CMSC 131 Introduction to Computer Organization and Machine-Level Programming
Handout 1: Introduction to Assembly Programming

Learning Outcomes

At the end of this session, the student should be able to:

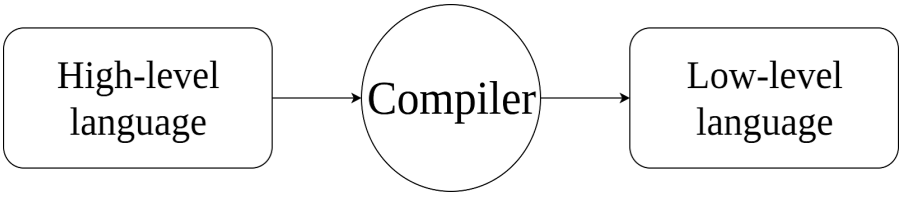
1. understand what Assembly language is;
2. examine the x86-64 architecture; and
3. create a simple program using Assembly language.

Content

Assembly Language

Assembly language is a low-level language that provides the basic instructional interface to the computer processor.

Programs written in a high-level language are translated into assembly language in order for the processor to execute the program. The high-level language is an abstraction between the language and the actual processor instructions.



Assembly language gives you direct control of the system's resources. This involves setting the processor registers, accessing memory locations, and interfacing with other hardware elements.

The x86-64 Architecture

Data Storage Sizes

The x86-64 architecture supports a specific set of data storage size elements:

Storage Size	Size (bits)	Size (bytes)
Byte	8	1
Word	16	2
Double-word	32	4
Quadword	64	8
Double quadword	128	16

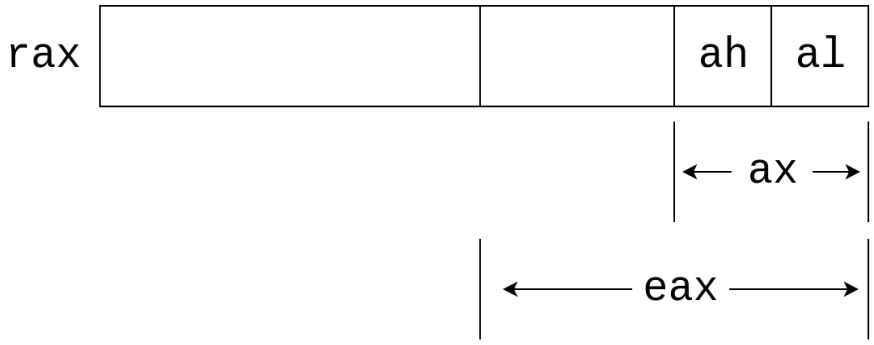
These storage sizes have a direct correlation to variable declarations in high level languages (e.g., C, C++, Java, etc.). For instance, a character in C/C++ is 8 bits long which means it can be assigned to a byte-sized storage.

CPU Registers

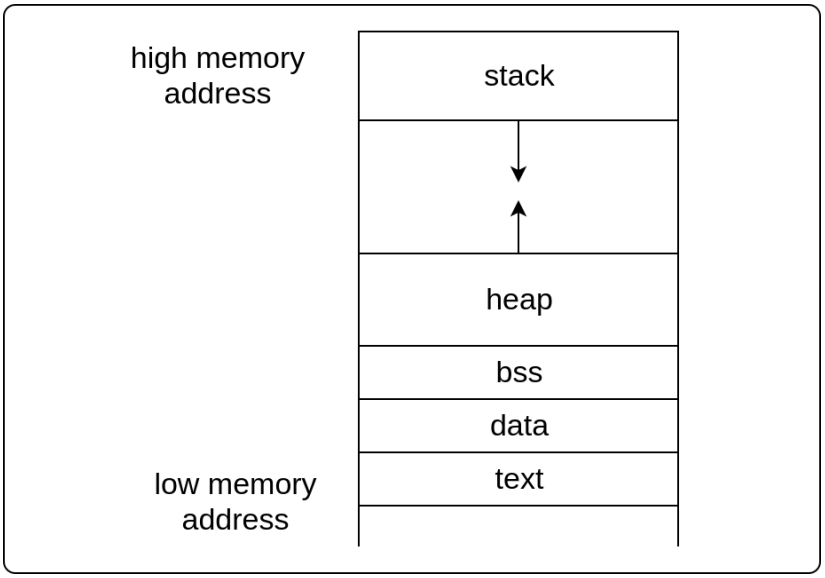
A register, is a temporary storage built into the CPU itself. Computations are typically performed by the CPU using registers. There are sixteen, 64-bit **General Purpose Registers** (GPRs):

64-bit Register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	e12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

When using data element sizes less than 64-bits (i.e., 32-bit, 16-bit, or 8-bit), the lower portion of a GPR register can be accessed by using a different register name as shown on the table. For example, when accessing the portions of the 64-bit `rax` register, the portions that can be accessed are shown in this layout:



Memory Layout
The general memory layout for a program:



The **text** (or code) section is where the machine language (i.e. the 1's and 0's that represent the code) is stored. The **data** section is where the initialized data is stored. The **bss** section is where uninitialized declared variables are stored. The **heap** is where dynamically allocated data will be stored (if requested). The **stack** starts in high memory and grows downward.

