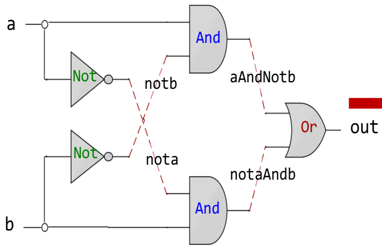
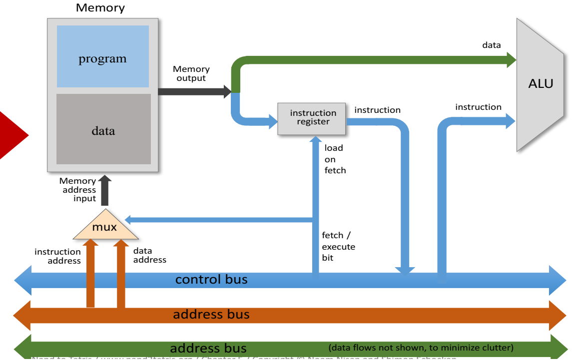
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 30

Structure of x86-64 Assembly Program

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

For resources visit my personal website:
https://www.arifbutt.me
and course bitbucket repository:
https://bitbucket.org/arifpucit/coal-repo

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Review of Tool Chain & Programming Environment
- Scuba Diving in ***`first.nasm`***
  - Assembly Language Statements
  - Assembly Instruction Format
  - Layout of x86_64 Assembly Programs
    - Section .data
    - Section .bss
    - Section .text
  - X86 `MOV` Instruction
  - X86-64 `SYSCALL` Instruction

Review of Tool Chain & Programming Env

first.nasm

```
; COAL Video Lecture: 29
;  Programmer: Arif Butt
;  first.nasm
SECTION .data
   msg db "Learning…", 0xA
   EXIT_STATUS equ 54
SECTION .bss
;nothing here
SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```

Assemble →

first.o

```
10001000
01000001
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
01001001
1000101001001000
0101011000011000
0001010010010001
10001010
01001011
```

Link →

myexe

```
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
10001000
01000001
0101011000011111
0001010011110000
10001000
1000101001001000
0001010010010001
10001010
01001011
0001010011110000
10001000
01001101
10001000
```

Load & Execute →

```
$ nasm –felf64 first.nasm –o first.o
$ ld first.o –o myexe
$./myexe
```

Learning is fun with Arif

Scuba Diving inside `first.nasm`

Assembly Language Statements

first.nasm

```
; COAL Video Lecture: 30
;  Programmer: Arif Butt
;  first.nasm
SECTION .data
   msg db "Learning…", 0xA
   EXIT_STATUS equ 54
SECTION .bss
;nothing here
SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```

There are three types of statements in assembly language programming. Typically, each statement should appear on a separate line

- **x86-64 Assembly Instructions:** These instructions are actually converted into machine code, and when executed, instruct the processor what to do. Some x86 specific assembly instructions are `mov`, `add`, `sub`, `syscall`
- **Pseudo Instruction:** These are not real x86 machine instructions but are normally used in the real instruction field. Some `nasm` specific pseudo instructions are `DB`, `DW`, `RESB`, `RESW` , `EQU`
- **Assembler Directives:** Assembly directives are the statements that direct the assembler to do something. The specialty of these statements is that they are effective only during the assembly of a program and they do not generate any machine executable code. Some `nasm` specific directives are `SECTION`, `EXTERN`, `GLOBAL`, `BITS`

Assembly Language Instruction Format

Label/var:	Mnemonic	Operand	Comment

```
; first.nasm

        SECTION   .data                              ; a comment
msg     db        "Learning…Arif", 0xA
X_STAT  equ       54

        SECTION   .bss
                                                     ; nothing here

        SECTION   .text
        global     _start
_start:
        mov        rax,1
        mov        rdi,1
        mov        rsi,msg
        mov        rdx,26
        syscall
        mov        rax,60
        mov        rdi, X_STAT
        syscall
```

Structure of Assembly Program

```
; first.nasm
```

Initialized Data Section

```
        SECTION   .data                         ; a comment
msg     db        "Learning…Arif", 0xA
X_STAT  equ       54
```

