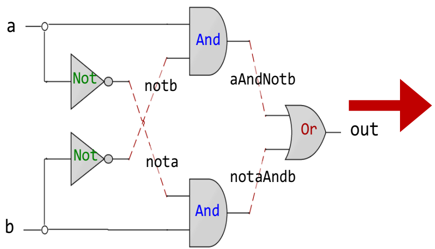
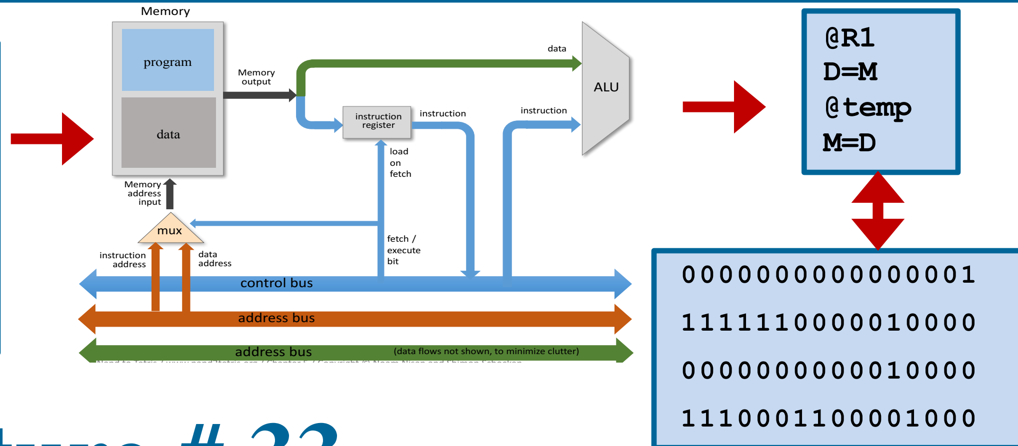
Computer Organization & Assembly Language Programming




```
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 # 33

Data Transfer Instructions & Process Stack

```
#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

- Move instruction (**`mov`**, `movzx`, `movsx`, `cmovcc`)
- Load Effective Address (**`lea`**, `lds`, `lss`, `les`, `lfs`, `lgs`)
- Exchange Instruction (**`xchg`**)
- Process Stack
- Stack Operations (**`push`**, **`pop`**)

mov Instruction

- The MOV instruction is used for moving data from one storage space to another
- It 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
- MOV instruction does not change the flags register
- If both operands are same registers, it acts a a NOP instruction
- The MOV instruction may have one of the following five forms −

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

Note: Transfer of data from one memory location to another is not allowed

mov Instruction (Immediate data to register)

MOV register, immediate

- **Examples:**

```
mov rax, 0xaaaaaaaabbbbbbbb

mov eax, 0xaaaaaaaa

mov ax, 0xdddd

mov al, 0x11

mov ah, 0xcc
```

`mov` Instruction (Register to Register)

`MOV register, register`

- **Examples:**

```
mov rbp, rax
mov r10, rbp
mov r11d, r10d
mov r12w, r11w
mov r13b, r12b
```

mov Instruction (Register to Memory)

`MOV memory, register`

- **Examples:**

```
mov byte [var], al
mov word [var], ax
mov dword [var], eax
mov qword [var], rax
```

mov Instruction (Memory to Register)

MOV register, memory

- **Examples:**

```
mov rsi, qword [var]

mov r14d, dword [var]

mov r15w, word [var]

mov dil, byte [var]
```

`lea` Instruction

- The address of a variable can be obtained with the load effective address, or `lea`, instruction. So `lea` instruction is used to load address of a variable into a register and later manipulate the data indirectly with the register as a pointer
- The `lea` instruction has no effect on the rflags register
- The format of load effective address instruction is as follows:

```
lea register, memory
```

- **Examples:**

```
lea rax, var
mov byte ptr [rax], 54
```

Note: `MOV` instruction moves the contents of the source into the destination, while the `LEA` instruction moves the address of the source into the destination

xchg Instruction

- The `xchg` instruction is used to exchange or swap the contents of two registers or the contents of a a register and a memory location:

```
xchg register, register
xchg register, memory
```

