8086 Programmimg with X86-64 architecture

1. Objective:

1. The goal is to provide a comprehensive introduction to programming in assembly language.

2. Writing non-trivial programs to perform specific low-level actions including *arithmetic operations, function calls, using stack-dynamic local variables,* and operating system interaction for activities such as input/output.

Tools Required:

1. **YASM Assembler**

The YASM assembler is an open source assembler commonly available on Linux-based Systems.

Use following command to install YASM in Ubuntu 18.04 LTS

sudo apt-get install yasm

To assemble YASM file in 64-bit x86 source example1.asm into ELF file example1.o

yasm -g dwarf2 -f elf64 example1.asm -l example1.lst

-***elf*** means Executable & Linkable file format which is supported by YASM.

-***-g dwarf2*** option is used to inform the assembler to include debugging information in the final object file. Basically dwarf is a debugging data format.

2. **DDD Debugger**

The DDD debugger is an open source debugger capable of supporting assembly language.

To install DDD in Ubuntu, use command

sudo apt-get install ddd

2. Architecture Overview

Data Storage Sizes

The x86-64 architecture supports a specific set of data storage size elements, all based on powers of two. The supported storage sizes are as follows:

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

- Mapping of variable declaration corresponding to Storage sizes
-

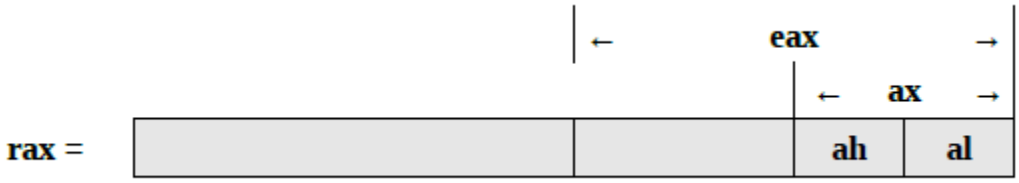
C/C++ Declaration	Storage	Size (bits)	Size (bytes)
char	Byte	8-bits	1 byte
short	Word	16-bits	2 bytes
int	Double-word	32-bits	4 bytes
unsigned int	Double-word	32-bits	4 bytes
long[5]	Quadword	64-bits	8 bytes
long long	Quadword	64-bits	8 bytes
char *	Quadword	64-bits	8 bytes
int *	Quadword	64-bits	8 bytes
float	Double-word	32-bits	4 bytes
double	Quadword	64-bits	8 bytes

- **General Purpose Registers (GPRs)**

  There are sixteen, 64-bit General Purpose Registers (GPRs). The GPRs are described in the following table. A GPR register can be accessed with all 64-bits or some portion or subset accessed.

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	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

For example, when accessing the lower portions of the 64-bit rax register, the layout is as follows:



Flag Register

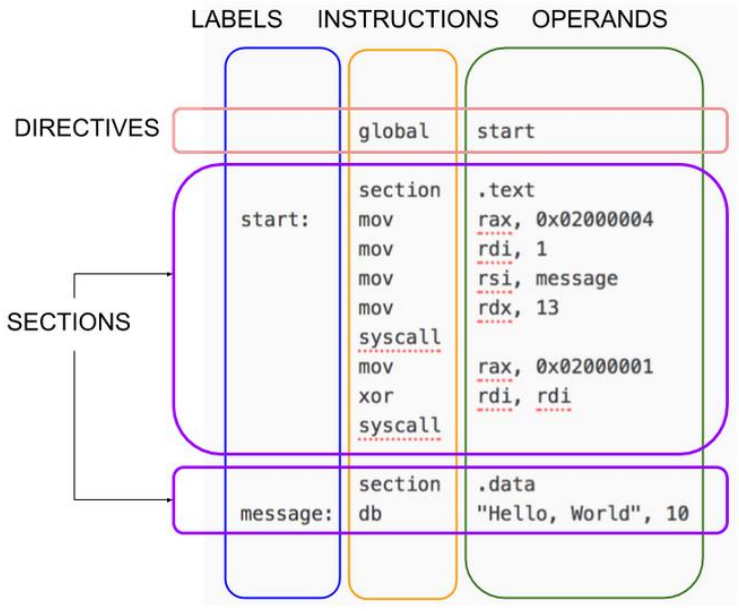
The following table shows some of the status bits in the flag register.

