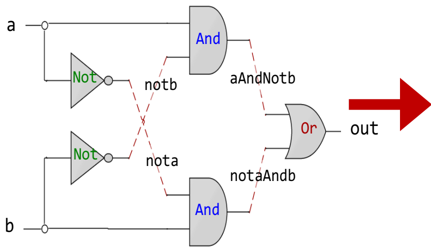
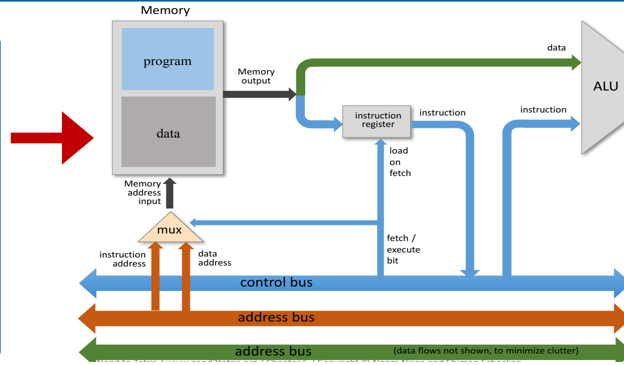
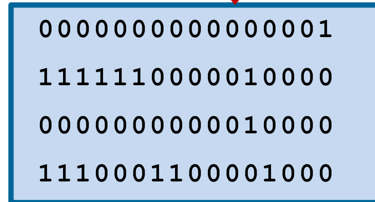
Computer Organization & Assembly Language Programming




```
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
```

