not yet defined, the assembler does the following:
  - It leaves the instruction operand field empty in the assembled binary instruction
  - The symbol used as an operand is entered in the symbol table and the table entry is flagged to indicate that the symbol is undefined
  - The address of the operand field in the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry

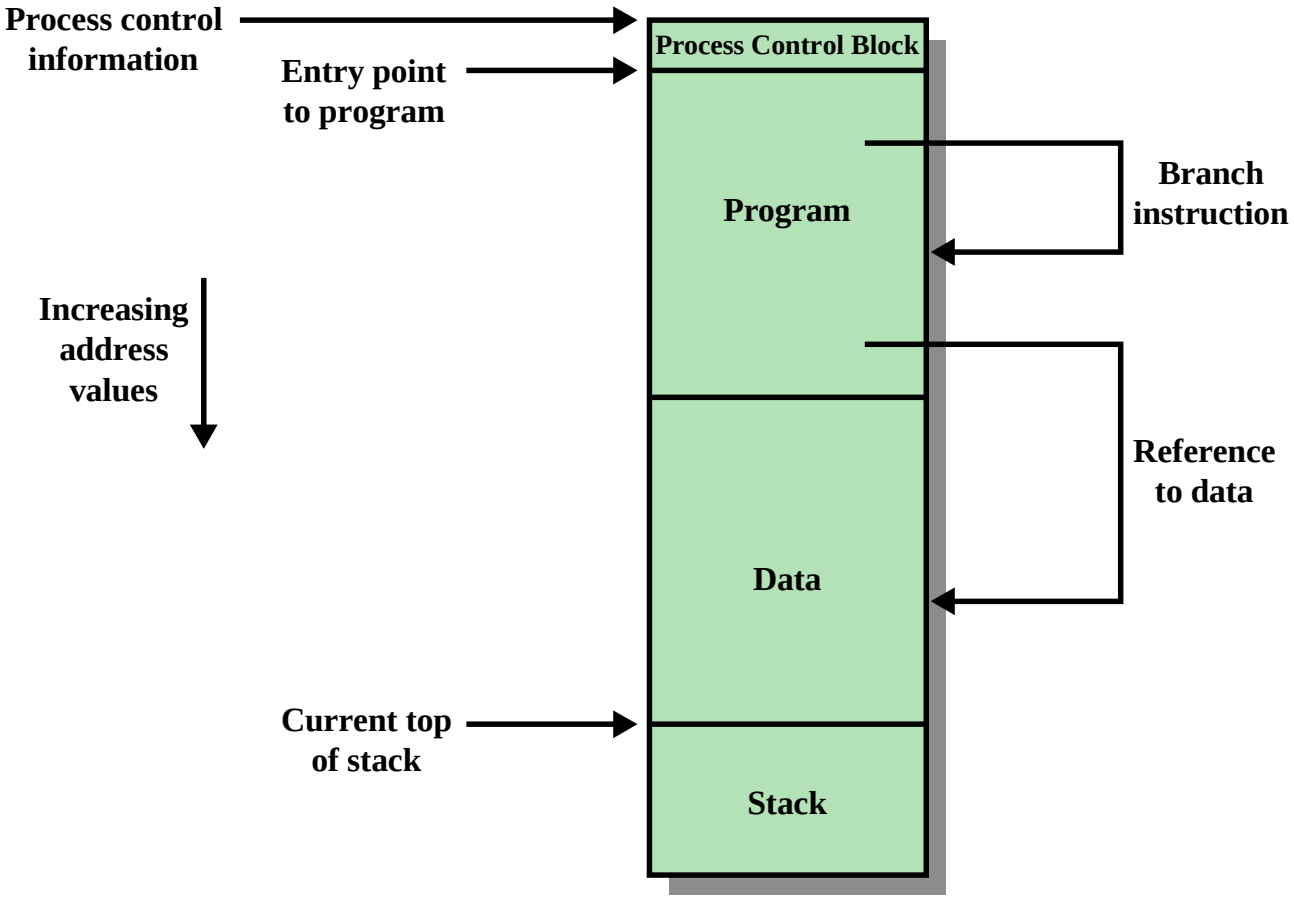
The Loading Function



A Linking and Loading Scenario



Addressing Requirements for a Process



Address Binding

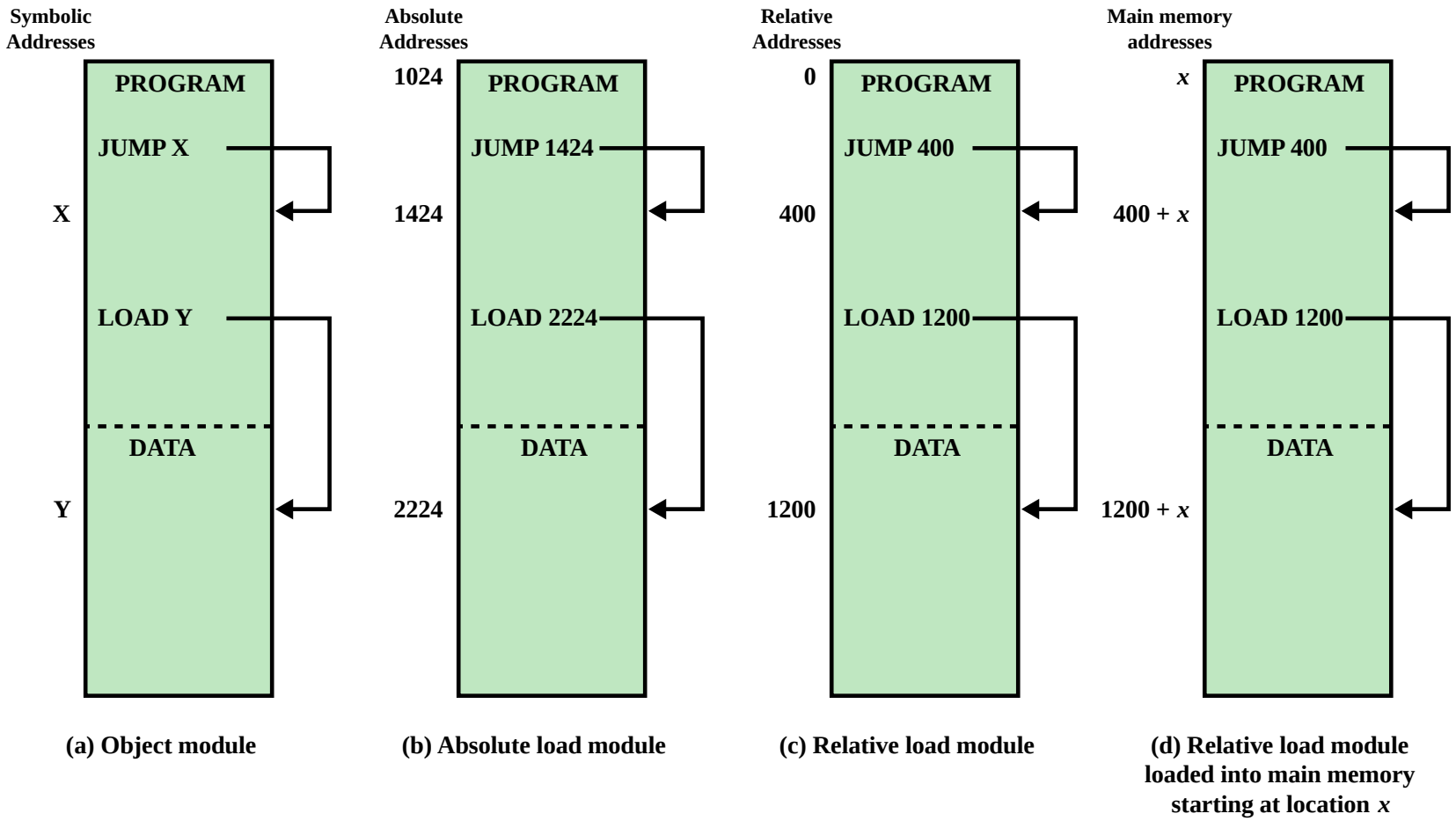
(a) Loader

Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together, and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

