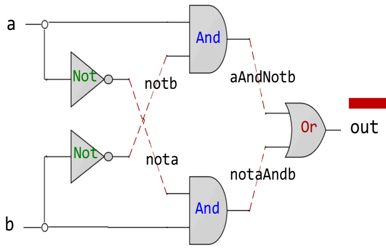
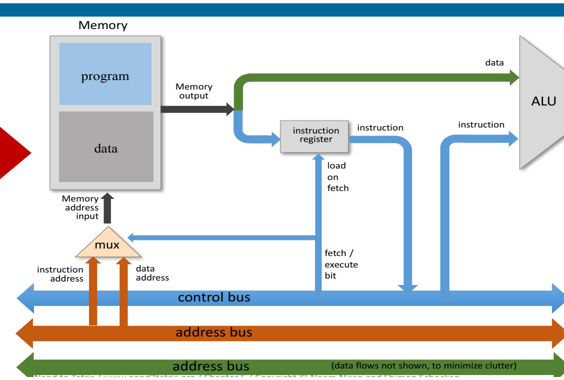
Computer Organization & Assembly Language Programming




```
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 # 39

Control Instructions - 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
- Control of Flow of Program Execution
- Unconditional Jump Instruction (**`JMP`**)
  - Demo (***`uncondjump1.nasm`***)
  - Demo (***`uncondjump2.nasm`***)
- Conditional Jump Instructions (**`Jcc`**)
  - Demo (***`condjump1.nasm`***)
  - Demo (***`condjump2.nasm`***)
  - Demo (***`condjump3.nasm`***)
  - Demo (***`condjump4.nasm`***)
  - Demo (***`condjump5.nasm`***)

https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

Recap

Review: x86-64 Register Set

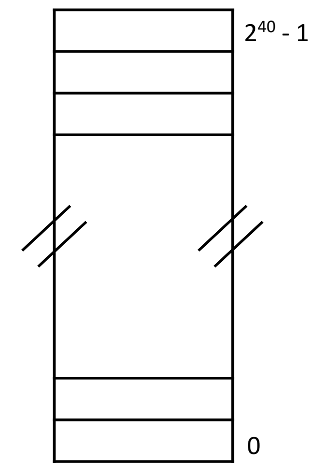
General Purpose Registers

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

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

Memory



RIP	63	0
		EIP

Segment Registers

15 … 0
CS
DS
SS
ES
FS
GS

FP Registers

79 … 0
ST0
ST1
ST2
. . .
ST7

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

Review: x86-64 Tool Chain

first.nasm → Assemble → **first.o** → Link → **myexe** → Load & Execute

first.nasm

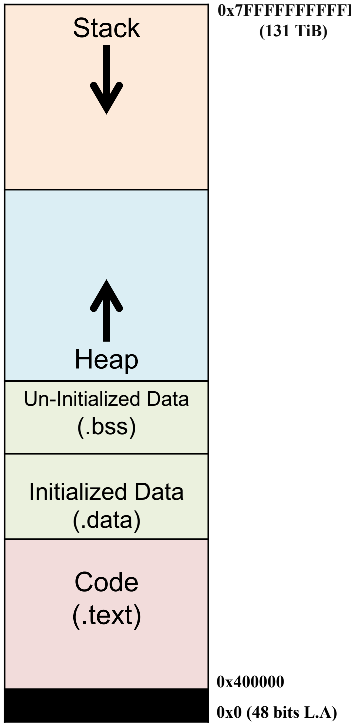
```
; COAL Video Lecture: 30
;  Programmer: Arif Butt
;  first.nasm
SECTION .data
   msg db "Learning…", 0xA
   EXIT_STATUS equ 54
SECTION .bss
;nothing here
SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
    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

