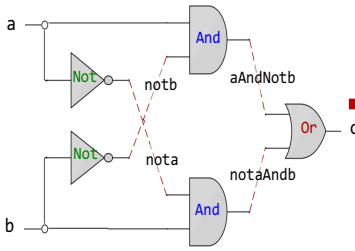
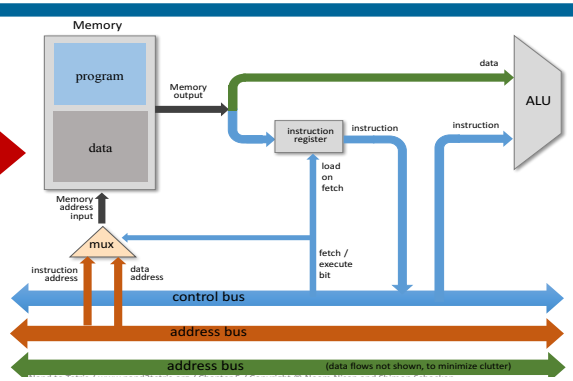
Computer Organization & Assembly Language Programming




```
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 # 42

Functions in Assembly Language - I

```
#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

- Recap: x86-64 Registers, Tool Chain & Instructions
- Defining an Assembly Function
- Calling/Returning an Assembly Function
- Use of Stack in Function Calls
  - Demo (***proc1.nasm***)
  - Demo (***proc2.nasm***)
- Caller Saved vs Callee Saved Registers
  - Demo (***proc3.nasm***)

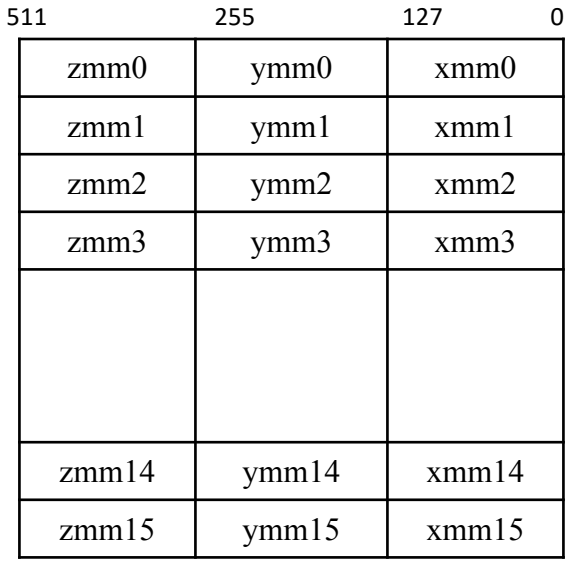
Recap

Review: x86-64 Register Set

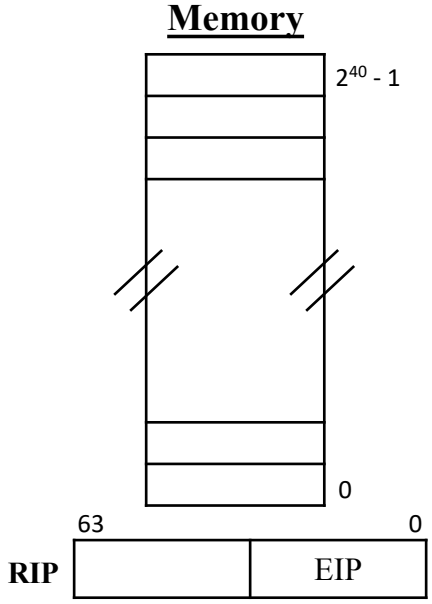
General Purpose Registers

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	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