Absolute and Relocatable Load Modules



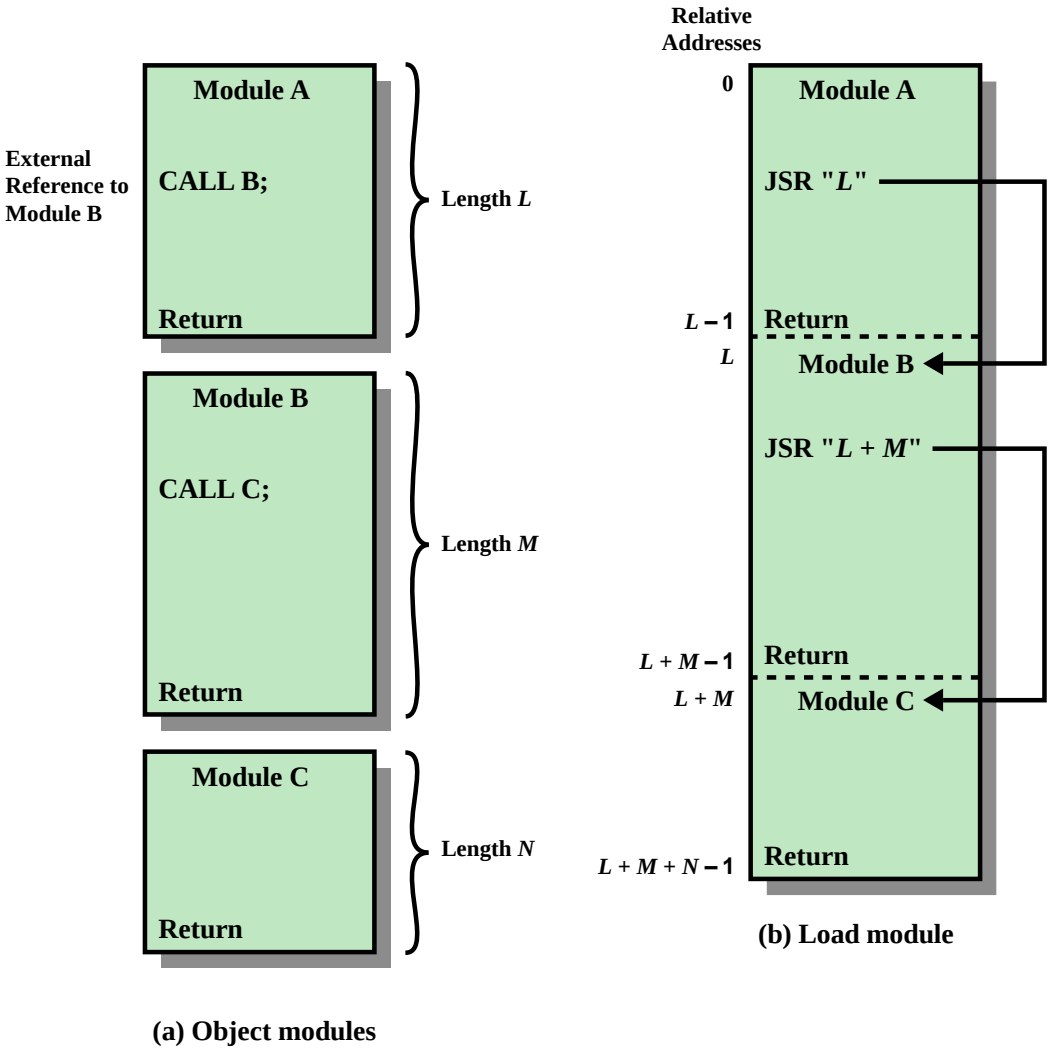
(a) Object module

(b) Absolute load module

(c) Relative load module

(d) Relative load module loaded into main memory starting at location *x*

The Linking Function



Load-Time Dynamic Linking

- Dynamic Linking is used to refer to the practice of deferring the linkage of some external modules until after the load module has been created
- For load-time dynamic linking the steps occur following:
  - The load module to be loaded is read into memory
  - Any reference to an external module causes the loader to find the target module, load it, and alter the reference to a relative address in memory from the beginning of the application module
- Advantages to approach over what might be called static linking
  - It becomes easier to incorporate changed or upgraded versions of the target module
  - Having target code in a dynamic link file paves the way for automatic code sharing
  - It becomes easier for independent software developers to extend the functionality of a widely-used operating system such as Linux

Run-Time Dynamic Linking

- With *run-time dynamic linking* some of the linking is postponed until execution time
  - External references to target modules remain in the loaded program
  - When a call is made to the absent module, the operating system locates the module, loads it, and links it to the calling module
  - Such modules are typically shareable
  - In the Windows environment these are called dynamic-link libraries (DLLs)
  - If one process is already making use of a dynamically linked shared module, then that module is in main memory and a new process can simply link to the already-loaded module
- The use of DLLs can lead to a problem commonly referred to as *DLL hell*
  - DLL hell occurs if two or more processes are sharing a DLL module, but expect different versions of the module