Name	Symbol	Bit	Use
Carry	CF	0	Used to indicate if the previous operation resulted in a carry.
Parity	PF	2	Used to indicate if the last byte has an even number of 1's (i.e., even parity).
Adjust	AF	4	Used to support Binary Coded Decimal operations.
Zero	ZF	6	Used to indicate if the previous operation resulted in a zero result.
Sign	SF	7	Used to indicate if the result of the previous operation resulted in a 1 in the most significant bit (indicating negative in the context of signed data).
Direction	DF	10	Used to specify the direction (increment or decrement) for some string operations.
Overflow	OF	11	Used to indicate if the previous operation resulted in an overflow.

3. Program Format

A properly formatted assembly source file consists of several main parts;

• Data section where initialized data is declared and defined.
• BSS section where uninitialized data is declared.
• Text section where code is placed.



3.1 Defining Constants

Constants are defined with equ. The general format is:
<name> equ <value>
The value of a constant cannot be changed during program execution. e.g.
SIZE equ 10000

3.2 Data Section

The initialized data must be declared in the "section .data" section. The general format is:

<variableName> <dataType> <initialValue>

For e.g. bVar db 10 ; byte variable

Supported Data Types:

Declaration	
db	8-bit variable(s)
dw	16-bit variable(s)
dd	32-bit variable(s)
dq	64-bit variable(s)
ddq	128-bit variable(s) → integer
dt	128-bit variable(s) → float

Some simple examples include:

```
bVar    db     10              ; byte variable
cVar    db    "H"              ; single character
strng   db   "Hello World"     ; string
wVar    dw     5000            ; word variable
dVar    dd     50000           ; 32-bit variable
arr     dd     100, 200, 300   ; 3 element array
flt1    dd    3.14159          ; 32-bit float
qVar    dq     1000000000      ; 64-bit variable
```

3.3 BSS Section

Uninitialized data is declared in the "section .bss" section.

Declaration	
resb	8-bit variable(s)
resw	16-bit variable(s)
resd	32-bit variable(s)
resq	64-bit variable(s)
resdq	128-bit variable(s)

Some simple examples include:

```
bArr    resb    10    ; 10 element byte array
wArr    resw    50    ; 50 element word array
```

3.4 Text Section

The code is placed in the "section .text" section. The text section will include some headers or labels that define the initial program entry point. For example,

```
global _start
_start:
```

Sample Program: A sample program is provided you in separate sheets. Please check the different variable declarations and the layout of simple arithmetic program.

4. Assemble Commands

a) yasm assembler command for reading the assembly language source file

yasm -g dwarf2 -f elf64 example1.asm -l example1.lst

-g dwarf2: It is used to inform the assembler to include debugging information in the final object file.

-f elf64 : It informs the assembler to create the object file in the ELF64 27 format which is appropriate for 64-bit, Linux based systems.

b) **Linker Command**

ld -g -o example1 example1.o

The -g option is used to inform the linker to include debugging information in the final executable file

c) **Linking Multiple Files**

For e.g. ***ld -g -o example main.o funcs.o***

5. DDD Debugger

The open source GNU Data Display Debugger (DDD35) is a visual front-end to the GNU Debugger (GDB36) and is widely available.

5.1 Starting DDD

ddd example1 where example1 is the file for debugging

The debugging is basically used for following purposes:

1. Monitor the flow of execution of program
2. Examining various memory and registers contents during run time
3. Examining possible line numbers where you can find the error etc.

5.2 Setting Breakpoints

A breakpoint is usually execution pause location where the user can pause the flow of control at particular line number.