- **Example:**

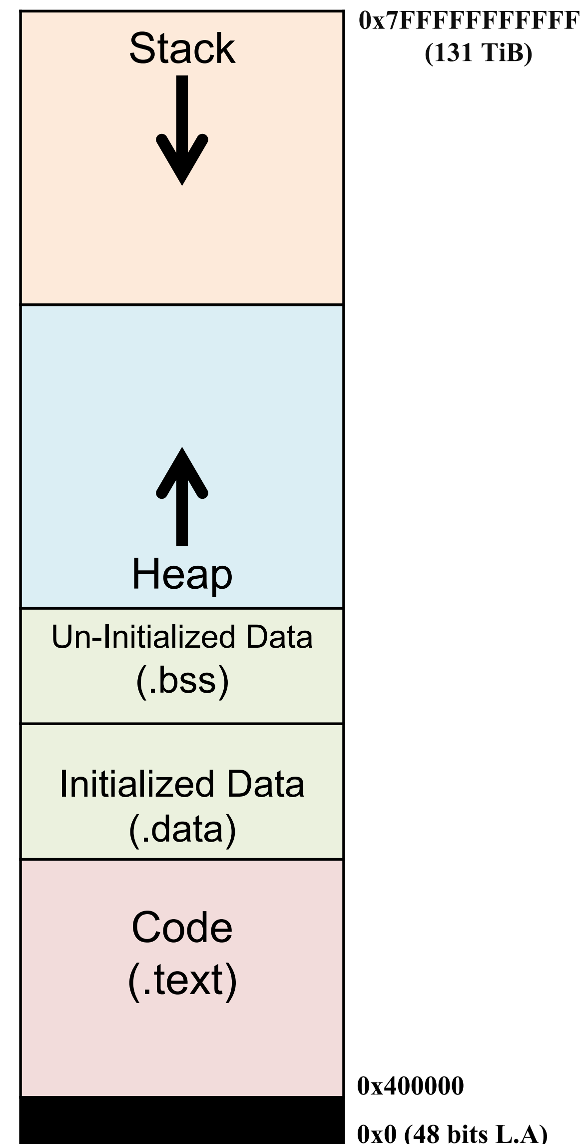
```
mov rax, 0x1234567890abcdef
mov rbx, 0x9999999999999999
xchg rax, rbx
```

Example Code: movingdata.nasm

Process Stack

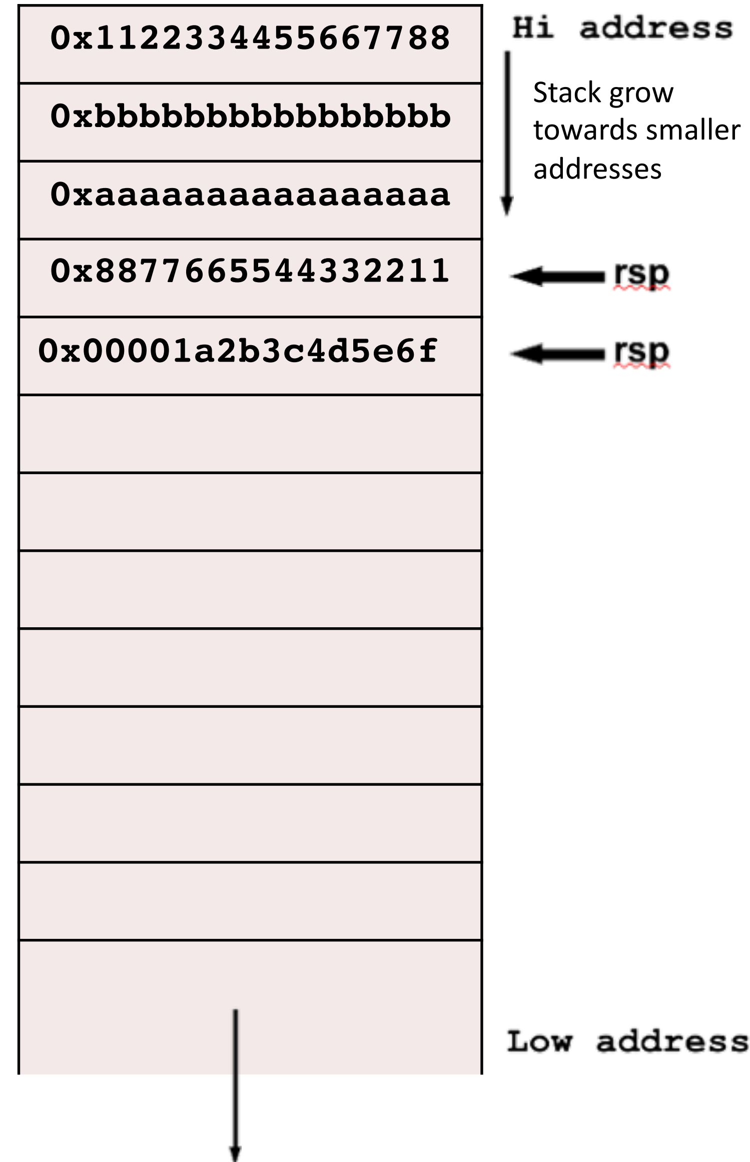
Logical Process Address Space

- The diagram shows the logical process address space
- At the lowest address, we have the code section that contains machine code instructions of our executable program
- Above the code section we have initialized and un-initialized data sections for global variables
- Heap is used for dynamic memory allocation and it grows towards higher addresses
- Finally, the process stack is at the top of the memory and grows from higher memory addresses towards lower memory addresses in architectures like Intel, MIPS, Motorola, SPARC
- High level languages like C/C++ make extensive use of the stack like temporary storing the arguments passed to a function, local variables and so on. (we will discuss this in detail in later part of the course)



Process Stack

- From Assembly programmer perspective, the use of process stack is quite simple and consist of either of the following two operations
  - A PUSH operation that stores data on the stack (`push reg/immediate`)
  - A POP operation that removes data from the stack (`pop reg`)



Example Code: `stack.nasm`

Things To Do

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