first.o

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

myexe

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



- **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`

Control of Flow

Control of Flow of Program Execution

- The execution of every program starts from the label named **`_start`**, that define the initial program entry point. All code runs from top to bottom by default and the direction a program flows is called program flow
- The instruction pointer (RIP) register holds the address of the next instruction to be executed. After each instruction it is incremented by the instruction size (suppose 1 byte), thus making the control flow naturally flow from top to bottom
- X86-64 control instructions are used to alter the flow of execution of a program based on some event/calculation/comparison
- **Types:**
  - **Unconditional jump: `JMP <label>`**
  - **Conditional jump: `Jcc <label>`**
  - **Procedure Call: `CALL <label> & RET`**

```
; COAL Video Lecture: 39
;  Programmer: Arif Butt
;  example.nasm
SECTION .text
   global start
_start:
    mov rax, 15       ; rip = x
    mov rbx, 20       ; rip = x + 1
    add rax, rbx      ; rip = x + 2
    mov r9, 35        ; rip = x + 3
    mov r10, 23       ; rip = x + 4
    sub r9, r10       ; rip = x + 5
    mov rax,60        ; rip = x + 6
    mov rdi, 0        ; rip = x + 7
    syscall
```

Unconditional JUMP Instruction

- **Format:** `JMP destination`
- **Operation:** Transfers program control to a different location in the instruction stream (unconditionally). The destination operand specifies the address of the instruction being jumped to
- **Operand:** Destination operand is normally a label, i.e., a memory address pointing to some instruction. But can also be a register or immediate value
- **Types:**
  - **Short jump:** A jump where the jump range is limited to -128 to +127 from the current RIP value. (CS do not change)
  - **Near jump:** A jump within the current code segment. (CS do not change)
  - **Far jump:** A jump to an instruction located in a different segment than the current code segment
  - **Task switch:** A jump to an instruction located in different task

```
; COAL Video Lecture: 39
;  Programmer: Arif Butt
;  uncondjump1.nasm
SECTION .data
   msg1  db   "Study COAL", 0xA
   len_msg1  equ  $ - msg1
   msg2  db   "Play Cricket", 0xA
   len_msg2  equ  $ - msg2
SECTION .text
   global _start
_start:
   mov rax, 1
   mov rdi, 1
   mov rsi, msg1
   mov rdx, len_msg1
   syscall
   JMP _end
   mov rax, 1
   mov rdi, 1
   mov rsi, msg2
   mov rdx, len_msg2
   syscall
_end:
   mov rax, 60
   mov rdi, 0
   syscall
```

Assembling & Executing x86-64 Program

Conditional JUMP Instructions

`Jcc destination`

- If the condition code (cc) is **true**, the next instruction to be executed is the one at the destination (which is the address of the instruction being jumped to, can be a label, a register or immediate value)
- If the condition code (cc) is **false**, the instruction following the **`Jcc`** instruction gets executed
- To implement a conditional jump, the CPU looks at **one or more** status flags (CF, ZF, OF, PF, and SF), that are set by last instruction executed by the processor
- Conditional Jump instructions are further categorized into two types:
  - Unsigned Jumps (operate on ZF, and CF)
  - Signed Jumps (operate on ZF, SF, and OF)
- Most of assembly programmers use the **`cmp`** instruction before using the conditional jump

Recap: CMP Instruction

`CMP operand1, operand2`

- The **`cmp`** instruction subtracts the second operand from first operand, no operand is modified, however the flags (AF, CF, OF, PF, SF, ZF) are set according to the result
- First operand can be a register or memory, while the second operand can be an immediate value as well

cmp op1, op2	**ZF**	**CF**
op1 = op2	1	0
op1 > op2	0	0
op1 < op2	0	1

- Example:

```
MOV al, 5d
CMP al, 5d
;ZF=PF=1, CF=OF=SF=AF=0
```

```
MOV al, 6d
CMP al, 5d
;ZF=CF=OF=PF=SF=AF=0
```

```
MOV al, 5d
CMP al, 6d
;CF=PF=AF=SF=1, ZF=OF=0
```