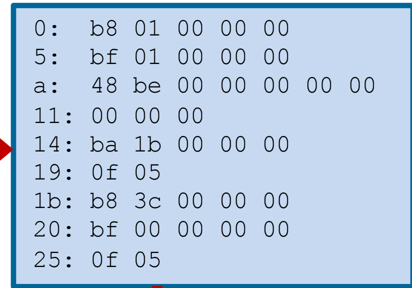


Lecture # 29

Hello World in x86-64 Assembly

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

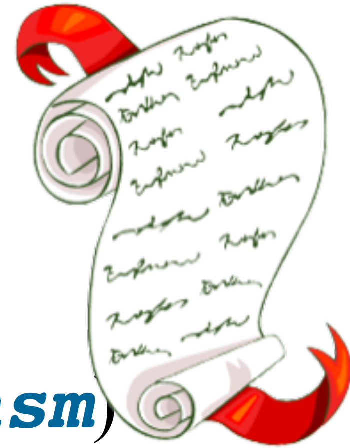
```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```



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



- Micro Processors & Assembly Language(s)
- Review of x86-64 Programming Model
- Tool Chain & Programming Environment
- A Hello World Assembly Program (`first.nasm`)
- Demo

Micro Processors
&
Assembly Languages

Micro Processors & Assembly Languages

- **Motorola 68000** (16/32 bits architecture, 1979)
- **Accorn ARM** (Advanced RISC Machine, 32/64 bits architecture, 1981)
- **MIPS** (Microprocessor without Interlocked Pipeline Stages, 32/64 bits architecture, 1981)
- **Intel IA-32** (Intel Architecture, 32 bits architecture, 1986)
- **Sun Sparc** (Scalable Processor ARChitecture, 32/64 bits architecture, 1987)
- **Motorola PowerPC** (Performance Optimization With Enhanced RISC Performance Computing, 32/64 bits architecture, 1992)
- **DEC Alpha** (Digital Equipment Corporation Alpha (now Compacc), 64 bits architecture, 1992)
- **AMD x86-64** (Advanced Micro Devices, 64 bits architecture, 2000)
- **Intel IA-64** (64 bits architecture, 2001)
- **………..**

Intel CPUs History

i386
i386
i386sl
i386dx
i386sx

i486
i486dx
i486sl
i486sx
Overdrive

Pentium - "P5"
P5
P54C
Pentium MMX (P55C)
Mobile Pentium MMX
Mobile Pentium
Overdrive

Larrabee
Knight's Corner

Pentium Pro - "P6"
Overdrive

Pentium II - "P6"
Klamath
Deschutes
Tonga
Dixon
Mobile Pentium II
Mendocino
Covington
Celeron
Overdrive

Atom
Silverthorne
Menlow
Atom 300
Atom 200
Poulsbo
US15W
Diamondville
Atom Z Series

Atom Gen 2
Lincroft
Sodaville
Z6xx
Oak Trail
Whitney Point
Briertown
Pineview
D4xx
N4xx
D5xx
N5xx
Tunnel Creek
Stellarton
E6xx
Queens Bay
Crossville
Top Cliff

Atom Gen 3
Cedarview

Legend

Architecture Names
Core Names
Marketing, Box, *Board,*
Concept, or Chipset
Names

Color Key
32-bit
64-bit

Pentium III - "P6M"
Coppermine
Katmai
Tualatin
Celeron
Xeon

Netburst - "P7" Pentium 4
Willamette
Prescott
Northwood
Cedar Mill
Pentium Extreme Edition
Prescott 2M (Irvindale)
Mobile Pentium 4 M
Mobile Pentium 4
Pentium D
Smithfield *Celeron D*
Gallatin *Celeron*
Presler *Xeon*

"Stealey"
Dothan
McCaslin
A110
A100

Pentium M
Dothan
Banias