Un-Initialized Data Section

```
        SECTION   .bss
                                                ; nothing here
```

Code Section

```
        SECTION   .text
        global     _start
_start:
        mov        rax,1
        mov        rdi,1
        mov        rsi,msg
        mov        rdx,26
        syscall
        mov        rax,60
        mov        rdi, X_STAT
        syscall
```

Section .data: Initialized Data

- All initialized data like variables and constants are placed in the `.data` section
- A constant is defined with `equ` instruction:

```
<constName>   equ   <value>
```

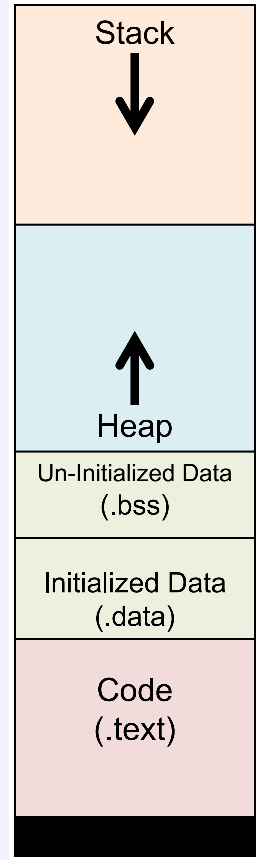
- A define directive sets aside storage in memory for variables. The general format for variable declaration is:

```
<varName>  <defineDir>  <iv> [,iv]
```

- NASM provides various directives for reserving storage space for variables, which are given in following table:

Directives	Purpose	Storage Space
DB	Define Byte	8 bits
DW	Define Word	16 bits
DD	Define Double Word	32 bits
DQ	Define Quad Word	64 bits
DT	Define Ten Bytes	80 bits
DO	IEEE-754 Quad	128 bits

```
; COAL Video Lecture: 30

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```

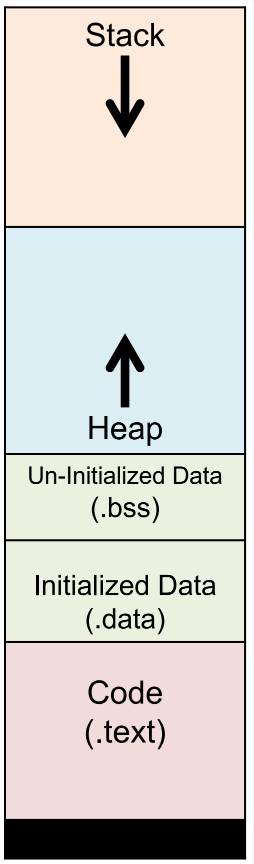


Section .data: Initialized Data (cont…)

Examples:

```
bVar   db   10          ;byte variable
cVar   db   "H"         ;single character
msg    db   "Hello!"    ;string variable
wVar   dw   5000        ;16-bit variable
dVar   dd   50000       ;32-bit variable
qVar   dq   5000000     ;64-bit variable
arr    dd   10,20,30    ;array of 32-bit var


flt1   dd   3.14159   ;IEEE-754 single precision
flt2   dt   3.14159   ;IEEE-754 double precision
flt3   do   3.14159   ;IEEE-754 quad precision
```

```
; COAL Video Lecture: 30

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



Section .bss: Un-Initialized Data

- All uninitialized data is declared in the `.bss` section (Block Storage Start)
- You use the RES directive to reserve uninitialized space in memory for your variables
- The general format for variable declaration is:

<varName> <resDirective> <count>

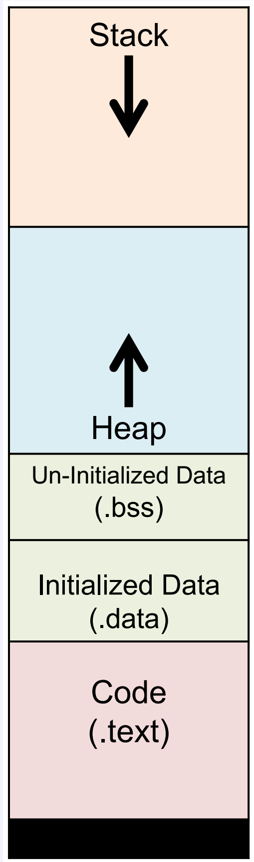
- NASM present various RES directives, which are given in following table:

Directives	Purpose
RESB	Reserve Byte
RESW	Reserve Word
RESD	Reserve Double Word
RESQ	Reserve Quad Word
REST	Reserve 10 Bytes
RESO	Reserve 16 Bytes

Examples:

```
bArr    resb    10    ;10 elements byte array
wArr    resw    50    ;50 elements word array
dArr    resd    25    ;25 elements double array
qArr    resq    15    ;15 elements quad array
```

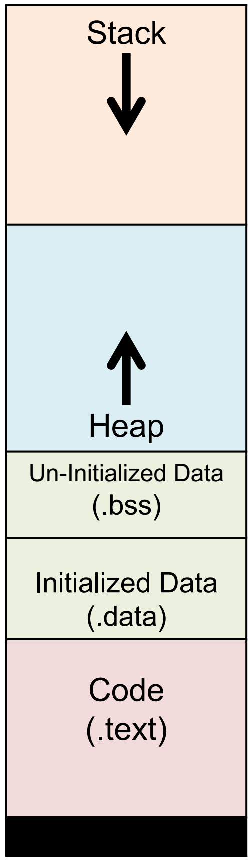
```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



Section .text

- The way a variable is a named memory location containing data, similarly a label is also a named memory location containing some instruction. Upon assembling, the assembler will replace all the occurrences of the label with the corresponding memory address. The text section will always include at least one label named **`_start`** or **`main`**, that define the initial program entry point. The Linux linker `ld(1)`, expect the program entry point label with the name of `_start`, while `gcc(1)` expect the program entry point label with the name of `main`
- The **`global`** directive is used to define a symbol, which is expected to be used by another module using the `extern` directive. The `extern` directive is 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

```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



mov Instruction

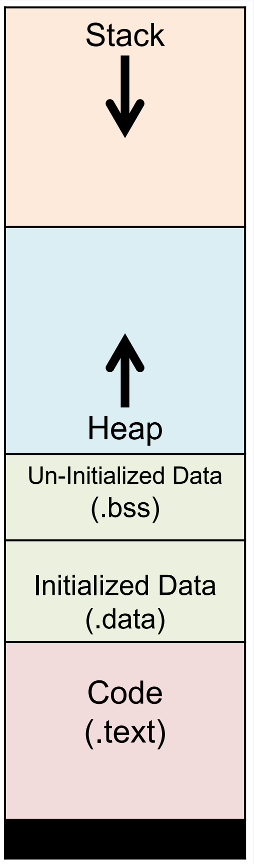
- The MOV instruction is used for moving data from one storage space to another
- The MOV instruction takes two operands and its general syntax is:

  `MOV destination, source`

- Both the operands of MOV instruction should be of same size and the value of source operand remains unchanged
- The MOV instruction may have one of the following five forms –

```
MOV register, immediate
MOV register, register
MOV register, memory
MOV memory, register
MOV memory, immediate
```

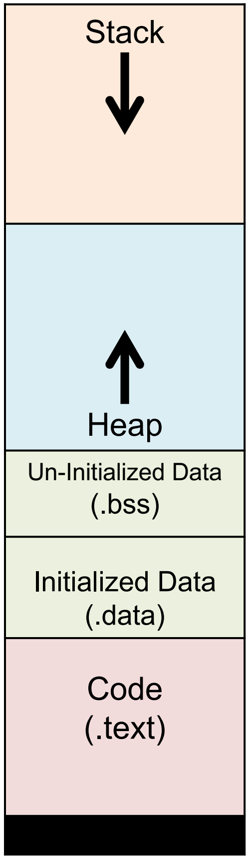
```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



syscall Instruction

- The two methods using which a program can request the operating system to perform a service like printing on screen or reading from keyboard and so on are:
  - By making a system call
  - By making a library call
- We will mostly be using the system call, which is a controlled entry point into the Operating System code, allowing a process to request the OS to perform a privileged operation