Conditional JUMP Instructions (cont…)

```
cmp op1, op2
Jcc <label>
```

Unsigned Jumps	Description	Condition for jump
JE	Jump if equal	If op1==op2, ZF=1
JNE	Jump if not equal	If op1<>op2, ZF=0
JA	Jump if above than	If op1 > op2, ZF=0 and CF=0
JAE	Jump if above than or equal to	If op1 >= op2, CF=0
JB	Jump if below than	If op1 < op2, CF=1
JBE	Jump if below than or equal to	If op1 <= op2, ZF=1 and CF=1
JNE	Jump if not equal	If op1<>op2, ZF=0

Signed Jumps	Description	Condition for jump
JO	Jump if overflow	OF=1
JG	Jump if greater than	If op1 > op2, ZF=0 and SF=OF
JGE	Jump if greater than or equal to	If op1 >= op2, SF=OF
JL	Jump if less than	if op1< op2, SF<>OF
JLE	Jump if less than or equal to	If op1 <= op2, ZF=1 and SF<>OF

Example 1:

High Level Code:

```
if (AX < 0) then
   print("Negative Number!");
end if
```

```
 11111011
 00000000
 --------
 11111011
```

SF=1, OF=0, Since SF<>OF, therefore false

```
;   COAL Video Lecture: 39
;   condjump1.nasm
SECTION .data
    msg: db "Negative Number!",0xa
    len_msg: equ $ - msg
SECTION .text
global _start
_start:
    mov ax, -5d
    cmp ax, 0        ;SF=1, OF=0
    jge _end
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, len_msg
    syscall
_end:
    mov rax, 60
    mov rdi, 0
    syscall
```

Example 2:

High Level Code:

```
if (AX < 0) then
   print("Negative Number!");
else
   print("Positive Number!");
end if
```

```
;  COAL Video Lecture: 39
;  condjump2.nasm
SECTION .data
   msg1: db "Negative Number!",0xa
   len_msg1: equ $ - msg1
   msg2: db "Positive Number!",0xa
   len_msg2: equ $ - msg2
;  cont…
```

```
;  cont…
SECTION .text
global _start
_start:
.   mov ax, 5d
    cmp ax, 0
    jge _positive
    mov rax, 1
    mov rdi, 1
    mov rsi, msg1
    mov rdx, len_msg1
    syscall
    jmp _end
_positive:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg2
    mov rdx, len_msg2
    syscall
_end:
    mov rax, 60
    mov rdi, 0
    syscall
```

Assembling & Executing x86-64 Program

Example 3:

High Level Code:

```
if (AX == BX) then
   print("Equal");
else if (AX > BX) then
   print("AX > BX");
else
   print("BX > AX");
end if
```

```
SECTION .data
   msg1 db "AX == BX", 0xa
   len_msg1 equ $ - msg1
   msg2 db "AX > BX", 0xa
   len_msg2 equ $ - msg2
   msg3 db "BX > AX", 0xa
   len_msg3 equ $ - msg3
SECTION .text
   global _start
_start:
   mov ax, 5d
   mov bx, -25d
   cmp ax, bx
   je  equal
   cmp ax, bx
   jg  axbigger
;print (BX > AX")
   jmp _end
_axbigger:
;print (AX > BX")
   jmp _end
_equal:
;print (AX == BX")
_end:
   mov rax, 60
   mov rdi, 0
   syscall
```

Example 4:

High Level Code:

```
if (AL >= 'A' && AL <= 'Z') then
   print("Upper Case Alphabet");
end if
```

```
;   COAL Video Lecture: 39
;   condjump4.nasm
SECTION .data
   msg: db "Upper Case Alphabet",0xa
   len_msg: equ $ - msg
SECTION .text
global _start
_start:
.  mov al, 'P'
   cmp al, 'A'
   jb  end
   cmp al, 'Z'
   ja  end
   mov rax, 1
   mov rdi, 1
   mov rsi, msg
   mov rdx, len_msg
   syscall
_end:
   mov rax, 60
   mov rdi, 0
   syscall
```