Centrino Sonoma
Centrino Carmel
Celeron

Core (brand)
Yonah
Centrino Napa
Core Solo
Core Duo
Xeon LV

Core Microarchitecture (=Core2 brand)
Conroe Merom
Allendale Wolfdale
Kentsfield Yorkfield
Penryn
Centrino Santa Rosa
Centrino2 Montevina
Core2 Quad *Xeon*
Celeron *Woodcrest*
Core2 Solo *Clovertown*
Core2 Duo *Tigerton*

Nehalem
Beckton Westmere
Lynnfield Gulftown
Gainestown Clarkdale
Bloomfield Arrandale
Clarksfield
Core i7 Extreme
Centrino2 Calpella
Core i3
Core i5
Core i7
Core i9
Xeon

Sandy Bridge
MB EP
EN B2
DT EX
Ivy Bridge
Gesher
Core i3
Core i5
Core i7
Core i9
Xeon

Haswell
Broadwell

(c) 2011 MontaVista Software, LLC - Klaas van Gend

Motivation & Warning

What is Assembly **GOOD** for?

- Teaches us how the hardware works
- Optimize programs for speed and size
- Writing device drivers, operating system kernel, compiler and embedded systems
- Reverse engineering

What is Assembly **BAD** for?

- Portability is lost
- Only a few programmers can read assembly
- Debugging is difficult

Unstructured Programming

Assembly is unstructured programming language, meaning that it provides only extremely basic programming control structures such as:

- Basic Expressions
- Read/Write over memory
- Jump operators
- Tests

Note that there are no:

- Procedure calls (argument possessing is done manually)
- Loops facility (need to use jump instead)
- No variable/function scope (every thing is global)

Yet, jumps, tests and basic read/write operations are enough to implement any program

Review of x86-64

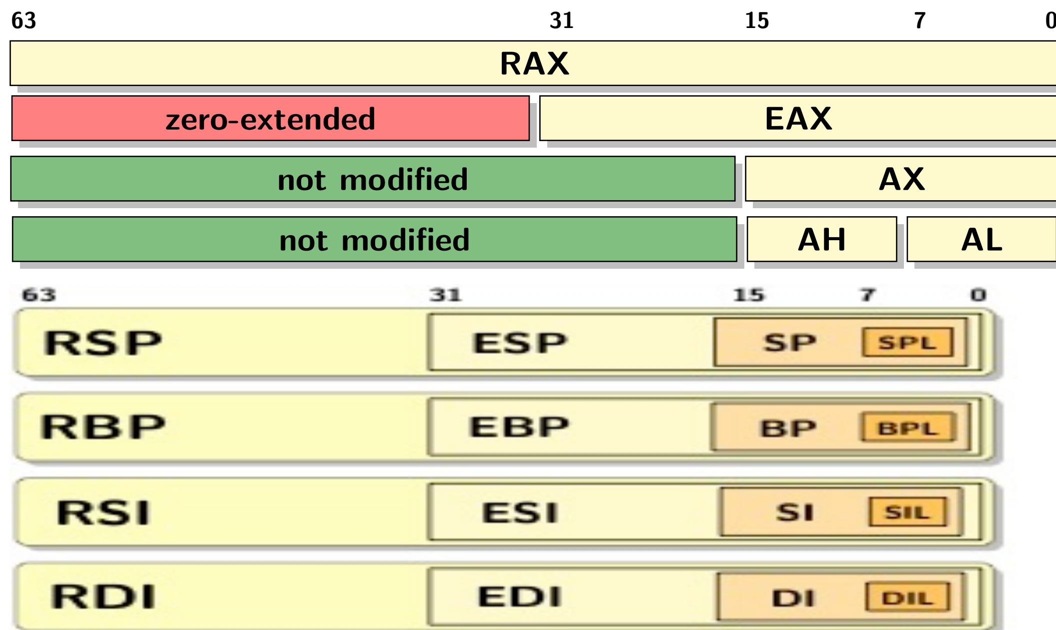
Register Set & Programming Model

Recap: x86-64 General Purpose Registers

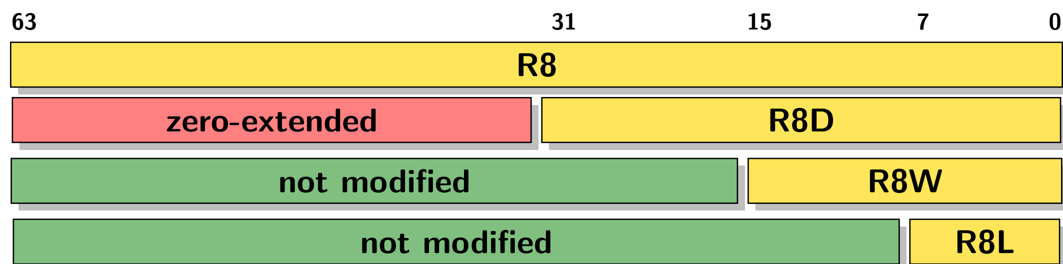
General Purpose Registers

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
r0/rax	eax	ax	al
r1/rbx	ebx	bx	bl
r2/rcx	ecx	cx	cl
r3/rdx	edx	dx	dl
r4/rsi	esi	si	sil
r5/rdi	edi	di	dil
r6/rbp	ebp	bp	bpl
r7/rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r8d	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

Already Existing registers (RAX-RDX)



New Registers (R8-R15)



Recap: x86-64 SSE Registers

SSE Media Registers

511	255	127 0
zmm0	ymm0	xmm0
zmm1	ymm1	xmm1
zmm2	ymm2	xmm2
zmm3	ymm3	xmm3
zmm14	ymm14	xmm14
zmm15	ymm15	xmm15

Streaming SIMD Extensions (SSE Registers)