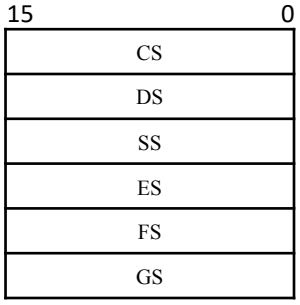
SSE Media Registers



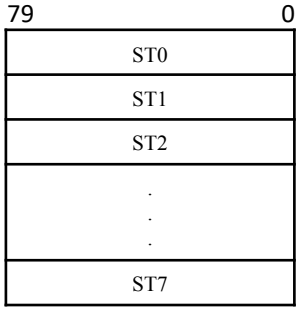
Memory

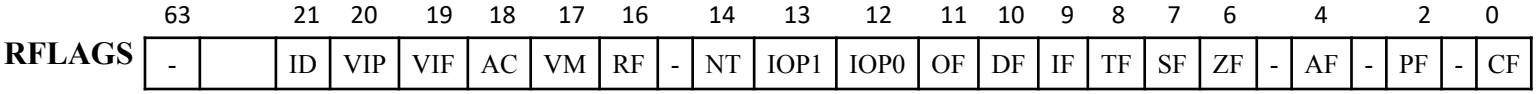


Segment Registers



FP Registers





Review: x86-64 Tool Chain

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
```

Assemble →

first.o

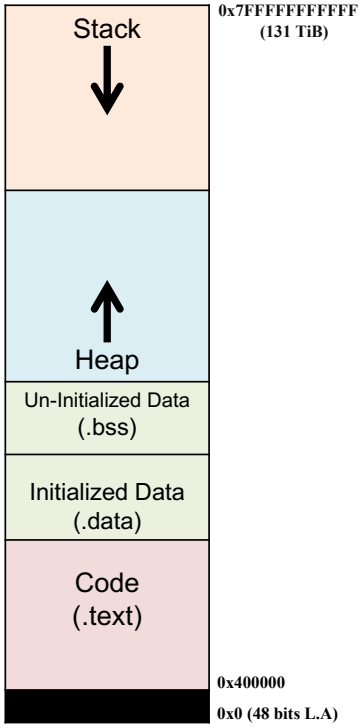
```
10001000
01000001
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
01001001
1000101001001000
0101011000011000
0001010010010001
10001010
01001011
```

Link →

myexe

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

Load & Execute →



- **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

Review: Categories of x86-64 Instructions

Category	Description	Examples
Data Transfer	Move from source to destination	`mov, movzx, movsx, lea, lds, lss, xchg, push, pop, pusha, popa, pushf, popf`
Arithmetic	Arithmetic on integer	`add, addc, sub, subb, mul, imul, div, idiv, neg, inc, dec, cmp`
Bit Manipulation	Logical & bit shifting operations	`and, or, not, xor, test, shl/sal, shr, sar, ror, rol, rcr, rcl`

Control Transfer	Conditional and unconditional jumps, and procedure calls	`jmp`
		`jcc(jz,jnz,jg,jge,jl,jle,jc,jnc,...)`
		`call, ret`

String	Move, compare, input and output	`movsb, movsw, lodsb, lodsw, stosb, stosw, rep, repz, repe, repnz, repne`
Floating Point	Arithmetic	`fld, fst, fstp, fadd, fsub, fmul, fdiv`
Conversion	Data type conversions	`cbw, cwd, cdq, xlat`
Input Output	For input and output	`in, out`
Miscellaneous	Manipulate individual flags	`clc, stc, cld, std, sti`

Procedures/Functions in Assembly Language

Defining an Assembly Procedure/Function

- In computer programming languages, a procedure, function, sub-routine, or method is a named piece of code (set of instructions) that can be called from a program in order to perform some specific task, thus making a program more structural, easier to understand and manageable
- One of the main difference between procedure and function is:
  - The return statement of a function returns the control as well as the function's result value to the calling program
  - The return statement of a procedure returns only control to the calling program

Defining a Procedure in NASM

```
          <procname>:
0xf70       <1st instr>
0xf71       <2nd instr>
0xf72       <3rd instr>
              …


0xf8a         ret
```

Defining a Procedure in MASM

```
          <procname> proc
0xf70       <1st instr>
0xf71       <2nd instr>
0xf72       <3rd instr>
              …


0xf8a         ret
          <procname> endp
```

Calling/Returning from an Assembly Function

- An assembly procedure is defined as a set of logically related instructions having a name that:
  - is meant to be called from different places
  - can accept parameters (via registers, global memory locations, stack)
  - do some processing (e.g., add numbers, print string, get input, …)
  - may return some value to its caller (via register, global memory location)
- The agreement on how to pass parameters and return values, and how to share CPU registers between caller and callee is called **calling convention**
- To call a procedure named **foo**, use the x86 **call** instruction