```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



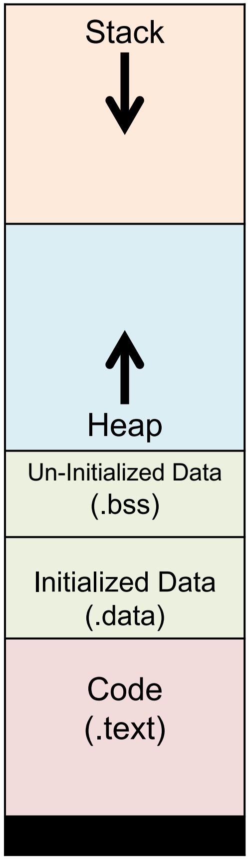
syscall Instruction (cont…)

List of available System Calls

- Every operating system has its own set of system calls and every system call has an associated ID
- On my Intel Core i7 CPU, running Kali Linux 5.3, there are a total of 433 system calls, whose IDs can be seen from the file /usr/include/x86_64-linux-gnu/asm/unistd_64.h

System Calls	ID
read()	0
write()	1
open()	2
close()	3
getpid()	39
shutdown()	48
fork()	47
exit()	60

```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



`syscall` Instruction (cont…)

How to make a System Call

1. First of all depending on your architecture, you need to place the system call ID in an appropriate register

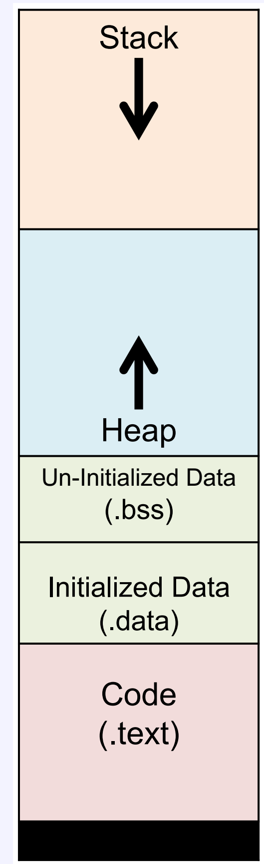
Architecture	Instruction	System call ID
X86_64	syscall	rax
80386	int 0x80	eax
ARM	svc 0	r7
ARM-64	svc 0	x8

2. Next step is to place the system call arguments (if any) in appropriate register(s)

Architecture	arg1	arg2	arg3	arg4	arg5	arg6
X86_64	rdi	rsi	rdx	r10	r8	r9
80386	ebx	ecx	edx	esi	edi	ebp
ARM	r0	r1	r2	r3	r4	r5
ARM-64	x0	x1	x2	x3	x4	x5

3. After the system call the return value can be found in **rax, eax, r7** and **r8** registers respectively for the above four architectures

```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



`syscall` Instruction (cont…)

Architecture	ID	arg1	arg2	arg3	arg4	arg5	arg6
X86_64	rax	rdi	rsi	rdx	r10	r8	r9

How to Display a Messsage on Screen

```
int write(int fd, void *buf, int count);
```

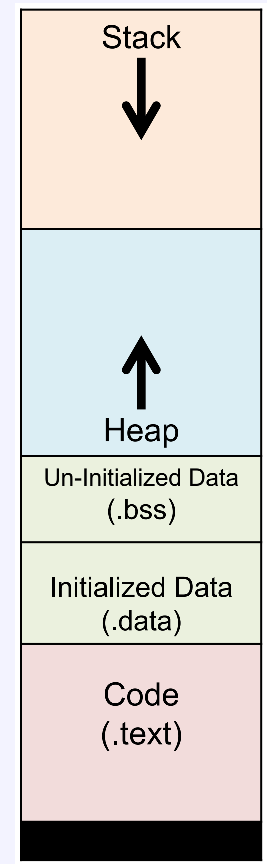
ID	rax	1
arg1 (file descriptor)	rdi	1
arg2 (address of string)	rsi	msg
arg3 (length of string)	rdx	26

How to Terminate the Program

```
void exit(int status);
```

ID	rax	60
arg1 (exit status)	rdi	54

```
; COAL Video Lecture: 29

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



Things To Do

Coming to office hours does NOT mean you are academically week!