- SIMD instruction set was introduced in the Pentium III (1999), that allowed a single instruction to be applied simultaneously to multiple packed data items using 128 bit registers (xmm0, xmm1, xmm2, …)
- The registers are used to store/pack
  - four 32-bit single-precision floating point numbers, or
  - two 64-bit double-precision floating point numbers, or
  - two 64-bit integers, or
  - four 32-bit integers, or
  - eight 16-bit short integers, or
  - sixteen 8-bit bytes or characters
- Latest processors also support Streaming SIMD Extensions (SSE), that extended the size of the registers to 256 and even 512 bytes namely ymm and zmm respectively

Recap: x86-64 Flags Register

	63		21	20	19	18	17	16		14	13	12	11	10	9	8	7	6		4		2		0
RFLAGS	-		ID	VIP	VIF	AC	VM	RF	-	NT	IOP1	IOP0	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

The `rflags` register is used for status and CPU control information. Out of the 64 bits mostly are unused and reserved for future use. These flags are divided into three categoriges:

Status Flags

- Carry flag (CF) holds the carry out after addition or the borrow in after subtraction out/in of msb (Identify an unsigned overflow)
- Parity flag (PF) is the count of one bits in a number, expressed as odd or even, represented by 0 or 1 respectively
- Auxiliary flag (AF) holds the carry out after addition or the borrow in after subtraction between bit position 3 and 4 of the result (BCD)
- Zero flag (ZF) is set if the previous operation resulted in a zero result
- Sign flag (SF) holds the msb of the result (sign bit) after an arithmetic or logic operation
- Overflow flag (OF) is set if the previous signed arithmetic operation resulted in an overflow

Control Flags

- Direction flag (DF) is used for moving or comparing strings. When DF is zero , the string operations takes left to right direction, and when DF is one, the string operations takes right to left direction

System Flags

- Interrupt enable flag (IF) is used to enable external interrupts like keyboard
- Trap flag (TF) is used to enable single step mode for debugging
- Resume flag (RF) is used to control the CPU response to debug exceptions

Review of x86-64 Programming Model

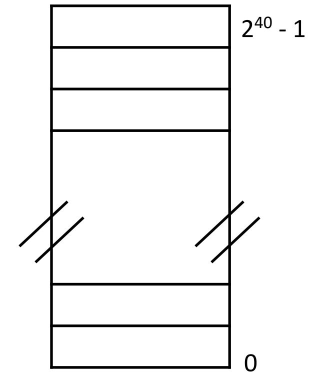
General Purpose Registers

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
r0/rax	eax	ax	al
r1/rbx	ebx	bx	bl
r2/rcx	ecx	cx	cl
r3/rdx	edx	dx	dl
r4/rsi	esi	si	sil