```
call foo          →   push rip
<next instr>          jmp foo
```

- To shift back the control of execution from a function, use the x86 **ret** instruction

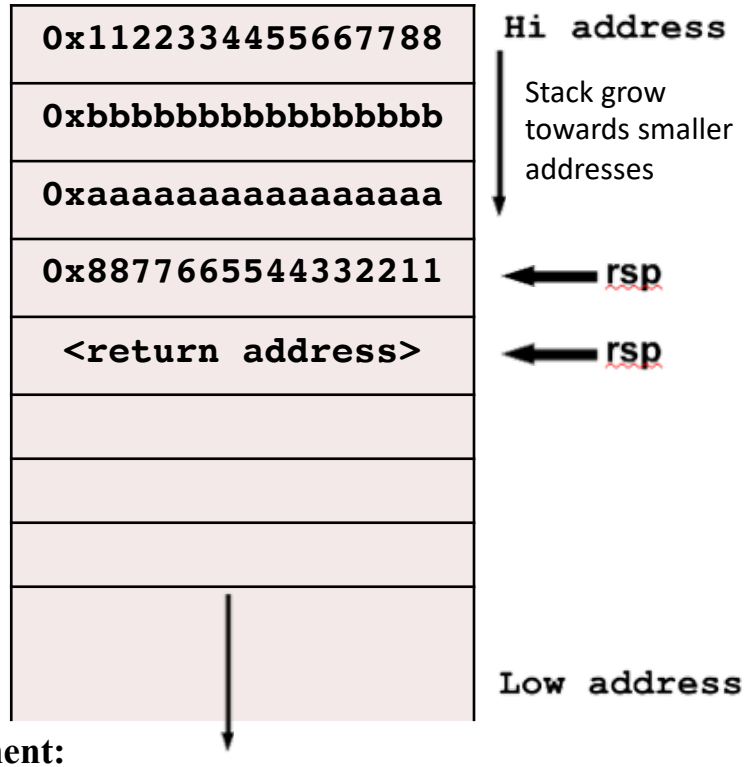
```
foo:
    … code …
    ret           →   pop rip
```

```
main:
0xf00   xor rax, rax
0xf01   mov rax, 0x2a3f
0xf02   call foo
0xf03   mov r12, 54
        …
        …
0xf69   ret
foo:
0xf70   <1st instr>
0xf71   <2nd instr>
0xf72    …
0xf7f   ret
```

Use of Stack in Function Calls

```
; COAL Video Lecture: 42
;  proc1.nasm
SECTION .data
    msg db "PUCIT", 0xA
    len_msg equ $ - msg
SECTION .text
    global main
    global printmsg
main:
rip  call printmsg          push rip
                            jmp printmsg
rip  mov rax,60
     mov rdi, 0
     syscall

printmsg:
rip  mov rax, 1
     mov rdi,1
     mov rsi,msg
     mov rdx,26
     syscall
rip  ret                    pop rip
```



Stack Alignment:

- The stack pointer `rsp` should be kept 16 byte aligned upon entry to a function, i.e., the hex value of `rsp` should end in a zero

Near vs Far Procedure

- If the caller and callee both lies in the same CS, the callee is a near procedure and the call instruction pushes only IP on the stack. However, for far procedure call instruction pushes both IP and CS on the stack

Assembling & Executing x86-64 Program

Example: proc2.nasm

```
; COAL Video Lecture: 42
;  proc2.nasm
SECTION .data
    msg db "PUCIT", 0xA
    len_msg equ $ - msg
SECTION .text
    global main
main:
    mov rcx, 0x10
    repeat:
        call printmsg
        loop repeat
; exit gracefully
    mov rax,60
    mov rdi, 0
    syscall
```

CALLER

```
; cont…
printmsg:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, len_msg
    syscall
    ret
```

CALLEE

Caller Saved vs Callee Saved Registers

Scratch / Callee Owned / Caller Saved

- In x86-64, the nine general purpose registers: `rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11` fall in this category
- They are callee (`printmsg`) owned, therefore the callee can freely use these registers
- They are caller (`main`) saved, therefore, (if the caller wants to preserve them) the caller must push them before making the function call and later pop them after the function returns
- Used for passing arguments to functions

Preserved / Caller Owned / Callee Saved

- In x86-64, the seven general purpose registers: `rbx, rbp, rsp, r12-r15` fall in this category
- They are caller (`main`) owned, therefore, the callee CANNOT freely use these registers
- They are callee (`printmsg`) saved, therefore, (if the callee wants to use them) the callee itself must push them at the start of function and pop them at the end of function
- Used for local state of the caller that needs to preserved across further function calls

Example: proc3.nasm

```
; COAL Video Lecture: 42
;  proc2.nasm
SECTION .data
   msg db "PUCIT", 0xA
   len_msg equ $ - msg
SECTION .text
   global main
main:
   mov rcx, 0x10
   repeat:
      push rcx
      call printmsg
      pop rcx
      loop repeat
; exit gracefully
   mov rax,60
   mov rdi, 0
   syscall
```

CALLER

```
; cont…
printmsg:
   mov rax, 1
   mov rdi, 1
   mov rsi, msg
   mov rdx, len_msg
   syscall
   ret
```

CALLEE

Caller is pushing rcx before calling the function

Caller is popping rcx before after the function returns

Assembling & Executing x86-64 Program

Things To Do

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