Program Format

A properly formatted Assembly source file consists of the **data** section, **bss** section, and **text** section. The formatting requirements discussed here are specific to the **NASM** assembler.

Data Section
All initialized variables and constants are placed in this section.

The syntax to define a variable is:

```
<variableName> <directive> <initialValue>
```

Variable names must start with a letter, followed by letters or numbers, including some special characters. **Assembler directives** are instructions to the assembler that direct the assembler to do something. The following directives are used for initialized data declarations:

Directive	Size	Example
db	8-bit	`grade db "A"`
dw	16-bit	`cnt dw 500`
dd	32-bit	`fee dd 70000`
dq	64-bit	`students dq 110900`

Constants are defined with `equ`. The syntax to define a constant is:

```
<name> equ <value>
```

The value of a constant cannot be changed during program execution. A constant is not assigned a memory location. It is also not assigned with a specific type/size (byte, word, double-word, etc.).

BSS Section

All uninitialized variables are declared in this section. The syntax is:

```
<variableName> <directive> <count>
```

The following directives are used for uninitialized data declarations:

Directive	Size	Example
resb	8-bit	`list resb 5`
resw	16-bit	`num resw 3`
resd	32-bit	`arr resd 100`
resq	64-bit	`cnt resq 10`

Text Section

The instructions are specified in this section - one instruction per line and each must be a valid instruction with the appropriate required operands.

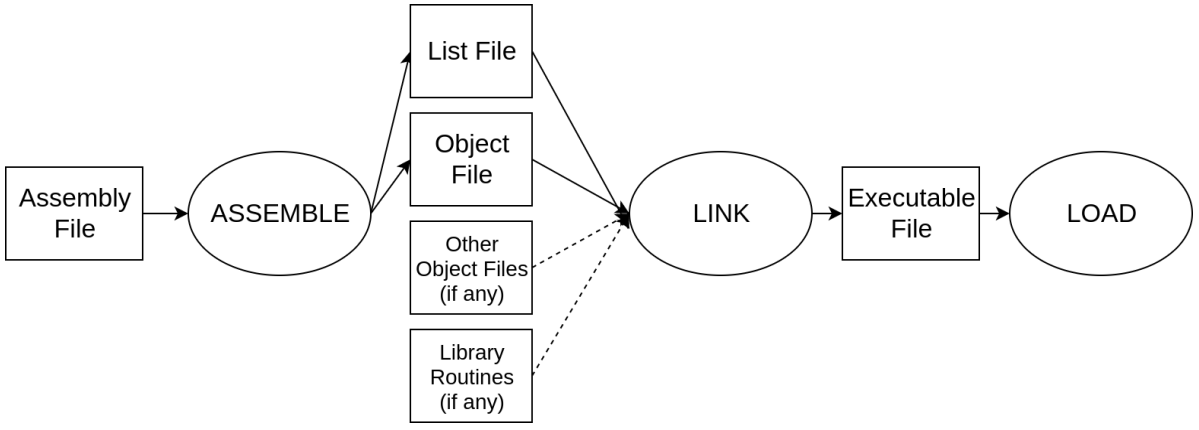
Below is an example of a simple Assembly program:

```
 1  global _start
 2
 3  section .data
 4     SYS_WRITE   equ 1
 5     SYS_EXIT    equ 60
 6     message db "Hello World!", 10
 7
 8  section .text
 9  _start:
10     mov rax, SYS_WRITE      ; system call for write
11     mov rdi, 1              ; file handle 1 is stdout
12     mov rsi, message        ; address of string to output
13     mov rdx, 13             ; length of string
14     syscall                 ; invoke os to do the write
15
16     mov rax, SYS_EXIT       ; system call for exit
17     xor rdi, rdi            ; exit code 0
18     syscall                 ; invoke os to exit
```

The commands in lines 1 and 9 define the initial entry point of the program. The semicolon (;) is used to note single-line program comments.

Tool Chain

Tool chain refers to the set of programming tools used to create a program. It consists of the **assembler**, **linker**, **loader**, and **debugger**.



Assembler

The assembler is a program that will read an assembly language input file and convert the code into an object file. The **NASM** assembler command is:

```
nasm -felf64 <asmFilename>.asm
```

Linker

The linker will combine one or more object files into a single executable file. Additionally, any routines from user or system libraries are included as necessary. The **GNU linker** command is:

```
ld [-o <execFilename>] <ObjectFilename>.o
```

In the given Assembly program example, `global` is a NASM directive for exporting symbols in the code to where it points in the object code generated. The `_start` symbol is marked as global so its name is added in the object code. The linker can read that symbol in the object code and its value so it knows where to start in the program.

Loader

The loader is a part of the operating system that will load the program from secondary storage into primary storage (i.e., main memory) ready for execution.

```
./<execFilename>
```

Debugger

The debugger is used to control execution of a program. This allows for testing and debugging activities to be performed.

References

Jorgensen, Ed. 2019. x86-64 Assembly Language Programming with Ubuntu. Version 1.1.40.
NASM Tutorial. https://cs.lmu.edu/~ray/notes/nasmtutorial/.