The breakpoint can be done one of three ways:

· Right click on the line number and select: *Set Breakpoint*
· In the GDB Command Console, at the (gdb) prompt, type: **break last**
· In the GDB Command Console, at the (gdb) prompt, type: **break 95**

5.3 Functionalities of DDD

1. Run the program
2. Step mode
3. Display register & Memory Contents
4. Look for next or previous instruction
5. Kill
6. Finish execution etc.

5.4 DDD/GDB Commands

Suppose the variables declared are as follows:

bnum1 db 5
wnum2 dw -2000
dnum3 dd 100000

So the commands for displaying contents will be : **x/<n><f><u> &<variable>**

x/db &bnum1
x/dh &wnum2

Command	Description
quit | q	Quit the debugger.
break <label/addr> **| b <label/addr>**	Set a break point (stop point) at <label> or <address>.
run <args> | r <args>	Execute the program (to the first breakpoint).
continue | c	Continue execution (to the next breakpoint).
continue <n> | c <n>	Continue execution (to the next breakpoint), skipping ***n***-1 crossing of the breakpoint. This is can be used to quickly get to the ***n***th iteration of a loop.
step | s	Step into next instruction (i.e., steps into function/procedure calls).
next | n	Next instruction (steps through function/procedure calls).
F3	Re-start program (and stop at first breakpoint).
where	Current activation (call depth).
x/<n><f><u> $rsp	Examine contents of the stack.
x/<n><f><u> &<variable>	Examine memory location <variable> <n> number of locations to display, 1 is default. <f> format: d – decimal (signed) x – hex u – decimal (unsigned) c – character s – string f – floating-point <u> unit size: b – byte (8-bits) h – halfword (16-bits) w – word (32-bits) g – giant (64-bits)
source <filename>	Read commands from file <filename>.
set logging file <filename>	Set logging file to <filename>, default is *gdb.txt*.
set logging on	Turn logging (to a file) on.
set logging off	Turn logging (to a file) off.
set logging overwrite	When logging (to a file) is turned on, overwrite previous log file (if any).

6. Addressing Modes

The basic addressing modes are:

- Register
- Immediate
- Memory

6.1 Register Mode Addressing

Register mode addressing means that the operand is a CPU register (**eax**, **ebx**, etc.). For example:

```
Mov    eax,    ebx
```

Both **eax** and **ebx** are in register mode addressing.

6.2 Immediate Mode Addressing

Immediate mode addressing means that the operand is an immediate value. For example:

```
mov    eax,    123
```

The destination operand, **eax** , is register mode addressing. The **123** is immediate mode addressing. It should be clear that the destination operand in this example cannot be immediate mode.

6.3 Memory Mode Addressing

Memory mode addressing means that the operand is a location in memory (accessed via an address). This is referred to as *indirection* or *dereferencing. For e.g.*

```
mov    rax,    qword [qNum]
```

Array Initialization

When accessing arrays, a more generalized method is required. Specifically, an address can be placed in a register and indirection performed using the register (instead of the variable name). For example, assuming the following declaration:

```
lst    dd    101, 103, 105, 107
```

The first element of the array could be accessed as follows:

```
Mov    eax,    dword [list]
```

Another way to access the first element is as follows:

```
mov    rbx,    list
mov    eax,    dword [rbx]
```

The general format of memory addressing is as follows:

[baseAddr + (indexReg * scaleValue) + displacement]

Where ***baseAddr*** is a register or a variable name. The ***indexReg*** must be a register. The ***scaleValue*** is an immediate value of 1, 2, 4, 8 (1 is legal, but not useful). The ***displacement*** must be an immediate value. The total represents a 64-bit address

For e.g. if you want to access 3rd element i.e. 105, given the initializations:

```
Mov    rbx,    list
mov    rsi,    8
```

Each of the following instructions access the third element (105 in the above list).

```
Mov    eax,    dword [list+8]
mov    eax,    dword [rbx+8]
mov    eax,    dword [rbx+rsi]
```