Computer Organization and Architecture Designing for Performance

11th Edition



Chapter 15

Assembly Language and Related Topics

Copyright © 2019, 2016, 2013 Pearson Education, Inc. All Rights Reserved

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.

Table 15.1

Key Tern For This an 22 (Table can be found on page 508 in the textbook.)

universidad de Costa

Figure 15.1 **Source 15.1 Figure 15.1 Fig**

	Address		Conte	ents T		r II	Address	Contents		
	Opcode Operand				ካ ተካ	h h				
	101	0010	0010	1100	1001		101	22C9		
	102	0001	0010	1100	1010		102	12CA		
\odot	103	0001	0010	1100	1011		103	12CB		
	104	0011	0010	1100	1100		104	32CC		
									-	
	201	0000	0000	0000	0010		201	0002		
	202	0000	0000	0000	0011		202	0003		
0	203	0000	0000	0000	0100		203	0004		
	204	0000	0000	0000	0000		204	0000		
						-	-			
		(a) Binary	y program				(b) Hexadec:	imal program	n	
	Address	Inst	ruction		Ι	lab el	Operation	Operand		
	101	LDA	201		FC	DRMUL	LDA			
	102	ADD	202				ADD	J		
	103	ADD	203				ADD	ĸ		
	104	STA	204	ΡΨ			STA	N		
	201	DAT	0002			1	DATA	2		
	202	DAT	0003			J	DATA	3		
	203	DAT	0004			K	DATA	4		
	204	DAT	0000			N	DATA	0		
				and the second se						
) (\odot						
	(c) S	ymbolic p	orogram	0		l) Assen	nbly 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

Universidad de Costa Rica

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

2022

- Application code can use intrinsic functions or vector classes instead of assembly
- Compilers have been improved a lot in recent years

Universidad de Costa Rica

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

Universidad de Costa Rica

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



Statements (1 of 3)

- 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 (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

Figure 15.3 O O O O O O O Figure 15.3 Figu



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

Table 15.2 Image: Comparison of the second seco

	(a) Letters	I I I RESX and D	Directives		
	U	nit	Letter		
	by	byte			
	word (2	2 bytes)	W		
	double wo	rd (4 bytes)	D		
	quad word	d (8 bytes)	Q		
	ten l	oytes	Т		
	-9	(b) Directives		· Da	
Name	Description		Example		
DB, DW, DD, DQ, DT	Initialize locations	L6 DD 1A92H ;doubleword a	nt L6 initialize	ed to 1A92H	
RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	BUFFER RESB 6 ;reserve 64 b	64 bytes starting a	t BUFFER	
INCBIN	Include binary file in output	<pre>INCBIN "file.dat" ; include this file</pre>			
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 ;initialize 6	5 64 DB 0 54-byte buffer t	o 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)

In NASM and many other assemblers, a distinction is made between a single-line macro and a multi-line macro 2022 In NASM, single-line macros Multiline macros are are defined using the %DEFINE directive defined using the mnemonic %MACRO

Universidad de Costa Rica

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 64bit 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
- 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

		(a) Com	piled program	(b) W:	ritten direc	tly in assemb ly l	anguage
	L5:	ret					
		jmp	L3				
	L4:	sub	ebx,eax				
		jmp	L3				
		sub	eax,ebx				
		jae	L4				
	,	ie.	L5				
\odot -	L3:	cmp	ebx.eax				
	64.	iest ie	LS				
	1.2.	ret test	ahy ahy				
0-		mov	eax,ebx	L4:	ret		
		jne	L2	T 0.	inc	eax	
	Ll:	test	eax,eax		jne	L4	
		ret		L3:	add	eax,edx	
		mov	eax,l		jne	Ll	
\bigcirc		jne	Ll		jg	L2	
		test	edx,edx	L2:	sub	eax,edx	
		jne	Ll		xchg	eax,edx	
\bigcirc		test	ebx,ebx	L1:	neg	eax	
	2	mov	eax,edx		je	L3	
	acd:	mov	ebx,eax	 acd:	neg	eax	

Figure 15.5 \bigcirc **C Program for Generating Prime Numbers**

```
unsigned guess;
unsigned limit ;
```

/* current guess for prime */ unsigned factor ; /* possible factor of guess */ /* 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 */
                /* initial guess */
quess = 5;
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 */
```

Figure 15.6 Assembly **Program for Generating Prime Numbers**

%include "asm io.inc" segment .data Message db "Find primes up to: ", O 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 jube 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 ; edx = edx:eax % ebx div ebx cmp edx, 0 ; if !(guess % factor != 0) je end while factor add ebx_2 ; factor += 2; jmp while factor end while factor: ; if !(guess % factor != 0) je end if mov eax, [Guess] ; printf("%u\n") call print int call print nl add dword [Guess], 2 ; guess += 2 jmp while limit end while limit: popa mov eax, 0 : return back to C leave

end if:

ret

Table 15.3 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.

Universidad de Costa Rica

section .text	
global main	; must be declared for using gcc
main:	;tell linker entry point
mov ecx.len	
mov esi, sl	
mov, edi, s2	
cld	
rep movsb	
mov edx,20_	;message length
mov ecx,s2	;message to write
mov ebx,l	;file_descriptor_(stdout)
mov eax,4	;system.call_number(sys_write)
int 0×80_	call kernel
mov eax.1	;system.call_number(sys_exit)
int 0×80	; call kernel
section data	
st db 'Hello, world!',0	;string 1
len equ \$-sl	
section .bss	
s2 resb 20	;destination
U	

Jniversidad de Costa Rica

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

Jniversidad de Costa Rica



Figure 15.9 **O O O O O** 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

Universidad de Costa Rica







Table 15.4Address Binding

(a) Loader

 \odot

	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 hese to absolute addresses at the time of program loading.					
	Run time	e loaded program retains relative addresses. These are converted dynami- lly to absolute addresses by processor hardware.					
		(b) Linker	-0				
	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.					
can be 1d on 529 in atbook)	Load module creation	All object modules have been assembled using relative addresses. These mod- ules 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.	C				
	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.	<u> </u>				

(Table can be found on page 529 in the textbook

 \odot



Universidad de Costa Ric



po. Arrovd

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

Universidad de Costa Rica

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

Universidad de Costa Rica

Summary

Chapter 15

- Assembly language concepts
- Motivation for assembly language programming
- Assembly language elements
 - Statements
 - Pseudo-instructions
 - Macro definitions
 - Directives
 - System calls

Types of assemblers

Assembly

Language and

Related Topics

- Assemblers
 - Two-pass assembler
 - One-pass assembler

.

- Loading and linking
 - Relocation

Loading

Linking