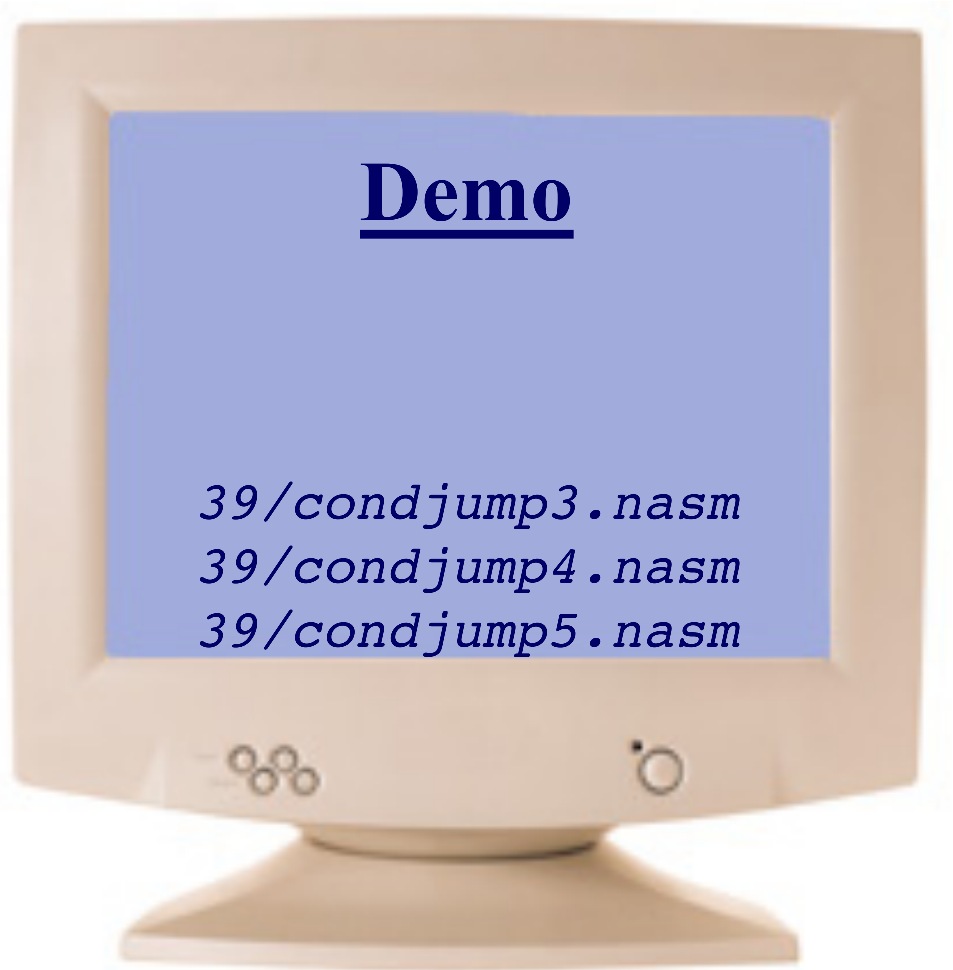
Example 5:

High Level Code:

```
if (AL == 'Y' OR AL == 'y') then
   print("YES");
end if
```

```
;  condjump5.nasm
SECTION .data
   msg: db "YES",0xa
   len_msg: equ $ - msg
SECTION .text
global _start
_start:
.  mov al, 'P'
   cmp al, 'Y'
   je _true
   cmp al, 'y'
   je _true
   jmp _end
_true:
   mov rax, 1
   mov rdi, 1
   mov rsi, msg
   mov rdx, len_msg
   syscall
_end:
   mov rax, 60
   mov rdi, 0
   syscall
```

Assembling & Executing x86-64 Program



Things To Do

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