r5/rdi	edi	di	dil
r6/rbp	ebp	bp	bpl
r7/rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r8d	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

SSE Media Registers

511	255	127 0
zmm0	ymm0	xmm0
zmm1	ymm1	xmm1
zmm2	ymm2	xmm2
zmm3	ymm3	xmm3
zmm14	ymm14	xmm14
zmm15	ymm15	xmm15

Memory



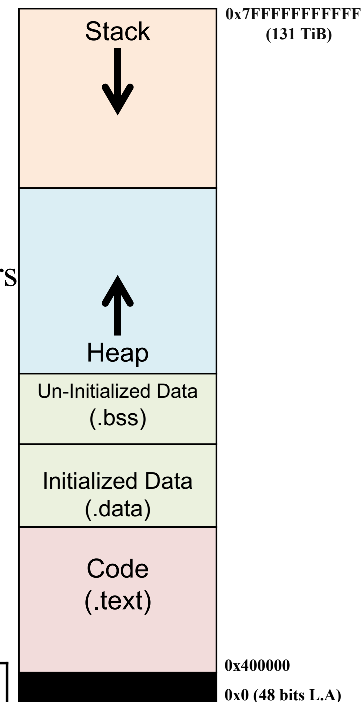
Segment Registers

15 0
CS
DS
SS
ES
FS
GS

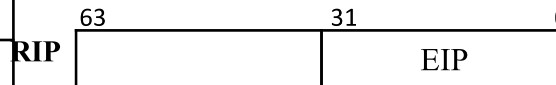
Misc Registers

Control Registers
Memory Management Registers
Debug Registers
Machine Specific Registers

Process Image



RIP



RFLAGS

63		21	20	19	18	17	16		14	13	12	11	10	9	8	7	6		4		2		0
-		ID	VIP	VIF	AC	VM	RF	-	NT	IOP1	IOP0	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

Process Address Space

Highest Address

Address-space	
argument2 argument1 ------------- var1 var2	Stack
var3 var4 var5	Heap
var6 var7 var8	Data
main() instr1 instr2 foo() ...	Code

Lowest Address

Registers: SP, PC, GPR0, GPR1, GPR2

Address-space

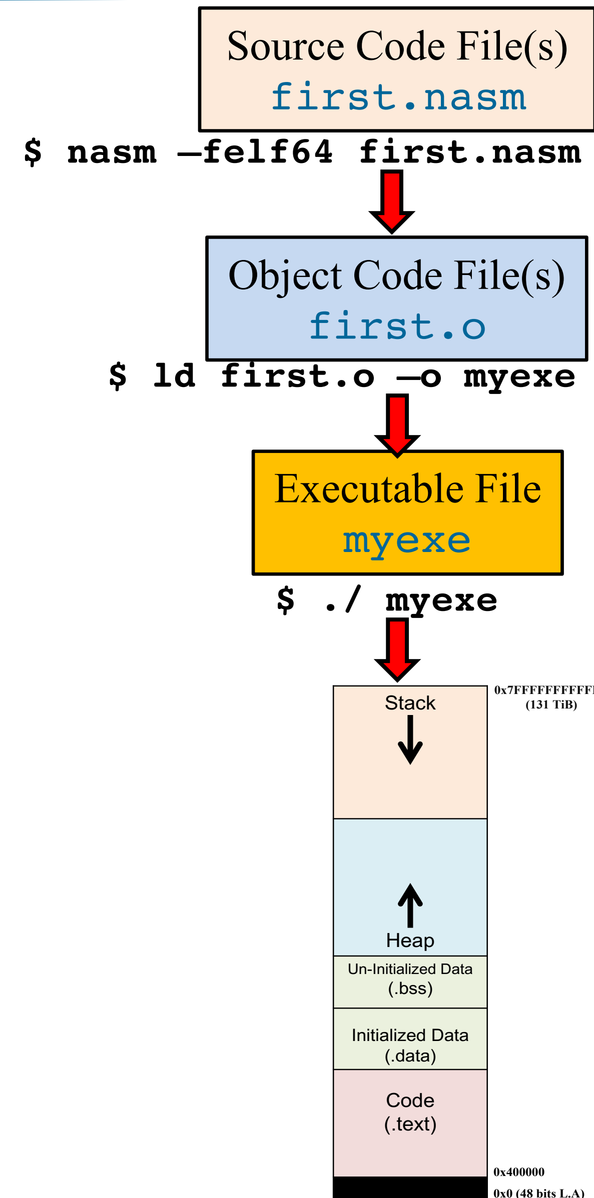
Segment	Address
Stack ↓	0x7FFFFFFFFFFF (131 TiB)
↑ Heap	
Un-Initialized Data (.bss)	
Initialized Data (.data)	
Code (.text)	0x400000
	0x0 (48 bits L.A)

Tool Chain
&
Programming Environment

Tool Chain & Programming Environment

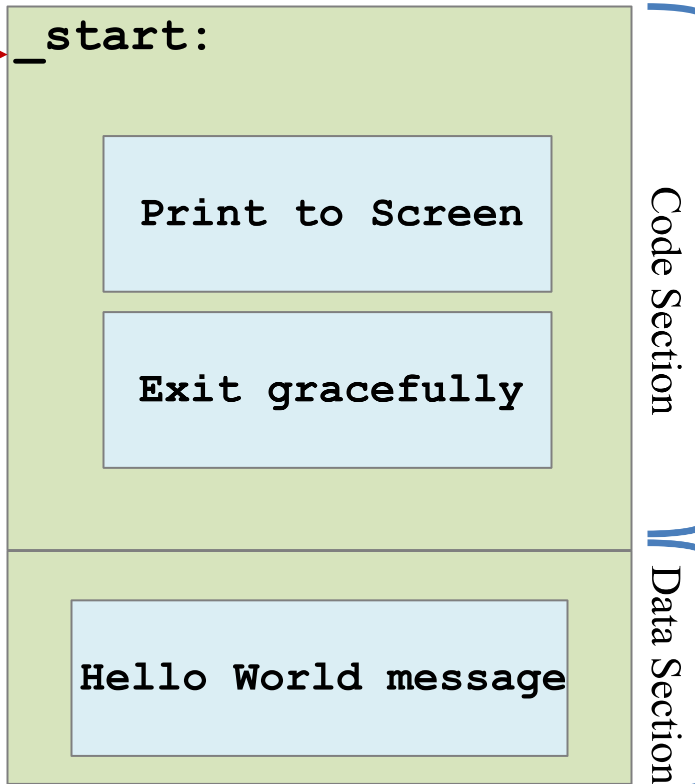
The set of programming tools used to create a program is referred to as the Tool chain. In this part of the course we intend learning x86-64 assembly, and the environment/tool chain involved are:

- **Processor:** Core 2duo/i3/i5/i7 (64 bit processor)
- **Operating System:** 64 bit Linux Distro (Ubuntu, Kali)
- **Editor:** gedit, vim, atom, sublime, Visual Studio, Eclipse, Xcode
- **Assembler:** NASM, YASM, GAS, MASM
- **Linker:** LD a GNU linker
- **Loader:** Default OS
- **Debugging/RE:** gdb, radare2, objdump and readelf



Hello World in Assembly

A Hello World Assembly Program




```
; COAL Video Lecture: 29
;  Programmer: Arif Butt
;  first.nasm
;  Assemble: nasm -felf64 first.nasm
;  Link:     ld first.o  -o first
;  usage:     ./first ; echo $?
SECTION .data
   msg db "Learning is fun with Arif",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
```

Assembling & Executing x86_64 Program

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
```

Assembling & Executing x86_64 Program

Things To Do

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